Isabelle/HOL — Higher-Order Logic

October 24, 2026

Contents

1 Loading the code generator and related modules **33**

2 The basis of Higher-Order Logic **34**
2.1 Primitive logic . . . 35
2.1.1 Core syntax . . . 36
2.1.2 Defined connectives and quantifiers . . . 36
2.1.3 Additional concrete syntax . . . 37
2.1.4 Axioms and basic definitions . . . 39
2.2 Fundamental rules . . . 40
2.2.1 Equality . . . 40
2.2.2 Congruence rules for application . . . 40
2.2.3 Equality of booleans – iff . . . 41
2.2.4 True (1) . . . 41
2.2.5 Universal quantifier (1) . . . 41
2.2.6 False . . . 42
2.2.7 Negation . . . 42
2.2.8 Implication . . . 42
2.2.9 Disjunction (1) . . . 43
2.2.10 Derivation of *iffI* . . . 43
2.2.11 True (2) . . . 44
2.2.12 Universal quantifier (2) . . . 44
2.2.13 Existential quantifier . . . 44
2.2.14 Conjunction . . . 44
2.2.15 Disjunction (2) . . . 45
2.2.16 Classical logic . . . 45
2.2.17 Unique existence . . . 46
2.2.18 Classical intro rules for disjunction and existential quantifiers . . . 46
2.2.19 Intuitionistic Reasoning . . . 47
2.2.20 Atomizing meta-level connectives . . . 48
2.2.21 Atomizing elimination rules . . . 50

2.3 Package setup . . . 50
2.3.1 Sledgehammer setup . . . 50
2.3.2 Classical Reasoner setup . . . 50
2.3.3 THE: definite description operator . . . 53
2.3.4 Simplifier . . . 53
2.3.5 Generic cases and induction . . . 64
2.3.6 Coherent logic . . . 68
2.3.7 Reorienting equalities . . . 68
2.4 Other simple lemmas and lemma duplicates . . . 69
2.5 Basic ML bindings . . . 70

3 *NO-MATCH* simproc 72
3.1 Code generator setup . . . 73
3.1.1 Generic code generator preprocessor setup . . . 73
3.1.2 Generic code generator foundation . . . 73
3.1.3 Equality . . . 74
3.1.4 Generic code generator target languages . . . 76
3.1.5 Evaluation and normalization by evaluation . . . 77
3.2 Counterexample Search Units . . . 77
3.2.1 Quickcheck . . . 77
3.2.2 Nitpick setup . . . 77
3.3 Preprocessing for the predicate compiler . . . 78
3.4 Legacy tactics and ML bindings . . . 78

4 Abstract orderings 78
4.1 Abstract ordering . . . 79
4.2 Syntactic orders . . . 82
4.3 Quasi orders . . . 82
4.4 Partial orders . . . 84
4.5 Linear (total) orders . . . 87
4.6 Reasoning tools setup . . . 88
4.7 Bounded quantifiers . . . 92
4.8 Transitivity reasoning . . . 95
4.9 min and max – fundamental . . . 99
4.10 (Unique) top and bottom elements . . . 100
4.11 Dense orders . . . 101
4.12 Wellorders . . . 103
4.13 Order on *bool* . . . 105
4.14 Order on - ⇒ - . . . 106
4.15 Order on unary and binary predicates . . . 108
4.16 Name duplicates . . . 109

5 Groups, also combined with orderings **109**
5.1 Dynamic facts . . . 110
5.2 Abstract structures . . . 110
5.3 Generic operations . . . 112
5.4 Semigroups and Monoids . . . 114
5.5 Groups . . . 118
5.6 (Partially) Ordered Groups . . . 122
5.7 Support for reasoning about signs . . . 124
5.8 Canonically ordered monoids . . . 137
5.9 Tools setup . . . 139

6 Abstract lattices **140**
6.1 Abstract semilattice . . . 140
6.2 Syntactic infimum and supremum operations . . . 143
6.3 Concrete lattices . . . 143
6.3.1 Intro and elim rules . . . 144
6.3.2 Equational laws . . . 145
6.3.3 Strict order . . . 148
6.4 Distributive lattices . . . 148
6.5 Bounded lattices . . . 149
6.6 *min*/*max* as special case of lattice . . . 151
6.7 Uniqueness of inf and sup . . . 152
6.8 Lattice on - ⇒ - . . . 153

7 Boolean Algebras **154**
7.1 Abstract boolean algebra . . . 154
7.1.1 Complement . . . 155
7.1.2 Conjunction . . . 156
7.1.3 Disjunction . . . 157
7.1.4 De Morgan's Laws . . . 157
7.2 Symmetric Difference . . . 158
7.3 Type classes . . . 160
7.4 Lattice on *bool* . . . 162
7.5 Lattice on unary and binary predicates . . . 163
7.6 Simproc setup . . . 164

8 Set theory for higher-order logic **166**
8.1 Sets as predicates . . . 166
8.2 Subsets and bounded quantifiers . . . 169
8.3 Basic operations . . . 176
8.3.1 Subsets . . . 176
8.3.2 Equality . . . 177
8.3.3 The empty set . . . 177
8.3.4 The universal set – UNIV . . . 178

8.3.5 The Powerset operator – Pow . . . 179
8.3.6 Set complement . . . 179
8.3.7 Binary intersection . . . 179
8.3.8 Binary union . . . 180
8.3.9 Set difference . . . 181
8.3.10 Augmenting a set – *insert* . . . 181
8.3.11 Singletons, using insert . . . 182
8.3.12 Image of a set under a function . . . 184
8.3.13 Some rules with *if* . . . 187
8.4 Further operations and lemmas . . . 187
8.4.1 The "proper subset" relation . . . 187
8.4.2 Derived rules involving subsets. . . . 188
8.4.3 Equalities involving union, intersection, inclusion, etc. 189
8.4.4 Monotonicity of various operations . . . 199
8.4.5 Inverse image of a function . . . 200
8.4.6 Singleton sets . . . 202
8.4.7 Getting the contents of a singleton set . . . 202
8.4.8 Monad operation . . . 203
8.4.9 Operations for execution . . . 203

9 HOL type definitions 207

10 Notions about functions 209
10.1 The Identity Function *id* . . . 209
10.2 The Composition Operator $f \circ g$ . . . 210
10.3 The Forward Composition Operator *fcomp* . . . 211
10.4 Mapping functions . . . 211
10.5 Injectivity and Bijectivity . . . 212
10.5.1 Inj/surj/bij of Algebraic Operations . . . 224
10.6 Function Updating . . . 226
10.7 *override-on* . . . 228
10.8 Inversion of injective functions . . . 228
10.9 Monotonicity . . . 229
10.9.1 Specializations For *ord* Type Class And More . . . 230
10.9.2 Least value operator . . . 238
10.10 Setup . . . 238
10.10.1 Proof tools . . . 238
10.10.2 Functorial structure of types . . . 239

11 Complete lattices 239
11.1 Syntactic infimum and supremum operations . . . 239
11.2 Abstract complete lattices . . . 241
11.3 Complete lattice on *bool* . . . 252
11.4 Complete lattice on - ⇒ - . . . 252

11.5 Complete lattice on unary and binary predicates 253
11.6 Complete lattice on - *set* 255
11.6.1 Inter 255
11.6.2 Intersections of families 256
11.6.3 Union 258
11.6.4 Unions of families 260
11.6.5 Distributive laws 263
11.7 Injections and bijections 264
11.7.1 Complement 266
11.7.2 Miniscoping and maxiscoping 266

12 Wrapping Existing Freely Generated Type's Constructors 267

13 Knaster-Tarski Fixpoint Theorem and inductive definitions 268
13.1 Least fixed points 269
13.2 General induction rules for least fixed points 270
13.3 Greatest fixed points 271
13.4 Coinduction rules for greatest fixed points 272
13.5 Even Stronger Coinduction Rule, by Martin Coen 274
13.6 Rules for fixed point calculus 275
13.7 Inductive predicates and sets 276
13.8 The Schroeder-Bernstein Theorem 277
13.9 Inductive datatypes and primitive recursion 279

14 Cartesian products 279
14.1 *bool* is a datatype 279
14.2 The *unit* type 280
14.3 The product type 283
14.3.1 Type definition 283
14.3.2 Tuple syntax 285
14.3.3 Code generator setup 287
14.3.4 Fundamental operations and properties 287
14.3.5 Derived operations 294
14.4 Code generator setup for paired and tripled bounded set comprehension 306
14.5 Simproc for rewriting a set comprehension into a pointfree expression 306
14.6 Lemmas about disjointness 306
14.7 Inductively defined sets 306
14.8 Legacy theorem bindings and duplicates 307

15 The Disjoint Sum of Two Types **307**
15.1 Construction of the sum type and its basic abstract operations 308
15.2 Projections . . . 310
15.3 The Disjoint Sum of Sets . . . 312

16 Rings **312**
16.1 Semirings and rings . . . 312
16.2 Abstract divisibility . . . 315
16.3 Towards integral domains . . . 321
16.4 (Partial) Division . . . 326
16.5 Quotient and remainder in integral domains . . . 349
16.6 Interlude: basic tool support for algebraic and arithmetic calculations . . . 350
16.7 Ordered semirings and rings . . . 351
16.8 Dioids . . . 371

17 Natural numbers **372**
17.1 Type *ind* . . . 372
17.2 Type nat . . . 372
17.3 Arithmetic operators . . . 376
17.3.1 Addition . . . 378
17.3.2 Difference . . . 378
17.3.3 Multiplication . . . 379
17.4 Orders on *nat* . . . 380
17.4.1 Operation definition . . . 380
17.4.2 Introduction properties . . . 382
17.4.3 Elimination properties . . . 382
17.4.4 Inductive (?) properties . . . 383
17.4.5 Monotonicity of Addition . . . 387
17.4.6 *min* and *max* . . . 389
17.4.7 Additional theorems about ($\leq$) . . . 391
17.4.8 More results about difference . . . 396
17.4.9 Monotonicity of multiplication . . . 398
17.5 Natural operation of natural numbers on functions . . . 401
17.6 Kleene iteration . . . 404
17.7 Embedding of the naturals into any *semiring-1*: *of-nat* . . . 406
17.8 The set of natural numbers . . . 410
17.9 Further arithmetic facts concerning the natural numbers . . . 411
17.9.1 Greatest operator . . . 420
17.10 Monotonicity of *funpow* . . . 420
17.11 Kleene's fixed point theorem for continuous functions . . . 421
17.12 The divides relation on *nat* . . . 422
17.13 Aliasses . . . 424
17.14 Size of a datatype value . . . 426

17.15Code module namespace 426

18 Fields **426**
18.1 Division rings 426
18.2 Fields 433
18.3 Ordered fields 437

19 Relations – as sets of pairs, and binary predicates **453**
19.1 Fundamental 454
19.1.1 Relations as sets of pairs 454
19.1.2 Conversions between set and predicate relations 455
19.2 Properties of relations 456
19.2.1 Reflexivity 456
19.2.2 Irreflexivity 459
19.2.3 Asymmetry 461
19.2.4 Symmetry 463
19.2.5 Antisymmetry 465
19.2.6 Transitivity 468
19.2.7 Totality 471
19.2.8 Left uniqueness 474
19.2.9 Right uniqueness 474
19.3 Relation operations 475
19.3.1 The identity relation 475
19.3.2 Diagonal: identity over a set 476
19.3.3 Composition 477
19.3.4 Converse 479
19.3.5 Domain, range and field 483
19.3.6 Image of a set under a relation 486
19.3.7 Inverse image 488
19.3.8 Powerset 488

20 Finite sets **489**
20.1 Predicate for finite sets 489
20.1.1 Choice principles 490
20.1.2 Finite sets are the images of initial segments of natural numbers 490
20.2 Finiteness and common set operations 492
20.3 Further induction rules on finite sets 501
20.4 Class *finite* 503
20.5 A basic fold functional for finite sets 504
20.5.1 From *fold-graph* to *fold* 506
20.5.2 Liftings to *comp-fun-commute-on* etc. 512
20.5.3 *UNIV* as carrier set 513
20.5.4 Expressing set operations via *fold* 515

20.5.5 Expressing relation operations via *fold* 519
20.6 Locales as mini-packages for fold operations 521
20.6.1 The natural case 521
20.6.2 With idempotency 522
20.6.3 *UNIV* as the carrier set 522
20.7 Finite cardinality 523
20.7.1 Cardinality of image 537
20.7.2 Pigeonhole Principles 539
20.7.3 Cardinality of sums 540
20.8 Minimal and maximal elements of finite sets 541
20.8.1 Finite orders 547
20.8.2 Relating injectivity and surjectivity 548
20.9 Infinite Sets 549
20.10 The finite powerset operator 552

21 Reflexive and Transitive closure of a relation 552
21.1 Reflexive closure 554
21.2 Reflexive-transitive closure 556
21.3 Transitive closure 561
21.4 Symmetric closure 571
21.5 The power operation on relations 571
21.6 Bounded transitive closure 582
21.7 Acyclic relations 583
21.8 Setup of transitivity reasoner 584

22 Well-founded Recursion 586
22.1 Basic Definitions 586
22.2 Equivalence of Definitions 586
22.3 Induction Principles 587
22.4 Introduction Rules 588
22.5 Ordering Properties 589
22.6 Basic Results 590
22.6.1 Minimal-element characterization of well-foundedness 591
22.6.2 Finite characterization of well-foundedness 592
22.6.3 Antimonotonicity 593
22.6.4 Equivalence between *wfp-on* and *wfp* 594
22.6.5 Well-foundedness of transitive closure 594
22.6.6 Well-foundedness of image 598
22.7 Well-Foundedness Results for Unions 600
22.8 Well-Foundedness of Composition 602
22.9 Acyclic relations 604
22.9.1 Wellfoundedness of finite acyclic relations 604
22.10 *nat* is well-founded 605
22.11 Accessible Part 606

22.12 Tools for building wellfounded relations . . . 608
22.12.1 Conversion to a known well-founded relation . . . 609
22.12.2 Measure functions into *nat* . . . 610
22.12.3 Lexicographic combinations . . . 610
22.12.4 Bounded increase must terminate . . . 615
22.13 Code Generation Setup . . . 616

23 Well-Founded Recursion Combinator 616
23.0.1 Well-founded recursion via genuine fixpoints . . . 618
23.1 Wellfoundedness of *same-fst* . . . 618

24 Orders as Relations 619
24.1 Orders on a set . . . 619
24.2 Orders on the field . . . 621
24.3 Relations given by a predicate and the field . . . 622
24.4 Orders on a type . . . 624
24.5 Order-like relations . . . 624
24.5.1 Auxiliaries . . . 624
24.5.2 The upper and lower bounds operators . . . 625
24.6 Variations on Well-Founded Relations . . . 628
24.6.1 Characterizations of well-foundedness . . . 629
24.6.2 Characterizations of well-foundedness . . . 631

25 Hilbert's Epsilon-Operator and the Axiom of Choice 633
25.1 Hilbert's epsilon . . . 633
25.2 Hilbert's Epsilon-operator . . . 633
25.3 Axiom of Choice, Proved Using the Description Operator . . . 634
25.4 Getting an element of a nonempty set . . . 635
25.5 Function Inverse . . . 636
25.6 Other Consequences of Hilbert's Epsilon . . . 646
25.7 An aside: bounded accessible part . . . 648
25.8 More on injections, bijections, and inverses . . . 650
25.9 Specification package – Hilbertized version . . . 653
25.10 Complete Distributive Lattices – Properties depending on Hilbert Choice . . . 653

26 Zorn's Lemma and the Well-ordering Theorem 659
26.1 Zorn's Lemma for the Subset Relation . . . 659
26.1.1 Results that do not require an order . . . 659
26.1.2 Hausdorff's Maximum Principle . . . 665
26.1.3 Results for the proper subset relation . . . 666
26.1.4 Zorn's lemma . . . 666
26.2 Zorn's Lemma for Partial Orders . . . 667
26.3 Other variants of Zorn's Lemma . . . 669

26.4 The Well Ordering Theorem 671

27 Well-Order Relations as Needed by Bounded Natural Functors 677
27.1 Auxiliaries 678
27.2 Well-founded induction and recursion adapted to non-strict well-order relations 679
27.3 The notions of maximum, minimum, supremum, successor and order filter 679
27.3.1 Properties of max2 680
27.3.2 Existence and uniqueness for isMinim and well-definedness of minim 681
27.3.3 Properties of minim 682
27.3.4 Properties of successor 683
27.3.5 Properties of order filters 684
27.3.6 Other properties 686

28 Well-Order Embeddings as Needed by Bounded Natural Functors 688
28.1 Auxiliaries 689
28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions 689
28.3 Given any two well-orders, one can be embedded in the other 698
28.4 Uniqueness of embeddings 704
28.5 More properties of embeddings, strict embeddings and isomorphisms 706

29 Constructions on Wellorders as Needed by Bounded Natural Functors 712
29.1 Restriction to a set 712
29.2 Order filters versus restrictions and embeddings 714
29.3 The strict inclusion on proper ofilters is well-founded 718
29.4 Ordering the well-orders by existence of embeddings 719
29.5 $<o$ is well-founded 728
29.6 Copy via direct images 730
29.7 Bounded square 734

30 Cardinal-Order Relations as Needed by Bounded Natural Functors 746
30.1 Cardinal orders 747
30.2 Cardinal of a set 748
30.3 Cardinals versus set operations on arbitrary sets 753
30.4 Cardinals versus set operations involving infinite sets 761
30.5 The cardinal ω and the finite cardinals 768

30.5.1 First as well-orders . . . 768
30.5.2 Then as cardinals . . . 770
30.6 The successor of a cardinal . . . 771
30.7 Regular cardinals . . . 777
30.8 Others . . . 780
30.9 Regular vs. stable cardinals . . . 781

31 Cardinal Arithmetic as Needed by Bounded Natural Functors 786
31.1 Zero . . . 787
31.2 (In)finite cardinals . . . 787
31.3 Binary sum . . . 789
31.4 One . . . 791
31.5 Two . . . 791
31.6 Family sum . . . 791
31.7 Product . . . 792
31.8 Exponentiation . . . 795

32 Function Definition Base 803

33 Definition of Bounded Natural Functors 803

34 Composition of Bounded Natural Functors 809

35 Registration of Basic Types as Bounded Natural Functors 814

36 Shared Fixpoint Operations on Bounded Natural Functors 820

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors 826

38 Equivalence Relations in Higher-Order Set Theory 831
38.1 Equivalence relations – set version . . . 831
38.2 Equivalence classes . . . 832
38.3 Quotients . . . 833
38.4 Refinement of one equivalence relation WRT another . . . 835
38.5 Defining unary operations upon equivalence classes . . . 835
38.6 Defining binary operations upon equivalence classes . . . 837
38.7 Quotients and finiteness . . . 838
38.8 Kernel of a Function . . . 839
38.9 Projection . . . 840
38.10 Equivalence relations – predicate version . . . 841
38.11 Equivalence closure . . . 842

39 MESON Proof Method **847**
39.1 Negation Normal Form . . . 847
39.2 Pulling out the existential quantifiers . . . 847
39.3 Lemmas for Forward Proof . . . 848
39.4 Clausification helper . . . 849
39.5 Skolemization helpers . . . 850
39.6 Meson package . . . 850

40 Automatic Theorem Provers (ATPs) **851**
40.1 ATP problems and proofs . . . 851
40.2 Higher-order reasoning helpers . . . 851
40.3 Basic connection between ATPs and HOL . . . 853

41 Metis Proof Method **854**
41.1 Literal selection and lambda-lifting helpers . . . 854
41.2 Metis package . . . 854

42 Generic theorem transfer using relations **855**
42.1 Relator for function space . . . 855
42.2 Transfer method . . . 855
42.3 Predicates on relations, i.e. “class constraints” . . . 857
42.4 Properties of relators . . . 861
42.5 Transfer rules . . . 862
42.6 *of-bool* and *of-nat* . . . 868

43 Lifting package **868**
43.1 Function map . . . 869
43.2 Quotient Predicate . . . 869
43.3 Quotient composition . . . 872
43.4 Respects predicate . . . 873
43.5 Domains . . . 878
43.6 ML setup . . . 879

44 Definition of Quotient Types **880**
44.1 Quotient Predicate . . . 881
44.2 lemmas for regularisation of ball and bex . . . 884
44.3 Bounded abstraction . . . 886
44.4 *Bex1-rel* quantifier . . . 888
44.5 Various respects and preserve lemmas . . . 889
44.6 Quotient composition . . . 891
44.7 Quotient3 to Quotient . . . 893
44.8 ML setup . . . 893
44.9 Methods / Interface . . . 895

45 Lifting of BNFs **896**

46 Binary Numerals **898**
46.1 The *num* type . . . 899
46.2 Numeral operations . . . 900
46.3 Binary numerals . . . 903
46.4 Class-specific numeral rules . . . 905
46.4.1 Structures with addition: class *numeral* . . . 905
46.4.2 Structures with negation: class *neg-numeral* . . . 906
46.4.3 Structures with multiplication: class *semiring-numeral* 909
46.4.4 Structures with a zero: class *semiring-1* . . . 910
46.4.5 Equality: class *semiring-char-0* . . . 910
46.4.6 Comparisons: class *linordered-nonzero-semiring* . . . 911
46.4.7 Multiplication and negation: class *ring-1* . . . 914
46.4.8 Equality using *iszero* for rings with non-zero characteristic . . . 914
46.4.9 Equality and negation: class *ring-char-0* . . . 915
46.4.10 Structures with negation and order: class *linordered-idom* 917
46.4.11 Natural numbers . . . 919
46.5 Particular lemmas concerning *2*::*'a* . . . 923
46.6 Numeral equations as default simplification rules . . . 924
46.6.1 Special Simplification for Constants . . . 924
46.6.2 Optional Simplification Rules Involving Constants . . . 926
46.7 Setting up simprocs . . . 926
46.7.1 Simplification of arithmetic operations on integer constants . . . 927
46.7.2 Simplification of arithmetic when nested to the right . . . 928
46.8 Code module namespace . . . 929
46.9 Printing of evaluated natural numbers as numerals . . . 929
46.10 More on auxiliary conversion . . . 929

47 Exponentiation **930**
47.1 Powers for Arbitrary Monoids . . . 930
47.2 Exponentiation on ordered types . . . 939
47.3 Miscellaneous rules . . . 948
47.4 Exponentiation for the Natural Numbers . . . 949
47.4.1 Cardinality of the Powerset . . . 951
47.5 Code generator tweak . . . 952

48 Big sum and product over finite (non-empty) sets **952**
48.1 Generic monoid operation over a set . . . 952
48.1.1 Standard sum or product indexed by a finite set . . . 952
48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset . . . 963
48.2 Generalized summation over a set . . . 965
48.2.1 Properties in more restricted classes of structures . . . 966

48.2.2 Cardinality as special case of *sum* 975
48.2.3 Cardinality of products 980
48.3 Generalized product over a set 981
48.3.1 Properties in more restricted classes of structures . . . 982

49 Chain-complete partial orders and their fixpoints 991
49.1 Chains 991
49.2 Chain-complete partial orders 992
49.3 Transfinite iteration of a function 992
49.4 Fixpoint combinator 993
49.5 Fixpoint induction 994

50 Datatype option 998
50.0.1 Operations 999
50.1 Transfer rules for the Transfer package 1005
50.1.1 Interaction with finite sets 1005
50.1.2 Code generator setup 1005

51 Partial Function Definitions 1006
51.1 Axiomatic setup 1007
51.2 Flat interpretation: tailrec and option 1011

52 Reconstructing external resolution proofs for propositional logic 1016

53 Function Definitions and Termination Proofs 1016
53.1 Definitions with default value 1017
53.2 Measure functions 1018
53.3 Congruence rules 1019
53.4 Simp rules for termination proofs 1019
53.5 Decomposition 1020
53.6 Reduction pairs 1020
53.7 Concrete orders for SCNP termination proofs 1020
53.8 Yet more induction principles on the natural numbers 1023
53.9 Tool setup 1023

54 The Integers as Equivalence Classes over Pairs of Natural Numbers 1023
54.1 Definition of integers as a quotient type 1023
54.2 Integers form a commutative ring 1024
54.3 Integers are totally ordered 1025
54.4 Ordering properties of arithmetic operations 1025
54.5 Embedding of the Integers into any *ring-1*: *of-int* 1028
54.6 Magnitude of an Integer, as a Natural Number: *nat* 1034
54.7 Lemmas about the Function *of-nat* and Orderings 1038

54.8 Cases and induction . . . 1039
54.8.1 Binary comparisons . . . 1041
54.8.2 Comparisons, for Ordered Rings . . . 1041
54.9 The Set of Integers . . . 1042
54.10*sum* and *prod* . . . 1046
54.11Setting up simplification procedures . . . 1047
54.12More Inequality Reasoning . . . 1048
54.13The functions *nat* and *int* . . . 1049
54.14Induction principles for int . . . 1051
54.15Intermediate value theorems . . . 1054
54.16Products and 1, by T. M. Rasmussen . . . 1054
54.17The divides relation . . . 1055
54.18Powers with integer exponents . . . 1060
54.19Finiteness of intervals . . . 1068
54.20Configuration of the code generator . . . 1068
54.21Duplicates . . . 1071

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets 1072
55.1 Generic lattice operations over a set . . . 1072
55.1.1 Without neutral element . . . 1072
55.1.2 With neutral element . . . 1075
55.2 Lattice operations on finite sets . . . 1078
55.3 Infimum and Supremum over non-empty sets . . . 1078
55.4 Minimum and Maximum over non-empty sets . . . 1081
55.5 An aside: code generation for *LEAST* and *GREATEST* . . . 1091
55.6 Arg Min . . . 1091
55.7 Arg Max . . . 1094

56 Division in euclidean (semi)rings 1095
56.1 Euclidean (semi)rings with explicit division and remainder . . . 1096
56.2 Euclidean (semi)rings with cancel rules . . . 1099
56.3 Uniquely determined division . . . 1108
56.4 Division on *nat* . . . 1113
56.5 Division on *int* . . . 1130
56.5.1 Basic instantiation . . . 1130
56.5.2 Algebraic foundations . . . 1131
56.5.3 Basic conversions . . . 1132
56.5.4 Euclidean division . . . 1133
56.5.5 Trivial reduction steps . . . 1136
56.5.6 More uniqueness rules . . . 1137
56.5.7 Laws for unary minus . . . 1138
56.5.8 Borders . . . 1138
56.5.9 Splitting Rules for div and mod . . . 1140

56.5.10 Algebraic rewrites . . . 1141
56.5.11 Distributive laws for conversions. . . . 1142
56.5.12 Monotonicity in the First Argument (Dividend) . . . 1143
56.5.13 Monotonicity in the Second Argument (Divisor) . . . 1143
56.5.14 Quotients of Signs . . . 1144
56.5.15 Further properties . . . 1148
56.5.16 Computing *div* and *mod* by shifting . . . 1149
56.6 Code generation . . . 1151

57 Parity in rings and semirings **1152**
57.1 Ring structures with parity and *even*/*odd* predicates . . . 1152
57.2 Instance for *nat* . . . 1157
57.3 Parity and powers . . . 1159
57.4 Instance for *int* . . . 1162
57.5 Special case: euclidean rings structurally containing the natural numbers . . . 1163
57.6 Generic symbolic computations . . . 1172
57.6.1 Computation by simplification . . . 1178
57.7 Computing congruences modulo *2* ^ *q* . . . 1180

58 Combination and Cancellation Simprocs for Numeral Expressions **1181**

59 Semiring normalization **1187**

60 Groebner bases **1191**
60.1 Groebner Bases . . . 1191

61 Set intervals **1193**
61.1 Various equivalences . . . 1195
61.2 Logical Equivalences for Set Inclusion and Equality . . . 1196
61.3 Two-sided intervals . . . 1197
61.3.1 Emptyness, singletons, subset . . . 1198
61.4 Infinite intervals . . . 1206
61.4.1 Intersection . . . 1208
61.5 Intervals of natural numbers . . . 1210
61.5.1 The Constant *lessThan* . . . 1210
61.5.2 The Constant *greaterThan* . . . 1210
61.5.3 The Constant *atLeast* . . . 1211
61.5.4 The Constant *atMost* . . . 1211
61.5.5 The Constant *atLeastLessThan* . . . 1211
61.5.6 The Constant *atLeastAtMost* . . . 1212
61.5.7 Intervals of nats with *Suc* . . . 1212
61.5.8 Intervals and numerals . . . 1213

61.5.9 Image . . . 1213
61.5.10 Finiteness . . . 1220
61.5.11 Proving Inclusions and Equalities between Unions . . 1221
61.5.12 Cardinality . . . 1223
61.6 Intervals of integers . . . 1226
61.6.1 Finiteness . . . 1226
61.6.2 Cardinality . . . 1227
61.7 Lemmas useful with the summation operator sum . . . 1229
61.7.1 Disjoint Unions . . . 1229
61.7.2 Disjoint Intersections . . . 1230
61.7.3 Some Differences . . . 1231
61.7.4 Some Subset Conditions . . . 1231
61.8 Generic big monoid operation over intervals . . . 1231
61.9 Summation indexed over intervals . . . 1234
61.9.1 Shifting bounds . . . 1241
61.9.2 Telescoping sums . . . 1243
61.9.3 The formula for geometric sums . . . 1244
61.9.4 Geometric progressions . . . 1246
61.9.5 The formulae for arithmetic sums . . . 1247
61.9.6 Division remainder . . . 1249
61.10 Products indexed over intervals . . . 1249
61.10.1 Telescoping products . . . 1251
61.11 Efficient folding over intervals . . . 1251

62 Decision Procedure for Presburger Arithmetic 1253
62.1 The $-\infty$ and $+\infty$ Properties . . . 1253
62.2 The A and B sets . . . 1255
62.3 Cooper's Theorem $-\infty$ and $+\infty$ Version . . . 1259
62.3.1 First some trivial facts about periodic sets or predicates 1259
62.3.2 The $-\infty$ Version . . . 1260
62.3.3 The $+\infty$ Version . . . 1261
62.4 Nice facts about division by $4::'a$ . . . 1265

63 Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2 1267
63.1 A skolemization tactic and proof method . . . 1267
63.2 Triggers for quantifier instantiation . . . 1268
63.3 Higher-order encoding . . . 1268
63.4 Normalization . . . 1268
63.5 Integer division and modulo for Z3 . . . 1270
63.6 Extra theorems for Alethe reconstruction . . . 1270
63.7 Setup . . . 1288
63.8 Configuration . . . 1289
63.9 General configuration options . . . 1289

63.10 Certificates . . . 1290
63.11 Tracing . . . 1291
63.12 Schematic rules for Z3 proof reconstruction . . . 1291

64 Sledgehammer: Isabelle–ATP Linkup **1293**

65 Setup for Lifting/Transfer for the set type **1294**
65.1 Relator and predicator properties . . . 1294
65.2 Quotient theorem for the Lifting package . . . 1296
65.3 Transfer rules for the Transfer package . . . 1297
65.3.1 Unconditional transfer rules . . . 1297
65.3.2 Rules requiring bi-unique, bi-total or right-total relations . . . 1298

66 The datatype of finite lists **1302**
66.1 Basic list processing functions . . . 1303
66.1.1 List homomorphism . . . 1311
66.1.2 List comprehension . . . 1311
66.1.3 [] and (#) . . . 1319
66.1.4 *length* . . . 1320
66.1.5 @ – append . . . 1321
66.1.6 *map* . . . 1325
66.1.7 *rev* . . . 1328
66.1.8 *set* . . . 1330
66.1.9 *concat* . . . 1334
66.1.10 *filter* . . . 1336
66.1.11 List partitioning . . . 1339
66.1.12 (!) . . . 1340
66.1.13 *list-update* . . . 1343
66.1.14 *last* and *butlast* . . . 1345
66.1.15 *take* and *drop* . . . 1347
66.1.16 *takeWhile* and *dropWhile* . . . 1354
66.1.17 *zip* . . . 1358
66.1.18 *list-all2* . . . 1364
66.1.19 *List.product* and *product-lists* . . . 1368
66.1.20 *fold* with natural argument order . . . 1369
66.1.21 Fold variants: *foldr* and *foldl* . . . 1372
66.1.22 *upt* . . . 1373
66.1.23 *upto*: interval-list on *int* . . . 1377
66.1.24 *successively* . . . 1378
66.1.25 *distinct* and *remdups* and *remdups-adj* . . . 1380
66.2 *distinct-adj* . . . 1393
66.2.1 *insert* . . . 1394
66.2.2 *List.union* . . . 1395

66.2.3 *find* . . . 1395
66.2.4 *count-list* . . . 1396
66.2.5 *List.extract* . . . 1398
66.2.6 *remove1* . . . 1398
66.2.7 *removeAll* . . . 1399
66.2.8 *minus-list-mset* . . . 1401
66.2.9 *minus-list-set* . . . 1402
66.2.10 *inter-list-set* . . . 1403
66.2.11 *replicate* . . . 1403
66.2.12 *indexed-from* . . . 1408
66.2.13 *rotate1* and *rotate* . . . 1409
66.2.14 *nths* — a generalization of (!) to sets . . . 1412
66.2.15 *subseqs* and *List.n-lists* . . . 1414
66.2.16 *splice* . . . 1415
66.2.17 *shuffles* . . . 1416
66.2.18 Transpose . . . 1419
66.2.19 *min* and *arg-min* . . . 1421
66.2.20 (In)finiteness . . . 1421
66.3 Sorting . . . 1424
66.3.1 *sorted-wrt* . . . 1424
66.3.2 *sorted* . . . 1426
66.3.3 Sorting functions . . . 1431
66.3.4 *transpose* on sorted lists . . . 1435
66.3.5 *sorted-key-list-of-set* . . . 1438
66.3.6 *lists*: the list-forming operator over sets . . . 1445
66.3.7 Inductive definition for membership . . . 1447
66.3.8 Lists as Cartesian products . . . 1447
66.3.9 Transitive Closure on Lists . . . 1448
66.4 Relations on Lists . . . 1449
66.4.1 Length Lexicographic Ordering . . . 1449
66.4.2 Lexicographic Ordering . . . 1454
66.4.3 Lexicographic combination of measure functions . . . 1462
66.4.4 Lifting Relations to Lists: one element . . . 1463
66.4.5 Lifting Relations to Lists: all elements . . . 1466
66.5 Size function . . . 1469
66.6 Monad operation . . . 1469
66.7 Code generation . . . 1470
66.7.1 Counterparts for set-related operations . . . 1470
66.7.2 Special implementations . . . 1477
66.7.3 Operations for optimization and efficiency . . . 1478
66.7.4 Implementation of sets by lists . . . 1479
66.7.5 Pretty lists . . . 1482
66.7.6 Use convenient predefined operations . . . 1483
66.8 Setup for Lifting/Transfer . . . 1484

66.8.1 Transfer rules for the Transfer package 1484
66.9 Misc 1489

67 Sum and product over lists **1489**
67.1 List summation 1491
67.2 Horner sums 1498
67.3 Further facts about *List.n-lists* 1501
67.4 Tools setup 1502
67.5 List product 1502

68 Bit operations in suitable algebraic structures **1504**
68.1 Abstract bit structures 1504
68.2 Bit operations 1516
68.3 Common algebraic structure 1537
68.4 Instance *int* 1540
68.5 Instance *nat* 1555
68.6 Symbolic computations on numeral expressions 1561
68.7 Symbolic computations for code generation 1572
68.8 More properties 1577
68.9 Bit concatenation 1578
68.10Taking bits with sign propagation 1580
68.11Key ideas of bit operations 1586
68.12Lemma duplicates and other 1588

69 Numeric types for code generation onto target language numerals only **1592**
69.1 Type of target language integers 1593
69.2 Code theorems for target language integers 1601
69.3 Serializer setup for target language integers 1611
69.4 Type of target language naturals 1616
69.5 Inductive representation of target language naturals 1622
69.6 Code refinement for target language naturals 1623

70 A HOL random engine **1626**
70.1 Auxiliary functions 1626
70.2 Random seeds 1626
70.3 Base selectors 1627
70.4 *ML* interface 1629

71 Maps **1630**
71.1 *empty* 1632
71.2 *map-upd* 1632
71.3 *map-of* 1633
71.4 *map-option* related 1636

71.5 *map-comp* related . . . 1636
71.6 ++ . . . 1637
71.7 *restrict-map* . . . 1638
71.8 *map-upds* . . . 1639
71.9 *dom* . . . 1641
71.10 *ran* . . . 1644
71.11 *graph* . . . 1646
71.12 *map-le* . . . 1647
71.13 Various . . . 1649

72 Finite types as explicit enumerations **1651**
72.1 Class *enum* . . . 1651
72.2 Implementations using *enum* . . . 1652
72.2.1 Unbounded operations and quantifiers . . . 1652
72.2.2 An executable choice operator . . . 1652
72.2.3 Equality and order on functions . . . 1653
72.2.4 Operations on relations . . . 1654
72.2.5 Bounded accessible part . . . 1654
72.3 Default instances for *enum* . . . 1656
72.4 Small finite types . . . 1661
72.5 Closing up . . . 1675

73 Character and string types **1675**
73.1 Strings as list of bytes . . . 1675
73.1.1 Bytes as datatype . . . 1675
73.2 Strings as dedicated type for target language code generation 1682
73.2.1 Logical specification . . . 1682
73.2.2 Syntactic representation . . . 1684
73.2.3 Operations . . . 1686
73.2.4 Executable conversions . . . 1688
73.2.5 Technical code generation setup . . . 1689
73.2.6 Code generation utility . . . 1694
73.2.7 Finally . . . 1694

74 Reflecting Pure types into HOL **1694**

75 Predicates as enumerations **1696**
75.1 The type of predicate enumerations (a monad) . . . 1696
75.2 Emptiness check and definite choice . . . 1700
75.3 Derived operations . . . 1703
75.4 Implementation . . . 1705

76 Lazy sequences **1711**
76.1 Type of lazy sequences . . . 1711
76.2 Code setup . . . 1715
76.3 Generator Sequences . . . 1715
76.3.1 General lazy sequence operation . . . 1715
76.3.2 Small lazy typeclasses . . . 1716
76.4 With Hit Bound Value . . . 1717

77 Depth-Limited Sequences with failure element **1718**
77.1 Depth-Limited Sequence . . . 1718
77.2 Positive Depth-Limited Sequence . . . 1720
77.3 Negative Depth-Limited Sequence . . . 1721
77.4 Negation . . . 1721

78 Term evaluation using the generic code generator **1723**
78.1 Term representation . . . 1723
78.1.1 Terms and class *term-of* . . . 1723
78.1.2 Syntax . . . 1723
78.2 Tools setup and evaluation . . . 1724
78.3 Dedicated *term-of* instances . . . 1724
78.4 Generic reification . . . 1725
78.5 Diagnostic . . . 1725

79 A simple counterexample generator performing random testing **1726**
79.1 Catching Match exceptions . . . 1726
79.2 The *random* class . . . 1726
79.3 Fundamental and numeric types . . . 1726
79.4 Complex generators . . . 1729
79.5 Deriving random generators for datatypes . . . 1731
79.6 Code setup . . . 1731

80 The Random-Predicate Monad **1732**

81 Various kind of sequences inside the random monad **1733**

82 A simple counterexample generator performing exhaustive testing **1737**
82.1 Basic operations for exhaustive generators . . . 1737
82.2 Exhaustive generator type classes . . . 1737
82.2.1 A smarter enumeration scheme for functions over finite datatypes . . . 1743
82.3 Bounded universal quantifiers . . . 1750
82.4 Fast exhaustive combinators . . . 1750
82.5 Continuation passing style functions as plus monad . . . 1751

82.6 Defining generators for any first-order data type 1753
82.7 Defining generators for abstract types 1753

83 A compiler for predicates defined by introduction rules 1753
83.1 Set membership as a generator predicate 1754

84 Counterexample generator performing narrowing-based testing 1756
84.1 Counterexample generator 1756
84.1.1 Code generation setup 1756
84.1.2 Narrowing's deep representation of types and terms . 1757
84.1.3 From narrowing's deep representation of terms to *HOL.Code-Evaluation*'s terms 1757
84.1.4 Auxilary functions for Narrowing 1757
84.1.5 Narrowing's basic operations 1758
84.1.6 Narrowing generator type class 1759
84.1.7 class *is-testable* 1759
84.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way 1759
84.1.9 Setting up the counterexample generator 1760
84.2 Narrowing for sets 1760
84.3 Narrowing for integers 1761
84.4 The *find-unused-assms* command 1763
84.5 Closing up 1763

85 Program extraction for HOL 1763
85.1 Setup 1763
85.2 Type of extracted program 1764
85.3 Realizability 1765
85.4 Computational content of basic inference rules 1766

86 Extensible records with structural subtyping 1772
86.1 Introduction 1772
86.2 Operators and lemmas for types isomorphic to tuples 1773
86.3 Logical infrastructure for records 1774
86.4 Concrete record syntax 1780
86.5 Record package 1781

87 Greatest common divisor and least common multiple 1781
87.1 Abstract bounded quasi semilattices as common foundation . 1781
87.2 Abstract GCD and LCM 1784
87.3 An aside: GCD and LCM on finite sets for incomplete gcd rings 1803
87.4 Coprimality 1806

87.5 GCD and LCM for multiplicative normalisation functions . . 1815
87.6 GCD and LCM on *nat* and *int* 1817
87.7 Bezout's theorem 1825
87.8 LCM properties on *nat* and *int* 1830
87.9 The complete divisibility lattice on *nat* and *int* 1832
87.9.1 Setwise GCD and LCM for integers 1834
87.10 GCD and LCM on *integer* 1836
87.11 Characteristic of a semiring 1839

88 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL 1843

89 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors 1849
89.1 Equivalence relations, quotients, and Hilbert's choice 1853

90 Filters on predicates 1855
90.1 Filters 1855
90.1.1 Eventually 1856
90.2 Frequently as dual to eventually 1858
90.2.1 Finer-than relation 1861
90.2.2 Map function for filters 1867
90.2.3 Contravariant map function for filters 1868
90.2.4 Standard filters 1871
90.2.5 Order filters 1872
90.3 Sequentially 1875
90.4 Increasing finite subsets 1876
90.5 The cofinite filter 1877
90.5.1 Product of filters 1878
90.6 Limits 1884
90.7 Limits to *at-top* and *at-bot* 1887
90.8 Setup *'a filter* for lifting and transfer 1890

91 Conditionally-complete Lattices 1901
91.1 Preorders 1902
91.2 Lattices 1905
91.3 Conditionally complete lattices 1906
91.4 Complete lattices 1912
91.5 Instances 1914

92 Factorial Function, Rising Factorials 1923
92.1 Factorial Function 1924
92.2 Pochhammer's symbol: generalized rising factorial 1927
92.3 Misc 1932

93 Binomial Coefficients, Binomial Theorem, Inclusion-exclusion Principle **1932**
93.1 Binomial coefficients . . . 1932
93.2 The binomial theorem (courtesy of Tobias Nipkow): . . . 1936
93.3 Generalized binomial coefficients . . . 1940
93.4 Summation on the upper index . . . 1950
93.5 More on Binomial Coefficients . . . 1956
93.6 Inclusion-exclusion principle . . . 1959
93.7 Versions for unrestrictedly additive functions . . . 1962
93.8 General "Moebius inversion" inclusion-exclusion principle . . . 1964
93.9 Executable code . . . 1966
93.10 Parse arguments . . . 1993
93.11 Replay `rare_rules` . . . 1994
93.12 Register `rare_rules` . . . 1994

94 Main HOL **1996**
94.1 Namespace cleanup . . . 1997
94.2 Syntax cleanup . . . 1997
94.3 Lattice syntax . . . 1998

95 Archimedean Fields, Floor and Ceiling Functions **1998**
95.1 Class of Archimedean fields . . . 1999
95.2 Existence and uniqueness of floor function . . . 2001
95.3 Floor function . . . 2002
95.4 Ceiling function . . . 2007
95.4.1 Ceiling with numerals. . . . 2008
95.4.2 Addition and subtraction of integers. . . . 2010
95.5 Negation . . . 2011
95.6 Natural numbers . . . 2011
95.7 Frac Function . . . 2011
95.8 Fractional part arithmetic . . . 2013
95.9 Rounding to the nearest integer . . . 2015

96 Rational numbers **2017**
96.1 Rational numbers as quotient . . . 2017
96.1.1 Construction of the type of rational numbers . . . 2017
96.1.2 Representation and basic operations . . . 2018
96.1.3 Function *normalize* . . . 2022
96.1.4 Various . . . 2025
96.1.5 The ordered field of rational numbers . . . 2026
96.1.6 Rationals are an Archimedean field . . . 2029
96.2 Linear arithmetic setup . . . 2030
96.3 Embedding from Rationals to other Fields . . . 2030
96.4 The Set of Rational Numbers . . . 2033

96.5 Implementation of rational numbers as pairs of integers . . . 2035
96.6 Setup for Nitpick . . . 2040
96.7 Float syntax . . . 2041
96.8 Hiding implementation details . . . 2041

97 Development of the Reals using Cauchy Sequences 2041
97.1 Preliminary lemmas . . . 2042
97.2 Sequences that converge to zero . . . 2042
97.3 Cauchy sequences . . . 2044
97.4 Equivalence relation on Cauchy sequences . . . 2048
97.5 The field of real numbers . . . 2049
97.6 Positive reals . . . 2052
97.7 Completeness . . . 2055
97.8 Supremum of a set of reals . . . 2059
97.9 Hiding implementation details . . . 2060
97.10 Embedding numbers into the Reals . . . 2060
97.11 Embedding the Naturals into the Reals . . . 2062
97.12 The Archimedean Property of the Reals . . . 2062
97.13 Rationals . . . 2063
97.14 Density of the Rational Reals in the Reals . . . 2065
97.15 Numerals and Arithmetic . . . 2066
97.16 Simprules combining $x + y$ and *0* . . . 2066
97.17 Lemmas about powers . . . 2067
97.18 Density of the Reals . . . 2068
97.19 Archimedean properties and useful consequences . . . 2068
97.20 Floor and Ceiling Functions from the Reals to the Integers . . . 2071
97.21 Exponentiation with floor . . . 2074
97.22 Implementation of rational real numbers . . . 2074
97.23 Setup for Nitpick . . . 2077
97.24 Setup for SMT . . . 2078
97.25 Setup for Argo . . . 2079

98 Topological Spaces 2079
98.1 Topological space . . . 2079
98.2 Hausdorff and other separation properties . . . 2083
98.3 Generators for topologies . . . 2084
98.4 Order topologies . . . 2085
98.5 Setup some topologies . . . 2088
98.5.1 Boolean is an order topology . . . 2088
98.5.2 Topological filters . . . 2090
98.5.3 Tendsto . . . 2096
98.5.4 Rules about *Lim* . . . 2103
98.6 Limits on sequences . . . 2105
98.7 Monotone sequences and subsequences . . . 2106

98.7.1 Definition of subsequence. . . . 2106
98.7.2 Subsequence (alternative definition, (e.g. Hoskins) . . 2108
98.7.3 Increasing and Decreasing Series 2113
98.8 First countable topologies 2113
98.9 Function limit at a point 2117
98.9.1 Relation of *LIM* and *LIMSEQ* 2119
98.10 Continuity 2122
98.10.1 Continuity on a set 2122
98.10.2 Continuity at a point 2127
98.10.3 Open-cover compactness 2133
98.11 Finite intersection property 2134
98.12 Connectedness 2138

99 Linear Continuum Topologies 2144
99.1 Intermediate Value Theorem 2147
99.2 Uniform spaces 2152
99.2.1 Totally bounded sets 2154
99.2.2 Cauchy filter 2154
99.2.3 Uniformly continuous functions 2156

100 Product Topology 2157
100.1 Product is a topological space 2157
100.1.1 Continuity of operations 2160
100.1.2 Connectedness of products 2162
100.1.3 Separation axioms 2162
100.2 A generic notion of the convex, affine, conic hull, or closed "hull". . . . 2164

101 Modules 2166
101.1 Locale for additive functions 2166

102 Subspace 2168

103 Span: subspace generated by a set 2169

104 Dependent and independent sets 2174

105 Representation of a vector on a specific basis 2176

106 Vector Spaces 2186

107 Vector Spaces and Algebras over the Reals 2225
107.1 Real vector spaces 2225
107.2 Embedding of the Reals into any *real-algebra-1*: *of-real* . . . 2230
107.3 The Set of Real Numbers 2233

107.4 Ordered real vector spaces . . . 2235
107.5 Real normed vector spaces . . . 2240
107.6 Metric spaces . . . 2249
107.7 Class instances for real numbers . . . 2252
107.8 Extra type constraints . . . 2254
107.9 Sign function . . . 2254
107.10 Bounded Linear and Bilinear Operators . . . 2256
107.11 Filters and Limits on Metric Space . . . 2262
107.11.1 Limits of Sequences . . . 2266
107.11.2 Limits of Functions . . . 2266
107.12 Complete metric spaces . . . 2268
107.13 Cauchy sequences . . . 2268
107.13.1 Cauchy Sequences are Convergent . . . 2273
107.14 The set of real numbers is a complete metric space . . . 2274

108 Limits on Real Vector Spaces 2278
108.1 Filter going to infinity norm . . . 2278
108.1.1 Boundedness . . . 2279
108.1.2 Bounded Sequences . . . 2280
108.1.3 A Few More Equivalence Theorems for Boundedness . . . 2282
108.1.4 Upper Bounds and Lubs of Bounded Sequences . . . 2283
108.1.5 Polynomal function extremal theorem, from HOL Light . . . 2285
108.2 Convergence to Zero . . . 2287
108.3 Distance and norms . . . 2290
108.4 Topological Monoid . . . 2292
108.4.1 Topological group . . . 2293
108.4.2 Linear operators and multiplication . . . 2295
108.4.3 Inverse and division . . . 2301
108.5 Relate *at*, *at-left* and *at-right* . . . 2309
108.6 Floor and Ceiling . . . 2324
108.7 Limits of Sequences . . . 2325
108.8 Convergence on sequences . . . 2329
108.9 More about *filterlim* (thanks to Wenda Li) . . . 2334
108.10 Power Sequences . . . 2337
108.11 Limits of Functions . . . 2338
108.12 Continuity . . . 2340
108.13 Uniform Continuity . . . 2341
108.14 Nested Intervals and Bisection – Needed for Compactness . . . 2343
108.15 Boundedness of continuous functions . . . 2346

109 Infinite Series 2349
109.1 Definition of infinite summability . . . 2349
109.2 Infinite summability on topological monoids . . . 2350
109.3 Infinite summability on ordered, topological monoids . . . 2353

109.4 Infinite summability on topological monoids . . . 2355
109.5 Infinite summability on real normed vector spaces . . . 2357
109.6 Infinite summability on real normed algebras . . . 2360
109.7 Infinite summability on real normed fields . . . 2361
109.8 Telescoping . . . 2363
109.9 Infinite summability on Banach spaces . . . 2364
109.10 The Ratio Test . . . 2366
109.11 Cauchy Product Formula . . . 2368
109.12 Series on *reals* . . . 2370

110 Differentiation **2377**
110.1 Frechet derivative . . . 2378
110.1.1 Limit transformation for derivatives . . . 2383
110.2 Continuity . . . 2384
110.3 Composition . . . 2385
110.4 Uniqueness . . . 2391
110.5 Differentiability predicate . . . 2392
110.6 Vector derivative . . . 2396
110.7 Derivatives . . . 2399
110.8 Local extrema . . . 2408
110.9 Rolle's Theorem . . . 2411
110.10 Mean Value Theorem . . . 2414
110.10.1 A function is constant if its derivative is 0 over an interval. . . . 2415
110.10.2 A function with positive derivative is increasing . . . 2417
110.11 Generalized Mean Value Theorem . . . 2422
110.12 L'Hopitals rule . . . 2424
110.13 Logarithmic derivatives . . . 2432

111 Nth Roots of Real Numbers **2436**
111.1 Existence of Nth Root . . . 2436
111.2 Nth Root . . . 2437
111.3 Square Root . . . 2444
111.4 Square Root of Sum of Squares . . . 2448

112 Power Series, Transcendental Functions etc. **2452**
112.1 Properties of Power Series . . . 2454
112.2 Alternating series test / Leibniz formula . . . 2458
112.3 Term-by-Term Differentiability of Power Series . . . 2462
112.4 The Derivative of a Power Series Has the Same Radius of Convergence . . . 2467
112.5 Derivability of power series . . . 2471
112.6 Exponential Function . . . 2477
112.6.1 Properties of the Exponential Function . . . 2479

112.6.2 Properties of the Exponential Function on Reals . . . 2484
112.7 Natural Logarithm . . . 2487
112.7.1 A couple of simple bounds . . . 2503
112.8 The general logarithm . . . 2504
112.9 Sine and Cosine . . . 2523
112.10 Properties of Sine and Cosine . . . 2528
112.11 Deriving the Addition Formulas . . . 2528
112.12 The Constant Pi . . . 2533
112.13 More Corollaries about Sine and Cosine . . . 2548
112.14 Tangent . . . 2553
112.15 Cotangent . . . 2561
112.16 Inverse Trigonometric Functions . . . 2562
112.17 Prove Totality of the Trigonometric Functions . . . 2572
112.18 Machin's formula . . . 2578
112.19 Introducing the inverse tangent power series . . . 2580
112.20 Existence of Polar Coordinates . . . 2589
112.21 Basics about polynomial functions: products, extremal behaviour and root counts . . . 2590
112.22 Hyperbolic functions . . . 2595
112.22.1 More specific properties of the real functions . . . 2601
112.22.2 Limits . . . 2605
112.22.3 Properties of the inverse hyperbolic functions . . . 2606
112.23 Simprocs for root and power literals . . . 2615

113 Complex Numbers: Rectangular and Polar Representation 2619
113.1 Addition and Subtraction . . . 2620
113.2 Multiplication and Division . . . 2620
113.3 Scalar Multiplication . . . 2622
113.4 Numerals, Arithmetic, and Embedding from R . . . 2622
113.5 The Complex Number i . . . 2624
113.6 Vector Norm . . . 2626
113.7 Absolute value . . . 2628
113.8 Completeness of the Complexes . . . 2628
113.9 Complex Conjugation . . . 2630
113.10 Basic Lemmas . . . 2634
113.11 Polar Form for Complex Numbers . . . 2637
113.11.1 $\cos\theta + i\sin\theta$ . . . 2637
113.11.2 $r(\cos\theta + i\sin\theta)$ . . . 2638
113.11.3 Complex exponential . . . 2639
113.11.4 Complex argument . . . 2642
113.12 Complex n-th roots . . . 2644
113.13 Square root of complex numbers . . . 2647

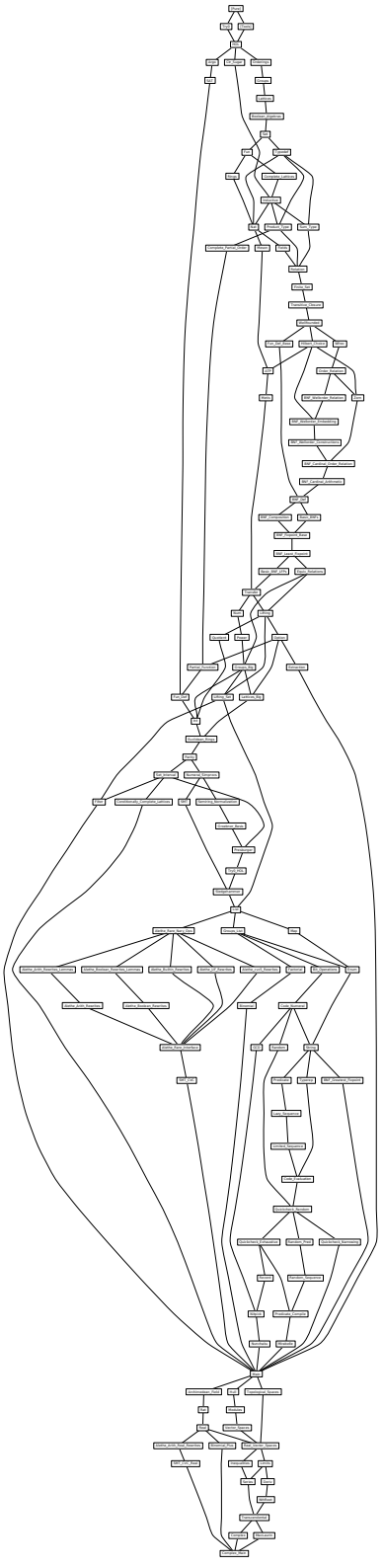
114 MacLaurin and Taylor Series **2651**
114.1 Maclaurin's Theorem with Lagrange Form of Remainder . . . 2651
114.2 More Convenient "Bidirectional" Version. 2654
114.3 Version for Exponential Function 2656
114.4 Version for Sine Function 2656
114.5 Maclaurin Expansion for Cosine Function 2657

115 Taylor series **2659**

116 More facts about binomial coefficients **2661**
116.1 More facts about binomial coefficients 2662
116.2 Results about binomials and integers, thanks to Alexander Maletzky 2665
116.3 Sums 2670

117 Lemmas to reconstruct `alethe_poly_simp_rel` **2672**

118 Comprehensive Complex Theory **2676**



1 Loading the code generator and related modules

theory *Code-Generator*
imports *Pure*
keywords
  print-codeproc code-thms code-deps :: *diag* **and**
  export-code code-identifier code-printing code-reserved
    code-monad code-reflect :: *thy-decl* **and**
  checking **and**
  datatypes functions module-name file file-prefix
    constant type-constructor type-class class-relation class-instance code-module
    :: *quasi-command*
begin

ML-file ‹~~/*src*/*Tools*/*cache-io.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-preproc.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-symbol.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-thingol.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-simp.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-printer.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-target.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-namespace.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-ml.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-haskell.ML*›
ML-file ‹~~/*src*/*Tools*/*Code*/*code-scala.ML*›

code-datatype *TYPE*('*a*::{})

definition *holds* :: *prop* **where**
  holds ≡ ((λ*x*::*prop*. *x*) ≡ (λ*x*. *x*))

lemma *holds*: *PROP holds*
  by (*unfold holds-def*) (*rule reflexive*)

code-datatype *holds*

lemma *implies-code* [*code*]:
  (*PROP P* ⟹ *PROP holds*) ≡ *PROP holds*
  (*PROP holds* ⟹ *PROP P*) ≡ *PROP P*
proof −
  show (*PROP holds* ⟹ *PROP P*) ≡ *PROP P*
  proof
    assume *PROP holds* ⟹ *PROP P*
    then show *PROP P* **using** *holds* **.**
  next
    assume *PROP P*
    then show *PROP P* **.**
  qed
next

show (*PROP P* $\Longrightarrow$ *PROP holds*) $\equiv$ *PROP holds*
by *rule* (*rule holds*)+
qed

ML-file ‹~~/*src*/*Tools*/*Code*/*code-runtime.ML*›
ML-file ‹~~/*src*/*Tools*/*nbe.ML*›

hide-const (**open**) *holds*

end

theory *Try0*
imports *Pure*
keywords *try0* :: *diag*
begin

ML-file ‹*Tools*/*try0.ML*›
ML-file ‹*Tools*/*try0-util.ML*›

ML ‹
val () =
Try0.register-proof-method simp {*run-if-auto-try* = *true*}
(*Try0-Util.apply-raw-named-method simp false Try0-Util.simp-attrs Simplifier-Trace.disable*)
handle Symtab.DUP - => ()
›

end

2 The basis of Higher-Order Logic

theory *HOL*
imports *Pure Try0 Tools.Code-Generator*
keywords
try solve-direct quickcheck print-coercions print-claset
print-induct-rules :: *diag* **and**
quickcheck-params :: *thy-decl*
abbrevs *?<* = $\exists_{\leq 1}$
begin

ML-file ‹~~/*src*/*Tools*/*misc-legacy.ML*›
ML-file ‹~~/*src*/*Tools*/*try.ML*›
ML-file ‹~~/*src*/*Tools*/*quickcheck.ML*›
ML-file ‹~~/*src*/*Tools*/*solve-direct.ML*›
ML-file ‹~~/*src*/*Tools*/*IsaPlanner*/*zipper.ML*›
ML-file ‹~~/*src*/*Tools*/*IsaPlanner*/*isand.ML*›
ML-file ‹~~/*src*/*Tools*/*IsaPlanner*/*rw-inst.ML*›
ML-file ‹~~/*src*/*Provers*/*hypsubst.ML*›
ML-file ‹~~/*src*/*Provers*/*splitter.ML*›

ML-file ‹~~/*src*/*Provers*/*classical.ML*›
ML-file ‹~~/*src*/*Provers*/*blast.ML*›
ML-file ‹~~/*src*/*Provers*/*clasimp.ML*›
ML-file ‹~~/*src*/*Tools*/*eqsubst.ML*›
ML-file ‹~~/*src*/*Provers*/*quantifier1.ML*›
ML-file ‹~~/*src*/*Tools*/*atomize-elim.ML*›
ML-file ‹~~/*src*/*Tools*/*cong-tac.ML*›
ML-file ‹~~/*src*/*Tools*/*intuitionistic.ML*› **setup** ‹*Intuitionistic.method-setup* ***binding*** ‹*iprover*››
ML-file ‹~~/*src*/*Tools*/*project-rule.ML*›
ML-file ‹~~/*src*/*Tools*/*subtyping.ML*›
ML-file ‹~~/*src*/*Tools*/*case-product.ML*›

ML ‹
val - =
Try.tool-setup
{*name = try0*, *weight = 30*, *auto-option =* ***system-option*** ‹*auto-methods*›,
body = fn auto => fst o Try0.generic-try0 (if auto then Try0.Auto-Try else Try0.Try) NONE Try0.empty-facts}
›

ML ‹*Plugin-Name.declare-setup* ***binding*** ‹*extraction*››

ML ‹
Plugin-Name.declare-setup ***binding*** ‹*quickcheck-random*›;
Plugin-Name.declare-setup ***binding*** ‹*quickcheck-exhaustive*›;
Plugin-Name.declare-setup ***binding*** ‹*quickcheck-bounded-forall*›;
Plugin-Name.declare-setup ***binding*** ‹*quickcheck-full-exhaustive*›;
Plugin-Name.declare-setup ***binding*** ‹*quickcheck-narrowing*›;
›
ML ‹
Plugin-Name.define-setup ***binding*** ‹*quickcheck*›
[***plugin*** ‹*quickcheck-exhaustive*›,
plugin ‹*quickcheck-random*›,
plugin ‹*quickcheck-bounded-forall*›,
plugin ‹*quickcheck-full-exhaustive*›,
plugin ‹*quickcheck-narrowing*›]
›

2.1 Primitive logic

The definition of the logic is based on Mike Gordon's technical report [2] that describes the first implementation of HOL. However, there are a number of differences. In particular, we start with the definite description operator and introduce Hilbert's ε operator only much later. Moreover, axiom $(P \longrightarrow Q) \longrightarrow (Q \longrightarrow P) \longrightarrow (P = Q)$ is derived from the other axioms. The fact that this axiom is derivable was first noticed by Bruno Barras (for Mike Gordon's line of HOL systems) and later independently by Alexander

Maletzky (for Isabelle/HOL).

2.1.1 Core syntax

setup ‹*Axclass.class-axiomatization* (***binding*** ‹*type*›, []›
default-sort *type*
setup ‹*Object-Logic.add-base-sort* ***sort*** ‹*type*››

setup ‹*Proofterm.set-preproc* (*Proof-Rewrite-Rules.standard-preproc* []›

axiomatization where *fun-arity*: *OFCLASS*('*a* ⇒ '*b*, *type-class*)
instance *fun* :: (*type*, *type*) *type* **by** (*rule fun-arity*)

axiomatization where *itself-arity*: *OFCLASS*('*a itself*, *type-class*)
instance *itself* :: (*type*) *type* **by** (*rule itself-arity*)

typedecl *bool*

judgment *Trueprop* :: *bool* ⇒ *prop* (‹(‹*notation=judgment*›-)› *5*)

axiomatization *implies* :: [*bool*, *bool*] ⇒ *bool* (**infixr** ‹⟶› *25*)
and *eq* :: ['*a*, '*a*] ⇒ *bool*
and *The* :: ('*a* ⇒ *bool*) ⇒ '*a*

notation (*input*)
eq (**infixl** ‹=› *50*)
notation (**output**)
eq (**infix** ‹=› *50*)

The input syntax for *eq* is more permissive than the output syntax because of the large amount of material that relies on infixl.

2.1.2 Defined connectives and quantifiers

definition *True* :: *bool*
where *True* ≡ ((λ*x*::*bool*. *x*) = (λ*x*. *x*))

definition *All* :: ('*a* ⇒ *bool*) ⇒ *bool* (**binder** ‹∀› *10*)
where *All P* ≡ (*P* = (λ*x*. *True*))

definition *Ex* :: ('*a* ⇒ *bool*) ⇒ *bool* (**binder** ‹∃› *10*)
where *Ex P* ≡ ∀ *Q*. (∀ *x*. *P x* ⟶ *Q*) ⟶ *Q*

definition *False* :: *bool*
where *False* ≡ (∀ *P*. *P*)

definition *Not* :: *bool* ⇒ *bool* (‹(‹*open-block notation=*‹*prefix* ¬››¬ -)› [*40*] *40*)
where *not-def*: ¬ *P* ≡ *P* ⟶ *False*

definition *conj* :: [*bool*, *bool*] ⇒ *bool* (**infixr** ‹∧› *35*)
where *and-def*: $P \wedge Q \equiv \forall R.\ (P \longrightarrow Q \longrightarrow R) \longrightarrow R$

definition *disj* :: [*bool*, *bool*] ⇒ *bool* (**infixr** ‹∨› *30*)
where *or-def*: $P \vee Q \equiv \forall R.\ (P \longrightarrow R) \longrightarrow (Q \longrightarrow R) \longrightarrow R$

definition *Uniq* :: ('*a* ⇒ *bool*) ⇒ *bool*
where *Uniq P* ≡ $(\forall x\ y.\ P\ x \longrightarrow P\ y \longrightarrow y = x)$

definition *Ex1* :: ('*a* ⇒ *bool*) ⇒ *bool*
where *Ex1 P* ≡ $\exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow y = x)$

2.1.3 Additional concrete syntax

syntax (*ASCII*) *-Uniq* :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=4 notation=‹binder ?<››?<* -./ -)› [*0*, *10*] *10*)
syntax *-Uniq* :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=2 notation=‹binder* $\exists_{\leq 1}$*››*$\exists_{\leq 1}$ -./ -)› [*0*, *10*] *10*)

syntax-consts *-Uniq* ⇌ *Uniq*

translations $\exists_{\leq 1}x.\ P$ ⇌ *CONST Uniq* $(\lambda x.\ P)$

typed-print-translation ‹
[(***const-syntax*** ‹*Uniq*›, *Syntax-Trans.preserve-binder-abs-tr'* ***syntax-const*** ‹*-Uniq*›)]
› — to avoid eta-contraction of body

syntax (*ASCII*)
-Ex1 :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder EX!››EX!* -./ -)› [*0*, *10*] *10*)
syntax (*input*)
-Ex1 :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder ?!››?!* -./ -)› [*0*, *10*] *10*)
syntax *-Ex1* :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder* ∃*!››*∃*!*-./ -)› [*0*, *10*] *10*)

syntax-consts *-Ex1* ⇌ *Ex1*

translations $\exists!x.\ P$ ⇌ *CONST Ex1* $(\lambda x.\ P)$

typed-print-translation ‹
[(***const-syntax*** ‹*Ex1*›, *Syntax-Trans.preserve-binder-abs-tr'* ***syntax-const*** ‹*-Ex1*›)]
› — to avoid eta-contraction of body

syntax
-Not-Ex :: *idts* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder* ∄*››*∄ -./ -)› [*0*, *10*] *10*)

-Not-Ex1 :: *pttrn* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∄!››∄!-./ -*)› [*0*, *10*] *10*)
syntax-consts
-Not-Ex ⇌ *Ex* **and**
-Not-Ex1 ⇌ *Ex1*
translations
∄*x*. *P* ⇌ ¬ (∃*x*. *P*)
∄!*x*. *P* ⇌ ¬ (∃!*x*. *P*)

abbreviation *not-equal* :: [*'a*, *'a*] ⇒ *bool* (**infix** ‹≠› *50*)
where *x* ≠ *y* ≡ ¬ (*x* = *y*)

notation (*ASCII*)
Not (‹(‹*open-block notation=‹prefix ~››~ -*)› [*40*] *40*) **and**
conj (**infixr** ‹&› *35*) **and**
disj (**infixr** ‹|› *30*) **and**
implies (**infixr** ‹–>› *25*) **and**
not-equal (**infix** ‹~=› *50*)

abbreviation (*iff*)
iff :: [*bool*, *bool*] ⇒ *bool* (**infixr** ‹⟷› *25*)
where *A* ⟷ *B* ≡ *A* = *B*

syntax *-The* :: [*pttrn*, *bool*] ⇒ *'a* (‹(‹*indent=3 notation=‹binder THE›› THE -./ -*)› [*0*, *10*] *10*)
syntax-consts *-The* ⇌ *The*
translations *THE x*. *P* ⇌ *CONST The* (λ*x*. *P*)
print-translation ‹
[(***const-syntax***‹*The*›, *fn ctxt* => *fn* [*Abs abs*] =>
let val (*x*, *t*) = *Syntax-Trans.atomic-abs-tr' ctxt abs*
in Syntax.const ***syntax-const***‹*-The*› $ *x* $ *t end*)]
› — To avoid eta-contraction of body

nonterminal *case-syn* **and** *cases-syn*
syntax
-case-syntax :: [*'a*, *cases-syn*] ⇒ *'b* (‹(‹*notation=‹mixfix case expression›› case - of/ -*)› *10*)
-case1 :: [*'a*, *'b*] ⇒ *case-syn*
(‹(‹*indent=2 notation=‹mixfix case clause›› (‹open-block notation=‹pattern case›› -*) ⇒/ -)› *10*)
:: *case-syn* ⇒ *cases-syn* (‹-›)
-case2 :: [*case-syn*, *cases-syn*] ⇒ *cases-syn* (‹-/ | -›)
syntax (*ASCII*)
-case1 :: [*'a*, *'b*] ⇒ *case-syn*
(‹(‹*indent=2 notation=‹mixfix case clause›› (‹open-block notation=‹pattern case›› -*) =>/ -)› *10*)

notation (*ASCII*)

All (**binder** ‹*ALL* › *10*) **and**
Ex (**binder** ‹*EX* › *10*)

notation (*input*)
All (**binder** ‹! › *10*) **and**
Ex (**binder** ‹*?* › *10*)

2.1.4 Axioms and basic definitions

axiomatization where
refl: $t = (t::'a)$ **and**
subst: $s = t \Longrightarrow P\ s \Longrightarrow P\ t$ **and**
ext: $(\bigwedge x::'a.\ (f\ x ::'b) = g\ x) \Longrightarrow (\lambda x.\ f\ x) = (\lambda x.\ g\ x)$
— Extensionality is built into the meta-logic, and this rule expresses a related property. It is an eta-expanded version of the traditional rule, and similar to the ABS rule of HOL **and**

the-eq-trivial: $(THE\ x.\ x = a) = (a::'a)$

axiomatization where
impI: $(P \Longrightarrow Q) \Longrightarrow P \longrightarrow Q$ **and**
mp: $[\![P \longrightarrow Q;\ P]\!] \Longrightarrow Q$ **and**

True-or-False: $(P = True) \vee (P = False)$

definition *If* :: $bool \Rightarrow 'a \Rightarrow 'a \Rightarrow 'a$ (‹(‹*notation=‹mixfix if expression››if* (-)/ *then* (-)/ *else* (-))› [*0*, *0*, *10*] *10*)
where $If\ P\ x\ y \equiv (THE\ z::'a.\ (P = True \longrightarrow z = x) \wedge (P = False \longrightarrow z = y))$

definition *Let* :: $'a \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'b$
where $Let\ s\ f \equiv f\ s$

nonterminal *letbinds* **and** *letbind*

open-bundle *let-syntax*
begin

syntax
-bind :: [*pttrn*, *'a*] ⇒ *letbind* (‹(‹*indent=2 notation=‹mixfix let binding››- =/ -*)› *10*)
:: *letbind* ⇒ *letbinds* (‹-›)
-binds :: [*letbind*, *letbinds*] ⇒ *letbinds* (‹-;/ -›)
-Let :: [*letbinds*, *'a*] ⇒ *'a* (‹(‹*notation=‹mixfix let expression››let* (-)/ *in* (-))› [*0*, *10*] *10*)
syntax-consts
-bind -binds -Let ⇌ *Let*
translations
-Let (*-binds b bs*) *e* ⇌ *-Let b* (*-Let bs e*)
let x = a in e ⇌ *CONST Let a* (λ*x*. *e*)

end

axiomatization *undefined* :: *'a*

class *default* = **fixes** *default* :: *'a*

2.2 Fundamental rules

2.2.1 Equality

lemma *sym*: $s = t \Longrightarrow t = s$
by (*erule subst*) (*rule refl*)

lemma *ssubst*: $t = s \Longrightarrow P\ s \Longrightarrow P\ t$
by (*drule sym*) (*erule subst*)

lemma *trans*: $[\![r = s;\ s = t]\!] \Longrightarrow r = t$
by (*erule subst*)

lemma *trans-sym* [*Pure.elim?*]: $r = s \Longrightarrow t = s \Longrightarrow r = t$
by (*rule trans* [*OF - sym*])

lemma *meta-eq-to-obj-eq*:
assumes $A \equiv B$
shows $A = B$
unfolding *assms* **by** (*rule refl*)

Useful with *erule* for proving equalities from known equalities.

lemma *box-equals*: $[\![a = b;\ a = c;\ b = d]\!] \Longrightarrow c = d$
by (*iprover intro*: *sym trans*)

For calculational reasoning:

lemma *forw-subst*: $a = b \Longrightarrow P\ b \Longrightarrow P\ a$
by (*rule ssubst*)

lemma *back-subst*: $P\ a \Longrightarrow a = b \Longrightarrow P\ b$
by (*rule subst*)

2.2.2 Congruence rules for application

Similar to *AP-THM* in Gordon's HOL.

lemma *fun-cong*: $(f :: 'a \Rightarrow 'b) = g \Longrightarrow f\ x = g\ x$
by (*iprover intro*: *refl elim*: *subst*)

Similar to *AP-TERM* in Gordon's HOL and FOL's *subst-context*.

lemma *arg-cong*: $x = y \Longrightarrow f\ x = f\ y$
by (*iprover intro*: *refl elim*: *subst*)

lemma *arg-cong2*: $[\![a = b;\ c = d]\!] \Longrightarrow f\ a\ c = f\ b\ d$
by (*iprover intro*: *refl elim*: *subst*)

lemma *arg-cong3*: $[\![x = x';\ y = y';\ z = z']\!] \Longrightarrow f\ x\ y\ z = f\ x'\ y'\ z'$
by (*iprover intro*: *refl elim*: *subst*)

lemma *arg-cong4*: $[\![w = w';\ x = x';\ y = y';\ z = z']\!] \Longrightarrow f\ w\ x\ y\ z = f\ w'\ x'\ y'\ z'$
by (*iprover intro*: *refl elim*: *subst*)

lemma *cong*: $[\![f = g;\ (x::'a) = y]\!] \Longrightarrow f\ x = g\ y$
by (*iprover intro*: *refl elim*: *subst*)

ML ‹*fun cong-tac ctxt = Cong-Tac.cong-tac ctxt* @{*thm cong*}›

2.2.3 Equality of booleans – iff

lemma *iffD2*: $[\![P = Q;\ Q]\!] \Longrightarrow P$
by (*erule ssubst*)

lemma *rev-iffD2*: $[\![Q;\ P = Q]\!] \Longrightarrow P$
by (*erule iffD2*)

lemma *iffD1*: $Q = P \Longrightarrow Q \Longrightarrow P$
by (*drule sym*) (*rule iffD2*)

lemma *rev-iffD1*: $Q \Longrightarrow Q = P \Longrightarrow P$
by (*drule sym*) (*rule rev-iffD2*)

lemma *iffE*:
assumes *major*: $P = Q$
and *minor*: $[\![P \longrightarrow Q;\ Q \longrightarrow P]\!] \Longrightarrow R$
shows R
by (*iprover intro*: *minor impI major* [*THEN iffD2*] *major* [*THEN iffD1*])

2.2.4 True (1)

lemma *TrueI*: *True*
unfolding *True-def* **by** (*rule refl*)

lemma *eqTrueE*: $P = True \Longrightarrow P$
by (*erule iffD2*) (*rule TrueI*)

2.2.5 Universal quantifier (1)

lemma *spec*: $\forall x::'a.\ P\ x \Longrightarrow P\ x$
unfolding *All-def* **by** (*iprover intro*: *eqTrueE fun-cong*)

lemma *allE*:
assumes *major*: $\forall x.\ P\ x$ **and** *minor*: $P\ x \Longrightarrow R$

shows R
by (*iprover intro*: *minor major* [*THEN spec*])

lemma *all-dupE*:
assumes *major*: $\forall x.\ P\ x$ **and** *minor*: $[\![P\ x; \forall x.\ P\ x]\!] \Longrightarrow R$
shows R
by (*iprover intro*: *minor major major* [*THEN spec*])

2.2.6 False

Depends upon *spec*; it is impossible to do propositional logic before quantifiers!

lemma *FalseE*: $False \Longrightarrow P$
unfolding *False-def* **by** (*erule spec*)

lemma *False-neq-True*: $False = True \Longrightarrow P$
by (*erule eqTrueE* [*THEN FalseE*])

2.2.7 Negation

lemma *notI*:
assumes $P \Longrightarrow False$
shows $\neg P$
unfolding *not-def* **by** (*iprover intro*: *impI assms*)

lemma *False-not-True*: $False \neq True$
by (*iprover intro*: *notI elim*: *False-neq-True*)

lemma *True-not-False*: $True \neq False$
by (*iprover intro*: *notI dest*: *sym elim*: *False-neq-True*)

lemma *notE*: $[\![\neg P; P]\!] \Longrightarrow R$
unfolding *not-def*
by (*iprover intro*: *mp* [*THEN FalseE*])

2.2.8 Implication

lemma *impE*:
assumes $P \longrightarrow Q\ P\ Q \Longrightarrow R$
shows R
by (*iprover intro*: *assms mp*)

Reduces Q to $P \longrightarrow Q$, allowing substitution in P.

lemma *rev-mp*: $[\![P; P \longrightarrow Q]\!] \Longrightarrow Q$
by (*rule mp*)

lemma *contrapos-nn*:
assumes *major*: $\neg Q$
and *minor*: $P \Longrightarrow Q$

shows $\neg P$
by (*iprover intro*: *notI minor major* [*THEN notE*])

Not used at all, but we already have the other 3 combinations.

lemma *contrapos-pn*:
assumes *major*: Q
and *minor*: $P \Longrightarrow \neg Q$
shows $\neg P$
by (*iprover intro*: *notI minor major notE*)

lemma *not-sym*: $t \neq s \Longrightarrow s \neq t$
by (*erule contrapos-nn*) (*erule sym*)

lemma *eq-neq-eq-imp-neq*: $[\![x = a; a \neq b; b = y]\!] \Longrightarrow x \neq y$
by (*erule subst*, *erule ssubst*, *assumption*)

2.2.9 Disjunction (1)

lemma *disjE*:
assumes *major*: $P \vee Q$
and *minorP*: $P \Longrightarrow R$
and *minorQ*: $Q \Longrightarrow R$
shows R
by (*iprover intro*: *minorP minorQ impI*
major [*unfolded or-def*, *THEN spec*, *THEN mp*, *THEN mp*])

2.2.10 Derivation of *iffI*

In an intuitionistic version of HOL *iffI* needs to be an axiom.

lemma *iffI*:
assumes $P \Longrightarrow Q$ **and** $Q \Longrightarrow P$
shows $P = Q$
proof (*rule disjE*[*OF True-or-False*[*of P*]])
assume *1*: $P = True$
note $Q = assms(1)$[*OF eqTrueE*[*OF this*]]
from *1* **show** *?thesis*
proof (*rule ssubst*)
from *True-or-False*[*of Q*] **show** $True = Q$
proof (*rule disjE*)
assume $Q = True$
thus *?thesis* **by**(*rule sym*)
next
assume $Q = False$
with Q **have** *False* **by** (*rule rev-iffD1*)
thus *?thesis* **by** (*rule FalseE*)
qed
qed
next
assume *2*: $P = False$

thus *?thesis*
proof (*rule ssubst*)
from *True-or-False*[*of Q*] **show** *False* = *Q*
proof (*rule disjE*)
assume *Q* = *True*
from *2 assms*(*2*)[*OF eqTrueE*[*OF this*]] **have** *False* **by** (*rule iffD1*)
thus *?thesis* **by** (*rule FalseE*)
next
assume *Q* = *False*
thus *?thesis* **by**(*rule sym*)
qed
qed
qed

2.2.11 True (2)

lemma *eqTrueI*: *P* ⟹ *P* = *True*
by (*iprover intro*: *iffI TrueI*)

2.2.12 Universal quantifier (2)

lemma *allI*:
assumes ⋀*x*::′*a*. *P x*
shows ∀ *x*. *P x*
unfolding *All-def* **by** (*iprover intro*: *ext eqTrueI assms*)

2.2.13 Existential quantifier

lemma *exI*: *P x* ⟹ ∃ *x*::′*a*. *P x*
unfolding *Ex-def* **by** (*iprover intro*: *allI allE impI mp*)

lemma *exE*:
assumes *major*: ∃ *x*::′*a*. *P x*
and *minor*: ⋀*x*. *P x* ⟹ *Q*
shows *Q*
by (*rule major* [*unfolded Ex-def*, *THEN spec*, *THEN mp*]) (*iprover intro*: *impI* [*THEN allI*] *minor*)

2.2.14 Conjunction

lemma *conjI*: ⟦*P*; *Q*⟧ ⟹ *P* ∧ *Q*
unfolding *and-def* **by** (*iprover intro*: *impI* [*THEN allI*] *mp*)

lemma *conjunct1*: ⟦*P* ∧ *Q*⟧ ⟹ *P*
unfolding *and-def* **by** (*iprover intro*: *impI dest*: *spec mp*)

lemma *conjunct2*: ⟦*P* ∧ *Q*⟧ ⟹ *Q*
unfolding *and-def* **by** (*iprover intro*: *impI dest*: *spec mp*)

lemma *conjE*:

assumes *major*: $P \wedge Q$
and *minor*: $[\![P; Q]\!] \Longrightarrow R$
shows R
proof (*rule minor*)
show P **by** (*rule major* [*THEN conjunct1*])
show Q **by** (*rule major* [*THEN conjunct2*])
qed

lemma *context-conjI*:
assumes P $P \Longrightarrow Q$
shows $P \wedge Q$
by (*iprover intro*: *conjI assms*)

2.2.15 Disjunction (2)

lemma *disjI1*: $P \Longrightarrow P \vee Q$
unfolding *or-def* **by** (*iprover intro*: *allI impI mp*)

lemma *disjI2*: $Q \Longrightarrow P \vee Q$
unfolding *or-def* **by** (*iprover intro*: *allI impI mp*)

2.2.16 Classical logic

lemma *classical*:
assumes $\neg P \Longrightarrow P$
shows P
proof (*rule True-or-False* [*THEN disjE*])
show P **if** $P = True$
using *that* **by** (*iprover intro*: *eqTrueE*)
show P **if** $P = False$
proof (*intro notI assms*)
assume P
with *that* **show** *False*
by (*iprover elim*: *subst*)
qed
qed

lemmas *ccontr* = *FalseE* [*THEN classical*]

notE with premises exchanged; it discharges $\neg R$ so that it can be used to make elimination rules.

lemma *rev-notE*:
assumes *premp*: P
and *premnot*: $\neg R \Longrightarrow \neg P$
shows R
by (*iprover intro*: *ccontr notE* [*OF premnot premp*])

Double negation law.

lemma *notnotD*: $\neg\neg P \Longrightarrow P$

by (*iprover intro*: *ccontr notE*)

lemma *contrapos-pp*:
assumes *p1*: Q
and *p2*: $\neg\ P \Longrightarrow \neg\ Q$
shows P
by (*iprover intro*: *classical p1 p2 notE*)

2.2.17 Unique existence

lemma *Uniq-I* [*intro?*]:
assumes $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow y = x$
shows *Uniq P*
unfolding *Uniq-def* **by** (*iprover intro*: *assms allI impI*)

lemma *Uniq-D* [*dest?*]: $[\![Uniq\ P;\ P\ a;\ P\ b]\!] \Longrightarrow a{=}b$
unfolding *Uniq-def* **by** (*iprover dest*: *spec mp*)

lemma *ex1I*:
assumes $P\ a\ \bigwedge x.\ P\ x \Longrightarrow x = a$
shows $\exists!x.\ P\ x$
unfolding *Ex1-def* **by** (*iprover intro*: *assms exI conjI allI impI*)

Sometimes easier to use: the premises have no shared variables. Safe!

lemma *ex-ex1I*:
assumes *ex-prem*: $\exists\, x.\ P\ x$
and *eq*: $\bigwedge x\ y.\ [\![P\ x;\ P\ y]\!] \Longrightarrow x = y$
shows $\exists!x.\ P\ x$
by (*iprover intro*: *ex-prem* [*THEN exE*] *ex1I eq*)

lemma *ex1E*:
assumes *major*: $\exists!x.\ P\ x$ **and** *minor*: $\bigwedge x.\ [\![P\ x;\ \forall\, y.\ P\ y \longrightarrow y = x]\!] \Longrightarrow R$
shows R
proof (*rule major* [*unfolded Ex1-def*, *THEN exE*])
show $\bigwedge x.\ P\ x \wedge (\forall\, y.\ P\ y \longrightarrow y = x) \Longrightarrow R$
by (*iprover intro*: *minor elim*: *conjE*)
qed

lemma *ex1-implies-ex*: $\exists!x.\ P\ x \Longrightarrow \exists\, x.\ P\ x$
by (*iprover intro*: *exI elim*: *ex1E*)

2.2.18 Classical intro rules for disjunction and existential quantifiers

lemma *disjCI*:
assumes $\neg\ Q \Longrightarrow P$
shows $P \vee Q$
by (*rule classical*) (*iprover intro*: *assms disjI1 disjI2 notI elim*: *notE*)

lemma *excluded-middle*: $\neg\ P \vee P$
by (*iprover intro*: *disjCI*)

case distinction as a natural deduction rule. Note that $\neg\ P$ is the second case, not the first.

lemma *case-split* [*case-names True False*]:
assumes $P \Longrightarrow Q$ $\neg\ P \Longrightarrow Q$
shows Q
using *excluded-middle* [*of P*]
by (*iprover intro*: *assms elim*: *disjE*)

Classical implies ($\longrightarrow$) elimination.

lemma *impCE*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $\neg\ P \Longrightarrow R$ $Q \Longrightarrow R$
shows R
using *excluded-middle* [*of P*]
by (*iprover intro*: *minor major* [*THEN mp*] *elim*: *disjE*)+

This version of $\longrightarrow$ elimination works on Q before P. It works best for those cases in which P holds "almost everywhere". Can't install as default: would break old proofs.

lemma *impCE'*:
assumes *major*: $P \longrightarrow Q$
and *minor*: $Q \Longrightarrow R$ $\neg\ P \Longrightarrow R$
shows R
using *assms* **by** (*elim impCE*)

The analogous introduction rule for conjunction, above, is even constructive

lemma *context-disjE*:
assumes *major*: $P \vee Q$ **and** *minor*: $P \Longrightarrow R$ $\neg P \Longrightarrow Q \Longrightarrow R$
shows R
by (*iprover intro*: *disjE* [*OF major*] *disjE* [*OF excluded-middle*] *assms*)

Classical $\longleftrightarrow$ elimination.

lemma *iffCE*:
assumes *major*: $P = Q$
and *minor*: $[\![P;\ Q]\!] \Longrightarrow R$ $[\![\neg\ P;\ \neg\ Q]\!] \Longrightarrow R$
shows R
by (*rule major* [*THEN iffE*]) (*iprover intro*: *minor elim*: *impCE notE*)

lemma *exCI*:
assumes $\forall\, x.\ \neg\ P\ x \Longrightarrow P\ a$
shows $\exists\, x.\ P\ x$
by (*rule ccontr*) (*iprover intro*: *assms exI allI notI notE* [*of* $\exists\, x.\ P\ x$])

2.2.19 Intuitionistic Reasoning

lemma *impE'*:

assumes *1*: $P \longrightarrow Q$
and *2*: $Q \Longrightarrow R$
and *3*: $P \longrightarrow Q \Longrightarrow P$
shows R
proof −
from *3* **and** *1* **have** P **.**
with *1* **have** Q **by** (*rule impE*)
with *2* **show** R **.**
qed

lemma *allE′*:
assumes *1*: $\forall x.\ P\ x$
and *2*: $P\ x \Longrightarrow \forall x.\ P\ x \Longrightarrow Q$
shows Q
proof −
from *1* **have** $P\ x$ **by** (*rule spec*)
from *this* **and** *1* **show** Q **by** (*rule 2*)
qed

lemma *notE′*:
assumes *1*: $\neg P$
and *2*: $\neg P \Longrightarrow P$
shows R
proof −
from *2* **and** *1* **have** P **.**
with *1* **show** R **by** (*rule notE*)
qed

lemma *TrueE*: *True* $\Longrightarrow P \Longrightarrow P$ **.**
lemma *notFalseE*: $\neg$ *False* $\Longrightarrow P \Longrightarrow P$ **.**

lemmas [*Pure.elim*!] = *disjE iffE FalseE conjE exE TrueE notFalseE*
and [*Pure.intro*!] = *iffI conjI impI TrueI notI allI refl*
and [*Pure.elim 2*] = *allE notE′ impE′*
and [*Pure.intro*] = *exI disjI2 disjI1*

lemmas [*trans*] = *trans*
and [*sym*] = *sym not-sym*
and [*Pure.elim?*] = *iffD1 iffD2 impE*

2.2.20 Atomizing meta-level connectives

axiomatization where
eq-reflection: $x = y \Longrightarrow x \equiv y$ — admissible axiom

lemma *atomize-all* [*atomize*]: $(\bigwedge x.\ P\ x) \equiv$ *Trueprop* $(\forall x.\ P\ x)$
proof
assume $\bigwedge x.\ P\ x$
then show $\forall x.\ P\ x$ **..**

next
assume $\forall x.\ P\ x$
then show $\bigwedge x.\ P\ x$ **by** (*rule allE*)
qed

lemma *atomize-imp* [*atomize*]: $(A \Longrightarrow B) \equiv \mathit{Trueprop}\ (A \longrightarrow B)$
proof
assume *r*: $A \Longrightarrow B$
show $A \longrightarrow B$ **by** (*rule impI*) (*rule r*)
next
assume $A \longrightarrow B$ **and** A
then show B **by** (*rule mp*)
qed

lemma *atomize-not*: $(A \Longrightarrow \mathit{False}) \equiv \mathit{Trueprop}\ (\neg\ A)$
proof
assume *r*: $A \Longrightarrow \mathit{False}$
show $\neg\ A$ **by** (*rule notI*) (*rule r*)
next
assume $\neg\ A$ **and** A
then show *False* **by** (*rule notE*)
qed

lemma *atomize-eq* [*atomize*, *code*]: $(x \equiv y) \equiv \mathit{Trueprop}\ (x = y)$
proof
assume $x \equiv y$
show $x = y$ **by** (*unfold* ‹$x \equiv y$›) (*rule refl*)
next
assume $x = y$
then show $x \equiv y$ **by** (*rule eq-reflection*)
qed

lemma *atomize-conj* [*atomize*]: $(A\ \&\&\&\ B) \equiv \mathit{Trueprop}\ (A \wedge B)$
proof
assume *conj*: $A\ \&\&\&\ B$
show $A \wedge B$
proof (*rule conjI*)
from *conj* **show** A **by** (*rule conjunctionD1*)
from *conj* **show** B **by** (*rule conjunctionD2*)
qed
next
assume *conj*: $A \wedge B$
show $A\ \&\&\&\ B$
proof −
from *conj* **show** A **..**
from *conj* **show** B **..**
qed
qed

lemmas [*symmetric*, *rulify*] = *atomize-all atomize-imp*
and [*symmetric*, *defn*] = *atomize-all atomize-imp atomize-eq*

2.2.21 Atomizing elimination rules

lemma *atomize-exL*[*atomize-elim*]: $(\bigwedge x.\ P\ x \Longrightarrow Q) \equiv ((\exists x.\ P\ x) \Longrightarrow Q)$
by (*rule equal-intr-rule*) *iprover+*

lemma *atomize-conjL*[*atomize-elim*]: $(A \Longrightarrow B \Longrightarrow C) \equiv (A \wedge B \Longrightarrow C)$
by (*rule equal-intr-rule*) *iprover+*

lemma *atomize-disjL*[*atomize-elim*]: $((A \Longrightarrow C) \Longrightarrow (B \Longrightarrow C) \Longrightarrow C) \equiv ((A \vee B \Longrightarrow C) \Longrightarrow C)$
by (*rule equal-intr-rule*) *iprover+*

lemma *atomize-elimL*[*atomize-elim*]: $(\bigwedge B.\ (A \Longrightarrow B) \Longrightarrow B) \equiv \mathit{Trueprop}\ A$ **..**

2.3 Package setup

ML-file ‹*Tools/hologic.ML*›
ML-file ‹*Tools/rewrite-hol-proof.ML*›

setup ‹*Proofterm.set-preproc* (*Proof-Rewrite-Rules.standard-preproc Rewrite-HOL-Proof.rews*)›

2.3.1 Sledgehammer setup

Theorems blacklisted to Sledgehammer. These theorems typically produce clauses that are prolific (match too many equality or membership literals) and relate to seldom-used facts. Some duplicate other rules.

named-theorems *no-atp theorems that should be filtered out by Sledgehammer*

2.3.2 Classical Reasoner setup

lemma *imp-elim*: $P \longrightarrow Q \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by (*rule classical*) *iprover*

lemma *swap*: $\neg P \Longrightarrow (\neg R \Longrightarrow P) \Longrightarrow R$
by (*rule classical*) *iprover*

lemma *thin-refl*: $[\![x = x;\ \mathit{PROP}\ W]\!] \Longrightarrow \mathit{PROP}\ W$ **.**

ML ‹
structure Hypsubst = Hypsubst
(
val dest-eq = HOLogic.dest-eq
val dest-Trueprop = HOLogic.dest-Trueprop
val dest-imp = HOLogic.dest-imp
val eq-reflection = @{thm eq-reflection}
val rev-eq-reflection = @{thm meta-eq-to-obj-eq}

```
  val imp-intr = @{thm impI}
  val rev-mp = @{thm rev-mp}
  val subst = @{thm subst}
  val sym = @{thm sym}
  val thin-refl = @{thm thin-refl};
);
open Hypsubst;

structure Classical = Classical
(
  val imp-elim = @{thm imp-elim}
  val not-elim = @{thm notE}
  val swap = @{thm swap}
  val classical = @{thm classical}
  val sizef = Drule.size-of-thm
  val hyp-subst-tacs = [Hypsubst.hyp-subst-tac]
);

structure Basic-Classical: BASIC-CLASSICAL = Classical;
open Basic-Classical;
›

setup ‹
  (*prevent substitution on bool*)
  let
    fun non-bool-eq Const-‹HOL.eq T› = T <> Type ‹bool›
      | non-bool-eq - = false;
    fun hyp-subst-tac' ctxt =
      SUBGOAL (fn (goal, i) =>
        if Term.exists-subterm non-bool-eq goal
        then Hypsubst.hyp-subst-tac ctxt i
        else no-tac);
  in
   Context-Rules.addSWrapper (fn ctxt => fn tac => hyp-subst-tac' ctxt ORELSE'
tac)
  end
›

declare iffI [intro!]
  and notI [intro!]
  and impI [intro!]
  and disjCI [intro!]
  and conjI [intro!]
  and TrueI [intro!]
  and refl [intro!]

declare iffCE [elim!]
  and FalseE [elim!]
  and impCE [elim!]
```

and *disjE* [*elim*!]
and *conjE* [*elim*!]

declare *ex-ex1I* [*intro*!]
and *allI* [*intro*!]
and *exI* [*intro*]

declare *exE* [*elim*!]
allE [*elim*]

ML ‹*val HOL-cs = claset-of* ***context***›

lemma *contrapos-np*: $\neg\ Q \Longrightarrow (\neg\ P \Longrightarrow Q) \Longrightarrow P$
by (*erule swap*)

declare *ex-ex1I* [*rule del*, *intro*! *2*]
and *ex1I* [*intro*]

declare *ext* [*intro*]

lemmas [*intro?*] = *ext*
and [*elim?*] = *ex1-implies-ex*

Better than *ex1E* for classical reasoner: needs no quantifier duplication!

lemma *alt-ex1E* [*elim*!]:
assumes *major*: $\exists!x.\ P\ x$
and *minor*: $\bigwedge x.\ [\![P\ x;\ \forall y\ y'.\ P\ y \wedge P\ y' \longrightarrow y = y']\!] \Longrightarrow R$
shows *R*
proof (*rule ex1E* [*OF major minor*])
show $\forall y\ y'.\ P\ y \wedge P\ y' \longrightarrow y = y'$ **if** $P\ x$ **and** §: $\forall y.\ P\ y \longrightarrow y = x$ **for** *x*
using ‹*P x*› § § **by** *fast*
qed *assumption*

And again using Uniq

lemma *alt-ex1E'*:
assumes $\exists!x.\ P\ x\ \bigwedge x.\ [\![P\ x;\ \exists_{\leq 1}x.\ P\ x]\!] \Longrightarrow R$
shows *R*
using *assms* **unfolding** *Uniq-def* **by** *fast*

lemma *ex1-iff-ex-Uniq*: $(\exists!x.\ P\ x) \longleftrightarrow (\exists x.\ P\ x) \wedge (\exists_{\leq 1}x.\ P\ x)$
unfolding *Uniq-def* **by** *fast*

ML ‹
structure Blast = Blast
(
structure Classical = Classical
val Trueprop-const = dest-Const ***Const*** ‹*Trueprop*›
val equality-name = ***const-name*** ‹*HOL.eq*›

```
  val not-name = const-name ‹Not›
  val notE = @{thm notE}
  val ccontr = @{thm ccontr}
  val hyp-subst-tac = Hypsubst.blast-hyp-subst-tac
 );
 val blast-tac = Blast.blast-tac;
›
```

2.3.3 THE: definite description operator

lemma *the-equality* [*intro*]:
 assumes $P\ a$
  and $\bigwedge x.\ P\ x \Longrightarrow x = a$
 shows $(THE\ x.\ P\ x) = a$
 by (*blast intro*: *assms trans* [*OF arg-cong* [**where** *f*=*The*] *the-eq-trivial*])

lemma *theI*:
 assumes $P\ a$
  and $\bigwedge x.\ P\ x \Longrightarrow x = a$
 shows $P\ (THE\ x.\ P\ x)$
 by (*iprover intro*: *assms the-equality* [*THEN ssubst*])

lemma *theI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\ x.\ P\ x)$
 by (*blast intro*: *theI*)

Easier to apply than *theI*: only one occurrence of *P*.

lemma *theI2*:
 assumes $P\ a\ \bigwedge x.\ P\ x \Longrightarrow x = a\ \bigwedge x.\ P\ x \Longrightarrow Q\ x$
 shows $Q\ (THE\ x.\ P\ x)$
 by (*iprover intro*: *assms theI*)

lemma *the1I2*:
 assumes $\exists!x.\ P\ x\ \bigwedge x.\ P\ x \Longrightarrow Q\ x$
 shows $Q\ (THE\ x.\ P\ x)$
 by (*iprover intro*: *assms*(*2*) *theI2*[**where** *P*=*P* **and** *Q*=*Q*] *ex1E*[*OF assms*(*1*)] *elim*: *allE impE*)

lemma *the1-equality* [*elim?*]: $[\![\exists!x.\ P\ x;\ P\ a]\!] \Longrightarrow (THE\ x.\ P\ x) = a$
 by *blast*

lemma *the1-equality'*: $[\![\exists_{\leq 1}x.\ P\ x;\ P\ a]\!] \Longrightarrow (THE\ x.\ P\ x) = a$
 unfolding *Uniq-def* **by** *blast*

lemma *the-sym-eq-trivial*: $(THE\ y.\ x = y) = x$
 by *blast*

2.3.4 Simplifier

lemma *eta-contract-eq*: $(\lambda s.\ f\ s) = f$ **..**

lemma *subst-all*:
  ‹($\bigwedge x.\ x = a \Longrightarrow PROP\ P\ x$) $\equiv PROP\ P\ a$›
  ‹($\bigwedge x.\ a = x \Longrightarrow PROP\ P\ x$) $\equiv PROP\ P\ a$›
proof −
  show ‹($\bigwedge x.\ x = a \Longrightarrow PROP\ P\ x$) $\equiv PROP\ P\ a$›
  proof (*rule equal-intr-rule*)
    assume ∗: ‹$\bigwedge x.\ x = a \Longrightarrow PROP\ P\ x$›
    show ‹$PROP\ P\ a$›
      by (*rule* ∗) (*rule refl*)
  next
    fix x
    assume ‹$PROP\ P\ a$› **and** ‹$x = a$›
    from ‹$x = a$› **have** ‹$x \equiv a$›
      by (*rule eq-reflection*)
    with ‹$PROP\ P\ a$› **show** ‹$PROP\ P\ x$›
      by *simp*
  qed
  show ‹($\bigwedge x.\ a = x \Longrightarrow PROP\ P\ x$) $\equiv PROP\ P\ a$›
  proof (*rule equal-intr-rule*)
    assume ∗: ‹$\bigwedge x.\ a = x \Longrightarrow PROP\ P\ x$›
    show ‹$PROP\ P\ a$›
      by (*rule* ∗) (*rule refl*)
  next
    fix x
    assume ‹$PROP\ P\ a$› **and** ‹$a = x$›
    from ‹$a = x$› **have** ‹$a \equiv x$›
      by (*rule eq-reflection*)
    with ‹$PROP\ P\ a$› **show** ‹$PROP\ P\ x$›
      by *simp*
  qed
qed

lemma *simp-thms*:
  shows *not-not*: $(\neg\ \neg\ P) = P$
  and *Not-eq-iff*: $((\neg\ P) = (\neg\ Q)) = (P = Q)$
  and
    $(P \neq Q) = (P = (\neg\ Q))$
    $(P \vee \neg P) = True \quad (\neg\ P \vee P) = True$
    $(x = x) = True$
  and *not-True-eq-False* [*code*]: $(\neg\ True) = False$
  and *not-False-eq-True* [*code*]: $(\neg\ False) = True$
  and
    $(\neg\ P) \neq P \ \ P \neq (\neg\ P)$
    $(True = P) = P$
  and *eq-True*: $(P = True) = P$
  and $(False = P) = (\neg\ P)$
  and *eq-False*: $(P = False) = (\neg\ P)$
  and
    $(True \longrightarrow P) = P \ \ (False \longrightarrow P) = True$

$(P \longrightarrow \mathit{True}) = \mathit{True}$ $(P \longrightarrow P) = \mathit{True}$
$(P \longrightarrow \mathit{False}) = (\neg\ P)$ $(P \longrightarrow \neg\ P) = (\neg\ P)$
$(P \wedge \mathit{True}) = P$ $(\mathit{True} \wedge P) = P$
$(P \wedge \mathit{False}) = \mathit{False}$ $(\mathit{False} \wedge P) = \mathit{False}$
$(P \wedge P) = P$ $(P \wedge (P \wedge Q)) = (P \wedge Q)$
$(P \wedge \neg\ P) = \mathit{False}$ $(\neg\ P \wedge P) = \mathit{False}$
$(P \vee \mathit{True}) = \mathit{True}$ $(\mathit{True} \vee P) = \mathit{True}$
$(P \vee \mathit{False}) = P$ $(\mathit{False} \vee P) = P$
$(P \vee P) = P$ $(P \vee (P \vee Q)) = (P \vee Q)$ **and**
$(\forall x.\ P) = P$ $(\exists x.\ P) = P$ $\exists x.\ x = t$ $\exists x.\ t = x$
and
$\bigwedge P.\ (\exists x.\ x = t \wedge P\ x) = P\ t$
$\bigwedge P.\ (\exists x.\ t = x \wedge P\ x) = P\ t$
$\bigwedge P.\ (\forall x.\ x = t \longrightarrow P\ x) = P\ t$
$\bigwedge P.\ (\forall x.\ t = x \longrightarrow P\ x) = P\ t$
$(\forall x.\ x \neq t) = \mathit{False}$ $(\forall x.\ t \neq x) = \mathit{False}$
by (*blast*, *blast*, *blast*, *blast*, *blast*, *iprover*+)

lemma *disj-absorb*: $A \vee A \longleftrightarrow A$
by *blast*

lemma *disj-left-absorb*: $A \vee (A \vee B) \longleftrightarrow A \vee B$
by *blast*

lemma *conj-absorb*: $A \wedge A \longleftrightarrow A$
by *blast*

lemma *conj-left-absorb*: $A \wedge (A \wedge B) \longleftrightarrow A \wedge B$
by *blast*

lemma *eq-ac*:
shows *eq-commute*: $a = b \longleftrightarrow b = a$
and *iff-left-commute*: $(P \longleftrightarrow (Q \longleftrightarrow R)) \longleftrightarrow (Q \longleftrightarrow (P \longleftrightarrow R))$
and *iff-assoc*: $((P \longleftrightarrow Q) \longleftrightarrow R) \longleftrightarrow (P \longleftrightarrow (Q \longleftrightarrow R))$
by (*iprover*, *blast*+)

lemma *neq-commute*: $a \neq b \longleftrightarrow b \neq a$ **by** *iprover*

lemma *conj-comms*:
shows *conj-commute*: $P \wedge Q \longleftrightarrow Q \wedge P$
and *conj-left-commute*: $P \wedge (Q \wedge R) \longleftrightarrow Q \wedge (P \wedge R)$ **by** *iprover*+
lemma *conj-assoc*: $(P \wedge Q) \wedge R \longleftrightarrow P \wedge (Q \wedge R)$ **by** *iprover*

lemmas *conj-ac* = *conj-commute conj-left-commute conj-assoc*

lemma *disj-comms*:
shows *disj-commute*: $P \vee Q \longleftrightarrow Q \vee P$
and *disj-left-commute*: $P \vee (Q \vee R) \longleftrightarrow Q \vee (P \vee R)$ **by** *iprover*+
lemma *disj-assoc*: $(P \vee Q) \vee R \longleftrightarrow P \vee (Q \vee R)$ **by** *iprover*

lemmas *disj-ac* = *disj-commute disj-left-commute disj-assoc*

lemma *conj-disj-distribL*: $P \wedge (Q \vee R) \longleftrightarrow P \wedge Q \vee P \wedge R$ **by** *iprover*
lemma *conj-disj-distribR*: $(P \vee Q) \wedge R \longleftrightarrow P \wedge R \vee Q \wedge R$ **by** *iprover*

lemma *disj-conj-distribL*: $P \vee (Q \wedge R) \longleftrightarrow (P \vee Q) \wedge (P \vee R)$ **by** *iprover*
lemma *disj-conj-distribR*: $(P \wedge Q) \vee R \longleftrightarrow (P \vee R) \wedge (Q \vee R)$ **by** *iprover*

lemma *imp-conjR*: $(P \longrightarrow (Q \wedge R)) = ((P \longrightarrow Q) \wedge (P \longrightarrow R))$ **by** *iprover*
lemma *imp-conjL*: $((P \wedge Q) \longrightarrow R) = (P \longrightarrow (Q \longrightarrow R))$ **by** *iprover*
lemma *imp-disjL*: $((P \vee Q) \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ **by** *iprover*

These two are specialized, but *imp-disj-not1* is useful in *Auth/ Yahalom.*

lemma *imp-disj-not1*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg Q \longrightarrow P \longrightarrow R)$ **by** *blast*
lemma *imp-disj-not2*: $(P \longrightarrow Q \vee R) \longleftrightarrow (\neg R \longrightarrow P \longrightarrow Q)$ **by** *blast*

lemma *imp-disj1*: $((P \longrightarrow Q) \vee R) \longleftrightarrow (P \longrightarrow Q \vee R)$ **by** *blast*
lemma *imp-disj2*: $(Q \vee (P \longrightarrow R)) \longleftrightarrow (P \longrightarrow Q \vee R)$ **by** *blast*

lemma *imp-cong*: $(P = P') \Longrightarrow (P' \Longrightarrow (Q = Q')) \Longrightarrow ((P \longrightarrow Q) \longleftrightarrow (P' \longrightarrow Q'))$
by *iprover*

lemma *de-Morgan-disj*: $\neg (P \vee Q) \longleftrightarrow \neg P \wedge \neg Q$ **by** *iprover*
lemma *de-Morgan-conj*: $\neg (P \wedge Q) \longleftrightarrow \neg P \vee \neg Q$ **by** *blast*
lemma *not-imp*: $\neg (P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q$ **by** *blast*
lemma *not-iff*: $P \neq Q \longleftrightarrow (P \longleftrightarrow \neg Q)$ **by** *blast*
lemma *disj-not1*: $\neg P \vee Q \longleftrightarrow (P \longrightarrow Q)$ **by** *blast*
lemma *disj-not2*: $P \vee \neg Q \longleftrightarrow (Q \longrightarrow P)$ **by** *blast* — changes orientation :-(
lemma *imp-conv-disj*: $(P \longrightarrow Q) \longleftrightarrow (\neg P) \vee Q$ **by** *blast*
lemma *disj-imp*: $P \vee Q \longleftrightarrow \neg P \longrightarrow Q$ **by** *blast*

lemma *iff-conv-conj-imp*: $(P \longleftrightarrow Q) \longleftrightarrow (P \longrightarrow Q) \wedge (Q \longrightarrow P)$ **by** *iprover*

lemma *cases-simp*: $(P \longrightarrow Q) \wedge (\neg P \longrightarrow Q) \longleftrightarrow Q$
— Avoids duplication of subgoals after *if-split*, when the true and false
— cases boil down to the same thing.
by *blast*

lemma *not-all*: $\neg (\forall x.\ P\ x) \longleftrightarrow (\exists x.\ \neg P\ x)$ **by** *blast*
lemma *imp-all*: $((\forall x.\ P\ x) \longrightarrow Q) \longleftrightarrow (\exists x.\ P\ x \longrightarrow Q)$ **by** *blast*
lemma *not-ex*: $\neg (\exists x.\ P\ x) \longleftrightarrow (\forall x.\ \neg P\ x)$ **by** *iprover*
lemma *imp-ex*: $((\exists x.\ P\ x) \longrightarrow Q) \longleftrightarrow (\forall x.\ P\ x \longrightarrow Q)$ **by** *iprover*
lemma *all-not-ex*: $(\forall x.\ P\ x) \longleftrightarrow \neg (\exists x.\ \neg P\ x)$ **by** *blast*

declare *All-def* [*no-atp*]

lemma *ex-disj-distrib*: $(\exists\, x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists\, x.\ P\ x) \vee (\exists\, x.\ Q\ x)$ **by** *iprover*
lemma *all-conj-distrib*: $(\forall\, x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall\, x.\ P\ x) \wedge (\forall\, x.\ Q\ x)$ **by** *iprover*
lemma *all-imp-conj-distrib*: $(\forall\, x.\ P\ x \longrightarrow Q\ x \wedge R\ x) \longleftrightarrow (\forall\, x.\ P\ x \longrightarrow Q\ x) \wedge (\forall\, x.\ P\ x \longrightarrow R\ x)$
by *iprover*

The $\wedge$ congruence rule: not included by default! May slow rewrite proofs down by as much as 50%

lemma *conj-cong*: $P = P' \Longrightarrow (P' \Longrightarrow Q = Q') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
by *iprover*

lemma *rev-conj-cong*: $Q = Q' \Longrightarrow (Q' \Longrightarrow P = P') \Longrightarrow (P \wedge Q) = (P' \wedge Q')$
by *iprover*

The | congruence rule: not included by default!

lemma *disj-cong*: $P = P' \Longrightarrow (\neg\ P' \Longrightarrow Q = Q') \Longrightarrow (P \vee Q) = (P' \vee Q')$
by *blast*

if-then-else rules

lemma *if-True* [*code*]: (*if True then x else y*) = *x*
unfolding *If-def* **by** *blast*

lemma *if-False* [*code*]: (*if False then x else y*) = *y*
unfolding *If-def* **by** *blast*

lemma *if-P*: $P \Longrightarrow$ (*if P then x else y*) = *x*
unfolding *If-def* **by** *blast*

lemma *if-not-P*: $\neg\ P \Longrightarrow$ (*if P then x else y*) = *y*
unfolding *If-def* **by** *blast*

lemma *if-split*: *P* (*if Q then x else y*) = $((Q \longrightarrow P\ x) \wedge (\neg\ Q \longrightarrow P\ y))$
proof (*rule case-split* [*of Q*])
show *?thesis* **if** *Q*
using *that* **by** (*simplesubst if-P*) *blast+*
show *?thesis* **if** $\neg$ *Q*
using *that* **by** (*simplesubst if-not-P*) *blast+*
qed

lemma *if-split-asm*: *P* (*if Q then x else y*) = $(\neg\ ((Q \wedge \neg\ P\ x) \vee (\neg\ Q \wedge \neg\ P\ y)))$
by (*simplesubst if-split*) *blast*

lemmas *if-splits* [*no-atp*] = *if-split if-split-asm*

lemma *if-cancel*: (*if c then x else x*) = *x*
by (*simplesubst if-split*) *blast*

lemma *if-eq-cancel*: (*if x = y then y else x*) = *x*

by (*simplesubst if-split*) *blast*

lemma *if-bool-eq-conj*: (*if P then Q else R*) = ((*P* ⟶ *Q*) ∧ (¬ *P* ⟶ *R*))
— This form is useful for expanding *if*s on the RIGHT of the ⟹ symbol.
by (*rule if-split*)

lemma *if-bool-eq-disj*: (*if P then Q else R*) = ((*P* ∧ *Q*) ∨ (¬ *P* ∧ *R*))
— And this form is useful for expanding *if*s on the LEFT.
by (*simplesubst if-split*) *blast*

lemma *Eq-TrueI*: *P* ⟹ *P* ≡ *True* **unfolding** *atomize-eq* **by** *iprover*
lemma *Eq-FalseI*: ¬ *P* ⟹ *P* ≡ *False* **unfolding** *atomize-eq* **by** *iprover*

let rules for simproc

lemma *Let-folded*: *f x* ≡ *g x* ⟹ *Let x f* ≡ *Let x g*
by (*unfold Let-def*)

lemma *Let-unfold*: *f x* ≡ *g* ⟹ *Let x f* ≡ *g*
by (*unfold Let-def*)

The following copy of the implication operator is useful for fine-tuning congruence rules. It instructs the simplifier to simplify its premise.

definition *simp-implies* :: *prop* ⇒ *prop* ⇒ *prop* (**infixr** ‹=*simp*=>› *1*)
where *simp-implies* ≡ (⟹)

lemma *simp-impliesI*:
assumes *PQ*: (*PROP P* ⟹ *PROP Q*)
shows *PROP P* =*simp*=> *PROP Q*
unfolding *simp-implies-def*
by (*iprover intro*: *PQ*)

lemma *simp-impliesE*:
assumes *PQ*: *PROP P* =*simp*=> *PROP Q*
and *P*: *PROP P*
and *QR*: *PROP Q* ⟹ *PROP R*
shows *PROP R*
by (*iprover intro*: *QR P PQ* [*unfolded simp-implies-def*])

lemma *simp-implies-cong*:
assumes *PP'* :*PROP P* ≡ *PROP P'*
and *P'QQ'*: *PROP P'* ⟹ (*PROP Q* ≡ *PROP Q'*)
shows (*PROP P* =*simp*=> *PROP Q*) ≡ (*PROP P'* =*simp*=> *PROP Q'*)
unfolding *simp-implies-def*
proof (*rule equal-intr-rule*)
assume *PQ*: *PROP P* ⟹ *PROP Q*
and *P'*: *PROP P'*
from *PP'* [*symmetric*] **and** *P'* **have** *PROP P*
by (*rule equal-elim-rule1*)

then have *PROP Q* **by** (*rule PQ*)
with *P'QQ'* [*OF P'*] **show** *PROP Q'* **by** (*rule equal-elim-rule1*)
next
assume *P'Q'*: *PROP P'* ⟹ *PROP Q'*
and *P*: *PROP P*
from *PP'* **and** *P* **have** *P'*: *PROP P'* **by** (*rule equal-elim-rule1*)
then have *PROP Q'* **by** (*rule P'Q'*)
with *P'QQ'* [*OF P'*, *symmetric*] **show** *PROP Q*
by (*rule equal-elim-rule1*)
qed

lemma *uncurry*:
assumes *P* ⟶ *Q* ⟶ *R*
shows *P* ∧ *Q* ⟶ *R*
using *assms* **by** *blast*

lemma *iff-allI*:
assumes ⋀*x*. *P x* = *Q x*
shows (∀ *x*. *P x*) = (∀ *x*. *Q x*)
using *assms* **by** *blast*

lemma *iff-exI*:
assumes ⋀*x*. *P x* = *Q x*
shows (∃ *x*. *P x*) = (∃ *x*. *Q x*)
using *assms* **by** *blast*

lemma *all-comm*: (∀ *x y*. *P x y*) = (∀ *y x*. *P x y*)
by *blast*

lemma *ex-comm*: (∃ *x y*. *P x y*) = (∃ *y x*. *P x y*)
by *blast*

ML-file ‹*Tools/simpdata.ML*›
ML ‹*open Simpdata*›

setup ‹
Simplifier.map-theory-simpset (*put-simpset HOL-basic-ss*) #>
Simplifier.method-setup Splitter.split-modifiers
›

simproc-setup *defined-Ex* (∃ *x*. *P x*) = ‹*K Quantifier1.rearrange-Ex*›
simproc-setup *defined-All* (∀ *x*. *P x*) = ‹*K Quantifier1.rearrange-All*›
simproc-setup *defined-all*(⋀*x*. *PROP P x*) = ‹*K Quantifier1.rearrange-all*›

Simproc for proving (*y* = *x*) ≡ *False* from premise ¬ (*x* = *y*):

simproc-setup *neq* (*x* = *y*) = ‹
let
val neq-to-EQ-False = @{*thm not-sym*} *RS* @{*thm Eq-FalseI*};
fun is-neq eq lhs rhs thm =

```
    (case Thm.prop-of thm of
      - $ (Not $ (eq' $ l' $ r')) =>
        Not = HOLogic.Not andalso eq' = eq andalso
        r' aconv lhs andalso l' aconv rhs
    | - => false);
  fun proc ss ct =
    (case Thm.term-of ct of
      eq $ lhs $ rhs =>
        (case find-first (is-neq eq lhs rhs) (Simplifier.prems-of ss) of
          SOME thm => SOME (thm RS neq-to-EQ-False)
        | NONE => NONE)
     | - => NONE);
 in K proc end
›

simproc-setup let-simp (Let x f) = ‹
 let
  fun count-loose (Bound i) k = if i >= k then 1 else 0
    | count-loose (s $ t) k = count-loose s k + count-loose t k
    | count-loose (Abs (-, -, t)) k = count-loose  t (k + 1)
    | count-loose - - = 0;
  fun is-trivial-let Const- ‹Let - - for x t› =
    (case t of
      Abs (-, -, t') => count-loose t' 0 <= 1
    | - => true);
 in
  K (fn ctxt => fn ct =>
    if is-trivial-let (Thm.term-of ct)
    then SOME @{thm Let-def} (*no or one ocurrence of bound variable*)
    else
      let (*Norbert Schirmer's case*)
        val t = Thm.term-of ct;
        val (t', ctxt') = yield-singleton (Variable.import-terms false) t ctxt;
      in
        Option.map (hd o Variable.export ctxt' ctxt o single)
          (case t' of Const- ‹Let - - for x f› => (* x and f are already in normal
form *)
            if is-Free x orelse is-Bound x orelse is-Const x
            then SOME @{thm Let-def}
            else
              let
                val n = case f of (Abs (x, -, -)) => x | - => x;
                val cx = Thm.cterm-of ctxt x;
                val xT = Thm.typ-of-cterm cx;
                val cf = Thm.cterm-of ctxt f;
                val fx-g = Simplifier.rewrite ctxt (Thm.apply cf cx);
                val (- $ - $ g) = Thm.prop-of fx-g;
                val g' = abstract-over (x, g);
                val abs-g'= Abs (n, xT, g');
```

```
            in
              if g aconv g' then
                let
                  val rl =
                  infer-instantiate ctxt [((f, 0), cf), ((x, 0), cx)] @{thm Let-unfold};
                  in SOME (rl OF [fx-g]) end
                  else if (Envir.beta-eta-contract f) aconv (Envir.beta-eta-contract
abs-g')
              then NONE (*avoid identity conversion*)
              else
                let
                  val g'x = abs-g' $ x;
                        val g-g'x = Thm.symmetric (Thm.beta-conversion false
(Thm.cterm-of ctxt g'x));
                  val rl =
                    @{thm Let-folded} |> infer-instantiate ctxt
                      [((f, 0), Thm.cterm-of ctxt f),
                       ((x, 0), cx),
                       ((g, 0), Thm.cterm-of ctxt abs-g')];
                in SOME (rl OF [Thm.transitive fx-g g-g'x]) end
            end
          | - => NONE)
      end)
  end
›

lemma True-implies-equals: (True ⟹ PROP P) ≡ PROP P
proof
  assume True ⟹ PROP P
  from this [OF TrueI] show PROP P .
next
  assume PROP P
  then show PROP P .
qed

lemma implies-True-equals: (PROP P ⟹ True) ≡ Trueprop True
  by standard (intro TrueI)

lemma False-implies-equals: (False ⟹ P) ≡ Trueprop True
  by standard simp-all


lemma implies-False-swap:
  (False ⟹ PROP P ⟹ PROP Q) ≡ (PROP P ⟹ False ⟹ PROP Q)
  by (rule swap-prems-eq)

simproc-setup eliminate-false-implies (False ⟹ PROP P) = ‹
  let
    fun conv n =
```

```
    if n > 1 then
      Conv.rewr-conv @{thm Pure.swap-prems-eq}
      then-conv Conv.arg-conv (conv (n - 1))
      then-conv Conv.rewr-conv @{thm HOL.implies-True-equals}
    else
      Conv.rewr-conv @{thm HOL.False-implies-equals}
in
  fn - => fn - => fn ct =>
    let
      val t = Thm.term-of ct
      val n = length (Logic.strip-imp-prems t)
    in
      (case Logic.strip-imp-concl t of
        Const- ‹Trueprop for -› => SOME (conv n ct)
      | - => NONE)
    end
end
›
```

lemma *ex-simps*:
$\bigwedge P\ Q.\ (\exists x.\ P\ x \wedge Q) = ((\exists x.\ P\ x) \wedge Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \wedge Q\ x) = (P \wedge (\exists x.\ Q\ x))$
$\bigwedge P\ Q.\ (\exists x.\ P\ x \vee Q) = ((\exists x.\ P\ x) \vee Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \vee Q\ x) = (P \vee (\exists x.\ Q\ x))$
$\bigwedge P\ Q.\ (\exists x.\ P\ x \longrightarrow Q) = ((\forall x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (\exists x.\ P \longrightarrow Q\ x) = (P \longrightarrow (\exists x.\ Q\ x))$
— Miniscoping: pushing in existential quantifiers.
by (*iprover* | *blast*)+

lemma *all-simps*:
$\bigwedge P\ Q.\ (\forall x.\ P\ x \wedge Q) = ((\forall x.\ P\ x) \wedge Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \wedge Q\ x) = (P \wedge (\forall x.\ Q\ x))$
$\bigwedge P\ Q.\ (\forall x.\ P\ x \vee Q) = ((\forall x.\ P\ x) \vee Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \vee Q\ x) = (P \vee (\forall x.\ Q\ x))$
$\bigwedge P\ Q.\ (\forall x.\ P\ x \longrightarrow Q) = ((\exists x.\ P\ x) \longrightarrow Q)$
$\bigwedge P\ Q.\ (\forall x.\ P \longrightarrow Q\ x) = (P \longrightarrow (\forall x.\ Q\ x))$
— Miniscoping: pushing in universal quantifiers.
by (*iprover* | *blast*)+

lemmas [*simp*] =
triv-forall-equality — prunes params
True-implies-equals implies-True-equals — prune *True* in asms
False-implies-equals — prune *False* in asms
if-True
if-False
if-cancel
if-eq-cancel
imp-disjL — In general it seems wrong to add distributive laws by default: they

might cause exponential blow-up. But *imp-disjL* has been in for a while and cannot be removed without affecting existing proofs. Moreover, rewriting by $(P \vee Q \longrightarrow R) = ((P \longrightarrow R) \wedge (Q \longrightarrow R))$ might be justified on the grounds that it allows simplification of R in the two cases.

conj-assoc
disj-assoc
de-Morgan-conj
de-Morgan-disj
imp-disj1
imp-disj2
not-imp
disj-not1
not-all
not-ex
cases-simp
the-eq-trivial
the-sym-eq-trivial
ex-simps
all-simps
simp-thms
subst-all

lemmas [*cong*] = *imp-cong simp-implies-cong*
lemmas [*split*] = *if-split*

ML ‹*val HOL-ss = Simplifier.simpset-of* ***context***›

Simplifies x assuming c and y assuming $\neg\ c$.

lemma *if-cong*:
assumes $b = c$
and $c \Longrightarrow x = u$
and $\neg\ c \Longrightarrow y = v$
shows (*if b then x else y*) = (*if c then u else v*)
using *assms* **by** *simp*

Prevents simplification of x and y: faster and allows the execution of functional programs.

lemma *if-weak-cong* [*cong*]:
assumes $b = c$
shows (*if b then x else y*) = (*if c then x else y*)
using *assms* **by** (*rule arg-cong*)

Prevents simplification of t: much faster

lemma *let-weak-cong*:
assumes $a = b$
shows (*let x = a in t x*) = (*let x = b in t x*)
using *assms* **by** (*rule arg-cong*)

To tidy up the result of a simproc. Only the RHS will be simplified.

lemma *eq-cong2*:
assumes $u = u'$
shows $(t \equiv u) \equiv (t \equiv u')$
using *assms* **by** *simp*

lemma *if-distrib*: $f\ (\mathit{if}\ c\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ f\ x\ \mathit{else}\ f\ y)$
by *simp*

lemma *if-distribR*: $(\mathit{if}\ b\ \mathit{then}\ f\ \mathit{else}\ g)\ x = (\mathit{if}\ b\ \mathit{then}\ f\ x\ \mathit{else}\ g\ x)$
by *simp*

lemma *all-if-distrib*: $(\forall x.\ \mathit{if}\ x = a\ \mathit{then}\ P\ x\ \mathit{else}\ Q\ x) \longleftrightarrow P\ a \wedge (\forall x.\ x \neq a \longrightarrow Q\ x)$
by *auto*

lemma *ex-if-distrib*: $(\exists x.\ \mathit{if}\ x = a\ \mathit{then}\ P\ x\ \mathit{else}\ Q\ x) \longleftrightarrow P\ a \vee (\exists x.\ x \neq a \wedge Q\ x)$
by *auto*

lemma *if-if-eq-conj*: $(\mathit{if}\ P\ \mathit{then}\ \mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y\ \mathit{else}\ y) = (\mathit{if}\ P \wedge Q\ \mathit{then}\ x\ \mathit{else}\ y)$
by *simp*

As a simplification rule, it replaces all function equalities by first-order equalities.

lemma *fun-eq-iff*: $f = g \longleftrightarrow (\forall x.\ f\ x = g\ x)$
by *auto*

2.3.5 Generic cases and induction

Rule projections:

ML ‹
structure Project-Rule = Project-Rule
(
  val conjunct1 = @{thm conjunct1}
  val conjunct2 = @{thm conjunct2}
  val mp = @{thm mp}
);
›

context
begin

qualified definition *induct-forall* $P \equiv \forall x.\ P\ x$
qualified definition *induct-implies* $A\ B \equiv A \longrightarrow B$
qualified definition *induct-equal* $x\ y \equiv x = y$
qualified definition *induct-conj* $A\ B \equiv A \wedge B$
qualified definition *induct-true* $\equiv \mathit{True}$
qualified definition *induct-false* $\equiv \mathit{False}$

lemma *induct-forall-eq*: $(\bigwedge x.\ P\ x) \equiv \mathit{Trueprop}\ (\mathit{induct\text{-}forall}\ (\lambda x.\ P\ x))$
by (*unfold atomize-all induct-forall-def*)

lemma *induct-implies-eq*: $(A \Longrightarrow B) \equiv \mathit{Trueprop}\ (\mathit{induct\text{-}implies}\ A\ B)$
by (*unfold atomize-imp induct-implies-def*)

lemma *induct-equal-eq*: $(x \equiv y) \equiv \mathit{Trueprop}\ (\mathit{induct\text{-}equal}\ x\ y)$
by (*unfold atomize-eq induct-equal-def*)

lemma *induct-conj-eq*: $(A\ \&\&\&\ B) \equiv \mathit{Trueprop}\ (\mathit{induct\text{-}conj}\ A\ B)$
by (*unfold atomize-conj induct-conj-def*)

lemmas *induct-atomize'* = *induct-forall-eq induct-implies-eq induct-conj-eq*
lemmas *induct-atomize* = *induct-atomize' induct-equal-eq*
lemmas *induct-rulify'* [*symmetric*] = *induct-atomize'*
lemmas *induct-rulify* [*symmetric*] = *induct-atomize*
lemmas *induct-rulify-fallback* =
induct-forall-def induct-implies-def induct-equal-def induct-conj-def
induct-true-def induct-false-def

lemma *induct-forall-conj*: *induct-forall* $(\lambda x.$ *induct-conj* $(A\ x)\ (B\ x)) =$
induct-conj (*induct-forall A*) (*induct-forall B*)
by (*unfold induct-forall-def induct-conj-def*) *iprover*

lemma *induct-implies-conj*: *induct-implies C* (*induct-conj A B*) =
induct-conj (*induct-implies C A*) (*induct-implies C B*)
by (*unfold induct-implies-def induct-conj-def*) *iprover*

lemma *induct-conj-curry*: $(\mathit{induct\text{-}conj}\ A\ B \Longrightarrow \mathit{PROP}\ C) \equiv (A \Longrightarrow B \Longrightarrow \mathit{PROP}\ C)$
proof
assume *r*: *induct-conj A B* $\Longrightarrow$ *PROP C*
assume *ab*: *A B*
show *PROP C* **by** (*rule r*) (*simp add*: *induct-conj-def ab*)
next
assume *r*: $A \Longrightarrow B \Longrightarrow \mathit{PROP}\ C$
assume *ab*: *induct-conj A B*
show *PROP C* **by** (*rule r*) (*simp-all add*: *ab* [*unfolded induct-conj-def*])
qed

lemmas *induct-conj* = *induct-forall-conj induct-implies-conj induct-conj-curry*

lemma *induct-trueI*: *induct-true*
by (*simp add*: *induct-true-def*)

Method setup.

ML-file ‹~~/*src*/*Tools*/*induct.ML*›
ML ‹

```
structure Induct = Induct
(
  val cases-default = @{thm case-split}
  val atomize = @{thms induct-atomize}
  val rulify = @{thms induct-rulify'}
  val rulify-fallback = @{thms induct-rulify-fallback}
  val equal-def = @{thm induct-equal-def}
  fun dest-def Const- ‹induct-equal - for t u› = SOME (t, u)
    | dest-def - = NONE
  fun trivial-tac ctxt = match-tac ctxt @{thms induct-trueI}
)
›
```

ML-file ‹~~/src/Tools/induction.ML›

```
simproc-setup passive swap-induct-false (induct-false ⟹ PROP P ⟹ PROP
Q) =
  ‹fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      - $ (P as - $ Const- ‹induct-false›) $ (- $ Q $ -) =>
        if P <> Q then SOME Drule.swap-prems-eq else NONE
    | - => NONE)›

simproc-setup passive induct-equal-conj-curry (induct-conj P Q ⟹ PROP R)
=
  ‹fn - => fn - => fn ct =>
    (case Thm.term-of ct of
      - $ (- $ P) $ - =>
        let
          fun is-conj Const- ‹induct-conj for P Q› = is-conj P andalso is-conj Q
            | is-conj Const- ‹induct-equal - for - -› = true
            | is-conj Const- ‹induct-true› = true
            | is-conj Const- ‹induct-false› = true
            | is-conj - = false
        in if is-conj P then SOME @{thm induct-conj-curry} else NONE end
    | - => NONE)›

declaration ‹
  K (Induct.map-simpset
    (Simplifier.add-proc simproc ‹swap-induct-false› #>
     Simplifier.add-proc simproc ‹induct-equal-conj-curry› #>
     Simplifier.set-mksimps (fn ctxt =>
       Simpdata.mksimps Simpdata.mksimps-pairs ctxt #>
     map (rewrite-rule ctxt (map Thm.symmetric @{thms induct-rulify-fallback})))))
›
```

Pre-simplification of induction and cases rules

```
lemma [induct-simp]: (⋀x. induct-equal x t ⟹ PROP P x) ≡ PROP P t
  unfolding induct-equal-def
```

proof
assume *r*: $\bigwedge x.\ x = t \Longrightarrow PROP\ P\ x$
show *PROP P t* **by** (*rule r* [*OF refl*])
next
fix *x*
assume *PROP P t x* = *t*
then show *PROP P x* **by** *simp*
qed

lemma [*induct-simp*]: $(\bigwedge x.\ induct\text{-}equal\ t\ x \Longrightarrow PROP\ P\ x) \equiv PROP\ P\ t$
unfolding *induct-equal-def*
proof
assume *r*: $\bigwedge x.\ t = x \Longrightarrow PROP\ P\ x$
show *PROP P t* **by** (*rule r* [*OF refl*])
next
fix *x*
assume *PROP P t t* = *x*
then show *PROP P x* **by** *simp*
qed

lemma [*induct-simp*]: $(induct\text{-}false \Longrightarrow P) \equiv Trueprop\ induct\text{-}true$
unfolding *induct-false-def induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $(induct\text{-}true \Longrightarrow PROP\ P) \equiv PROP\ P$
unfolding *induct-true-def*
proof
assume *True* $\Longrightarrow$ *PROP P*
then show *PROP P* **using** *TrueI* **.**
next
assume *PROP P*
then show *PROP P* **.**
qed

lemma [*induct-simp*]: $(PROP\ P \Longrightarrow induct\text{-}true) \equiv Trueprop\ induct\text{-}true$
unfolding *induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: $(\bigwedge x{::}'a{::}\{\}.\ induct\text{-}true) \equiv Trueprop\ induct\text{-}true$
unfolding *induct-true-def*
by (*iprover intro*: *equal-intr-rule*)

lemma [*induct-simp*]: *induct-implies induct-true P* ≡ *P*
by (*simp add*: *induct-implies-def induct-true-def*)

lemma [*induct-simp*]: *x* = *x* ⟷ *True*
by (*rule simp-thms*)

end

ML-file ‹~~/src/Tools/induct-tacs.ML›

2.3.6 Coherent logic

ML-file ‹~~/src/Tools/coherent.ML›
ML ‹
```
structure Coherent = Coherent
(
  val atomize-elimL = @{thm atomize-elimL};
  val atomize-exL = @{thm atomize-exL};
  val atomize-conjL = @{thm atomize-conjL};
  val atomize-disjL = @{thm atomize-disjL};
 val operator-names = [const-name ‹HOL.disj›, const-name ‹HOL.conj›, const-name ‹Ex›];
);
```
›

2.3.7 Reorienting equalities

ML ‹
```
signature REORIENT-PROC =
sig
  val add : (term -> bool) -> theory -> theory
  val proc : Simplifier.proc
end;

structure Reorient-Proc : REORIENT-PROC =
struct
  structure Data = Theory-Data
  (
    type T = ((term -> bool) * stamp) list;
    val empty = [];
    fun merge data : T = Library.merge (eq-snd (op =)) data;
  );
  fun add m = Data.map (cons (m, stamp ()));
  fun matches thy t = exists (fn (m, -) => m t) (Data.get thy);

  val meta-reorient = @{thm eq-commute [THEN eq-reflection]};
  fun proc ctxt ct =
    let
      val thy = Proof-Context.theory-of ctxt;
    in
      case Thm.term-of ct of
        (- $ t $ u) => if matches thy u then NONE else SOME meta-reorient
      | - => NONE
    end;
end;
```
›

2.4 Other simple lemmas and lemma duplicates

lemma *eq-iff-swap*: $(x = y \longleftrightarrow P) \Longrightarrow (y = x \longleftrightarrow P)$
by *blast*

lemma *all-cong1*: $(\bigwedge x.\ P\ x = P'\ x) \Longrightarrow (\forall x.\ P\ x) = (\forall x.\ P'\ x)$
by *auto*

lemma *ex-cong1*: $(\bigwedge x.\ P\ x = P'\ x) \Longrightarrow (\exists x.\ P\ x) = (\exists x.\ P'\ x)$
by *auto*

lemma *all-cong*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x = P'\ x) \Longrightarrow (\forall x.\ Q\ x \longrightarrow P\ x) = (\forall x.\ Q\ x \longrightarrow P'\ x)$
by *auto*

lemma *ex-cong*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x = P'\ x) \Longrightarrow (\exists x.\ Q\ x \wedge P\ x) = (\exists x.\ Q\ x \wedge P'\ x)$
by *auto*

lemma *ex1-eq* [*iff*]: $\exists!x.\ x = t\ \exists!x.\ t = x$
by *blast+*

lemma *choice-eq*: $(\forall x.\ \exists!y.\ P\ x\ y) = (\exists!f.\ \forall x.\ P\ x\ (f\ x))$ (**is** *?lhs = ?rhs*)
proof (*intro iffI allI*)
assume *L*: *?lhs*
then have $*$: $\forall x.\ P\ x\ (THE\ y.\ P\ x\ y)$
by (*best intro*: *theI'*)
show *?rhs*
by (*rule ex1I*) (*use L* $*$ **in** ‹*fast+*›)
next
fix *x*
assume *R*: *?rhs*
then obtain *f* **where** *f*: $\forall x.\ P\ x\ (f\ x)$ **and** *f1*: $\bigwedge y.\ (\forall x.\ P\ x\ (y\ x)) \Longrightarrow y = f$
by (*blast elim*: *ex1E*)
show $\exists!y.\ P\ x\ y$
proof (*rule ex1I*)
show $P\ x\ (f\ x)$
using *f* **by** *blast*
show $y = f\ x$ **if** $P\ x\ y$ **for** *y*
proof −
have $P\ z$ (*if* $z = x$ *then y else f z*) **for** *z*
using *f that* **by** (*auto split*: *if-split*)
with *f1* [*of* $\lambda z.$ *if* $z = x$ *then y else f z*] *f*
show *?thesis*
by (*auto simp add*: *split*: *if-split-asm dest*: *fun-cong*)
qed
qed
qed

lemmas *eq-sym-conv* = *eq-commute*

lemma *nnf-simps*:
$(\neg\ (P \wedge Q)) = (\neg\ P \vee \neg\ Q)$
$(\neg\ (P \vee Q)) = (\neg\ P \wedge \neg\ Q)$
$(P \longrightarrow Q) = (\neg\ P \vee Q)$
$(P = Q) = ((P \wedge Q) \vee (\neg\ P \wedge \neg\ Q))$
$(\neg\ (P = Q)) = ((P \wedge \neg\ Q) \vee (\neg\ P \wedge Q))$
$(\neg \neg\ P) = P$
by *blast+*

2.5 Basic ML bindings

ML ‹
val FalseE = @{thm FalseE}
val Let-def = @{thm Let-def}
val TrueI = @{thm TrueI}
val allE = @{thm allE}
val allI = @{thm allI}
val all-dupE = @{thm all-dupE}
val arg-cong = @{thm arg-cong}
val box-equals = @{thm box-equals}
val ccontr = @{thm ccontr}
val classical = @{thm classical}
val conjE = @{thm conjE}
val conjI = @{thm conjI}
val conjunct1 = @{thm conjunct1}
val conjunct2 = @{thm conjunct2}
val disjCI = @{thm disjCI}
val disjE = @{thm disjE}
val disjI1 = @{thm disjI1}
val disjI2 = @{thm disjI2}
val eq-reflection = @{thm eq-reflection}
val ex1E = @{thm ex1E}
val ex1I = @{thm ex1I}
val ex1-implies-ex = @{thm ex1-implies-ex}
val exE = @{thm exE}
val exI = @{thm exI}
val excluded-middle = @{thm excluded-middle}
val ext = @{thm ext}
val fun-cong = @{thm fun-cong}
val iffD1 = @{thm iffD1}
val iffD2 = @{thm iffD2}
val iffI = @{thm iffI}
val impE = @{thm impE}
val impI = @{thm impI}
val meta-eq-to-obj-eq = @{thm meta-eq-to-obj-eq}
val mp = @{thm mp}
val notE = @{thm notE}
val notI = @{thm notI}

val not-all = @{*thm not-all*}
val not-ex = @{*thm not-ex*}
val not-iff = @{*thm not-iff*}
val not-not = @{*thm not-not*}
val not-sym = @{*thm not-sym*}
val refl = @{*thm refl*}
val rev-mp = @{*thm rev-mp*}
val spec = @{*thm spec*}
val ssubst = @{*thm ssubst*}
val subst = @{*thm subst*}
val sym = @{*thm sym*}
val trans = @{*thm trans*}
›

locale *cnf*
begin

lemma *clause2raw-notE*: $[\![P; \neg P]\!] \Longrightarrow$ *False* **by** *auto*
lemma *clause2raw-not-disj*: $[\![\neg\ P; \neg\ Q]\!] \Longrightarrow \neg\ (P \vee Q)$ **by** *auto*
lemma *clause2raw-not-not*: $P \Longrightarrow \neg\neg\ P$ **by** *auto*

lemma *iff-refl*: $(P::bool) = P$ **by** *auto*
lemma *iff-trans*: $[|\ (P::bool) = Q;\ Q = R\ |] ==> P = R$ **by** *auto*
lemma *conj-cong*: $[|\ P = P';\ Q = Q'\ |] ==> (P \wedge Q) = (P' \wedge Q')$ **by** *auto*
lemma *disj-cong*: $[|\ P = P';\ Q = Q'\ |] ==> (P \vee Q) = (P' \vee Q')$ **by** *auto*

lemma *make-nnf-imp*: $[|\ (\neg P) = P';\ Q = Q'\ |] ==> (P \longrightarrow Q) = (P' \vee Q')$ **by** *auto*
lemma *make-nnf-iff*: $[|\ P = P';\ (\neg P) = NP;\ Q = Q';\ (\neg Q) = NQ\ |] ==> (P = Q) = ((P' \vee NQ) \wedge (NP \vee Q'))$ **by** *auto*
lemma *make-nnf-not-false*: $(\neg False) = True$ **by** *auto*
lemma *make-nnf-not-true*: $(\neg True) = False$ **by** *auto*
lemma *make-nnf-not-conj*: $[|\ (\neg P) = P';\ (\neg Q) = Q'\ |] ==> (\neg(P \wedge Q)) = (P' \vee Q')$ **by** *auto*
lemma *make-nnf-not-disj*: $[|\ (\neg P) = P';\ (\neg Q) = Q'\ |] ==> (\neg(P \vee Q)) = (P' \wedge Q')$ **by** *auto*
lemma *make-nnf-not-imp*: $[|\ P = P';\ (\neg Q) = Q'\ |] ==> (\neg(P \longrightarrow Q)) = (P' \wedge Q')$ **by** *auto*
lemma *make-nnf-not-iff*: $[|\ P = P';\ (\neg P) = NP;\ Q = Q';\ (\neg Q) = NQ\ |] ==> (\neg(P = Q)) = ((P' \vee Q') \wedge (NP \vee NQ))$ **by** *auto*
lemma *make-nnf-not-not*: $P = P' ==> (\neg\neg P) = P'$ **by** *auto*

lemma *simp-TF-conj-True-l*: $[|\ P = True;\ Q = Q'\ |] ==> (P \wedge Q) = Q'$ **by** *auto*
lemma *simp-TF-conj-True-r*: $[|\ P = P';\ Q = True\ |] ==> (P \wedge Q) = P'$ **by** *auto*
lemma *simp-TF-conj-False-l*: $P = False ==> (P \wedge Q) = False$ **by** *auto*
lemma *simp-TF-conj-False-r*: $Q = False ==> (P \wedge Q) = False$ **by** *auto*
lemma *simp-TF-disj-True-l*: $P = True ==> (P \vee Q) = True$ **by** *auto*
lemma *simp-TF-disj-True-r*: $Q = True ==> (P \vee Q) = True$ **by** *auto*
lemma *simp-TF-disj-False-l*: $[|\ P = False;\ Q = Q'\ |] ==> (P \vee Q) = Q'$ **by** *auto*

lemma *simp-TF-disj-False-r*: [| $P = P'$; $Q = False$ |] ==> $(P \vee Q) = P'$ **by** *auto*

lemma *make-cnf-disj-conj-l*: [| $(P \vee R) = PR$; $(Q \vee R) = QR$ |] ==> $((P \wedge Q) \vee R) = (PR \wedge QR)$ **by** *auto*
lemma *make-cnf-disj-conj-r*: [| $(P \vee Q) = PQ$; $(P \vee R) = PR$ |] ==> $(P \vee (Q \wedge R)) = (PQ \wedge PR)$ **by** *auto*

lemma *make-cnfx-disj-ex-l*: $((\exists (x{::}bool).\ P\ x) \vee Q) = (\exists x.\ P\ x \vee Q)$ **by** *auto*
lemma *make-cnfx-disj-ex-r*: $(P \vee (\exists (x{::}bool).\ Q\ x)) = (\exists x.\ P \vee Q\ x)$ **by** *auto*
lemma *make-cnfx-newlit*: $(P \vee Q) = (\exists x.\ (P \vee x) \wedge (Q \vee \neg x))$ **by** *auto*
lemma *make-cnfx-ex-cong*: $(\forall (x{::}bool).\ P\ x = Q\ x) \Longrightarrow (\exists x.\ P\ x) = (\exists x.\ Q\ x)$ **by** *auto*

lemma *weakening-thm*: [| P; Q |] ==> Q **by** *auto*

lemma *cnftac-eq-imp*: [| $P = Q$; P |] ==> Q **by** *auto*

end

ML-file ‹*Tools/cnf.ML*›

3 *NO-MATCH* simproc

The simplification procedure can be used to avoid simplification of terms of a certain form.

definition *NO-MATCH* :: $'a \Rightarrow 'b \Rightarrow bool$
  where *NO-MATCH pat val* $\equiv$ *True*

lemma *NO-MATCH-cong*[*cong*]: *NO-MATCH pat val* = *NO-MATCH pat val*
  by (*rule refl*)

declare [[*coercion-args NO-MATCH* − −]]

simproc-setup *NO-MATCH* (*NO-MATCH pat val*) = ‹*K* (*fn ctxt* => *fn ct* =>
  let
    val thy = *Proof-Context.theory-of ctxt*
    val dest-binop = *Term.dest-comb* #> *apfst* (*Term.dest-comb* #> *snd*)
    val m = *Pattern.matches thy* (*dest-binop* (*Thm.term-of ct*))
  in if m then NONE else SOME @{*thm NO-MATCH-def*} *end*)
›

This setup ensures that a rewrite rule of the form *NO-MATCH pat val* $\Longrightarrow$ t is only applied, if the pattern *pat* does not match the value *val*.

Tagging a premise of a simp rule with ASSUMPTION forces the simplifier not to simplify the argument and to solve it by an assumption.

definition *ASSUMPTION* :: *bool* $\Rightarrow$ *bool*
  where *ASSUMPTION A* $\equiv$ *A*

lemma *ASSUMPTION-cong*[*cong*]: *ASSUMPTION A* = *ASSUMPTION A*
by (*rule refl*)

lemma *ASSUMPTION-I*: *A* ⟹ *ASSUMPTION A*
by (*simp add*: *ASSUMPTION-def*)

lemma *ASSUMPTION-D*: *ASSUMPTION A* ⟹ *A*
by (*simp add*: *ASSUMPTION-def*)

setup ‹
Simplifier.map-theory-simpset (*Simplifier.add-unsafe-solver* (
Simplifier.mk-solver ASSUMPTION (*fn ctxt* =>
resolve-tac ctxt @{*thms ASSUMPTION-I*} *THEN'*
resolve-tac ctxt (*Simplifier.prems-of ctxt*))))
›

3.1 Code generator setup

3.1.1 Generic code generator preprocessor setup

lemma *conj-left-cong*: *P* ⟷ *Q* ⟹ *P* ∧ *R* ⟷ *Q* ∧ *R*
by (*fact arg-cong*)

lemma *disj-left-cong*: *P* ⟷ *Q* ⟹ *P* ∨ *R* ⟷ *Q* ∨ *R*
by (*fact arg-cong*)

setup ‹
Code-Preproc.map-pre (*put-simpset HOL-basic-ss*) #>
Code-Preproc.map-post (*put-simpset HOL-basic-ss*) #>
Code-Simp.map-ss (*put-simpset HOL-basic-ss* #>
Simplifier.add-cong @{*thm conj-left-cong*} #>
Simplifier.add-cong @{*thm disj-left-cong*})
›

3.1.2 Generic code generator foundation

Datatype *bool*

code-datatype *True False*

lemma [*code*]:
P ∧ *True* ⟷ *P*
P ∧ *False* ⟷ *False*
True ∧ *P* ⟷ *P*
False ∧ *P* ⟷ *False*
by *simp-all*

lemma [*code*]:
P ∨ *True* ⟷ *True*

$P \vee False \longleftrightarrow P$
$True \vee P \longleftrightarrow True$
$False \vee P \longleftrightarrow P$
by *simp-all*

lemma [*code*]:
$(P \longrightarrow True) \longleftrightarrow True$
$(P \longrightarrow False) \longleftrightarrow \neg P$
$(True \longrightarrow P) \longleftrightarrow P$
$(False \longrightarrow P) \longleftrightarrow True$
by *simp-all*

More about *prop*

lemma [*code nbe*]:
$(P \Longrightarrow R) \equiv Trueprop\ (P \longrightarrow R)$
$(PROP\ Q \Longrightarrow True) \equiv Trueprop\ True$
$(True \Longrightarrow PROP\ Q) \equiv PROP\ Q$
by (*auto intro*!: *equal-intr-rule*)

lemma *Trueprop-code* [*code*]: $Trueprop\ True \equiv Code\text{-}Generator.holds$
by (*auto intro*!: *equal-intr-rule holds*)

declare *Trueprop-code* [*symmetric*, *code-post*]

Cases

lemma *Let-case-cert*:
assumes $CASE \equiv (\lambda x.\ Let\ x\ f)$
shows $CASE\ x \equiv f\ x$
using *assms* **by** *simp-all*

setup ‹
Code.declare-case-global @{*thm Let-case-cert*} #>
Code.declare-undefined-global ***const-name*** ‹*undefined*›
›

declare [[*code abort*: *undefined*]]

3.1.3 Equality

lemma [*code nbe*]:
‹$x = x \longleftrightarrow True$›
by *iprover*

class *equal* =
fixes *equal* :: $'a \Rightarrow 'a \Rightarrow bool$
assumes *equal-eq*: $equal\ x\ y \longleftrightarrow x = y$
begin

lemma *eq-equal* [*code*]: $(=) \equiv equal$

by (*rule eq-reflection*) (*rule ext*, *rule ext*, *rule sym*, *rule equal-eq*)

lemma *equal* [*code-post*]: *equal* = (=)
by (*rule ext equal-eq*)+

lemma *equal-refl*: *equal x x* ⟷ *True*
unfolding *equal* **by** (*rule iffI TrueI refl*)+

end

simproc-setup passive *equal* (*HOL.eq*) =
‹*fn - => fn - => fn ct =>*
(*case Thm.term-of ct of*
Const- ‹*HOL.eq T*› *=> if is-Type T then SOME @*{*thm eq-equal*} *else NONE*
| *- => NONE*)›

setup ‹*Code-Preproc.map-pre* (*Simplifier.add-proc* ***simproc*** ‹*equal*›)›

instantiation *itself* :: (*type*) *equal*
begin

definition *equal-itself* :: *'a itself* ⇒ *'a itself* ⇒ *bool*
where *equal-itself x y* ⟷ *x* = *y*

instance
by *standard* (*fact equal-itself-def*)

end

lemma *equal-itself-code* [*code*]: *equal TYPE*(*'a*) *TYPE*(*'a*) ⟷ *True*
by (*simp add*: *equal*)

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*equal*›, *SOME* ***typ*** ‹*'a*::*type* ⇒ *'a*
⇒ *bool*›)›

lemma *equal-alias-cert*: *OFCLASS*(*'a*, *equal-class*) ≡ (((=) :: *'a* ⇒ *'a* ⇒ *bool*) ≡
equal)
(**is** *?ofclass* ≡ *?equal*)
proof
assume *PROP ?ofclass*
show *PROP ?equal*
by (*tactic* ‹*ALLGOALS* (*resolve-tac* ***context*** [*Thm.unconstrainT* @{*thm eq-equal*}])›)
(*fact* ‹*PROP ?ofclass*›)
next
assume *PROP ?equal*
show *PROP ?ofclass* **proof**
qed (*simp add*: ‹*PROP ?equal*›)
qed

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*equal*›, *SOME* ***typ*** ‹*'a*::*equal* ⇒ *'a* ⇒ *bool*›)›

setup ‹*Nbe.add-const-alias* @{*thm equal-alias-cert*}›

3.1.4 Generic code generator target languages

type *bool*

code-printing
  type-constructor *bool* ⇀
    (*SML*) *bool* **and** (*OCaml*) *bool* **and** (*Haskell*) *Bool* **and** (*Scala*) *Boolean*
| **constant** *True* ⇀
    (*SML*) *true* **and** (*OCaml*) *true* **and** (*Haskell*) *True* **and** (*Scala*) *true*
| **constant** *False* ⇀
    (*SML*) *false* **and** (*OCaml*) *false* **and** (*Haskell*) *False* **and** (*Scala*) *false*

code-reserved
  (*SML*) *bool true false*
  and (*OCaml*) *bool*
  and (*Scala*) *Boolean*

code-printing
  constant *Not* ⇀
    (*SML*) *not* **and** (*OCaml*) *not* **and** (*Haskell*) *not* **and** (*Scala*) *'!* -
| **constant** *HOL.conj* ⇀
    (*SML*) **infixl** *1 andalso* **and** (*OCaml*) **infixl** *3* && **and** (*Haskell*) **infixr** *3* && **and** (*Scala*) **infixl** *3* &&
| **constant** *HOL.disj* ⇀
    (*SML*) **infixl** *0 orelse* **and** (*OCaml*) **infixl** *2* || **and** (*Haskell*) **infixl** *2* || **and** (*Scala*) **infixl** *1* ||
| **constant** *HOL.implies* ⇀
    (*SML*) !(*if* (-)/ *then* (-)/ *else true*)
    and (*OCaml*) !(*if* (-)/ *then* (-)/ *else true*)
    and (*Haskell*) !(*if* (-)/ *then* (-)/ *else True*)
    and (*Scala*) !((-) *match* {/ *case true* => (-)/ *case false* => *true*/ })
| **constant** *If* ⇀
    (*SML*) !(*if* (-)/ *then* (-)/ *else* (-))
    and (*OCaml*) !(*if* (-)/ *then* (-)/ *else* (-))
    and (*Haskell*) !(*if* (-)/ *then* (-)/ *else* (-))
    and (*Scala*) !((-) *match* {/ *case true* => (-)/ *case false* => (-)/ })

code-reserved
  (*SML*) *not*
  and (*OCaml*) *not*

code-identifier
  code-module *Pure* ⇀
    (*SML*) *HOL* **and** (*OCaml*) *HOL* **and** (*Haskell*) *HOL* **and** (*Scala*) *HOL*

Using built-in Haskell equality.

code-printing
  type-class *equal* ⇀ (*Haskell*) *Eq*
| **constant** *HOL.equal* ⇀ (*Haskell*) **infix** *4* ==
| **constant** *HOL.eq* ⇀ (*Haskell*) **infix** *4* ==

undefined

code-printing
  constant *undefined* ⇀
    (*SML*) !(*raise*/ *Fail*/ *undefined*)
    and (*OCaml*) *failwith*/ *undefined*
    and (*Haskell*) *error*/ *undefined*
    and (*Scala*) !*sys.error*(*undefined*)

3.1.5 Evaluation and normalization by evaluation

method-setup *eval* = ‹
  let
    fun eval-tac ctxt =
      let val conv = *Code-Runtime.dynamic-holds-conv*
      in
        CONVERSION (*Conv.params-conv* ~*1* (*Conv.concl-conv* ~*1 o conv*) *ctxt*)
THEN'
        resolve-tac ctxt [*TrueI*]
      end
  in
    Scan.succeed (*SIMPLE-METHOD' o eval-tac*)
  end
› *solve goal by evaluation*

method-setup *normalization* = ‹
  Scan.succeed (*fn ctxt* =>
    SIMPLE-METHOD'
      (*CHANGED-PROP o*
        (*CONVERSION* (*Nbe.dynamic-conv ctxt*)
          THEN-ALL-NEW (*TRY o resolve-tac ctxt* [*TrueI*]))))
› *solve goal by normalization*

3.2 Counterexample Search Units

3.2.1 Quickcheck

quickcheck-params [*size* = *5*, *iterations* = *50*]

3.2.2 Nitpick setup

named-theorems *nitpick-unfold alternative definitions of constants as needed by Nitpick*
  and *nitpick-simp equational specification of constants as needed by Nitpick*

and *nitpick-psimp partial equational specification of constants as needed by Nitpick*
and *nitpick-choice-spec choice specification of constants as needed by Nitpick*

declare *if-bool-eq-conj* [*nitpick-unfold*, *no-atp*]
and *if-bool-eq-disj* [*no-atp*]

3.3 Preprocessing for the predicate compiler

named-theorems *code-pred-def alternative definitions of constants for the Predicate Compiler*
and *code-pred-inline inlining definitions for the Predicate Compiler*
and *code-pred-simp simplification rules for the optimisations in the Predicate Compiler*

3.4 Legacy tactics and ML bindings

ML ‹
(∗ *combination of* (*spec RS spec RS ...*(*j times*) *... spec RS mp*) ∗)
local
fun wrong-prem ***Const-*** ‹*All - for* ‹*Abs* (-, -, *t*)›› = *wrong-prem t*
| *wrong-prem* (*Bound* -) = *true*
| *wrong-prem* - = *false*;
val filter-right = *filter* (*not o wrong-prem o HOLogic.dest-Trueprop o hd o Thm.take-prems-of 1*);
fun smp i = *funpow i* (*fn m* => *filter-right* ([*spec*] *RL m*)) [*mp*];
in
fun smp-tac ctxt j = *EVERY'* [*dresolve-tac ctxt* (*smp j*), *assume-tac ctxt*];
end;

local
val nnf-ss =
HOL-basic-ss
|> *Simplifier.simpset-map* ***context*** (*Simplifier.add-simps* @{*thms simp-thms nnf-simps*});
in
fun nnf-conv ctxt = *Simplifier.rewrite* (*put-simpset nnf-ss ctxt*);
end
›

hide-const (**open**) *eq equal*

end

4 Abstract orderings

theory *Orderings*
imports *HOL*
keywords *print-orders* :: *diag*
begin

4.1 Abstract ordering

locale *partial-preordering* =
  fixes *less-eq* :: ‹'*a* ⇒ '*a* ⇒ *bool*› (**infix** ‹≤› *50*)
  assumes *refl*: ‹*a* ≤ *a*› — not *iff*: makes problems due to multiple (dual) interpretations
    and *trans*: ‹*a* ≤ *b* ⟹ *b* ≤ *c* ⟹ *a* ≤ *c*›

locale *preordering* = *partial-preordering* +
  fixes *less* :: ‹'*a* ⇒ '*a* ⇒ *bool*› (**infix** ‹<› *50*)
  assumes *strict-iff-not*: ‹*a* < *b* ⟷ *a* ≤ *b* ∧ ¬ *b* ≤ *a*›
begin

lemma *strict-implies-order*:
  ‹*a* < *b* ⟹ *a* ≤ *b*›
  by (*simp add*: *strict-iff-not*)

lemma *irrefl*: — not *iff*: makes problems due to multiple (dual) interpretations
  ‹¬ *a* < *a*›
  by (*simp add*: *strict-iff-not*)

lemma *asym*:
  ‹*a* < *b* ⟹ *b* < *a* ⟹ *False*›
  by (*auto simp add*: *strict-iff-not*)

lemma *strict-trans1*:
  ‹*a* ≤ *b* ⟹ *b* < *c* ⟹ *a* < *c*›
  by (*auto simp add*: *strict-iff-not intro*: *trans*)

lemma *strict-trans2*:
  ‹*a* < *b* ⟹ *b* ≤ *c* ⟹ *a* < *c*›
  by (*auto simp add*: *strict-iff-not intro*: *trans*)

lemma *strict-trans*:
  ‹*a* < *b* ⟹ *b* < *c* ⟹ *a* < *c*›
  by (*auto intro*: *strict-trans1 strict-implies-order*)

end

lemma *preordering-strictI*: — Alternative introduction rule with bias towards strict order
  fixes *less-eq* (**infix** ‹≤› *50*)
    and *less* (**infix** ‹<› *50*)
  assumes *less-eq-less*: ‹⋀*a b*. *a* ≤ *b* ⟷ *a* < *b* ∨ *a* = *b*›
    assumes *asym*: ‹⋀*a b*. *a* < *b* ⟹ ¬ *b* < *a*›
  assumes *irrefl*: ‹⋀*a*. ¬ *a* < *a*›
  assumes *trans*: ‹⋀*a b c*. *a* < *b* ⟹ *b* < *c* ⟹ *a* < *c*›
  shows ‹*preordering* (≤) (<)›
proof
  fix *a b*

```
  show ‹a < b ⟷ a ≤ b ∧ ¬ b ≤ a›
    by (auto simp add: less-eq-less asym irrefl)
next
  fix a
  show ‹a ≤ a›
    by (auto simp add: less-eq-less)
next
  fix a b c
  assume ‹a ≤ b› and ‹b ≤ c› then show ‹a ≤ c›
    by (auto simp add: less-eq-less intro: trans)
qed

lemma preordering-dualI:
  fixes less-eq (infix ‹≤› 50)
    and less (infix ‹<› 50)
  assumes ‹preordering (λa b. b ≤ a) (λa b. b < a)›
  shows ‹preordering (≤) (<)›
proof −
  from assms interpret preordering ‹λa b. b ≤ a› ‹λa b. b < a› .
  show ?thesis
    by standard (auto simp: strict-iff-not refl intro: trans)
qed

locale ordering = partial-preordering +
  fixes less :: ‹'a ⇒ 'a ⇒ bool› (infix ‹<› 50)
  assumes strict-iff-order: ‹a < b ⟷ a ≤ b ∧ a ≠ b›
  assumes antisym: ‹a ≤ b ⟹ b ≤ a ⟹ a = b›
begin

sublocale preordering ‹(≤)› ‹(<)›
proof
  show ‹a < b ⟷ a ≤ b ∧ ¬ b ≤ a› for a b
    by (auto simp add: strict-iff-order intro: antisym)
qed

lemma strict-implies-not-eq:
  ‹a < b ⟹ a ≠ b›
  by (simp add: strict-iff-order)

lemma not-eq-order-implies-strict:
  ‹a ≠ b ⟹ a ≤ b ⟹ a < b›
  by (simp add: strict-iff-order)

lemma order-iff-strict:
  ‹a ≤ b ⟷ a < b ∨ a = b›
  by (auto simp add: strict-iff-order refl)

lemma eq-iff: ‹a = b ⟷ a ≤ b ∧ b ≤ a›
  by (auto simp add: refl intro: antisym)
```

end

lemma *ordering-strictI*: — Alternative introduction rule with bias towards strict order
fixes *less-eq* (**infix** ‹≤› *50*)
and *less* (**infix** ‹<› *50*)
assumes *less-eq-less*: ‹⋀*a b*. *a* ≤ *b* ⟷ *a* < *b* ∨ *a* = *b*›
assumes *asym*: ‹⋀*a b*. *a* < *b* ⟹ ¬ *b* < *a*›
assumes *irrefl*: ‹⋀*a*. ¬ *a* < *a*›
assumes *trans*: ‹⋀*a b c*. *a* < *b* ⟹ *b* < *c* ⟹ *a* < *c*›
shows ‹*ordering* (≤) (<)›
proof
fix *a b*
show ‹*a* < *b* ⟷ *a* ≤ *b* ∧ *a* ≠ *b*›
by (*auto simp add*: *less-eq-less asym irrefl*)
next
fix *a*
show ‹*a* ≤ *a*›
by (*auto simp add*: *less-eq-less*)
next
fix *a b c*
assume ‹*a* ≤ *b*› **and** ‹*b* ≤ *c*› **then show** ‹*a* ≤ *c*›
by (*auto simp add*: *less-eq-less intro*: *trans*)
next
fix *a b*
assume ‹*a* ≤ *b*› **and** ‹*b* ≤ *a*› **then show** ‹*a* = *b*›
by (*auto simp add*: *less-eq-less asym*)
qed

lemma *ordering-dualI*:
fixes *less-eq* (**infix** ‹≤› *50*)
and *less* (**infix** ‹<› *50*)
assumes ‹*ordering* (λ*a b*. *b* ≤ *a*) (λ*a b*. *b* < *a*)›
shows ‹*ordering* (≤) (<)›
proof −
from *assms* **interpret** *ordering* ‹λ*a b*. *b* ≤ *a*› ‹λ*a b*. *b* < *a*› **.**
show *?thesis*
by *standard* (*auto simp*: *strict-iff-order refl intro*: *antisym trans*)
qed

locale *ordering-top* = *ordering* +
fixes *top* :: ‹'*a*› (‹⊤›)
assumes *extremum* [*simp*]: ‹*a* ≤ ⊤›
begin

lemma *extremum-uniqueI*:
‹⊤ ≤ *a* ⟹ *a* = ⊤›
by (*rule antisym*) *auto*

lemma *extremum-unique*:
‹⊤ ≤ *a* ⟷ *a* = ⊤›
by (*auto intro*: *antisym*)

lemma *extremum-strict* [*simp*]:
‹¬ (⊤ < *a*)›
using *extremum* [*of a*] **by** (*auto simp add*: *order-iff-strict intro*: *asym irrefl*)

lemma *not-eq-extremum*:
‹*a* ≠ ⊤ ⟷ *a* < ⊤›
by (*auto simp add*: *order-iff-strict intro*: *not-eq-order-implies-strict extremum*)

end

4.2 Syntactic orders

class *ord* =
fixes *less-eq* :: *'a* ⇒ *'a* ⇒ *bool*
and *less* :: *'a* ⇒ *'a* ⇒ *bool*
begin

notation
less-eq (‹*'*(≤*'*)›) **and**
less-eq (‹(‹*notation*=‹*infix* ≤››-/ ≤ -)› [*51*, *51*] *50*) **and**
less (‹*'*(<*'*)›) **and**
less (‹(‹*notation*=‹*infix* <››-/ < -)› [*51*, *51*] *50*)

abbreviation (*input*)
greater-eq (**infix** ‹≥› *50*)
where *x* ≥ *y* ≡ *y* ≤ *x*

abbreviation (*input*)
greater (**infix** ‹>› *50*)
where *x* > *y* ≡ *y* < *x*

notation (*ASCII*)
less-eq (‹*'*(<=*'*)›) **and**
less-eq (‹(‹*notation*=‹*infix* <=››-/ <= -)› [*51*, *51*] *50*)

notation (*input*)
greater-eq (**infix** ‹>=› *50*)

end

4.3 Quasi orders

class *preorder* = *ord* +
assumes *less-le-not-le*: *x* < *y* ⟷ *x* ≤ *y* ∧ ¬ (*y* ≤ *x*)
and *order-refl* [*iff*]: *x* ≤ *x*

and *order-trans*: $x \leq y \Longrightarrow y \leq z \Longrightarrow x \leq z$
begin

sublocale *order*: *preordering less-eq less* + *dual-order*: *preordering greater-eq greater*
proof −
interpret *preordering less-eq less*
by *standard* (*auto intro*: *order-trans simp add*: *less-le-not-le*)
show ‹*preordering less-eq less*›
by (*fact preordering-axioms*)
then show ‹*preordering greater-eq greater*›
by (*rule preordering-dualI*)
qed

Reflexivity.

lemma *eq-refl*: $x = y \Longrightarrow x \leq y$
— This form is useful with the classical reasoner.
by (*erule ssubst*) (*rule order-refl*)

lemma *less-irrefl* [*iff*]: $\neg\ x < x$
by (*simp add*: *less-le-not-le*)

lemma *less-imp-le*: $x < y \Longrightarrow x \leq y$
by (*simp add*: *less-le-not-le*)

Asymmetry.

lemma *less-not-sym*: $x < y \Longrightarrow \neg\ (y < x)$
by (*simp add*: *less-le-not-le*)

lemma *less-asym*: $x < y \Longrightarrow (\neg\ P \Longrightarrow y < x) \Longrightarrow P$
by (*drule less-not-sym*, *erule contrapos-np*) *simp*

Transitivity.

lemma *less-trans*: $x < y \Longrightarrow y < z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

lemma *le-less-trans*: $x \leq y \Longrightarrow y < z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

lemma *less-le-trans*: $x < y \Longrightarrow y \leq z \Longrightarrow x < z$
by (*auto simp add*: *less-le-not-le intro*: *order-trans*)

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-less*: $x < y \Longrightarrow (\neg\ y < x) \longleftrightarrow \mathit{True}$
by (*blast elim*: *less-asym*)

lemma *less-imp-triv*: $x < y \Longrightarrow (y < x \longrightarrow P) \longleftrightarrow \mathit{True}$
by (*blast elim*: *less-asym*)

Transitivity rules for calculational reasoning

lemma *less-asym'*: $a < b \Longrightarrow b < a \Longrightarrow P$
by (*rule less-asym*)

Dual order

lemma *dual-preorder*:
  ‹*class.preorder* $(\geq)$ $(>)$›
  by *standard* (*auto simp add*: *less-le-not-le intro*: *order-trans*)

end

lemma *preordering-preorderI*:
  ‹*class.preorder* $(\leq)$ $(<)$› **if** ‹*preordering* $(\leq)$ $(<)$›
    for *less-eq* (**infix** ‹$\leq$› *50*) **and** *less* (**infix** ‹$<$› *50*)
proof −
  from *that* **interpret** *preordering* ‹$(\leq)$› ‹$(<)$› **.**
  show *?thesis*
    by *standard* (*auto simp add*: *strict-iff-not refl intro*: *trans*)
qed

4.4 Partial orders

class *order* = *preorder* +
  assumes *order-antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x = y$
begin

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge x \neq y$
  by (*auto simp add*: *less-le-not-le intro*: *order-antisym*)

sublocale *order*: *ordering less-eq less* + *dual-order*: *ordering greater-eq greater*
proof −
  interpret *ordering less-eq less*
    by *standard* (*auto intro*: *order-antisym order-trans simp add*: *less-le*)
  show *ordering less-eq less*
    by (*fact ordering-axioms*)
  then show *ordering greater-eq greater*
    by (*rule ordering-dualI*)
qed

Reflexivity.

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x = y$
    — NOT suitable for iff, since it can cause PROOF FAILED.
by (*fact order.order-iff-strict*)

lemma *le-imp-less-or-eq*: $x \leq y \Longrightarrow x < y \vee x = y$
by (*simp add*: *less-le*)

Useful for simplification, but too risky to include by default.

lemma *less-imp-not-eq*: $x < y \Longrightarrow (x = y) \longleftrightarrow \mathit{False}$
by *auto*

lemma *less-imp-not-eq2*: $x < y \Longrightarrow (y = x) \longleftrightarrow \mathit{False}$
by *auto*

Transitivity rules for calculational reasoning

lemma *neq-le-trans*: $a \neq b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*fact order.not-eq-order-implies-strict*)

lemma *le-neq-trans*: $a \leq b \Longrightarrow a \neq b \Longrightarrow a < b$
by (*rule order.not-eq-order-implies-strict*)

Asymmetry.

lemma *order-eq-iff*: $x = y \longleftrightarrow x \leq y \wedge y \leq x$
by (*fact order.eq-iff*)

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x = y$
by (*simp add*: *order.eq-iff*)

lemma *less-imp-neq*: $x < y \Longrightarrow x \neq y$
by (*fact order.strict-implies-not-eq*)

lemma *antisym-conv1*: $\neg\ x < y \Longrightarrow x \leq y \longleftrightarrow x = y$
by (*simp add*: *local.le-less*)

lemma *antisym-conv2*: $x \leq y \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
by (*simp add*: *local.less-le*)

lemma *leD*: $y \leq x \Longrightarrow \neg\ x < y$
by (*auto simp*: *less-le order.antisym*)

Least value operator

definition (**in** *ord*)
Least :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a$ (**binder** ‹*LEAST* › *10*) **where**
$\mathit{Least}\ P = (\mathit{THE}\ x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \leq y))$

lemma *Least-equality*:
assumes $P\ x$
and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
shows $\mathit{Least}\ P = x$
unfolding *Least-def* **by** (*rule the-equality*)
(*blast intro*: *assms order.antisym*)+

lemma *LeastI2-order*:
assumes $P\ x$
and $\bigwedge y.\ P\ y \Longrightarrow x \leq y$
and $\bigwedge x.\ P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow x \leq y \Longrightarrow Q\ x$
shows $Q\ (\mathit{Least}\ P)$
unfolding *Least-def* **by** (*rule theI2*)
(*blast intro*: *assms order.antisym*)+

lemma *Least-ex1*:
assumes $\exists!x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow x \leq y)$
shows *Least1I*: P (*Least* P) **and** *Least1-le*: $P\ z \Longrightarrow$ *Least* $P \leq z$
using *theI'*[*OF assms*]
unfolding *Least-def*
by *auto*

Greatest value operator

definition *Greatest* :: ('$a \Rightarrow$ *bool*) $\Rightarrow$ 'a (**binder** ‹*GREATEST* › *10*) **where**
Greatest P = (*THE* x. $P\ x \wedge (\forall y.\ P\ y \longrightarrow x \geq y)$)

lemma *GreatestI2-order*:
$[\![\ P\ x;$
$\bigwedge y.\ P\ y \Longrightarrow x \geq y;$
$\bigwedge x.\ [\![\ P\ x; \forall y.\ P\ y \longrightarrow x \geq y\ ]\!] \Longrightarrow Q\ x\ ]\!]$
$\Longrightarrow Q$ (*Greatest* P)
unfolding *Greatest-def*
by (*rule theI2*) (*blast intro*: *order.antisym*)+

lemma *Greatest-equality*:
$[\![\ P\ x;\ \bigwedge y.\ P\ y \Longrightarrow x \geq y\ ]\!] \Longrightarrow$ *Greatest* $P = x$
unfolding *Greatest-def*
by (*rule the-equality*) (*blast intro*: *order.antisym*)+

end

lemma *ordering-orderI*:
fixes *less-eq* (**infix** ‹≤› *50*)
and *less* (**infix** ‹<› *50*)
assumes *ordering less-eq less*
shows *class.order less-eq less*
proof −
from *assms* **interpret** *ordering less-eq less* **.**
show *?thesis*
by *standard* (*auto intro*: *antisym trans simp add*: *refl strict-iff-order*)
qed

lemma *order-strictI*:
fixes *less* (**infix** ‹<› *50*)
and *less-eq* (**infix** ‹≤› *50*)
assumes $\bigwedge a\ b.\ a \leq b \longleftrightarrow a < b \vee a = b$
assumes $\bigwedge a\ b.\ a < b \Longrightarrow \neg\ b < a$
assumes $\bigwedge a.\ \neg\ a < a$
assumes $\bigwedge a\ b\ c.\ a < b \Longrightarrow b < c \Longrightarrow a < c$
shows *class.order less-eq less*
by (*rule ordering-orderI*) (*rule ordering-strictI*, (*fact assms*)+)

context *order*

begin

Dual order

lemma *dual-order*:
class.order $(\geq)$ $(>)$
using *dual-order.ordering-axioms* **by** (*rule ordering-orderI*)

end

4.5 Linear (total) orders

class *linorder* = *order* +
assumes *linear*: $x \leq y \vee y \leq x$
begin

lemma *less-linear*: $x < y \vee x = y \vee y < x$
unfolding *less-le* **using** *less-le linear* **by** *blast*

lemma *le-less-linear*: $x \leq y \vee y < x$
by (*simp add*: *le-less less-linear*)

lemma *le-cases* [*case-names le ge*]:
$(x \leq y \Longrightarrow P) \Longrightarrow (y \leq x \Longrightarrow P) \Longrightarrow P$
using *linear* **by** *blast*

lemma (**in** *linorder*) *le-cases3*:
$[\![[\![x \leq y; y \leq z]\!] \Longrightarrow P; [\![y \leq x; x \leq z]\!] \Longrightarrow P; [\![x \leq z; z \leq y]\!] \Longrightarrow P;$
$[\![z \leq y; y \leq x]\!] \Longrightarrow P; [\![y \leq z; z \leq x]\!] \Longrightarrow P; [\![z \leq x; x \leq y]\!] \Longrightarrow P]\!] \Longrightarrow P$
by (*blast intro*: *le-cases*)

lemma *linorder-cases* [*case-names less equal greater*]:
$(x < y \Longrightarrow P) \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow (y < x \Longrightarrow P) \Longrightarrow P$
using *less-linear* **by** *blast*

lemma *linorder-wlog*[*case-names le sym*]:
$(\bigwedge a\ b.\ a \leq b \Longrightarrow P\ a\ b) \Longrightarrow (\bigwedge a\ b.\ P\ b\ a \Longrightarrow P\ a\ b) \Longrightarrow P\ a\ b$
by (*cases rule*: *le-cases*[*of a b*]) *blast+*

lemma *not-less*: $\neg\ x < y \longleftrightarrow y \leq x$
unfolding *less-le*
using *linear* **by** (*blast intro*: *order.antisym*)

lemma *not-less-iff-gr-or-eq*: $\neg(x < y) \longleftrightarrow (x > y \vee x = y)$
by (*auto simp add*:*not-less le-less*)

lemma *not-le*: $\neg\ x \leq y \longleftrightarrow y < x$
unfolding *less-le*
using *linear* **by** (*blast intro*: *order.antisym*)

lemma *neq-iff*: $x \neq y \longleftrightarrow x < y \vee y < x$
by (*cut-tac* $x = x$ **and** $y = y$ **in** *less-linear*, *auto*)

lemma *neqE*: $x \neq y \Longrightarrow (x < y \Longrightarrow R) \Longrightarrow (y < x \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *neq-iff*) *blast*

lemma *antisym-conv3*: $\neg\ y < x \Longrightarrow \neg\ x < y \longleftrightarrow x = y$
by (*blast intro*: *order.antisym dest*: *not-less* [*THEN iffD1*])

lemma *leI*: $\neg\ x < y \Longrightarrow y \leq x$
unfolding *not-less* **.**

lemma *not-le-imp-less*: $\neg\ y \leq x \Longrightarrow x < y$
unfolding *not-le* **.**

lemma *linorder-less-wlog*[*case-names less refl sym*]:
$\llbracket \bigwedge a\ b.\ a < b \Longrightarrow P\ a\ b;\ \bigwedge a.\ P\ a\ a;\ \bigwedge a\ b.\ P\ b\ a \Longrightarrow P\ a\ b \rrbracket \Longrightarrow P\ a\ b$
using *antisym-conv3* **by** *blast*

Dual order

lemma *dual-linorder*:
class.linorder $(\geq)$ $(>)$
by (*rule class.linorder.intro*, *rule dual-order*) (*unfold-locales*, *rule linear*)

end

Alternative introduction rule with bias towards strict order

lemma *linorder-strictI*:
fixes *less-eq* (**infix** ‹$\leq$› *50*)
and *less* (**infix** ‹$<$› *50*)
assumes *class.order less-eq less*
assumes *trichotomy*: $\bigwedge a\ b.\ a < b \vee a = b \vee b < a$
shows *class.linorder less-eq less*
proof −
interpret *order less-eq less*
by (*fact* ‹*class.order less-eq less*›)
show *?thesis*
proof
fix *a b*
show $a \leq b \vee b \leq a$
using *trichotomy* **by** (*auto simp add*: *le-less*)
qed
qed

4.6 Reasoning tools setup

ML-file ‹~~/*src/Provers/order-procedure.ML*›
ML-file ‹~~/*src/Provers/order-tac.ML*›

ML ‹
```
structure Logic-Signature : LOGIC-SIGNATURE = struct
  val mk-Trueprop = HOLogic.mk-Trueprop
  val dest-Trueprop = HOLogic.dest-Trueprop
  val Trueprop-conv = HOLogic.Trueprop-conv
  val Not = HOLogic.Not
  val conj = HOLogic.conj
  val disj = HOLogic.disj

  val notI = @{thm notI}
  val ccontr = @{thm ccontr}
  val conjI = @{thm conjI}
  val conjE = @{thm conjE}
  val disjE = @{thm disjE}

  val not-not-conv = Conv.rewr-conv @{thm eq-reflection[OF not-not]}
 val de-Morgan-conj-conv = Conv.rewr-conv @{thm eq-reflection[OF de-Morgan-conj]}
 val de-Morgan-disj-conv = Conv.rewr-conv @{thm eq-reflection[OF de-Morgan-disj]}
 val conj-disj-distribL-conv = Conv.rewr-conv @{thm eq-reflection[OF conj-disj-distribL]}
 val conj-disj-distribR-conv = Conv.rewr-conv @{thm eq-reflection[OF conj-disj-distribR]}
end

structure HOL-Base-Order-Tac = Base-Order-Tac(
  structure Logic-Sig = Logic-Signature;
  (* Exclude types with specialised solvers. *)
  val excluded-types = [HOLogic.natT, HOLogic.intT, HOLogic.realT]
)

structure HOL-Order-Tac = Order-Tac(structure Base-Tac = HOL-Base-Order-Tac)

fun print-orders ctxt0 =
  let
    val ctxt = Config.put show-sorts true ctxt0
    val orders = HOL-Order-Tac.Data.get (Context.Proof ctxt)
    fun pretty-term t = Pretty.block
      [Pretty.quote (Syntax.pretty-term ctxt t), Pretty.brk 1,
        Pretty.str ::, Pretty.brk 1,
        Pretty.quote (Syntax.pretty-typ ctxt (type-of t)), Pretty.brk 1]
    fun pretty-order ({kind = kind, ops = ops, ...}, -) =
      Pretty.block ([Pretty.str (@{make-string} kind), Pretty.str :, Pretty.brk 1]
                  @ map pretty-term [#eq ops, #le ops, #lt ops])
  in
    Pretty.writeln (Pretty.big-list order structures: (map pretty-order orders))
  end

val - =
  Outer-Syntax.command command-keyword ‹print-orders›
    print order structures available to order reasoner
    (Scan.succeed (Toplevel.keep (print-orders o Toplevel.context-of)))
```

›

method-setup *order* = ‹
  Scan.succeed (*fn ctxt => SIMPLE-METHOD'* (*HOL-Order-Tac.tac* [] *ctxt*))
› *partial and linear order reasoner*

The method *order* allows one to use the order tactic located in `../Provers/order_tac.ML` in a standalone fashion.

The tactic rearranges the goal to prove *False*, then retrieves order literals of partial and linear orders (i.e. $x = y$, $x \leq y$, $x < y$, and their negated versions) from the premises and finally tries to derive a contradiction. Its main use case is as a solver to *simp* (see below), where it e.g. solves premises of conditional rewrite rules.

The tactic has two configuration attributes that control its behaviour:

- *order-trace* toggles tracing for the solver.
- *order-split-limit* limits the number of order literals of the form $\neg\ x < y$ that are passed to the tactic. This is helpful since these literals lead to case splitting and thus exponential runtime. This only applies to partial orders.

  We setup the solver for HOL with the structure `HOL_Order_Tac` here but the prover is agnostic to the object logic. It is possible to register orders with the prover using the functions `HOL_Order_Tac.declare_order` and `HOL_Order_Tac.declare_linorder`, which we do below for the type classes *order* and *linorder*. If possible, one should instantiate these type classes instead of registering new orders with the solver. One can also interpret the type class locales *order* and *linorder*. An example can be seen in `Library/Sublist.thy`, which contains e.g. the prefix order on lists.

  The diagnostic command **print-orders** shows all orders known to the tactic in the current context.

Declarations to set up transitivity reasoner of partial and linear orders.

context *order*
begin

lemma *nless-le*: $(\neg\ a < b) \longleftrightarrow (\neg\ a \leq b) \vee a = b$
  using *local.dual-order.order-iff-strict* **by** *blast*

local-setup ‹
  HOL-Order-Tac.declare-order {
   ops = {*eq* = @{*term* ‹(=) :: 'a ⇒ 'a ⇒ bool›}, *le* = @{*term* ‹(≤)›}, *lt* = @{*term* ‹(<)›}},

```
    thms = {trans = @{thm order-trans}, refl = @{thm order-refl}, eqD1 = @{thm
eq-refl},
          eqD2 = @{thm eq-refl[OF sym]}, antisym = @{thm order-antisym}, contr
= @{thm notE}},
     conv-thms = {less-le = @{thm eq-reflection[OF less-le]},
                  nless-le = @{thm eq-reflection[OF nless-le]}}
  }
›

end

context linorder
begin

lemma nle-le: (¬ a ≤ b) ⟷ b ≤ a ∧ b ≠ a
  using not-le less-le by simp

local-setup ‹
  HOL-Order-Tac.declare-linorder {
    ops = {eq = @{term ‹(=) :: 'a ⇒ 'a ⇒ bool›}, le = @{term ‹(≤)›}, lt = @{term
‹(<)›}},
    thms = {trans = @{thm order-trans}, refl = @{thm order-refl}, eqD1 = @{thm
eq-refl},
          eqD2 = @{thm eq-refl[OF sym]}, antisym = @{thm order-antisym}, contr
= @{thm notE}},
     conv-thms = {less-le = @{thm eq-reflection[OF less-le]},
                  nless-le = @{thm eq-reflection[OF not-less]},
                  nle-le = @{thm eq-reflection[OF nle-le]}}
  }
›

end

setup ‹
  Simplifier.map-theory-simpset (Simplifier.add-unsafe-solver
    (Simplifier.mk-solver partial and linear orders (fn ctxt => HOL-Order-Tac.tac
(Simplifier.prems-of ctxt) ctxt)))
›

ML ‹
local
  fun prp t thm = Thm.prop-of thm = t;  (* FIXME proper aconv!? *)
in

fun antisym-le-simproc ctxt ct =
  (case Thm.term-of ct of
    (le as Const (-, T)) $ r $ s =>
     (let
        val prems = Simplifier.prems-of ctxt;
```

```
      val less = Const (const-name‹less›, T);
      val t = HOLogic.mk-Trueprop(le $ s $ r);
    in
      (case find-first (prp t) prems of
        NONE =>
          let val t = HOLogic.mk-Trueprop(HOLogic.Not $ (less $ r $ s)) in
            (case find-first (prp t) prems of
              NONE => NONE
            | SOME thm => SOME(mk-meta-eq(thm RS @{thm antisym-conv1})))
           end
      | SOME thm => SOME (mk-meta-eq (thm RS @{thm order-class.antisym-conv})))
    end handle THM - => NONE)
  | - => NONE);

fun antisym-less-simproc ctxt ct =
  (case Thm.term-of ct of
    NotC $ ((less as Const(-,T)) $ r $ s) =>
     (let
       val prems = Simplifier.prems-of ctxt;
       val le = Const (const-name‹less-eq›, T);
       val t = HOLogic.mk-Trueprop(le $ r $ s);
      in
       (case find-first (prp t) prems of
         NONE =>
           let val t = HOLogic.mk-Trueprop (NotC $ (less $ s $ r)) in
             (case find-first (prp t) prems of
               NONE => NONE
         | SOME thm => SOME (mk-meta-eq(thm RS @{thm linorder-class.antisym-conv3})))
           end
       | SOME thm => SOME (mk-meta-eq (thm RS @{thm antisym-conv2})))
     end handle THM - => NONE)
  | - => NONE);

end;
›
```

simproc-setup *antisym-le* ((*x*::'*a*::*order*) ≤ *y*) = *K antisym-le-simproc*
simproc-setup *antisym-less* (¬ (*x*::'*a*::*linorder*) < *y*) = *K antisym-less-simproc*

4.7 Bounded quantifiers

syntax (*ASCII*)
-All-less :: [*idt*, '*a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder ALL››ALL -<-./ -)›* [*0*, *0*, *10*] *10*)
-Ex-less :: [*idt*, '*a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder EX››EX -<-./ -)›* [*0*, *0*, *10*] *10*)
-All-less-eq :: [*idt*, '*a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder ALL››ALL -<=-./ -)›* [*0*, *0*, *10*] *10*)
-Ex-less-eq :: [*idt*, '*a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder EX››EX*

-<=-./ -)› [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ALL›› ALL ->-./ -)› [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder EX›› EX ->-./ -)› [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ALL›› ALL ->=-./ -)› [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder EX›› EX ->=-./ -)› [0, 0, 10] 10)

-All-neq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ALL›› ALL -~=-./ -)› [0, 0, 10] 10)
-Ex-neq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder EX›› EX -~=-./ -)› [0, 0, 10] 10)

syntax
-All-less :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∀ ›› ∀ -<-./ -)› [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∃ ›› ∃ -<-./ -)› [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∀ ›› ∀ -≤-./ -)› [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∃ ›› ∃ -≤-./ -)› [0, 0, 10] 10)

-All-greater :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∀ ›› ∀ ->-./ -)› [0, 0, 10] 10)
-Ex-greater :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∃ ›› ∃ ->-./ -)› [0, 0, 10] 10)
-All-greater-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∀ ›› ∀ -≥-./ -)› [0, 0, 10] 10)
-Ex-greater-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∃ ›› ∃ -≥-./ -)› [0, 0, 10] 10)

-All-neq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∀ ›› ∀ -≠-./ -)› [0, 0, 10] 10)
-Ex-neq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ∃ ›› ∃ -≠-./ -)› [0, 0, 10] 10)

syntax (*input*)
-All-less :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder !›› ! -<-./ -)› [0, 0, 10] 10)
-Ex-less :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ?›› ? -<-./ -)› [0, 0, 10] 10)
-All-less-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder !›› ! -<=-./ -)› [0, 0, 10] 10)
-Ex-less-eq :: [idt, 'a, bool] => bool (‹(‹indent=3 notation=‹binder ?›› ? -<=-./ -)› [0, 0, 10] 10)

-All-neq :: [*idt*, *'a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder !››*! -~=-./ -)› [*0*, *0*, *10*] *10*)
-Ex-neq :: [*idt*, *'a*, *bool*] => *bool* (‹(‹*indent=3 notation=‹binder ?››*? -~=-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-All-less -All-less-eq -All-greater -All-greater-eq -All-neq ⇌ *All* **and**
-Ex-less -Ex-less-eq -Ex-greater -Ex-greater-eq -Ex-neq ⇌ *Ex*

translations
$\forall x{<}y.\ P \rightharpoonup \forall x.\ x < y \longrightarrow P$
$\exists x{<}y.\ P \rightharpoonup \exists x.\ x < y \wedge P$
$\forall x{\leq}y.\ P \rightharpoonup \forall x.\ x \leq y \longrightarrow P$
$\exists x{\leq}y.\ P \rightharpoonup \exists x.\ x \leq y \wedge P$
$\forall x{>}y.\ P \rightharpoonup \forall x.\ x > y \longrightarrow P$
$\exists x{>}y.\ P \rightharpoonup \exists x.\ x > y \wedge P$
$\forall x{\geq}y.\ P \rightharpoonup \forall x.\ x \geq y \longrightarrow P$
$\exists x{\geq}y.\ P \rightharpoonup \exists x.\ x \geq y \wedge P$
$\forall x{\neq}y.\ P \rightharpoonup \forall x.\ x \neq y \longrightarrow P$
$\exists x{\neq}y.\ P \rightharpoonup \exists x.\ x \neq y \wedge P$

print-translation ‹
let
val All-binder = Mixfix.binder-name ***const-syntax*** ‹*All*›;
val Ex-binder = Mixfix.binder-name ***const-syntax*** ‹*Ex*›;
val impl = ***const-syntax*** ‹*HOL.implies*›;
val conj = ***const-syntax*** ‹*HOL.conj*›;
val less = ***const-syntax*** ‹*less*›;
val less-eq = ***const-syntax*** ‹*less-eq*›;

val trans =
[((*All-binder*, *impl*, *less*),
(***syntax-const*** ‹*-All-less*›, ***syntax-const*** ‹*-All-greater*›)),
((*All-binder*, *impl*, *less-eq*),
(***syntax-const*** ‹*-All-less-eq*›, ***syntax-const*** ‹*-All-greater-eq*›)),
((*Ex-binder*, *conj*, *less*),
(***syntax-const*** ‹*-Ex-less*›, ***syntax-const*** ‹*-Ex-greater*›)),
((*Ex-binder*, *conj*, *less-eq*),
(***syntax-const*** ‹*-Ex-less-eq*›, ***syntax-const*** ‹*-Ex-greater-eq*›))];

fun matches-bound v t =
(*case t of*
Const (***syntax-const*** ‹*-bound*›, -) $ *Free* (*v'*, -) => *v* = *v'*
| - => *false*);
fun contains-var v = *Term.exists-subterm* (*fn Free* (*x*, -) => *x* = *v* | - => *false*);
fun mk x c n P = *Syntax.const c* $ *Syntax-Trans.mark-bound-body x* $ *n* $ *P*;

fun tr' q = (*q*, *fn* - =>
(*fn* [*Const* (***syntax-const*** ‹*-bound*›, -) $ *Free* (*v*, *T*),

```
      Const (c, -) $ (Const (d, -) $ t $ u) $ P] =>
      (case AList.lookup (=) trans (q, c, d) of
        NONE => raise Match
      | SOME (l, g) =>
          if matches-bound v t andalso not (contains-var v u) then mk (v, T) l u P
          else if matches-bound v u andalso not (contains-var v t) then mk (v, T)
g t P
          else raise Match)
    | - => raise Match));
in [tr' All-binder, tr' Ex-binder] end
›
```

4.8 Transitivity reasoning

context *ord*
begin

lemma *ord-le-eq-trans*: $a \leq b \Longrightarrow b = c \Longrightarrow a \leq c$
by (*rule subst*)

lemma *ord-eq-le-trans*: $a = b \Longrightarrow b \leq c \Longrightarrow a \leq c$
by (*rule ssubst*)

lemma *ord-less-eq-trans*: $a < b \Longrightarrow b = c \Longrightarrow a < c$
by (*rule subst*)

lemma *ord-eq-less-trans*: $a = b \Longrightarrow b < c \Longrightarrow a < c$
by (*rule ssubst*)

end

lemma *order-less-subst2*: $a < b \Longrightarrow f\ b < (c::'c::preorder) \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
by (*rule less-trans*)

lemma *order-less-subst1*: $(a::'a::preorder) < f\ b \Longrightarrow b < c \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
by (*rule less-trans*)

lemma *order-le-less-subst2*: $a \leq b \Longrightarrow f\ b < (c::'c::preorder) \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a < c$
by (*rule dual-order.strict-trans2*)

lemma *order-le-less-subst1*: $(a::'a::preorder) \leq f\ b \Longrightarrow b < c \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
by (*rule order.strict-trans1*)

lemma *order-less-le-subst2*: $(a::'a::ord) < b \Longrightarrow f\ b \leq (c::'c::preorder) \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$

by (*rule dual-order.strict-trans1*)

lemma *order-less-le-subst1*: $(a::'a::preorder) < f\ b \Longrightarrow b \leq c \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a < f\ c$
by (*rule order.strict-trans2*)

lemma *order-subst1*: $(a::'a::preorder) \leq f\ b \Longrightarrow b \leq c \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a \leq f\ c$
by (*rule order.trans*)

lemma *order-subst2*: $a \leq b \Longrightarrow f\ b \leq (c::'c::preorder) \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a \leq c$
by (*rule dual-order.trans*)

lemma *ord-le-eq-subst*: $a \leq b \Longrightarrow f\ b = c \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow f\ a \leq c$
by (*rule subst*)

lemma *ord-eq-le-subst*: $a = f\ b \Longrightarrow b \leq c \Longrightarrow$
$(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow a \leq f\ c$
by (*rule ssubst*)

lemma *ord-less-eq-subst*: $a < b \Longrightarrow f\ b = c \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow f\ a < c$
by (*rule subst*)

lemma *ord-eq-less-subst*: $a = f\ b \Longrightarrow b < c \Longrightarrow$
$(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow a < f\ c$
by (*rule ssubst*)

Note that this list of rules is in reverse order of priorities.

lemmas [*trans*] =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp

lemmas (**in** *order*) [*trans*] =
neq-le-trans
le-neq-trans

lemmas (**in** *preorder*) [*trans*] =
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans

lemmas (**in** *order*) [*trans*] =
order.antisym

lemmas (**in** *ord*) [*trans*] =
ord-le-eq-trans
ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans

lemmas [*trans*] =
trans

lemmas *order-trans-rules* =
order-less-subst2
order-less-subst1
order-le-less-subst2
order-le-less-subst1
order-less-le-subst2
order-less-le-subst1
order-subst2
order-subst1
ord-le-eq-subst
ord-eq-le-subst
ord-less-eq-subst
ord-eq-less-subst
forw-subst
back-subst
rev-mp
mp
neq-le-trans
le-neq-trans
less-trans
less-asym'
le-less-trans
less-le-trans
order-trans
order.antisym
ord-le-eq-trans

ord-eq-le-trans
ord-less-eq-trans
ord-eq-less-trans
trans

These support proving chains of decreasing inequalities a ≥ b ≥ c ... in Isar proofs.

lemma *xt1* [*no-atp*]:
$a = b \Longrightarrow b > c \Longrightarrow a > c$
$a > b \Longrightarrow b = c \Longrightarrow a > c$
$a = b \Longrightarrow b \geq c \Longrightarrow a \geq c$
$a \geq b \Longrightarrow b = c \Longrightarrow a \geq c$
$(x::'a::order) \geq y \Longrightarrow y \geq x \Longrightarrow x = y$
$(x::'a::order) \geq y \Longrightarrow y \geq z \Longrightarrow x \geq z$
$(x::'a::order) > y \Longrightarrow y \geq z \Longrightarrow x > z$
$(x::'a::order) \geq y \Longrightarrow y > z \Longrightarrow x > z$
$(a::'a::order) > b \Longrightarrow b > a \Longrightarrow P$
$(x::'a::order) > y \Longrightarrow y > z \Longrightarrow x > z$
$(a::'a::order) \geq b \Longrightarrow a \neq b \Longrightarrow a > b$
$(a::'a::order) \neq b \Longrightarrow a \geq b \Longrightarrow a > b$
$a = f\ b \Longrightarrow b > c \Longrightarrow (\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow a > f\ c$
$a > b \Longrightarrow f\ b = c \Longrightarrow (\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y) \Longrightarrow f\ a > c$
$a = f\ b \Longrightarrow b \geq c \Longrightarrow (\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y) \Longrightarrow a \geq f\ c$
$a \geq b \Longrightarrow f\ b = c \Longrightarrow (\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y) \Longrightarrow f\ a \geq c$
by *auto*

lemma *xt2* [*no-atp*]:
assumes $(a::'a::preorder) \geq f\ b$
and $b \geq c$
and $\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y$
shows $a \geq f\ c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-subst2*)

lemma *xt3* [*no-atp*]:
assumes $a \geq b$
and $(f\ b::'b::preorder) \geq c$
and $\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y$
shows $f\ a \geq c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-subst1*)

lemma *xt4* [*no-atp*]:
assumes $(a::'a::preorder) > f\ b$
and $b \geq c$
and $\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y$
shows $a > f\ c$
using *assms*(*2*,*1*,*3*) **by** (*rule Orderings.order-le-less-subst2*)

lemma *xt5* [*no-atp*]:
assumes $a > b$

and $(f\ b::'b::preorder) \geq c$
and $\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y$
shows $f\ a > c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-le-less-subst1*)

lemma *xt6* [*no-atp*]:
assumes $(a::'a::preorder) \geq f\ b$
and $b > c$
and $\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y$
shows $a > f\ c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-less-le-subst2*)

lemma *xt7* [*no-atp*]:
assumes $a \geq b$
and $(f\ b::'b::preorder) > c$
and $\bigwedge x\ y.\ x \geq y \Longrightarrow f\ x \geq f\ y$
shows $f\ a > c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-less-le-subst1*)

lemma *xt8* [*no-atp*]:
assumes $(a::'a::preorder) > f\ b$
and $b > c$
and $\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y$
shows $a > f\ c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-less-subst2*)

lemma *xt9* [*no-atp*]:
assumes $a > b$
and $(f\ b::'b::preorder) > c$
and $\bigwedge x\ y.\ x > y \Longrightarrow f\ x > f\ y$
shows $f\ a > c$
using *assms*(*2*,*1*,*3*) **by** (*rule order-less-subst1*)

lemmas *xtrans* = *xt1 xt2 xt3 xt4 xt5 xt6 xt7 xt8 xt9*

4.9 min and max – fundamental

definition (**in** *ord*) $min :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**
$min\ a\ b = (if\ a \leq b\ then\ a\ else\ b)$

definition (**in** *ord*) $max :: 'a \Rightarrow 'a \Rightarrow 'a$ **where**
$max\ a\ b = (if\ a \leq b\ then\ b\ else\ a)$

lemma *min-absorb1*: $x \leq y \Longrightarrow min\ x\ y = x$
by (*simp add*: *min-def*)

lemma *max-absorb2*: $x \leq y \Longrightarrow max\ x\ y = y$
by (*simp add*: *max-def*)

lemma *min-absorb2*: $(y::'a::order) \leq x \Longrightarrow min\ x\ y = y$
by (*simp add:min-def*)

lemma *max-absorb1*: $(y::'a::order) \leq x \Longrightarrow max\ x\ y = x$
by (*simp add*: *max-def*)

lemma *max-min-same* [*simp*]:
fixes $x\ y :: 'a :: linorder$
shows $max\ x\ (min\ x\ y) = x\ max\ (min\ x\ y)\ x = x\ max\ (min\ x\ y)\ y = y\ max\ y\ (min\ x\ y) = y$
by(*auto simp add*: *max-def min-def*)

4.10 (Unique) top and bottom elements

class *bot* =
fixes *bot* :: $'a$ ($\langle\bot\rangle$)

class *order-bot* = *order* + *bot* +
assumes *bot-least*: $\bot \leq a$
begin

sublocale *bot*: *ordering-top greater-eq greater bot*
by *standard* (*fact bot-least*)

lemma *le-bot*:
$a \leq \bot \Longrightarrow a = \bot$
by (*fact bot.extremum-uniqueI*)

lemma *bot-unique*:
$a \leq \bot \longleftrightarrow a = \bot$
by (*fact bot.extremum-unique*)

lemma *not-less-bot*:
$\neg\ a < \bot$
by (*fact bot.extremum-strict*)

lemma *bot-less*:
$a \neq \bot \longleftrightarrow \bot < a$
by (*fact bot.not-eq-extremum*)

lemma *max-bot*[*simp*]: $max\ bot\ x = x$
by(*simp add*: *max-def bot-unique*)

lemma *max-bot2*[*simp*]: $max\ x\ bot = x$
by(*simp add*: *max-def bot-unique*)

lemma *min-bot*[*simp*]: $min\ bot\ x = bot$
by(*simp add*: *min-def bot-unique*)

lemma *min-bot2*[*simp*]: *min x bot = bot*
by(*simp add*: *min-def bot-unique*)

end

class *top* =
  fixes *top* :: *'a* (‹⊤›)

class *order-top* = *order* + *top* +
  assumes *top-greatest*: $a \leq \top$
begin

sublocale *top*: *ordering-top less-eq less top*
  by *standard* (*fact top-greatest*)

lemma *top-le*:
  $\top \leq a \Longrightarrow a = \top$
  by (*fact top.extremum-uniqueI*)

lemma *top-unique*:
  $\top \leq a \longleftrightarrow a = \top$
  by (*fact top.extremum-unique*)

lemma *not-top-less*:
  $\neg \top < a$
  by (*fact top.extremum-strict*)

lemma *less-top*:
  $a \neq \top \longleftrightarrow a < \top$
  by (*fact top.not-eq-extremum*)

lemma *max-top*[*simp*]: *max top x = top*
by(*simp add*: *max-def top-unique*)

lemma *max-top2*[*simp*]: *max x top = top*
by(*simp add*: *max-def top-unique*)

lemma *min-top*[*simp*]: *min top x = x*
by(*simp add*: *min-def top-unique*)

lemma *min-top2*[*simp*]: *min x top = x*
by(*simp add*: *min-def top-unique*)

end

4.11 Dense orders

class *dense-order* = *order* +
  assumes *dense*: $x < y \Longrightarrow (\exists z.\ x < z \wedge z < y)$

class *dense-linorder* = *linorder* + *dense-order*
begin

lemma *dense-le*:
  fixes $y\ z :: 'a$
  assumes $\bigwedge x.\ x < y \Longrightarrow x \leq z$
  shows $y \leq z$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  hence $z < y$ **by** *simp*
  from *dense*[*OF this*]
  obtain x **where** $x < y$ **and** $z < x$ **by** *safe*
  moreover have $x \leq z$ **using** *assms*[*OF* ‹$x < y$›] **.**
  ultimately show *False* **by** *auto*
qed

lemma *dense-le-bounded*:
  fixes $x\ y\ z :: 'a$
  assumes $x < y$
  assumes $*$: $\bigwedge w.\ [\![\ x < w\ ;\ w < y\ ]\!] \Longrightarrow w \leq z$
  shows $y \leq z$
proof (*rule dense-le*)
  fix w **assume** $w < y$
  from *dense*[*OF* ‹$x < y$›] **obtain** u **where** $x < u\ u < y$ **by** *safe*
  from *linear*[*of u w*]
  show $w \leq z$
  proof (*rule disjE*)
    assume $u \leq w$
    from *less-le-trans*[*OF* ‹$x < u$› ‹$u \leq w$›] ‹$w < y$›
    show $w \leq z$ **by** (*rule* $*$)
  next
    assume $w \leq u$
    from ‹$w \leq u$› $*$[*OF* ‹$x < u$› ‹$u < y$›]
    show $w \leq z$ **by** (*rule order-trans*)
  qed
qed

lemma *dense-ge*:
  fixes $y\ z :: 'a$
  assumes $\bigwedge x.\ z < x \Longrightarrow y \leq x$
  shows $y \leq z$
proof (*rule ccontr*)
  assume ¬ *?thesis*
  hence $z < y$ **by** *simp*
  from *dense*[*OF this*]
  obtain x **where** $x < y$ **and** $z < x$ **by** *safe*
  moreover have $y \leq x$ **using** *assms*[*OF* ‹$z < x$›] **.**
  ultimately show *False* **by** *auto*

qed

lemma *dense-ge-bounded*:
fixes $x\ y\ z :: 'a$
assumes $z < x$
assumes $*$: $\bigwedge w.\ [\![\ z < w\ ;\ w < x\ ]\!] \Longrightarrow y \leq w$
shows $y \leq z$
proof (*rule dense-ge*)
fix w **assume** $z < w$
from *dense*[*OF* ‹$z < x$›] **obtain** u **where** $z < u\ u < x$ **by** *safe*
from *linear*[*of u w*]
show $y \leq w$
proof (*rule disjE*)
assume $w \leq u$
from ‹$z < w$› *le-less-trans*[*OF* ‹$w \leq u$› ‹$u < x$›]
show $y \leq w$ **by** (*rule* $*$)
next
assume $u \leq w$
from $*$[*OF* ‹$z < u$› ‹$u < x$›] ‹$u \leq w$›
show $y \leq w$ **by** (*rule order-trans*)
qed
qed

end

class *no-top* = *order* +
assumes *gt-ex*: $\exists\, y.\ x < y$

class *no-bot* = *order* +
assumes *lt-ex*: $\exists\, y.\ y < x$

class *unbounded-dense-linorder* = *dense-linorder* + *no-top* + *no-bot*

class *unbounded-dense-order* = *dense-order* + *no-top* + *no-bot*

instance *unbounded-dense-linorder* $\subseteq$ *unbounded-dense-order* **..**

4.12 Wellorders

class *wellorder* = *linorder* +
assumes *less-induct* [*case-names less*]: $(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$
begin

lemma *wellorder-Least-lemma*:
fixes $k :: 'a$
assumes $P\ k$
shows *LeastI*: $P\ (LEAST\ x.\ P\ x)$ **and** *Least-le*: $(LEAST\ x.\ P\ x) \leq k$
proof −

```
  have P (LEAST x. P x) ∧ (LEAST x. P x) ≤ k
  using assms proof (induct k rule: less-induct)
    case (less x) then have P x by simp
    show ?case proof (rule classical)
      assume assm: ¬ (P (LEAST a. P a) ∧ (LEAST a. P a) ≤ x)
      have ⋀y. P y ⟹ x ≤ y
      proof (rule classical)
        fix y
        assume P y and ¬ x ≤ y
        with less have P (LEAST a. P a) and (LEAST a. P a) ≤ y
          by (auto simp add: not-le)
        with assm have x < (LEAST a. P a) and (LEAST a. P a) ≤ y
          by auto
        then show x ≤ y by auto
      qed
      with ‹P x› have Least: (LEAST a. P a) = x
        by (rule Least-equality)
      with ‹P x› show ?thesis by simp
    qed
  qed
  then show P (LEAST x. P x) and (LEAST x. P x) ≤ k by auto
qed

— The following 3 lemmas are due to Brian Huffman
lemma LeastI-ex: ∃ x. P x ⟹ P (Least P)
  by (erule exE) (erule LeastI)

lemma LeastI2:
  P a ⟹ (⋀x. P x ⟹ Q x) ⟹ Q (Least P)
  by (blast intro: LeastI)

lemma LeastI2-ex:
  ∃ a. P a ⟹ (⋀x. P x ⟹ Q x) ⟹ Q (Least P)
  by (blast intro: LeastI-ex)

lemma LeastI2-wellorder:
  assumes P a
  and ⋀a. ⟦ P a; ∀ b. P b ⟶ a ≤ b ⟧ ⟹ Q a
  shows Q (Least P)
proof (rule LeastI2-order)
  show P (Least P) using ‹P a› by (rule LeastI)
next
  fix y assume P y thus Least P ≤ y by (rule Least-le)
next
  fix x assume P x ∀ y. P y ⟶ x ≤ y thus Q x by (rule assms(2))
qed

lemma LeastI2-wellorder-ex:
  assumes ∃ x. P x
```

and $\bigwedge a.\ [\![\ P\ a;\ \forall b.\ P\ b \longrightarrow a \leq b\ ]\!] \Longrightarrow Q\ a$
shows Q (*Least* P)
using *assms* **by** *clarify* (*blast intro*!: *LeastI2-wellorder*)

lemma *not-less-Least*: $k < (LEAST\ x.\ P\ x) \Longrightarrow \neg\ P\ k$
apply (*simp add*: *not-le* [*symmetric*])
apply (*erule contrapos-nn*)
apply (*erule Least-le*)
done

lemma *exists-least-iff*: $(\exists\, n.\ P\ n) \longleftrightarrow (\exists\, n.\ P\ n \wedge (\forall\, m < n.\ \neg\ P\ m))$ (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs* **thus** *?lhs* **by** *blast*
next
assume *H*: *?lhs* **then obtain** *n* **where** *n*: $P\ n$ **by** *blast*
let *?x* = *Least P*
{ fix *m* **assume** *m*: $m < ?x$
from *not-less-Least*[*OF m*] **have** $\neg\ P\ m$ **. }**
with *LeastI-ex*[*OF H*] **show** *?rhs* **by** *blast*
qed

lemma *exists-least-iff'*:
shows $(\exists\, n.\ P\ n) \longleftrightarrow P\ (Least\ P) \wedge (\forall\, m < (Least\ P).\ \neg\ P\ m)$
using *LeastI-ex not-less-Least* **by** *auto*

end

4.13 Order on *bool*

instantiation *bool* :: {*order-bot*, *order-top*, *linorder*}
begin

definition
le-bool-def [*simp*]: $P \leq Q \longleftrightarrow P \longrightarrow Q$

definition
[*simp*]: $(P{::}bool) < Q \longleftrightarrow \neg\ P \wedge Q$

definition
[*simp*]: $\bot \longleftrightarrow False$

definition
[*simp*]: $\top \longleftrightarrow True$

instance proof
qed *auto*

end

lemma *le-boolI*: $(P \Longrightarrow Q) \Longrightarrow P \leq Q$
by *simp*

lemma *le-boolI'*: $P \longrightarrow Q \Longrightarrow P \leq Q$
by *simp*

lemma *le-boolE*: $P \leq Q \Longrightarrow P \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by *simp*

lemma *le-boolD*: $P \leq Q \Longrightarrow P \longrightarrow Q$
by *simp*

lemma *bot-boolE*: $\bot \Longrightarrow P$
by *simp*

lemma *top-boolI*: $\top$
by *simp*

lemma [*code*]:
$False \leq b \longleftrightarrow True$
$True \leq b \longleftrightarrow b$
$False < b \longleftrightarrow b$
$True < b \longleftrightarrow False$
by *simp-all*

4.14 Order on - ⇒ -

instantiation *fun* :: (*type*, *ord*) *ord*
begin

definition
le-fun-def: $f \leq g \longleftrightarrow (\forall x.\ f\ x \leq g\ x)$

definition
$(f::'a \Rightarrow 'b) < g \longleftrightarrow f \leq g \wedge \neg\ (g \leq f)$

instance ..

end

instance *fun* :: (*type*, *preorder*) *preorder* **proof**
qed (*auto simp add*: *le-fun-def less-fun-def*
intro: *order-trans order.antisym*)

instance *fun* :: (*type*, *order*) *order* **proof**
qed (*auto simp add*: *le-fun-def intro*: *order.antisym*)

instantiation *fun* :: (*type*, *bot*) *bot*

begin

definition
$\bot = (\lambda x.\ \bot)$

instance ..

end

instantiation *fun* :: (*type*, *order-bot*) *order-bot*
begin

lemma *bot-apply* [*simp*, *code*]:
$\bot\ x = \bot$
by (*simp add*: *bot-fun-def*)

instance proof
qed (*simp add*: *le-fun-def*)

end

instantiation *fun* :: (*type*, *top*) *top*
begin

definition
[*no-atp*]: $\top = (\lambda x.\ \top)$

instance ..

end

instantiation *fun* :: (*type*, *order-top*) *order-top*
begin

lemma *top-apply* [*simp*, *code*]:
$\top\ x = \top$
by (*simp add*: *top-fun-def*)

instance proof
qed (*simp add*: *le-fun-def*)

end

lemma *le-funI*: $(\bigwedge x.\ f\ x \leq g\ x) \Longrightarrow f \leq g$
unfolding *le-fun-def* **by** *simp*

lemma *le-funE*: $f \leq g \Longrightarrow (f\ x \leq g\ x \Longrightarrow P) \Longrightarrow P$
unfolding *le-fun-def* **by** *simp*

lemma *le-funD*: $f \leq g \Longrightarrow f\ x \leq g\ x$
by (*rule le-funE*)

4.15 Order on unary and binary predicates

lemma *predicate1I*:
assumes *PQ*: $\bigwedge x.\ P\ x \Longrightarrow Q\ x$
shows $P \leq Q$
apply (*rule le-funI*)
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate1D*:
$P \leq Q \Longrightarrow P\ x \Longrightarrow Q\ x$
apply (*erule le-funE*)
apply (*erule le-boolE*)
apply *assumption*+
done

lemma *rev-predicate1D*:
$P\ x \Longrightarrow P \leq Q \Longrightarrow Q\ x$
by (*rule predicate1D*)

lemma *predicate2I*:
assumes *PQ*: $\bigwedge x\ y.\ P\ x\ y \Longrightarrow Q\ x\ y$
shows $P \leq Q$
apply (*rule le-funI*)+
apply (*rule le-boolI*)
apply (*rule PQ*)
apply *assumption*
done

lemma *predicate2D*:
$P \leq Q \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
apply (*erule le-funE*)+
apply (*erule le-boolE*)
apply *assumption*+
done

lemma *rev-predicate2D*:
$P\ x\ y \Longrightarrow P \leq Q \Longrightarrow Q\ x\ y$
by (*rule predicate2D*)

lemma *bot1E* [*no-atp*]: $\bot\ x \Longrightarrow P$
by (*simp add*: *bot-fun-def*)

lemma *bot2E*: $\bot\ x\ y \Longrightarrow P$

by (*simp add*: *bot-fun-def*)

lemma *top1I*: $\top\ x$
by (*simp add*: *top-fun-def*)

lemma *top2I*: $\top\ x\ y$
by (*simp add*: *top-fun-def*)

4.16 Name duplicates

lemmas *antisym* = *order.antisym*
lemmas *eq-iff* = *order.eq-iff*

lemmas *order-eq-refl* = *preorder-class.eq-refl*
lemmas *order-less-irrefl* = *preorder-class.less-irrefl*
lemmas *order-less-imp-le* = *preorder-class.less-imp-le*
lemmas *order-less-not-sym* = *preorder-class.less-not-sym*
lemmas *order-less-asym* = *preorder-class.less-asym*
lemmas *order-less-trans* = *preorder-class.less-trans*
lemmas *order-le-less-trans* = *preorder-class.le-less-trans*
lemmas *order-less-le-trans* = *preorder-class.less-le-trans*
lemmas *order-less-imp-not-less* = *preorder-class.less-imp-not-less*
lemmas *order-less-imp-triv* = *preorder-class.less-imp-triv*
lemmas *order-less-asym'* = *preorder-class.less-asym'*

lemmas *order-less-le* = *order-class.less-le*
lemmas *order-le-less* = *order-class.le-less*
lemmas *order-le-imp-less-or-eq* = *order-class.le-imp-less-or-eq*
lemmas *order-less-imp-not-eq* = *order-class.less-imp-not-eq*
lemmas *order-less-imp-not-eq2* = *order-class.less-imp-not-eq2*
lemmas *order-neq-le-trans* = *order-class.neq-le-trans*
lemmas *order-le-neq-trans* = *order-class.le-neq-trans*
lemmas *order-eq-iff* = *order-class.order.eq-iff*
lemmas *order-antisym-conv* = *order-class.antisym-conv*

lemmas *linorder-linear* = *linorder-class.linear*
lemmas *linorder-less-linear* = *linorder-class.less-linear*
lemmas *linorder-le-less-linear* = *linorder-class.le-less-linear*
lemmas *linorder-le-cases* = *linorder-class.le-cases*
lemmas *linorder-not-less* = *linorder-class.not-less*
lemmas *linorder-not-le* = *linorder-class.not-le*
lemmas *linorder-neq-iff* = *linorder-class.neq-iff*
lemmas *linorder-neqE* = *linorder-class.neqE*

end

5 Groups, also combined with orderings

theory *Groups*

imports *Orderings*
begin

5.1 Dynamic facts

named-theorems *ac-simps associativity and commutativity simplification rules*
and *algebra-simps algebra simplification rules for rings*
and *algebra-split-simps algebra simplification rules for rings, with potential goal splitting*
and *field-simps algebra simplification rules for fields*
and *field-split-simps algebra simplification rules for fields, with potential goal splitting*

The rewrites accumulated in *algebra-simps* deal with the classical algebraic structures of groups, rings and family. They simplify terms by multiplying everything out (in case of a ring) and bringing sums and products into a canonical form (by ordered rewriting). As a result it decides group and ring equalities but also helps with inequalities.

Of course it also works for fields, but it knows nothing about multiplicative inverses or division. This is catered for by *field-simps*.

Facts in *field-simps* multiply with denominators in (in)equations if they can be proved to be non-zero (for equations) or positive/negative (for inequalities). Can be too aggressive and is therefore separate from the more benign *algebra-simps*.

Collections *algebra-split-simps* and *field-split-simps* correspond to *algebra-simps* and *field-simps* but contain more aggresive rules that may lead to goal splitting.

5.2 Abstract structures

These locales provide basic structures for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale *semigroup* =
fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹∗› *70*)
assumes *assoc* [*ac-simps*]: $a * b * c = a * (b * c)$

locale *abel-semigroup* = *semigroup* +
assumes *commute* [*ac-simps*]: $a * b = b * a$
begin

lemma *left-commute* [*ac-simps*]: $b * (a * c) = a * (b * c)$
proof −
have $(b * a) * c = (a * b) * c$
by (*simp only*: *commute*)
then show *?thesis*

```
    by (simp only: assoc)
qed

end

locale monoid = semigroup +
  fixes z :: 'a (‹1›)
  assumes left-neutral [simp]: 1 * a = a
  assumes right-neutral [simp]: a * 1 = a

locale comm-monoid = abel-semigroup +
  fixes z :: 'a (‹1›)
  assumes comm-neutral: a * 1 = a
begin

sublocale monoid
  by standard (simp-all add: commute comm-neutral)

end

locale group = semigroup +
  fixes z :: 'a (‹1›)
  fixes inverse :: 'a ⇒ 'a
  assumes group-left-neutral: 1 * a = a
  assumes left-inverse [simp]:  inverse a * a = 1
begin

lemma left-cancel: a * b = a * c ⟷ b = c
proof
  assume a * b = a * c
  then have inverse a * (a * b) = inverse a * (a * c) by simp
  then have (inverse a * a) * b = (inverse a * a) * c
    by (simp only: assoc)
  then show b = c by (simp add: group-left-neutral)
qed simp

sublocale monoid
proof
  fix a
  have inverse a * a = 1 by simp
  then have inverse a * (a * 1) = inverse a * a
    by (simp add: group-left-neutral assoc [symmetric])
  with left-cancel show a * 1 = a
    by (simp only: left-cancel)
qed (fact group-left-neutral)

lemma inverse-unique:
  assumes a * b = 1
  shows inverse a = b
```

proof −
  from *assms* **have** *inverse a* ∗ (*a* ∗ *b*) = *inverse a*
    by *simp*
  then show *?thesis*
    by (*simp add*: *assoc* [*symmetric*])
qed

lemma *inverse-neutral* [*simp*]: *inverse* **1** = **1**
  by (*rule inverse-unique*) *simp*

lemma *inverse-inverse* [*simp*]: *inverse* (*inverse a*) = *a*
  by (*rule inverse-unique*) *simp*

lemma *right-inverse* [*simp*]: *a* ∗ *inverse a* = **1**
proof −
  have *a* ∗ *inverse a* = *inverse* (*inverse a*) ∗ *inverse a*
    by *simp*
  also have ... = **1**
    by (*rule left-inverse*)
  then show *?thesis* **by** *simp*
qed

lemma *inverse-distrib-swap*: *inverse* (*a* ∗ *b*) = *inverse b* ∗ *inverse a*
proof (*rule inverse-unique*)
  have *a* ∗ *b* ∗ (*inverse b* ∗ *inverse a*) =
    a ∗ (*b* ∗ *inverse b*) ∗ *inverse a*
    by (*simp only*: *assoc*)
  also have ... = **1**
    by *simp*
  finally show *a* ∗ *b* ∗ (*inverse b* ∗ *inverse a*) = **1** **.**
qed

lemma *right-cancel*: *b* ∗ *a* = *c* ∗ *a* ⟷ *b* = *c*
proof
  assume *b* ∗ *a* = *c* ∗ *a*
  then have *b* ∗ *a* ∗ *inverse a*= *c* ∗ *a* ∗ *inverse a*
    by *simp*
  then show *b* = *c*
    by (*simp add*: *assoc*)
qed *simp*

end

5.3 Generic operations

class *zero* =
  fixes *zero* :: *'a* (‹*0*›)

class *one* =

```
  fixes one :: 'a  (‹1›)

hide-const (open) zero one

lemma Let-0 [simp]: Let 0 f = f 0
  unfolding Let-def ..

lemma Let-1 [simp]: Let 1 f = f 1
  unfolding Let-def ..

setup ‹
  Reorient-Proc.add
    (fn Const(const-name‹Groups.zero›, -) => true
      | Const(const-name‹Groups.one›, -) => true
      | - => false)
›

simproc-setup reorient-zero (0 = x) = ‹K Reorient-Proc.proc›
simproc-setup reorient-one (1 = x) = ‹K Reorient-Proc.proc›

typed-print-translation ‹
  let
    fun tr' c = (c, fn ctxt => fn T => fn ts =>
      if null ts andalso Printer.type-emphasis ctxt T then
        Syntax.const syntax-const‹-constrain› $ Syntax.const c $
          Syntax-Phases.term-of-typ ctxt T
      else raise Match);
  in map tr' [const-syntax‹Groups.one›, const-syntax‹Groups.zero›] end
› — show types that are presumably too general

class plus =
  fixes plus :: 'a ⇒ 'a ⇒ 'a  (infixl ‹+› 65)

class minus =
  fixes minus :: 'a ⇒ 'a ⇒ 'a  (infixl ‹−› 65)

class uminus =
  fixes uminus :: 'a ⇒ 'a  (‹(‹open-block notation=‹prefix −››− -)› [81] 80)

class times =
  fixes times :: 'a ⇒ 'a ⇒ 'a  (infixl ‹*› 70)

bundle uminus-syntax
begin
notation uminus  (‹(‹open-block notation=‹prefix −››− -)› [81] 80)
end
```

5.4 Semigroups and Monoids

class *semigroup-add* = *plus* +
  assumes *add-assoc*: $(a + b) + c = a + (b + c)$
begin

sublocale *add*: *semigroup plus*
  by *standard* (*fact add-assoc*)

declare *add.assoc* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]

end

hide-fact *add-assoc*

class *ab-semigroup-add* = *semigroup-add* +
  assumes *add-commute*: $a + b = b + a$
begin

sublocale *add*: *abel-semigroup plus*
  by *standard* (*fact add-commute*)

declare *add.commute* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]
  add.left-commute [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]

lemmas *add-ac* = *add.assoc add.commute add.left-commute*

end

hide-fact *add-commute*

lemmas *add-ac* = *add.assoc add.commute add.left-commute*

class *semigroup-mult* = *times* +
  assumes *mult-assoc*: $(a * b) * c = a * (b * c)$
begin

sublocale *mult*: *semigroup times*
  by *standard* (*fact mult-assoc*)

declare *mult.assoc* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]

end

hide-fact *mult-assoc*

class *ab-semigroup-mult* = *semigroup-mult* +
  assumes *mult-commute*: $a * b = b * a$
begin

sublocale *mult*: *abel-semigroup times*
  by *standard* (*fact mult-commute*)

declare *mult.commute* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]
  mult.left-commute [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]

lemmas *mult-ac* = *mult.assoc mult.commute mult.left-commute*

end

hide-fact *mult-commute*

lemmas *mult-ac* = *mult.assoc mult.commute mult.left-commute*

class *monoid-add* = *zero* + *semigroup-add* +
  assumes *add-0-left*: *0* + *a* = *a*
    and *add-0-right*: *a* + *0* = *a*
begin

sublocale *add*: *monoid plus 0*
  by *standard* (*fact add-0-left add-0-right*)+

end

lemma *zero-reorient*: *0* = *x* ⟷ *x* = *0*
  by (*fact eq-commute*)

class *comm-monoid-add* = *zero* + *ab-semigroup-add* +
  assumes *add-0*: *0* + *a* = *a*
begin

subclass *monoid-add*
  by *standard* (*simp-all add*: *add-0 add.commute* [*of* - *0*])

sublocale *add*: *comm-monoid plus 0*
  by *standard* (*simp add*: *ac-simps*)

end

class *monoid-mult* = *one* + *semigroup-mult* +
  assumes *mult-1-left*: *1* ∗ *a* = *a*
    and *mult-1-right*: *a* ∗ *1* = *a*
begin

sublocale *mult*: *monoid times 1*
  by *standard* (*fact mult-1-left mult-1-right*)+

end

lemma *one-reorient*: $1 = x \longleftrightarrow x = 1$
by (*fact eq-commute*)

class *comm-monoid-mult* = *one* + *ab-semigroup-mult* +
assumes *mult-1*: $1 * a = a$
begin

subclass *monoid-mult*
by *standard* (*simp-all add*: *mult-1 mult.commute* [*of* - *1*])

sublocale *mult*: *comm-monoid times 1*
by *standard* (*simp add*: *ac-simps*)

end

class *cancel-semigroup-add* = *semigroup-add* +
assumes *add-left-imp-eq*: $a + b = a + c \Longrightarrow b = c$
assumes *add-right-imp-eq*: $b + a = c + a \Longrightarrow b = c$
begin

lemma *add-left-cancel* [*simp*]: $a + b = a + c \longleftrightarrow b = c$
by (*blast dest*: *add-left-imp-eq*)

lemma *add-right-cancel* [*simp*]: $b + a = c + a \longleftrightarrow b = c$
by (*blast dest*: *add-right-imp-eq*)

end

class *cancel-ab-semigroup-add* = *ab-semigroup-add* + *minus* +
assumes *add-diff-cancel-left'* [*simp*]: $(a + b) - a = b$
assumes *diff-diff-add* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a - b - c = a - (b + c)$
begin

lemma *add-diff-cancel-right'* [*simp*]: $(a + b) - b = a$
using *add-diff-cancel-left'* [*of b a*] **by** (*simp add*: *ac-simps*)

subclass *cancel-semigroup-add*
proof
fix $a\ b\ c :: 'a$
assume $a + b = a + c$
then have $a + b - a = a + c - a$
by *simp*
then show $b = c$
by *simp*
next
fix $a\ b\ c :: 'a$
assume $b + a = c + a$
then have $b + a - a = c + a - a$

```
    by simp
  then show b = c
    by simp
qed

lemma add-diff-cancel-left [simp]: (c + a) - (c + b) = a - b
  unfolding diff-diff-add [symmetric] by simp

lemma add-diff-cancel-right [simp]: (a + c) - (b + c) = a - b
  using add-diff-cancel-left [symmetric] by (simp add: ac-simps)

lemma diff-right-commute: a - c - b = a - b - c
  by (simp add: diff-diff-add add.commute)

end

class cancel-comm-monoid-add = cancel-ab-semigroup-add + comm-monoid-add
begin

lemma diff-zero [simp]: a - 0 = a
  using add-diff-cancel-right' [of a 0] by simp

lemma diff-cancel [simp]: a - a = 0
proof -
  have (a + 0) - (a + 0) = 0
    by (simp only: add-diff-cancel-left diff-zero)
  then show ?thesis by simp
qed

lemma add-implies-diff:
  assumes c + b = a
  shows c = a - b
proof -
  from assms have (b + c) - (b + 0) = a - b
    by (simp add: add.commute)
  then show c = a - b by simp
qed

lemma add-cancel-right-right [simp]: a = a + b ⟷ b = 0
  (is ?P ⟷ ?Q)
proof
  assume ?Q
  then show ?P by simp
next
  assume ?P
  then have a - a = a + b - a by simp
  then show ?Q by simp
qed
```

lemma *add-cancel-right-left* [*simp*]: $a = b + a \longleftrightarrow b = 0$
using *add-cancel-right-right* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *add-cancel-left-right* [*simp*]: $a + b = a \longleftrightarrow b = 0$
by (*auto dest*: *sym*)

lemma *add-cancel-left-left* [*simp*]: $b + a = a \longleftrightarrow b = 0$
by (*auto dest*: *sym*)

end

class *comm-monoid-diff* = *cancel-comm-monoid-add* +
assumes *zero-diff* [*simp*]: $0 - a = 0$
begin

lemma *diff-add-zero* [*simp*]: $a - (a + b) = 0$
proof −
have $a - (a + b) = (a + 0) - (a + b)$
by *simp*
also have $\ldots = 0$
by (*simp only*: *add-diff-cancel-left zero-diff*)
finally show *?thesis* **.**
qed

end

5.5 Groups

class *group-add* = *minus* + *uminus* + *monoid-add* +
assumes *left-minus*: $- a + a = 0$
assumes *add-uminus-conv-diff* [*simp*]: $a + (- b) = a - b$
begin

lemma *diff-conv-add-uminus*: $a - b = a + (- b)$
by *simp*

sublocale *add*: *group plus 0 uminus*
by *standard* (*simp-all add*: *left-minus*)

lemma *minus-unique*: $a + b = 0 \Longrightarrow - a = b$
by (*fact add.inverse-unique*)

lemma *minus-zero*: $- 0 = 0$
by (*fact add.inverse-neutral*)

lemma *minus-minus*: $- (- a) = a$
by (*fact add.inverse-inverse*)

lemma *right-minus*: $a + - a = 0$

by (*fact add.right-inverse*)

lemma *diff-self* [*simp*]: $a - a = 0$
using *right-minus* [*of a*] **by** *simp*

subclass *cancel-semigroup-add*
by *standard* (*simp-all add*: *add.left-cancel add.right-cancel*)

lemma *minus-add-cancel* [*simp*]: $- a + (a + b) = b$
by (*simp add*: *add.assoc* [*symmetric*])

lemma *add-minus-cancel* [*simp*]: $a + (- a + b) = b$
by (*simp add*: *add.assoc* [*symmetric*])

lemma *diff-add-cancel* [*simp*]: $a - b + b = a$
by (*simp only*: *diff-conv-add-uminus add.assoc*) *simp*

lemma *add-diff-cancel* [*simp*]: $a + b - b = a$
by (*simp only*: *diff-conv-add-uminus add.assoc*) *simp*

lemma *minus-add*: $- (a + b) = - b + - a$
by (*fact add.inverse-distrib-swap*)

lemma *right-minus-eq* [*simp*]: $a - b = 0 \longleftrightarrow a = b$
proof
assume $a - b = 0$
have $a = (a - b) + b$ **by** (*simp add*: *add.assoc*)
also have $\ldots = b$ **using** ‹$a - b = 0$› **by** *simp*
finally show $a = b$ **.**
next
assume $a = b$
then show $a - b = 0$ **by** *simp*
qed

lemma *eq-iff-diff-eq-0*: $a = b \longleftrightarrow a - b = 0$
by (*fact right-minus-eq* [*symmetric*])

lemma *diff-0* [*simp*]: $0 - a = - a$
by (*simp only*: *diff-conv-add-uminus add-0-left*)

lemma *diff-0-right* [*simp*]: $a - 0 = a$
by (*simp only*: *diff-conv-add-uminus minus-zero add-0-right*)

lemma *diff-minus-eq-add* [*simp*]: $a - - b = a + b$
by (*simp only*: *diff-conv-add-uminus minus-minus*)

lemma *neg-equal-iff-equal* [*simp*]: $- a = - b \longleftrightarrow a = b$
proof
assume $- a = - b$

then have $-(-a) = -(-b)$ **by** *simp*
then show $a = b$ **by** *simp*
next
assume $a = b$
then show $-a = -b$ **by** *simp*
qed

lemma *neg-equal-0-iff-equal* [*simp*]: $-a = 0 \longleftrightarrow a = 0$
by (*subst neg-equal-iff-equal* [*symmetric*]) *simp*

lemma *neg-0-equal-iff-equal* [*simp*]: $0 = -a \longleftrightarrow 0 = a$
by (*subst neg-equal-iff-equal* [*symmetric*]) *simp*

The next two equations can make the simplifier loop!

lemma *equation-minus-iff*: $a = -b \longleftrightarrow b = -a$
proof −
have $-(-a) = -b \longleftrightarrow -a = b$
by (*rule neg-equal-iff-equal*)
then show *?thesis*
by (*simp add*: *eq-commute*)
qed

lemma *minus-equation-iff*: $-a = b \longleftrightarrow -b = a$
proof −
have $-a = -(-b) \longleftrightarrow a = -b$
by (*rule neg-equal-iff-equal*)
then show *?thesis*
by (*simp add*: *eq-commute*)
qed

lemma *eq-neg-iff-add-eq-0*: $a = -b \longleftrightarrow a + b = 0$
proof
assume $a = -b$
then show $a + b = 0$ **by** *simp*
next
assume $a + b = 0$
moreover have $a + (b + -b) = (a + b) + -b$
by (*simp only*: *add.assoc*)
ultimately show $a = -b$
by *simp*
qed

lemma *add-eq-0-iff2*: $a + b = 0 \longleftrightarrow a = -b$
by (*fact eq-neg-iff-add-eq-0* [*symmetric*])

lemma *neg-eq-iff-add-eq-0*: $-a = b \longleftrightarrow a + b = 0$
by (*auto simp add*: *add-eq-0-iff2*)

lemma *add-eq-0-iff*: $a + b = 0 \longleftrightarrow b = -a$

by (*auto simp add*: *neg-eq-iff-add-eq-0* [*symmetric*])

lemma *minus-diff-eq* [*simp*]: $-(a - b) = b - a$
by (*simp only*: *neg-eq-iff-add-eq-0 diff-conv-add-uminus add.assoc minus-add-cancel*)
simp

lemma *add-diff-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a + (b - c) = (a + b) - c$
by (*simp only*: *diff-conv-add-uminus add.assoc*)

lemma *diff-add-eq-diff-diff-swap*: $a - (b + c) = a - c - b$
by (*simp only*: *diff-conv-add-uminus add.assoc minus-add*)

lemma *diff-eq-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a - b = c \longleftrightarrow a = c + b$
by *auto*

lemma *eq-diff-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a = c - b \longleftrightarrow a + b = c$
by *auto*

lemma *diff-diff-eq2* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a - (b - c) = (a + c) - b$
by (*simp only*: *diff-conv-add-uminus add.assoc*) *simp*

lemma *diff-eq-diff-eq*: $a - b = c - d \Longrightarrow a = b \longleftrightarrow c = d$
by (*simp only*: *eq-iff-diff-eq-0* [*of a b*] *eq-iff-diff-eq-0* [*of c d*])

end

class *ab-group-add* = *minus* + *uminus* + *comm-monoid-add* +
assumes *ab-left-minus*: $-a + a = 0$
assumes *ab-diff-conv-add-uminus*: $a - b = a + (-b)$
begin

subclass *group-add*
by *standard* (*simp-all add*: *ab-left-minus ab-diff-conv-add-uminus*)

subclass *cancel-comm-monoid-add*
proof
fix $a\ b\ c :: 'a$
have $b + a - a = b$
by *simp*
then show $a + b - a = b$
by (*simp add*: *ac-simps*)
show $a - b - c = a - (b + c)$
by (*simp add*: *algebra-simps*)
qed

lemma *uminus-add-conv-diff* [*simp*]: $- a + b = b - a$
by (*simp add*: *add.commute*)

lemma *minus-add-distrib* [*simp*]: $- (a + b) = - a + - b$
by (*simp add*: *algebra-simps*)

lemma *diff-add-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$(a - b) + c = (a + c) - b$
by (*simp add*: *algebra-simps*)

lemma *minus-diff-commute*:
$- b - a = - a - b$
by (*simp only*: *diff-conv-add-uminus add.commute*)

end

5.6 (Partially) Ordered Groups

The theory of partially ordered groups is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

class *ordered-ab-semigroup-add* = *order* + *ab-semigroup-add* +
assumes *add-left-mono*: $a \leq b \Longrightarrow c + a \leq c + b$
begin

lemma *add-right-mono*: $a \leq b \Longrightarrow a + c \leq b + c$
by (*simp add*: *add.commute* [*of* - *c*] *add-left-mono*)

non-strict, in both arguments

lemma *add-mono*: $a \leq b \Longrightarrow c \leq d \Longrightarrow a + c \leq b + d$
by (*simp add*: *add.commute add-left-mono add-right-mono* [*THEN order-trans*])

end

Strict monotonicity in both arguments

class *strict-ordered-ab-semigroup-add* = *ordered-ab-semigroup-add* +
assumes *add-strict-mono*: $a < b \Longrightarrow c < d \Longrightarrow a + c < b + d$

class *ordered-cancel-ab-semigroup-add* =
ordered-ab-semigroup-add + *cancel-ab-semigroup-add*
begin

lemma *add-strict-left-mono*: $a < b \Longrightarrow c + a < c + b$
by (*auto simp add*: *less-le add-left-mono*)

lemma *add-strict-right-mono*: $a < b \Longrightarrow a + c < b + c$
by (*simp add*: *add.commute* [*of* - *c*] *add-strict-left-mono*)

subclass *strict-ordered-ab-semigroup-add*
proof
show $\bigwedge a\ b\ c\ d.\ [\![a < b;\ c < d]\!] \Longrightarrow a + c < b + d$
by (*iprover intro*: *add-strict-left-mono add-strict-right-mono less-trans*)
qed

lemma *add-less-le-mono*: $a < b \Longrightarrow c \leq d \Longrightarrow a + c < b + d$
by (*iprover intro*: *add-left-mono add-strict-right-mono less-le-trans*)

lemma *add-le-less-mono*: $a \leq b \Longrightarrow c < d \Longrightarrow a + c < b + d$
by (*iprover intro*: *add-strict-left-mono add-right-mono less-le-trans*)

end

class *ordered-ab-semigroup-add-imp-le* = *ordered-cancel-ab-semigroup-add* +
assumes *add-le-imp-le-left*: $c + a \leq c + b \Longrightarrow a \leq b$
begin

lemma *add-less-imp-less-left*:
assumes *less*: $c + a < c + b$
shows $a < b$
proof −
from *less* **have** *le*: $c + a \leq c + b$
by (*simp add*: *order-le-less*)
have $a \leq b$
using *add-le-imp-le-left* [*OF le*] **.**
moreover have $a \neq b$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have $a = b$ **by** *simp*
then have $c + a = c + b$ **by** *simp*
with *less* **show** *False* **by** *simp*
qed
ultimately show $a < b$
by (*simp add*: *order-le-less*)
qed

lemma *add-less-imp-less-right*: $a + c < b + c \Longrightarrow a < b$
by (*rule add-less-imp-less-left* [*of c*]) (*simp add*: *add.commute*)

lemma *add-less-cancel-left* [*simp*]: $c + a < c + b \longleftrightarrow a < b$
by (*blast intro*: *add-less-imp-less-left add-strict-left-mono*)

lemma *add-less-cancel-right* [*simp*]: $a + c < b + c \longleftrightarrow a < b$
by (*blast intro*: *add-less-imp-less-right add-strict-right-mono*)

lemma *add-le-cancel-left* [*simp*]: $c + a \leq c + b \longleftrightarrow a \leq b$
by (*auto simp*: *dest*: *add-le-imp-le-left add-left-mono*)

lemma *add-le-cancel-right* [*simp*]: $a + c \leq b + c \longleftrightarrow a \leq b$
by (*simp add*: *add.commute* [*of a c*] *add.commute* [*of b c*])

lemma *add-le-imp-le-right*: $a + c \leq b + c \Longrightarrow a \leq b$
by *simp*

lemma *max-add-distrib-left*: $max\ x\ y + z = max\ (x + z)\ (y + z)$
unfolding *max-def* **by** *auto*

lemma *min-add-distrib-left*: $min\ x\ y + z = min\ (x + z)\ (y + z)$
unfolding *min-def* **by** *auto*

lemma *max-add-distrib-right*: $x + max\ y\ z = max\ (x + y)\ (x + z)$
unfolding *max-def* **by** *auto*

lemma *min-add-distrib-right*: $x + min\ y\ z = min\ (x + y)\ (x + z)$
unfolding *min-def* **by** *auto*

end

5.7 Support for reasoning about signs

class *ordered-comm-monoid-add* = *comm-monoid-add* + *ordered-ab-semigroup-add*
begin

lemma *add-nonneg-nonneg* [*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a + b$
using *add-mono*[*of 0 a 0 b*] **by** *simp*

lemma *add-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow a + b \leq 0$
using *add-mono*[*of a 0 b 0*] **by** *simp*

lemma *add-nonneg-eq-0-iff*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
using *add-left-mono*[*of 0 y x*] *add-right-mono*[*of 0 x y*] **by** *auto*

lemma *add-nonpos-eq-0-iff*: $x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
using *add-left-mono*[*of y 0 x*] *add-right-mono*[*of x 0 y*] **by** *auto*

lemma *add-increasing*: $0 \leq a \Longrightarrow b \leq c \Longrightarrow b \leq a + c$
using *add-mono* [*of 0 a b c*] **by** *simp*

lemma *add-increasing2*: $0 \leq c \Longrightarrow b \leq a \Longrightarrow b \leq a + c$
by (*simp add*: *add-increasing add.commute* [*of a*])

lemma *add-decreasing*: $a \leq 0 \Longrightarrow c \leq b \Longrightarrow a + c \leq b$
using *add-mono* [*of a 0 c b*] **by** *simp*

lemma *add-decreasing2*: $c \leq 0 \Longrightarrow a \leq b \Longrightarrow a + c \leq b$
using *add-mono*[*of a b c 0*] **by** *simp*

lemma *add-pos-nonneg*: $0 < a \Longrightarrow 0 \leq b \Longrightarrow 0 < a + b$
using *less-le-trans*[*of 0 a a + b*] **by** (*simp add*: *add-increasing2*)

lemma *add-pos-pos*: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a + b$
by (*intro add-pos-nonneg less-imp-le*)

lemma *add-nonneg-pos*: $0 \leq a \Longrightarrow 0 < b \Longrightarrow 0 < a + b$
using *add-pos-nonneg*[*of b a*] **by** (*simp add*: *add-commute*)

lemma *add-neg-nonpos*: $a < 0 \Longrightarrow b \leq 0 \Longrightarrow a + b < 0$
using *le-less-trans*[*of a + b a 0*] **by** (*simp add*: *add-decreasing2*)

lemma *add-neg-neg*: $a < 0 \Longrightarrow b < 0 \Longrightarrow a + b < 0$
by (*intro add-neg-nonpos less-imp-le*)

lemma *add-nonpos-neg*: $a \leq 0 \Longrightarrow b < 0 \Longrightarrow a + b < 0$
using *add-neg-nonpos*[*of b a*] **by** (*simp add*: *add-commute*)

lemmas *add-sign-intros* =
add-pos-nonneg add-pos-pos add-nonneg-pos add-nonneg-nonneg
add-neg-nonpos add-neg-neg add-nonpos-neg add-nonpos-nonpos

end

class *strict-ordered-comm-monoid-add* = *comm-monoid-add* + *strict-ordered-ab-semigroup-add*
begin

lemma *pos-add-strict*: $0 < a \Longrightarrow b < c \Longrightarrow b < a + c$
using *add-strict-mono* [*of 0 a b c*] **by** *simp*

end

class *ordered-cancel-comm-monoid-add* = *ordered-comm-monoid-add* + *cancel-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* **..**
subclass *strict-ordered-comm-monoid-add* **..**

lemma *add-strict-increasing*: $0 < a \Longrightarrow b \leq c \Longrightarrow b < a + c$

using *add-less-le-mono* [*of 0 a b c*] **by** *simp*

lemma *add-strict-increasing2*: $0 \leq a \Longrightarrow b < c \Longrightarrow b < a + c$
using *add-le-less-mono* [*of 0 a b c*] **by** *simp*

end

class *ordered-ab-semigroup-monoid-add-imp-le* = *monoid-add* + *ordered-ab-semigroup-add-imp-le*
begin

lemma *add-less-same-cancel1* [*simp*]: $b + a < b \longleftrightarrow a < 0$
using *add-less-cancel-left* [*of - - 0*] **by** *simp*

lemma *add-less-same-cancel2* [*simp*]: $a + b < b \longleftrightarrow a < 0$
using *add-less-cancel-right* [*of - - 0*] **by** *simp*

lemma *less-add-same-cancel1* [*simp*]: $a < a + b \longleftrightarrow 0 < b$
using *add-less-cancel-left* [*of - 0*] **by** *simp*

lemma *less-add-same-cancel2* [*simp*]: $a < b + a \longleftrightarrow 0 < b$
using *add-less-cancel-right* [*of 0*] **by** *simp*

lemma *add-le-same-cancel1* [*simp*]: $b + a \leq b \longleftrightarrow a \leq 0$
using *add-le-cancel-left* [*of - - 0*] **by** *simp*

lemma *add-le-same-cancel2* [*simp*]: $a + b \leq b \longleftrightarrow a \leq 0$
using *add-le-cancel-right* [*of - - 0*] **by** *simp*

lemma *le-add-same-cancel1* [*simp*]: $a \leq a + b \longleftrightarrow 0 \leq b$
using *add-le-cancel-left* [*of - 0*] **by** *simp*

lemma *le-add-same-cancel2* [*simp*]: $a \leq b + a \longleftrightarrow 0 \leq b$
using *add-le-cancel-right* [*of 0*] **by** *simp*

subclass *cancel-comm-monoid-add*
by *standard auto*

subclass *ordered-cancel-comm-monoid-add*
by *standard*

end

class *ordered-ab-group-add* = *ab-group-add* + *ordered-ab-semigroup-add*
begin

subclass *ordered-cancel-ab-semigroup-add* **..**

subclass *ordered-ab-semigroup-monoid-add-imp-le*
proof

fix $a\ b\ c :: 'a$
assume $c + a \leq c + b$
then have $(-c) + (c + a) \leq (-c) + (c + b)$
by (*rule add-left-mono*)
then have $((-c) + c) + a \leq ((-c) + c) + b$
by (*simp only*: *add.assoc*)
then show $a \leq b$ **by** *simp*
qed

lemma *max-diff-distrib-left*: $max\ x\ y - z = max\ (x - z)\ (y - z)$
using *max-add-distrib-left* [*of* $x\ y - z$] **by** *simp*

lemma *min-diff-distrib-left*: $min\ x\ y - z = min\ (x - z)\ (y - z)$
using *min-add-distrib-left* [*of* $x\ y - z$] **by** *simp*

lemma *le-imp-neg-le*:
assumes $a \leq b$
shows $- b \leq - a$
proof −
from *assms* **have** $- a + a \leq - a + b$
by (*rule add-left-mono*)
then have $0 \leq - a + b$
by *simp*
then have $0 + (- b) \leq (- a + b) + (- b)$
by (*rule add-right-mono*)
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

lemma *neg-le-iff-le* [*simp*]: $- b \leq - a \longleftrightarrow a \leq b$
proof
assume $- b \leq - a$
then have $- (- a) \leq - (- b)$
by (*rule le-imp-neg-le*)
then show $a \leq b$
by *simp*
next
assume $a \leq b$
then show $- b \leq - a$
by (*rule le-imp-neg-le*)
qed

lemma *neg-le-0-iff-le* [*simp*]: $- a \leq 0 \longleftrightarrow 0 \leq a$
by (*subst neg-le-iff-le* [*symmetric*]) *simp*

lemma *neg-0-le-iff-le* [*simp*]: $0 \leq - a \longleftrightarrow a \leq 0$
by (*subst neg-le-iff-le* [*symmetric*]) *simp*

lemma *neg-less-iff-less* [*simp*]: $- b < - a \longleftrightarrow a < b$

by (*auto simp add*: *less-le*)

lemma *neg-less-0-iff-less* [*simp*]: $- a < 0 \longleftrightarrow 0 < a$
by (*subst neg-less-iff-less* [*symmetric*]) *simp*

lemma *neg-0-less-iff-less* [*simp*]: $0 < - a \longleftrightarrow a < 0$
by (*subst neg-less-iff-less* [*symmetric*]) *simp*

The next several equations can make the simplifier loop!

lemma *less-minus-iff*: $a < - b \longleftrightarrow b < - a$
proof −
have $- (- a) < - b \longleftrightarrow b < - a$
by (*rule neg-less-iff-less*)
then show *?thesis* **by** *simp*
qed

lemma *minus-less-iff*: $- a < b \longleftrightarrow - b < a$
proof −
have $- a < - (- b) \longleftrightarrow - b < a$
by (*rule neg-less-iff-less*)
then show *?thesis* **by** *simp*
qed

lemma *le-minus-iff*: $a \leq - b \longleftrightarrow b \leq - a$
by (*auto simp*: *order.order-iff-strict less-minus-iff*)

lemma *minus-le-iff*: $- a \leq b \longleftrightarrow - b \leq a$
by (*auto simp add*: *le-less minus-less-iff*)

lemma *diff-less-0-iff-less* [*simp*]: $a - b < 0 \longleftrightarrow a < b$
proof −
have $a - b < 0 \longleftrightarrow a + (- b) < b + (- b)$
by *simp*
also have $\ldots \longleftrightarrow a < b$
by (*simp only*: *add-less-cancel-right*)
finally show *?thesis* **.**
qed

lemmas *less-iff-diff-less-0* = *diff-less-0-iff-less* [*symmetric*]

lemma *diff-less-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a - b < c \longleftrightarrow a < c + b$
proof (*subst less-iff-diff-less-0* [*of a*])
show $(a - b < c) = (a - (c + b) < 0)$
by (*simp add*: *algebra-simps less-iff-diff-less-0* [*of* - *c*])
qed

lemma *less-diff-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
$a < c - b \longleftrightarrow a + b < c$

proof (*subst less-iff-diff-less-0* [*of* $a + b$])
  show $(a < c - b) = (a + b - c < 0)$
    by (*simp add*: *algebra-simps less-iff-diff-less-0* [*of* a])
qed

lemma *diff-gt-0-iff-gt* [*simp*]: $a - b > 0 \longleftrightarrow a > b$
  by (*simp add*: *less-diff-eq*)

lemma *diff-le-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
  $a - b \leq c \longleftrightarrow a \leq c + b$
  by (*auto simp add*: *le-less diff-less-eq*)

lemma *le-diff-eq* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*]:
  $a \leq c - b \longleftrightarrow a + b \leq c$
  by (*auto simp add*: *le-less less-diff-eq*)

lemma *diff-le-0-iff-le* [*simp*]: $a - b \leq 0 \longleftrightarrow a \leq b$
  by (*simp add*: *algebra-simps*)

lemmas *le-iff-diff-le-0* $=$ *diff-le-0-iff-le* [*symmetric*]

lemma *diff-ge-0-iff-ge* [*simp*]: $a - b \geq 0 \longleftrightarrow a \geq b$
  by (*simp add*: *le-diff-eq*)

lemma *diff-eq-diff-less*: $a - b = c - d \Longrightarrow a < b \longleftrightarrow c < d$
  by (*auto simp only*: *less-iff-diff-less-0* [*of* a b] *less-iff-diff-less-0* [*of* c d])

lemma *diff-eq-diff-less-eq*: $a - b = c - d \Longrightarrow a \leq b \longleftrightarrow c \leq d$
  by (*auto simp only*: *le-iff-diff-le-0* [*of* a b] *le-iff-diff-le-0* [*of* c d])

lemma *diff-mono*: $a \leq b \Longrightarrow d \leq c \Longrightarrow a - c \leq b - d$
  by (*simp add*: *field-simps add-mono*)

lemma *diff-left-mono*: $b \leq a \Longrightarrow c - a \leq c - b$
  by (*simp add*: *field-simps*)

lemma *diff-right-mono*: $a \leq b \Longrightarrow a - c \leq b - c$
  by (*simp add*: *field-simps*)

lemma *diff-strict-mono*: $a < b \Longrightarrow d < c \Longrightarrow a - c < b - d$
  by (*simp add*: *field-simps add-strict-mono*)

lemma *diff-strict-left-mono*: $b < a \Longrightarrow c - a < c - b$
  by (*simp add*: *field-simps*)

lemma *diff-strict-right-mono*: $a < b \Longrightarrow a - c < b - c$
  by (*simp add*: *field-simps*)

end

locale *group-cancel*
begin

lemma *add1*: $(A::'a::comm\text{-}monoid\text{-}add) \equiv k + a \Longrightarrow A + b \equiv k + (a + b)$
by (*simp only*: *ac-simps*)

lemma *add2*: $(B::'a::comm\text{-}monoid\text{-}add) \equiv k + b \Longrightarrow a + B \equiv k + (a + b)$
by (*simp only*: *ac-simps*)

lemma *sub1*: $(A::'a::ab\text{-}group\text{-}add) \equiv k + a \Longrightarrow A - b \equiv k + (a - b)$
by (*simp only*: *add-diff-eq*)

lemma *sub2*: $(B::'a::ab\text{-}group\text{-}add) \equiv k + b \Longrightarrow a - B \equiv - k + (a - b)$
by (*simp only*: *minus-add diff-conv-add-uminus ac-simps*)

lemma *neg1*: $(A::'a::ab\text{-}group\text{-}add) \equiv k + a \Longrightarrow - A \equiv - k + - a$
by (*simp only*: *minus-add-distrib*)

lemma *rule0*: $(a::'a::comm\text{-}monoid\text{-}add) \equiv a + 0$
by (*simp only*: *add-0-right*)

end

ML-file ‹*Tools/group-cancel.ML*›

simproc-setup *group-cancel-add* $(a + b::'a::ab\text{-}group\text{-}add) =$
‹*fn phi => fn ss => try Group-Cancel.cancel-add-conv*›

simproc-setup *group-cancel-diff* $(a - b::'a::ab\text{-}group\text{-}add) =$
‹*fn phi => fn ss => try Group-Cancel.cancel-diff-conv*›

simproc-setup *group-cancel-eq* $(a = (b::'a::ab\text{-}group\text{-}add)) =$
‹*fn phi => fn ss => try Group-Cancel.cancel-eq-conv*›

simproc-setup *group-cancel-le* $(a \leq (b::'a::ordered\text{-}ab\text{-}group\text{-}add)) =$
‹*fn phi => fn ss => try Group-Cancel.cancel-le-conv*›

simproc-setup *group-cancel-less* $(a < (b::'a::ordered\text{-}ab\text{-}group\text{-}add)) =$
‹*fn phi => fn ss => try Group-Cancel.cancel-less-conv*›

class *linordered-ab-semigroup-add* =
linorder + *ordered-ab-semigroup-add*

class *linordered-cancel-ab-semigroup-add* =
linorder + *ordered-cancel-ab-semigroup-add*
begin

subclass *linordered-ab-semigroup-add* **..**

subclass *ordered-ab-semigroup-add-imp-le*
proof
  fix a b c :: $'a$
  assume *le1*: $c + a \leq c + b$
  show $a \leq b$
  proof (*rule ccontr*)
    assume $*$: $\neg$ *?thesis*
    then have $b \leq a$ **by** (*simp add*: *linorder-not-le*)
    then have $c + b \leq c + a$ **by** (*rule add-left-mono*)
    then have $c + a = c + b$
      using *le1* **by** (*iprover intro*: *order.antisym*)
    then have $a = b$
      by *simp*
    with $*$ **show** *False*
      by (*simp add*: *linorder-not-le* [*symmetric*])
  qed
qed

end

class *linordered-ab-group-add* = *linorder* + *ordered-ab-group-add*
begin

subclass *linordered-cancel-ab-semigroup-add* **..**

lemma *equal-neg-zero* [*simp*]: $a = - a \longleftrightarrow a = 0$
proof
  assume $a = 0$
  then show $a = - a$ **by** *simp*
next
  assume *A*: $a = - a$
  show $a = 0$
  proof (*cases* $0 \leq a$)
    case *True*
    with *A* **have** $0 \leq - a$ **by** *auto*
    with *le-minus-iff* **have** $a \leq 0$ **by** *simp*
    with *True* **show** *?thesis* **by** (*auto intro*: *order-trans*)
  next
    case *False*
    then have *B*: $a \leq 0$ **by** *auto*
    with *A* **have** $- a \leq 0$ **by** *auto*
    with *B* **show** *?thesis* **by** (*auto intro*: *order-trans*)
  qed
qed

lemma *neg-equal-zero* [*simp*]: $- a = a \longleftrightarrow a = 0$
  by (*auto dest*: *sym*)

lemma *neg-less-eq-nonneg* [*simp*]: $- a \leq a \longleftrightarrow 0 \leq a$
proof
  assume $*$: $- a \leq a$
  show $0 \leq a$
  proof (*rule classical*)
    assume $\neg$ *?thesis*
    then have $a < 0$ **by** *auto*
    with $*$ **have** $- a < 0$ **by** (*rule le-less-trans*)
    then show *?thesis* **by** *auto*
  qed
next
  assume $*$: $0 \leq a$
  then have $- a \leq 0$ **by** (*simp add*: *minus-le-iff*)
  from *this* $*$ **show** $- a \leq a$ **by** (*rule order-trans*)
qed

lemma *neg-less-pos* [*simp*]: $- a < a \longleftrightarrow 0 < a$
  by (*auto simp add*: *less-le*)

lemma *less-eq-neg-nonpos* [*simp*]: $a \leq - a \longleftrightarrow a \leq 0$
  using *neg-less-eq-nonneg* [*of* $- a$] **by** *simp*

lemma *less-neg-neg* [*simp*]: $a < - a \longleftrightarrow a < 0$
  using *neg-less-pos* [*of* $- a$] **by** *simp*

lemma *double-zero* [*simp*]: $a + a = 0 \longleftrightarrow a = 0$
proof
  assume $a + a = 0$
  then have a: $- a = a$ **by** (*rule minus-unique*)
  then show $a = 0$ **by** (*simp only*: *neg-equal-zero*)
next
  assume $a = 0$
  then show $a + a = 0$ **by** *simp*
qed

lemma *double-zero-sym* [*simp*]: $0 = a + a \longleftrightarrow a = 0$
  using *double-zero* [*of a*] **by** (*simp only*: *eq-commute*)

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: $0 < a + a \longleftrightarrow 0 < a$
proof
  assume $0 < a + a$
  then have $0 - a < a$ **by** (*simp only*: *diff-less-eq*)
  then have $- a < a$ **by** *simp*
  then show $0 < a$ **by** *simp*
next
  assume $0 < a$
  with *this* **have** $0 + 0 < a + a$
    by (*rule add-strict-mono*)
  then show $0 < a + a$ **by** *simp*

qed

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$
by (*auto simp add*: *le-less*)

lemma *double-add-less-zero-iff-single-add-less-zero* [*simp*]: $a + a < 0 \longleftrightarrow a < 0$
proof −
have $\neg\ a + a < 0 \longleftrightarrow \neg\ a < 0$
by (*simp add*: *not-less*)
then show *?thesis* **by** *simp*
qed

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: $a + a \leq 0 \longleftrightarrow a \leq 0$
proof −
have $\neg\ a + a \leq 0 \longleftrightarrow \neg\ a \leq 0$
by (*simp add*: *not-le*)
then show *?thesis* **by** *simp*
qed

lemma *minus-max-eq-min*: $-\ max\ x\ y = min\ (-\ x)\ (-\ y)$
by (*auto simp add*: *max-def min-def*)

lemma *minus-min-eq-max*: $-\ min\ x\ y = max\ (-\ x)\ (-\ y)$
by (*auto simp add*: *max-def min-def*)

end

class *abs* =
fixes *abs* :: $'a \Rightarrow 'a$ (‹(‹*open-block notation*=‹*mixfix abs*››|-|)›)

bundle *abs-syntax*
begin
notation *abs* (‹(‹*open-block notation*=‹*mixfix abs*››|-|)›)
end

class *sgn* =
fixes *sgn* :: $'a \Rightarrow 'a$

class *ordered-ab-group-add-abs* = *ordered-ab-group-add* + *abs* +
assumes *abs-ge-zero* [*simp*]: $|a| \geq 0$
and *abs-ge-self*: $a \leq |a|$
and *abs-leI*: $a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$
and *abs-minus-cancel* [*simp*]: $|-a| = |a|$
and *abs-triangle-ineq*: $|a + b| \leq |a| + |b|$
begin

lemma *abs-minus-le-zero*: $-\ |a| \leq 0$
unfolding *neg-le-0-iff-le* **by** *simp*

lemma *abs-of-nonneg* [*simp*]:
  assumes *nonneg*: $0 \leq a$
  shows $|a| = a$
proof (*rule order.antisym*)
  show $a \leq |a|$ **by** (*rule abs-ge-self*)
  from *nonneg le-imp-neg-le* **have** $-a \leq 0$ **by** *simp*
  from *this nonneg* **have** $-a \leq a$ **by** (*rule order-trans*)
  then show $|a| \leq a$ **by** (*auto intro*: *abs-leI*)
qed

lemma *abs-idempotent* [*simp*]: $||a|| = |a|$
  by (*rule order.antisym*) (*auto intro*!: *abs-ge-self abs-leI order-trans* [*of* $-|a|$ *0* $|a|$])

lemma *abs-eq-0* [*simp*]: $|a| = 0 \longleftrightarrow a = 0$
proof −
  have $|a| = 0 \Longrightarrow a = 0$
  proof (*rule order.antisym*)
    assume *zero*: $|a| = 0$
    with *abs-ge-self* **show** $a \leq 0$ **by** *auto*
    from *zero* **have** $|-a| = 0$ **by** *simp*
    with *abs-ge-self* [*of* $-a$] **have** $-a \leq 0$ **by** *auto*
    with *neg-le-0-iff-le* **show** $0 \leq a$ **by** *auto*
  qed
  then show *?thesis* **by** *auto*
qed

lemma *abs-zero* [*simp*]: $|0| = 0$
  by *simp*

lemma *abs-0-eq* [*simp*]: $0 = |a| \longleftrightarrow a = 0$
proof −
  have $0 = |a| \longleftrightarrow |a| = 0$ **by** (*simp only*: *eq-ac*)
  then show *?thesis* **by** *simp*
qed

lemma *abs-le-zero-iff* [*simp*]: $|a| \leq 0 \longleftrightarrow a = 0$
proof
  assume $|a| \leq 0$
  then have $|a| = 0$ **by** (*rule order.antisym*) *simp*
  then show $a = 0$ **by** *simp*
next
  assume $a = 0$
  then show $|a| \leq 0$ **by** *simp*
qed

lemma *abs-le-self-iff* [*simp*]: $|a| \leq a \longleftrightarrow 0 \leq a$
proof −
  have $0 \leq |a|$

```
    using abs-ge-zero by blast
  then have |a| ≤ a ⟹ 0 ≤ a
    using order.trans by blast
  then show ?thesis
    using abs-of-nonneg eq-refl by blast
qed

lemma zero-less-abs-iff [simp]: 0 < |a| ⟷ a ≠ 0
  by (simp add: less-le)

lemma abs-not-less-zero [simp]: ¬ |a| < 0
proof −
  have x ≤ y ⟹ ¬ y < x for x y by auto
  then show ?thesis by simp
qed

lemma abs-ge-minus-self: − a ≤ |a|
proof −
  have − a ≤ |−a| by (rule abs-ge-self)
  then show ?thesis by simp
qed

lemma abs-minus-commute: |a − b| = |b − a|
proof −
  have |a − b| = |− (a − b)|
    by (simp only: abs-minus-cancel)
  also have ... = |b − a| by simp
  finally show ?thesis .
qed

lemma abs-of-pos: 0 < a ⟹ |a| = a
  by (rule abs-of-nonneg) (rule less-imp-le)

lemma abs-of-nonpos [simp]:
  assumes a ≤ 0
  shows |a| = − a
proof −
  let ?b = − a
  have − ?b ≤ 0 ⟹ |− ?b| = − (− ?b)
    unfolding abs-minus-cancel [of ?b]
    unfolding neg-le-0-iff-le [of ?b]
    unfolding minus-minus by (erule abs-of-nonneg)
  then show ?thesis using assms by auto
qed

lemma abs-of-neg: a < 0 ⟹ |a| = − a
  by (rule abs-of-nonpos) (rule less-imp-le)

lemma abs-le-D1: |a| ≤ b ⟹ a ≤ b
```

using *abs-ge-self* **by** (*blast intro*: *order-trans*)

lemma *abs-le-D2*: $|a| \leq b \Longrightarrow - a \leq b$
using *abs-le-D1* [*of* $- a$] **by** *simp*

lemma *abs-le-iff*: $|a| \leq b \longleftrightarrow a \leq b \wedge - a \leq b$
by (*blast intro*: *abs-leI dest*: *abs-le-D1 abs-le-D2*)

lemma *abs-triangle-ineq2*: $|a| - |b| \leq |a - b|$
proof −
have $|a| = |b + (a - b)|$
by (*simp add*: *algebra-simps*)
then have $|a| \leq |b| + |a - b|$
by (*simp add*: *abs-triangle-ineq*)
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

lemma *abs-triangle-ineq2-sym*: $|a| - |b| \leq |b - a|$
by (*simp only*: *abs-minus-commute* [*of b*] *abs-triangle-ineq2*)

lemma *abs-triangle-ineq3*: $||a| - |b|| \leq |a - b|$
by (*simp add*: *abs-le-iff abs-triangle-ineq2 abs-triangle-ineq2-sym*)

lemma *abs-triangle-ineq4*: $|a - b| \leq |a| + |b|$
proof −
have $|a - b| = |a + - b|$
by (*simp add*: *algebra-simps*)
also have $\ldots \leq |a| + |- b|$
by (*rule abs-triangle-ineq*)
finally show *?thesis* **by** *simp*
qed

lemma *abs-diff-triangle-ineq*: $|a + b - (c + d)| \leq |a - c| + |b - d|$
proof −
have $|a + b - (c + d)| = |(a - c) + (b - d)|$
by (*simp add*: *algebra-simps*)
also have $\ldots \leq |a - c| + |b - d|$
by (*rule abs-triangle-ineq*)
finally show *?thesis* **.**
qed

lemma *abs-add-abs* [*simp*]: $||a| + |b|| = |a| + |b|$
(**is** *?L* = *?R*)
proof (*rule order.antisym*)
show *?L* $\geq$ *?R* **by** (*rule abs-ge-self*)
have *?L* $\leq ||a|| + ||b||$ **by** (*rule abs-triangle-ineq*)
also have $\ldots$ = *?R* **by** *simp*
finally show *?L* $\leq$ *?R* **.**

qed

end

lemma *dense-eq0-I*:
  fixes *x*::*'a*::{*dense-order*,*ordered-ab-group-add-abs*}
  assumes $\bigwedge e.\ 0 < e \Longrightarrow |x| \leq e$
  shows *x* = *0*
proof (*cases* |*x*| = *0*)
  case *False*
  then have |*x*| > *0*
    by *simp*
  then obtain *z* **where** *0* < *z z* < |*x*|
    using *dense* **by** *force*
  then show *?thesis*
    using *assms*[*of z*] **by** *auto*
qed *auto*

hide-fact (**open**) *ab-diff-conv-add-uminus add-0 mult-1 ab-left-minus*

lemmas *add-0* = *add-0-left*
lemmas *mult-1* = *mult-1-left*
lemmas *ab-left-minus* = *left-minus*
lemmas *diff-diff-eq* = *diff-diff-add*

5.8 Canonically ordered monoids

Canonically ordered monoids are never groups.

class *canonically-ordered-monoid-add* = *comm-monoid-add* + *order* +
  assumes *le-iff-add*: *a* ≤ *b* ⟷ (∃ *c*. *b* = *a* + *c*)
begin

lemma *zero-le*[*simp*]: *0* ≤ *x*
  by (*auto simp*: *le-iff-add*)

lemma *le-zero-eq*[*simp*]: *n* ≤ *0* ⟷ *n* = *0*
  by (*auto intro*: *order.antisym*)

lemma *not-less-zero*[*simp*]: ¬ *n* < *0*
  by (*auto simp*: *less-le*)

lemma *zero-less-iff-neq-zero*: *0* < *n* ⟷ *n* ≠ *0*
  by (*auto simp*: *less-le*)

This theorem is useful with *blast*

lemma *gr-zeroI*: (*n* = *0* ⟹ *False*) ⟹ *0* < *n*
  by (*rule zero-less-iff-neq-zero*[*THEN iffD2*]) *iprover*

lemma *not-gr-zero*[*simp*]: ¬ *0* < *n* ⟷ *n* = *0*

by (*simp add*: *zero-less-iff-neq-zero*)

subclass *ordered-comm-monoid-add*
proof qed (*auto simp*: *le-iff-add add-ac*)

lemma *gr-implies-not-zero*: $m < n \Longrightarrow n \neq 0$
by *auto*

lemma *add-eq-0-iff-both-eq-0*[*simp*]: $x + y = 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*intro add-nonneg-eq-0-iff zero-le*)

lemma *zero-eq-add-iff-both-eq-0*[*simp*]: $0 = x + y \longleftrightarrow x = 0 \wedge y = 0$
using *add-eq-0-iff-both-eq-0*[*of x y*] **unfolding** *eq-commute*[*of 0*] **.**

lemma *less-eqE*:
assumes ‹$a \leq b$›
obtains c **where** ‹$b = a + c$›
using *assms* **by** (*auto simp add*: *le-iff-add*)

lemma *lessE*:
assumes ‹$a < b$›
obtains c **where** ‹$b = a + c$› **and** ‹$c \neq 0$›
proof −
from *assms* **have** ‹$a \leq b$› ‹$a \neq b$›
by *simp-all*
from ‹$a \leq b$› **obtain** c **where** ‹$b = a + c$›
by (*rule less-eqE*)
moreover have ‹$c \neq 0$› **using** ‹$a \neq b$› ‹$b = a + c$›
by *auto*
ultimately show *?thesis*
by (*rule that*)
qed

lemmas *zero-order* = *zero-le le-zero-eq not-less-zero zero-less-iff-neq-zero not-gr-zero*
— This should be attributed with [*iff*], but then *blast* fails in *Set*.

end

class *ordered-cancel-comm-monoid-diff* =
canonically-ordered-monoid-add + *comm-monoid-diff* + *ordered-ab-semigroup-add-imp-le*
begin

context
fixes a b :: $'a$
assumes *le*: $a \leq b$
begin

lemma *add-diff-inverse*: $a + (b - a) = b$
using *le* **by** (*auto simp add*: *le-iff-add*)

lemma *add-diff-assoc*: $c + (b - a) = c + b - a$
using *le* **by** (*auto simp add*: *le-iff-add add.left-commute* [*of c*])

lemma *add-diff-assoc2*: $b - a + c = b + c - a$
using *le* **by** (*auto simp add*: *le-iff-add add.assoc*)

lemma *diff-add-assoc*: $c + b - a = c + (b - a)$
using *le* **by** (*simp add*: *add.commute add-diff-assoc*)

lemma *diff-add-assoc2*: $b + c - a = b - a + c$
using *le* **by** (*simp add*: *add.commute add-diff-assoc*)

lemma *diff-diff-right*: $c - (b - a) = c + a - b$
by (*simp add*: *add-diff-inverse add-diff-cancel-left* [*of a c b − a*, *symmetric*] *add.commute*)

lemma *diff-add*: $b - a + a = b$
by (*simp add*: *add.commute add-diff-inverse*)

lemma *le-add-diff*: $c \leq b + c - a$
by (*auto simp add*: *add.commute diff-add-assoc2 le-iff-add*)

lemma *le-imp-diff-is-add*: $a \leq b \Longrightarrow b - a = c \longleftrightarrow b = c + a$
by (*auto simp add*: *add.commute add-diff-inverse*)

lemma *le-diff-conv2*: $c \leq b - a \longleftrightarrow c + a \leq b$
(**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?P*
then have $c + a \leq b - a + a$
by (*rule add-right-mono*)
then show *?Q*
by (*simp add*: *add-diff-inverse add.commute*)
next
assume *?Q*
then have $a + c \leq a + (b - a)$
by (*simp add*: *add-diff-inverse add.commute*)
then show *?P* **by** *simp*
qed

end

end

5.9 Tools setup

lemma *add-mono-thms-linordered-semiring*:
fixes *i j k* :: *'a*::*ordered-ab-semigroup-add*

shows $i \leq j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k \leq l \Longrightarrow i + k \leq j + l$
and $i \leq j \wedge k = l \Longrightarrow i + k \leq j + l$
and $i = j \wedge k = l \Longrightarrow i + k = j + l$
by (*rule add-mono*, *clarify*+)+

lemma *add-mono-thms-linordered-field*:
fixes $i\ j\ k :: 'a$::*ordered-cancel-ab-semigroup-add*
shows $i < j \wedge k = l \Longrightarrow i + k < j + l$
and $i = j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k \leq l \Longrightarrow i + k < j + l$
and $i \leq j \wedge k < l \Longrightarrow i + k < j + l$
and $i < j \wedge k < l \Longrightarrow i + k < j + l$
by (*auto intro*: *add-strict-right-mono add-strict-left-mono add-less-le-mono add-le-less-mono add-strict-mono*)

code-identifier
code-module *Groups* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

6 Abstract lattices

theory *Lattices*
imports *Groups*
begin

6.1 Abstract semilattice

These locales provide a basic structure for interpretation into bigger structures; extensions require careful thinking, otherwise undesired effects may occur due to interpretation.

locale *semilattice* = *abel-semigroup* +
assumes *idem* [*simp*]: $a * a = a$
begin

lemma *left-idem* [*simp*]: $a * (a * b) = a * b$
by (*simp add*: *assoc* [*symmetric*])

lemma *right-idem* [*simp*]: $(a * b) * b = a * b$
by (*simp add*: *assoc*)

end

locale *semilattice-neutr* = *semilattice* + *comm-monoid*

locale *semilattice-order* = *semilattice* +
fixes *less-eq* :: $'a \Rightarrow 'a \Rightarrow$ *bool* (**infix** ‹≤› *50*)

and *less* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹<› *50*)
assumes *order-iff*: $a \leq b \longleftrightarrow a = a * b$
and *strict-order-iff*: $a < b \longleftrightarrow a = a * b \wedge a \neq b$
begin

lemma *orderI*: $a = a * b \Longrightarrow a \leq b$
by (*simp add*: *order-iff*)

lemma *orderE*:
assumes $a \leq b$
obtains $a = a * b$
using *assms* **by** (*unfold order-iff*)

sublocale *ordering less-eq less*
proof
show $a < b \longleftrightarrow a \leq b \wedge a \neq b$ **for** $a\ b$
by (*simp add*: *order-iff strict-order-iff*)
next
show $a \leq a$ **for** a
by (*simp add*: *order-iff*)
next
fix $a\ b$
assume $a \leq b\ b \leq a$
then have $a = a * b\ a * b = b$
by (*simp-all add*: *order-iff commute*)
then show $a = b$ **by** *simp*
next
fix $a\ b\ c$
assume $a \leq b\ b \leq c$
then have $a = a * b\ b = b * c$
by (*simp-all add*: *order-iff commute*)
then have $a = a * (b * c)$
by *simp*
then have $a = (a * b) * c$
by (*simp add*: *assoc*)
with ‹$a = a * b$› [*symmetric*] **have** $a = a * c$ **by** *simp*
then show $a \leq c$ **by** (*rule orderI*)
qed

lemma *cobounded1* [*simp*]: $a * b \leq a$
by (*simp add*: *order-iff commute*)

lemma *cobounded2* [*simp*]: $a * b \leq b$
by (*simp add*: *order-iff*)

lemma *boundedI*:
assumes $a \leq b$ **and** $a \leq c$
shows $a \leq b * c$
proof (*rule orderI*)

from *assms* **obtain** $a * b = a$ **and** $a * c = a$
by (*auto elim*!: *orderE*)
then show $a = a * (b * c)$
by (*simp add*: *assoc* [*symmetric*])
qed

lemma *boundedE*:
assumes $a \leq b * c$
obtains $a \leq b$ **and** $a \leq c$
using *assms* **by** (*blast intro*: *trans cobounded1 cobounded2*)

lemma *bounded-iff* [*simp*]: $a \leq b * c \longleftrightarrow a \leq b \wedge a \leq c$
by (*blast intro*: *boundedI elim*: *boundedE*)

lemma *strict-boundedE*:
assumes $a < b * c$
obtains $a < b$ **and** $a < c$
using *assms* **by** (*auto simp add*: *commute strict-iff-order elim*: *orderE intro*!: *that*)+

lemma *coboundedI1*: $a \leq c \Longrightarrow a * b \leq c$
by (*rule trans*) *auto*

lemma *coboundedI2*: $b \leq c \Longrightarrow a * b \leq c$
by (*rule trans*) *auto*

lemma *strict-coboundedI1*: $a < c \Longrightarrow a * b < c$
using *irrefl*
by (*auto intro*: *not-eq-order-implies-strict coboundedI1 strict-implies-order elim*: *strict-boundedE*)

lemma *strict-coboundedI2*: $b < c \Longrightarrow a * b < c$
using *strict-coboundedI1* [*of b c a*] **by** (*simp add*: *commute*)

lemma *mono*: $a \leq c \Longrightarrow b \leq d \Longrightarrow a * b \leq c * d$
by (*blast intro*: *boundedI coboundedI1 coboundedI2*)

lemma *absorb1*: $a \leq b \Longrightarrow a * b = a$
by (*rule antisym*) (*auto simp*: *refl*)

lemma *absorb2*: $b \leq a \Longrightarrow a * b = b$
by (*rule antisym*) (*auto simp*: *refl*)

lemma *absorb3*: $a < b \Longrightarrow a * b = a$
by (*rule absorb1*) (*rule strict-implies-order*)

lemma *absorb4*: $b < a \Longrightarrow a * b = b$
by (*rule absorb2*) (*rule strict-implies-order*)

lemma *absorb-iff1*: $a \leq b \longleftrightarrow a * b = a$
using *order-iff* **by** *auto*

lemma *absorb-iff2*: $b \leq a \longleftrightarrow a * b = b$
using *order-iff* **by** (*auto simp add*: *commute*)

end

locale *semilattice-neutr-order* = *semilattice-neutr* + *semilattice-order*
begin

sublocale *ordering-top less-eq less* **1**
by *standard* (*simp add*: *order-iff*)

lemma *eq-neutr-iff* [*simp*]: ‹$a * b = \mathbf{1} \longleftrightarrow a = \mathbf{1} \wedge b = \mathbf{1}$›
by (*simp add*: *eq-iff*)

lemma *neutr-eq-iff* [*simp*]: ‹$\mathbf{1} = a * b \longleftrightarrow a = \mathbf{1} \wedge b = \mathbf{1}$›
by (*simp add*: *eq-iff*)

end

Interpretations for boolean operators

interpretation *conj*: *semilattice-neutr* ‹(∧)› *True*
by *standard auto*

interpretation *disj*: *semilattice-neutr* ‹(∨)› *False*
by *standard auto*

declare *conj-assoc* [*ac-simps del*] *disj-assoc* [*ac-simps del*] — already simp by default

6.2 Syntactic infimum and supremum operations

class *inf* =
fixes *inf* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹⊓› *70*)

class *sup* =
fixes *sup* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹⊔› *65*)

6.3 Concrete lattices

class *semilattice-inf* = *order* + *inf* +
assumes *inf-le1* [*simp*]: $x \sqcap y \leq x$
and *inf-le2* [*simp*]: $x \sqcap y \leq y$
and *inf-greatest*: $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$

class *semilattice-sup* = *order* + *sup* +
assumes *sup-ge1* [*simp*]: $x \leq x \sqcup y$

and *sup-ge2* [*simp*]: $y \leq x \sqcup y$
and *sup-least*: $y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x$
begin

Dual lattice.

lemma *dual-semilattice*: *class.semilattice-inf sup greater-eq greater*
by (*rule class.semilattice-inf.intro*, *rule dual-order*)
(*unfold-locales*, *simp-all add*: *sup-least*)

end

class *lattice* = *semilattice-inf* + *semilattice-sup*

6.3.1 Intro and elim rules

context *semilattice-inf*
begin

lemma *le-infI1*: $a \leq x \Longrightarrow a \sqcap b \leq x$
by (*rule order-trans*) *auto*

lemma *le-infI2*: $b \leq x \Longrightarrow a \sqcap b \leq x$
by (*rule order-trans*) *auto*

lemma *le-infI*: $x \leq a \Longrightarrow x \leq b \Longrightarrow x \leq a \sqcap b$
by (*fact inf-greatest*)

lemma *le-infE*: $x \leq a \sqcap b \Longrightarrow (x \leq a \Longrightarrow x \leq b \Longrightarrow P) \Longrightarrow P$
by (*blast intro*: *order-trans inf-le1 inf-le2*)

lemma *le-inf-iff*: $x \leq y \sqcap z \longleftrightarrow x \leq y \wedge x \leq z$
by (*blast intro*: *le-infI elim*: *le-infE*)

lemma *le-iff-inf*: $x \leq y \longleftrightarrow x \sqcap y = x$
by (*auto intro*: *le-infI1 order.antisym dest*: *order.eq-iff* [*THEN iffD1*] *simp add*: *le-inf-iff*)

lemma *inf-mono*: $a \leq c \Longrightarrow b \leq d \Longrightarrow a \sqcap b \leq c \sqcap d$
by (*fast intro*: *inf-greatest le-infI1 le-infI2*)

end

context *semilattice-sup*
begin

lemma *le-supI1*: $x \leq a \Longrightarrow x \leq a \sqcup b$
by (*rule order-trans*) *auto*

lemma *le-supI2*: $x \leq b \Longrightarrow x \leq a \sqcup b$

by (*rule order-trans*) *auto*

lemma *le-supI*: $a \leq x \Longrightarrow b \leq x \Longrightarrow a \sqcup b \leq x$
by (*fact sup-least*)

lemma *le-supE*: $a \sqcup b \leq x \Longrightarrow (a \leq x \Longrightarrow b \leq x \Longrightarrow P) \Longrightarrow P$
by (*blast intro*: *order-trans sup-ge1 sup-ge2*)

lemma *le-sup-iff*: $x \sqcup y \leq z \longleftrightarrow x \leq z \wedge y \leq z$
by (*blast intro*: *le-supI elim*: *le-supE*)

lemma *le-iff-sup*: $x \leq y \longleftrightarrow x \sqcup y = y$
by (*auto intro*: *le-supI2 order.antisym dest*: *order.eq-iff* [*THEN iffD1*] *simp add*: *le-sup-iff*)

lemma *sup-mono*: $a \leq c \Longrightarrow b \leq d \Longrightarrow a \sqcup b \leq c \sqcup d$
by (*fast intro*: *sup-least le-supI1 le-supI2*)

end

6.3.2 Equational laws

context *semilattice-inf*
begin

sublocale *inf*: *semilattice inf*
proof
fix *a b c*
show $(a \sqcap b) \sqcap c = a \sqcap (b \sqcap c)$
by (*rule order.antisym*) (*auto intro*: *le-infI1 le-infI2 simp add*: *le-inf-iff*)
show $a \sqcap b = b \sqcap a$
by (*rule order.antisym*) (*auto simp add*: *le-inf-iff*)
show $a \sqcap a = a$
by (*rule order.antisym*) (*auto simp add*: *le-inf-iff*)
qed

sublocale *inf*: *semilattice-order inf less-eq less*
by *standard* (*auto simp add*: *le-iff-inf less-le*)

lemma *inf-assoc*: $(x \sqcap y) \sqcap z = x \sqcap (y \sqcap z)$
by (*fact inf.assoc*)

lemma *inf-commute*: $(x \sqcap y) = (y \sqcap x)$
by (*fact inf.commute*)

lemma *inf-left-commute*: $x \sqcap (y \sqcap z) = y \sqcap (x \sqcap z)$
by (*fact inf.left-commute*)

lemma *inf-idem*: $x \sqcap x = x$

by (*fact inf.idem*)

lemma *inf-left-idem*: $x \sqcap (x \sqcap y) = x \sqcap y$
by (*fact inf.left-idem*)

lemma *inf-right-idem*: $(x \sqcap y) \sqcap y = x \sqcap y$
by (*fact inf.right-idem*)

lemma *inf-absorb1*: $x \leq y \Longrightarrow x \sqcap y = x$
by (*rule order.antisym*) *auto*

lemma *inf-absorb2*: $y \leq x \Longrightarrow x \sqcap y = y$
by (*rule order.antisym*) *auto*

lemmas *inf-aci* = *inf-commute inf-assoc inf-left-commute inf-left-idem*

end

context *semilattice-sup*
begin

sublocale *sup*: *semilattice sup*
proof
fix *a b c*
show $(a \sqcup b) \sqcup c = a \sqcup (b \sqcup c)$
by (*rule order.antisym*) (*auto intro*: *le-supI1 le-supI2 simp add*: *le-sup-iff*)
show $a \sqcup b = b \sqcup a$
by (*rule order.antisym*) (*auto simp add*: *le-sup-iff*)
show $a \sqcup a = a$
by (*rule order.antisym*) (*auto simp add*: *le-sup-iff*)
qed

sublocale *sup*: *semilattice-order sup greater-eq greater*
by *standard* (*auto simp add*: *le-iff-sup sup.commute less-le*)

lemma *sup-assoc*: $(x \sqcup y) \sqcup z = x \sqcup (y \sqcup z)$
by (*fact sup.assoc*)

lemma *sup-commute*: $(x \sqcup y) = (y \sqcup x)$
by (*fact sup.commute*)

lemma *sup-left-commute*: $x \sqcup (y \sqcup z) = y \sqcup (x \sqcup z)$
by (*fact sup.left-commute*)

lemma *sup-idem*: $x \sqcup x = x$
by (*fact sup.idem*)

lemma *sup-left-idem* [*simp*]: $x \sqcup (x \sqcup y) = x \sqcup y$
by (*fact sup.left-idem*)

lemma *sup-absorb1*: $y \leq x \Longrightarrow x \sqcup y = x$
by (*rule order.antisym*) *auto*

lemma *sup-absorb2*: $x \leq y \Longrightarrow x \sqcup y = y$
by (*rule order.antisym*) *auto*

lemmas *sup-aci* = *sup-commute sup-assoc sup-left-commute sup-left-idem*

end

context *lattice*
begin

lemma *dual-lattice*: *class.lattice sup* (≥) (>) *inf*
by (*rule class.lattice.intro*,
rule dual-semilattice,
rule class.semilattice-sup.intro,
rule dual-order)
(*unfold-locales*, *auto*)

lemma *inf-sup-absorb* [*simp*]: $x \sqcap (x \sqcup y) = x$
by (*blast intro*: *order.antisym inf-le1 inf-greatest sup-ge1*)

lemma *sup-inf-absorb* [*simp*]: $x \sqcup (x \sqcap y) = x$
by (*blast intro*: *order.antisym sup-ge1 sup-least inf-le1*)

lemmas *inf-sup-aci* = *inf-aci sup-aci*

lemmas *inf-sup-ord* = *inf-le1 inf-le2 sup-ge1 sup-ge2*

Towards distributivity.

lemma *distrib-sup-le*: $x \sqcup (y \sqcap z) \leq (x \sqcup y) \sqcap (x \sqcup z)$
by (*auto intro*: *le-infI1 le-infI2 le-supI1 le-supI2*)

lemma *distrib-inf-le*: $(x \sqcap y) \sqcup (x \sqcap z) \leq x \sqcap (y \sqcup z)$
by (*auto intro*: *le-infI1 le-infI2 le-supI1 le-supI2*)

If you have one of them, you have them all.

lemma *distrib-imp1*:
assumes *distrib*: $\bigwedge x\ y\ z.\ x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
shows $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
proof−
have $x \sqcup (y \sqcap z) = (x \sqcup (x \sqcap z)) \sqcup (y \sqcap z)$
by *simp*
also have $\ldots = x \sqcup (z \sqcap (x \sqcup y))$
by (*simp add*: *distrib inf-commute sup-assoc del*: *sup-inf-absorb*)
also have $\ldots = ((x \sqcup y) \sqcap x) \sqcup ((x \sqcup y) \sqcap z)$
by (*simp add*: *inf-commute*)

also have ... $= (x \sqcup y) \sqcap (x \sqcup z)$ **by**(*simp add*:*distrib*)
finally show *?thesis* **.**
qed

lemma *distrib-imp2*:
assumes *distrib*: $\bigwedge x\ y\ z.\ x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$
shows $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
proof−
have $x \sqcap (y \sqcup z) = (x \sqcap (x \sqcup z)) \sqcap (y \sqcup z)$
by *simp*
also have ... $= x \sqcap (z \sqcup (x \sqcap y))$
by (*simp add*: *distrib sup-commute inf-assoc del*: *inf-sup-absorb*)
also have ... $= ((x \sqcap y) \sqcup x) \sqcap ((x \sqcap y) \sqcup z)$
by (*simp add*: *sup-commute*)
also have ... $= (x \sqcap y) \sqcup (x \sqcap z)$ **by** (*simp add*:*distrib*)
finally show *?thesis* **.**
qed

end

6.3.3 Strict order

context *semilattice-inf*
begin

lemma *less-infI1*: $a < x \Longrightarrow a \sqcap b < x$
by (*auto simp add*: *less-le inf-absorb1 intro*: *le-infI1*)

lemma *less-infI2*: $b < x \Longrightarrow a \sqcap b < x$
by (*auto simp add*: *less-le inf-absorb2 intro*: *le-infI2*)

end

context *semilattice-sup*
begin

lemma *less-supI1*: $x < a \Longrightarrow x < a \sqcup b$
using *dual-semilattice*
by (*rule semilattice-inf*.*less-infI1*)

lemma *less-supI2*: $x < b \Longrightarrow x < a \sqcup b$
using *dual-semilattice*
by (*rule semilattice-inf*.*less-infI2*)

end

6.4 Distributive lattices

class *distrib-lattice* = *lattice* +
assumes *sup-inf-distrib1*: $x \sqcup (y \sqcap z) = (x \sqcup y) \sqcap (x \sqcup z)$

context *distrib-lattice*
begin

lemma *sup-inf-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*simp add*: *sup-commute sup-inf-distrib1*)

lemma *inf-sup-distrib1*: $x \sqcap (y \sqcup z) = (x \sqcap y) \sqcup (x \sqcap z)$
by (*rule distrib-imp2* [*OF sup-inf-distrib1*])

lemma *inf-sup-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp add*: *inf-commute inf-sup-distrib1*)

lemma *dual-distrib-lattice*: *class.distrib-lattice sup* $(\geq)$ $(>)$ *inf*
by (*rule class.distrib-lattice.intro*, *rule dual-lattice*)
(*unfold-locales*, *fact inf-sup-distrib1*)

lemmas *sup-inf-distrib* = *sup-inf-distrib1 sup-inf-distrib2*

lemmas *inf-sup-distrib* = *inf-sup-distrib1 inf-sup-distrib2*

lemmas *distrib* = *sup-inf-distrib1 sup-inf-distrib2 inf-sup-distrib1 inf-sup-distrib2*

end

6.5 Bounded lattices

class *bounded-semilattice-inf-top* = *semilattice-inf* + *order-top*
begin

sublocale *inf-top*: *semilattice-neutr inf top*
+ *inf-top*: *semilattice-neutr-order inf top less-eq less*
proof
show $x \sqcap \top = x$ **for** x
by (*rule inf-absorb1*) *simp*
qed

lemma *inf-top-left*: $\top \sqcap x = x$
by (*fact inf-top.left-neutral*)

lemma *inf-top-right*: $x \sqcap \top = x$
by (*fact inf-top.right-neutral*)

lemma *inf-eq-top-iff*: $x \sqcap y = \top \longleftrightarrow x = \top \wedge y = \top$
by (*fact inf-top.eq-neutr-iff*)

lemma *top-eq-inf-iff*: $\top = x \sqcap y \longleftrightarrow x = \top \wedge y = \top$
by (*fact inf-top.neutr-eq-iff*)

end

class *bounded-semilattice-sup-bot* = *semilattice-sup* + *order-bot*
begin

sublocale *sup-bot*: *semilattice-neutr sup bot*
  + *sup-bot*: *semilattice-neutr-order sup bot greater-eq greater*
proof
  show $x \sqcup \bot = x$ **for** x
    by (*rule sup-absorb1*) *simp*
qed

lemma *sup-bot-left*: $\bot \sqcup x = x$
  by (*fact sup-bot.left-neutral*)

lemma *sup-bot-right*: $x \sqcup \bot = x$
  by (*fact sup-bot.right-neutral*)

lemma *sup-eq-bot-iff*: $x \sqcup y = \bot \longleftrightarrow x = \bot \wedge y = \bot$
  by (*fact sup-bot.eq-neutr-iff*)

lemma *bot-eq-sup-iff*: $\bot = x \sqcup y \longleftrightarrow x = \bot \wedge y = \bot$
  by (*fact sup-bot.neutr-eq-iff*)

end

class *bounded-lattice-bot* = *lattice* + *order-bot*
begin

subclass *bounded-semilattice-sup-bot* **..**

lemma *inf-bot-left* [*simp*]: $\bot \sqcap x = \bot$
  by (*rule inf-absorb1*) *simp*

lemma *inf-bot-right* [*simp*]: $x \sqcap \bot = \bot$
  by (*rule inf-absorb2*) *simp*

end

class *bounded-lattice-top* = *lattice* + *order-top*
begin

subclass *bounded-semilattice-inf-top* **..**

lemma *sup-top-left* [*simp*]: $\top \sqcup x = \top$
  by (*rule sup-absorb1*) *simp*

lemma *sup-top-right* [*simp*]: $x \sqcup \top = \top$
  by (*rule sup-absorb2*) *simp*

end

class *bounded-lattice* = *lattice* + *order-bot* + *order-top*
begin

subclass *bounded-lattice-bot* **..**
subclass *bounded-lattice-top* **..**

lemma *dual-bounded-lattice*: *class.bounded-lattice sup greater-eq greater inf* $\top$ $\bot$
by *unfold-locales* (*auto simp add*: *less-le-not-le*)

end

6.6 *min*/*max* as special case of lattice

context *linorder*
begin

sublocale *min*: *semilattice-order min less-eq less*
+ *max*: *semilattice-order max greater-eq greater*
by *standard* (*auto simp add*: *min-def max-def*)

declare *min.absorb1* [*simp*] *min.absorb2* [*simp*]
min.absorb3 [*simp*] *min.absorb4* [*simp*]
max.absorb1 [*simp*] *max.absorb2* [*simp*]
max.absorb3 [*simp*] *max.absorb4* [*simp*]

lemma *min-le-iff-disj*: *min x y* $\leq z \longleftrightarrow x \leq z \vee y \leq z$
unfolding *min-def* **using** *linear* **by** (*auto intro*: *order-trans*)

lemma *le-max-iff-disj*: $z \leq$ *max x y* $\longleftrightarrow z \leq x \vee z \leq y$
unfolding *max-def* **using** *linear* **by** (*auto intro*: *order-trans*)

lemma *min-less-iff-disj*: *min x y* $< z \longleftrightarrow x < z \vee y < z$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *less-max-iff-disj*: $z <$ *max x y* $\longleftrightarrow z < x \vee z < y$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *min-less-iff-conj* [*simp*]: $z <$ *min x y* $\longleftrightarrow z < x \wedge z < y$
unfolding *min-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *max-less-iff-conj* [*simp*]: *max x y* $< z \longleftrightarrow x < z \wedge y < z$
unfolding *max-def le-less* **using** *less-linear* **by** (*auto intro*: *less-trans*)

lemma *min-max-distrib1*: *min* (*max b c*) *a* = *max* (*min b a*) (*min c a*)
by (*auto simp add*: *min-def max-def not-le dest*: *le-less-trans less-trans intro*: *antisym*)

lemma *min-max-distrib2*: *min a* (*max b c*) = *max* (*min a b*) (*min a c*)
by (*auto simp add*: *min-def max-def not-le dest*: *le-less-trans less-trans intro*: *antisym*)

lemma *max-min-distrib1*: *max* (*min b c*) *a* = *min* (*max b a*) (*max c a*)
by (*auto simp add*: *min-def max-def not-le dest*: *le-less-trans less-trans intro*: *antisym*)

lemma *max-min-distrib2*: *max a* (*min b c*) = *min* (*max a b*) (*max a c*)
by (*auto simp add*: *min-def max-def not-le dest*: *le-less-trans less-trans intro*: *antisym*)

lemmas *min-max-distribs* = *min-max-distrib1 min-max-distrib2 max-min-distrib1 max-min-distrib2*

lemma *split-min* [*no-atp*]: *P* (*min i j*) ⟷ (*i* ≤ *j* ⟶ *P i*) ∧ (¬ *i* ≤ *j* ⟶ *P j*)
by (*simp add*: *min-def*)

lemma *split-max* [*no-atp*]: *P* (*max i j*) ⟷ (*i* ≤ *j* ⟶ *P j*) ∧ (¬ *i* ≤ *j* ⟶ *P i*)
by (*simp add*: *max-def*)

lemma *split-min-lin* [*no-atp*]:
‹*P* (*min a b*) ⟷ (*b* = *a* ⟶ *P a*) ∧ (*a* < *b* ⟶ *P a*) ∧ (*b* < *a* ⟶ *P b*)›
by (*cases a b rule*: *linorder-cases*) *auto*

lemma *split-max-lin* [*no-atp*]:
‹*P* (*max a b*) ⟷ (*b* = *a* ⟶ *P a*) ∧ (*a* < *b* ⟶ *P b*) ∧ (*b* < *a* ⟶ *P a*)›
by (*cases a b rule*: *linorder-cases*) *auto*

end

lemma *inf-min*: *inf* = (*min* :: *'a*::{*semilattice-inf*,*linorder*} ⇒ *'a* ⇒ *'a*)
by (*auto intro*: *antisym simp add*: *min-def fun-eq-iff*)

lemma *sup-max*: *sup* = (*max* :: *'a*::{*semilattice-sup*,*linorder*} ⇒ *'a* ⇒ *'a*)
by (*auto intro*: *antisym simp add*: *max-def fun-eq-iff*)

6.7 Uniqueness of inf and sup

lemma (**in** *semilattice-inf*) *inf-unique*:
fixes *f* (**infixl** ‹△› *70*)
assumes *le1*: ⋀*x y*. *x* △ *y* ≤ *x*
and *le2*: ⋀*x y*. *x* △ *y* ≤ *y*
and *greatest*: ⋀*x y z*. *x* ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *y* △ *z*
shows *x* ⊓ *y* = *x* △ *y*
proof (*rule order.antisym*)
show *x* △ *y* ≤ *x* ⊓ *y*
by (*rule le-infI*) (*rule le1*, *rule le2*)

have *leI*: $\bigwedge x\ y\ z.\ x \le y \Longrightarrow x \le z \Longrightarrow x \le y \vartriangle z$
by (*blast intro*: *greatest*)
show $x \sqcap y \le x \vartriangle y$
by (*rule leI*) *simp-all*
qed

lemma (**in** *semilattice-sup*) *sup-unique*:
fixes f (**infixl** ‹∇› *70*)
assumes *ge1* [*simp*]: $\bigwedge x\ y.\ x \le x \nabla y$
and *ge2*: $\bigwedge x\ y.\ y \le x \nabla y$
and *least*: $\bigwedge x\ y\ z.\ y \le x \Longrightarrow z \le x \Longrightarrow y \nabla z \le x$
shows $x \sqcup y = x \nabla y$
proof (*rule order.antisym*)
show $x \sqcup y \le x \nabla y$
by (*rule le-supI*) (*rule ge1*, *rule ge2*)
have *leI*: $\bigwedge x\ y\ z.\ x \le z \Longrightarrow y \le z \Longrightarrow x \nabla y \le z$
by (*blast intro*: *least*)
show $x \nabla y \le x \sqcup y$
by (*rule leI*) *simp-all*
qed

6.8 Lattice on - ⇒ -

instantiation *fun* :: (*type*, *semilattice-sup*) *semilattice-sup*
begin

definition $f \sqcup g = (\lambda x.\ f\ x \sqcup g\ x)$

lemma *sup-apply* [*simp*, *code*]: $(f \sqcup g)\ x = f\ x \sqcup g\ x$
by (*simp add*: *sup-fun-def*)

instance
by *standard* (*simp-all add*: *le-fun-def*)

end

instantiation *fun* :: (*type*, *semilattice-inf*) *semilattice-inf*
begin

definition $f \sqcap g = (\lambda x.\ f\ x \sqcap g\ x)$

lemma *inf-apply* [*simp*, *code*]: $(f \sqcap g)\ x = f\ x \sqcap g\ x$
by (*simp add*: *inf-fun-def*)

instance by *standard* (*simp-all add*: *le-fun-def*)

end

instance *fun* :: (*type*, *lattice*) *lattice* **..**

instance *fun* :: (*type*, *distrib-lattice*) *distrib-lattice*
  by *standard* (*rule ext*, *simp add*: *sup-inf-distrib1*)

instance *fun* :: (*type*, *bounded-lattice*) *bounded-lattice* **..**

instantiation *fun* :: (*type*, *uminus*) *uminus*
begin

definition *fun-Compl-def*: $- A = (\lambda x.\ - A\ x)$

lemma *uminus-apply* [*simp*, *code*]: $(- A)\ x = - (A\ x)$
  by (*simp add*: *fun-Compl-def*)

instance ..

end

instantiation *fun* :: (*type*, *minus*) *minus*
begin

definition *fun-diff-def*: $A - B = (\lambda x.\ A\ x - B\ x)$

lemma *minus-apply* [*simp*, *code*]: $(A - B)\ x = A\ x - B\ x$
  by (*simp add*: *fun-diff-def*)

instance ..

end

end

7 Boolean Algebras

theory *Boolean-Algebras*
  imports *Lattices*
begin

7.1 Abstract boolean algebra

locale *abstract-boolean-algebra* = *conj*: *abel-semigroup* ‹(⊓)› + *disj*: *abel-semigroup* ‹(⊔)›
  for *conj* :: ‹'*a* ⇒ '*a* ⇒ '*a*› (**infixr** ‹⊓› *70*)
    and *disj* :: ‹'*a* ⇒ '*a* ⇒ '*a*› (**infixr** ‹⊔› *65*) +
  fixes *compl* :: ‹'*a* ⇒ '*a*› (‹(‹*open-block notation*=‹*prefix* −››− -)› [*81*] *80*)
    and *zero* :: ‹'*a*› (‹**0**›)
    and *one* :: ‹'*a*› (‹**1**›)
  assumes *conj-disj-distrib*: ‹*x* ⊓ (*y* ⊔ *z*) = (*x* ⊓ *y*) ⊔ (*x* ⊓ *z*)›
    and *disj-conj-distrib*: ‹*x* ⊔ (*y* ⊓ *z*) = (*x* ⊔ *y*) ⊓ (*x* ⊔ *z*)›

and *conj-one-right*: ‹$x \sqcap \mathbf{1} = x$›
and *disj-zero-right*: ‹$x \sqcup \mathbf{0} = x$›
and *conj-cancel-right* [*simp*]: ‹$x \sqcap - x = \mathbf{0}$›
and *disj-cancel-right* [*simp*]: ‹$x \sqcup - x = \mathbf{1}$›
begin

sublocale *conj*: *semilattice-neutr* ‹$(\sqcap)$› ‹$\mathbf{1}$›
proof
show $x \sqcap \mathbf{1} = x$ **for** x
by (*fact conj-one-right*)
show $x \sqcap x = x$ **for** x
proof −
have $x \sqcap x = (x \sqcap x) \sqcup \mathbf{0}$
by (*simp add*: *disj-zero-right*)
also have $\ldots = (x \sqcap x) \sqcup (x \sqcap - x)$
by *simp*
also have $\ldots = x \sqcap (x \sqcup - x)$
by (*simp only*: *conj-disj-distrib*)
also have $\ldots = x \sqcap \mathbf{1}$
by *simp*
also have $\ldots = x$
by (*simp add*: *conj-one-right*)
finally show *?thesis* **.**
qed
qed

sublocale *disj*: *semilattice-neutr* ‹$(\sqcup)$› ‹$\mathbf{0}$›
proof
show $x \sqcup \mathbf{0} = x$ **for** x
by (*fact disj-zero-right*)
show $x \sqcup x = x$ **for** x
proof −
have $x \sqcup x = (x \sqcup x) \sqcap \mathbf{1}$
by *simp*
also have $\ldots = (x \sqcup x) \sqcap (x \sqcup - x)$
by *simp*
also have $\ldots = x \sqcup (x \sqcap - x)$
by (*simp only*: *disj-conj-distrib*)
also have $\ldots = x \sqcup \mathbf{0}$
by *simp*
also have $\ldots = x$
by (*simp add*: *disj-zero-right*)
finally show *?thesis* **.**
qed
qed

7.1.1 Complement

lemma *complement-unique*:

assumes *1*: $a \sqcap x = \mathbf{0}$
assumes *2*: $a \sqcup x = \mathbf{1}$
assumes *3*: $a \sqcap y = \mathbf{0}$
assumes *4*: $a \sqcup y = \mathbf{1}$
shows $x = y$
proof −
from *1 3* **have** $(a \sqcap x) \sqcup (x \sqcap y) = (a \sqcap y) \sqcup (x \sqcap y)$
by *simp*
then have $(x \sqcap a) \sqcup (x \sqcap y) = (y \sqcap a) \sqcup (y \sqcap x)$
by (*simp add*: *ac-simps*)
then have $x \sqcap (a \sqcup y) = y \sqcap (a \sqcup x)$
by (*simp add*: *conj-disj-distrib*)
with *2 4* **have** $x \sqcap \mathbf{1} = y \sqcap \mathbf{1}$
by *simp*
then show $x = y$
by *simp*
qed

lemma *compl-unique*: $x \sqcap y = \mathbf{0} \Longrightarrow x \sqcup y = \mathbf{1} \Longrightarrow - x = y$
by (*rule complement-unique* [*OF conj-cancel-right disj-cancel-right*])

lemma *double-compl* [*simp*]: $- (- x) = x$
proof (*rule compl-unique*)
show $- x \sqcap x = \mathbf{0}$
by (*simp only*: *conj-cancel-right conj.commute*)
show $- x \sqcup x = \mathbf{1}$
by (*simp only*: *disj-cancel-right disj.commute*)
qed

lemma *compl-eq-compl-iff* [*simp*]:
$‹- x = - y \longleftrightarrow x = y›$ (**is** $‹?P \longleftrightarrow ?Q›$)
proof
assume $‹?Q›$
then show *?P* **by** *simp*
next
assume $‹?P›$
then have $‹- (- x) = - (- y)›$
by *simp*
then show *?Q*
by *simp*
qed

7.1.2 Conjunction

lemma *conj-zero-right* [*simp*]: $x \sqcap \mathbf{0} = \mathbf{0}$
using *conj.left-idem conj-cancel-right* **by** *fastforce*

lemma *compl-one* [*simp*]: $- \mathbf{1} = \mathbf{0}$
by (*rule compl-unique* [*OF conj-zero-right disj-zero-right*])

lemma *conj-zero-left* [*simp*]: $\mathbf{0} \sqcap x = \mathbf{0}$
by (*subst conj.commute*) (*rule conj-zero-right*)

lemma *conj-cancel-left* [*simp*]: $- x \sqcap x = \mathbf{0}$
by (*subst conj.commute*) (*rule conj-cancel-right*)

lemma *conj-disj-distrib2*: $(y \sqcup z) \sqcap x = (y \sqcap x) \sqcup (z \sqcap x)$
by (*simp only*: *conj.commute conj-disj-distrib*)

lemmas *conj-disj-distribs* = *conj-disj-distrib conj-disj-distrib2*

7.1.3 Disjunction

context
begin

interpretation *dual*: *abstract-boolean-algebra* ‹(⊔)› ‹(⊓)› *compl* ‹**1**› ‹**0**›
apply *standard*
apply (*rule disj-conj-distrib*)
apply (*rule conj-disj-distrib*)
apply *simp-all*
done

lemma *disj-one-right* [*simp*]: $x \sqcup \mathbf{1} = \mathbf{1}$
by (*fact dual.conj-zero-right*)

lemma *compl-zero* [*simp*]: $- \mathbf{0} = \mathbf{1}$
by (*fact dual.compl-one*)

lemma *disj-one-left* [*simp*]: $\mathbf{1} \sqcup x = \mathbf{1}$
by (*fact dual.conj-zero-left*)

lemma *disj-cancel-left* [*simp*]: $- x \sqcup x = \mathbf{1}$
by (*fact dual.conj-cancel-left*)

lemma *disj-conj-distrib2*: $(y \sqcap z) \sqcup x = (y \sqcup x) \sqcap (z \sqcup x)$
by (*fact dual.conj-disj-distrib2*)

lemmas *disj-conj-distribs* = *disj-conj-distrib disj-conj-distrib2*

end

7.1.4 De Morgan's Laws

lemma *de-Morgan-conj* [*simp*]: $- (x \sqcap y) = - x \sqcup - y$
proof (*rule compl-unique*)
have $(x \sqcap y) \sqcap (- x \sqcup - y) = ((x \sqcap y) \sqcap - x) \sqcup ((x \sqcap y) \sqcap - y)$
by (*rule conj-disj-distrib*)
also have $\ldots = (y \sqcap (x \sqcap - x)) \sqcup (x \sqcap (y \sqcap - y))$

by (*simp only*: *ac-simps*)
finally show $(x \sqcap y) \sqcap (- x \sqcup - y) = \mathbf{0}$
by (*simp only*: *conj-cancel-right conj-zero-right disj-zero-right*)
next
have $(x \sqcap y) \sqcup (- x \sqcup - y) = (x \sqcup (- x \sqcup - y)) \sqcap (y \sqcup (- x \sqcup - y))$
by (*rule disj-conj-distrib2*)
also have $\ldots = (- y \sqcup (x \sqcup - x)) \sqcap (- x \sqcup (y \sqcup - y))$
by (*simp only*: *ac-simps*)
finally show $(x \sqcap y) \sqcup (- x \sqcup - y) = \mathbf{1}$
by (*simp only*: *disj-cancel-right disj-one-right conj-one-right*)
qed

context
begin

interpretation *dual*: *abstract-boolean-algebra* ‹(⊔)› ‹(⊓)› *compl* ‹**1**› ‹**0**›
apply *standard*
apply (*rule disj-conj-distrib*)
apply (*rule conj-disj-distrib*)
apply *simp-all*
done

lemma *de-Morgan-disj* [*simp*]: $- (x \sqcup y) = - x \sqcap - y$
by (*fact dual.de-Morgan-conj*)

end

end

7.2 Symmetric Difference

locale *abstract-boolean-algebra-sym-diff* = *abstract-boolean-algebra* +
fixes *xor* :: ‹'*a* ⇒ '*a* ⇒ '*a*› (**infixr** ‹⊖› *65*)
assumes *xor-def* : ‹$x \ominus y = (x \sqcap - y) \sqcup (- x \sqcap y)$›
begin

sublocale *xor*: *comm-monoid xor* ‹**0**›
proof
fix *x y z* :: '*a*
let *?t* = $(x \sqcap y \sqcap z) \sqcup (x \sqcap - y \sqcap - z) \sqcup (- x \sqcap y \sqcap - z) \sqcup (- x \sqcap - y \sqcap z)$
have $?t \sqcup (z \sqcap x \sqcap - x) \sqcup (z \sqcap y \sqcap - y) = ?t \sqcup (x \sqcap y \sqcap - y) \sqcup (x \sqcap z \sqcap - z)$
by (*simp only*: *conj-cancel-right conj-zero-right*)
then show $(x \ominus y) \ominus z = x \ominus (y \ominus z)$
by (*simp only*: *xor-def de-Morgan-disj de-Morgan-conj double-compl*)
(*simp only*: *conj-disj-distribs conj-ac ac-simps*)
show $x \ominus y = y \ominus x$
by (*simp only*: *xor-def ac-simps*)

show $x \ominus \mathbf{0} = x$
by (*simp add*: *xor-def*)
qed

lemma *xor-def2*:
$\langle x \ominus y = (x \sqcup y) \sqcap (- x \sqcup - y)\rangle$
proof −
note *xor-def* [*of x y*]
also have $\langle x \sqcap - y \sqcup - x \sqcap y = ((x \sqcup - x) \sqcap (- y \sqcup - x)) \sqcap (x \sqcup y) \sqcap (- y \sqcup y)\rangle$
by (*simp add*: *ac-simps disj-conj-distribs*)
also have $\langle \ldots = (x \sqcup y) \sqcap (- x \sqcup - y)\rangle$
by (*simp add*: *ac-simps*)
finally show *?thesis* **.**
qed

lemma *xor-one-right* [*simp*]: $x \ominus \mathbf{1} = - x$
by (*simp only*: *xor-def compl-one conj-zero-right conj-one-right disj.left-neutral*)

lemma *xor-one-left* [*simp*]: $\mathbf{1} \ominus x = - x$
using *xor-one-right* [*of x*] **by** (*simp add*: *ac-simps*)

lemma *xor-self* [*simp*]: $x \ominus x = \mathbf{0}$
by (*simp only*: *xor-def conj-cancel-right conj-cancel-left disj-zero-right*)

lemma *xor-left-self* [*simp*]: $x \ominus (x \ominus y) = y$
by (*simp only*: *xor.assoc* [*symmetric*] *xor-self xor.left-neutral*)

lemma *xor-compl-left* [*simp*]: $- x \ominus y = - (x \ominus y)$
by (*simp add*: *ac-simps flip*: *xor-one-left*)

lemma *xor-compl-right* [*simp*]: $x \ominus - y = - (x \ominus y)$
using *xor.commute xor-compl-left* **by** *auto*

lemma *xor-cancel-right* [*simp*]: $x \ominus - x = \mathbf{1}$
by (*simp only*: *xor-compl-right xor-self compl-zero*)

lemma *xor-cancel-left* [*simp*]: $- x \ominus x = \mathbf{1}$
by (*simp only*: *xor-compl-left xor-self compl-zero*)

lemma *conj-xor-distrib*: $x \sqcap (y \ominus z) = (x \sqcap y) \ominus (x \sqcap z)$
proof −
have ∗: $(x \sqcap y \sqcap - z) \sqcup (x \sqcap - y \sqcap z) =$
$(y \sqcap x \sqcap - x) \sqcup (z \sqcap x \sqcap - x) \sqcup (x \sqcap y \sqcap - z) \sqcup (x \sqcap - y \sqcap z)$
by (*simp only*: *conj-cancel-right conj-zero-right disj.left-neutral*)
then show $x \sqcap (y \ominus z) = (x \sqcap y) \ominus (x \sqcap z)$
by (*simp* (*no-asm-use*) *only*:
xor-def de-Morgan-disj de-Morgan-conj double-compl
conj-disj-distribs ac-simps)

qed

lemma *conj-xor-distrib2*: $(y \ominus z) \sqcap x = (y \sqcap x) \ominus (z \sqcap x)$
by (*simp add*: *conj.commute conj-xor-distrib*)

lemmas *conj-xor-distribs* = *conj-xor-distrib conj-xor-distrib2*

end

7.3 Type classes

class *boolean-algebra* = *distrib-lattice* + *bounded-lattice* + *minus* + *uminus* +
assumes *inf-compl-bot*: ‹$x \sqcap - x = \bot$›
and *sup-compl-top*: ‹$x \sqcup - x = \top$›
assumes *diff-eq*: ‹$x - y = x \sqcap - y$›
begin

sublocale *boolean-algebra*: *abstract-boolean-algebra* ‹(⊓)› ‹(⊔)› *uminus* $\bot$ $\top$
apply *standard*
apply (*rule inf-sup-distrib1*)
apply (*rule sup-inf-distrib1*)
apply (*simp-all add*: *ac-simps inf-compl-bot sup-compl-top*)
done

lemma *compl-inf-bot*: $- x \sqcap x = \bot$
by (*fact boolean-algebra.conj-cancel-left*)

lemma *compl-sup-top*: $- x \sqcup x = \top$
by (*fact boolean-algebra.disj-cancel-left*)

lemma *compl-unique*:
assumes $x \sqcap y = \bot$
and $x \sqcup y = \top$
shows $- x = y$
using *assms* **by** (*rule boolean-algebra.compl-unique*)

lemma *double-compl*: $- (- x) = x$
by (*fact boolean-algebra.double-compl*)

lemma *compl-eq-compl-iff*: $- x = - y \longleftrightarrow x = y$
by (*fact boolean-algebra.compl-eq-compl-iff*)

lemma *compl-bot-eq*: $- \bot = \top$
by (*fact boolean-algebra.compl-zero*)

lemma *compl-top-eq*: $- \top = \bot$
by (*fact boolean-algebra.compl-one*)

lemma *compl-inf*: $- (x \sqcap y) = - x \sqcup - y$

by (*fact boolean-algebra.de-Morgan-conj*)

lemma *compl-sup*: $-(x \sqcup y) = -x \sqcap -y$
by (*fact boolean-algebra.de-Morgan-disj*)

lemma *compl-mono*:
assumes $x \leq y$
shows $-y \leq -x$
proof −
from *assms* **have** $x \sqcup y = y$ **by** (*simp only*: *le-iff-sup*)
then have $-(x \sqcup y) = -y$ **by** *simp*
then have $-x \sqcap -y = -y$ **by** *simp*
then have $-y \sqcap -x = -y$ **by** (*simp only*: *inf-commute*)
then show *?thesis* **by** (*simp only*: *le-iff-inf*)
qed

lemma *compl-le-compl-iff* [*simp*]: $-x \leq -y \longleftrightarrow y \leq x$
by (*auto dest*: *compl-mono*)

lemma *compl-le-swap1*:
assumes $y \leq -x$
shows $x \leq -y$
proof −
from *assms* **have** $-(-x) \leq -y$ **by** (*simp only*: *compl-le-compl-iff*)
then show *?thesis* **by** *simp*
qed

lemma *compl-le-swap2*:
assumes $-y \leq x$
shows $-x \leq y$
proof −
from *assms* **have** $-x \leq -(-y)$ **by** (*simp only*: *compl-le-compl-iff*)
then show *?thesis* **by** *simp*
qed

lemma *compl-less-compl-iff* [*simp*]: $-x < -y \longleftrightarrow y < x$
by (*auto simp add*: *less-le*)

lemma *compl-less-swap1*:
assumes $y < -x$
shows $x < -y$
proof −
from *assms* **have** $-(-x) < -y$ **by** (*simp only*: *compl-less-compl-iff*)
then show *?thesis* **by** *simp*
qed

lemma *compl-less-swap2*:
assumes $-y < x$
shows $-x < y$

proof −
from *assms* **have** $- x < - (- y)$
by (*simp only*: *compl-less-compl-iff*)
then show *?thesis* **by** *simp*
qed

lemma *sup-cancel-left1*: ‹$x \sqcup a \sqcup (- x \sqcup b) = \top$›
by (*simp add*: *ac-simps*)

lemma *sup-cancel-left2*: ‹$- x \sqcup a \sqcup (x \sqcup b) = \top$›
by (*simp add*: *ac-simps*)

lemma *inf-cancel-left1*: ‹$x \sqcap a \sqcap (- x \sqcap b) = \bot$›
by (*simp add*: *ac-simps*)

lemma *inf-cancel-left2*: ‹$- x \sqcap a \sqcap (x \sqcap b) = \bot$›
by (*simp add*: *ac-simps*)

lemma *sup-compl-top-left1* [*simp*]: ‹$- x \sqcup (x \sqcup y) = \top$›
by (*simp add*: *sup-assoc* [*symmetric*])

lemma *sup-compl-top-left2* [*simp*]: ‹$x \sqcup (- x \sqcup y) = \top$›
using *sup-compl-top-left1* [*of* $- x\ y$] **by** *simp*

lemma *inf-compl-bot-left1* [*simp*]: ‹$- x \sqcap (x \sqcap y) = \bot$›
by (*simp add*: *inf-assoc* [*symmetric*])

lemma *inf-compl-bot-left2* [*simp*]: ‹$x \sqcap (- x \sqcap y) = \bot$›
using *inf-compl-bot-left1* [*of* $- x\ y$] **by** *simp*

lemma *inf-compl-bot-right* [*simp*]: ‹$x \sqcap (y \sqcap - x) = \bot$›
by (*subst inf-left-commute*) *simp*

end

7.4 Lattice on *bool*

instantiation *bool* :: *boolean-algebra*
begin

definition *bool-Compl-def* [*simp*]: *uminus* = *Not*

definition *bool-diff-def* [*simp*]: $A - B \longleftrightarrow A \wedge \neg B$

definition [*simp*]: $P \sqcap Q \longleftrightarrow P \wedge Q$

definition [*simp*]: $P \sqcup Q \longleftrightarrow P \vee Q$

instance by *standard auto*

end

lemma *sup-boolI1*: $P \Longrightarrow P \sqcup Q$
by *simp*

lemma *sup-boolI2*: $Q \Longrightarrow P \sqcup Q$
by *simp*

lemma *sup-boolE*: $P \sqcup Q \Longrightarrow (P \Longrightarrow R) \Longrightarrow (Q \Longrightarrow R) \Longrightarrow R$
by *auto*

instance *fun* :: (*type*, *boolean-algebra*) *boolean-algebra*
by *standard* (*rule ext*, *simp-all add*: *inf-compl-bot sup-compl-top diff-eq*)+

7.5 Lattice on unary and binary predicates

lemma *inf1I*: $A\ x \Longrightarrow B\ x \Longrightarrow (A \sqcap B)\ x$
by (*simp add*: *inf-fun-def*)

lemma *inf2I*: $A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow (A \sqcap B)\ x\ y$
by (*simp add*: *inf-fun-def*)

lemma *inf1E*: $(A \sqcap B)\ x \Longrightarrow (A\ x \Longrightarrow B\ x \Longrightarrow P) \Longrightarrow P$
by (*simp add*: *inf-fun-def*)

lemma *inf2E*: $(A \sqcap B)\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow B\ x\ y \Longrightarrow P) \Longrightarrow P$
by (*simp add*: *inf-fun-def*)

lemma *inf1D1*: $(A \sqcap B)\ x \Longrightarrow A\ x$
by (*rule inf1E*)

lemma *inf2D1*: $(A \sqcap B)\ x\ y \Longrightarrow A\ x\ y$
by (*rule inf2E*)

lemma *inf1D2*: $(A \sqcap B)\ x \Longrightarrow B\ x$
by (*rule inf1E*)

lemma *inf2D2*: $(A \sqcap B)\ x\ y \Longrightarrow B\ x\ y$
by (*rule inf2E*)

lemma *sup1I1*: $A\ x \Longrightarrow (A \sqcup B)\ x$
by (*simp add*: *sup-fun-def*)

lemma *sup2I1*: $A\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
by (*simp add*: *sup-fun-def*)

lemma *sup1I2*: $B\ x \Longrightarrow (A \sqcup B)\ x$
by (*simp add*: *sup-fun-def*)

lemma *sup2I2*: $B\ x\ y \Longrightarrow (A \sqcup B)\ x\ y$
by (*simp add*: *sup-fun-def*)

lemma *sup1E*: $(A \sqcup B)\ x \Longrightarrow (A\ x \Longrightarrow P) \Longrightarrow (B\ x \Longrightarrow P) \Longrightarrow P$
by (*simp add*: *sup-fun-def*) *iprover*

lemma *sup2E*: $(A \sqcup B)\ x\ y \Longrightarrow (A\ x\ y \Longrightarrow P) \Longrightarrow (B\ x\ y \Longrightarrow P) \Longrightarrow P$
by (*simp add*: *sup-fun-def*) *iprover*

Classical introduction rule: no commitment to A vs B.

lemma *sup1CI*: $(\neg\ B\ x \Longrightarrow A\ x) \Longrightarrow (A \sqcup B)\ x$
by (*auto simp add*: *sup-fun-def*)

lemma *sup2CI*: $(\neg\ B\ x\ y \Longrightarrow A\ x\ y) \Longrightarrow (A \sqcup B)\ x\ y$
by (*auto simp add*: *sup-fun-def*)

7.6 Simproc setup

locale *boolean-algebra-cancel*
begin

lemma *sup1*: $(A::'a::semilattice\text{-}sup) \equiv sup\ k\ a \Longrightarrow sup\ A\ b \equiv sup\ k\ (sup\ a\ b)$
by (*simp only*: *ac-simps*)

lemma *sup2*: $(B::'a::semilattice\text{-}sup) \equiv sup\ k\ b \Longrightarrow sup\ a\ B \equiv sup\ k\ (sup\ a\ b)$
by (*simp only*: *ac-simps*)

lemma *sup0*: $(a::'a::bounded\text{-}semilattice\text{-}sup\text{-}bot) \equiv sup\ a\ bot$
by *simp*

lemma *inf1*: $(A::'a::semilattice\text{-}inf) \equiv inf\ k\ a \Longrightarrow inf\ A\ b \equiv inf\ k\ (inf\ a\ b)$
by (*simp only*: *ac-simps*)

lemma *inf2*: $(B::'a::semilattice\text{-}inf) \equiv inf\ k\ b \Longrightarrow inf\ a\ B \equiv inf\ k\ (inf\ a\ b)$
by (*simp only*: *ac-simps*)

lemma *inf0*: $(a::'a::bounded\text{-}semilattice\text{-}inf\text{-}top) \equiv inf\ a\ top$
by *simp*

end

ML-file ‹*Tools/boolean-algebra-cancel.ML*›

simproc-setup *boolean-algebra-cancel-sup* (*sup a b*::'*a*::*boolean-algebra*) =
‹*K* (*K* (*try Boolean-Algebra-Cancel.cancel-sup-conv*))›

simproc-setup *boolean-algebra-cancel-inf* (*inf a b*::'*a*::*boolean-algebra*) =
‹*K* (*K* (*try Boolean-Algebra-Cancel.cancel-inf-conv*))›

context *boolean-algebra*
begin

lemma *shunt1*: $(x \sqcap y \leq z) \longleftrightarrow (x \leq -y \sqcup z)$
proof
  assume $x \sqcap y \leq z$
  hence $-y \sqcup (x \sqcap y) \leq -y \sqcup z$
    using *sup.mono* **by** *blast*
  hence $-y \sqcup x \leq -y \sqcup z$
    by (*simp add*: *sup-inf-distrib1*)
  thus $x \leq -y \sqcup z$
    by *simp*
next
  assume $x \leq -y \sqcup z$
  hence $x \sqcap y \leq (-y \sqcup z) \sqcap y$
    using *inf-mono* **by** *auto*
  thus $x \sqcap y \leq z$
    using *inf.boundedE inf-sup-distrib2* **by** *auto*
qed

lemma *shunt2*: $(x \sqcap -y \leq z) \longleftrightarrow (x \leq y \sqcup z)$
  by (*simp add*: *shunt1*)

lemma *inf-shunt*: $(x \sqcap y = \bot) \longleftrightarrow (x \leq - y)$
  by (*simp add*: *order.eq-iff shunt1*)

lemma *sup-shunt*: $(x \sqcup y = \top) \longleftrightarrow (- x \leq y)$
  using *inf-shunt* [*of* ‹$- x$› ‹$- y$›, *symmetric*]
  by (*simp flip*: *compl-sup compl-top-eq*)

lemma *diff-shunt-var*[*simp*]: $(x - y = \bot) \longleftrightarrow (x \leq y)$
  by (*simp add*: *diff-eq inf-shunt*)

lemma *diff-shunt*[*simp*]: $(\bot = x - y) \longleftrightarrow (x \leq y)$
  by (*auto simp flip*: *diff-shunt-var*)

lemma *sup-neg-inf*:
  ‹$p \leq q \sqcup r \longleftrightarrow p \sqcap -q \leq r$› (**is** ‹*?P* $\longleftrightarrow$ *?Q*›)
proof
  assume *?P*
  then have ‹$p \sqcap - q \leq (q \sqcup r) \sqcap - q$›
    by (*rule inf-mono*) *simp*
  then show *?Q*
    by (*simp add*: *inf-sup-distrib2*)
next
  assume *?Q*
  then have ‹$p \sqcap - q \sqcup q \leq r \sqcup q$›

by (*rule sup-mono*) *simp*
then show *?P*
by (*simp add*: *sup-inf-distrib ac-simps*)
qed

end

end

8 Set theory for higher-order logic

theory *Set*
imports *Lattices Boolean-Algebras*
begin

8.1 Sets as predicates

typedecl *'a set*

axiomatization *Collect* :: (*'a* ⇒ *bool*) ⇒ *'a set* — comprehension
and *member* :: *'a* ⇒ *'a set* ⇒ *bool* — membership
where *mem-Collect-eq* [*iff*, *code-unfold*]: *member a* (*Collect P*) = *P a*
and *Collect-mem-eq* [*simp*, *code-unfold*]: *Collect* (λ*x*. *member x A*) = *A*

notation
member (‹*'*(∈*'*)›) **and**
member (‹(‹*notation*=‹*infix* ∈››-/ ∈ -)› [*51*, *51*] *50*)

abbreviation *not-member*
where *not-member x A* ≡ ¬ (*x* ∈ *A*) — non-membership
notation
not-member (‹*'*(∉*'*)›) **and**
not-member (‹(‹*notation*=‹*infix* ∉››-/ ∉ -)› [*51*, *51*] *50*)

open-bundle *member-ASCII-syntax*
begin
notation (*ASCII*)
member (‹*'*(:*'*)›) **and**
member (‹(‹*notation*=‹*infix* :››-/ : -)› [*51*, *51*] *50*) **and**
not-member (‹*'*(~:*'*)›) **and**
not-member (‹(‹*notation*=‹*infix* ~:››-/ ~: -)› [*51*, *51*] *50*)
end

Set comprehensions

syntax
-Coll :: *pttrn* ⇒ *bool* ⇒ *'a set* (‹(‹*indent*=*1 notation*=‹*mixfix set comprehension*››{-./ -})›)
syntax-consts
-Coll ⇌ *Collect*

translations
  $\{x.\ P\} \rightleftharpoons CONST\ Collect\ (\lambda x.\ P)$

syntax (*ASCII*)
  -Collect :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *'a set* (‹(‹*indent=1 notation=‹mixfix set comprehension›*›{(-/: -)./ -})›)
syntax
  -Collect :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *'a set* (‹(‹*indent=1 notation=‹mixfix set comprehension›*›{(-/ ∈ -)./ -})›)
translations
  $\{p{:}A.\ P\} \rightharpoonup CONST\ Collect\ (\lambda p.\ p \in A \wedge P)$

ML ‹
  fun Collect-binder-tr' c [*Abs* (*x*, *T*, *t*), *Const* (***const-syntax***‹*Collect*›, -) $ *Abs* (*y*, -, *P*)] =
      if x = y then
        let
          val x' = Syntax-Trans.mark-bound-body (*x*, *T*);
          val t' = subst-bound (*x'*, *t*);
          val P' = subst-bound (*x'*, *P*);
        in Syntax.const c $ *Syntax-Trans.mark-bound-abs* (*x*, *T*) $ *P'* $ *t' end*
      else raise Match
  | *Collect-binder-tr'* - - = *raise Match*
›

lemma *CollectI*: $P\ a \Longrightarrow a \in \{x.\ P\ x\}$
  by *simp*

lemma *CollectD*: $a \in \{x.\ P\ x\} \Longrightarrow P\ a$
  by *simp*

lemma *Collect-cong*: $(\bigwedge x.\ P\ x = Q\ x) \Longrightarrow \{x.\ P\ x\} = \{x.\ Q\ x\}$
  by *simp*

Simproc for pulling $x = t$ in $\{x.\ \ldots \wedge x = t \wedge \ldots\}$ to the front (and similarly for $t = x$):

simproc-setup *defined-Collect* ($\{x.\ P\ x \wedge Q\ x\}$) = ‹
  K (*Quantifier1.rearrange-Collect*
    (*fn ctxt =>*
      resolve-tac ctxt @{*thms Collect-cong*} *1 THEN*
      resolve-tac ctxt @{*thms iffI*} *1 THEN*
      ALLGOALS
        (*EVERY'* [*REPEAT-DETERM o eresolve-tac ctxt* @{*thms conjE*},
          DEPTH-SOLVE-1 o (*assume-tac ctxt ORELSE' resolve-tac ctxt* @{*thms conjI*})])))
›

lemmas *CollectE* = *CollectD* [*elim-format*]

lemma *set-eqI*:
  assumes $\bigwedge x.\ x \in A \longleftrightarrow x \in B$
  shows $A = B$
proof −
  from *assms* **have** $\{x.\ x \in A\} = \{x.\ x \in B\}$
    by *simp*
  then show *?thesis* **by** *simp*
qed

lemma *set-eq-iff*: $A = B \longleftrightarrow (\forall x.\ x \in A \longleftrightarrow x \in B)$
  by (*auto intro*:*set-eqI*)

lemma *Collect-eqI*:
  assumes $\bigwedge x.\ P\ x = Q\ x$
  shows *Collect* P = *Collect* Q
  using *assms* **by** (*auto intro*: *set-eqI*)

Lifting of predicate class instances

instantiation *set* :: (*type*) *boolean-algebra*
begin

definition *less-eq-set*
  where $A \leq B \longleftrightarrow (\lambda x.\ member\ x\ A) \leq (\lambda x.\ member\ x\ B)$

definition *less-set*
  where $A < B \longleftrightarrow (\lambda x.\ member\ x\ A) < (\lambda x.\ member\ x\ B)$

definition *inf-set*
  where $A \sqcap B$ = *Collect* $((\lambda x.\ member\ x\ A) \sqcap (\lambda x.\ member\ x\ B))$

definition *sup-set*
  where $A \sqcup B$ = *Collect* $((\lambda x.\ member\ x\ A) \sqcup (\lambda x.\ member\ x\ B))$

definition *bot-set*
  where $\bot$ = *Collect* $\bot$

definition *top-set*
  where $\top$ = *Collect* $\top$

definition *uminus-set*
  where $-\ A$ = *Collect* $(-\ (\lambda x.\ member\ x\ A))$

definition *minus-set*
  where $A - B$ = *Collect* $((\lambda x.\ member\ x\ A) - (\lambda x.\ member\ x\ B))$

instance
  by *standard*
    (*simp-all add*: *less-eq-set-def less-set-def inf-set-def sup-set-def*
      bot-set-def top-set-def uminus-set-def minus-set-def

less-le-not-le sup-inf-distrib1 diff-eq set-eqI fun-eq-iff
del: *inf-apply sup-apply bot-apply top-apply minus-apply uminus-apply*)

end

Set enumerations

abbreviation *empty* :: *'a set* (‹{}›)
where {} ≡ *bot*

definition *insert* :: *'a* ⇒ *'a set* ⇒ *'a set*
where *insert-compr*: *insert a B* = {*x*. *x* = *a* ∨ *x* ∈ *B*}

open-bundle *set-enumeration-syntax*
begin

syntax
-Finset :: *args* ⇒ *'a set* (‹(‹*indent=1 notation=‹mixfix set enumeration›*›{-})›)
syntax-consts
-Finset ⇌ *insert*
translations
{*x*, *xs*} ⇌ *CONST insert x* {*xs*}
{*x*} ⇌ *CONST insert x* {}

end

8.2 Subsets and bounded quantifiers

abbreviation *subset* :: *'a set* ⇒ *'a set* ⇒ *bool*
where *subset* ≡ *less*

abbreviation *subset-eq* :: *'a set* ⇒ *'a set* ⇒ *bool*
where *subset-eq* ≡ *less-eq*

notation
subset (‹*'*(⊂*')*›) **and**
subset (‹(‹*notation=‹infix ⊂›*›-/ ⊂ -)› [*51*, *51*] *50*) **and**
subset-eq (‹*'*(⊆*')*›) **and**
subset-eq (‹(‹*notation=‹infix ⊆›*›-/ ⊆ -)› [*51*, *51*] *50*)

abbreviation (*input*)
supset :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
supset ≡ *greater*

abbreviation (*input*)
supset-eq :: *'a set* ⇒ *'a set* ⇒ *bool* **where**
supset-eq ≡ *greater-eq*

notation
supset (‹*'*(⊃*')*›) **and**

supset (‹(‹*notation*=‹*infix* ⊃››-/ ⊃ -)› [*51*, *51*] *50*) **and**
supset-eq (‹'(⊇')›) **and**
supset-eq (‹(‹*notation*=‹*infix* ⊇››-/ ⊇ -)› [*51*, *51*] *50*)

notation (*ASCII* **output**)
subset (‹'(<')›) **and**
subset (‹(‹*notation*=‹*infix* <››-/ < -)› [*51*, *51*] *50*) **and**
subset-eq (‹'(<=')›) **and**
subset-eq (‹(‹*notation*=‹*infix* <=››-/ <= -)› [*51*, *51*] *50*)

definition *Ball* :: *'a set* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Ball A P* ⟷ (∀ *x*. *x* ∈ *A* ⟶ *P x*) — bounded universal quantifiers

definition *Bex* :: *'a set* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Bex A P* ⟷ (∃ *x*. *x* ∈ *A* ∧ *P x*) — bounded existential quantifiers

syntax (*ASCII*)
-Ball :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder ALL* :›› *ALL* (-/:-)./ -)› [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder EX* :›› *EX* (-/:-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder EX*! :›› *EX*! (-/:-)./ -)› [*0*, *0*, *10*] *10*)
-Bleast :: *id* ⇒ *'a set* ⇒ *bool* ⇒ *'a* (‹(‹*indent*=*3 notation*=‹*binder LEAST* :›› *LEAST* (-/:-)./ -)› [*0*, *0*, *10*] *10*)

syntax (*input*)
-Ball :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder* ! :›› ! (-/:-)./ -)› [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder ?* :›› *?* (-/:-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder ?*! :›› *?*! (-/:-)./ -)› [*0*, *0*, *10*] *10*)

syntax
-Ball :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder* ∀ ›› ∀ (-/∈-)./ -)› [*0*, *0*, *10*] *10*)
-Bex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder* ∃ ›› ∃ (-/∈-)./ -)› [*0*, *0*, *10*] *10*)
-Bex1 :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool* (‹(‹*indent*=*3 notation*=‹*binder* ∃!›› ∃!(-/∈-)./ -)› [*0*, *0*, *10*] *10*)
-Bleast :: *id* ⇒ *'a set* ⇒ *bool* ⇒ *'a* (‹(‹*indent*=*3 notation*=‹*binder LEAST*›› *LEAST*(-/∈-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-Ball ⇌ *Ball* **and**
-Bex ⇌ *Bex* **and**
-Bex1 ⇌ *Ex1* **and**
-Bleast ⇌ *Least*

translations
$\forall x{\in}A.\ P \rightleftharpoons \mathit{CONST\ Ball}\ A\ (\lambda x.\ P)$
$\exists x{\in}A.\ P \rightleftharpoons \mathit{CONST\ Bex}\ A\ (\lambda x.\ P)$
$\exists!x{\in}A.\ P \rightharpoonup \exists!x.\ x \in A \wedge P$
$\mathit{LEAST}\ x{:}A.\ P \rightharpoonup \mathit{LEAST}\ x.\ x \in A \wedge P$

syntax (*ASCII* **output**)
-setlessAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ALL››ALL -<-./ -)›* [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder EX››EX -<-./ -)›* [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ALL››ALL -<=-./ -)›* [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder EX››EX -<=-./ -)›* [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder EX!››EX! -<=-./ -)›* [*0*, *0*, *10*] *10*)

syntax
-setlessAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∀››∀ -⊂-./ -)›* [*0*, *0*, *10*] *10*)
-setlessEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∃››∃ -⊂-./ -)›* [*0*, *0*, *10*] *10*)
-setleAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∀››∀ -⊆-./ -)›* [*0*, *0*, *10*] *10*)
-setleEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∃››∃ -⊆-./ -)›* [*0*, *0*, *10*] *10*)
-setleEx1 :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder ∃!››∃!-⊆-./ -)›* [*0*, *0*, *10*] *10*)

syntax-consts
-setlessAll -setleAll ⇌ *All* **and**
-setlessEx -setleEx ⇌ *Ex* **and**
-setleEx1 ⇌ *Ex1*

translations
$\forall A{\subset}B.\ P \rightharpoonup \forall A.\ A \subset B \longrightarrow P$
$\exists A{\subset}B.\ P \rightharpoonup \exists A.\ A \subset B \wedge P$
$\forall A{\subseteq}B.\ P \rightharpoonup \forall A.\ A \subseteq B \longrightarrow P$
$\exists A{\subseteq}B.\ P \rightharpoonup \exists A.\ A \subseteq B \wedge P$
$\exists!A{\subseteq}B.\ P \rightharpoonup \exists!A.\ A \subseteq B \wedge P$

print-translation ‹
let
val All-binder = Mixfix.binder-name ***const-syntax*** ‹*All*›;
val Ex-binder = Mixfix.binder-name ***const-syntax*** ‹*Ex*›;
val impl = ***const-syntax*** ‹*HOL.implies*›;
val conj = ***const-syntax*** ‹*HOL.conj*›;

```
  val sbset = const-syntax ‹subset›;
  val sbset-eq = const-syntax ‹subset-eq›;

  val trans =
   [((All-binder, impl, sbset), syntax-const ‹-setlessAll›),
    ((All-binder, impl, sbset-eq), syntax-const ‹-setleAll›),
    ((Ex-binder, conj, sbset), syntax-const ‹-setlessEx›),
    ((Ex-binder, conj, sbset-eq), syntax-const ‹-setleEx›)];

  fun mk v (v', T) c n P =
    if v = v' andalso not (Term.exists-subterm (fn Free (x, -) => x = v | - =>
false) n)
    then Syntax.const c $ Syntax-Trans.mark-bound-body (v', T) $ n $ P
    else raise Match;

  fun tr' q = (q, fn - =>
    (fn [Const (syntax-const ‹-bound›, -) $ Free (v, Type ‹set -›),
        Const (c, -) $
          (Const (d, -) $ (Const (syntax-const ‹-bound›, -) $ Free (v', T)) $ n)
$ P] =>
        (case AList.lookup (=) trans (q, c, d) of
          NONE => raise Match
        | SOME l => mk v (v', T) l n P)
      | - => raise Match));
in
  [tr' All-binder, tr' Ex-binder]
end
›
```

Translate between $\{e \mid x1 \ldots xn.\ P\}$ and $\{u.\ \exists x1 \ldots xn.\ u = e \wedge P\}$; $\{y.\ \exists x1 \ldots xn.\ y = e \wedge P\}$ is only translated if $[0..n] \subseteq bvs\ e$.

```
syntax
  -Setcompr :: 'a ⇒ idts ⇒ bool ⇒ 'a set
    (‹(‹indent=1 notation=‹mixfix set comprehension››{- |/-./ -})›)
syntax-consts
  -Setcompr ⇌ Collect

parse-translation ‹
  let
    val ex-tr = snd (Syntax-Trans.mk-binder-tr (EX , const-syntax ‹Ex›));

    fun nvars (Const (syntax-const ‹-idts›, -) $ - $ idts) = nvars idts + 1
      | nvars - = 1;

    fun setcompr-tr ctxt [e, idts, b] =
      let
        val eq = Syntax.const const-syntax ‹HOL.eq› $ Bound (nvars idts) $ e;
        val P = Syntax.const const-syntax ‹HOL.conj› $ eq $ b;
        val exP = ex-tr ctxt [idts, P];
```

```
    in Syntax.const const-syntax ‹Collect› $ absdummy dummyT exP end;

  in [(syntax-const ‹-Setcompr›, setcompr-tr)] end
›

typed-print-translation ‹
[(const-syntax ‹Ball›, Syntax-Trans.preserve-binder-abs2-tr' syntax-const ‹-Ball›),
 (const-syntax ‹Bex›, Syntax-Trans.preserve-binder-abs2-tr' syntax-const ‹-Bex›)]
› — to avoid eta-contraction of body

print-translation ‹
let
  val ex-tr' = snd (Syntax-Trans.mk-binder-tr' (const-syntax ‹Ex›, DUMMY));

  fun setcompr-tr' ctxt [Abs (abs as (-, -, P))] =
    let
      fun check (Const (const-syntax ‹Ex›, -) $ Abs (-, -, P), n) = check (P, n +
1)
        | check (Const (const-syntax ‹HOL.conj›, -) $
            (Const (const-syntax ‹HOL.eq›, -) $ Bound m $ e) $ P, n) =
          n > 0 andalso m = n andalso not (loose-bvar1 (P, n)) andalso
          subset (=) (0 upto (n - 1), add-loose-bnos (e, 0, []))
        | check - = false;

      fun tr' (- $ abs) =
        let val - $ idts $ (- $ (- $ - $ e) $ Q) = ex-tr' ctxt [abs]
        in Syntax.const syntax-const ‹-Setcompr› $ e $ idts $ Q end;
    in
      if check (P, 0) then tr' P
      else
        let
          val (x as - $ Free(xN, -), t) = Syntax-Trans.atomic-abs-tr' ctxt abs;
          val M = Syntax.const syntax-const ‹-Coll› $ x $ t;
        in
          case t of
            Const (const-syntax ‹HOL.conj›, -) $
              (Const (const-syntax ‹Set.member›, -) $
                (Const (syntax-const ‹-bound›, -) $ Free (yN, -)) $ A) $ P =>
              if xN = yN then Syntax.const syntax-const ‹-Collect› $ x $ A $ P else
M
          | - => M
        end
    end;
  in [(const-syntax ‹Collect›, setcompr-tr')] end
›

simproc-setup defined-Bex (∃ x∈A. P x ∧ Q x) = ‹
  K (Quantifier1.rearrange-Bex (fn ctxt => unfold-tac ctxt @{thms Bex-def}))
›
```

simproc-setup *defined-All* $(\forall x{\in}A.\ P\ x \longrightarrow Q\ x) = \langle$
K (*Quantifier1.rearrange-Ball* (*fn ctxt => unfold-tac ctxt* @{*thms Ball-def*}))
$\rangle$

lemma *ballI* [*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \Longrightarrow \forall x{\in}A.\ P\ x$
by (*simp add*: *Ball-def*)

lemmas *strip* = *impI allI ballI*

lemma *bspec* [*dest?*]: $\forall x{\in}A.\ P\ x \Longrightarrow x \in A \Longrightarrow P\ x$
by (*simp add*: *Ball-def*)

Gives better instantiation for bound:

setup $\langle$
map-theory-claset (*fn ctxt =>*
ctxt addbefore (*bspec*, *fn ctxt' => dresolve-tac ctxt'* @{*thms bspec*} *THEN' assume-tac ctxt'*))
$\rangle$

ML $\langle$
structure Simpdata =
struct
open Simpdata;
val mksimps-pairs = [(***const-name*** $\langle$*Ball*$\rangle$, @{*thms bspec*})] @ *mksimps-pairs*;
end;

open Simpdata;
$\rangle$

declaration $\langle$*fn - => Simplifier.map-simpset* (*Simplifier.set-mksimps* (*mksimps mksimps-pairs*))$\rangle$

lemma *ballE* [*elim*]: $\forall x{\in}A.\ P\ x \Longrightarrow (P\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
unfolding *Ball-def* **by** *blast*

lemma *bexI* [*intro*]: $P\ x \Longrightarrow x \in A \Longrightarrow \exists x{\in}A.\ P\ x$
— Normally the best argument order: $P\ x$ constrains the choice of $x \in A$.
unfolding *Bex-def* **by** *blast*

lemma *rev-bexI* [*intro?*]: $x \in A \Longrightarrow P\ x \Longrightarrow \exists x{\in}A.\ P\ x$
— The best argument order when there is only one $x \in A$.
unfolding *Bex-def* **by** *blast*

lemma *bexCI*: $(\forall x{\in}A.\ \neg\ P\ x \Longrightarrow P\ a) \Longrightarrow a \in A \Longrightarrow \exists x{\in}A.\ P\ x$
unfolding *Bex-def* **by** *blast*

lemma *bexE* [*elim*!]: $\exists x{\in}A.\ P\ x \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ x \Longrightarrow Q) \Longrightarrow Q$
unfolding *Bex-def* **by** *blast*

lemma *ball-triv* [*simp*]: $(\forall x{\in}A.\ P) \longleftrightarrow ((\exists x.\ x \in A) \longrightarrow P)$
— trivial rewrite rule.
by (*simp add*: *Ball-def*)

lemma *bex-triv* [*simp*]: $(\exists x{\in}A.\ P) \longleftrightarrow ((\exists x.\ x \in A) \wedge P)$
— Dual form for existentials.
by (*simp add*: *Bex-def*)

lemma *bex-triv-one-point1* [*simp*]: $(\exists x{\in}A.\ x = a) \longleftrightarrow a \in A$
by *blast*

lemma *bex-triv-one-point2* [*simp*]: $(\exists x{\in}A.\ a = x) \longleftrightarrow a \in A$
by *blast*

lemma *bex-one-point1* [*simp*]: $(\exists x{\in}A.\ x = a \wedge P\ x) \longleftrightarrow a \in A \wedge P\ a$
by *blast*

lemma *bex-one-point2* [*simp*]: $(\exists x{\in}A.\ a = x \wedge P\ x) \longleftrightarrow a \in A \wedge P\ a$
by *blast*

lemma *ball-one-point1* [*simp*]: $(\forall x{\in}A.\ x = a \longrightarrow P\ x) \longleftrightarrow (a \in A \longrightarrow P\ a)$
by *blast*

lemma *ball-one-point2* [*simp*]: $(\forall x{\in}A.\ a = x \longrightarrow P\ x) \longleftrightarrow (a \in A \longrightarrow P\ a)$
by *blast*

lemma *ball-conj-distrib*: $(\forall x{\in}A.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall x{\in}A.\ P\ x) \wedge (\forall x{\in}A.\ Q\ x)$
by *blast*

lemma *bex-disj-distrib*: $(\exists x{\in}A.\ P\ x \vee Q\ x) \longleftrightarrow (\exists x{\in}A.\ P\ x) \vee (\exists x{\in}A.\ Q\ x)$
by *blast*

Congruence rules

lemma *ball-cong*:
$[\![\ A = B;\ \bigwedge x.\ x \in B \Longrightarrow P\ x \longleftrightarrow Q\ x\ ]\!] \Longrightarrow$
$(\forall x{\in}A.\ P\ x) \longleftrightarrow (\forall x{\in}B.\ Q\ x)$
by (*simp add*: *Ball-def*)

lemma *ball-cong-simp* [*cong*]:
$[\![\ A = B;\ \bigwedge x.\ x \in B =simp=> P\ x \longleftrightarrow Q\ x\ ]\!] \Longrightarrow$
$(\forall x{\in}A.\ P\ x) \longleftrightarrow (\forall x{\in}B.\ Q\ x)$
by (*simp add*: *simp-implies-def Ball-def*)

lemma *bex-cong*:
$[\![\ A = B;\ \bigwedge x.\ x \in B \Longrightarrow P\ x \longleftrightarrow Q\ x\ ]\!] \Longrightarrow$
$(\exists x{\in}A.\ P\ x) \longleftrightarrow (\exists x{\in}B.\ Q\ x)$
by (*simp add*: *Bex-def cong*: *conj-cong*)

lemma *bex-cong-simp* [*cong*]:
⟦ $A = B$; $\bigwedge x.\ x \in B \Rightarrow P\ x \longleftrightarrow Q\ x$ ⟧ ⟹
($\exists\, x{\in}A.\ P\ x$) ⟷ ($\exists\, x{\in}B.\ Q\ x$)
by (*simp add*: *simp-implies-def Bex-def cong*: *conj-cong*)

lemma *bex1-def*: ($\exists!\, x{\in}X.\ P\ x$) ⟷ ($\exists\, x{\in}X.\ P\ x$) ∧ ($\forall\, x{\in}X.\ \forall\, y{\in}X.\ P\ x \longrightarrow P\ y \longrightarrow x = y$)
by *auto*

8.3 Basic operations

8.3.1 Subsets

lemma *subsetI* [*intro*!]: ($\bigwedge x.\ x \in A \Longrightarrow x \in B$) ⟹ $A \subseteq B$
by (*simp add*: *less-eq-set-def le-fun-def*)

Map the type $'a\ set \Rightarrow anything$ to just $'a$; for overloading constants whose first argument has type $'a\ set$.

lemma *subsetD* [*elim*, *intro?*]: $A \subseteq B \Longrightarrow c \in A \Longrightarrow c \in B$
by (*simp add*: *less-eq-set-def le-fun-def*)
— Rule in Modus Ponens style.

lemma *rev-subsetD* [*intro?*,*no-atp*]: $c \in A \Longrightarrow A \subseteq B \Longrightarrow c \in B$
— The same, with reversed premises for use with *erule* – cf. ⟦*?P*; *?P* ⟶ *?Q*⟧ ⟹ *?Q*.
by (*rule subsetD*)

lemma *subsetCE* [*elim*,*no-atp*]: $A \subseteq B \Longrightarrow (c \notin A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
— Classical elimination rule.
by (*auto simp add*: *less-eq-set-def le-fun-def*)

lemma *subset-eq*: $A \subseteq B \longleftrightarrow (\forall\, x{\in}A.\ x \in B)$
by *blast*

lemma *contra-subsetD* [*no-atp*]: $A \subseteq B \Longrightarrow c \notin B \Longrightarrow c \notin A$
by *blast*

lemma *subset-refl*: $A \subseteq A$
by (*fact order-refl*)

lemma *subset-trans*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow A \subseteq C$
by (*fact order-trans*)

lemma *subset-not-subset-eq* [*code*]: $A \subset B \longleftrightarrow A \subseteq B \wedge \neg\, B \subseteq A$
by (*fact less-le-not-le*)

lemma *eq-mem-trans*: $a = b \Longrightarrow b \in A \Longrightarrow a \in A$
by *simp*

lemmas *basic-trans-rules* [*trans*] =
order-trans-rules rev-subsetD subsetD eq-mem-trans

8.3.2 Equality

lemma *subset-antisym* [*intro*!]: $A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow A = B$
— Anti-symmetry of the subset relation.
by (*iprover intro*: *set-eqI subsetD*)

Equality rules from ZF set theory – are they appropriate here?

lemma *equalityD1*: $A = B \Longrightarrow A \subseteq B$
by *simp*

lemma *equalityD2*: $A = B \Longrightarrow B \subseteq A$
by *simp*

Be careful when adding this to the claset as *subset-empty* is in the simpset: $A = \{\}$ goes to $\{\} \subseteq A$ and $A \subseteq \{\}$ and then back to $A = \{\}$!

lemma *equalityE*: $A = B \Longrightarrow (A \subseteq B \Longrightarrow B \subseteq A \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *equalityCE* [*elim*]: $A = B \Longrightarrow (c \in A \Longrightarrow c \in B \Longrightarrow P) \Longrightarrow (c \notin A \Longrightarrow c \notin B \Longrightarrow P) \Longrightarrow P$
by *blast*

lemma *eqset-imp-iff*: $A = B \Longrightarrow x \in A \longleftrightarrow x \in B$
by *simp*

lemma *eqelem-imp-iff*: $x = y \Longrightarrow x \in A \longleftrightarrow y \in A$
by *simp*

8.3.3 The empty set

lemma *empty-def*: $\{\} = \{x.\ \mathit{False}\}$
by (*simp add*: *bot-set-def bot-fun-def*)

lemma *empty-iff* [*simp*]: $c \in \{\} \longleftrightarrow \mathit{False}$
by (*simp add*: *empty-def*)

lemma *emptyE* [*elim*!]: $a \in \{\} \Longrightarrow P$
by *simp*

lemma *empty-subsetI* [*iff*]: $\{\} \subseteq A$
— One effect is to delete the ASSUMPTION $\{\} \subseteq A$
by *blast*

lemma *equals0I*: $(\bigwedge y.\ y \in A \Longrightarrow \mathit{False}) \Longrightarrow A = \{\}$

by *blast*

lemma *equals0D*: $A = \{\} \Longrightarrow a \notin A$
— Use for reasoning about disjointness: $A \cap B = \{\}$
by *blast*

lemma *ball-empty* [*simp*]: *Ball* $\{\}$ $P \longleftrightarrow$ *True*
by (*simp add*: *Ball-def*)

lemma *bex-empty* [*simp*]: *Bex* $\{\}$ $P \longleftrightarrow$ *False*
by (*simp add*: *Bex-def*)

8.3.4 The universal set – UNIV

abbreviation *UNIV* :: *'a set*
where *UNIV* $\equiv$ *top*

lemma *UNIV-def*: *UNIV* $= \{x.\ True\}$
by (*simp add*: *top-set-def top-fun-def*)

lemma *UNIV-I* [*simp*]: $x \in$ *UNIV*
by (*simp add*: *UNIV-def*)

declare *UNIV-I* [*intro*] — unsafe makes it less likely to cause problems

lemma *UNIV-witness* [*intro?*]: $\exists x.\ x \in$ *UNIV*
by *simp*

lemma *subset-UNIV*: $A \subseteq$ *UNIV*
by (*fact top-greatest*)

Eta-contracting these two rules (to remove P) causes them to be ignored because of their interaction with congruence rules.

lemma *ball-UNIV* [*simp*]: *Ball UNIV P* $\longleftrightarrow$ *All P*
by (*simp add*: *Ball-def*)

lemma *bex-UNIV* [*simp*]: *Bex UNIV P* $\longleftrightarrow$ *Ex P*
by (*simp add*: *Bex-def*)

lemma *UNIV-eq-I*: $(\bigwedge x.\ x \in A) \Longrightarrow$ *UNIV* $= A$
by *auto*

lemma *UNIV-not-empty* [*iff*]: *UNIV* $\neq \{\}$
by (*blast elim*: *equalityE*)

lemma *empty-not-UNIV*[*simp*]: $\{\} \neq$ *UNIV*
by *blast*

8.3.5 The Powerset operator – Pow

definition *Pow* :: *'a set ⇒ 'a set set*
where *Pow-def*: $Pow\ A = \{B.\ B \subseteq A\}$

lemma *Pow-iff* [*iff*]: $A \in Pow\ B \longleftrightarrow A \subseteq B$
by (*simp add*: *Pow-def*)

lemma *PowI*: $A \subseteq B \Longrightarrow A \in Pow\ B$
by (*simp add*: *Pow-def*)

lemma *PowD*: $A \in Pow\ B \Longrightarrow A \subseteq B$
by (*simp add*: *Pow-def*)

lemma *Pow-bottom*: $\{\} \in Pow\ B$
by *simp*

lemma *Pow-top*: $A \in Pow\ A$
by *simp*

lemma *Pow-not-empty*: $Pow\ A \neq \{\}$
using *Pow-top* **by** *blast*

8.3.6 Set complement

lemma *Compl-iff* [*simp*]: $c \in -\ A \longleftrightarrow c \notin A$
by (*simp add*: *fun-Compl-def uminus-set-def*)

lemma *ComplI* [*intro*!]: $(c \in A \Longrightarrow False) \Longrightarrow c \in -\ A$
by (*simp add*: *fun-Compl-def uminus-set-def*) *blast*

This form, with negated conclusion, works well with the Classical prover. Negated assumptions behave like formulae on the right side of the notional turnstile ...

lemma *ComplD* [*dest*!]: $c \in -\ A \Longrightarrow c \notin A$
by *simp*

lemmas *ComplE* = *ComplD* [*elim-format*]

lemma *Compl-eq*: $-\ A = \{x.\ \neg\ x \in A\}$
by *blast*

8.3.7 Binary intersection

abbreviation *inter* :: *'a set ⇒ 'a set ⇒ 'a set* (**infixl** ‹∩› *70*)
where $(\cap) \equiv inf$

notation (*ASCII*)
inter (**infixl** ‹*Int*› *70*)

lemma *Int-def*: $A \cap B = \{x.\ x \in A \wedge x \in B\}$
by (*simp add*: *inf-set-def inf-fun-def*)

lemma *Int-iff* [*simp*]: $c \in A \cap B \longleftrightarrow c \in A \wedge c \in B$
unfolding *Int-def* **by** *blast*

lemma *IntI* [*intro*!]: $c \in A \Longrightarrow c \in B \Longrightarrow c \in A \cap B$
by *simp*

lemma *IntD1*: $c \in A \cap B \Longrightarrow c \in A$
by *simp*

lemma *IntD2*: $c \in A \cap B \Longrightarrow c \in B$
by *simp*

lemma *IntE* [*elim*!]: $c \in A \cap B \Longrightarrow (c \in A \Longrightarrow c \in B \Longrightarrow P) \Longrightarrow P$
by *simp*

8.3.8 Binary union

abbreviation *union* :: *'a set* ⇒ *'a set* ⇒ *'a set* (**infixl** ‹∪› *65*)
where *union* ≡ *sup*

notation (*ASCII*)
union (**infixl** ‹*Un*› *65*)

lemma *Un-def*: $A \cup B = \{x.\ x \in A \vee x \in B\}$
by (*simp add*: *sup-set-def sup-fun-def*)

lemma *Un-iff* [*simp*]: $c \in A \cup B \longleftrightarrow c \in A \vee c \in B$
unfolding *Un-def* **by** *blast*

lemma *UnI1* [*elim?*]: $c \in A \Longrightarrow c \in A \cup B$
by *simp*

lemma *UnI2* [*elim?*]: $c \in B \Longrightarrow c \in A \cup B$
by *simp*

Classical introduction rule: no commitment to A vs. B.

lemma *UnCI* [*intro*!]: $(c \notin B \Longrightarrow c \in A) \Longrightarrow c \in A \cup B$
by *auto*

lemma *UnE* [*elim*!]: $c \in A \cup B \Longrightarrow (c \in A \Longrightarrow P) \Longrightarrow (c \in B \Longrightarrow P) \Longrightarrow P$
unfolding *Un-def* **by** *blast*

lemma *insert-def*: *insert a B* = $\{x.\ x = a\} \cup B$
by (*simp add*: *insert-compr Un-def*)

8.3.9 Set difference

The minus-sign syntax continues to be the default

abbreviation (*input*) *set-difference* :: [*'a set*, *'a set*] ⇒ *'a set*
(**infixl** \ *65*) **where** $A \setminus B \equiv A - B$

lemma *Diff-iff* [*simp*]: $c \in A - B \longleftrightarrow c \in A \wedge c \notin B$
by (*simp add*: *minus-set-def fun-diff-def*)

lemma *DiffI* [*intro*!]: $c \in A \Longrightarrow c \notin B \Longrightarrow c \in A - B$
by *simp*

lemma *DiffD1*: $c \in A - B \Longrightarrow c \in A$
by *simp*

lemma *DiffD2*: $c \in A - B \Longrightarrow c \in B \Longrightarrow P$
by *simp*

lemma *DiffE* [*elim*!]: $c \in A - B \Longrightarrow (c \in A \Longrightarrow c \notin B \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *set-diff-eq*: $A - B = \{x.\ x \in A \wedge x \notin B\}$
by *blast*

lemma *Compl-eq-Diff-UNIV*: $-A = (UNIV - A)$
by *blast*

abbreviation *sym-diff* :: *'a set* ⇒ *'a set* ⇒ *'a set* **where**
sym-diff A B ≡ $((A - B) \cup (B - A))$

8.3.10 Augmenting a set − *insert*

lemma *insert-iff* [*simp*]: $a \in insert\ b\ A \longleftrightarrow a = b \vee a \in A$
unfolding *insert-def* **by** *blast*

lemma *insertI1*: $a \in insert\ a\ B$
by *simp*

lemma *insertI2*: $a \in B \Longrightarrow a \in insert\ b\ B$
by *simp*

lemma *insertE* [*elim*!]: $a \in insert\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a \in A \Longrightarrow P) \Longrightarrow P$
unfolding *insert-def* **by** *blast*

lemma *insertCI* [*intro*!]: $(a \notin B \Longrightarrow a = b) \Longrightarrow a \in insert\ b\ B$
— Classical introduction rule.
by *auto*

lemma *subset-insert-iff*: $A \subseteq \mathit{insert}\ x\ B \longleftrightarrow (\mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subseteq B\ \mathit{else}\ A \subseteq B)$
by *auto*

lemma *set-insert*:
assumes $x \in A$
obtains B **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
proof
show $A = \mathit{insert}\ x\ (A - \{x\})$ **using** *assms* **by** *blast*
show $x \notin A - \{x\}$ **by** *blast*
qed

lemma *insert-ident*: $x \notin A \Longrightarrow x \notin B \Longrightarrow \mathit{insert}\ x\ A = \mathit{insert}\ x\ B \longleftrightarrow A = B$
by *auto*

lemma *insert-eq-iff*:
assumes $a \notin A\ b \notin B$
shows $\mathit{insert}\ a\ A = \mathit{insert}\ b\ B \longleftrightarrow$
$(\mathit{if}\ a = b\ \mathit{then}\ A = B\ \mathit{else}\ \exists C.\ A = \mathit{insert}\ b\ C \wedge b \notin C \wedge B = \mathit{insert}\ a\ C \wedge a \notin C)$
(**is** *?L* $\longleftrightarrow$ *?R*)
proof
show *?R* **if** *?L*
proof (*cases* $a = b$)
case *True*
with *assms* ‹*?L*› **show** *?R*
by (*simp add*: *insert-ident*)
next
case *False*
let $?C = A - \{b\}$
have $A = \mathit{insert}\ b\ ?C \wedge b \notin ?C \wedge B = \mathit{insert}\ a\ ?C \wedge a \notin ?C$
using *assms* ‹*?L*› ‹$a \neq b$› **by** *auto*
then show *?R* **using** ‹$a \neq b$› **by** *auto*
qed
show *?L* **if** *?R*
using *that* **by** (*auto split*: *if-splits*)
qed

lemma *insert-UNIV*[*simp*]: $\mathit{insert}\ x\ \mathit{UNIV} = \mathit{UNIV}$
by *auto*

8.3.11 Singletons, using insert

lemma *singletonI* [*intro*!]: $a \in \{a\}$
— Redundant? But unlike *insertCI*, it proves the subgoal immediately!
by (*rule insertI1*)

lemma *singletonD* [*dest*!]: $b \in \{a\} \Longrightarrow b = a$
by *blast*

lemmas *singletonE* = *singletonD* [*elim-format*]

lemma *singleton-iff*: $b \in \{a\} \longleftrightarrow b = a$
by *blast*

lemma *singleton-inject* [*dest*!]: $\{a\} = \{b\} \Longrightarrow a = b$
by *blast*

lemma *singleton-insert-inj-eq* [*iff*]: $\{b\} = \mathit{insert}\ a\ A \longleftrightarrow a = b \wedge A \subseteq \{b\}$
by *blast*

lemma *singleton-insert-inj-eq'* [*iff*]: $\mathit{insert}\ a\ A = \{b\} \longleftrightarrow a = b \wedge A \subseteq \{b\}$
by *blast*

lemma *subset-singletonD*: $A \subseteq \{x\} \Longrightarrow A = \{\} \vee A = \{x\}$
by *fast*

lemma *subset-singleton-iff*: $X \subseteq \{a\} \longleftrightarrow X = \{\} \vee X = \{a\}$
by *blast*

lemma *subset-singleton-iff-Uniq*: $(\exists\, a.\ A \subseteq \{a\}) \longleftrightarrow (\exists_{\leq 1} x.\ x \in A)$
unfolding *Uniq-def* **by** *blast*

lemma *singleton-conv* [*simp*]: $\{x.\ x = a\} = \{a\}$
by *blast*

lemma *singleton-conv2* [*simp*]: $\{x.\ a = x\} = \{a\}$
by *blast*

lemma *Diff-single-insert*: $A - \{x\} \subseteq B \Longrightarrow A \subseteq \mathit{insert}\ x\ B$
by *blast*

lemma *subset-Diff-insert*: $A \subseteq B - \mathit{insert}\ x\ C \longleftrightarrow A \subseteq B - C \wedge x \notin A$
by *blast*

lemma *doubleton-eq-iff*: $\{a, b\} = \{c, d\} \longleftrightarrow a = c \wedge b = d \vee a = d \wedge b = c$
by (*blast elim*: *equalityE*)

lemma *Un-singleton-iff*: $A \cup B = \{x\} \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
by *auto*

lemma *singleton-Un-iff*: $\{x\} = A \cup B \longleftrightarrow A = \{\} \wedge B = \{x\} \vee A = \{x\} \wedge B = \{\} \vee A = \{x\} \wedge B = \{x\}$
by *auto*

8.3.12 Image of a set under a function

Frequently b does not have the syntactic form of $f\ x$.

definition *image* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set$ (**infixr** ‹‘› *90*)
where $f\ `\ A = \{y.\ \exists x{\in}A.\ y = f\ x\}$

lemma *image-eqI* [*simp*, *intro*]: $b = f\ x \Longrightarrow x \in A \Longrightarrow b \in f\ `\ A$
unfolding *image-def* **by** *blast*

lemma *imageI*: $x \in A \Longrightarrow f\ x \in f\ `\ A$
by (*rule image-eqI*) (*rule refl*)

lemma *rev-image-eqI*: $x \in A \Longrightarrow b = f\ x \Longrightarrow b \in f\ `\ A$
— This version's more effective when we already have the required x.
by (*rule image-eqI*)

lemma *imageE* [*elim*!]:
assumes $b \in (\lambda x.\ f\ x)\ `\ A$ — The eta-expansion gives variable-name preservation.
obtains x **where** $b = f\ x$ **and** $x \in A$
using *assms* **unfolding** *image-def* **by** *blast*

lemma *Compr-image-eq*: $\{x \in f\ `\ A.\ P\ x\} = f\ `\ \{x \in A.\ P\ (f\ x)\}$
by *auto*

lemma *image-Un*: $f\ `\ (A \cup B) = f\ `\ A \cup f\ `\ B$
by *blast*

lemma *image-iff*: $z \in f\ `\ A \longleftrightarrow (\exists x{\in}A.\ z = f\ x)$
by *blast*

lemma *image-subsetI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f\ `\ A \subseteq B$
— Replaces the three steps *subsetI*, *imageE*, *hypsubst*, but breaks too many existing proofs.
by *blast*

lemma *image-subset-iff*: $f\ `\ A \subseteq B \longleftrightarrow (\forall x{\in}A.\ f\ x \in B)$
— This rewrite rule would confuse users if made default.
by *blast*

lemma *subset-imageE*:
assumes $B \subseteq f\ `\ A$
obtains C **where** $C \subseteq A$ **and** $B = f\ `\ C$
proof −
from *assms* **have** $B = f\ `\ \{a \in A.\ f\ a \in B\}$ **by** *fast*
moreover have $\{a \in A.\ f\ a \in B\} \subseteq A$ **by** *blast*
ultimately show *thesis* **by** (*blast intro*: *that*)
qed

lemma *subset-image-iff*: $B \subseteq f\ `\ A \longleftrightarrow (\exists AA{\subseteq}A.\ B = f\ `\ AA)$

by (*blast elim*: *subset-imageE*)

lemma *image-ident* [*simp*]: $(\lambda x.\ x)\ `\ Y = Y$
by *blast*

lemma *image-empty* [*simp*]: $f\ `\ \{\} = \{\}$
by *blast*

lemma *image-insert* [*simp*]: $f\ `\ insert\ a\ B = insert\ (f\ a)\ (f\ `\ B)$
by *blast*

lemma *image-constant*: $x \in A \Longrightarrow (\lambda x.\ c)\ `\ A = \{c\}$
by *auto*

lemma *image-constant-conv*: $(\lambda x.\ c)\ `\ A = (if\ A = \{\}\ then\ \{\}\ else\ \{c\})$
by *auto*

lemma *image-image*: $f\ `\ (g\ `\ A) = (\lambda x.\ f\ (g\ x))\ `\ A$
by *blast*

lemma *insert-image* [*simp*]: $x \in A \Longrightarrow insert\ (f\ x)\ (f\ `\ A) = f\ `\ A$
by *blast*

lemma *image-is-empty* [*iff*]: $f\ `\ A = \{\} \longleftrightarrow A = \{\}$
by *blast*

lemma *empty-is-image* [*iff*]: $\{\} = f\ `\ A \longleftrightarrow A = \{\}$
by *blast*

lemma *image-Collect*: $f\ `\ \{x.\ P\ x\} = \{f\ x \mid x.\ P\ x\}$
— NOT suitable as a default simp rule: the RHS isn't simpler than the LHS, with its implicit quantifier and conjunction. Also image enjoys better equational properties than does the RHS.
by *blast*

lemma *if-image-distrib* [*simp*]:
$(\lambda x.\ if\ P\ x\ then\ f\ x\ else\ g\ x)\ `\ S = f\ `\ (S \cap \{x.\ P\ x\}) \cup g\ `\ (S \cap \{x.\ \neg\ P\ x\})$
by *auto*

lemma *image-cong*:
$f\ `\ M = g\ `\ N$ **if** $M = N\ \bigwedge x.\ x \in N \Longrightarrow f\ x = g\ x$
using *that* **by** (*simp add*: *image-def*)

lemma *image-cong-simp* [*cong*]:
$f\ `\ M = g\ `\ N$ **if** $M = N\ \bigwedge x.\ x \in N\ =simp=>\ f\ x = g\ x$
using *that image-cong* [*of M N f g*] **by** (*simp add*: *simp-implies-def*)

lemma *image-Int-subset*: $f\ `\ (A \cap B) \subseteq f\ `\ A \cap f\ `\ B$
by *blast*

lemma *image-diff-subset*: $f\ `\ A - f\ `\ B \subseteq f\ `\ (A - B)$
by *blast*

lemma *Setcompr-eq-image*: $\{f\ x\ |x.\ x \in A\} = f\ `\ A$
by *blast*

lemma *setcompr-eq-image*: $\{f\ x\ |x.\ P\ x\} = f\ `\ \{x.\ P\ x\}$
by *auto*

lemma *ball-imageD*: $\forall x{\in}f\ `\ A.\ P\ x \Longrightarrow \forall x{\in}A.\ P\ (f\ x)$
by *simp*

lemma *bex-imageD*: $\exists x{\in}f\ `\ A.\ P\ x \Longrightarrow \exists x{\in}A.\ P\ (f\ x)$
by *auto*

lemma *image-add-0* [*simp*]: $(+)\ (0{::}'a{::}\textit{comm-monoid-add})\ `\ S = S$
by *auto*

theorem *Cantors-theorem*: $\nexists f.\ f\ `\ A = Pow\ A$
proof
assume $\exists f.\ f\ `\ A = Pow\ A$
then obtain f **where** f: $f\ `\ A = Pow\ A$ **..**
let $?X = \{a \in A.\ a \notin f\ a\}$
have $?X \in Pow\ A$ **by** *blast*
then have $?X \in f\ `\ A$ **by** (*simp only*: f)
then obtain x **where** $x \in A$ **and** $f\ x = ?X$ **by** *blast*
then show *False* **by** *blast*
qed

Range of a function – just an abbreviation for image!

abbreviation *range* :: $('a \Rightarrow 'b) \Rightarrow 'b\ set$ — of function
where $range\ f \equiv f\ `\ UNIV$

lemma *range-eqI*: $b = f\ x \Longrightarrow b \in range\ f$
by *simp*

lemma *rangeI*: $f\ x \in range\ f$
by *simp*

lemma *rangeE* [*elim?*]: $b \in range\ (\lambda x.\ f\ x) \Longrightarrow (\bigwedge x.\ b = f\ x \Longrightarrow P) \Longrightarrow P$
by (*rule imageE*)

lemma *range-subsetD*: $range\ f \subseteq B \Longrightarrow f\ i \in B$
by *blast*

lemma *full-SetCompr-eq*: $\{u.\ \exists x.\ u = f\ x\} = range\ f$
by *auto*

lemma *range-composition*: *range* (λ*x*. *f* (*g x*)) = *f* ‘ *range g*
by *auto*

lemma *range-constant* [*simp*]: *range* (λ-. *x*) = {*x*}
by (*simp add*: *image-constant*)

lemma *range-eq-singletonD*: *range f* = {*a*} ⟹ *f x* = *a*
by *auto*

8.3.13 Some rules with *if*

Elimination of {*x*. ... ∧ *x* = *t* ∧ ...}.

lemma *Collect-conv-if*: {*x*. *x* = *a* ∧ *P x*} = (*if P a then* {*a*} *else* {})
by *auto*

lemma *Collect-conv-if2*: {*x*. *a* = *x* ∧ *P x*} = (*if P a then* {*a*} *else* {})
by *auto*

Rewrite rules for boolean case-splitting: faster than *if-split* [*split*].

lemma *if-split-eq1*: (*if Q then x else y*) = *b* ⟷ (*Q* ⟶ *x* = *b*) ∧ (¬ *Q* ⟶ *y* = *b*)
by (*rule if-split*)

lemma *if-split-eq2*: *a* = (*if Q then x else y*) ⟷ (*Q* ⟶ *a* = *x*) ∧ (¬ *Q* ⟶ *a* = *y*)
by (*rule if-split*)

Split ifs on either side of the membership relation. Not for [*simp*] – can cause goals to blow up!

lemma *if-split-mem1*: (*if Q then x else y*) ∈ *b* ⟷ (*Q* ⟶ *x* ∈ *b*) ∧ (¬ *Q* ⟶ *y* ∈ *b*)
by (*rule if-split*)

lemma *if-split-mem2*: (*a* ∈ (*if Q then x else y*)) ⟷ (*Q* ⟶ *a* ∈ *x*) ∧ (¬ *Q* ⟶ *a* ∈ *y*)
by (*rule if-split* [**where** *P* = λ*S*. *a* ∈ *S*])

lemmas *split-ifs* = *if-bool-eq-conj if-split-eq1 if-split-eq2 if-split-mem1 if-split-mem2*

8.4 Further operations and lemmas

8.4.1 The “proper subset” relation

lemma *psubsetI* [*intro*!]: *A* ⊆ *B* ⟹ *A* ≠ *B* ⟹ *A* ⊂ *B*
unfolding *less-le* **by** *blast*

lemma *psubsetE* [*elim*!]: *A* ⊂ *B* ⟹ (*A* ⊆ *B* ⟹ ¬ *B* ⊆ *A* ⟹ *R*) ⟹ *R*
unfolding *less-le* **by** *blast*

lemma *psubset-insert-iff*:
$A \subset \mathit{insert}\ x\ B \longleftrightarrow (\mathit{if}\ x \in B\ \mathit{then}\ A \subset B\ \mathit{else}\ \mathit{if}\ x \in A\ \mathit{then}\ A - \{x\} \subset B\ \mathit{else}\ A \subseteq B)$
by (*auto simp add*: *less-le subset-insert-iff*)

lemma *psubset-eq*: $A \subset B \longleftrightarrow A \subseteq B \wedge A \neq B$
by (*simp only*: *less-le*)

lemma *psubset-imp-subset*: $A \subset B \Longrightarrow A \subseteq B$
by (*simp add*: *psubset-eq*)

lemma *psubset-trans*: $A \subset B \Longrightarrow B \subset C \Longrightarrow A \subset C$
unfolding *less-le* **by** (*auto dest*: *subset-antisym*)

lemma *psubsetD*: $A \subset B \Longrightarrow c \in A \Longrightarrow c \in B$
unfolding *less-le* **by** (*auto dest*: *subsetD*)

lemma *psubset-subset-trans*: $A \subset B \Longrightarrow B \subseteq C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *subset-psubset-trans*: $A \subseteq B \Longrightarrow B \subset C \Longrightarrow A \subset C$
by (*auto simp add*: *psubset-eq*)

lemma *psubset-imp-ex-mem*: $A \subset B \Longrightarrow \exists\, b.\ b \in B - A$
unfolding *less-le* **by** *blast*

lemma *atomize-ball*: $(\bigwedge x.\ x \in A \Longrightarrow P\ x) \equiv \mathit{Trueprop}\ (\forall\, x{\in}A.\ P\ x)$
by (*simp only*: *Ball-def atomize-all atomize-imp*)

lemmas [*symmetric*, *rulify*] = *atomize-ball*
and [*symmetric*, *defn*] = *atomize-ball*

lemma *image-Pow-mono*: $f\ `\ A \subseteq B \Longrightarrow \mathit{image}\ f\ `\ \mathit{Pow}\ A \subseteq \mathit{Pow}\ B$
by *blast*

lemma *image-Pow-surj*: $f\ `\ A = B \Longrightarrow \mathit{image}\ f\ `\ \mathit{Pow}\ A = \mathit{Pow}\ B$
by (*blast elim*: *subset-imageE*)

8.4.2 Derived rules involving subsets.

insert.

lemma *subset-insertI*: $B \subseteq \mathit{insert}\ a\ B$
by (*rule subsetI*) (*erule insertI2*)

lemma *subset-insertI2*: $A \subseteq B \Longrightarrow A \subseteq \mathit{insert}\ b\ B$
by *blast*

lemma *subset-insert*: $x \notin A \Longrightarrow A \subseteq \mathit{insert}\ x\ B \longleftrightarrow A \subseteq B$
by *blast*

Finite Union – the least upper bound of two sets.

lemma *Un-upper1*: $A \subseteq A \cup B$
by (*fact sup-ge1*)

lemma *Un-upper2*: $B \subseteq A \cup B$
by (*fact sup-ge2*)

lemma *Un-least*: $A \subseteq C \Longrightarrow B \subseteq C \Longrightarrow A \cup B \subseteq C$
by (*fact sup-least*)

Finite Intersection – the greatest lower bound of two sets.

lemma *Int-lower1*: $A \cap B \subseteq A$
by (*fact inf-le1*)

lemma *Int-lower2*: $A \cap B \subseteq B$
by (*fact inf-le2*)

lemma *Int-greatest*: $C \subseteq A \Longrightarrow C \subseteq B \Longrightarrow C \subseteq A \cap B$
by (*fact inf-greatest*)

Set difference.

lemma *Diff-subset*[*simp*]: $A - B \subseteq A$
by *blast*

lemma *Diff-subset-conv*: $A - B \subseteq C \longleftrightarrow A \subseteq B \cup C$
by *blast*

8.4.3 Equalities involving union, intersection, inclusion, etc.

{}.

lemma *Collect-const* [*simp*]: $\{s.\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
— supersedes *Collect-False-empty*
by *auto*

lemma *subset-empty* [*simp*]: $A \subseteq \{\} \longleftrightarrow A = \{\}$
by (*fact bot-unique*)

lemma *not-psubset-empty* [*iff*]: $\neg\ (A < \{\})$
by (*fact not-less-bot*)

lemma *Collect-subset* [*simp*]: $\{x{\in}A.\ P\ x\} \subseteq A$ **by** *auto*

lemma *Collect-empty-eq* [*simp*]: $\mathit{Collect}\ P = \{\} \longleftrightarrow (\forall x.\ \neg\ P\ x)$
by *blast*

lemma *empty-Collect-eq* [*simp*]: $\{\} = \mathit{Collect}\ P \longleftrightarrow (\forall x.\ \neg\ P\ x)$
by *blast*

lemma *Collect-neg-eq*: $\{x.\ \neg\ P\ x\} = -\ \{x.\ P\ x\}$
by *blast*

lemma *Collect-disj-eq*: $\{x.\ P\ x \vee Q\ x\} = \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-imp-eq*: $\{x.\ P\ x \longrightarrow Q\ x\} = -\ \{x.\ P\ x\} \cup \{x.\ Q\ x\}$
by *blast*

lemma *Collect-conj-eq*: $\{x.\ P\ x \wedge Q\ x\} = \{x.\ P\ x\} \cap \{x.\ Q\ x\}$
by *blast*

lemma *Collect-conj-eq2*: $\{x \in A.\ P\ x \wedge Q\ x\} = \{x \in A.\ P\ x\} \cap \{x \in A.\ Q\ x\}$
by *blast*

lemma *Collect-mono-iff*: $Collect\ P \subseteq Collect\ Q \longleftrightarrow (\forall x.\ P\ x \longrightarrow Q\ x)$
by *blast*

insert.

lemma *insert-is-Un*: $insert\ a\ A = \{a\} \cup A$
— NOT SUITABLE FOR REWRITING since $\{a\} \equiv insert\ a\ \{\}$
by *blast*

lemma *insert-not-empty* [*simp*]: $insert\ a\ A \neq \{\}$
and *empty-not-insert* [*simp*]: $\{\} \neq insert\ a\ A$
by *blast+*

lemma *insert-absorb*: $a \in A \Longrightarrow insert\ a\ A = A$
— [*simp*] causes recursive calls when there are nested inserts
— with *quadratic* running time
by *blast*

lemma *insert-absorb2* [*simp*]: $insert\ x\ (insert\ x\ A) = insert\ x\ A$
by *blast*

lemma *insert-commute*: $insert\ x\ (insert\ y\ A) = insert\ y\ (insert\ x\ A)$
by *blast*

lemma *insert-subset* [*simp*]: $insert\ x\ A \subseteq B \longleftrightarrow x \in B \wedge A \subseteq B$
by *blast*

lemma *mk-disjoint-insert*: $a \in A \Longrightarrow \exists B.\ A = insert\ a\ B \wedge a \notin B$
— use new B rather than $A - \{a\}$ to avoid infinite unfolding
by (*rule exI* [**where** $x = A - \{a\}$]) *blast*

lemma *insert-Collect*: $insert\ a\ (Collect\ P) = \{u.\ u \neq a \longrightarrow P\ u\}$
by *auto*

lemma *insert-inter-insert* [*simp*]: *insert a A* ∩ *insert a B* = *insert a* (*A* ∩ *B*)
by *blast*

lemma *insert-disjoint* [*simp*]:
insert a A ∩ *B* = {} ⟷ *a* ∉ *B* ∧ *A* ∩ *B* = {}
{} = *insert a A* ∩ *B* ⟷ *a* ∉ *B* ∧ {} = *A* ∩ *B*
by *auto*

lemma *disjoint-insert* [*simp*]:
B ∩ *insert a A* = {} ⟷ *a* ∉ *B* ∧ *B* ∩ *A* = {}
{} = *A* ∩ *insert b B* ⟷ *b* ∉ *A* ∧ {} = *A* ∩ *B*
by *auto*

Int

lemma *Int-absorb*: *A* ∩ *A* = *A*
by (*fact inf-idem*)

lemma *Int-left-absorb*: *A* ∩ (*A* ∩ *B*) = *A* ∩ *B*
by (*fact inf-left-idem*)

lemma *Int-commute*: *A* ∩ *B* = *B* ∩ *A*
by (*fact inf-commute*)

lemma *Int-left-commute*: *A* ∩ (*B* ∩ *C*) = *B* ∩ (*A* ∩ *C*)
by (*fact inf-left-commute*)

lemma *Int-assoc*: (*A* ∩ *B*) ∩ *C* = *A* ∩ (*B* ∩ *C*)
by (*fact inf-assoc*)

lemmas *Int-ac* = *Int-assoc Int-left-absorb Int-commute Int-left-commute*
— Intersection is an AC-operator

lemma *Int-absorb1*: *B* ⊆ *A* ⟹ *A* ∩ *B* = *B*
by (*fact inf-absorb2*)

lemma *Int-absorb2*: *A* ⊆ *B* ⟹ *A* ∩ *B* = *A*
by (*fact inf-absorb1*)

lemma *Int-empty-left*: {} ∩ *B* = {}
by (*fact inf-bot-left*)

lemma *Int-empty-right*: *A* ∩ {} = {}
by (*fact inf-bot-right*)

lemma *disjoint-eq-subset-Compl*: *A* ∩ *B* = {} ⟷ *A* ⊆ − *B*
by *blast*

lemma *disjoint-iff*: *A* ∩ *B* = {} ⟷ (∀ *x*. *x*∈*A* ⟶ *x* ∉ *B*)
by *blast*

lemma *disjoint-iff-not-equal*: $A \cap B = \{\} \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}B.\ x \neq y)$
by *blast*

lemma *Int-UNIV-left*: $UNIV \cap B = B$
by (*fact inf-top-left*)

lemma *Int-UNIV-right*: $A \cap UNIV = A$
by (*fact inf-top-right*)

lemma *Int-Un-distrib*: $A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$
by (*fact inf-sup-distrib1*)

lemma *Int-Un-distrib2*: $(B \cup C) \cap A = (B \cap A) \cup (C \cap A)$
by (*fact inf-sup-distrib2*)

lemma *Int-UNIV*: $A \cap B = UNIV \longleftrightarrow A = UNIV \wedge B = UNIV$
by (*fact inf-eq-top-iff*)

lemma *Int-subset-iff*: $C \subseteq A \cap B \longleftrightarrow C \subseteq A \wedge C \subseteq B$
by (*fact le-inf-iff*)

lemma *Int-Collect*: $x \in A \cap \{x.\ P\ x\} \longleftrightarrow x \in A \wedge P\ x$
by *blast*

Un.

lemma *Un-absorb*: $A \cup A = A$
by (*fact sup-idem*)

lemma *Un-left-absorb*: $A \cup (A \cup B) = A \cup B$
by (*fact sup-left-idem*)

lemma *Un-commute*: $A \cup B = B \cup A$
by (*fact sup-commute*)

lemma *Un-left-commute*: $A \cup (B \cup C) = B \cup (A \cup C)$
by (*fact sup-left-commute*)

lemma *Un-assoc*: $(A \cup B) \cup C = A \cup (B \cup C)$
by (*fact sup-assoc*)

lemmas *Un-ac = Un-assoc Un-left-absorb Un-commute Un-left-commute*
— Union is an AC-operator

lemma *Un-absorb1*: $A \subseteq B \Longrightarrow A \cup B = B$
by (*fact sup-absorb2*)

lemma *Un-absorb2*: $B \subseteq A \Longrightarrow A \cup B = A$
by (*fact sup-absorb1*)

lemma *Un-empty-left*: $\{\} \cup B = B$
by (*fact sup-bot-left*)

lemma *Un-empty-right*: $A \cup \{\} = A$
by (*fact sup-bot-right*)

lemma *Un-UNIV-left*: $UNIV \cup B = UNIV$
by (*fact sup-top-left*)

lemma *Un-UNIV-right*: $A \cup UNIV = UNIV$
by (*fact sup-top-right*)

lemma *Un-insert-left* [*simp*]: $(insert\ a\ B) \cup C = insert\ a\ (B \cup C)$
by *blast*

lemma *Un-insert-right* [*simp*]: $A \cup (insert\ a\ B) = insert\ a\ (A \cup B)$
by *blast*

lemma *Int-insert-left*: $(insert\ a\ B) \cap C = (if\ a \in C\ then\ insert\ a\ (B \cap C)\ else\ B \cap C)$
by *auto*

lemma *Int-insert-left-if0* [*simp*]: $a \notin C \Longrightarrow (insert\ a\ B) \cap C = B \cap C$
by *auto*

lemma *Int-insert-left-if1* [*simp*]: $a \in C \Longrightarrow (insert\ a\ B) \cap C = insert\ a\ (B \cap C)$
by *auto*

lemma *Int-insert-right*: $A \cap (insert\ a\ B) = (if\ a \in A\ then\ insert\ a\ (A \cap B)\ else\ A \cap B)$
by *auto*

lemma *Int-insert-right-if0* [*simp*]: $a \notin A \Longrightarrow A \cap (insert\ a\ B) = A \cap B$
by *auto*

lemma *Int-insert-right-if1* [*simp*]: $a \in A \Longrightarrow A \cap (insert\ a\ B) = insert\ a\ (A \cap B)$
by *auto*

lemma *Un-Int-distrib*: $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
by (*fact sup-inf-distrib1*)

lemma *Un-Int-distrib2*: $(B \cap C) \cup A = (B \cup A) \cap (C \cup A)$
by (*fact sup-inf-distrib2*)

lemma *Un-Int-crazy*: $(A \cap B) \cup (B \cap C) \cup (C \cap A) = (A \cup B) \cap (B \cup C) \cap (C \cup A)$
by *blast*

lemma *subset-Un-eq*: $A \subseteq B \longleftrightarrow A \cup B = B$
by (*fact le-iff-sup*)

lemma *Un-empty* [*iff*]: $A \cup B = \{\} \longleftrightarrow A = \{\} \wedge B = \{\}$
by (*fact sup-eq-bot-iff*)

lemma *Un-subset-iff*: $A \cup B \subseteq C \longleftrightarrow A \subseteq C \wedge B \subseteq C$
by (*fact le-sup-iff*)

lemma *Un-Diff-Int*: $(A - B) \cup (A \cap B) = A$
by *blast*

lemma *Diff-Int2*: $A \cap C - B \cap C = A \cap C - B$
by *blast*

lemma *subset-UnE*:
assumes $C \subseteq A \cup B$
obtains A' B' **where** $A' \subseteq A$ $B' \subseteq B$ $C = A' \cup B'$
proof
show $C \cap A \subseteq A$ $C \cap B \subseteq B$ $C = (C \cap A) \cup (C \cap B)$
using *assms* **by** *blast+*
qed

lemma *Un-Int-eq* [*simp*]: $(S \cup T) \cap S = S$ $(S \cup T) \cap T = T$ $S \cap (S \cup T) = S$ $T \cap (S \cup T) = T$
by *auto*

lemma *Int-Un-eq* [*simp*]: $(S \cap T) \cup S = S$ $(S \cap T) \cup T = T$ $S \cup (S \cap T) = S$ $T \cup (S \cap T) = T$
by *auto*

Set complement

lemma *Compl-disjoint* [*simp*]: $A \cap - A = \{\}$
by (*fact inf-compl-bot*)

lemma *Compl-disjoint2* [*simp*]: $- A \cap A = \{\}$
by (*fact compl-inf-bot*)

lemma *Compl-partition*: $A \cup - A = UNIV$
by (*fact sup-compl-top*)

lemma *Compl-partition2*: $- A \cup A = UNIV$
by (*fact compl-sup-top*)

lemma *double-complement*: $- (-A) = A$ **for** $A :: 'a\ set$
by (*fact double-compl*)

lemma *Compl-Un*: $- (A \cup B) = (- A) \cap (- B)$

by (*fact compl-sup*)

lemma *Compl-Int*: $-(A \cap B) = (-A) \cup (-B)$
by (*fact compl-inf*)

lemma *subset-Compl-self-eq*: $A \subseteq -A \longleftrightarrow A = \{\}$
by *blast*

lemma *Un-Int-assoc-eq*: $(A \cap B) \cup C = A \cap (B \cup C) \longleftrightarrow C \subseteq A$
— Halmos, Naive Set Theory, page 16.
by *blast*

lemma *Compl-UNIV-eq*: $-UNIV = \{\}$
by (*fact compl-top-eq*)

lemma *Compl-empty-eq*: $-\{\} = UNIV$
by (*fact compl-bot-eq*)

lemma *Compl-subset-Compl-iff* [*iff*]: $-A \subseteq -B \longleftrightarrow B \subseteq A$
by (*fact compl-le-compl-iff*)

lemma *Compl-eq-Compl-iff* [*iff*]: $-A = -B \longleftrightarrow A = B$
for $A\ B :: 'a\ set$
by (*fact compl-eq-compl-iff*)

lemma *Compl-insert*: $-(insert\ x\ A) = (-A) - \{x\}$
by *blast*

Bounded quantifiers.

The following are not added to the default simpset because (a) they duplicate the body and (b) there are no similar rules for *Int*.

lemma *ball-Un*: $(\forall x \in A \cup B.\ P\ x) \longleftrightarrow (\forall x{\in}A.\ P\ x) \wedge (\forall x{\in}B.\ P\ x)$
by *blast*

lemma *bex-Un*: $(\exists x \in A \cup B.\ P\ x) \longleftrightarrow (\exists x{\in}A.\ P\ x) \vee (\exists x{\in}B.\ P\ x)$
by *blast*

Set difference.

lemma *Diff-eq*: $A - B = A \cap (-B)$
by(*rule boolean-algebra-class.diff-eq*)

lemma *Diff-eq-empty-iff*: $A - B = \{\} \longleftrightarrow A \subseteq B$
by(*rule boolean-algebra-class.diff-shunt-var*)

lemma *Diff-cancel* [*simp*]: $A - A = \{\}$
by *blast*

lemma *Diff-idemp* [*simp*]: $(A - B) - B = A - B$

for A B :: $'a$ *set*
by *blast*

lemma *Diff-triv*: $A \cap B = \{\} \Longrightarrow A - B = A$
by (*blast elim*: *equalityE*)

lemma *empty-Diff* [*simp*]: $\{\} - A = \{\}$
by *blast*

lemma *Diff-empty* [*simp*]: $A - \{\} = A$
by *blast*

lemma *Diff-UNIV* [*simp*]: $A - UNIV = \{\}$
by *blast*

lemma *Diff-insert0* [*simp*]: $x \notin A \Longrightarrow A - insert\ x\ B = A - B$
by *blast*

lemma *Diff-insert*: $A - insert\ a\ B = A - B - \{a\}$
— NOT SUITABLE FOR REWRITING since $\{a\} \equiv insert\ a\ 0$
by *blast*

lemma *Diff-insert2*: $A - insert\ a\ B = A - \{a\} - B$
— NOT SUITABLE FOR REWRITING since $\{a\} \equiv insert\ a\ 0$
by *blast*

lemma *insert-Diff-if*: $insert\ x\ A - B = (if\ x \in B\ then\ A - B\ else\ insert\ x\ (A - B))$
by *auto*

lemma *insert-Diff1* [*simp*]: $x \in B \Longrightarrow insert\ x\ A - B = A - B$
by *blast*

lemma *insert-Diff-single*[*simp*]: $insert\ a\ (A - \{a\}) = insert\ a\ A$
by *blast*

lemma *insert-Diff*: $a \in A \Longrightarrow insert\ a\ (A - \{a\}) = A$
by *blast*

lemma *Diff-insert-absorb*: $x \notin A \Longrightarrow (insert\ x\ A) - \{x\} = A$
by *auto*

lemma *Diff-disjoint* [*simp*]: $A \cap (B - A) = \{\}$
by *blast*

lemma *Diff-partition*: $A \subseteq B \Longrightarrow A \cup (B - A) = B$
by *blast*

lemma *double-diff*: $A \subseteq B \Longrightarrow B \subseteq C \Longrightarrow B - (C - A) = A$

by *blast*

lemma *Un-Diff-cancel* [*simp*]: $A \cup (B - A) = A \cup B$
by *blast*

lemma *Un-Diff-cancel2* [*simp*]: $(B - A) \cup A = B \cup A$
by *blast*

lemma *Diff-Un*: $A - (B \cup C) = (A - B) \cap (A - C)$
by *blast*

lemma *Diff-Int*: $A - (B \cap C) = (A - B) \cup (A - C)$
by *blast*

lemma *Diff-Diff-Int*: $A - (A - B) = A \cap B$
by *blast*

lemma *Un-Diff*: $(A \cup B) - C = (A - C) \cup (B - C)$
by *blast*

lemma *Int-Diff*: $(A \cap B) - C = A \cap (B - C)$
by *blast*

lemma *Diff-Int-distrib*: $C \cap (A - B) = (C \cap A) - (C \cap B)$
by *blast*

lemma *Diff-Int-distrib2*: $(A - B) \cap C = (A \cap C) - (B \cap C)$
by *blast*

lemma *Diff-Compl* [*simp*]: $A - (- B) = A \cap B$
by *auto*

lemma *Compl-Diff-eq* [*simp*]: $- (A - B) = - A \cup B$
by *blast*

lemma *subset-Compl-singleton* [*simp*]: $A \subseteq - \{b\} \longleftrightarrow b \notin A$
by *blast*

Quantification over type *bool*.

lemma *bool-induct*: $P\ True \Longrightarrow P\ False \Longrightarrow P\ x$
by (*cases x*) *auto*

lemma *all-bool-eq*: $(\forall b.\ P\ b) \longleftrightarrow P\ True \wedge P\ False$
by (*auto intro*: *bool-induct*)

lemma *bool-contrapos*: $P\ x \Longrightarrow \neg\ P\ False \Longrightarrow P\ True$
by (*cases x*) *auto*

lemma *ex-bool-eq*: $(\exists b.\ P\ b) \longleftrightarrow P\ True \vee P\ False$

by (*auto intro*: *bool-contrapos*)

lemma *UNIV-bool*: $UNIV = \{False, True\}$
by (*auto intro*: *bool-induct*)

Pow

lemma *Pow-empty* [*simp*]: $Pow\ \{\} = \{\{\}\}$
by (*auto simp add*: *Pow-def*)

lemma *Pow-singleton-iff* [*simp*]: $Pow\ X = \{Y\} \longleftrightarrow X = \{\} \wedge Y = \{\}$
by *blast*

lemma *Pow-insert*: $Pow\ (insert\ a\ A) = Pow\ A \cup (insert\ a\ `\ Pow\ A)$
by (*blast intro*: *image-eqI* [**where** *?x* $= u - \{a\}$ **for** *u*])

lemma *Pow-Compl*: $Pow\ (-\ A) = \{-\ B \mid B.\ A \in Pow\ B\}$
by (*blast intro*: *exI* [**where** *?x* $= -\ u$ **for** *u*])

lemma *Pow-UNIV* [*simp*]: $Pow\ UNIV = UNIV$
by *blast*

lemma *Un-Pow-subset*: $Pow\ A \cup Pow\ B \subseteq Pow\ (A \cup B)$
by *blast*

lemma *Pow-Int-eq* [*simp*]: $Pow\ (A \cap B) = Pow\ A \cap Pow\ B$
by *blast*

Miscellany.

lemma *Int-Diff-disjoint*: $A \cap B \cap (A - B) = \{\}$
by *blast*

lemma *Int-Diff-Un*: $A \cap B \cup (A - B) = A$
by *blast*

lemma *set-eq-subset*: $A = B \longleftrightarrow A \subseteq B \wedge B \subseteq A$
by *blast*

lemma *subset-iff*: $A \subseteq B \longleftrightarrow (\forall t.\ t \in A \longrightarrow t \in B)$
by *blast*

lemma *subset-iff-psubset-eq*: $A \subseteq B \longleftrightarrow A \subset B \vee A = B$
unfolding *less-le* **by** *blast*

lemma *all-not-in-conv* [*simp*]: $(\forall x.\ x \notin A) \longleftrightarrow A = \{\}$
by *blast*

lemma *ex-in-conv*: $(\exists x.\ x \in A) \longleftrightarrow A \neq \{\}$
by *blast*

lemma *ball-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P\ x \vee Q) \longleftrightarrow ((\forall x{\in}A.\ P\ x) \vee Q)$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P \vee Q\ x) \longleftrightarrow (P \vee (\forall x{\in}A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P \longrightarrow Q\ x) \longleftrightarrow (P \longrightarrow (\forall x{\in}A.\ Q\ x))$
$\bigwedge A\ P\ Q.\ (\forall x{\in}A.\ P\ x \longrightarrow Q) \longleftrightarrow ((\exists x{\in}A.\ P\ x) \longrightarrow Q)$
$\bigwedge P.\ (\forall x{\in}\{\}.\ P\ x) \longleftrightarrow True$
$\bigwedge P.\ (\forall x{\in}UNIV.\ P\ x) \longleftrightarrow (\forall x.\ P\ x)$
$\bigwedge a\ B\ P.\ (\forall x{\in}insert\ a\ B.\ P\ x) \longleftrightarrow (P\ a \wedge (\forall x{\in}B.\ P\ x))$
$\bigwedge P\ Q.\ (\forall x{\in}Collect\ Q.\ P\ x) \longleftrightarrow (\forall x.\ Q\ x \longrightarrow P\ x)$
$\bigwedge A\ P\ f.\ (\forall x{\in}f`A.\ P\ x) \longleftrightarrow (\forall x{\in}A.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ (\forall x{\in}A.\ P\ x)) \longleftrightarrow (\exists x{\in}A.\ \neg\ P\ x)$
by *auto*

lemma *bex-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ (\exists x{\in}A.\ P\ x \wedge Q) \longleftrightarrow ((\exists x{\in}A.\ P\ x) \wedge Q)$
$\bigwedge A\ P\ Q.\ (\exists x{\in}A.\ P \wedge Q\ x) \longleftrightarrow (P \wedge (\exists x{\in}A.\ Q\ x))$
$\bigwedge P.\ (\exists x{\in}\{\}.\ P\ x) \longleftrightarrow False$
$\bigwedge P.\ (\exists x{\in}UNIV.\ P\ x) \longleftrightarrow (\exists x.\ P\ x)$
$\bigwedge a\ B\ P.\ (\exists x{\in}insert\ a\ B.\ P\ x) \longleftrightarrow (P\ a \vee (\exists x{\in}B.\ P\ x))$
$\bigwedge P\ Q.\ (\exists x{\in}Collect\ Q.\ P\ x) \longleftrightarrow (\exists x.\ Q\ x \wedge P\ x)$
$\bigwedge A\ P\ f.\ (\exists x{\in}f`A.\ P\ x) \longleftrightarrow (\exists x{\in}A.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg(\exists x{\in}A.\ P\ x)) \longleftrightarrow (\forall x{\in}A.\ \neg\ P\ x)$
by *auto*

lemma *ex-image-cong-iff* [*simp*, *no-atp*]:
$(\exists x.\ x{\in}f`A) \longleftrightarrow A \neq \{\}$ $(\exists x.\ x{\in}f`A \wedge P\ x) \longleftrightarrow (\exists x{\in}A.\ P\ (f\ x))$
by *auto*

8.4.4 Monotonicity of various operations

lemma *image-mono*: $A \subseteq B \Longrightarrow f\ `\ A \subseteq f\ `\ B$
by *blast*

lemma *Pow-mono*: $A \subseteq B \Longrightarrow Pow\ A \subseteq Pow\ B$
by *blast*

lemma *insert-mono*: $C \subseteq D \Longrightarrow insert\ a\ C \subseteq insert\ a\ D$
by *blast*

lemma *Un-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cup B \subseteq C \cup D$
by (*fact sup-mono*)

lemma *Int-mono*: $A \subseteq C \Longrightarrow B \subseteq D \Longrightarrow A \cap B \subseteq C \cap D$
by (*fact inf-mono*)

lemma *Diff-mono*: $A \subseteq C \Longrightarrow D \subseteq B \Longrightarrow A - B \subseteq C - D$
by *blast*

lemma *Compl-anti-mono*: $A \subseteq B \Longrightarrow - B \subseteq - A$
by (*fact compl-mono*)

Monotonicity of implications.

lemma *in-mono*: $A \subseteq B \Longrightarrow x \in A \longrightarrow x \in B$
by (*rule impI*) (*erule subsetD*)

lemma *conj-mono*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \wedge P2) \longrightarrow (Q1 \wedge Q2)$
by *iprover*

lemma *disj-mono*: $P1 \longrightarrow Q1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \vee P2) \longrightarrow (Q1 \vee Q2)$
by *iprover*

lemma *imp-mono*: $Q1 \longrightarrow P1 \Longrightarrow P2 \longrightarrow Q2 \Longrightarrow (P1 \longrightarrow P2) \longrightarrow (Q1 \longrightarrow Q2)$
by *iprover*

lemma *imp-refl*: $P \longrightarrow P$ **..**

lemma *not-mono*: $Q \longrightarrow P \Longrightarrow \neg P \longrightarrow \neg Q$
by *iprover*

lemma *ex-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow (\exists x.\ P\ x) \longrightarrow (\exists x.\ Q\ x)$
by *iprover*

lemma *all-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow (\forall x.\ P\ x) \longrightarrow (\forall x.\ Q\ x)$
by *iprover*

lemma *Collect-mono*: $(\bigwedge x.\ P\ x \longrightarrow Q\ x) \Longrightarrow Collect\ P \subseteq Collect\ Q$
by *blast*

lemma *Int-Collect-mono*: $A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow P\ x \longrightarrow Q\ x) \Longrightarrow A \cap Collect\ P \subseteq B \cap Collect\ Q$
by *blast*

lemmas *basic-monos* =
subset-refl imp-refl disj-mono conj-mono ex-mono Collect-mono in-mono

lemma *eq-to-mono*: $a = b \Longrightarrow c = d \Longrightarrow b \longrightarrow d \Longrightarrow a \longrightarrow c$
by *iprover*

8.4.5 Inverse image of a function

definition *vimage* :: $('a \Rightarrow 'b) \Rightarrow 'b\ set \Rightarrow 'a\ set$ (**infixr** $\leftarrow$ *90*)
where $f -` B \equiv \{x.\ f\ x \in B\}$

lemma *vimage-eq* [*simp*]: $a \in f -` B \longleftrightarrow f\ a \in B$
unfolding *vimage-def* **by** *blast*

lemma *vimage-singleton-eq*: $a \in f -` \{b\} \longleftrightarrow f\ a = b$
by *simp*

lemma *vimageI* [*intro*]: $f\ a = b \Longrightarrow b \in B \Longrightarrow a \in f -` B$
unfolding *vimage-def* **by** *blast*

lemma *vimageI2*: $f\ a \in A \Longrightarrow a \in f -` A$
unfolding *vimage-def* **by** *fast*

lemma *vimageE* [*elim!*]: $a \in f -` B \Longrightarrow (\bigwedge x.\ f\ a = x \Longrightarrow x \in B \Longrightarrow P) \Longrightarrow P$
unfolding *vimage-def* **by** *blast*

lemma *vimageD*: $a \in f -` A \Longrightarrow f\ a \in A$
unfolding *vimage-def* **by** *fast*

lemma *vimage-empty* [*simp*]: $f -` \{\} = \{\}$
by *blast*

lemma *vimage-Compl*: $f -` (-\ A) = -\ (f -` A)$
by *blast*

lemma *vimage-Un* [*simp*]: $f -` (A \cup B) = (f -` A) \cup (f -` B)$
by *blast*

lemma *vimage-Int* [*simp*]: $f -` (A \cap B) = (f -` A) \cap (f -` B)$
by *fast*

lemma *vimage-Collect-eq* [*simp*]: $f -` Collect\ P = \{y.\ P\ (f\ y)\}$
by *blast*

lemma *vimage-Collect*: $(\bigwedge x.\ P\ (f\ x) = Q\ x) \Longrightarrow f -` (Collect\ P) = Collect\ Q$
by *blast*

lemma *vimage-insert*: $f -` (insert\ a\ B) = (f -` \{a\}) \cup (f -` B)$
— NOT suitable for rewriting because of the recurrence of $\{a\}$.
by *blast*

lemma *vimage-Diff*: $f -` (A - B) = (f -` A) - (f -` B)$
by *blast*

lemma *vimage-UNIV* [*simp*]: $f -` UNIV = UNIV$
by *blast*

lemma *vimage-mono*: $A \subseteq B \Longrightarrow f -` A \subseteq f -` B$
— monotonicity
by *blast*

lemma *vimage-image-eq*: $f -` (f ` A) = \{y.\ \exists x{\in}A.\ f\ x = f\ y\}$
by (*blast intro*: *sym*)

lemma *image-vimage-subset*: $f \mathbin{`} (f -` A) \subseteq A$
by *blast*

lemma *image-vimage-eq* [*simp*]: $f \mathbin{`} (f -` A) = A \cap \mathit{range}\ f$
by *blast*

lemma *image-subset-iff-subset-vimage*: $f \mathbin{`} A \subseteq B \longleftrightarrow A \subseteq f -` B$
by *blast*

lemma *subset-vimage-iff*: $A \subseteq f -` B \longleftrightarrow (\forall x{\in}A.\ f\ x \in B)$
by *auto*

lemma *vimage-const* [*simp*]: $((\lambda x.\ c) -` A) = (\mathit{if}\ c \in A\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
by *auto*

lemma *vimage-if* [*simp*]: $((\lambda x.\ \mathit{if}\ x \in B\ \mathit{then}\ c\ \mathit{else}\ d) -` A) =$
$(\mathit{if}\ c \in A\ \mathit{then}\ (\mathit{if}\ d \in A\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ B)$
$\mathit{else}\ \mathit{if}\ d \in A\ \mathit{then} - B\ \mathit{else}\ \{\})$
by (*auto simp add*: *vimage-def*)

lemma *vimage-inter-cong*: $(\bigwedge w.\ w \in S \Longrightarrow f\ w = g\ w) \Longrightarrow f -` y \cap S = g -` y \cap S$
by *auto*

lemma *vimage-ident* [*simp*]: $(\lambda x.\ x) -` Y = Y$
by *blast*

8.4.6 Singleton sets

definition *is-singleton* :: *'a set* ⇒ *bool*
where $\mathit{is\text{-}singleton}\ A \longleftrightarrow (\exists x.\ A = \{x\})$

lemma *is-singletonI* [*simp*, *intro*!]: *is-singleton* $\{x\}$
unfolding *is-singleton-def* **by** *simp*

lemma *is-singletonI'*: $A \neq \{\} \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x = y) \Longrightarrow \mathit{is\text{-}singleton}\ A$
unfolding *is-singleton-def* **by** *blast*

lemma *is-singletonE*: *is-singleton* $A \Longrightarrow (\bigwedge x.\ A = \{x\} \Longrightarrow P) \Longrightarrow P$
unfolding *is-singleton-def* **by** *blast*

lemma *is-singleton-iff-ex1*:
‹*is-singleton* $A \longleftrightarrow (\exists!x.\ x \in A)$›
by (*auto simp add*: *is-singleton-def*)

8.4.7 Getting the contents of a singleton set

definition *the-elem* :: *'a set* ⇒ *'a*

where *the-elem X* = (*THE x*. *X* = {*x*})

lemma *the-elem-eq* [*simp*]: *the-elem* {*x*} = *x*
by (*simp add*: *the-elem-def*)

lemma *is-singleton-the-elem*: *is-singleton A* ⟷ *A* = {*the-elem A*}
by (*auto simp*: *is-singleton-def*)

lemma *the-elem-image-unique*:
assumes *A* ≠ {}
and ∗: ⋀*y*. *y* ∈ *A* ⟹ *f y* = *a*
shows *the-elem* (*f* ‘ *A*) = *a*
unfolding *the-elem-def*
proof (*rule the1-equality*)
from ‹*A* ≠ {}› **obtain** *y* **where** *y* ∈ *A* **by** *auto*
with ∗ ‹*y* ∈ *A*› **have** *a* ∈ *f* ‘ *A* **by** *blast*
with ∗ **show** *f* ‘ *A* = {*a*} **by** *auto*
then show ∃!*x*. *f* ‘ *A* = {*x*} **by** *auto*
qed

8.4.8 Monad operation

definition *bind* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ *'b set*
where *bind A f* = {*x*. ∃ *B* ∈ *f*‘*A*. *x* ∈ *B*}

hide-const (**open**) *bind*

lemma *bind-bind*: *Set.bind* (*Set.bind A B*) *C* = *Set.bind A* (λ*x*. *Set.bind* (*B x*) *C*)
for *A* :: *'a set*
by (*auto simp*: *bind-def*)

lemma *empty-bind* [*simp*]: *Set.bind* {} *f* = {}
by (*simp add*: *bind-def*)

lemma *nonempty-bind-const*: *A* ≠ {} ⟹ *Set.bind A* (λ-. *B*) = *B*
by (*auto simp*: *bind-def*)

lemma *bind-const*: *Set.bind A* (λ-. *B*) = (*if A* = {} *then* {} *else B*)
by (*auto simp*: *bind-def*)

lemma *bind-singleton-conv-image*: *Set.bind A* (λ*x*. {*f x*}) = *f* ‘ *A*
by (*auto simp*: *bind-def*)

8.4.9 Operations for execution

Use those operations only for generating executable / efficient code. Otherwise use the RHSs directly.

context

begin

qualified definition *is-empty* :: *'a set* ⇒ *bool* — only for code generation
where *is-empty-iff* [*code-abbrev*, *simp*]: *is-empty A* ⟷ *A* = {}

qualified definition *remove* :: *'a* ⇒ *'a set* ⇒ *'a set* — only for code generation
where *remove-eq* [*code-abbrev*, *simp*]: *remove x A* = *A* − {*x*}

qualified definition *filter* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *'a set* — only for code generation
where *filter-eq* [*code-abbrev*, *simp*]: *filter P A* = {*a* ∈ *A*. *P a*}

qualified definition *can-select* :: (*'a* ⇒ *bool*) ⇒ *'a set* ⇒ *bool* — only for code generation
where *can-select-iff* [*code-abbrev*, *simp*]: *can-select P A* = (∃!*x*∈*A*. *P x*)

qualified lemma *can-select-iff-is-singleton*:
‹*Set.can-select P A* ⟷ *is-singleton* (*Set.filter P A*)›
by (*simp add*: *is-singleton-iff-ex1*)

end

instantiation *set* :: (*equal*) *equal*
begin

definition *HOL.equal A B* ⟷ *A* ⊆ *B* ∧ *B* ⊆ *A*

instance
by *standard* (*auto simp add*: *equal-set-def*)

end

Misc

definition *pairwise* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a set* ⇒ *bool*
where *pairwise R S* ⟷ (∀ *x* ∈ *S*. ∀ *y* ∈ *S*. *x* ≠ *y* ⟶ *R x y*)

lemma *pairwise-alt*: *pairwise R S* ⟷ (∀ *x*∈*S*. ∀ *y*∈*S*−{*x*}. *R x y*)
by (*auto simp add*: *pairwise-def*)

lemma *pairwise-trivial* [*simp*]: *pairwise* (λ*i j*. *j* ≠ *i*) *I*
by (*auto simp*: *pairwise-def*)

lemma *pairwiseI* [*intro?*]:
pairwise R S **if** ⋀*x y*. *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *x* ≠ *y* ⟹ *R x y*
using *that* **by** (*simp add*: *pairwise-def*)

lemma *pairwiseD*:
R x y **and** *R y x*
if *pairwise R S x* ∈ *S* **and** *y* ∈ *S* **and** *x* ≠ *y*

using *that* **by** (*simp-all add*: *pairwise-def*)

lemma *pairwise-empty* [*simp*]: *pairwise P* {}
by (*simp add*: *pairwise-def*)

lemma *pairwise-singleton* [*simp*]: *pairwise P* {*A*}
by (*simp add*: *pairwise-def*)

lemma *pairwise-insert*:
pairwise r (*insert x s*) ⟷ (∀ *y*. *y* ∈ *s* ∧ *y* ≠ *x* ⟶ *r x y* ∧ *r y x*) ∧ *pairwise r s*
by (*force simp*: *pairwise-def*)

lemma *pairwise-subset*: *pairwise P S* ⟹ *T* ⊆ *S* ⟹ *pairwise P T*
by (*force simp*: *pairwise-def*)

lemma *pairwise-mono*: ⟦*pairwise P A*; ⋀*x y*. *P x y* ⟹ *Q x y*; *B* ⊆ *A*⟧ ⟹ *pairwise Q B*
by (*fastforce simp*: *pairwise-def*)

lemma *pairwise-imageI*:
pairwise P (*f* ‘ *A*)
if ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *x* ≠ *y* ⟹ *f x* ≠ *f y* ⟹ *P* (*f x*) (*f y*)
using *that* **by** (*auto intro*: *pairwiseI*)

lemma *pairwise-image*: *pairwise r* (*f* ‘ *s*) ⟷ *pairwise* (λ*x y*. (*f x* ≠ *f y*) ⟶ *r* (*f x*) (*f y*)) *s*
by (*force simp*: *pairwise-def*)

definition *disjnt* :: *'a set* ⇒ *'a set* ⇒ *bool*
where *disjnt A B* ⟷ *A* ∩ *B* = {}

lemma *disjnt-self-iff-empty* [*simp*]: *disjnt S S* ⟷ *S* = {}
by (*auto simp*: *disjnt-def*)

lemma *disjnt-commute*: *disjnt A B* = *disjnt B A*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-iff*: *disjnt A B* ⟷ (∀ *x*. ¬ (*x* ∈ *A* ∧ *x* ∈ *B*))
by (*force simp*: *disjnt-def*)

lemma *disjnt-sym*: *disjnt A B* ⟹ *disjnt B A*
using *disjnt-iff* **by** *blast*

lemma *disjnt-empty1* [*simp*]: *disjnt* {} *A* **and** *disjnt-empty2* [*simp*]: *disjnt A* {}
by (*auto simp*: *disjnt-def*)

lemma *disjnt-insert1* [*simp*]: *disjnt* (*insert a X*) *Y* ⟷ *a* ∉ *Y* ∧ *disjnt X Y*
by (*simp add*: *disjnt-def*)

lemma *disjnt-insert2* [*simp*]: *disjnt Y* (*insert a X*) ⟷ *a* ∉ *Y* ∧ *disjnt Y X*
by (*simp add*: *disjnt-def*)

lemma *disjnt-subset1* : ⟦*disjnt X Y*; *Z* ⊆ *X*⟧ ⟹ *disjnt Z Y*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-subset2* : ⟦*disjnt X Y*; *Z* ⊆ *Y*⟧ ⟹ *disjnt X Z*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Un1* [*simp*]: *disjnt* (*A* ∪ *B*) *C* ⟷ *disjnt A C* ∧ *disjnt B C*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Un2* [*simp*]: *disjnt C* (*A* ∪ *B*) ⟷ *disjnt C A* ∧ *disjnt C B*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Diff1*: *disjnt* (*X*−*Y*) (*U*−*V*) **and** *disjnt-Diff2*: *disjnt* (*U*−*V*) (*X*−*Y*)
if *X* ⊆ *V*
using *that* **by** (*auto simp*: *disjnt-def*)

lemma *disjoint-image-subset*: ⟦*pairwise disjnt* 𝒜; ⋀*X*. *X* ∈ 𝒜 ⟹ *f X* ⊆ *X*⟧ ⟹
pairwise disjnt (*f* ‘𝒜)
unfolding *disjnt-def pairwise-def* **by** *fast*

lemma *pairwise-disjnt-iff*: *pairwise disjnt* 𝒜 ⟷ (∀ *x*. ∃≤1 *X*. *X* ∈ 𝒜 ∧ *x* ∈ *X*)
by (*auto simp*: *Uniq-def disjnt-iff pairwise-def*)

lemma *disjnt-insert*:
‹*disjnt* (*insert x M*) *N*› **if** ‹*x* ∉ *N*› ‹*disjnt M N*›
using *that* **by** (*simp add*: *disjnt-def*)

lemma *Int-emptyI*: (⋀*x*. *x* ∈ *A* ⟹ *x* ∈ *B* ⟹ *False*) ⟹ *A* ∩ *B* = {}
by *blast*

lemma *in-image-insert-iff*:
assumes ⋀*C*. *C* ∈ *B* ⟹ *x* ∉ *C*
shows *A* ∈ *insert x* ‘ *B* ⟷ *x* ∈ *A* ∧ *A* − {*x*} ∈ *B* (**is** *?P* ⟷ *?Q*)
proof
assume *?P* **then show** *?Q*
using *assms* **by** *auto*
next
assume *?Q*
then have *x* ∈ *A* **and** *A* − {*x*} ∈ *B*
by *simp-all*
from ‹*A* − {*x*} ∈ *B*› **have** *insert x* (*A* − {*x*}) ∈ *insert x* ‘ *B*
by (*rule imageI*)
also from ‹*x* ∈ *A*›
have *insert x* (*A* − {*x*}) = *A*
by *auto*
finally show *?P* **.**

qed

hide-const (**open**) *member not-member*

lemmas *equalityI* = *subset-antisym*
lemmas *set-mp* = *subsetD*
lemmas *set-rev-mp* = *rev-subsetD*

end

9 HOL type definitions

theory *Typedef*
imports *Set*
keywords
typedef :: *thy-goal-defn* **and**
morphisms :: *quasi-command*
begin

locale *type-definition* =
fixes *Rep* **and** *Abs* **and** *A*
assumes *Rep*: *Rep x* ∈ *A*
and *Rep-inverse*: *Abs* (*Rep x*) = *x*
and *Abs-inverse*: *y* ∈ *A* ⟹ *Rep* (*Abs y*) = *y*
— This will be axiomatized for each typedef!
begin

lemma *Rep-inject*: *Rep x* = *Rep y* ⟷ *x* = *y*
proof
assume *Rep x* = *Rep y*
then have *Abs* (*Rep x*) = *Abs* (*Rep y*) **by** (*simp only*:)
also have *Abs* (*Rep x*) = *x* **by** (*rule Rep-inverse*)
also have *Abs* (*Rep y*) = *y* **by** (*rule Rep-inverse*)
finally show *x* = *y* **.**
next
show *x* = *y* ⟹ *Rep x* = *Rep y* **by** (*simp only*:)
qed

lemma *Abs-inject*:
assumes *x* ∈ *A* **and** *y* ∈ *A*
shows *Abs x* = *Abs y* ⟷ *x* = *y*
proof
assume *Abs x* = *Abs y*
then have *Rep* (*Abs x*) = *Rep* (*Abs y*) **by** (*simp only*:)
also from ‹*x* ∈ *A*› **have** *Rep* (*Abs x*) = *x* **by** (*rule Abs-inverse*)
also from ‹*y* ∈ *A*› **have** *Rep* (*Abs y*) = *y* **by** (*rule Abs-inverse*)
finally show *x* = *y* **.**
next
show *x* = *y* ⟹ *Abs x* = *Abs y* **by** (*simp only*:)

qed

lemma *Rep-cases* [*cases set*]:
  assumes $y \in A$
    and *hyp*: $\bigwedge x.\ y = Rep\ x \Longrightarrow P$
  shows P
proof (*rule hyp*)
  from ‹$y \in A$› **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
  then show $y = Rep\ (Abs\ y)$ **..**
qed

lemma *Abs-cases* [*cases type*]:
  assumes *r*: $\bigwedge y.\ x = Abs\ y \Longrightarrow y \in A \Longrightarrow P$
  shows P
proof (*rule r*)
  have $Abs\ (Rep\ x) = x$ **by** (*rule Rep-inverse*)
  then show $x = Abs\ (Rep\ x)$ **..**
  show $Rep\ x \in A$ **by** (*rule Rep*)
qed

lemma *Rep-induct* [*induct set*]:
  assumes *y*: $y \in A$
    and *hyp*: $\bigwedge x.\ P\ (Rep\ x)$
  shows $P\ y$
proof −
  have $P\ (Rep\ (Abs\ y))$ **by** (*rule hyp*)
  also from y **have** $Rep\ (Abs\ y) = y$ **by** (*rule Abs-inverse*)
  finally show $P\ y$ **.**
qed

lemma *Abs-induct* [*induct type*]:
  assumes *r*: $\bigwedge y.\ y \in A \Longrightarrow P\ (Abs\ y)$
  shows $P\ x$
proof −
  have $Rep\ x \in A$ **by** (*rule Rep*)
  then have $P\ (Abs\ (Rep\ x))$ **by** (*rule r*)
  also have $Abs\ (Rep\ x) = x$ **by** (*rule Rep-inverse*)
  finally show $P\ x$ **.**
qed

lemma *Rep-range*: $range\ Rep = A$
proof
  show $range\ Rep \subseteq A$ **using** *Rep* **by** (*auto simp add*: *image-def*)
  show $A \subseteq range\ Rep$
  proof
    fix x **assume** $x \in A$
    then have $x = Rep\ (Abs\ x)$ **by** (*rule Abs-inverse* [*symmetric*])
    then show $x \in range\ Rep$ **by** (*rule range-eqI*)
  qed

qed

lemma *Abs-image*: *Abs* ‘ *A* = *UNIV*
proof
  show *Abs* ‘ *A* ⊆ *UNIV* **by** (*rule subset-UNIV*)
  show *UNIV* ⊆ *Abs* ‘ *A*
  proof
    show *x* ∈ *Abs* ‘ *A* **for** *x*
    proof (*rule image-eqI*)
      show *x* = *Abs* (*Rep x*) **by** (*rule Rep-inverse* [*symmetric*])
      show *Rep x* ∈ *A* **by** (*rule Rep*)
    qed
  qed
qed

end

ML-file ‹*Tools/typedef.ML*›

end

10 Notions about functions

theory *Fun*
  imports *Set*
  keywords *functor* :: *thy-goal-defn*
begin

lemma *apply-inverse*: *f x* = *u* ⟹ (⋀*x*. *P x* ⟹ *g* (*f x*) = *x*) ⟹ *P x* ⟹ *x* = *g u*
  by *auto*

Uniqueness, so NOT the axiom of choice.

lemma *uniq-choice*: ∀ *x*. ∃!*y*. *Q x y* ⟹ ∃*f*. ∀ *x*. *Q x* (*f x*)
  by (*force intro*: *theI'*)

lemma *b-uniq-choice*: ∀ *x*∈*S*. ∃!*y*. *Q x y* ⟹ ∃*f*. ∀ *x*∈*S*. *Q x* (*f x*)
  by (*force intro*: *theI'*)

10.1 The Identity Function *id*

definition *id* :: *'a* ⇒ *'a*
  where *id* = (λ*x*. *x*)

lemma *id-apply* [*simp*]: *id x* = *x*
  by (*simp add*: *id-def*)

lemma *image-id* [*simp*]: *image id* = *id*
  by (*simp add*: *id-def fun-eq-iff*)

lemma *vimage-id* [*simp*]: *vimage id = id*
by (*simp add*: *id-def fun-eq-iff*)

lemma *eq-id-iff*: $(\forall x.\ f\ x = x) \longleftrightarrow f = id$
by *auto*

code-printing
constant *id* ⇀ (*Haskell*) *id*

10.2 The Composition Operator $f \circ g$

definition *comp* :: $('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ (**infixl** ‹∘› *55*)
where $f \circ g = (\lambda x.\ f\ (g\ x))$

notation (*ASCII*)
comp (**infixl** ‹o› *55*)

lemma *comp-apply* [*simp*]: $(f \circ g)\ x = f\ (g\ x)$
by (*simp add*: *comp-def*)

lemma *comp-assoc*: $(f \circ g) \circ h = f \circ (g \circ h)$
by (*simp add*: *fun-eq-iff*)

lemma *id-comp* [*simp*]: $id \circ g = g$
by (*simp add*: *fun-eq-iff*)

lemma *comp-id* [*simp*]: $f \circ id = f$
by (*simp add*: *fun-eq-iff*)

lemma *comp-eq-dest*: $a \circ b = c \circ d \Longrightarrow a\ (b\ v) = c\ (d\ v)$
by (*simp add*: *fun-eq-iff*)

lemma *comp-eq-elim*: $a \circ b = c \circ d \Longrightarrow ((\bigwedge v.\ a\ (b\ v) = c\ (d\ v)) \Longrightarrow R) \Longrightarrow R$
by (*simp add*: *fun-eq-iff*)

lemma *comp-eq-dest-lhs*: $a \circ b = c \Longrightarrow a\ (b\ v) = c\ v$
by *clarsimp*

lemma *comp-eq-id-dest*: $a \circ b = id \circ c \Longrightarrow a\ (b\ v) = c\ v$
by *clarsimp*

lemma *image-comp*: $f\ `\ (g\ `\ r) = (f \circ g)\ `\ r$
by *auto*

lemma *vimage-comp*: $f\ -`\ (g\ -`\ x) = (g \circ f)\ -`\ x$
by *auto*

lemma *image-eq-imp-comp*: $f\ `\ A = g\ `\ B \Longrightarrow (h \circ f)\ `\ A = (h \circ g)\ `\ B$

by (*auto simp*: *comp-def elim*!: *equalityE*)

lemma *image-bind*: f ‘ (*Set.bind* A g) = *Set.bind* A ((‘) $f \circ g$)
by (*auto simp add*: *Set.bind-def*)

lemma *bind-image*: *Set.bind* (f ‘ A) g = *Set.bind* A ($g \circ f$)
by (*auto simp add*: *Set.bind-def*)

lemma (**in** *group-add*) *minus-comp-minus* [*simp*]: *uminus* ∘ *uminus* = *id*
by (*simp add*: *fun-eq-iff*)

lemma (**in** *boolean-algebra*) *minus-comp-minus* [*simp*]: *uminus* ∘ *uminus* = *id*
by (*simp add*: *fun-eq-iff*)

code-printing
constant *comp* ⇀ (*SML*) **infixl** *5 o* **and** (*Haskell*) **infixr** *9* .

10.3 The Forward Composition Operator *fcomp*

definition *fcomp* :: ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*c*) ⇒ '*a* ⇒ '*c* (**infixl** ‹∘>› *60*)
where f ∘> g = (λx. g (f x))

lemma *fcomp-apply* [*simp*]: (f ∘> g) x = g (f x)
by (*simp add*: *fcomp-def*)

lemma *fcomp-assoc*: (f ∘> g) ∘> h = f ∘> (g ∘> h)
by (*simp add*: *fcomp-def*)

lemma *id-fcomp* [*simp*]: *id* ∘> g = g
by (*simp add*: *fcomp-def*)

lemma *fcomp-id* [*simp*]: f ∘> *id* = f
by (*simp add*: *fcomp-def*)

lemma *fcomp-comp*: *fcomp* f g = *comp* g f
by (*simp add*: *ext*)

code-printing
constant *fcomp* ⇀ (*Eval*) **infixl** *1* #>

no-notation *fcomp* (**infixl** ‹∘>› *60*)

10.4 Mapping functions

definition *map-fun* :: ('*c* ⇒ '*a*) ⇒ ('*b* ⇒ '*d*) ⇒ ('*a* ⇒ '*b*) ⇒ '*c* ⇒ '*d*
where *map-fun* f g h = g ∘ h ∘ f

lemma *map-fun-apply* [*simp*]: *map-fun* f g h x = g (h (f x))
by (*simp add*: *map-fun-def*)

10.5 Injectivity and Bijectivity

definition *inj-on* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow bool$ — injective
where $inj\text{-}on\ f\ A \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}A.\ f\ x = f\ y \longrightarrow x = y)$

definition *bij-betw* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$ — bijective
where $bij\text{-}betw\ f\ A\ B \longleftrightarrow inj\text{-}on\ f\ A \wedge f\ `\ A = B$

A common special case: functions injective, surjective or bijective over the entire domain type.

abbreviation *inj* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $inj\ f \equiv inj\text{-}on\ f\ UNIV$

abbreviation *surj* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $surj\ f \equiv range\ f = UNIV$

translations — The negated case:
$\neg\ CONST\ surj\ f \leftharpoonup CONST\ range\ f \neq CONST\ UNIV$

abbreviation *bij* :: $('a \Rightarrow 'b) \Rightarrow bool$
where $bij\ f \equiv bij\text{-}betw\ f\ UNIV\ UNIV$

lemma *inj-def*: $inj\ f \longleftrightarrow (\forall x\ y.\ f\ x = f\ y \longrightarrow x = y)$
unfolding *inj-on-def* **by** *blast*

lemma *injI*: $(\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y) \Longrightarrow inj\ f$
unfolding *inj-def* **by** *blast*

theorem *range-ex1-eq*: $inj\ f \Longrightarrow b \in range\ f \longleftrightarrow (\exists!x.\ b = f\ x)$
unfolding *inj-def* **by** *blast*

lemma *injD*: $inj\ f \Longrightarrow f\ x = f\ y \Longrightarrow x = y$
by (*simp add*: *inj-def*)

lemma *inj-on-eq-iff*: $inj\text{-}on\ f\ A \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
by (*auto simp*: *inj-on-def*)

lemma *inj-on-cong*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a = g\ a) \Longrightarrow inj\text{-}on\ f\ A \longleftrightarrow inj\text{-}on\ g\ A$
by (*auto simp*: *inj-on-def*)

lemma *image-strict-mono*: $inj\text{-}on\ f\ B \Longrightarrow A \subset B \Longrightarrow f\ `\ A \subset f\ `\ B$
unfolding *inj-on-def* **by** *blast*

lemma *inj-compose*: $inj\ f \Longrightarrow inj\ g \Longrightarrow inj\ (f \circ g)$
by (*simp add*: *inj-def*)

lemma *inj-fun*: $inj\ f \Longrightarrow inj\ (\lambda x\ y.\ f\ x)$
by (*simp add*: *inj-def fun-eq-iff*)

lemma *inj-eq*: $inj\ f \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$

by (*simp add*: *inj-on-eq-iff*)

lemma *inj-on-iff-Uniq*: *inj-on f A* $\longleftrightarrow$ ($\forall x{\in}A.\ \exists_{\leq 1} y.\ y{\in}A \wedge f\ x = f\ y$)
by (*auto simp*: *Uniq-def inj-on-def*)

lemma *inj-on-id*[*simp*]: *inj-on id A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-id2*[*simp*]: *inj-on* ($\lambda x.\ x$) *A*
by (*simp add*: *inj-on-def*)

lemma *inj-on-Int*: *inj-on f A* $\vee$ *inj-on f B* $\Longrightarrow$ *inj-on f* (*A* $\cap$ *B*)
unfolding *inj-on-def* **by** *blast*

lemma *surj-id*: *surj id*
by *simp*

lemma *bij-id*[*simp*]: *bij id*
by (*simp add*: *bij-betw-def*)

lemma *bij-uminus*: *bij* (*uminus* :: *'a* $\Rightarrow$ *'a*::*group-add*)
unfolding *bij-betw-def inj-on-def*
by (*force intro*: *minus-minus* [*symmetric*])

lemma *bij-betwE*: *bij-betw f A B* $\Longrightarrow$ $\forall a{\in}A.\ f\ a \in B$
unfolding *bij-betw-def* **by** *auto*

lemma *inj-onI* [*intro?*]: ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow x = y$) $\Longrightarrow$ *inj-on f A*
by (*simp add*: *inj-on-def*)

For those frequent proofs by contradiction

lemma *inj-onCI*: ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow x \neq y \Longrightarrow$ *False*) $\Longrightarrow$ *inj-on f A*
by (*force simp*: *inj-on-def*)

lemma *inj-on-inverseI*: ($\bigwedge x.\ x \in A \Longrightarrow g\ (f\ x) = x$) $\Longrightarrow$ *inj-on f A*
by (*auto dest*: *arg-cong* [*of concl*: *g*] *simp add*: *inj-on-def*)

lemma *inj-onD*: *inj-on f A* $\Longrightarrow$ *f x* = *f y* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *y* $\in$ *A* $\Longrightarrow$ *x* = *y*
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-subset*:
⟦ *inj-on f A*; *B* $\subseteq$ *A* ⟧ $\Longrightarrow$ *inj-on f B*
unfolding *inj-on-def* **by** *blast*

lemma *comp-inj-on*: *inj-on f A* $\Longrightarrow$ *inj-on g* (*f* ‘ *A*) $\Longrightarrow$ *inj-on* (*g* $\circ$ *f*) *A*
by (*simp add*: *comp-def inj-on-def*)

lemma *inj-on-imageI*: *inj-on* $(g \circ f)$ $A \Longrightarrow$ *inj-on* g $(f$ ‘ $A)$
by (*auto simp add*: *inj-on-def*)

lemma *inj-on-image-iff*:
$\forall x{\in}A.\ \forall y{\in}A.\ g\ (f\ x) = g\ (f\ y) \longleftrightarrow g\ x = g\ y \Longrightarrow$ *inj-on* $f\ A \Longrightarrow$ *inj-on* g $(f$ ‘ $A) \longleftrightarrow$ *inj-on* $g\ A$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-contraD*: *inj-on* $f\ A \Longrightarrow x \neq y \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow f\ x \neq f\ y$
unfolding *inj-on-def* **by** *blast*

lemma *inj-singleton* [*simp*]: *inj-on* $(\lambda x.\ \{x\})\ A$
by (*simp add*: *inj-on-def*)

lemma *inj-on-empty*[*iff*]: *inj-on* $f\ \{\}$
by (*simp add*: *inj-on-def*)

lemma *inj-on-Un*: *inj-on* $f\ (A \cup B) \longleftrightarrow$ *inj-on* $f\ A \wedge$ *inj-on* $f\ B \wedge f$ ‘ $(A - B) \cap f$ ‘ $(B - A) = \{\}$
unfolding *inj-on-def* **by** (*blast intro*: *sym*)

lemma *inj-on-insert* [*iff*]: *inj-on* f (*insert* $a\ A) \longleftrightarrow$ *inj-on* $f\ A \wedge f\ a \notin f$ ‘ $(A - \{a\})$
unfolding *inj-on-def* **by** (*blast intro*: *sym*)

lemma *inj-on-diff*: *inj-on* $f\ A \Longrightarrow$ *inj-on* f $(A - B)$
unfolding *inj-on-def* **by** *blast*

lemma *comp-inj-on-iff*: *inj-on* $f\ A \Longrightarrow$ *inj-on* f' $(f$ ‘ $A) \longleftrightarrow$ *inj-on* $(f' \circ f)\ A$
by (*auto simp*: *comp-inj-on inj-on-def*)

lemma *inj-on-imageI2*: *inj-on* $(f' \circ f)\ A \Longrightarrow$ *inj-on* $f\ A$
by (*auto simp*: *comp-inj-on inj-on-def*)

lemma *inj-img-insertE*:
assumes *inj-on* $f\ A$
assumes $x \notin B$
and *insert* $x\ B = f$ ‘ A
obtains $x'\ A'$ **where** $x' \notin A'$ **and** $A =$ *insert* $x'\ A'$ **and** $x = f\ x'$ **and** $B = f$ ‘ A'
proof −
from *assms* **have** $x \in f$ ‘ A **by** *auto*
then obtain x' **where** $*$: $x' \in A\ x = f\ x'$ **by** *auto*
then have A: $A =$ *insert* x' $(A - \{x'\})$ **by** *auto*
with *assms* $*$ **have** B: $B = f$ ‘ $(A - \{x'\})$ **by** (*auto dest*: *inj-on-contraD*)
have $x' \notin A - \{x'\}$ **by** *simp*
from *this* A ‹$x = f\ x'$› B **show** *?thesis* **..**
qed

lemma *linorder-inj-onI*:
fixes A :: $'a$::*order set*
assumes *ne*: $\bigwedge x\ y.\ [\![x < y;\ x{\in}A;\ y{\in}A]\!] \Longrightarrow f\ x \neq f\ y$ **and** *lin*: $\bigwedge x\ y.\ [\![x{\in}A;\ y{\in}A]\!]$ $\Longrightarrow x{\leq}y \vee y{\leq}x$
shows *inj-on f A*
proof (*rule inj-onI*)
fix $x\ y$
assume *eq*: $f\ x = f\ y$ **and** $x{\in}A\ y{\in}A$
then show $x = y$
using *lin* [*of x y*] *ne* **by** (*force simp*: *dual-order.order-iff-strict*)
qed

type class version of the above

lemma *linorder-inj-onI'*:
fixes A :: $'a$ :: *linorder set*
assumes $\bigwedge i\ j.\ i \in A \Longrightarrow j \in A \Longrightarrow i < j \Longrightarrow f\ i \neq f\ j$
shows *inj-on f A*
by (*intro linorder-inj-onI*) (*auto simp add*: *assms*)

lemma *linorder-injI*:
assumes $\bigwedge x\ y{::}'a{::}$*linorder*. $x < y \Longrightarrow f\ x \neq f\ y$
shows *inj f*
using *assms* **by** (*simp add*: *linorder-inj-onI'*)

lemma *linorder-inj-on-iff*:
fixes A :: $'a$ :: *linorder set*
shows *inj-on f A* $\longleftrightarrow$ $(\forall\, i\ j.\ i \in A \longrightarrow j \in A \longrightarrow i < j \longrightarrow f\ i \neq f\ j)$ (**is** *?L* = *?R*)
proof
assume *rhs*: *?R*
then show *?L*
by (*blast intro*: *linorder-inj-onI'*)
qed (*auto simp*: *inj-on-def*)

lemma *inj-on-image-Pow*: *inj-on f A* $\Longrightarrow$*inj-on* (*image f*) (*Pow A*)
unfolding *Pow-def inj-on-def* **by** *blast*

lemma *inj-on-vimage-image*: *inj-on* ($\lambda b.\ f\ -`\ \{b\}$) ($f\ `\ A$)
using *inj-on-def* **by** *fastforce*

lemma *bij-betw-image-Pow*: *bij-betw f A B* $\Longrightarrow$ *bij-betw* (*image f*) (*Pow A*) (*Pow B*)
by (*auto simp add*: *bij-betw-def inj-on-image-Pow image-Pow-surj*)

lemma *surj-def*: *surj f* $\longleftrightarrow$ $(\forall\, y.\ \exists\, x.\ y = f\ x)$
by *auto*

lemma *surjI*:
assumes $\bigwedge x.\ g\ (f\ x) = x$

shows *surj g*
using *assms* [*symmetric*] **by** *auto*

lemma *surjD*: *surj f* $\Longrightarrow$ $\exists x.\ y = f\ x$
by (*simp add*: *surj-def*)

lemma *surjE*: *surj f* $\Longrightarrow$ $(\bigwedge x.\ y = f\ x \Longrightarrow C) \Longrightarrow C$
by (*simp add*: *surj-def*) *blast*

lemma *comp-surj*: *surj f* $\Longrightarrow$ *surj g* $\Longrightarrow$ *surj* $(g \circ f)$
using *image-comp* [*of g f UNIV*] **by** *simp*

lemma *bij-betw-imageI*: *inj-on f A* $\Longrightarrow$ $f\ `\ A = B \Longrightarrow$ *bij-betw f A B*
unfolding *bij-betw-def* **by** *clarify*

lemma *bij-betw-imp-surj-on*: *bij-betw f A B* $\Longrightarrow$ $f\ `\ A = B$
unfolding *bij-betw-def* **by** *clarify*

lemma *bij-betw-imp-surj*: *bij-betw f A UNIV* $\Longrightarrow$ *surj f*
unfolding *bij-betw-def* **by** *auto*

lemma *bij-betw-empty1*: *bij-betw f* $\{\}$ *A* $\Longrightarrow$ $A = \{\}$
unfolding *bij-betw-def* **by** *blast*

lemma *bij-betw-empty2*: *bij-betw f A* $\{\}$ $\Longrightarrow$ $A = \{\}$
unfolding *bij-betw-def* **by** *blast*

lemma *inj-on-imp-bij-betw*: *inj-on f A* $\Longrightarrow$ *bij-betw f A* $(f\ `\ A)$
unfolding *bij-betw-def* **by** *simp*

lemma *bij-betw-DiffI*:
assumes *bij-betw f A B bij-betw f C D* $C \subseteq A$ $D \subseteq B$
shows *bij-betw f* $(A - C)$ $(B - D)$
using *assms* **unfolding** *bij-betw-def inj-on-def* **by** *auto*

lemma *bij-betw-singleton-iff* [*simp*]: *bij-betw f* $\{x\}$ $\{y\}$ $\longleftrightarrow$ $f\ x = y$
by (*auto simp*: *bij-betw-def*)

lemma *bij-betw-singletonI* [*intro*]: $f\ x = y \Longrightarrow$ *bij-betw f* $\{x\}$ $\{y\}$
by *auto*

lemma *bij-betw-imp-empty-iff*: *bij-betw f A B* $\Longrightarrow$ $A = \{\} \longleftrightarrow B = \{\}$
unfolding *bij-betw-def* **by** *blast*

lemma *bij-betw-imp-Ex-iff*: *bij-betw f* $\{x.\ P\ x\}$ $\{x.\ Q\ x\} \Longrightarrow (\exists x.\ P\ x) \longleftrightarrow (\exists x.\ Q\ x)$
unfolding *bij-betw-def* **by** *blast*

lemma *bij-betw-imp-Bex-iff*: *bij-betw f* $\{x{\in}A.\ P\ x\}$ $\{x{\in}B.\ Q\ x\} \Longrightarrow (\exists x{\in}A.\ P\ x)$

$\longleftrightarrow$ $(\exists\, x{\in}B.\ Q\ x)$
unfolding *bij-betw-def* **by** *blast*

lemma *bij-betw-apply*: $[\![bij\text{-}betw\ f\ A\ B;\ a \in A]\!] \Longrightarrow f\ a \in B$
unfolding *bij-betw-def* **by** *auto*

lemma *bij-def*: $bij\ f \longleftrightarrow inj\ f \wedge surj\ f$
by (*rule bij-betw-def*)

lemma *bijI*: $inj\ f \Longrightarrow surj\ f \Longrightarrow bij\ f$
by (*rule bij-betw-imageI*)

lemma *bij-is-inj*: $bij\ f \Longrightarrow inj\ f$
by (*simp add*: *bij-def*)

lemma *bij-is-surj*: $bij\ f \Longrightarrow surj\ f$
by (*simp add*: *bij-def*)

lemma *bij-betw-imp-inj-on*: $bij\text{-}betw\ f\ A\ B \Longrightarrow inj\text{-}on\ f\ A$
by (*simp add*: *bij-betw-def*)

lemma *bij-betw-trans*: $bij\text{-}betw\ f\ A\ B \Longrightarrow bij\text{-}betw\ g\ B\ C \Longrightarrow bij\text{-}betw\ (g \circ f)\ A\ C$
by (*auto simp add*:*bij-betw-def comp-inj-on*)

lemma *bij-comp*: $bij\ f \Longrightarrow bij\ g \Longrightarrow bij\ (g \circ f)$
by (*rule bij-betw-trans*)

lemma *bij-betw-comp-iff*: $bij\text{-}betw\ f\ A\ A' \Longrightarrow bij\text{-}betw\ f'\ A'\ A'' \longleftrightarrow bij\text{-}betw\ (f' \circ f)\ A\ A''$
by (*auto simp add*: *bij-betw-def inj-on-def*)

lemma *bij-betw-Collect*:
assumes $bij\text{-}betw\ f\ A\ B\ \bigwedge x.\ x \in A \Longrightarrow Q\ (f\ x) \longleftrightarrow P\ x$
shows $bij\text{-}betw\ f\ \{x{\in}A.\ P\ x\}\ \{y{\in}B.\ Q\ y\}$
using *assms* **by** (*auto simp add*: *bij-betw-def inj-on-def*)

lemma *bij-betw-comp-iff2*:
assumes *bij*: $bij\text{-}betw\ f'\ A'\ A''$
and *img*: $f\ `\ A \leq A'$
shows $bij\text{-}betw\ f\ A\ A' \longleftrightarrow bij\text{-}betw\ (f' \circ f)\ A\ A''$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
then show *?R*
using *assms* **by** (*auto simp add*: *bij-betw-comp-iff*)
next
assume $*$: *?R*
have $inj\text{-}on\ (f' \circ f)\ A \Longrightarrow inj\text{-}on\ f\ A$
using *inj-on-imageI2* **by** *blast*
moreover have $A' \subseteq f\ `\ A$

```
    proof
      fix a'
      assume **: a' ∈ A'
      with bij have f' a' ∈ A''
        unfolding bij-betw-def by auto
      with * obtain a where 1: a ∈ A ∧ f' (f a) = f' a'
        unfolding bij-betw-def by force
      with img have f a ∈ A' by auto
      with bij ** 1 have f a = a'
        unfolding bij-betw-def inj-on-def by auto
      with 1 show a' ∈ f ` A by auto
    qed
    ultimately show ?L
      using img * by (auto simp add: bij-betw-def)
qed

lemma bij-betw-inv:
  assumes bij-betw f A B
  shows ∃ g. bij-betw g B A
proof −
  have i: inj-on f A and s: f ` A = B
    using assms by (auto simp: bij-betw-def)
  let ?P = λb a. a ∈ A ∧ f a = b
  let ?g = λb. The (?P b)
  have g: ?g b = a if P: ?P b a for a b
  proof −
    from that s have ex1: ∃ a. ?P b a by blast
    then have uex1: ∃!a. ?P b a by (blast dest:inj-onD[OF i])
    then show ?thesis
      using the1-equality[OF uex1, OF P] P by simp
  qed
  have inj-on ?g B
  proof (rule inj-onI)
    fix x y
    assume x ∈ B y ∈ B ?g x = ?g y
    from s ‹x ∈ B› obtain a1 where a1: ?P x a1 by blast
    from s ‹y ∈ B› obtain a2 where a2: ?P y a2 by blast
    from g [OF a1] a1 g [OF a2] a2 ‹?g x = ?g y› show x = y by simp
  qed
  moreover have ?g ` B = A
  proof safe
    fix b
    assume b ∈ B
    with s obtain a where P: ?P b a by blast
    with g[OF P] show ?g b ∈ A by auto
  next
    fix a
    assume a ∈ A
    with s obtain b where P: ?P b a by blast
```

```
    with s have b ∈ B by blast
    with g[OF P] have ∃ b∈B. a = ?g b by blast
    then show a ∈ ?g ‘ B
      by auto
  qed
  ultimately show ?thesis
    by (auto simp: bij-betw-def)
qed

lemma bij-betw-cong: (⋀a. a ∈ A ⟹ f a = g a) ⟹ bij-betw f A A′ = bij-betw g
A A′
  unfolding bij-betw-def inj-on-def by safe force+

lemma bij-betw-id[intro, simp]: bij-betw id A A
  unfolding bij-betw-def id-def by auto

lemma bij-betw-id-iff: bij-betw id A B ⟷ A = B
  by (auto simp add: bij-betw-def)

lemma bij-betw-combine:
  bij-betw f A B ⟹ bij-betw f C D ⟹ B ∩ D = {} ⟹ bij-betw f (A ∪ C) (B ∪
D)
  unfolding bij-betw-def inj-on-Un image-Un by auto

lemma bij-betw-subset: bij-betw f A A′ ⟹ B ⊆ A ⟹ f ‘ B = B′ ⟹ bij-betw f
B B′
  by (auto simp add: bij-betw-def inj-on-def)

lemma bij-betw-ball: bij-betw f A B ⟹ (∀ b ∈ B. phi b) = (∀ a ∈ A. phi (f a))
  unfolding bij-betw-def inj-on-def by blast

lemma bij-pointE:
  assumes bij f
  obtains x where y = f x and ⋀x′. y = f x′ ⟹ x′ = x
proof −
  from assms have inj f by (rule bij-is-inj)
  moreover from assms have surj f by (rule bij-is-surj)
  then have y ∈ range f by simp
  ultimately have ∃!x. y = f x by (simp add: range-ex1-eq)
  with that show thesis by blast
qed

lemma bij-iff:
  ‹bij f ⟷ (∀ x. ∃!y. f y = x)›  (is ‹?P ⟷ ?Q›)
proof
  assume ?P
  then have ‹inj f› ‹surj f›
    by (simp-all add: bij-def)
  show ?Q
```

proof
fix *y*
from ‹*surj f*› **obtain** *x* **where** ‹*y* = *f x*›
by (*auto simp add*: *surj-def*)
with ‹*inj f*› **show** ‹∃!*x*. *f x* = *y*›
by (*auto simp add*: *inj-def*)
qed
next
assume *?Q*
then have ‹*inj f*›
by (*auto simp add*: *inj-def*)
moreover have ‹∃ *x*. *y* = *f x*› **for** *y*
proof −
from ‹*?Q*› **obtain** *x* **where** ‹*f x* = *y*›
by *blast*
then have ‹*y* = *f x*›
by *simp*
then show *?thesis* **..**
qed
then have ‹*surj f*›
by (*auto simp add*: *surj-def*)
ultimately show *?P*
by (*rule bijI*)
qed

lemma *bij-betw-partition*:
‹*bij-betw f A B*›
if ‹*bij-betw f* (*A* ∪ *C*) (*B* ∪ *D*)› ‹*bij-betw f C D*› ‹*A* ∩ *C* = {}› ‹*B* ∩ *D* = {}›
proof −
from *that* **have** ‹*inj-on f* (*A* ∪ *C*)› ‹*inj-on f C*› ‹*f* ‘ (*A* ∪ *C*) = *B* ∪ *D*› ‹*f* ‘ *C* = *D*›
by (*simp-all add*: *bij-betw-def*)
then have ‹*inj-on f A*› **and** ‹*f* ‘ (*A* − *C*) ∩ *f* ‘ (*C* − *A*) = {}›
by (*simp-all add*: *inj-on-Un*)
with ‹*A* ∩ *C* = {}› **have** ‹*f* ‘ *A* ∩ *f* ‘ *C* = {}›
by *auto*
with ‹*f* ‘ (*A* ∪ *C*) = *B* ∪ *D*› ‹*f* ‘ *C* = *D*› ‹*B* ∩ *D* = {}›
have ‹*f* ‘ *A* = *B*›
by *blast*
with ‹*inj-on f A*› **show** *?thesis*
by (*simp add*: *bij-betw-def*)
qed

lemma *surj-image-vimage-eq*: *surj f* ⟹ *f* ‘ (*f* −‘ *A*) = *A*
by *simp*

lemma *surj-vimage-empty*:
assumes *surj f*
shows *f* −‘ *A* = {} ⟷ *A* = {}

using *surj-image-vimage-eq* [*OF* ‹*surj f*›, *of A*]
by (*intro iffI*) *fastforce+*

lemma *inj-vimage-image-eq*: *inj f* ⟹ *f* −‘ (*f* ‘ *A*) = *A*
unfolding *inj-def* **by** *blast*

lemma *vimage-subsetD*: *surj f* ⟹ *f* −‘ *B* ⊆ *A* ⟹ *B* ⊆ *f* ‘ *A*
by (*blast intro*: *sym*)

lemma *vimage-subsetI*: *inj f* ⟹ *B* ⊆ *f* ‘ *A* ⟹ *f* −‘ *B* ⊆ *A*
unfolding *inj-def* **by** *blast*

lemma *vimage-subset-eq*: *bij f* ⟹ *f* −‘ *B* ⊆ *A* ⟷ *B* ⊆ *f* ‘ *A*
unfolding *bij-def* **by** (*blast del*: *subsetI intro*: *vimage-subsetI vimage-subsetD*)

lemma *inj-on-image-eq-iff*: *inj-on f C* ⟹ *A* ⊆ *C* ⟹ *B* ⊆ *C* ⟹ *f* ‘ *A* = *f* ‘ *B* ⟷ *A* = *B*
by (*fastforce simp*: *inj-on-def*)

lemma *inj-on-Un-image-eq-iff*: *inj-on f* (*A* ∪ *B*) ⟹ *f* ‘ *A* = *f* ‘ *B* ⟷ *A* = *B*
by (*erule inj-on-image-eq-iff*) *simp-all*

lemma *inj-on-image-Int*: *inj-on f C* ⟹ *A* ⊆ *C* ⟹ *B* ⊆ *C* ⟹ *f* ‘ (*A* ∩ *B*) = *f* ‘ *A* ∩ *f* ‘ *B*
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-image-set-diff*: *inj-on f C* ⟹ *A* − *B* ⊆ *C* ⟹ *B* ⊆ *C* ⟹ *f* ‘ (*A* − *B*) = *f* ‘ *A* − *f* ‘ *B*
unfolding *inj-on-def* **by** *blast*

lemma *image-Int*: *inj f* ⟹ *f* ‘ (*A* ∩ *B*) = *f* ‘ *A* ∩ *f* ‘ *B*
unfolding *inj-def* **by** *blast*

lemma *image-set-diff*: *inj f* ⟹ *f* ‘ (*A* − *B*) = *f* ‘ *A* − *f* ‘ *B*
unfolding *inj-def* **by** *blast*

lemma *inj-on-image-mem-iff*: *inj-on f B* ⟹ *a* ∈ *B* ⟹ *A* ⊆ *B* ⟹ *f a* ∈ *f* ‘ *A* ⟷ *a* ∈ *A*
by (*auto simp*: *inj-on-def*)

lemma *inj-image-mem-iff*: *inj f* ⟹ *f a* ∈ *f* ‘ *A* ⟷ *a* ∈ *A*
by (*blast dest*: *injD*)

lemma *inj-image-subset-iff*: *inj f* ⟹ *f* ‘ *A* ⊆ *f* ‘ *B* ⟷ *A* ⊆ *B*
by (*blast dest*: *injD*)

lemma *inj-image-eq-iff*: *inj f* ⟹ *f* ‘ *A* = *f* ‘ *B* ⟷ *A* = *B*
by (*blast dest*: *injD*)

lemma *surj-Compl-image-subset*: *surj f* $\Longrightarrow$ $-$ (*f* ‘ *A*) $\subseteq$ *f* ‘ ($-$ *A*)
by *auto*

lemma *inj-image-Compl-subset*: *inj f* $\Longrightarrow$ *f* ‘ ($-$ *A*) $\subseteq$ $-$ (*f* ‘ *A*)
by (*auto simp*: *inj-def*)

lemma *bij-image-Compl-eq*: *bij f* $\Longrightarrow$ *f* ‘ ($-$ *A*) = $-$ (*f* ‘ *A*)
by (*simp add*: *bij-def inj-image-Compl-subset surj-Compl-image-subset equalityI*)

lemma *inj-vimage-singleton*: *inj f* $\Longrightarrow$ *f* $-$‘ {*a*} $\subseteq$ {*THE x*. *f x* = *a*}
— The inverse image of a singleton under an injective function is included in a singleton.
by (*simp add*: *inj-def*) (*blast intro*: *the-equality* [*symmetric*])

lemma *inj-on-vimage-singleton*: *inj-on f A* $\Longrightarrow$ *f* $-$‘ {*a*} $\cap$ *A* $\subseteq$ {*THE x*. *x* $\in$ *A* $\wedge$ *f x* = *a*}
by (*auto simp add*: *inj-on-def intro*: *the-equality* [*symmetric*])

lemma *bij-betw-byWitness*:
assumes *left*: $\forall$ *a* $\in$ *A*. *f'* (*f a*) = *a*
and *right*: $\forall$ *a'* $\in$ *A'*. *f* (*f'* *a'*) = *a'*
and *f* ‘ *A* $\subseteq$ *A'*
and *img2*: *f'* ‘ *A'* $\subseteq$ *A*
shows *bij-betw f A A'*
using *assms*
unfolding *bij-betw-def inj-on-def*
proof *safe*
fix *a b*
assume *a* $\in$ *A* *b* $\in$ *A*
with *left* **have** *a* = *f'* (*f a*) $\wedge$ *b* = *f'* (*f b*) **by** *simp*
moreover assume *f a* = *f b*
ultimately show *a* = *b* **by** *simp*
next
fix *a'* **assume** $*$: *a'* $\in$ *A'*
with *img2* **have** *f'* *a'* $\in$ *A* **by** *blast*
moreover from $*$ *right* **have** *a'* = *f* (*f'* *a'*) **by** *simp*
ultimately show *a'* $\in$ *f* ‘ *A* **by** *blast*
qed

corollary *notIn-Un-bij-betw*:
assumes *b* $\notin$ *A*
and *f b* $\notin$ *A'*
and *bij-betw f A A'*
shows *bij-betw f* (*A* $\cup$ {*b*}) (*A'* $\cup$ {*f b*})
proof $-$
have *bij-betw f* {*b*} {*f b*}
unfolding *bij-betw-def inj-on-def* **by** *simp*
with *assms* **show** *?thesis*
using *bij-betw-combine*[*of f A A'* {*b*} {*f b*}] **by** *blast*

qed

lemma *notIn-Un-bij-betw3*:
assumes $b \notin A$
and $f\ b \notin A'$
shows *bij-betw f A A'* = *bij-betw f* $(A \cup \{b\})$ $(A' \cup \{f\ b\})$
proof
assume *bij-betw f A A'*
then show *bij-betw f* $(A \cup \{b\})$ $(A' \cup \{f\ b\})$
using *assms notIn-Un-bij-betw* [*of b A f A'*] **by** *blast*
next
assume $*$: *bij-betw f* $(A \cup \{b\})$ $(A' \cup \{f\ b\})$
have f ‘ $A = A'$
proof *safe*
fix a
assume $**$: $a \in A$
then have $f\ a \in A' \cup \{f\ b\}$
using $*$ **unfolding** *bij-betw-def* **by** *blast*
moreover
have *False* **if** $f\ a = f\ b$
proof −
have $a = b$
using $*$ $**$ *that* **unfolding** *bij-betw-def inj-on-def* **by** *blast*
with ‹$b \notin A$› $**$ **show** *?thesis* **by** *blast*
qed
ultimately show $f\ a \in A'$ **by** *blast*
next
fix a'
assume $**$: $a' \in A'$
then have $a' \in f$ ‘ $(A \cup \{b\})$
using $*$ **by** (*auto simp add*: *bij-betw-def*)
then obtain a **where** *1*: $a \in A \cup \{b\} \wedge f\ a = a'$ **by** *blast*
moreover
have *False* **if** $a = b$ **using** *1* $**$ ‹$f\ b \notin A'$› *that* **by** *blast*
ultimately have $a \in A$ **by** *blast*
with *1* **show** $a' \in f$ ‘ A **by** *blast*
qed
then show *bij-betw f A A'*
using $*$ *bij-betw-subset*[*of f A* $\cup$ $\{b\}$ - *A*] **by** *blast*
qed

lemma *inj-on-disjoint-Un*:
assumes *inj-on f A* **and** *inj-on g B*
and f ‘ $A \cap g$ ‘ $B = \{\}$
shows *inj-on* ($\lambda x.$ *if* $x \in A$ *then f x else g x*) $(A \cup B)$
using *assms* **by** (*simp add*: *inj-on-def disjoint-iff*) (*blast*)

lemma *bij-betw-disjoint-Un*:
assumes *bij-betw f A C* **and** *bij-betw g B D*

and $A \cap B = \{\}$
and $C \cap D = \{\}$
shows *bij-betw* $(\lambda x.\ \textit{if } x \in A \textit{ then } f\ x \textit{ else } g\ x)\ (A \cup B)\ (C \cup D)$
using *assms* **by** (*auto simp*: *inj-on-disjoint-Un bij-betw-def*)

lemma *involuntory-imp-bij*:
$\langle$*bij f*$\rangle$ **if** $\langle\bigwedge x.\ f\ (f\ x) = x\rangle$
proof (*rule bijI*)
from *that* **show** $\langle$*surj f*$\rangle$
by (*rule surjI*)
show $\langle$*inj f*$\rangle$
proof (*rule injI*)
fix *x y*
assume $\langle f\ x = f\ y\rangle$
then have $\langle f\ (f\ x) = f\ (f\ y)\rangle$
by *simp*
then show $\langle x = y\rangle$
by (*simp add*: *that*)
qed
qed

10.5.1 Inj/surj/bij of Algebraic Operations

context *cancel-semigroup-add*
begin

lemma *inj-on-add* [*simp*]:
inj-on $((+)\ a)\ A$
by (*rule inj-onI*) *simp*

lemma *inj-on-add'* [*simp*]:
inj-on $(\lambda b.\ b + a)\ A$
by (*rule inj-onI*) *simp*

lemma *bij-betw-add* [*simp*]:
bij-betw $((+)\ a)\ A\ B \longleftrightarrow (+)\ a\ `\ A = B$
by (*simp add*: *bij-betw-def*)

end

context *group-add*
begin

lemma *diff-left-imp-eq*: $a - b = a - c \Longrightarrow b = c$
unfolding *add-uminus-conv-diff*[*symmetric*]
by(*drule local.add-left-imp-eq*) *simp*

lemma *inj-uminus*[*simp*, *intro*]: *inj-on uminus A*
by (*auto intro*!: *inj-onI*)

lemma *surj-uminus*[*simp*]: *surj uminus*
using *surjI minus-minus* **by** *blast*

lemma *surj-plus* [*simp*]:
  surj ((+) *a*)
proof (*standard*, *simp*, *standard*, *simp*)
  fix *x*
  have *x* = *a* + (−*a* + *x*) **by** (*simp add*: *add.assoc*)
  thus *x* ∈ *range* ((+) *a*) **by** *blast*
qed

lemma *surj-plus-right* [*simp*]:
  surj (λ*b*. *b*+*a*)
proof (*standard*, *simp*, *standard*, *simp*)
  fix *b* **show** *b* ∈ *range* (λ*b*. *b*+*a*)
    using *diff-add-cancel*[*of b a*, *symmetric*] **by** *blast*
qed

lemma *inj-on-diff-left* [*simp*]:
  ‹*inj-on* ((−) *a*) *A*›
by (*auto intro*: *inj-onI dest*!: *diff-left-imp-eq*)

lemma *inj-on-diff-right* [*simp*]:
  ‹*inj-on* (λ*b*. *b* − *a*) *A*›
by (*auto intro*: *inj-onI simp add*: *algebra-simps*)

lemma *surj-diff* [*simp*]:
  surj ((−) *a*)
proof (*standard*, *simp*, *standard*, *simp*)
  fix *x*
  have *x* = *a* − (− *x* + *a*) **by** (*simp add*: *algebra-simps*)
  thus *x* ∈ *range* ((−) *a*) **by** *blast*
qed

lemma *surj-diff-right* [*simp*]:
  surj (λ*x*. *x* − *a*)
proof (*standard*, *simp*, *standard*, *simp*)
  fix *x*
  have *x* = *x* + *a* − *a* **by** *simp*
  thus *x* ∈ *range* (λ*x*. *x* − *a*) **by** *fast*
qed

lemma shows *bij-plus*: *bij* ((+) *a*) **and** *bij-plus-right*: *bij* (λ*x*. *x* + *a*)
  and *bij-uminus*: *bij uminus*
  and *bij-diff*: *bij* ((−) *a*) **and** *bij-diff-right*: *bij* (λ*x*. *x* − *a*)
by(*simp-all add*: *bij-def*)

lemma *translation-subtract-Compl*:

$(\lambda x.\ x - a)\ `\ (-\ t) = -\ ((\lambda x.\ x - a)\ `\ t)$
by(*rule bij-image-Compl-eq*)
(*auto simp add*: *bij-def surj-def inj-def diff-eq-eq intro*!: *add-diff-cancel*[*symmetric*])

lemma *translation-diff*:
$(+)\ a\ `\ (s - t) = ((+)\ a\ `\ s) - ((+)\ a\ `\ t)$
by *auto*

lemma *translation-subtract-diff*:
$(\lambda x.\ x - a)\ `\ (s - t) = ((\lambda x.\ x - a)\ `\ s) - ((\lambda x.\ x - a)\ `\ t)$
by(*rule image-set-diff*)(*simp add*: *inj-on-def diff-eq-eq*)

lemma *translation-Int*:
$(+)\ a\ `\ (s \cap t) = ((+)\ a\ `\ s) \cap ((+)\ a\ `\ t)$
by *auto*

lemma *translation-subtract-Int*:
$(\lambda x.\ x - a)\ `\ (s \cap t) = ((\lambda x.\ x - a)\ `\ s) \cap ((\lambda x.\ x - a)\ `\ t)$
by(*rule image-Int*)(*simp add*: *inj-on-def diff-eq-eq*)

lemma *translation-Compl*:
$(+)\ a\ `\ (-\ t) = -\ ((+)\ a\ `\ t)$
proof (*rule set-eqI*)
fix *b*
show $b \in (+)\ a\ `\ (-\ t) \longleftrightarrow b \in -\ (+)\ a\ `\ t$
by (*auto simp*: *image-iff algebra-simps intro*!: *bexI* [*of* - − *a* + *b*])
qed

end

10.6 Function Updating

definition *fun-upd* :: $('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow ('a \Rightarrow 'b)$
where *fun-upd f a b* = $(\lambda x.$ *if x* = *a then b else f x*)

nonterminal *updbinds* **and** *updbind*

open-bundle *update-syntax*
begin

syntax
-*updbind* :: $'a \Rightarrow 'a \Rightarrow$ *updbind* (‹(‹*indent*=2 *notation*=‹*mixfix update*››-:=/ -)›)
:: *updbind* ⇒ *updbinds* (‹-›)
-*updbinds*:: *updbind* ⇒ *updbinds* ⇒ *updbinds* (‹-,/ -›)
-*Update* :: $'a \Rightarrow$ *updbinds* ⇒ $'a$
(‹(‹*open-block notation*=‹*mixfix function update*››-/$'$((2-)$'$))› [1000, 0] 900)
syntax-consts
-*Update* ⇌ *fun-upd*

translations
-Update f (-updbinds b bs) ⇌ *-Update (-Update f b) bs*
f(x:=y) ⇌ *CONST fun-upd f x y*

end

lemma *fun-upd-idem-iff*: $f(x{:=}y) = f \longleftrightarrow f\ x = y$
unfolding *fun-upd-def*
apply *safe*
apply (*erule subst*)
apply *auto*
done

lemma *fun-upd-idem*: $f\ x = y \Longrightarrow f(x := y) = f$
by (*simp only*: *fun-upd-idem-iff*)

lemma *fun-upd-triv* [*iff*]: $f(x := f\ x) = f$
by (*simp only*: *fun-upd-idem*)

lemma *fun-upd-apply* [*simp*]: $(f(x := y))\ z = (\mathit{if}\ z = x\ \mathit{then}\ y\ \mathit{else}\ f\ z)$
by (*simp add*: *fun-upd-def*)

lemma *fun-upd-same*: $(f(x := y))\ x = y$
by *simp*

lemma *fun-upd-other*: $z \neq x \Longrightarrow (f(x := y))\ z = f\ z$
by *simp*

lemma *fun-upd-upd* [*simp*]: $f(x := y,\ x := z) = f(x := z)$
by (*simp add*: *fun-eq-iff*)

lemma *fun-upd-twist*: $a \neq c \Longrightarrow (m(a := b))(c := d) = (m(c := d))(a := b)$
by *auto*

lemma *inj-on-fun-updI*: *inj-on f A* $\Longrightarrow$ $y \notin f\ `\ A \Longrightarrow$ *inj-on* $(f(x := y))\ A$
by (*auto simp*: *inj-on-def*)

lemma *fun-upd-image*: $f(x := y)\ `\ A = (\mathit{if}\ x \in A\ \mathit{then}\ \mathit{insert}\ y\ (f\ `\ (A - \{x\}))\ \mathit{else}\ f\ `\ A)$
by *auto*

lemma *fun-upd-comp*: $f \circ (g(x := y)) = (f \circ g)(x := f\ y)$
by *auto*

lemma *fun-upd-eqD*: $f(x := y) = g(x := z) \Longrightarrow y = z$
by (*simp add*: *fun-eq-iff split*: *if-split-asm*)

10.7 *override-on*

definition *override-on* :: $('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where *override-on f g A* = $(\lambda a.\ if\ a \in A\ then\ g\ a\ else\ f\ a)$

lemma *override-on-emptyset*[*simp*]: *override-on f g* $\{\}$ = *f*
by (*simp add*: *override-on-def*)

lemma *override-on-apply-notin*[*simp*]: $a \notin A \Longrightarrow$ (*override-on f g A*) *a* = *f a*
by (*simp add*: *override-on-def*)

lemma *override-on-apply-in*[*simp*]: $a \in A \Longrightarrow$ (*override-on f g A*) *a* = *g a*
by (*simp add*: *override-on-def*)

lemma *override-on-insert*: *override-on f g* (*insert x X*) = (*override-on f g X*)(*x*:=*g x*)
by (*simp add*: *override-on-def fun-eq-iff*)

lemma *override-on-insert'*: *override-on f g* (*insert x X*) = (*override-on* (*f*(*x*:=*g x*)) *g X*)
by (*simp add*: *override-on-def fun-eq-iff*)

10.8 Inversion of injective functions

definition *the-inv-into* :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'a)$
where *the-inv-into A f* = $(\lambda x.\ THE\ y.\ y \in A \wedge f\ y = x)$

lemma *the-inv-into-f-f*: *inj-on f A* $\Longrightarrow x \in A \Longrightarrow$ *the-inv-into A f* (*f x*) = *x*
unfolding *the-inv-into-def inj-on-def* **by** *blast*

lemma *f-the-inv-into-f*: *inj-on f A* $\Longrightarrow y \in f\ `\ A \Longrightarrow$ *f* (*the-inv-into A f y*) = *y*
unfolding *the-inv-into-def*
by (*rule the1I2*; *blast dest*: *inj-onD*)

lemma *f-the-inv-into-f-bij-betw*:
bij-betw f A B $\Longrightarrow$ (*bij-betw f A B* $\Longrightarrow x \in B) \Longrightarrow$ *f* (*the-inv-into A f x*) = *x*
unfolding *bij-betw-def* **by** (*blast intro*: *f-the-inv-into-f*)

lemma *the-inv-into-into*: *inj-on f A* $\Longrightarrow x \in f\ `\ A \Longrightarrow A \subseteq B \Longrightarrow$ *the-inv-into A f x* $\in B$
unfolding *the-inv-into-def*
by (*rule the1I2*; *blast dest*: *inj-onD*)

lemma *the-inv-into-onto* [*simp*]: *inj-on f A* $\Longrightarrow$ *the-inv-into A f* ` (*f* ` *A*) = *A*
by (*fast intro*: *the-inv-into-into the-inv-into-f-f* [*symmetric*])

lemma *the-inv-into-f-eq*: *inj-on f A* $\Longrightarrow f\ x = y \Longrightarrow x \in A \Longrightarrow$ *the-inv-into A f y* = *x*
by (*force simp add*: *the-inv-into-f-f*)

lemma *the-inv-into-comp*:
inj-on f (g ' A) ⟹ inj-on g A ⟹ x ∈ f ' g ' A ⟹
the-inv-into A (f ∘ g) x = (the-inv-into A g ∘ the-inv-into (g ' A) f) x
apply (*rule the-inv-into-f-eq*)
apply (*fast intro*: *comp-inj-on*)
apply (*simp add*: *f-the-inv-into-f the-inv-into-into*)
apply (*simp add*: *the-inv-into-into*)
done

lemma *inj-on-the-inv-into*: *inj-on f A ⟹ inj-on (the-inv-into A f) (f ' A)*
by (*auto intro*: *inj-onI simp*: *the-inv-into-f-f*)

lemma *bij-betw-the-inv-into*: *bij-betw f A B ⟹ bij-betw (the-inv-into A f) B A*
by (*auto simp add*: *bij-betw-def inj-on-the-inv-into the-inv-into-into*)

lemma *bij-betw-iff-bijections*:
bij-betw f A B ⟷ (∃ g. (∀ x ∈ A. f x ∈ B ∧ g(f x) = x) ∧ (∀ y ∈ B. g y ∈ A ∧ f(g y) = y))
(**is** *?lhs = ?rhs*)
proof
show *?lhs ⟹ ?rhs*
by (*auto simp*: *bij-betw-def f-the-inv-into-f the-inv-into-f-f the-inv-into-into exI*[**where** *?x=the-inv-into A f*])
next
show *?rhs ⟹ ?lhs*
by (*force intro*: *bij-betw-byWitness*)
qed

abbreviation *the-inv* :: *('a ⇒ 'b) ⇒ ('b ⇒ 'a)*
where *the-inv f ≡ the-inv-into UNIV f*

lemma *the-inv-f-f*: *the-inv f (f x) = x* **if** *inj f*
using *that UNIV-I* **by** (*rule the-inv-into-f-f*)

10.9 Monotonicity

definition *monotone-on* :: *'a set ⇒ ('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool*
where *monotone-on A orda ordb f ⟷ (∀ x∈A. ∀ y∈A. orda x y ⟶ ordb (f x) (f y))*

abbreviation *monotone* :: *('a ⇒ 'a ⇒ bool) ⇒ ('b ⇒ 'b ⇒ bool) ⇒ ('a ⇒ 'b) ⇒ bool*
where *monotone ≡ monotone-on UNIV*

lemma *monotone-def*[*no-atp*]: *monotone orda ordb f ⟷ (∀ x y. orda x y ⟶ ordb (f x) (f y))*
by (*simp add*: *monotone-on-def*)

Lemma *monotone-def* is provided for backward compatibility.

lemma *monotone-onI*:
(⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *orda x y* ⟹ *ordb* (*f x*) (*f y*)) ⟹ *monotone-on A orda ordb f*
by (*simp add*: *monotone-on-def*)

lemma *monotoneI*[*intro?*]: (⋀*x y*. *orda x y* ⟹ *ordb* (*f x*) (*f y*)) ⟹ *monotone orda ordb f*
by (*rule monotone-onI*)

lemma *monotone-onD*:
monotone-on A orda ordb f ⟹ *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *orda x y* ⟹ *ordb* (*f x*) (*f y*)
by (*simp add*: *monotone-on-def*)

lemma *monotoneD*[*dest?*]: *monotone orda ordb f* ⟹ *orda x y* ⟹ *ordb* (*f x*) (*f y*)
by (*rule monotone-onD*[*of UNIV*, *simplified*])

lemma *monotone-on-subset*: *monotone-on A orda ordb f* ⟹ *B* ⊆ *A* ⟹ *monotone-on B orda ordb f*
by (*auto intro*: *monotone-onI dest*: *monotone-onD*)

lemma *monotone-on-empty*[*simp*]: *monotone-on* {} *orda ordb f*
by (*auto intro*: *monotone-onI dest*: *monotone-onD*)

lemma *monotone-on-o*:
assumes
mono-f: *monotone-on A orda ordb f* **and**
mono-g: *monotone-on B ordc orda g* **and**
g ‘ *B* ⊆ *A*
shows *monotone-on B ordc ordb* (*f* ∘ *g*)
proof (*rule monotone-onI*)
fix *x y* **assume** *x* ∈ *B* **and** *y* ∈ *B* **and** *ordc x y*
hence *orda* (*g x*) (*g y*)
by (*rule mono-g*[*THEN monotone-onD*])
moreover from ‹*g* ‘ *B* ⊆ *A*› ‹*x* ∈ *B*› ‹*y* ∈ *B*› **have** *g x* ∈ *A* **and** *g y* ∈ *A*
unfolding *image-subset-iff* **by** *simp-all*
ultimately show *ordb* ((*f* ∘ *g*) *x*) ((*f* ∘ *g*) *y*)
using *mono-f*[*THEN monotone-onD*] **by** *simp*
qed

10.9.1 Specializations For *ord* Type Class And More

context *ord* **begin**

abbreviation *mono-on* :: *'a set* ⇒ (*'a* ⇒ *'b* :: *ord*) ⇒ *bool*
where *mono-on A* ≡ *monotone-on A* (≤) (≤)

abbreviation *strict-mono-on* :: *'a set* ⇒ (*'a* ⇒ *'b* :: *ord*) ⇒ *bool*
where *strict-mono-on A* ≡ *monotone-on A* (<) (<)

abbreviation *antimono-on* :: *'a set* ⇒ (*'a* ⇒ *'b* :: *ord*) ⇒ *bool*
where *antimono-on A* ≡ *monotone-on A* (≤) (λ*x y*. *y* ≤ *x*)

abbreviation *strict-antimono-on* :: *'a set* ⇒ (*'a* ⇒ *'b* :: *ord*) ⇒ *bool*
where *strict-antimono-on A* ≡ *monotone-on A* (<) (λ*x y*. *y* < *x*)

lemma *mono-on-def*[*no-atp*]: *mono-on A f* ⟷ (∀ *r s*. *r* ∈ *A* ∧ *s* ∈ *A* ∧ *r* ≤ *s* ⟶ *f r* ≤ *f s*)
by (*auto simp add*: *monotone-on-def*)

lemma *strict-mono-on-def*[*no-atp*]:
strict-mono-on A f ⟷ (∀ *r s*. *r* ∈ *A* ∧ *s* ∈ *A* ∧ *r* < *s* ⟶ *f r* < *f s*)
by (*auto simp add*: *monotone-on-def*)

Lemmas *mono-on-def* and *strict-mono-on-def* are provided for backward compatibility.

lemma *mono-onI*:
(⋀*r s*. *r* ∈ *A* ⟹ *s* ∈ *A* ⟹ *r* ≤ *s* ⟹ *f r* ≤ *f s*) ⟹ *mono-on A f*
by (*rule monotone-onI*)

lemma *strict-mono-onI*:
(⋀*r s*. *r* ∈ *A* ⟹ *s* ∈ *A* ⟹ *r* < *s* ⟹ *f r* < *f s*) ⟹ *strict-mono-on A f*
by (*rule monotone-onI*)

lemma *mono-onD*: ⟦*mono-on A f*; *r* ∈ *A*; *s* ∈ *A*; *r* ≤ *s*⟧ ⟹ *f r* ≤ *f s*
by (*rule monotone-onD*)

lemma *strict-mono-onD*: ⟦*strict-mono-on A f*; *r* ∈ *A*; *s* ∈ *A*; *r* < *s*⟧ ⟹ *f r* < *f s*
by (*rule monotone-onD*)

lemma *mono-on-subset*: *mono-on A f* ⟹ *B* ⊆ *A* ⟹ *mono-on B f*
by (*rule monotone-on-subset*)

end

context *order* **begin**

abbreviation *mono* :: (*'a* ⇒ *'b*::*order*) ⇒ *bool*
where *mono* ≡ *mono-on UNIV*

abbreviation *strict-mono* :: (*'a* ⇒ *'b*::*order*) ⇒ *bool*
where *strict-mono* ≡ *strict-mono-on UNIV*

abbreviation *antimono* :: (*'a* ⇒ *'b*::*order*) ⇒ *bool*
where *antimono* ≡ *monotone* (≤) (λ*x y*. *y* ≤ *x*)

abbreviation *strict-antimono* :: (*'a* ⇒ *'b*::*preorder*) ⇒ *bool*
where *strict-antimono* ≡ *monotone* (<) (λ*x y*. *y* < *x*)

lemma *mono-def*[*no-atp*]: *mono f* $\longleftrightarrow$ $(\forall x\ y.\ x \leq y \longrightarrow f\ x \leq f\ y)$
by (*simp add*: *monotone-on-def*)

lemma *strict-mono-def*[*no-atp*]: *strict-mono f* $\longleftrightarrow$ $(\forall x\ y.\ x < y \longrightarrow f\ x < f\ y)$
by (*simp add*: *monotone-on-def*)

lemma *antimono-def*[*no-atp*]: *antimono f* $\longleftrightarrow$ $(\forall x\ y.\ x \leq y \longrightarrow f\ x \geq f\ y)$
by (*simp add*: *monotone-on-def*)

Lemmas *mono-def*, *strict-mono-def*, and *antimono-def* are provided for backward compatibility.

lemma *monoI* [*intro?*]: $(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \leq f\ y) \Longrightarrow$ *mono f*
by (*rule monotoneI*)

lemma *strict-monoI* [*intro?*]: $(\bigwedge x\ y.\ x < y \Longrightarrow f\ x < f\ y) \Longrightarrow$ *strict-mono f*
by (*rule monotoneI*)

lemma *antimonoI* [*intro?*]: $(\bigwedge x\ y.\ x \leq y \Longrightarrow f\ x \geq f\ y) \Longrightarrow$ *antimono f*
by (*rule monotoneI*)

lemma *monoD* [*dest?*]: *mono f* $\Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
by (*rule monotoneD*)

lemma *strict-monoD* [*dest?*]: *strict-mono f* $\Longrightarrow x < y \Longrightarrow f\ x < f\ y$
by (*rule monotoneD*)

lemma *antimonoD* [*dest?*]: *antimono f* $\Longrightarrow x \leq y \Longrightarrow f\ x \geq f\ y$
by (*rule monotoneD*)

lemma *monoE*:
assumes *mono f*
assumes $x \leq y$
obtains $f\ x \leq f\ y$
proof
from *assms* **show** $f\ x \leq f\ y$ **by** (*simp add*: *mono-def*)
qed

lemma *antimonoE*:
fixes $f :: 'a \Rightarrow 'b$::*order*
assumes *antimono f*
assumes $x \leq y$
obtains $f\ x \geq f\ y$
proof
from *assms* **show** $f\ x \geq f\ y$ **by** (*simp add*: *antimono-def*)
qed

end

lemma *mono-imp-mono-on*: *mono f* $\Longrightarrow$ *mono-on A f*
by (*rule monotone-on-subset*[*OF - subset-UNIV*])

lemma *strict-mono-on-imp-mono-on*: *strict-mono-on A f* $\Longrightarrow$ *mono-on A f*
for *f* :: *'a*::*order* $\Rightarrow$ *'b*::*preorder*
proof (*intro mono-onI*)
fix *r s* :: *'a* **assume** *asm*: *r* $\leq$ *s strict-mono-on A f r* $\in$ *A s* $\in$ *A*
from *this*(*1*) **consider** *r* $<$ *s* | *r* = *s* **by** *fastforce*
then show *f r* $\leq$ *f s*
proof(*cases*)
case *1*
from *strict-mono-onD*[*OF asm*(*2*−*4*) *this*] **show** *?thesis* **by** (*fact order.strict-implies-order*)
qed *simp*
qed

lemma *strict-mono-mono* [*dest?*]:
strict-mono f $\Longrightarrow$ *mono f*
by (*fact strict-mono-on-imp-mono-on*)

lemma *mono-on-ident*: *mono-on S* (*λx. x*)
by (*intro monotone-onI*)

lemma *mono-on-id*: *mono-on S id*
unfolding *id-def* **by** (*fact mono-on-ident*)

lemma *strict-mono-on-ident*: *strict-mono-on S* (*λx. x*)
by (*intro monotone-onI*)

lemma *strict-mono-on-id*: *strict-mono-on S id*
unfolding *id-def* **by** (*fact strict-mono-on-ident*)

lemma *mono-on-const*:
fixes *a* :: *'b*::*preorder* **shows** *mono-on S* (*λx. a*)
by (*intro monotone-onI order.refl*)

lemma *antimono-on-const*:
fixes *a* :: *'b*::*preorder* **shows** *antimono-on S* (*λx. a*)
by (*intro monotone-onI order.refl*)

context *linorder* **begin**

lemma *mono-on-strict-invE*:
fixes *f* :: *'a* $\Rightarrow$ *'b*::*preorder*
assumes *mono-on S f*
assumes *x* $\in$ *S y* $\in$ *S*
assumes *f x* $<$ *f y*
obtains *x* $<$ *y*
proof

```
  show x < y
  proof (rule ccontr)
    assume ¬ x < y
    then have y ≤ x by simp
     with ‹mono-on S f› ‹x ∈ S› ‹y ∈ S› have f y ≤ f x by (simp only: mono-
tone-onD)
    with ‹f x < f y› show False by (simp add: preorder-class.less-le-not-le)
  qed
qed

corollary mono-on-invE:
  fixes f :: 'a ⇒ 'b::preorder
  assumes mono-on S f
  assumes x ∈ S y ∈ S
  assumes f x < f y
  obtains x ≤ y
  using assms mono-on-strict-invE[of S f x y thesis] by simp

lemma strict-mono-on-eq:
  assumes strict-mono-on S (f::'a ⇒ 'b::preorder)
  assumes x ∈ S y ∈ S
  shows f x = f y ⟷ x = y
proof
  assume f x = f y
  show x = y proof (cases x y rule: linorder-cases)
    case less with assms have f x < f y by (simp add: monotone-onD)
    with ‹f x = f y› show ?thesis by simp
  next
    case equal then show ?thesis .
  next
    case greater with assms have f y < f x by (simp add: monotone-onD)
    with ‹f x = f y› show ?thesis by simp
  qed
qed simp

lemma strict-mono-on-less-eq:
  assumes strict-mono-on S (f::'a ⇒ 'b::preorder)
  assumes x ∈ S y ∈ S
  shows f x ≤ f y ⟷ x ≤ y
proof
  assume x ≤ y
  then show f x ≤ f y
    using nless-le[of x y] monotone-onD[OF assms] order-less-imp-le[of f x f y]
    by blast
next
  assume f x ≤ f y
  show x ≤ y
  proof (rule ccontr)
    assume ¬ x ≤ y
```

```
    then have y < x by simp
    with assms have f y < f x by (simp add: monotone-onD)
    with ‹f x ≤ f y› show False by (simp add: preorder-class.less-le-not-le)
  qed
qed

lemma strict-mono-on-less:
  assumes strict-mono-on S (f::'a ⇒ -::preorder)
  assumes x ∈ S y ∈ S
  shows f x < f y ⟷ x < y
  using assms strict-mono-on-eq[of S f x y]
  by (auto simp add: strict-mono-on-less-eq preorder-class.less-le-not-le)

lemmas mono-invE = mono-on-invE[OF - UNIV-I UNIV-I]
lemmas mono-strict-invE = mono-on-strict-invE[OF - UNIV-I UNIV-I]
lemmas strict-mono-eq = strict-mono-on-eq[OF - UNIV-I UNIV-I]
lemmas strict-mono-less-eq = strict-mono-on-less-eq[OF - UNIV-I UNIV-I]
lemmas strict-mono-less = strict-mono-on-less[OF - UNIV-I UNIV-I]

end

lemma strict-mono-inv:
  fixes f :: ('a::linorder) ⇒ ('b::preorder)
  assumes strict-mono-on UNIV f and surj f and inv: ⋀x. g (f x) = x
  shows strict-mono-on UNIV g
proof
  fix x y :: 'b assume x < y
  from ‹surj f› obtain x' y' where [simp]: x = f x' y = f y' by blast
   with ‹x < y› and ‹strict-mono-on UNIV f› have x' < y' by (simp add:
strict-mono-less)
  with inv show g x < g y by simp
qed

lemma strict-mono-on-imp-inj-on:
  fixes f :: 'a::linorder ⇒ 'b::preorder
  assumes strict-mono-on A f
  shows inj-on f A
proof (rule inj-onI)
  fix x y assume x ∈ A y ∈ A f x = f y
  thus x = y
    by (cases x y rule: linorder-cases)
       (auto dest: strict-mono-onD[OF assms, of x y] strict-mono-onD[OF assms,
of y x])
qed

lemma strict-mono-on-leD:
  fixes f :: 'a::order ⇒ 'b::preorder
  assumes strict-mono-on A f x ∈ A y ∈ A x ≤ y
  shows f x ≤ f y
```

proof (*cases* $x = y$)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  with *assms* **have** $f\ x < f\ y$
    using *strict-mono-onD*[*OF assms*(*1*)] **by** *simp*
  then show *?thesis* **by** (*rule less-imp-le*)
qed

lemma *strict-mono-on-eqD*:
  fixes $f :: 'c::linorder \Rightarrow 'd::preorder$
  assumes *strict-mono-on A f f x* $= f\ y\ x \in A\ y \in A$
  shows $y = x$
  using *assms* **by** (*cases rule*: *linorder-cases*) (*auto dest*: *strict-mono-onD*)

lemma *mono-imp-strict-mono*:
  fixes $f :: 'a::preorder \Rightarrow 'b::order$
  shows $[\![mono\text{-}on\ S\ f;\ inj\text{-}on\ f\ S]\!] \Longrightarrow strict\text{-}mono\text{-}on\ S\ f$
  by (*auto simp add*: *monotone-on-def less-imp-le order-less-le inj-on-eq-iff*)

lemma *strict-mono-iff-mono*:
  fixes $f :: 'a::linorder \Rightarrow 'b::order$
  shows $strict\text{-}mono\text{-}on\ S\ f \longleftrightarrow mono\text{-}on\ S\ f \wedge inj\text{-}on\ f\ S$
proof
  show $strict\text{-}mono\text{-}on\ S\ f \Longrightarrow mono\text{-}on\ S\ f \wedge inj\text{-}on\ f\ S$
    by (*simp add*: *strict-mono-on-imp-inj-on strict-mono-on-imp-mono-on*)
qed (*auto intro*: *mono-imp-strict-mono*)

lemma *antimono-imp-strict-antimono*:
  fixes $f :: 'a::preorder \Rightarrow 'b::order$
  shows $[\![antimono\text{-}on\ S\ f;\ inj\text{-}on\ f\ S]\!] \Longrightarrow strict\text{-}antimono\text{-}on\ S\ f$
  by (*auto simp add*: *monotone-on-def less-imp-le order-less-le inj-on-eq-iff*)

lemma *strict-antimono-iff-antimono*:
  fixes $f :: 'a::linorder \Rightarrow 'b::order$
  shows $strict\text{-}antimono\text{-}on\ S\ f \longleftrightarrow antimono\text{-}on\ S\ f \wedge inj\text{-}on\ f\ S$
proof
  show $strict\text{-}antimono\text{-}on\ S\ f \Longrightarrow antimono\text{-}on\ S\ f \wedge inj\text{-}on\ f\ S$
    by (*force simp add*: *monotone-on-def intro*: *linorder-inj-onI*)
qed (*auto intro*: *antimono-imp-strict-antimono*)

lemma *mono-compose*: $mono\ Q \Longrightarrow mono\ (\lambda i\ x.\ Q\ i\ (f\ x))$
  unfolding *mono-def le-fun-def* **by** *auto*

lemma *mono-add*:
  fixes $a :: 'a::ordered\text{-}ab\text{-}semigroup\text{-}add$
  shows $mono\ ((+)\ a)$
  by (*simp add*: *add-left-mono monoI*)

lemma (**in** *semilattice-inf*) *mono-inf*: *mono f* $\Longrightarrow$ *f* (*A* $\sqcap$ *B*) $\leq$ *f A* $\sqcap$ *f B*
for *f* :: *'a* $\Rightarrow$ *'b*::*semilattice-inf*
by (*auto simp add*: *mono-def intro*: *Lattices.inf-greatest*)

lemma (**in** *semilattice-sup*) *mono-sup*: *mono f* $\Longrightarrow$ *f A* $\sqcup$ *f B* $\leq$ *f* (*A* $\sqcup$ *B*)
for *f* :: *'a* $\Rightarrow$ *'b*::*semilattice-sup*
by (*auto simp add*: *mono-def intro*: *Lattices.sup-least*)

lemma *monotone-on-sup-fun*:
fixes *f g* :: - $\Rightarrow$ -:: *semilattice-sup*
shows *monotone-on A P* ($\leq$) *f* $\Longrightarrow$ *monotone-on A P* ($\leq$) *g* $\Longrightarrow$ *monotone-on A P* ($\leq$) (*f* $\sqcup$ *g*)
by (*auto intro*: *monotone-onI sup-mono dest*: *monotone-onD simp*: *sup-fun-def*)

lemma *monotone-on-inf-fun*:
fixes *f g* :: - $\Rightarrow$ -:: *semilattice-inf*
shows *monotone-on A P* ($\leq$) *f* $\Longrightarrow$ *monotone-on A P* ($\leq$) *g* $\Longrightarrow$ *monotone-on A P* ($\leq$) (*f* $\sqcap$ *g*)
by (*auto intro*: *monotone-onI inf-mono dest*: *monotone-onD simp*: *inf-fun-def*)

lemma *antimonotone-on-sup-fun*:
fixes *f g* :: - $\Rightarrow$ -:: *semilattice-sup*
shows *monotone-on A P* ($\geq$) *f* $\Longrightarrow$ *monotone-on A P* ($\geq$) *g* $\Longrightarrow$ *monotone-on A P* ($\geq$) (*f* $\sqcup$ *g*)
by (*auto intro*: *monotone-onI sup-mono dest*: *monotone-onD simp*: *sup-fun-def*)

lemma *antimonotone-on-inf-fun*:
fixes *f g* :: - $\Rightarrow$ -:: *semilattice-inf*
shows *monotone-on A P* ($\geq$) *f* $\Longrightarrow$ *monotone-on A P* ($\geq$) *g* $\Longrightarrow$ *monotone-on A P* ($\geq$) (*f* $\sqcap$ *g*)
by (*auto intro*: *monotone-onI inf-mono dest*: *monotone-onD simp*: *inf-fun-def*)

lemma (**in** *linorder*) *min-of-mono*: *mono f* $\Longrightarrow$ *min* (*f m*) (*f n*) = *f* (*min m n*)
by (*auto simp*: *mono-def Orderings.min-def min-def intro*: *Orderings.antisym*)

lemma (**in** *linorder*) *max-of-mono*: *mono f* $\Longrightarrow$ *max* (*f m*) (*f n*) = *f* (*max m n*)
by (*auto simp*: *mono-def Orderings.max-def max-def intro*: *Orderings.antisym*)

lemma (**in** *linorder*)
max-of-antimono: *antimono f* $\Longrightarrow$ *max* (*f x*) (*f y*) = *f* (*min x y*) **and**
min-of-antimono: *antimono f* $\Longrightarrow$ *min* (*f x*) (*f y*) = *f* (*max x y*)
by (*auto simp*: *antimono-def Orderings.max-def max-def Orderings.min-def min-def intro*!: *antisym*)

lemma (**in** *linorder*) *strict-mono-imp-inj-on*: *strict-mono f* $\Longrightarrow$ *inj-on f A*
by (*auto intro*!: *inj-onI dest*: *strict-mono-eq*)

lemma *mono-Int*: *mono f* $\Longrightarrow$ *f* (*A* $\cap$ *B*) $\subseteq$ *f A* $\cap$ *f B*

by (*fact mono-inf*)

lemma *mono-Un*: *mono f* $\Longrightarrow$ *f A* $\cup$ *f B* $\subseteq$ *f* (*A* $\cup$ *B*)
by (*fact mono-sup*)

10.9.2 Least value operator

lemma *Least-mono*: *mono f* $\Longrightarrow$ $\exists$ *x*$\in$*S*. $\forall$ *y*$\in$*S*. *x* $\leq$ *y* $\Longrightarrow$ (*LEAST y*. *y* $\in$ *f* ' *S*) = *f* (*LEAST x*. *x* $\in$ *S*)
for *f* :: *'a*::*order* $\Rightarrow$ *'b*::*order*
— Courtesy of Stephan Merz
apply *clarify*
apply (*erule-tac P* = λ*x*. *x* $\in$ *S* **in** *LeastI2-order*)
apply *fast*
apply (*rule LeastI2-order*)
apply (*auto elim*: *monoD intro*!: *order-antisym*)
done

10.10 Setup

10.10.1 Proof tools

Simplify terms of the form *f*(...,*x*:=*y*,...,*x*:=*z*,...) to *f*(...,*x*:=*z*,...)

simproc-setup *fun-upd2* (*f*(*v* := *w*, *x* := *y*)) = ‹
let
fun gen-fun-upd - - - - NONE = *NONE*
| *gen-fun-upd A B x y* (*SOME f*) = *SOME* ***Const*** ‹*fun-upd A B for f x y*›
fun find-double (*t as* ***Const-*** ‹*fun-upd A B for f x y*›) =
let
fun find ***Const-*** ‹*fun-upd - - for g v w*› =
if v aconv x then SOME g
else gen-fun-upd A B v w (*find g*)
| *find t* = *NONE*
in gen-fun-upd A B x y (*find f*) *end*

val ss = *Simplifier.simpset-of* ***context***
in
fn - => fn ctxt => fn ct =>
let val t = *Thm.term-of ct in*
find-double t |> *Option.map* (*fn rhs =>*
Goal.prove ctxt [] [] (*Logic.mk-equals* (*t*, *rhs*))
(*fn - =>*
resolve-tac ctxt [*eq-reflection*] *1 THEN*
resolve-tac ctxt @{*thms ext*} *1 THEN*
simp-tac (*put-simpset ss ctxt*) *1*))
end
end
›

10.10.2 Functorial structure of types

ML-file ‹*Tools/functor.ML*›

functor *map-fun*: *map-fun*
  by (*simp-all add*: *fun-eq-iff*)

functor *vimage*
  by (*simp-all add*: *fun-eq-iff vimage-comp*)

Legacy theorem names

lemmas *o-def* = *comp-def*
lemmas *o-apply* = *comp-apply*
lemmas *o-assoc* = *comp-assoc* [*symmetric*]
lemmas *id-o* = *id-comp*
lemmas *o-id* = *comp-id*
lemmas *o-eq-dest* = *comp-eq-dest*
lemmas *o-eq-elim* = *comp-eq-elim*
lemmas *o-eq-dest-lhs* = *comp-eq-dest-lhs*
lemmas *o-eq-id-dest* = *comp-eq-id-dest*

end

11 Complete lattices

theory *Complete-Lattices*
  imports *Fun*
begin

11.1 Syntactic infimum and supremum operations

class *Inf* =
  fixes *Inf* :: *'a set* ⇒ *'a* (‹(‹*open-block notation=‹prefix* ⨅››⨅ -)› [*900*] *900*)

class *Sup* =
  fixes *Sup* :: *'a set* ⇒ *'a* (‹(‹*open-block notation=‹prefix* ⨆››⨆ -)› [*900*] *900*)

syntax
  -INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder INF››INF* -./ -)› [*0*, *10*] *10*)
  -INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder INF››INF* -∈-./ -)› [*0*, *0*, *10*] *10*)
  -SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder SUP››SUP* -./ -)› [*0*, *10*] *10*)
  -SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder SUP››SUP* -∈-./ -)› [*0*, *0*, *10*] *10*)

syntax

-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* ⨅››⨅ -./ -)› [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* ⨅››⨅ -∈-./ -)› [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* ⨆››⨆ -./ -)› [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* ⨆››⨆ -∈-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-INF1 -INF ⇌ *Inf* **and**
-SUP1 -SUP ⇌ *Sup*

translations
⨅*x y. f* ⇌ ⨅*x.* ⨅*y. f*
⨅*x. f* ⇌ ⨅(*CONST range* (λ*x. f*))
⨅*x*∈*A. f* ⇌ *CONST Inf* ((λ*x. f*) ‘ *A*)
⨆*x y. f* ⇌ ⨆*x.* ⨆*y. f*
⨆*x. f* ⇌ ⨆(*CONST range* (λ*x. f*))
⨆*x*∈*A. f* ⇌ *CONST Sup* ((λ*x. f*) ‘ *A*)

context *Inf*
begin

lemma *INF-image*: ⨅ (*g* ‘ *f* ‘ *A*) = ⨅ ((*g* ∘ *f*) ‘ *A*)
by (*simp add*: *image-comp*)

lemma *INF-identity-eq* [*simp*]: (⨅*x*∈*A. x*) = ⨅*A*
by *simp*

lemma *INF-id-eq* [*simp*]: ⨅(*id* ‘ *A*) = ⨅*A*
by *simp*

lemma *INF-cong*: *A* = *B* ⟹ (⋀*x. x* ∈ *B* ⟹ *C x* = *D x*) ⟹ ⨅(*C* ‘ *A*) = ⨅(*D* ‘ *B*)
by (*simp add*: *image-def*)

lemma *INF-cong-simp*:
A = *B* ⟹ (⋀*x. x* ∈ *B* =*simp*=> *C x* = *D x*) ⟹ ⨅(*C* ‘ *A*) = ⨅(*D* ‘ *B*)
unfolding *simp-implies-def* **by** (*fact INF-cong*)

end

context *Sup*
begin

lemma *SUP-image*: ⨆ (*g* ‘ *f* ‘ *A*) = ⨆ ((*g* ∘ *f*) ‘ *A*)
by(*fact Inf.INF-image*)

lemma *SUP-identity-eq* [*simp*]: $(\bigsqcup x{\in}A.\ x) = \bigsqcup A$
by(*fact Inf.INF-identity-eq*)

lemma *SUP-id-eq* [*simp*]: $\bigsqcup (id\ `\ A) = \bigsqcup A$
by(*fact Inf.INF-id-eq*)

lemma *SUP-cong*: $A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow C\ x = D\ x) \Longrightarrow \bigsqcup (C\ `\ A) = \bigsqcup (D\ `\ B)$
by (*fact Inf.INF-cong*)

lemma *SUP-cong-simp*:
$A = B \Longrightarrow (\bigwedge x.\ x \in B =simp=> C\ x = D\ x) \Longrightarrow \bigsqcup (C\ `\ A) = \bigsqcup (D\ `\ B)$
by (*fact Inf.INF-cong-simp*)

end

11.2 Abstract complete lattices

A complete lattice always has a bottom and a top, so we include them into the following type class, along with assumptions that define bottom and top in terms of infimum and supremum.

class *complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *Inf-lower*: $x \in A \Longrightarrow \bigsqcap A \leq x$
and *Inf-greatest*: $(\bigwedge x.\ x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq \bigsqcap A$
and *Sup-upper*: $x \in A \Longrightarrow x \leq \bigsqcup A$
and *Sup-least*: $(\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow \bigsqcup A \leq z$
and *Inf-empty* [*simp*]: $\bigsqcap \{\} = \top$
and *Sup-empty* [*simp*]: $\bigsqcup \{\} = \bot$
begin

subclass *bounded-lattice*
proof
fix *a*
show $\bot \leq a$
by (*auto intro*: *Sup-least simp only*: *Sup-empty* [*symmetric*])
show $a \leq \top$
by (*auto intro*: *Inf-greatest simp only*: *Inf-empty* [*symmetric*])
qed

lemma *dual-complete-lattice*: *class.complete-lattice Sup Inf sup* $(\geq)$ $(>)$ *inf* $\top$ $\bot$
by (*auto intro*!: *class.complete-lattice.intro dual-lattice*)
(*unfold-locales*, (*fact Inf-empty Sup-empty Sup-upper Sup-least Inf-lower Inf-greatest*)+)

end

context *complete-lattice*
begin

lemma *Sup-eqI*:

$(\bigwedge y.\ y \in A \Longrightarrow y \leq x) \Longrightarrow (\bigwedge y.\ (\bigwedge z.\ z \in A \Longrightarrow z \leq y) \Longrightarrow x \leq y) \Longrightarrow \bigsqcup A = x$
by (*blast intro*: *order.antisym Sup-least Sup-upper*)

lemma *Inf-eqI*:
$(\bigwedge i.\ i \in A \Longrightarrow x \leq i) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow y \leq i) \Longrightarrow y \leq x) \Longrightarrow \bigsqcap A = x$
by (*blast intro*: *order.antisym Inf-greatest Inf-lower*)

lemma *SUP-eqI*:
$(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq x) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow f\ i \leq y) \Longrightarrow x \leq y) \Longrightarrow (\bigsqcup i{\in}A.\ f\ i) = x$
using *Sup-eqI* [*of f ' A x*] **by** *auto*

lemma *INF-eqI*:
$(\bigwedge i.\ i \in A \Longrightarrow x \leq f\ i) \Longrightarrow (\bigwedge y.\ (\bigwedge i.\ i \in A \Longrightarrow f\ i \geq y) \Longrightarrow x \geq y) \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) = x$
using *Inf-eqI* [*of f ' A x*] **by** *auto*

lemma *INF-lower*: $i \in A \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) \leq f\ i$
using *Inf-lower* [*of - f ' A*] **by** *simp*

lemma *INF-greatest*: $(\bigwedge i.\ i \in A \Longrightarrow u \leq f\ i) \Longrightarrow u \leq (\bigsqcap i{\in}A.\ f\ i)$
using *Inf-greatest* [*of f ' A*] **by** *auto*

lemma *SUP-upper*: $i \in A \Longrightarrow f\ i \leq (\bigsqcup i{\in}A.\ f\ i)$
using *Sup-upper* [*of - f ' A*] **by** *simp*

lemma *SUP-least*: $(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow (\bigsqcup i{\in}A.\ f\ i) \leq u$
using *Sup-least* [*of f ' A*] **by** *auto*

lemma *Inf-lower2*: $u \in A \Longrightarrow u \leq v \Longrightarrow \bigsqcap A \leq v$
using *Inf-lower* [*of u A*] **by** *auto*

lemma *INF-lower2*: $i \in A \Longrightarrow f\ i \leq u \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) \leq u$
using *INF-lower* [*of i A f*] **by** *auto*

lemma *Sup-upper2*: $u \in A \Longrightarrow v \leq u \Longrightarrow v \leq \bigsqcup A$
using *Sup-upper* [*of u A*] **by** *auto*

lemma *SUP-upper2*: $i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq (\bigsqcup i{\in}A.\ f\ i)$
using *SUP-upper* [*of i A f*] **by** *auto*

lemma *le-Inf-iff*: $b \leq \bigsqcap A \longleftrightarrow (\forall\, a{\in}A.\ b \leq a)$
by (*auto intro*: *Inf-greatest dest*: *Inf-lower*)

lemma *le-INF-iff*: $u \leq (\bigsqcap i{\in}A.\ f\ i) \longleftrightarrow (\forall\, i{\in}A.\ u \leq f\ i)$
using *le-Inf-iff* [*of - f ' A*] **by** *simp*

lemma *Sup-le-iff*: $\bigsqcup A \leq b \longleftrightarrow (\forall\, a{\in}A.\ a \leq b)$

by (*auto intro*: *Sup-least dest*: *Sup-upper*)

lemma *SUP-le-iff*: $(\bigsqcup i{\in}A.\ f\ i) \leq u \longleftrightarrow (\forall\, i{\in}A.\ f\ i \leq u)$
using *Sup-le-iff* [*of f ' A*] **by** *simp*

lemma *Inf-insert* [*simp*]: $\bigsqcap(\mathit{insert}\ a\ A) = a \sqcap \bigsqcap A$
by (*auto intro*: *le-infI le-infI1 le-infI2 order.antisym Inf-greatest Inf-lower*)

lemma *INF-insert*: $(\bigsqcap x{\in}\mathit{insert}\ a\ A.\ f\ x) = f\ a \sqcap \bigsqcap(f\ `\ A)$
by *simp*

lemma *Sup-insert* [*simp*]: $\bigsqcup(\mathit{insert}\ a\ A) = a \sqcup \bigsqcup A$
by (*auto intro*: *le-supI le-supI1 le-supI2 order.antisym Sup-least Sup-upper*)

lemma *SUP-insert*: $(\bigsqcup x{\in}\mathit{insert}\ a\ A.\ f\ x) = f\ a \sqcup \bigsqcup(f\ `\ A)$
by *simp*

lemma *INF-empty*: $(\bigsqcap x{\in}\{\}.\ f\ x) = \top$
by *simp*

lemma *SUP-empty*: $(\bigsqcup x{\in}\{\}.\ f\ x) = \bot$
by *simp*

lemma *Inf-UNIV* [*simp*]: $\bigsqcap \mathit{UNIV} = \bot$
by (*auto intro*!: *order.antisym Inf-lower*)

lemma *Sup-UNIV* [*simp*]: $\bigsqcup \mathit{UNIV} = \top$
by (*auto intro*!: *order.antisym Sup-upper*)

lemma *Inf-eq-Sup*: $\bigsqcap A = \bigsqcup\{b.\ \forall\, a \in A.\ b \leq a\}$
by (*auto intro*: *order.antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Sup-eq-Inf*: $\bigsqcup A = \bigsqcap\{b.\ \forall\, a \in A.\ a \leq b\}$
by (*auto intro*: *order.antisym Inf-lower Inf-greatest Sup-upper Sup-least*)

lemma *Inf-superset-mono*: $B \subseteq A \Longrightarrow \bigsqcap A \leq \bigsqcap B$
by (*auto intro*: *Inf-greatest Inf-lower*)

lemma *Sup-subset-mono*: $A \subseteq B \Longrightarrow \bigsqcup A \leq \bigsqcup B$
by (*auto intro*: *Sup-least Sup-upper*)

lemma *Inf-mono*:
assumes $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
shows $\bigsqcap A \leq \bigsqcap B$
proof (*rule Inf-greatest*)
fix *b* **assume** $b \in B$
with *assms* **obtain** *a* **where** $a \in A$ **and** $a \leq b$ **by** *blast*
from ‹$a \in A$› **have** $\bigsqcap A \leq a$ **by** (*rule Inf-lower*)
with ‹$a \leq b$› **show** $\bigsqcap A \leq b$ **by** *auto*

qed

lemma *INF-mono*: $(\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow (\bigsqcap n{\in}A.\ f\ n) \leq (\bigsqcap n{\in}B.\ g\ n)$
using *Inf-mono* [*of g ' B f ' A*] **by** *auto*

lemma *INF-mono'*: $(\bigwedge x.\ f\ x \leq g\ x) \Longrightarrow (\bigsqcap x{\in}A.\ f\ x) \leq (\bigsqcap x{\in}A.\ g\ x)$
by (*rule INF-mono*) *auto*

lemma *Sup-mono*:
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\, b{\in}B.\ a \leq b$
shows $\bigsqcup A \leq \bigsqcup B$
proof (*rule Sup-least*)
fix *a* **assume** $a \in A$
with *assms* **obtain** *b* **where** $b \in B$ **and** $a \leq b$ **by** *blast*
from ‹$b \in B$› **have** $b \leq \bigsqcup B$ **by** (*rule Sup-upper*)
with ‹$a \leq b$› **show** $a \leq \bigsqcup B$ **by** *auto*
qed

lemma *SUP-mono*: $(\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow (\bigsqcup n{\in}A.\ f\ n) \leq (\bigsqcup n{\in}B.\ g\ n)$
using *Sup-mono* [*of f ' A g ' B*] **by** *auto*

lemma *SUP-mono'*: $(\bigwedge x.\ f\ x \leq g\ x) \Longrightarrow (\bigsqcup x{\in}A.\ f\ x) \leq (\bigsqcup x{\in}A.\ g\ x)$
by (*rule SUP-mono*) *auto*

lemma *INF-superset-mono*: $B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcap x{\in}A.\ f\ x) \leq (\bigsqcap x{\in}B.\ g\ x)$
— The last inclusion is POSITIVE!
by (*blast intro*: *INF-mono dest*: *subsetD*)

lemma *SUP-subset-mono*: $A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\bigsqcup x{\in}A.\ f\ x) \leq (\bigsqcup x{\in}B.\ g\ x)$
by (*blast intro*: *SUP-mono dest*: *subsetD*)

lemma *Inf-less-eq*:
assumes $\bigwedge v.\ v \in A \Longrightarrow v \leq u$
and $A \neq \{\}$
shows $\bigsqcap A \leq u$
proof −
from ‹$A \neq \{\}$› **obtain** *v* **where** $v \in A$ **by** *blast*
moreover from ‹$v \in A$› *assms*(*1*) **have** $v \leq u$ **by** *blast*
ultimately show *?thesis* **by** (*rule Inf-lower2*)
qed

lemma *less-eq-Sup*:
assumes $\bigwedge v.\ v \in A \Longrightarrow u \leq v$
and $A \neq \{\}$
shows $u \leq \bigsqcup A$

proof −
from ‹$A \neq \{\}$› **obtain** v **where** $v \in A$ **by** *blast*
moreover from ‹$v \in A$› *assms*(*1*) **have** $u \leq v$ **by** *blast*
ultimately show *?thesis* **by** (*rule Sup-upper2*)
qed

lemma *INF-eq*:
assumes $\bigwedge i.\ i \in A \Longrightarrow \exists j{\in}B.\ f\ i \geq g\ j$
and $\bigwedge j.\ j \in B \Longrightarrow \exists i{\in}A.\ g\ j \geq f\ i$
shows $\bigsqcap (f\ `\ A) = \bigsqcap (g\ `\ B)$
by (*intro order.antisym INF-greatest*) (*blast intro*: *INF-lower2 dest*: *assms*)+

lemma *SUP-eq*:
assumes $\bigwedge i.\ i \in A \Longrightarrow \exists j{\in}B.\ f\ i \leq g\ j$
and $\bigwedge j.\ j \in B \Longrightarrow \exists i{\in}A.\ g\ j \leq f\ i$
shows $\bigsqcup (f\ `\ A) = \bigsqcup (g\ `\ B)$
by (*intro order.antisym SUP-least*) (*blast intro*: *SUP-upper2 dest*: *assms*)+

lemma *less-eq-Inf-inter*: $\bigsqcap A \sqcup \bigsqcap B \leq \bigsqcap (A \cap B)$
by (*auto intro*: *Inf-greatest Inf-lower*)

lemma *Sup-inter-less-eq*: $\bigsqcup (A \cap B) \leq \bigsqcup A \sqcap \bigsqcup B$
by (*auto intro*: *Sup-least Sup-upper*)

lemma *Inf-union-distrib*: $\bigsqcap (A \cup B) = \bigsqcap A \sqcap \bigsqcap B$
by (*rule order.antisym*) (*auto intro*: *Inf-greatest Inf-lower le-infI1 le-infI2*)

lemma *INF-union*: $(\bigsqcap i \in A \cup B.\ M\ i) = (\bigsqcap i \in A.\ M\ i) \sqcap (\bigsqcap i{\in}B.\ M\ i)$
by (*auto intro*!: *order.antisym INF-mono intro*: *le-infI1 le-infI2 INF-greatest INF-lower*)

lemma *Sup-union-distrib*: $\bigsqcup (A \cup B) = \bigsqcup A \sqcup \bigsqcup B$
by (*rule order.antisym*) (*auto intro*: *Sup-least Sup-upper le-supI1 le-supI2*)

lemma *SUP-union*: $(\bigsqcup i \in A \cup B.\ M\ i) = (\bigsqcup i \in A.\ M\ i) \sqcup (\bigsqcup i{\in}B.\ M\ i)$
by (*auto intro*!: *order.antisym SUP-mono intro*: *le-supI1 le-supI2 SUP-least SUP-upper*)

lemma *INF-inf-distrib*: $(\bigsqcap a{\in}A.\ f\ a) \sqcap (\bigsqcap a{\in}A.\ g\ a) = (\bigsqcap a{\in}A.\ f\ a \sqcap g\ a)$
by (*rule order.antisym*) (*rule INF-greatest*, *auto intro*: *le-infI1 le-infI2 INF-lower INF-mono*)

lemma *SUP-sup-distrib*: $(\bigsqcup a{\in}A.\ f\ a) \sqcup (\bigsqcup a{\in}A.\ g\ a) = (\bigsqcup a{\in}A.\ f\ a \sqcup g\ a)$
(**is** *?L* = *?R*)
proof (*rule order.antisym*)
show *?L* ≤ *?R*
by (*auto intro*: *le-supI1 le-supI2 SUP-upper SUP-mono*)
show *?R* ≤ *?L*
by (*rule SUP-least*) (*auto intro*: *le-supI1 le-supI2 SUP-upper*)
qed

lemma *Inf-top-conv* [*simp*]:
$\bigsqcap A = \top \longleftrightarrow (\forall x{\in}A.\ x = \top)$
$\top = \bigsqcap A \longleftrightarrow (\forall x{\in}A.\ x = \top)$
proof −
show $\bigsqcap A = \top \longleftrightarrow (\forall x{\in}A.\ x = \top)$
proof
assume $\forall x{\in}A.\ x = \top$
then have $A = \{\} \vee A = \{\top\}$ **by** *auto*
then show $\bigsqcap A = \top$ **by** *auto*
next
assume $\bigsqcap A = \top$
show $\forall x{\in}A.\ x = \top$
proof (*rule ccontr*)
assume $\neg\ (\forall x{\in}A.\ x = \top)$
then obtain x **where** $x \in A$ **and** $x \neq \top$ **by** *blast*
then obtain B **where** $A = \mathit{insert}\ x\ B$ **by** *blast*
with $\langle\bigsqcap A = \top\rangle$ $\langle x \neq \top\rangle$ **show** *False* **by** *simp*
qed
qed
then show $\top = \bigsqcap A \longleftrightarrow (\forall x{\in}A.\ x = \top)$ **by** *auto*
qed

lemma *INF-top-conv* [*simp*]:
$(\bigsqcap x{\in}A.\ B\ x) = \top \longleftrightarrow (\forall x{\in}A.\ B\ x = \top)$
$\top = (\bigsqcap x{\in}A.\ B\ x) \longleftrightarrow (\forall x{\in}A.\ B\ x = \top)$
using *Inf-top-conv* [*of B ' A*] **by** *simp-all*

lemma *Sup-bot-conv* [*simp*]:
$\bigsqcup A = \bot \longleftrightarrow (\forall x{\in}A.\ x = \bot)$
$\bot = \bigsqcup A \longleftrightarrow (\forall x{\in}A.\ x = \bot)$
using *dual-complete-lattice*
by (*rule complete-lattice.Inf-top-conv*)+

lemma *SUP-bot-conv* [*simp*]:
$(\bigsqcup x{\in}A.\ B\ x) = \bot \longleftrightarrow (\forall x{\in}A.\ B\ x = \bot)$
$\bot = (\bigsqcup x{\in}A.\ B\ x) \longleftrightarrow (\forall x{\in}A.\ B\ x = \bot)$
using *Sup-bot-conv* [*of B ' A*] **by** *simp-all*

lemma *INF-constant*: $(\bigsqcap y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \top\ \mathit{else}\ c)$
by (*auto intro*: *order.antisym INF-lower INF-greatest*)

lemma *SUP-constant*: $(\bigsqcup y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \bot\ \mathit{else}\ c)$
by (*auto intro*: *order.antisym SUP-upper SUP-least*)

lemma *INF-const* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcap i{\in}A.\ f) = f$
by (*simp add*: *INF-constant*)

lemma *SUP-const* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcup i{\in}A.\ f) = f$

by (*simp add*: *SUP-constant*)

lemma *INF-top* [*simp*]: $(\bigsqcap x{\in}A.\ \top) = \top$
by (*cases* $A = \{\}$) *simp-all*

lemma *SUP-bot* [*simp*]: $(\bigsqcup x{\in}A.\ \bot) = \bot$
by (*cases* $A = \{\}$) *simp-all*

lemma *INF-commute*: $(\bigsqcap i{\in}A.\ \bigsqcap j{\in}B.\ f\ i\ j) = (\bigsqcap j{\in}B.\ \bigsqcap i{\in}A.\ f\ i\ j)$
by (*iprover intro*: *INF-lower INF-greatest order-trans order.antisym*)

lemma *SUP-commute*: $(\bigsqcup i{\in}A.\ \bigsqcup j{\in}B.\ f\ i\ j) = (\bigsqcup j{\in}B.\ \bigsqcup i{\in}A.\ f\ i\ j)$
by (*iprover intro*: *SUP-upper SUP-least order-trans order.antisym*)

lemma *INF-absorb*:
assumes $k \in I$
shows $A\ k \sqcap (\bigsqcap i{\in}I.\ A\ i) = (\bigsqcap i{\in}I.\ A\ i)$
proof −
from *assms* **obtain** J **where** $I = \mathit{insert}\ k\ J$ **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *SUP-absorb*:
assumes $k \in I$
shows $A\ k \sqcup (\bigsqcup i{\in}I.\ A\ i) = (\bigsqcup i{\in}I.\ A\ i)$
proof −
from *assms* **obtain** J **where** $I = \mathit{insert}\ k\ J$ **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *INF-inf-const1*: $I \neq \{\} \Longrightarrow (\bigsqcap i{\in}I.\ \mathit{inf}\ x\ (f\ i)) = \mathit{inf}\ x\ (\bigsqcap i{\in}I.\ f\ i)$
by (*intro order.antisym INF-greatest inf-mono order-refl INF-lower*)
(*auto intro*: *INF-lower2 le-infI2 intro*!: *INF-mono*)

lemma *INF-inf-const2*: $I \neq \{\} \Longrightarrow (\bigsqcap i{\in}I.\ \mathit{inf}\ (f\ i)\ x) = \mathit{inf}\ (\bigsqcap i{\in}I.\ f\ i)\ x$
using *INF-inf-const1*[*of I x f*] **by** (*simp add*: *inf-commute*)

lemma *less-INF-D*:
assumes $y < (\bigsqcap i{\in}A.\ f\ i)\ i \in A$
shows $y < f\ i$
proof −
note ‹$y < (\bigsqcap i{\in}A.\ f\ i)$›
also have $(\bigsqcap i{\in}A.\ f\ i) \leq f\ i$ **using** ‹$i \in A$›
by (*rule INF-lower*)
finally show $y < f\ i$ **.**
qed

lemma *SUP-lessD*:
assumes $(\bigsqcup i{\in}A.\ f\ i) < y\ i \in A$

shows $f\ i < y$
proof −
have $f\ i \leq (\bigsqcup i{\in}A.\ f\ i)$
using ‹$i \in A$› **by** (*rule SUP-upper*)
also note ‹$(\bigsqcup i{\in}A.\ f\ i) < y$›
finally show $f\ i < y$ **.**
qed

lemma *INF-UNIV-bool-expand*: $(\bigsqcap b.\ A\ b) = A\ True \sqcap A\ False$
by (*simp add*: *UNIV-bool inf-commute*)

lemma *SUP-UNIV-bool-expand*: $(\bigsqcup b.\ A\ b) = A\ True \sqcup A\ False$
by (*simp add*: *UNIV-bool sup-commute*)

lemma *Inf-le-Sup*: $A \neq \{\} \Longrightarrow Inf\ A \leq Sup\ A$
by (*blast intro*: *Sup-upper2 Inf-lower ex-in-conv*)

lemma *INF-le-SUP*: $A \neq \{\} \Longrightarrow \bigsqcap(f\ `\ A) \leq \bigsqcup(f\ `\ A)$
using *Inf-le-Sup* [*of f* ` *A*] **by** *simp*

lemma *INF-eq-const*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i = x) \Longrightarrow \bigsqcap(f\ `\ I) = x$
by (*auto intro*: *INF-eqI*)

lemma *SUP-eq-const*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i = x) \Longrightarrow \bigsqcup(f\ `\ I) = x$
by (*auto intro*: *SUP-eqI*)

lemma *INF-eq-iff*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i \leq c) \Longrightarrow \bigsqcap(f\ `\ I) = c \longleftrightarrow (\forall\, i{\in}I.\ f\ i = c)$
by (*auto intro*: *INF-eq-const INF-lower order.antisym*)

lemma *SUP-eq-iff*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow c \leq f\ i) \Longrightarrow \bigsqcup(f\ `\ I) = c \longleftrightarrow (\forall\, i{\in}I.\ f\ i = c)$
by (*auto intro*: *SUP-eq-const SUP-upper order.antisym*)

end

context *complete-lattice*
begin
lemma *Sup-Inf-le*: $Sup\ (Inf\ `\ \{f\ `\ A \mid f.\ (\forall\ Y \in A.\ f\ Y \in Y)\}) \leq Inf\ (Sup\ `\ A)$
by (*rule SUP-least*, *clarify*, *rule INF-greatest*, *simp add*: *INF-lower2 Sup-upper*)
end

class *complete-distrib-lattice* = *complete-lattice* +
assumes *Inf-Sup-le*: $Inf\ (Sup\ `\ A) \leq Sup\ (Inf\ `\ \{f\ `\ A \mid f.\ (\forall\ Y \in A.\ f\ Y \in Y)\})$
begin

lemma *Inf-Sup*: $Inf\ (Sup\ `\ A) = Sup\ (Inf\ `\ \{f\ `\ A \mid f.\ (\forall\ Y \in A.\ f\ Y \in Y)\})$
by (*rule order.antisym*, *rule Inf-Sup-le*, *rule Sup-Inf-le*)

subclass *distrib-lattice*
proof
  fix a b c
  show $a \sqcup b \sqcap c = (a \sqcup b) \sqcap (a \sqcup c)$
  proof (*rule order.antisym*, *simp-all*, *safe*)
    show $b \sqcap c \leq a \sqcup b$
      by (*rule le-infI1*, *simp*)
    show $b \sqcap c \leq a \sqcup c$
      by (*rule le-infI2*, *simp*)
    have [*simp*]: $a \sqcap c \leq a \sqcup b \sqcap c$
      by (*rule le-infI1*, *simp*)
    have [*simp*]: $b \sqcap a \leq a \sqcup b \sqcap c$
      by (*rule le-infI2*, *simp*)
    have $\bigsqcap (Sup \text{ ` } \{\{a, b\}, \{a, c\}\}) =$
      $\bigsqcup (Inf \text{ ` } \{f \text{ ` } \{\{a, b\}, \{a, c\}\} \mid f.\ \forall Y{\in}\{\{a, b\}, \{a, c\}\}.\ f\ Y \in Y\})$
      by (*rule Inf-Sup*)
    from *this* **show** $(a \sqcup b) \sqcap (a \sqcup c) \leq a \sqcup b \sqcap c$
      apply *simp*
      by (*rule SUP-least*, *safe*, *simp-all*)
  qed
qed
end

context *complete-lattice*
begin
context
  fixes $f :: 'a \Rightarrow 'b$::*complete-lattice*
  assumes *mono f*
begin

lemma *mono-Inf*: $f\ (\bigsqcap A) \leq (\bigsqcap x{\in}A.\ f\ x)$
  using ‹*mono f*› **by** (*auto intro*: *complete-lattice-class.INF-greatest Inf-lower dest*: *monoD*)

lemma *mono-Sup*: $(\bigsqcup x{\in}A.\ f\ x) \leq f\ (\bigsqcup A)$
  using ‹*mono f*› **by** (*auto intro*: *complete-lattice-class.SUP-least Sup-upper dest*: *monoD*)

lemma *mono-INF*: $f\ (\bigsqcap i{\in}I.\ A\ i) \leq (\bigsqcap x{\in}I.\ f\ (A\ x))$
  by (*intro complete-lattice-class.INF-greatest monoD*[*OF* ‹*mono f*›] *INF-lower*)

lemma *mono-SUP*: $(\bigsqcup x{\in}I.\ f\ (A\ x)) \leq f\ (\bigsqcup i{\in}I.\ A\ i)$
  by (*intro complete-lattice-class.SUP-least monoD*[*OF* ‹*mono f*›] *SUP-upper*)

end

end

class *complete-boolean-algebra* = *boolean-algebra* + *complete-distrib-lattice*
begin

lemma *uminus-Inf*: $-(\bigsqcap A) = \bigsqcup(\mathit{uminus}\ `\ A)$
proof (*rule order.antisym*)
show $-\bigsqcap A \leq \bigsqcup(\mathit{uminus}\ `\ A)$
by (*rule compl-le-swap2*, *rule Inf-greatest*, *rule compl-le-swap2*, *rule Sup-upper*)
simp
show $\bigsqcup(\mathit{uminus}\ `\ A) \leq -\bigsqcap A$
by (*rule Sup-least*, *rule compl-le-swap1*, *rule Inf-lower*) *auto*
qed

lemma *uminus-INF*: $-(\bigsqcap x{\in}A.\ B\ x) = (\bigsqcup x{\in}A.\ -\ B\ x)$
by (*simp add*: *uminus-Inf image-image*)

lemma *uminus-Sup*: $-(\bigsqcup A) = \bigsqcap(\mathit{uminus}\ `\ A)$
proof −
have $\bigsqcup A = -\bigsqcap(\mathit{uminus}\ `\ A)$
by (*simp add*: *image-image uminus-INF*)
then show *?thesis* **by** *simp*
qed

lemma *uminus-SUP*: $-(\bigsqcup x{\in}A.\ B\ x) = (\bigsqcap x{\in}A.\ -\ B\ x)$
by (*simp add*: *uminus-Sup image-image*)

end

class *complete-linorder* = *linorder* + *complete-lattice*
begin

lemma *dual-complete-linorder*:
class.complete-linorder Sup Inf sup (≥) (>) *inf* ⊤ ⊥
by (*rule class.complete-linorder.intro*, *rule dual-complete-lattice*, *rule dual-linorder*)

lemma *complete-linorder-inf-min*: *inf* = *min*
by (*auto intro*: *order.antisym simp add*: *min-def fun-eq-iff*)

lemma *complete-linorder-sup-max*: *sup* = *max*
by (*auto intro*: *order.antisym simp add*: *max-def fun-eq-iff*)

lemma *Inf-less-iff*: $\bigsqcap S < a \longleftrightarrow (\exists\, x{\in}S.\ x < a)$
by (*simp add*: *not-le* [*symmetric*] *le-Inf-iff*)

lemma *INF-less-iff*: $(\bigsqcap i{\in}A.\ f\ i) < a \longleftrightarrow (\exists\, x{\in}A.\ f\ x < a)$
by (*simp add*: *Inf-less-iff* [*of f* ` *A*])

lemma *less-Sup-iff*: $a < \bigsqcup S \longleftrightarrow (\exists\, x{\in}S.\ a < x)$
by (*simp add*: *not-le* [*symmetric*] *Sup-le-iff*)

lemma *less-SUP-iff*: $a < (\bigsqcup i{\in}A.\ f\ i) \longleftrightarrow (\exists\, x{\in}A.\ a < f\ x)$
by (*simp add*: *less-Sup-iff* [*of* - *f* ' *A*])

lemma *Sup-eq-top-iff* [*simp*]: $\bigsqcup A = \top \longleftrightarrow (\forall\, x{<}\top.\ \exists\, i{\in}A.\ x < i)$
proof
assume $*$: $\bigsqcup A = \top$
show $(\forall\, x{<}\top.\ \exists\, i{\in}A.\ x < i)$
unfolding $*$ [*symmetric*]
proof (*intro allI impI*)
fix x
assume $x < \bigsqcup A$
then show $\exists\, i{\in}A.\ x < i$
by (*simp add*: *less-Sup-iff*)
qed
next
assume $*$: $\forall\, x{<}\top.\ \exists\, i{\in}A.\ x < i$
show $\bigsqcup A = \top$
proof (*rule ccontr*)
assume $\bigsqcup A \neq \top$
with *top-greatest* [*of* $\bigsqcup A$] **have** $\bigsqcup A < \top$
unfolding *le-less* **by** *auto*
with $*$ **have** $\bigsqcup A < \bigsqcup A$
unfolding *less-Sup-iff* **by** *auto*
then show *False* **by** *auto*
qed
qed

lemma *SUP-eq-top-iff* [*simp*]: $(\bigsqcup i{\in}A.\ f\ i) = \top \longleftrightarrow (\forall\, x{<}\top.\ \exists\, i{\in}A.\ x < f\ i)$
using *Sup-eq-top-iff* [*of f* ' *A*] **by** *simp*

lemma *Inf-eq-bot-iff* [*simp*]: $\bigsqcap A = \bot \longleftrightarrow (\forall\, x{>}\bot.\ \exists\, i{\in}A.\ i < x)$
using *dual-complete-linorder*
by (*rule complete-linorder.Sup-eq-top-iff*)

lemma *INF-eq-bot-iff* [*simp*]: $(\bigsqcap i{\in}A.\ f\ i) = \bot \longleftrightarrow (\forall\, x{>}\bot.\ \exists\, i{\in}A.\ f\ i < x)$
using *Inf-eq-bot-iff* [*of f* ' *A*] **by** *simp*

lemma *Inf-le-iff*: $\bigsqcap A \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, a{\in}A.\ y > a)$
proof *safe*
fix y
assume $x \geq \bigsqcap A\ y > x$
then have $y > \bigsqcap A$ **by** *auto*
then show $\exists\, a{\in}A.\ y > a$
unfolding *Inf-less-iff* **.**
qed (*auto elim*!: *allE*[*of* - $\bigsqcap A$] *simp add*: *not-le*[*symmetric*] *Inf-lower*)

lemma *INF-le-iff*: $\bigsqcap (f\ `\ A) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ y > f\ i)$
using *Inf-le-iff* [*of f* ' *A*] **by** *simp*

lemma *le-Sup-iff*: $x \leq \bigsqcup A \longleftrightarrow (\forall\, y{<}x.\ \exists\, a{\in}A.\ y < a)$
proof *safe*
fix y
assume $x \leq \bigsqcup A$ $y < x$
then have $y < \bigsqcup A$ **by** *auto*
then show $\exists\, a{\in}A.\ y < a$
unfolding *less-Sup-iff* **.**
qed (*auto elim*!: *allE*[*of* - $\bigsqcup A$] *simp add*: *not-le*[*symmetric*] *Sup-upper*)

lemma *le-SUP-iff*: $x \leq \bigsqcup (f\ `\ A) \longleftrightarrow (\forall\, y{<}x.\ \exists\, i{\in}A.\ y < f\ i)$
using *le-Sup-iff* [*of* - $f\ `\ A$] **by** *simp*

end

11.3 Complete lattice on *bool*

instantiation *bool* :: *complete-lattice*
begin

definition [*simp*, *code*]: $\bigsqcap A \longleftrightarrow \mathit{False} \notin A$

definition [*simp*, *code*]: $\bigsqcup A \longleftrightarrow \mathit{True} \in A$

instance
by *standard* (*auto intro*: *bool-induct*)

end

lemma *not-False-in-image-Ball* [*simp*]: $\mathit{False} \notin P\ `\ A \longleftrightarrow \mathit{Ball}\ A\ P$
by *auto*

lemma *True-in-image-Bex* [*simp*]: $\mathit{True} \in P\ `\ A \longleftrightarrow \mathit{Bex}\ A\ P$
by *auto*

lemma *INF-bool-eq* [*simp*]: $(\lambda A\ f.\ \bigsqcap (f\ `\ A)) = \mathit{Ball}$
by (*simp add*: *fun-eq-iff*)

lemma *SUP-bool-eq* [*simp*]: $(\lambda A\ f.\ \bigsqcup (f\ `\ A)) = \mathit{Bex}$
by (*simp add*: *fun-eq-iff*)

instance *bool* :: *complete-boolean-algebra*
by (*standard*, *fastforce*)

11.4 Complete lattice on - ⇒ -

instantiation *fun* :: (*type*, *Inf*) *Inf*
begin

definition $\bigsqcap A = (\lambda x.\ \bigsqcap f{\in}A.\ f\ x)$

lemma *Inf-apply* [*simp*, *code*]: $(\bigsqcap A)\ x = (\bigsqcap f{\in}A.\ f\ x)$
by (*simp add*: *Inf-fun-def*)

instance ..

end

instantiation *fun* :: (*type*, *Sup*) *Sup*
begin

definition $\bigsqcup A = (\lambda x.\ \bigsqcup f{\in}A.\ f\ x)$

lemma *Sup-apply* [*simp*, *code*]: $(\bigsqcup A)\ x = (\bigsqcup f{\in}A.\ f\ x)$
by (*simp add*: *Sup-fun-def*)

instance ..

end

instantiation *fun* :: (*type*, *complete-lattice*) *complete-lattice*
begin

instance
by *standard* (*auto simp add*: *le-fun-def intro*: *INF-lower INF-greatest SUP-upper SUP-least*)

end

lemma *INF-apply* [*simp*]: $(\bigsqcap y{\in}A.\ f\ y)\ x = (\bigsqcap y{\in}A.\ f\ y\ x)$
by (*simp add*: *image-comp*)

lemma *SUP-apply* [*simp*]: $(\bigsqcup y{\in}A.\ f\ y)\ x = (\bigsqcup y{\in}A.\ f\ y\ x)$
by (*simp add*: *image-comp*)

11.5 Complete lattice on unary and binary predicates

lemma *Inf1-I*: $(\bigwedge P.\ P \in A \Longrightarrow P\ a) \Longrightarrow (\bigsqcap A)\ a$
by *auto*

lemma *INF1-I*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x\ b) \Longrightarrow (\bigsqcap x{\in}A.\ B\ x)\ b$
by *simp*

lemma *INF2-I*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x\ b\ c) \Longrightarrow (\bigsqcap x{\in}A.\ B\ x)\ b\ c$
by *simp*

lemma *Inf2-I*: $(\bigwedge r.\ r \in A \Longrightarrow r\ a\ b) \Longrightarrow (\bigsqcap A)\ a\ b$
by *auto*

lemma *Inf1-D*: $(\bigsqcap A)\ a \Longrightarrow P \in A \Longrightarrow P\ a$

by *auto*

lemma *INF1-D*: $(\bigsqcap x{\in}A.\ B\ x)\ b \Longrightarrow a \in A \Longrightarrow B\ a\ b$
by *simp*

lemma *Inf2-D*: $(\bigsqcap A)\ a\ b \Longrightarrow r \in A \Longrightarrow r\ a\ b$
by *auto*

lemma *INF2-D*: $(\bigsqcap x{\in}A.\ B\ x)\ b\ c \Longrightarrow a \in A \Longrightarrow B\ a\ b\ c$
by *simp*

lemma *Inf1-E*:
assumes $(\bigsqcap A)\ a$
obtains $P\ a \mid P \notin A$
using *assms* **by** *auto*

lemma *INF1-E*:
assumes $(\bigsqcap x{\in}A.\ B\ x)\ b$
obtains $B\ a\ b \mid a \notin A$
using *assms* **by** *auto*

lemma *Inf2-E*:
assumes $(\bigsqcap A)\ a\ b$
obtains $r\ a\ b \mid r \notin A$
using *assms* **by** *auto*

lemma *INF2-E*:
assumes $(\bigsqcap x{\in}A.\ B\ x)\ b\ c$
obtains $B\ a\ b\ c \mid a \notin A$
using *assms* **by** *auto*

lemma *Sup1-I*: $P \in A \Longrightarrow P\ a \Longrightarrow (\bigsqcup A)\ a$
by *auto*

lemma *SUP1-I*: $a \in A \Longrightarrow B\ a\ b \Longrightarrow (\bigsqcup x{\in}A.\ B\ x)\ b$
by *auto*

lemma *Sup2-I*: $r \in A \Longrightarrow r\ a\ b \Longrightarrow (\bigsqcup A)\ a\ b$
by *auto*

lemma *SUP2-I*: $a \in A \Longrightarrow B\ a\ b\ c \Longrightarrow (\bigsqcup x{\in}A.\ B\ x)\ b\ c$
by *auto*

lemma *Sup1-E*:
assumes $(\bigsqcup A)\ a$
obtains P **where** $P \in A$ **and** $P\ a$
using *assms* **by** *auto*

lemma *SUP1-E*:

assumes $(\bigsqcup x{\in}A.\ B\ x)\ b$
obtains x **where** $x \in A$ **and** $B\ x\ b$
using *assms* **by** *auto*

lemma *Sup2-E*:
assumes $(\bigsqcup A)\ a\ b$
obtains r **where** $r \in A\ r\ a\ b$
using *assms* **by** *auto*

lemma *SUP2-E*:
assumes $(\bigsqcup x{\in}A.\ B\ x)\ b\ c$
obtains x **where** $x \in A\ B\ x\ b\ c$
using *assms* **by** *auto*

11.6 Complete lattice on - *set*

instantiation *set* :: (*type*) *complete-lattice*
begin

definition $\bigsqcap A = \{x.\ \bigsqcap((\lambda B.\ x \in B)\ `\ A)\}$

definition $\bigsqcup A = \{x.\ \bigsqcup((\lambda B.\ x \in B)\ `\ A)\}$

instance
by *standard* (*auto simp add*: *less-eq-set-def Inf-set-def Sup-set-def le-fun-def*)

end

11.6.1 Inter

abbreviation *Inter* :: *'a set set* ⇒ *'a set* (‹$\bigcap$›)
where $\bigcap S \equiv \bigsqcap S$

lemma *Inter-eq*: $\bigcap A = \{x.\ \forall B \in A.\ x \in B\}$
proof (*rule set-eqI*)
fix x
have $(\forall Q{\in}\{P.\ \exists B{\in}A.\ P \longleftrightarrow x \in B\}.\ Q) \longleftrightarrow (\forall B{\in}A.\ x \in B)$
by *auto*
then show $x \in \bigcap A \longleftrightarrow x \in \{x.\ \forall B \in A.\ x \in B\}$
by (*simp add*: *Inf-set-def image-def*)
qed

lemma *Inter-iff* [*simp*]: $A \in \bigcap C \longleftrightarrow (\forall X{\in}C.\ A \in X)$
by (*unfold Inter-eq*) *blast*

lemma *InterI* [*intro*!]: $(\bigwedge X.\ X \in C \Longrightarrow A \in X) \Longrightarrow A \in \bigcap C$
by (*simp add*: *Inter-eq*)

A "destruct" rule – every X in C contains A as an element, but $A \in X$ can hold when $X \in C$ does not! This rule is analogous to *spec*.

lemma *InterD* [*elim*, *Pure.elim*]: $A \in \bigcap C \Longrightarrow X \in C \Longrightarrow A \in X$
by *auto*

lemma *InterE* [*elim*]: $A \in \bigcap C \Longrightarrow (X \notin C \Longrightarrow R) \Longrightarrow (A \in X \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination rule – does not require proving $X \in C$.
unfolding *Inter-eq* **by** *blast*

lemma *Inter-lower*: $B \in A \Longrightarrow \bigcap A \subseteq B$
by (*fact Inf-lower*)

lemma *Inter-subset*: $(\bigwedge X.\ X \in A \Longrightarrow X \subseteq B) \Longrightarrow A \neq \{\} \Longrightarrow \bigcap A \subseteq B$
by (*fact Inf-less-eq*)

lemma *Inter-greatest*: $(\bigwedge X.\ X \in A \Longrightarrow C \subseteq X) \Longrightarrow C \subseteq \bigcap A$
by (*fact Inf-greatest*)

lemma *Inter-empty*: $\bigcap \{\} = UNIV$
by (*fact Inf-empty*)

lemma *Inter-UNIV*: $\bigcap UNIV = \{\}$
by (*fact Inf-UNIV*)

lemma *Inter-insert*: $\bigcap (insert\ a\ B) = a \cap \bigcap B$
by (*fact Inf-insert*)

lemma *Inter-Un-subset*: $\bigcap A \cup \bigcap B \subseteq \bigcap (A \cap B)$
by (*fact less-eq-Inf-inter*)

lemma *Inter-Un-distrib*: $\bigcap (A \cup B) = \bigcap A \cap \bigcap B$
by (*fact Inf-union-distrib*)

lemma *Inter-UNIV-conv* [*simp*]:
$\bigcap A = UNIV \longleftrightarrow (\forall x{\in}A.\ x = UNIV)$
$UNIV = \bigcap A \longleftrightarrow (\forall x{\in}A.\ x = UNIV)$
by (*fact Inf-top-conv*)+

lemma *Inter-anti-mono*: $B \subseteq A \Longrightarrow \bigcap A \subseteq \bigcap B$
by (*fact Inf-superset-mono*)

11.6.2 Intersections of families

syntax (*ASCII*)
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* (‹(‹*indent=3 notation=‹binder INT››INT -./ -)› [0, 10] 10*)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* (‹(‹*indent=3 notation=‹binder INT››INT -:-./ -)› [0, 0, 10] 10*)

syntax
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* (‹(‹*indent=3 notation=‹binder*

$\bigcap$››$\bigcap$ -./ -)› [0, 10] 10)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* (‹(‹*indent=3 notation=‹binder* $\bigcap$››$\bigcap$ -∈-./ -)› [0, 0, 10] 10)

syntax (*latex* **output**)
-INTER1 :: *pttrns* ⇒ *'b set* ⇒ *'b set* (‹(3$\bigcap$(‹*unbreakable*›-)/ -)› [0, 10] 10)
-INTER :: *pttrn* ⇒ *'a set* ⇒ *'b set* ⇒ *'b set* (‹(3$\bigcap$(‹*unbreakable*›-∈-)/ -)› [0, 0, 10] 10)

syntax-consts
-INTER1 -INTER ⇌ *Inter*

translations
$\bigcap x\ y.\ f$ ⇌ $\bigcap x.\ \bigcap y.\ f$
$\bigcap x.\ f$ ⇌ $\bigcap$(*CONST range* ($\lambda x.\ f$))
$\bigcap x{\in}A.\ f$ ⇌ *CONST Inter* (($\lambda x.\ f$) ‘ *A*)

lemma *INTER-eq*: ($\bigcap x{\in}A.\ B\ x$) = {*y*. $\forall x{\in}A.\ y \in B\ x$}
by (*auto intro*!: *INF-eqI*)

lemma *INT-iff* [*simp*]: $b \in (\bigcap x{\in}A.\ B\ x) \longleftrightarrow (\forall x{\in}A.\ b \in B\ x)$
using *Inter-iff* [*of - B ‘ A*] **by** *simp*

lemma *INT-I* [*intro*!]: ($\bigwedge x.\ x \in A \Longrightarrow b \in B\ x$) ⟹ $b \in (\bigcap x{\in}A.\ B\ x)$
by *auto*

lemma *INT-D* [*elim*, *Pure.elim*]: $b \in (\bigcap x{\in}A.\ B\ x) \Longrightarrow a \in A \Longrightarrow b \in B\ a$
by *auto*

lemma *INT-E* [*elim*]: $b \in (\bigcap x{\in}A.\ B\ x) \Longrightarrow (b \in B\ a \Longrightarrow R) \Longrightarrow (a \notin A \Longrightarrow R) \Longrightarrow R$
— "Classical" elimination – by the Excluded Middle on $a \in A$.
by *auto*

lemma *Collect-ball-eq*: {*x*. $\forall y{\in}A.\ P\ x\ y$} = ($\bigcap y{\in}A.$ {*x*. *P x y*})
by *blast*

lemma *Collect-all-eq*: {*x*. $\forall y.\ P\ x\ y$} = ($\bigcap y.$ {*x*. *P x y*})
by *blast*

lemma *INT-lower*: $a \in A \Longrightarrow (\bigcap x{\in}A.\ B\ x) \subseteq B\ a$
by (*fact INF-lower*)

lemma *INT-greatest*: ($\bigwedge x.\ x \in A \Longrightarrow C \subseteq B\ x$) ⟹ $C \subseteq (\bigcap x{\in}A.\ B\ x)$
by (*fact INF-greatest*)

lemma *INT-empty*: ($\bigcap x{\in}\{\}.\ B\ x$) = *UNIV*
by (*fact INF-empty*)

lemma *INT-absorb*: $k \in I \Longrightarrow A\ k \cap (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ A\ i)$
by (*fact INF-absorb*)

lemma *INT-subset-iff*: $B \subseteq (\bigcap i{\in}I.\ A\ i) \longleftrightarrow (\forall\, i{\in}I.\ B \subseteq A\ i)$
by (*fact le-INF-iff*)

lemma *INT-insert* [*simp*]: $(\bigcap x \in \mathit{insert}\ a\ A.\ B\ x) = B\ a \cap \bigcap\ (B\ `\ A)$
by (*fact INF-insert*)

lemma *INT-Un*: $(\bigcap i \in A \cup B.\ M\ i) = (\bigcap i \in A.\ M\ i) \cap (\bigcap i{\in}B.\ M\ i)$
by (*fact INF-union*)

lemma *INT-insert-distrib*: $u \in A \Longrightarrow (\bigcap x{\in}A.\ \mathit{insert}\ a\ (B\ x)) = \mathit{insert}\ a\ (\bigcap x{\in}A.\ B\ x)$
by *blast*

lemma *INT-constant* [*simp*]: $(\bigcap y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ c)$
by (*fact INF-constant*)

lemma *INTER-UNIV-conv*:
$(\mathit{UNIV} = (\bigcap x{\in}A.\ B\ x)) = (\forall\, x{\in}A.\ B\ x = \mathit{UNIV})$
$((\bigcap x{\in}A.\ B\ x) = \mathit{UNIV}) = (\forall\, x{\in}A.\ B\ x = \mathit{UNIV})$
by (*fact INF-top-conv*)+

lemma *INT-bool-eq*: $(\bigcap b.\ A\ b) = A\ \mathit{True} \cap A\ \mathit{False}$
by (*fact INF-UNIV-bool-expand*)

lemma *INT-anti-mono*: $A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow (\bigcap x{\in}B.\ f\ x) \subseteq (\bigcap x{\in}A.\ g\ x)$
— The last inclusion is POSITIVE!
by (*fact INF-superset-mono*)

lemma *Pow-INT-eq*: $\mathit{Pow}\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ \mathit{Pow}\ (B\ x))$
by *blast*

lemma *vimage-INT*: $f\ -`\ (\bigcap x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ f\ -`\ B\ x)$
by *blast*

11.6.3 Union

abbreviation *Union* :: *'a set set* ⇒ *'a set* $(\langle\bigcup\rangle)$
where $\bigcup S \equiv \bigsqcup S$

lemma *Union-eq*: $\bigcup A = \{x.\ \exists\, B \in A.\ x \in B\}$
proof (*rule set-eqI*)
fix *x*
have $(\exists\, Q{\in}\{P.\ \exists\, B{\in}A.\ P \longleftrightarrow x \in B\}.\ Q) \longleftrightarrow (\exists\, B{\in}A.\ x \in B)$
by *auto*

then show $x \in \bigcup A \longleftrightarrow x \in \{x.\ \exists B{\in}A.\ x \in B\}$
by (*simp add*: *Sup-set-def image-def*)
qed

lemma *Union-iff* [*simp*]: $A \in \bigcup C \longleftrightarrow (\exists X{\in}C.\ A{\in}X)$
by (*unfold Union-eq*) *blast*

lemma *UnionI* [*intro*]: $X \in C \Longrightarrow A \in X \Longrightarrow A \in \bigcup C$
— The order of the premises presupposes that C is rigid; A may be flexible.
by *auto*

lemma *UnionE* [*elim*!]: $A \in \bigcup C \Longrightarrow (\bigwedge X.\ A \in X \Longrightarrow X \in C \Longrightarrow R) \Longrightarrow R$
by *auto*

lemma *Union-upper*: $B \in A \Longrightarrow B \subseteq \bigcup A$
by (*fact Sup-upper*)

lemma *Union-least*: $(\bigwedge X.\ X \in A \Longrightarrow X \subseteq C) \Longrightarrow \bigcup A \subseteq C$
by (*fact Sup-least*)

lemma *Union-empty*: $\bigcup \{\} = \{\}$
by (*fact Sup-empty*)

lemma *Union-UNIV*: $\bigcup UNIV = UNIV$
by (*fact Sup-UNIV*)

lemma *Union-insert*: $\bigcup (insert\ a\ B) = a \cup \bigcup B$
by (*fact Sup-insert*)

lemma *Union-Un-distrib* [*simp*]: $\bigcup (A \cup B) = \bigcup A \cup \bigcup B$
by (*fact Sup-union-distrib*)

lemma *Union-Int-subset*: $\bigcup (A \cap B) \subseteq \bigcup A \cap \bigcup B$
by (*fact Sup-inter-less-eq*)

lemma *Union-empty-conv*: $(\bigcup A = \{\}) \longleftrightarrow (\forall x{\in}A.\ x = \{\})$
by (*fact Sup-bot-conv*)

lemma *empty-Union-conv*: $(\{\} = \bigcup A) \longleftrightarrow (\forall x{\in}A.\ x = \{\})$
by (*fact Sup-bot-conv*)

lemma *subset-Pow-Union*: $A \subseteq Pow\ (\bigcup A)$
by *blast*

lemma *Union-Pow-eq* [*simp*]: $\bigcup (Pow\ A) = A$
by *blast*

lemma *Union-mono*: $A \subseteq B \Longrightarrow \bigcup A \subseteq \bigcup B$
by (*fact Sup-subset-mono*)

lemma *Union-subsetI*: $(\bigwedge x.\ x \in A \Longrightarrow \exists\, y.\ y \in B \wedge x \subseteq y) \Longrightarrow \bigcup A \subseteq \bigcup B$
by *blast*

lemma *disjnt-inj-on-iff*:
$[\![$*inj-on f* $(\bigcup \mathcal{A})$; $X \in \mathcal{A}$; $Y \in \mathcal{A}]\!] \Longrightarrow$ *disjnt* (*f* ' *X*) (*f* ' *Y*) $\longleftrightarrow$ *disjnt X Y*
unfolding *disjnt-def*
by *safe* (*use inj-on-eq-iff* **in** ‹*fastforce*+›)

lemma *disjnt-Union1* [*simp*]: *disjnt* $(\bigcup \mathcal{A})$ *B* $\longleftrightarrow$ $(\forall\, A \in \mathcal{A}.$ *disjnt A B*)
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Union2* [*simp*]: *disjnt B* $(\bigcup \mathcal{A})$ $\longleftrightarrow$ $(\forall\, A \in \mathcal{A}.$ *disjnt B A*)
by (*auto simp*: *disjnt-def*)

11.6.4 Unions of families

syntax (*ASCII*)
-UNION1 :: *pttrns* => *'b set* => *'b set* (‹(‹*indent=3 notation=‹binder UN›› UN -./ -)› [0, 10] 10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* (‹(‹*indent=3 notation=‹binder UN›› UN -:-./ -)› [0, 0, 10] 10*)

syntax
-UNION1 :: *pttrns* => *'b set* => *'b set* (‹(‹*indent=3 notation=‹binder* $\bigcup$*››*$\bigcup$*-./ -)› [0, 10] 10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* (‹(‹*indent=3 notation=‹binder* $\bigcup$*››*$\bigcup$*-∈-./ -)› [0, 0, 10] 10*)

syntax (*latex* **output**)
-UNION1 :: *pttrns* => *'b set* => *'b set* (‹(*3*$\bigcup$(‹*unbreakable*›*-*)*/ -)› [0, 10] 10*)
-UNION :: *pttrn* => *'a set* => *'b set* => *'b set* (‹(*3*$\bigcup$(‹*unbreakable*›*-∈-*)*/ -)› [0, 0, 10] 10*)

syntax-consts
-UNION1 -UNION $\rightleftharpoons$ *Union*

translations
$\bigcup x\ y.\ f \rightleftharpoons \bigcup x.\ \bigcup y.\ f$
$\bigcup x.\ f \rightleftharpoons \bigcup$(*CONST range* $(\lambda x.\ f)$)
$\bigcup x \in A.\ f \rightleftharpoons$ *CONST Union* $((\lambda x.\ f)$ ' *A*)

Note the difference between ordinary syntax of indexed unions and intersections (e.g. $\bigcup a_1 \in A_1.\ B$) and their LaTeX rendition: $\bigcup_{a_1 \in A_1} B$.

lemma *disjoint-UN-iff*: *disjnt A* $(\bigcup i \in I.\ B\ i) \longleftrightarrow (\forall\, i \in I.$ *disjnt A* $(B\ i))$
by (*auto simp*: *disjnt-def*)

lemma *UNION-eq*: $(\bigcup x \in A.\ B\ x) = \{y.\ \exists\, x \in A.\ y \in B\ x\}$

by (*auto intro*!: *SUP-eqI*)

lemma *bind-UNION* [*code*]: *Set.bind A f* = $\bigcup$(*f* ' *A*)
by (*simp add*: *bind-def UNION-eq*)

lemma *member-bind* [*simp*]: *x* ∈ *Set.bind A f* ⟷ *x* ∈ $\bigcup$(*f* ' *A*)
by (*simp add*: *bind-UNION*)

lemma *Union-SetCompr-eq*: $\bigcup${*f x*| *x*. *P x*} = {*a*. ∃ *x*. *P x* ∧ *a* ∈ *f x*}
by *blast*

lemma *UN-iff* [*simp*]: *b* ∈ ($\bigcup$*x*∈*A*. *B x*) ⟷ (∃ *x*∈*A*. *b* ∈ *B x*)
using *Union-iff* [*of* - *B* ' *A*] **by** *simp*

lemma *UN-I* [*intro*]: *a* ∈ *A* ⟹ *b* ∈ *B a* ⟹ *b* ∈ ($\bigcup$*x*∈*A*. *B x*)
— The order of the premises presupposes that *A* is rigid; *b* may be flexible.
by *auto*

lemma *UN-E* [*elim*!]: *b* ∈ ($\bigcup$*x*∈*A*. *B x*) ⟹ (⋀*x*. *x*∈*A* ⟹ *b* ∈ *B x* ⟹ *R*) ⟹ *R*
by *auto*

lemma *UN-upper*: *a* ∈ *A* ⟹ *B a* ⊆ ($\bigcup$*x*∈*A*. *B x*)
by (*fact SUP-upper*)

lemma *UN-least*: (⋀*x*. *x* ∈ *A* ⟹ *B x* ⊆ *C*) ⟹ ($\bigcup$*x*∈*A*. *B x*) ⊆ *C*
by (*fact SUP-least*)

lemma *Collect-bex-eq*: {*x*. ∃ *y*∈*A*. *P x y*} = ($\bigcup$*y*∈*A*. {*x*. *P x y*})
by *blast*

lemma *UN-insert-distrib*: *u* ∈ *A* ⟹ ($\bigcup$*x*∈*A*. *insert a* (*B x*)) = *insert a* ($\bigcup$*x*∈*A*. *B x*)
by *blast*

lemma *UN-empty*: ($\bigcup$*x*∈{}. *B x*) = {}
by (*fact SUP-empty*)

lemma *UN-empty2*: ($\bigcup$*x*∈*A*. {}) = {}
by (*fact SUP-bot*)

lemma *UN-absorb*: *k* ∈ *I* ⟹ *A k* ∪ ($\bigcup$*i*∈*I*. *A i*) = ($\bigcup$*i*∈*I*. *A i*)
by (*fact SUP-absorb*)

lemma *UN-insert* [*simp*]: ($\bigcup$*x*∈*insert a A*. *B x*) = *B a* ∪ $\bigcup$(*B* ' *A*)
by (*fact SUP-insert*)

lemma *UN-Un* [*simp*]: ($\bigcup$*i* ∈ *A* ∪ *B*. *M i*) = ($\bigcup$*i*∈*A*. *M i*) ∪ ($\bigcup$*i*∈*B*. *M i*)
by (*fact SUP-union*)

lemma *UN-UN-flatten*: $(\bigcup x \in (\bigcup y{\in}A.\ B\ y).\ C\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}B\ y.\ C\ x)$
by *blast*

lemma *UN-subset-iff*: $((\bigcup i{\in}I.\ A\ i) \subseteq B) = (\forall i{\in}I.\ A\ i \subseteq B)$
by (*fact SUP-le-iff*)

lemma *UN-constant* [*simp*]: $(\bigcup y{\in}A.\ c) = (\mathit{if}\ A = \{\}\ \mathit{then}\ \{\}\ \mathit{else}\ c)$
by (*fact SUP-constant*)

lemma *UNION-singleton-eq-range*: $(\bigcup x{\in}A.\ \{f\ x\}) = f\ `\ A$
by *blast*

lemma *image-Union*: $f\ `\ \bigcup S = (\bigcup x{\in}S.\ f\ `\ x)$
by *blast*

lemma *UNION-empty-conv*:
$\{\} = (\bigcup x{\in}A.\ B\ x) \longleftrightarrow (\forall x{\in}A.\ B\ x = \{\})$
$(\bigcup x{\in}A.\ B\ x) = \{\} \longleftrightarrow (\forall x{\in}A.\ B\ x = \{\})$
by (*fact SUP-bot-conv*)+

lemma *Collect-ex-eq*: $\{x.\ \exists y.\ P\ x\ y\} = (\bigcup y.\ \{x.\ P\ x\ y\})$
by *blast*

lemma *ball-UN*: $(\forall z \in \bigcup(B\ `\ A).\ P\ z) \longleftrightarrow (\forall x{\in}A.\ \forall z \in B\ x.\ P\ z)$
by *blast*

lemma *bex-UN*: $(\exists z \in \bigcup(B\ `\ A).\ P\ z) \longleftrightarrow (\exists x{\in}A.\ \exists z{\in}B\ x.\ P\ z)$
by *blast*

lemma *Un-eq-UN*: $A \cup B = (\bigcup b.\ \mathit{if}\ b\ \mathit{then}\ A\ \mathit{else}\ B)$
by *safe* (*auto simp add*: *if-split-mem2*)

lemma *UN-bool-eq*: $(\bigcup b.\ A\ b) = (A\ \mathit{True} \cup A\ \mathit{False})$
by (*fact SUP-UNIV-bool-expand*)

lemma *UN-Pow-subset*: $(\bigcup x{\in}A.\ \mathit{Pow}\ (B\ x)) \subseteq \mathit{Pow}\ (\bigcup x{\in}A.\ B\ x)$
by *blast*

lemma *UN-mono*:
$A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \subseteq g\ x) \Longrightarrow$
$(\bigcup x{\in}A.\ f\ x) \subseteq (\bigcup x{\in}B.\ g\ x)$
by (*fact SUP-subset-mono*)

lemma *vimage-Union*: $f\ -`\ (\bigcup A) = (\bigcup X{\in}A.\ f\ -`\ X)$
by *blast*

lemma *vimage-UN*: $f\ -`\ (\bigcup x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ f\ -`\ B\ x)$
by *blast*

lemma *vimage-eq-UN*: $f -` B = (\bigcup y{\in}B.\ f -` \{y\})$
— NOT suitable for rewriting
by *blast*

lemma *image-UN*: $f ` \bigcup(B ` A) = (\bigcup x{\in}A.\ f ` B\ x)$
by *blast*

lemma *UN-singleton* [*simp*]: $(\bigcup x{\in}A.\ \{x\}) = A$
by *blast*

lemma *inj-on-image*: *inj-on* $f\ (\bigcup A) \Longrightarrow$ *inj-on* $((`)\ f)\ A$
unfolding *inj-on-def* **by** *blast*

11.6.5 Distributive laws

lemma *Int-Union*: $A \cap \bigcup B = (\bigcup C{\in}B.\ A \cap C)$
by *blast*

lemma *Un-Inter*: $A \cup \bigcap B = (\bigcap C{\in}B.\ A \cup C)$
by *blast*

lemma *Int-Union2*: $\bigcup B \cap A = (\bigcup C{\in}B.\ C \cap A)$
by *blast*

lemma *INT-Int-distrib*: $(\bigcap i{\in}I.\ A\ i \cap B\ i) = (\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i)$
by (*rule sym*) (*rule INF-inf-distrib*)

lemma *UN-Un-distrib*: $(\bigcup i{\in}I.\ A\ i \cup B\ i) = (\bigcup i{\in}I.\ A\ i) \cup (\bigcup i{\in}I.\ B\ i)$
by (*rule sym*) (*rule SUP-sup-distrib*)

lemma *Int-Inter-image*: $(\bigcap x{\in}C.\ A\ x \cap B\ x) = \bigcap(A ` C) \cap \bigcap(B ` C)$
by (*simp add*: *INT-Int-distrib*)

lemma *Int-Inter-eq*: $A \cap \bigcap\mathcal{B} = (\textit{if}\ \mathcal{B}{=}\{\}\ \textit{then}\ A\ \textit{else}\ (\bigcap B{\in}\mathcal{B}.\ A \cap B))$
$\bigcap\mathcal{B} \cap A = (\textit{if}\ \mathcal{B}{=}\{\}\ \textit{then}\ A\ \textit{else}\ (\bigcap B{\in}\mathcal{B}.\ B \cap A))$
by *auto*

lemma *Un-Union-image*: $(\bigcup x{\in}C.\ A\ x \cup B\ x) = \bigcup(A ` C) \cup \bigcup(B ` C)$
— Devlin, Fundamentals of Contemporary Set Theory, page 12, exercise 5:
— Union of a family of unions
by (*simp add*: *UN-Un-distrib*)

lemma *Un-INT-distrib*: $B \cup (\bigcap i{\in}I.\ A\ i) = (\bigcap i{\in}I.\ B \cup A\ i)$
by *blast*

lemma *Int-UN-distrib*: $B \cap (\bigcup i{\in}I.\ A\ i) = (\bigcup i{\in}I.\ B \cap A\ i)$
— Halmos, Naive Set Theory, page 35.
by *blast*

lemma *Int-UN-distrib2*: $(\bigcup i{\in}I.\ A\ i) \cap (\bigcup j{\in}J.\ B\ j) = (\bigcup i{\in}I.\ \bigcup j{\in}J.\ A\ i \cap B\ j)$
by *blast*

lemma *Un-INT-distrib2*: $(\bigcap i{\in}I.\ A\ i) \cup (\bigcap j{\in}J.\ B\ j) = (\bigcap i{\in}I.\ \bigcap j{\in}J.\ A\ i \cup B\ j)$
by *blast*

lemma *Union-disjoint*: $(\bigcup C \cap A = \{\}) \longleftrightarrow (\forall B{\in}C.\ B \cap A = \{\})$
by *blast*

lemma *SUP-UNION*: $(\bigsqcup x{\in}(\bigcup y{\in}A.\ g\ y).\ f\ x) = (\bigsqcup y{\in}A.\ \bigsqcup x{\in}g\ y.\ f\ x$ :: - :: *complete-lattice*$)$
by (*rule order-antisym*) (*blast intro*: *SUP-least SUP-upper2*)+

11.7 Injections and bijections

lemma *inj-on-Inter*: $S \neq \{\} \Longrightarrow (\bigwedge A.\ A \in S \Longrightarrow$ *inj-on f A*$) \Longrightarrow$ *inj-on f* $(\bigcap S)$
unfolding *inj-on-def* **by** *blast*

lemma *inj-on-INTER*: $I \neq \{\} \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow$ *inj-on f* $(A\ i)) \Longrightarrow$ *inj-on f* $(\bigcap i \in I.\ A\ i)$
unfolding *inj-on-def* **by** *safe simp*

lemma *inj-on-UNION-chain*:
assumes *chain*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow A\ i \leq A\ j \vee A\ j \leq A\ i$
and *inj*: $\bigwedge i.\ i \in I \Longrightarrow$ *inj-on f* $(A\ i)$
shows *inj-on f* $(\bigcup i \in I.\ A\ i)$
proof −
have $x = y$
if $*$: $i \in I\ j \in I$
and $**$: $x \in A\ i\ y \in A\ j$
and $***$: $f\ x = f\ y$
for $i\ j\ x\ y$
using *chain* [*OF* $*$]
proof
assume $A\ i \leq A\ j$
with $**$ **have** $x \in A\ j$ **by** *auto*
with *inj* $*$ $**$ $***$ **show** *?thesis*
by (*auto simp add*: *inj-on-def*)
next
assume $A\ j \leq A\ i$
with $**$ **have** $y \in A\ i$ **by** *auto*
with *inj* $*$ $**$ $***$ **show** *?thesis*
by (*auto simp add*: *inj-on-def*)
qed
then show *?thesis*
by (*unfold inj-on-def UNION-eq*) *auto*

qed

lemma *bij-betw-UNION-chain*:
assumes *chain*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow A\ i \leq A\ j \vee A\ j \leq A\ i$
and *bij*: $\bigwedge i.\ i \in I \Longrightarrow$ *bij-betw f* $(A\ i)\ (A'\ i)$
shows *bij-betw f* $(\bigcup i \in I.\ A\ i)\ (\bigcup i \in I.\ A'\ i)$
unfolding *bij-betw-def*
proof *safe*
have $\bigwedge i.\ i \in I \Longrightarrow$ *inj-on f* $(A\ i)$
using *bij bij-betw-def*[*of f*] **by** *auto*
then show *inj-on f* $(\bigcup (A\ `\ I))$
using *chain inj-on-UNION-chain*[*of I A f*] **by** *auto*
next
fix *i x*
assume $*$: $i \in I\ x \in A\ i$
with *bij* **have** $f\ x \in A'\ i$
by (*auto simp*: *bij-betw-def*)
with $*$ **show** $f\ x \in \bigcup (A'\ `\ I)$ **by** *blast*
next
fix *i x'*
assume $*$: $i \in I\ x' \in A'\ i$
with *bij* **have** $\exists x \in A\ i.\ x' = f\ x$
unfolding *bij-betw-def* **by** *blast*
with $*$ **have** $\exists j \in I.\ \exists x \in A\ j.\ x' = f\ x$
by *blast*
then show $x' \in f\ `\ \bigcup (A\ `\ I)$
by *blast*
qed

lemma *image-INT*: *inj-on f C* $\Longrightarrow \forall x{\in}A.\ B\ x \subseteq C \Longrightarrow j \in A \Longrightarrow f\ `\ (\bigcap (B\ `\ A))$ $= (\bigcap x{\in}A.\ f\ `\ B\ x)$
by (*auto simp add*: *inj-on-def*) *blast*

lemma *bij-image-INT*: *bij f* $\Longrightarrow f\ `\ (\bigcap (B\ `\ A)) = (\bigcap x{\in}A.\ f\ `\ B\ x)$
by (*auto simp*: *bij-def inj-def surj-def*) *blast*

lemma *UNION-fun-upd*: $\bigcup (A(i := B)\ `\ J) = \bigcup (A\ `\ (J - \{i\})) \cup$ (*if* $i \in J$ *then* B *else* $\{\}$)
by (*auto simp add*: *set-eq-iff*)

lemma *bij-betw-Pow*:
assumes *bij-betw f A B*
shows *bij-betw* (*image f*) (*Pow A*) (*Pow B*)
proof −
from *assms* **have** *inj-on f A*
by (*rule bij-betw-imp-inj-on*)
then have *inj-on f* $(\bigcup (Pow\ A))$
by *simp*

then have *inj-on* (*image f*) (*Pow A*)
by (*rule inj-on-image*)
then have *bij-betw* (*image f*) (*Pow A*) (*image f ‘ Pow A*)
by (*rule inj-on-imp-bij-betw*)
moreover from *assms* **have** *f ‘ A = B*
by (*rule bij-betw-imp-surj-on*)
then have *image f ‘ Pow A = Pow B*
by (*rule image-Pow-surj*)
ultimately show *?thesis* **by** *simp*
qed

11.7.1 Complement

lemma *Compl-INT* [*simp*]: $-(\bigcap x{\in}A.\ B\ x) = (\bigcup x{\in}A.\ -B\ x)$
by *blast*

lemma *Compl-UN* [*simp*]: $-(\bigcup x{\in}A.\ B\ x) = (\bigcap x{\in}A.\ -B\ x)$
by *blast*

11.7.2 Miniscoping and maxiscoping

Miniscoping: pushing in quantifiers and big Unions and Intersections.

lemma *UN-simps* [*simp*]:
$\bigwedge a\ B\ C.\ (\bigcup x{\in}C.\ \mathit{insert}\ a\ (B\ x)) = (\mathit{if}\ C{=}\{\}\ \mathit{then}\ \{\}\ \mathit{else}\ \mathit{insert}\ a\ (\bigcup x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x \cup B) = ((\mathit{if}\ C{=}\{\}\ \mathit{then}\ \{\}\ \mathit{else}\ (\bigcup x{\in}C.\ A\ x) \cup B))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A \cup B\ x) = ((\mathit{if}\ C{=}\{\}\ \mathit{then}\ \{\}\ \mathit{else}\ A \cup (\bigcup x{\in}C.\ B\ x)))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x \cap B) = ((\bigcup x{\in}C.\ A\ x) \cap B)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A \cap B\ x) = (A \cap (\bigcup x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x - B) = ((\bigcup x{\in}C.\ A\ x) - B)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A - B\ x) = (A - (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B.\ (\bigcup x{\in}\bigcup A.\ B\ x) = (\bigcup y{\in}A.\ \bigcup x{\in}y.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcup z{\in}(\bigcup (B\ ‘\ A)).\ C\ z) = (\bigcup x{\in}A.\ \bigcup z{\in}B\ x.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcup x{\in}f‘A.\ B\ x) = (\bigcup a{\in}A.\ B\ (f\ a))$
by *auto*

lemma *INT-simps* [*simp*]:
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x \cap B) = (\mathit{if}\ C{=}\{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ (\bigcap x{\in}C.\ A\ x) \cap B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A \cap B\ x) = (\mathit{if}\ C{=}\{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ A \cap (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x - B) = (\mathit{if}\ C{=}\{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ (\bigcap x{\in}C.\ A\ x) - B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A - B\ x) = (\mathit{if}\ C{=}\{\}\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ A - (\bigcup x{\in}C.\ B\ x))$
$\bigwedge a\ B\ C.\ (\bigcap x{\in}C.\ \mathit{insert}\ a\ (B\ x)) = \mathit{insert}\ a\ (\bigcap x{\in}C.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x \cup B) = ((\bigcap x{\in}C.\ A\ x) \cup B)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A \cup B\ x) = (A \cup (\bigcap x{\in}C.\ B\ x))$
$\bigwedge A\ B.\ (\bigcap x{\in}\bigcup A.\ B\ x) = (\bigcap y{\in}A.\ \bigcap x{\in}y.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcap z{\in}(\bigcup (B\ ‘\ A)).\ C\ z) = (\bigcap x{\in}A.\ \bigcap z{\in}B\ x.\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcap x{\in}f‘A.\ B\ x) = (\bigcap a{\in}A.\ B\ (f\ a))$
by *auto*

lemma *UN-ball-bex-simps* [*simp*]:
$\bigwedge A\ P.\ (\forall x{\in}\bigcup A.\ P\ x) \longleftrightarrow (\forall y{\in}A.\ \forall x{\in}y.\ P\ x)$
$\bigwedge A\ B\ P.\ (\forall x{\in}(\bigcup(B\ `\ A)).\ P\ x) = (\forall a{\in}A.\ \forall x{\in}\ B\ a.\ P\ x)$
$\bigwedge A\ P.\ (\exists x{\in}\bigcup A.\ P\ x) \longleftrightarrow (\exists y{\in}A.\ \exists x{\in}y.\ P\ x)$
$\bigwedge A\ B\ P.\ (\exists x{\in}(\bigcup(B\ `\ A)).\ P\ x) \longleftrightarrow (\exists a{\in}A.\ \exists x{\in}B\ a.\ P\ x)$
by *auto*

Maxiscoping: pulling out big Unions and Intersections.

lemma *UN-extend-simps*:
$\bigwedge a\ B\ C.\ insert\ a\ (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ \{a\}\ else\ (\bigcup x{\in}C.\ insert\ a\ (B\ x)))$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}C.\ A\ x) \cup B = (if\ C{=}\{\}\ then\ B\ else\ (\bigcup x{\in}C.\ A\ x \cup B))$
$\bigwedge A\ B\ C.\ A \cup (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcup x{\in}C.\ A \cup B\ x))$
$\bigwedge A\ B\ C.\ ((\bigcup x{\in}C.\ A\ x) \cap B) = (\bigcup x{\in}C.\ A\ x \cap B)$
$\bigwedge A\ B\ C.\ (A \cap (\bigcup x{\in}C.\ B\ x)) = (\bigcup x{\in}C.\ A \cap B\ x)$
$\bigwedge A\ B\ C.\ ((\bigcup x{\in}C.\ A\ x) - B) = (\bigcup x{\in}C.\ A\ x - B)$
$\bigwedge A\ B\ C.\ (A - (\bigcap x{\in}C.\ B\ x)) = (\bigcup x{\in}C.\ A - B\ x)$
$\bigwedge A\ B.\ (\bigcup y{\in}A.\ \bigcup x{\in}y.\ B\ x) = (\bigcup x{\in}\bigcup A.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcup x{\in}A.\ \bigcup z{\in}B\ x.\ C\ z) = (\bigcup z{\in}(\bigcup(B\ `\ A)).\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcup a{\in}A.\ B\ (f\ a)) = (\bigcup x{\in}f`A.\ B\ x)$
by *auto*

lemma *INT-extend-simps*:
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x) \cap B = (if\ C{=}\{\}\ then\ B\ else\ (\bigcap x{\in}C.\ A\ x \cap B))$
$\bigwedge A\ B\ C.\ A \cap (\bigcap x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcap x{\in}C.\ A \cap B\ x))$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}C.\ A\ x) - B = (if\ C{=}\{\}\ then\ UNIV - B\ else\ (\bigcap x{\in}C.\ A\ x - B))$
$\bigwedge A\ B\ C.\ A - (\bigcup x{\in}C.\ B\ x) = (if\ C{=}\{\}\ then\ A\ else\ (\bigcap x{\in}C.\ A - B\ x))$
$\bigwedge a\ B\ C.\ insert\ a\ (\bigcap x{\in}C.\ B\ x) = (\bigcap x{\in}C.\ insert\ a\ (B\ x))$
$\bigwedge A\ B\ C.\ ((\bigcap x{\in}C.\ A\ x) \cup B) = (\bigcap x{\in}C.\ A\ x \cup B)$
$\bigwedge A\ B\ C.\ A \cup (\bigcap x{\in}C.\ B\ x) = (\bigcap x{\in}C.\ A \cup B\ x)$
$\bigwedge A\ B.\ (\bigcap y{\in}A.\ \bigcap x{\in}y.\ B\ x) = (\bigcap x{\in}\bigcup A.\ B\ x)$
$\bigwedge A\ B\ C.\ (\bigcap x{\in}A.\ \bigcap z{\in}B\ x.\ C\ z) = (\bigcap z{\in}(\bigcup(B\ `\ A)).\ C\ z)$
$\bigwedge A\ B\ f.\ (\bigcap a{\in}A.\ B\ (f\ a)) = (\bigcap x{\in}f`A.\ B\ x)$
by *auto*

Finally

lemmas *mem-simps* =
insert-iff empty-iff Un-iff Int-iff Compl-iff Diff-iff
mem-Collect-eq UN-iff Union-iff INT-iff Inter-iff
— Each of these has ALREADY been added [*simp*] above.

end

12 Wrapping Existing Freely Generated Type's Constructors

theory *Ctr-Sugar*

imports *HOL*
keywords
print-case-translations :: *diag* **and**
free-constructors :: *thy-goal*
begin

consts
case-guard :: *bool* ⇒ *'a* ⇒ (*'a* ⇒ *'b*) ⇒ *'b*
case-nil :: *'a* ⇒ *'b*
case-cons :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*
case-elem :: *'a* ⇒ *'b* ⇒ *'a* ⇒ *'b*
case-abs :: (*'c* ⇒ *'b*) ⇒ *'b*

declare [[*coercion-args case-guard* − + −]]
declare [[*coercion-args case-cons* − −]]
declare [[*coercion-args case-abs* −]]
declare [[*coercion-args case-elem* − +]]

ML-file ‹*Tools/Ctr-Sugar/case-translation.ML*›

lemma *iffI-np*: ⟦*x* ⟹ ¬ *y*; ¬ *x* ⟹ *y*⟧ ⟹ ¬ *x* ⟷ *y*
by (*erule iffI*) (*erule contrapos-pn*)

lemma *iff-contradict*:
¬ *P* ⟹ *P* ⟷ *Q* ⟹ *Q* ⟹ *R*
¬ *Q* ⟹ *P* ⟷ *Q* ⟹ *P* ⟹ *R*
by *blast*+

ML-file ‹*Tools/Ctr-Sugar/ctr-sugar-util.ML*›
ML-file ‹*Tools/Ctr-Sugar/ctr-sugar-tactics.ML*›
ML-file ‹*Tools/Ctr-Sugar/ctr-sugar-code.ML*›
ML-file ‹*Tools/Ctr-Sugar/ctr-sugar.ML*›

Coinduction method that avoids some boilerplate compared with coinduct.

ML-file ‹*Tools/coinduction.ML*›

end

13 Knaster-Tarski Fixpoint Theorem and inductive definitions

theory *Inductive*
imports *Complete-Lattices Ctr-Sugar*
keywords
inductive coinductive inductive-cases inductive-simps :: *thy-defn* **and**
monos **and**
print-inductives :: *diag* **and**
old-rep-datatype :: *thy-goal* **and**

primrec :: *thy-defn*
begin

13.1 Least fixed points

context *complete-lattice*
begin

definition *lfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
  where $lfp\ f = Inf\ \{u.\ f\ u \leq u\}$

lemma *lfp-lowerbound*: $f\ A \leq A \Longrightarrow lfp\ f \leq A$
  unfolding *lfp-def* **by** (*rule Inf-lower*) *simp*

lemma *lfp-greatest*: $(\bigwedge u.\ f\ u \leq u \Longrightarrow A \leq u) \Longrightarrow A \leq lfp\ f$
  unfolding *lfp-def* **by** (*rule Inf-greatest*) *simp*

end

lemma *lfp-fixpoint*:
  assumes *mono f*
  shows $f\ (lfp\ f) = lfp\ f$
  unfolding *lfp-def*
proof (*rule order-antisym*)
  let *?H* $= \{u.\ f\ u \leq u\}$
  let *?a* $= \bigsqcap$ *?H*
  show f *?a* $\leq$ *?a*
  proof (*rule Inf-greatest*)
    fix *x*
    assume $x \in$ *?H*
    then have *?a* $\leq x$ **by** (*rule Inf-lower*)
    with ‹*mono f*› **have** f *?a* $\leq f\ x$ **..**
    also from ‹$x \in$ *?H*› **have** $f\ x \leq x$ **..**
    finally show f *?a* $\leq x$ **.**
  qed
  show *?a* $\leq f$ *?a*
  proof (*rule Inf-lower*)
    from ‹*mono f*› **and** ‹f *?a* $\leq$ *?a*› **have** $f\ (f$ *?a*$) \leq f$ *?a* **..**
    then show f *?a* $\in$ *?H* **..**
  qed
qed

lemma *lfp-unfold*: *mono* $f \Longrightarrow lfp\ f = f\ (lfp\ f)$
  by (*rule lfp-fixpoint* [*symmetric*])

lemma *lfp-const*: $lfp\ (\lambda x.\ t) = t$
  by (*rule lfp-unfold*) (*simp add*: *mono-def*)

lemma *lfp-eqI*: *mono* $F \Longrightarrow F\ x = x \Longrightarrow (\bigwedge z.\ F\ z = z \Longrightarrow x \leq z) \Longrightarrow lfp\ F = x$

by (*rule antisym*) (*simp-all add*: *lfp-lowerbound lfp-unfold*[*symmetric*])

13.2 General induction rules for least fixed points

lemma *lfp-ordinal-induct* [*case-names mono step union*]:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono*: *mono f*
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow S \leq lfp\ f \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (Sup\ M)$
shows *P* (*lfp f*)
proof −
let *?M* = $\{S.\ S \leq lfp\ f \wedge P\ S\}$
from *P-Union* **have** *P* (*Sup ?M*) **by** *simp*
also have *Sup ?M* = *lfp f*
proof (*rule antisym*)
show *Sup ?M* ≤ *lfp f*
by (*blast intro*: *Sup-least*)
then have *f* (*Sup ?M*) ≤ *f* (*lfp f*)
by (*rule mono* [*THEN monoD*])
then have *f* (*Sup ?M*) ≤ *lfp f*
using *mono* [*THEN lfp-unfold*] **by** *simp*
then have *f* (*Sup ?M*) ∈ *?M*
using *P-Union* **by** *simp* (*intro P-f Sup-least*, *auto*)
then have *f* (*Sup ?M*) ≤ *Sup ?M*
by (*rule Sup-upper*)
then show *lfp f* ≤ *Sup ?M*
by (*rule lfp-lowerbound*)
qed
finally show *?thesis* **.**
qed

theorem *lfp-induct*:
assumes *mono*: *mono f*
and *ind*: *f* (*inf* (*lfp f*) *P*) ≤ *P*
shows *lfp f* ≤ *P*
proof (*induct rule*: *lfp-ordinal-induct*)
case *mono*
show *?case* **by** *fact*
next
case (*step S*)
then show *?case*
by (*intro order-trans*[*OF - ind*] *monoD*[*OF mono*]) *auto*
next
case (*union M*)
then show *?case*
by (*auto intro*: *Sup-least*)
qed

lemma *lfp-induct-set*:

assumes *lfp*: $a \in lfp\ f$
and *mono*: *mono f*
and *hyp*: $\bigwedge x.\ x \in f\ (lfp\ f \cap \{x.\ P\ x\}) \Longrightarrow P\ x$
shows $P\ a$
by (*rule lfp-induct* [*THEN subsetD*, *THEN CollectD*, *OF mono - lfp*]) (*auto intro*: *hyp*)

lemma *lfp-ordinal-induct-set*:
assumes *mono*: *mono f*
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (\bigcup M)$
shows $P\ (lfp\ f)$
using *assms* **by** (*rule lfp-ordinal-induct*)

Definition forms of *lfp-unfold* and *lfp-induct*, to control unfolding.

lemma *def-lfp-unfold*: $h \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow h = f\ h$
by (*auto intro*!: *lfp-unfold*)

lemma *def-lfp-induct*: $A \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow f\ (inf\ A\ P) \leq P \Longrightarrow A \leq P$
by (*blast intro*: *lfp-induct*)

lemma *def-lfp-induct-set*:
$A \equiv lfp\ f \Longrightarrow mono\ f \Longrightarrow a \in A \Longrightarrow (\bigwedge x.\ x \in f\ (A \cap \{x.\ P\ x\}) \Longrightarrow P\ x) \Longrightarrow P\ a$
by (*blast intro*: *lfp-induct-set*)

Monotonicity of *lfp*!

lemma *lfp-mono*: $(\bigwedge Z.\ f\ Z \leq g\ Z) \Longrightarrow lfp\ f \leq lfp\ g$
by (*rule lfp-lowerbound* [*THEN lfp-greatest*]) (*blast intro*: *order-trans*)

13.3 Greatest fixed points

context *complete-lattice*
begin

definition *gfp* :: $('a \Rightarrow 'a) \Rightarrow 'a$
where $gfp\ f = Sup\ \{u.\ u \leq f\ u\}$

lemma *gfp-upperbound*: $X \leq f\ X \Longrightarrow X \leq gfp\ f$
by (*auto simp add*: *gfp-def intro*: *Sup-upper*)

lemma *gfp-least*: $(\bigwedge u.\ u \leq f\ u \Longrightarrow u \leq X) \Longrightarrow gfp\ f \leq X$
by (*auto simp add*: *gfp-def intro*: *Sup-least*)

end

lemma *lfp-le-gfp*: $mono\ f \Longrightarrow lfp\ f \leq gfp\ f$
by (*rule gfp-upperbound*) (*simp add*: *lfp-fixpoint*)

lemma *gfp-fixpoint*:
  assumes *mono f*
  shows *f* (*gfp f*) = *gfp f*
  unfolding *gfp-def*
proof (*rule order-antisym*)
  let *?H* = {*u*. *u* ≤ *f u*}
  let *?a* = ⨆ *?H*
  show *?a* ≤ *f ?a*
  proof (*rule Sup-least*)
    fix *x*
    assume *x* ∈ *?H*
    then have *x* ≤ *f x* **..**
    also from ‹*x* ∈ *?H*› **have** *x* ≤ *?a* **by** (*rule Sup-upper*)
    with ‹*mono f*› **have** *f x* ≤ *f ?a* **..**
    finally show *x* ≤ *f ?a* **.**
  qed
  show *f ?a* ≤ *?a*
  proof (*rule Sup-upper*)
    from ‹*mono f*› **and** ‹*?a* ≤ *f ?a*› **have** *f ?a* ≤ *f* (*f ?a*) **..**
    then show *f ?a* ∈ *?H* **..**
  qed
qed

lemma *gfp-unfold*: *mono f* ⟹ *gfp f* = *f* (*gfp f*)
  by (*rule gfp-fixpoint* [*symmetric*])

lemma *gfp-const*: *gfp* (λ*x*. *t*) = *t*
  by (*rule gfp-unfold*) (*simp add*: *mono-def*)

lemma *gfp-eqI*: *mono F* ⟹ *F x* = *x* ⟹ (⋀*z*. *F z* = *z* ⟹ *z* ≤ *x*) ⟹ *gfp F* = *x*
  by (*rule antisym*) (*simp-all add*: *gfp-upperbound gfp-unfold*[*symmetric*])

13.4 Coinduction rules for greatest fixed points

Weak version.

lemma *weak-coinduct*: *a* ∈ *X* ⟹ *X* ⊆ *f X* ⟹ *a* ∈ *gfp f*
  by (*rule gfp-upperbound* [*THEN subsetD*]) *auto*

lemma *weak-coinduct-image*: *a* ∈ *X* ⟹ *g*‘*X* ⊆ *f* (*g*‘*X*) ⟹ *g a* ∈ *gfp f*
  apply (*erule gfp-upperbound* [*THEN subsetD*])
  apply (*erule imageI*)
  done

lemma *coinduct-lemma*: *X* ≤ *f* (*sup X* (*gfp f*)) ⟹ *mono f* ⟹ *sup X* (*gfp f*) ≤ *f* (*sup X* (*gfp f*))
  apply (*frule gfp-unfold* [*THEN eq-refl*])
  apply (*drule mono-sup*)
  apply (*rule le-supI*)

apply *assumption*
apply (*rule order-trans*)
apply (*rule order-trans*)
apply *assumption*
apply (*rule sup-ge2*)
apply *assumption*
done

Strong version, thanks to Coen and Frost.

lemma *coinduct-set*: *mono f* $\Longrightarrow$ $a \in X \Longrightarrow X \subseteq f\ (X \cup \mathit{gfp}\ f) \Longrightarrow a \in \mathit{gfp}\ f$
by (*rule weak-coinduct*[*rotated*], *rule coinduct-lemma*) *blast*+

lemma *gfp-fun-UnI2*: *mono f* $\Longrightarrow$ $a \in \mathit{gfp}\ f \Longrightarrow a \in f\ (X \cup \mathit{gfp}\ f)$
by (*blast dest*: *gfp-fixpoint mono-Un*)

lemma *gfp-ordinal-induct*[*case-names mono step union*]:
fixes $f :: 'a{::}\mathit{complete\text{-}lattice} \Rightarrow 'a$
assumes *mono*: *mono f*
and *P-f*: $\bigwedge S.\ P\ S \Longrightarrow \mathit{gfp}\ f \leq S \Longrightarrow P\ (f\ S)$
and *P-Union*: $\bigwedge M.\ \forall S{\in}M.\ P\ S \Longrightarrow P\ (\mathit{Inf}\ M)$
shows *P* (*gfp f*)
proof −
let *?M* = $\{S.\ \mathit{gfp}\ f \leq S \wedge P\ S\}$
from *P-Union* **have** *P* (*Inf ?M*) **by** *simp*
also have *Inf ?M* = *gfp f*
proof (*rule antisym*)
show *gfp f* ≤ *Inf ?M*
by (*blast intro*: *Inf-greatest*)
then have *f* (*gfp f*) ≤ *f* (*Inf ?M*)
by (*rule mono* [*THEN monoD*])
then have *gfp f* ≤ *f* (*Inf ?M*)
using *mono* [*THEN gfp-unfold*] **by** *simp*
then have *f* (*Inf ?M*) ∈ *?M*
using *P-Union* **by** *simp* (*intro P-f Inf-greatest*, *auto*)
then have *Inf ?M* ≤ *f* (*Inf ?M*)
by (*rule Inf-lower*)
then show *Inf ?M* ≤ *gfp f*
by (*rule gfp-upperbound*)
qed
finally show *?thesis* **.**
qed

lemma *coinduct*:
assumes *mono*: *mono f*
and *ind*: $X \leq f\ (\mathit{sup}\ X\ (\mathit{gfp}\ f))$
shows $X \leq \mathit{gfp}\ f$
proof (*induct rule*: *gfp-ordinal-induct*)
case *mono*
then show *?case* **by** *fact*

next
case (*step S*)
then show *?case*
by (*intro order-trans*[*OF ind* -] *monoD*[*OF mono*]) *auto*
next
case (*union M*)
then show *?case*
by (*auto intro*: *mono Inf-greatest*)
qed

13.5 Even Stronger Coinduction Rule, by Martin Coen

Weakens the condition $X \subseteq f\ X$ to one expressed using both *lfp* and *gfp*

lemma *coinduct3-mono-lemma*: *mono f* $\Longrightarrow$ *mono* ($\lambda x.\ f\ x \cup X \cup B$)
by (*iprover intro*: *subset-refl monoI Un-mono monoD*)

lemma *coinduct3-lemma*:
$X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f)) \Longrightarrow mono\ f \Longrightarrow$
lfp ($\lambda x.\ f\ x \cup X \cup gfp\ f$) $\subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f))$
apply (*rule subset-trans*)
apply (*erule coinduct3-mono-lemma* [*THEN lfp-unfold* [*THEN eq-refl*]])
apply (*rule Un-least* [*THEN Un-least*])
apply (*rule subset-refl*, *assumption*)
apply (*rule gfp-unfold* [*THEN equalityD1*, *THEN subset-trans*], *assumption*)
apply (*rule monoD*, *assumption*)
apply (*subst coinduct3-mono-lemma* [*THEN lfp-unfold*], *auto*)
done

lemma *coinduct3*: *mono f* $\Longrightarrow$ $a \in X \Longrightarrow X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup gfp\ f)) \Longrightarrow$
$a \in gfp\ f$
apply (*rule coinduct3-lemma* [*THEN* [*2*] *weak-coinduct*])
apply (*rule coinduct3-mono-lemma* [*THEN lfp-unfold*, *THEN ssubst*])
apply *simp-all*
done

Definition forms of *gfp-unfold* and *coinduct*, to control unfolding.

lemma *def-gfp-unfold*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow A = f\ A$
by (*auto intro*!: *gfp-unfold*)

lemma *def-coinduct*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow X \leq f\ (sup\ X\ A) \Longrightarrow X \leq A$
by (*iprover intro*!: *coinduct*)

lemma *def-coinduct-set*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (X \cup A)$
$\Longrightarrow a \in A$
by (*auto intro*!: *coinduct-set*)

lemma *def-Collect-coinduct*:
$A \equiv gfp\ (\lambda w.\ Collect\ (P\ w)) \Longrightarrow mono\ (\lambda w.\ Collect\ (P\ w)) \Longrightarrow a \in X \Longrightarrow$
$(\bigwedge z.\ z \in X \Longrightarrow P\ (X \cup A)\ z) \Longrightarrow a \in A$

by (*erule def-coinduct-set*) *auto*

lemma *def-coinduct3*: $A \equiv gfp\ f \Longrightarrow mono\ f \Longrightarrow a \in X \Longrightarrow X \subseteq f\ (lfp\ (\lambda x.\ f\ x \cup X \cup A)) \Longrightarrow a \in A$
by (*auto intro*!: *coinduct3*)

Monotonicity of *gfp*!

lemma *gfp-mono*: $(\bigwedge Z.\ f\ Z \leq g\ Z) \Longrightarrow gfp\ f \leq gfp\ g$
by (*rule gfp-upperbound* [*THEN gfp-least*]) (*blast intro*: *order-trans*)

13.6 Rules for fixed point calculus

lemma *lfp-rolling*:
assumes *mono g mono f*
shows $g\ (lfp\ (\lambda x.\ f\ (g\ x))) = lfp\ (\lambda x.\ g\ (f\ x))$
proof (*rule antisym*)
have $*$: $mono\ (\lambda x.\ f\ (g\ x))$
using *assms* **by** (*auto simp*: *mono-def*)
show $lfp\ (\lambda x.\ g\ (f\ x)) \leq g\ (lfp\ (\lambda x.\ f\ (g\ x)))$
by (*rule lfp-lowerbound*) (*simp add*: *lfp-unfold*[*OF* $*$, *symmetric*])
show $g\ (lfp\ (\lambda x.\ f\ (g\ x))) \leq lfp\ (\lambda x.\ g\ (f\ x))$
proof (*rule lfp-greatest*)
fix *u*
assume *u*: $g\ (f\ u) \leq u$
then have $g\ (lfp\ (\lambda x.\ f\ (g\ x))) \leq g\ (f\ u)$
by (*intro assms*[*THEN monoD*] *lfp-lowerbound*)
with *u* **show** $g\ (lfp\ (\lambda x.\ f\ (g\ x))) \leq u$
by *auto*
qed
qed

lemma *lfp-lfp*:
assumes *f*: $\bigwedge x\ y\ w\ z.\ x \leq y \Longrightarrow w \leq z \Longrightarrow f\ x\ w \leq f\ y\ z$
shows $lfp\ (\lambda x.\ lfp\ (f\ x)) = lfp\ (\lambda x.\ f\ x\ x)$
proof (*rule antisym*)
have $*$: $mono\ (\lambda x.\ f\ x\ x)$
by (*blast intro*: *monoI f*)
show $lfp\ (\lambda x.\ lfp\ (f\ x)) \leq lfp\ (\lambda x.\ f\ x\ x)$
by (*intro lfp-lowerbound*) (*simp add*: *lfp-unfold*[*OF* $*$, *symmetric*])
show $lfp\ (\lambda x.\ lfp\ (f\ x)) \geq lfp\ (\lambda x.\ f\ x\ x)$ (**is** *?F* $\geq$ -)
proof (*intro lfp-lowerbound*)
have $*$: *?F* $= lfp\ (f$ *?F*$)$
by (*rule lfp-unfold*) (*blast intro*: *monoI lfp-mono f*)
also have $\ldots = f$ *?F* $(lfp\ (f$ *?F*$))$
by (*rule lfp-unfold*) (*blast intro*: *monoI lfp-mono f*)
finally show *f ?F ?F* $\leq$ *?F*
by (*simp add*: $*$[*symmetric*])
qed
qed

lemma *gfp-rolling*:
  assumes *mono g mono f*
  shows *g (gfp (λx. f (g x))) = gfp (λx. g (f x))*
proof (*rule antisym*)
  have ∗: *mono (λx. f (g x))*
    using *assms* **by** (*auto simp*: *mono-def*)
  show *g (gfp (λx. f (g x))) ≤ gfp (λx. g (f x))*
    by (*rule gfp-upperbound*) (*simp add*: *gfp-unfold*[*OF* ∗, *symmetric*])
  show *gfp (λx. g (f x)) ≤ g (gfp (λx. f (g x)))*
  proof (*rule gfp-least*)
    fix *u*
    assume *u*: *u ≤ g (f u)*
    then have *g (f u) ≤ g (gfp (λx. f (g x)))*
      by (*intro assms*[*THEN monoD*] *gfp-upperbound*)
    with *u* **show** *u ≤ g (gfp (λx. f (g x)))*
      by *auto*
  qed
qed

lemma *gfp-gfp*:
  assumes *f*: *⋀x y w z. x ≤ y ⟹ w ≤ z ⟹ f x w ≤ f y z*
  shows *gfp (λx. gfp (f x)) = gfp (λx. f x x)*
proof (*rule antisym*)
  have ∗: *mono (λx. f x x)*
    by (*blast intro*: *monoI f*)
  show *gfp (λx. f x x) ≤ gfp (λx. gfp (f x))*
    by (*intro gfp-upperbound*) (*simp add*: *gfp-unfold*[*OF* ∗, *symmetric*])
  show *gfp (λx. gfp (f x)) ≤ gfp (λx. f x x)* (**is** *?F ≤* -)
  proof (*intro gfp-upperbound*)
    have ∗: *?F = gfp (f ?F)*
      by (*rule gfp-unfold*) (*blast intro*: *monoI gfp-mono f*)
    also have *... = f ?F (gfp (f ?F))*
      by (*rule gfp-unfold*) (*blast intro*: *monoI gfp-mono f*)
    finally show *?F ≤ f ?F ?F*
      by (*simp add*: ∗[*symmetric*])
  qed
qed

13.7 Inductive predicates and sets

Package setup.

lemmas *basic-monos =*
  subset-refl imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
  Collect-mono in-mono vimage-mono

lemma *le-rel-bool-arg-iff*: *X ≤ Y ⟷ X False ≤ Y False ∧ X True ≤ Y True*
  unfolding *le-fun-def le-bool-def* **using** *bool-induct* **by** *auto*

lemma *imp-conj-iff*: $((P \longrightarrow Q) \wedge P) = (P \wedge Q)$
by *blast*

lemma *meta-fun-cong*: $P \equiv Q \Longrightarrow P\ a \equiv Q\ a$
by *auto*

ML-file ‹*Tools/inductive.ML*›

lemmas [*mono*] =
imp-refl disj-mono conj-mono ex-mono all-mono if-bool-eq-conj
imp-mono not-mono
Ball-def Bex-def
induct-rulify-fallback

13.8 The Schroeder-Bernstein Theorem

See also:

- `$ISABELLE_HOME/src/HOL/ex/Set_Theory.thy`
- http://planetmath.org/proofofschroederbernsteintheoremusingtarskiknastertheorem
- Springer LNCS 828 (cover page)

theorem *Schroeder-Bernstein*:
fixes $f :: 'a \Rightarrow 'b$ **and** $g :: 'b \Rightarrow 'a$
and $A :: 'a\ set$ **and** $B :: 'b\ set$
assumes *inj1*: *inj-on f A* **and** *sub1*: $f\ `\ A \subseteq B$
and *inj2*: *inj-on g B* **and** *sub2*: $g\ `\ B \subseteq A$
shows $\exists h.\ bij\text{-}betw\ h\ A\ B$
proof (*rule exI*, *rule bij-betw-imageI*)
define X **where** $X = lfp\ (\lambda X.\ A - (g\ `\ (B - (f\ `\ X))))$
define g' **where** $g' = the\text{-}inv\text{-}into\ (B - (f\ `\ X))\ g$
let $?h = \lambda z.\ if\ z \in X\ then\ f\ z\ else\ g'\ z$

have X: $X = A - (g\ `\ (B - (f\ `\ X)))$
unfolding *X-def* **by** (*rule lfp-unfold*) (*blast intro*: *monoI*)
then have *X-compl*: $A - X = g\ `\ (B - (f\ `\ X))$
using *sub2* **by** *blast*

from *inj2* **have** *inj2'*: *inj-on g* $(B - (f\ `\ X))$
by (*rule inj-on-subset*) *auto*
with *X-compl* **have** $*$: $g'\ `\ (A - X) = B - (f\ `\ X)$
by (*simp add*: *g'-def*)

from *X* **have** *X-sub*: $X \subseteq A$ **by** *auto*
from *X sub1* **have** *fX-sub*: $f\ `\ X \subseteq B$ **by** *auto*

show $?h\ `\ A = B$

```
  proof −
    from X-sub have ?h ‘ A = ?h ‘ (X ∪ (A − X)) by auto
    also have ... = ?h ‘ X ∪ ?h ‘ (A − X) by (simp only: image-Un)
    also have ?h ‘ X = f ‘ X by auto
    also from ∗ have ?h ‘ (A − X) = B − (f ‘ X) by auto
    also from fX-sub have f ‘ X ∪ (B − f ‘ X) = B by blast
    finally show ?thesis .
  qed
  show inj-on ?h A
  proof −
    from inj1 X-sub have on-X: inj-on f X
      by (rule inj-on-subset)

    have on-X-compl: inj-on g′ (A − X)
      unfolding g′-def X-compl
      by (rule inj-on-the-inv-into) (rule inj2′)

    have impossible: False if eq: f a = g′ b and a: a ∈ X and b: b ∈ A − X for a
b
    proof −
      from a have fa: f a ∈ f ‘ X by (rule imageI)
      from b have g′ b ∈ g′ ‘ (A − X) by (rule imageI)
      with ∗ have g′ b ∈ − (f ‘ X) by simp
      with eq fa show False by simp
    qed

    show ?thesis
    proof (rule inj-onI)
      fix a b
      assume h: ?h a = ?h b
      assume a ∈ A and b ∈ A
      then consider a ∈ X b ∈ X | a ∈ A − X b ∈ A − X
        | a ∈ X b ∈ A − X | a ∈ A − X b ∈ X
        by blast
      then show a = b
      proof cases
        case 1
        with h on-X show ?thesis by (simp add: inj-on-eq-iff)
      next
        case 2
        with h on-X-compl show ?thesis by (simp add: inj-on-eq-iff)
      next
        case 3
        with h impossible [of a b] have False by simp
        then show ?thesis ..
      next
        case 4
        with h impossible [of b a] have False by simp
        then show ?thesis ..
```

qed
qed
qed
qed

13.9 Inductive datatypes and primitive recursion

Package setup.

ML-file ‹*Tools/Old-Datatype/old-datatype-aux.ML*›
ML-file ‹*Tools/Old-Datatype/old-datatype-prop.ML*›
ML-file ‹*Tools/Old-Datatype/old-datatype-data.ML*›
ML-file ‹*Tools/Old-Datatype/old-rep-datatype.ML*›
ML-file ‹*Tools/Old-Datatype/old-datatype-codegen.ML*›
ML-file ‹*Tools/BNF/bnf-fp-rec-sugar-util.ML*›
ML-file ‹*Tools/Old-Datatype/old-primrec.ML*›
ML-file ‹*Tools/BNF/bnf-lfp-rec-sugar.ML*›

Lambda-abstractions with pattern matching:

syntax (*ASCII*)
-lam-pats-syntax :: *cases-syn* ⇒ *'a* ⇒ *'b* (‹(‹*notation=abstraction*›%-)› *10*)
syntax
-lam-pats-syntax :: *cases-syn* ⇒ *'a* ⇒ *'b* (‹(‹*notation=abstraction*›λ-)› *10*)
parse-translation ‹
let
fun fun-tr ctxt [*cs*] =
let
val x =
Syntax.free (#*1* (*Name.variant x*
(*Name.build-context* (*Syntax-Trans.declare-term-names ctxt cs*))));
val ft = *Case-Translation.case-tr true ctxt* [*x*, *cs*];
in lambda x ft end
in [(***syntax-const*** ‹*-lam-pats-syntax*›, *fun-tr*)] *end*
›

end

14 Cartesian products

theory *Product-Type*
imports *Typedef Inductive Fun*
keywords *inductive-set coinductive-set* :: *thy-defn*
begin

14.1 *bool* is a datatype

free-constructors (*discs-sels*) *case-bool* **for** *True* | *False*
by *auto*

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup ‹*Sign.mandatory-path old*›

old-rep-datatype *True False* **by** (*auto intro*: *bool-induct*)

setup ‹*Sign.parent-path*›

But erase the prefix for properties that are not generated by *free-constructors*.

setup ‹*Sign.mandatory-path bool*›

lemmas *induct* = *old.bool.induct*
lemmas *inducts* = *old.bool.inducts*
lemmas *rec* = *old.bool.rec*
lemmas *simps* = *bool.distinct bool.case bool.rec*

setup ‹*Sign.parent-path*›

declare *case-split* [*cases type*: *bool*]
— prefer plain propositional version

lemma [*code*]:
HOL.equal False P ⟷ ¬ *P*
HOL.equal True P ⟷ *P*
HOL.equal P False ⟷ ¬ *P*
HOL.equal P True ⟷ *P*
by (*simp-all add*: *equal*)

lemma [*code nbe*]:
HOL.equal P P ⟷ *True* **for** *P* :: *bool*
by (*simp-all add*: *equal*)

lemma *If-case-cert*:
assumes *CASE* ≡ (λ*b*. *If b f g*)
shows (*CASE True* ≡ *f*) &&& (*CASE False* ≡ *g*)
using *assms* **by** *simp-all*

setup ‹*Code.declare-case-global* @{*thm If-case-cert*}›

code-printing
constant *HOL.equal* :: *bool* ⇒ *bool* ⇒ *bool* ⇀ (*Haskell*) **infix** *4* ==
| **class-instance** *bool* :: *equal* ⇀ (*Haskell*) –

14.2 The *unit* type

typedef *unit* = {*True*}
by *auto*

definition *Unity* :: *unit* (‹'(')›)
where () = *Abs-unit True*

lemma *unit-eq* [*no-atp*]: $u = ()$
by (*induct u*) (*simp add*: *Unity-def*)

Simplification procedure for *unit-eq*. Cannot use this rule directly — it loops!

simproc-setup *unit-eq* (*x*::*unit*) = ‹
K (*K* (*fn ct* =>
if HOLogic.is-unit (*Thm.term-of ct*) *then NONE*
else SOME (*mk-meta-eq* @{*thm unit-eq*})))
›

free-constructors *case-unit* **for** ()
by *auto*

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup ‹*Sign.mandatory-path old*›

old-rep-datatype () **by** *simp*

setup ‹*Sign.parent-path*›

But erase the prefix for properties that are not generated by *free-constructors*.

setup ‹*Sign.mandatory-path unit*›

lemmas *induct* = *old.unit.induct*
lemmas *inducts* = *old.unit.inducts*
lemmas *rec* = *old.unit.rec*
lemmas *simps* = *unit.case unit.rec*

setup ‹*Sign.parent-path*›

lemma *unit-all-eq1*: $(\bigwedge x{::}unit.\ PROP\ P\ x) \equiv PROP\ P\ ()$
by *simp*

lemma *unit-all-eq2*: $(\bigwedge x{::}unit.\ PROP\ P) \equiv PROP\ P$
by (*rule triv-forall-equality*)

This rewrite counters the effect of simproc *unit-eq* on $\lambda u{::}unit.\ f\ u$, replacing it by f rather than by $\lambda u.\ f\ ()$.

lemma *unit-abs-eta-conv* [*simp*]: $(\lambda u{::}unit.\ f\ ()) = f$
by (*rule ext*) *simp*

lemma *UNIV-unit*: $UNIV = \{()\}$
by *auto*

instantiation *unit* :: *default*
begin

definition *default* = ()

instance ..

end

instantiation *unit* :: {*complete-boolean-algebra*,*complete-linorder*,*wellorder*}
begin

definition *less-eq-unit* :: *unit* ⇒ *unit* ⇒ *bool*
  where (-::*unit*) ≤ - ⟷ *True*

lemma *less-eq-unit* [*iff*]: *u* ≤ *v* **for** *u v* :: *unit*
  by (*simp add*: *less-eq-unit-def*)

definition *less-unit* :: *unit* ⇒ *unit* ⇒ *bool*
  where (-::*unit*) < - ⟷ *False*

lemma *less-unit* [*iff*]: ¬ *u* < *v* **for** *u v* :: *unit*
  by (*simp-all add*: *less-eq-unit-def less-unit-def*)

definition *bot-unit* :: *unit*
  where [*code-unfold*]: ⊥ = ()

definition *top-unit* :: *unit*
  where [*code-unfold*]: ⊤ = ()

definition *inf-unit* :: *unit* ⇒ *unit* ⇒ *unit*
  where [*simp*]: - ⊓ - = ()

definition *sup-unit* :: *unit* ⇒ *unit* ⇒ *unit*
  where [*simp*]: - ⊔ - = ()

definition *Inf-unit* :: *unit set* ⇒ *unit*
  where [*simp*]: ⨅ - = ()

definition *Sup-unit* :: *unit set* ⇒ *unit*
  where [*simp*]: ⨆ - = ()

definition *uminus-unit* :: *unit* ⇒ *unit*
  where [*simp*]: − - = ()

declare *less-eq-unit-def* [*abs-def*, *code-unfold*]
  less-unit-def [*abs-def*, *code-unfold*]
  inf-unit-def [*abs-def*, *code-unfold*]
  sup-unit-def [*abs-def*, *code-unfold*]
  Inf-unit-def [*abs-def*, *code-unfold*]
  Sup-unit-def [*abs-def*, *code-unfold*]
  uminus-unit-def [*abs-def*, *code-unfold*]

instance
by *intro-classes auto*

end

lemma [*code*]: *HOL.equal u v* ⟷ *True* **for** *u v* :: *unit*
unfolding *equal unit-eq* [*of u*] *unit-eq* [*of v*] **by** (*rule iffI TrueI refl*)+

code-printing
type-constructor *unit* ⇀
(*SML*) *unit*
and (*OCaml*) *unit*
and (*Haskell*) ()
and (*Scala*) *Unit*
| **constant** *Unity* ⇀
(*SML*) ()
and (*OCaml*) ()
and (*Haskell*) ()
and (*Scala*) ()
| **class-instance** *unit* :: *equal* ⇀
(*Haskell*) –
| **constant** *HOL.equal* :: *unit* ⇒ *unit* ⇒ *bool* ⇀
(*Haskell*) **infix** *4* ==

code-reserved
(*SML*) *unit*
and (*OCaml*) *unit*
and (*Scala*) *Unit*

14.3 The product type

14.3.1 Type definition

definition *Pair-Rep* :: *'a* ⇒ *'b* ⇒ *'a* ⇒ *'b* ⇒ *bool*
where *Pair-Rep a b* = (λ*x y*. *x* = *a* ∧ *y* = *b*)

definition *prod* = {*f*. ∃ *a b*. *f* = *Pair-Rep* (*a*::*'a*) (*b*::*'b*)}

typedef (*'a*, *'b*) *prod* (‹(‹*notation*=‹*infix* ×››- ×/ -)› [*21*, *20*] *20*) = *prod* :: (*'a* ⇒ *'b* ⇒ *bool*) *set*
unfolding *prod-def* **by** *auto*

type-notation (*ASCII*)
prod (**infixr** ‹*›› *20*)

definition *Pair* :: *'a* ⇒ *'b* ⇒ *'a* × *'b*
where *Pair a b* = *Abs-prod* (*Pair-Rep a b*)

lemma *prod-cases*: (⋀*a b*. *P* (*Pair a b*)) ⟹ *P p*

by (*cases p*) (*auto simp add*: *prod-def Pair-def Pair-Rep-def*)

free-constructors *case-prod* **for** *Pair fst snd*
proof −
fix *P* :: *bool* **and** *p* :: *'a* × *'b*
show (⋀*x1 x2*. *p* = *Pair x1 x2* ⟹ *P*) ⟹ *P*
by (*cases p*) (*auto simp add*: *prod-def Pair-def Pair-Rep-def*)
next
fix *a c* :: *'a* **and** *b d* :: *'b*
have *Pair-Rep a b* = *Pair-Rep c d* ⟷ *a* = *c* ∧ *b* = *d*
by (*auto simp add*: *Pair-Rep-def fun-eq-iff*)
moreover have *Pair-Rep a b* ∈ *prod* **and** *Pair-Rep c d* ∈ *prod*
by (*auto simp add*: *prod-def*)
ultimately show *Pair a b* = *Pair c d* ⟷ *a* = *c* ∧ *b* = *d*
by (*simp add*: *Pair-def Abs-prod-inject*)
qed

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup ‹*Sign.mandatory-path old*›

old-rep-datatype *Pair*
by (*erule prod-cases*) (*rule prod.inject*)

setup ‹*Sign.parent-path*›

But erase the prefix for properties that are not generated by *free-constructors*.

setup ‹*Sign.mandatory-path prod*›

declare *old.prod.inject* [*iff del*]

lemmas *induct* = *old.prod.induct*
lemmas *inducts* = *old.prod.inducts*
lemmas *rec* = *old.prod.rec*
lemmas *simps* = *prod.inject prod.case prod.rec*

setup ‹*Sign.parent-path*›

declare *prod.case* [*nitpick-simp del*]
declare *old.prod.case-cong-weak* [*cong del*]
declare *prod.case-eq-if* [*mono*]
declare *prod.split* [*no-atp*]
declare *prod.split-asm* [*no-atp*]

prod.split could be declared as [*split*] done after the Splitter has been speeded up significantly; precompute the constants involved and don't do anything unless the current goal contains one of those constants.

14.3.2 Tuple syntax

Patterns – extends pre-defined type *pttrn* used in abstractions.

nonterminal *tuple-args* **and** *patterns*

open-bundle *tuple-syntax*
begin

syntax
-tuple :: *'a ⇒ tuple-args ⇒ 'a × 'b* (‹(‹*indent=1 notation=‹mixfix tuple›› '(-,/ -')*)›)
-tuple-arg :: *'a ⇒ tuple-args* (‹-›)
-tuple-args :: *'a ⇒ tuple-args ⇒ tuple-args* (‹-,/ -›)
-pattern :: *pttrn ⇒ patterns ⇒ pttrn* (‹(‹*open-block notation=‹pattern tuple›› '(-,/ -')*)›)
:: *pttrn ⇒ patterns* (‹-›)
-patterns :: *pttrn ⇒ patterns ⇒ patterns* (‹-,/ -›)
-unit :: *pttrn* (‹(‹*open-block notation=‹pattern unit›› '(')*)›)
syntax-consts
-pattern -patterns ⇌ *case-prod* **and**
-unit ⇌ *case-unit*
translations
(*x*, *y*) ⇌ *CONST Pair x y*
-pattern x y ⇌ *CONST Pair x y*
-patterns x y ⇌ *CONST Pair x y*
-tuple x (*-tuple-args y z*) ⇌ *-tuple x* (*-tuple-arg* (*-tuple y z*))
λ(*x*, *y*, *zs*). *b* ⇌ *CONST case-prod* (λ*x* (*y*, *zs*). *b*)
λ(*x*, *y*). *b* ⇌ *CONST case-prod* (λ*x y*. *b*)
-abs (*CONST Pair x y*) *t* ↽ λ(*x*, *y*). *t*
— This rule accommodates tuples in *case C* … (*x*, *y*) … ⇒ …: The (*x*, *y*) is parsed as *Pair x y* because it is *logic*, not *pttrn*.
λ(). *b* ⇌ *CONST case-unit b*
-abs (*CONST Unity*) *t* ↽ λ(). *t*

end

print *case-prod f* as *case-prod f* and *case-prod f* as *case-prod f*

typed-print-translation ‹
let
fun case-prod-guess-names-tr' - *T* [*Abs* (*x*, -, *Abs* -)] = *raise Match*
| *case-prod-guess-names-tr' ctxt T* [*Abs* (*x*, *xT*, *t*)] =
(*case* (*head-of t*) *of*
Const (***const-syntax***‹*case-prod*›, -) => *raise Match*
| - =>
let
val (- :: *yT* :: -) = *binder-types* (*domain-type T*) *handle Bind* => *raise Match*;
val (*y*, *t'*) = *Syntax-Trans.atomic-abs-tr' ctxt* (*y*, *yT*, *incr-boundvars 1*

```
t $ Bound 0);
            val (x', t'') = Syntax-Trans.atomic-abs-tr' ctxt (x, xT, t');
          in
            Syntax.const syntax-const‹-abs› $
              (Syntax.const syntax-const‹-pattern› $ x' $ y) $ t''
          end)
      | case-prod-guess-names-tr' ctxt T [t] =
          (case head-of t of
            Const (const-syntax‹case-prod›, -) => raise Match
          | - =>
            let
              val (xT :: yT :: -) = binder-types (domain-type T) handle Bind =>
raise Match;
              val (y, t') =
                Syntax-Trans.atomic-abs-tr' ctxt (y, yT, incr-boundvars 2 t $ Bound
1 $ Bound 0);
              val (x', t'') = Syntax-Trans.atomic-abs-tr' ctxt (x, xT, t');
            in
              Syntax.const syntax-const‹-abs› $
                (Syntax.const syntax-const‹-pattern› $ x' $ y) $ t''
            end)
      | case-prod-guess-names-tr' - - - = raise Match;
  in [(const-syntax‹case-prod›, case-prod-guess-names-tr')] end
›
```

Reconstruct pattern from (nested) *case-prod*s, avoiding eta-contraction of body; required for enclosing "let", if "let" does not avoid eta-contraction, which has been observed to occur.

print-translation ‹

```
  let
    fun case-prod-tr' ctxt [Abs (x, T, t as (Abs abs))] =
          (* case-prod (λx y. t) ⇒ λ(x, y) t *)
          let
            val (y, t') = Syntax-Trans.atomic-abs-tr' ctxt abs;
            val (x', t'') = Syntax-Trans.atomic-abs-tr' ctxt (x, T, t');
          in
            Syntax.const syntax-const‹-abs› $
              (Syntax.const syntax-const‹-pattern› $ x' $ y) $ t''
          end
      | case-prod-tr' ctxt [Abs (x, T, (s as Const (const-syntax‹case-prod›, -) $
t))] =
          (* case-prod (λx. (case-prod (λy z. t))) ⇒ λ(x, y, z). t *)
          let
            val Const (syntax-const‹-abs›, -) $
              (Const (syntax-const‹-pattern›, -) $ y $ z) $ t' =
                case-prod-tr' ctxt [t];
            val (x', t'') = Syntax-Trans.atomic-abs-tr' ctxt (x, T, t');
          in
            Syntax.const syntax-const‹-abs› $
```

```
          (Syntax.const syntax-const‹-pattern› $ x' $
            (Syntax.const syntax-const‹-patterns› $ y $ z)) $ t''
        end
    | case-prod-tr' ctxt [Const (const-syntax‹case-prod›, -) $ t] =
        (* case-prod (case-prod (λx y z. t)) ⇒ λ((x, y), z). t *)
        case-prod-tr' ctxt [(case-prod-tr' ctxt [t])]
          (* inner case-prod-tr' creates next pattern *)
    | case-prod-tr' ctxt [Const (syntax-const‹-abs›, -) $ x-y $ Abs abs] =
        (* case-prod (λpttrn z. t) ⇒ λ(pttrn, z). t *)
        let val (z, t) = Syntax-Trans.atomic-abs-tr' ctxt abs in
          Syntax.const syntax-const‹-abs› $
            (Syntax.const syntax-const‹-pattern› $ x-y $ z) $ t
        end
    | case-prod-tr' - - = raise Match;
  in [(const-syntax‹case-prod›, case-prod-tr')] end
›
```

14.3.3 Code generator setup

```
code-printing
  type-constructor prod ⇀
    (SML) infix 2 *
    and (OCaml) infix 2 *
    and (Haskell) !((-),/ (-))
    and (Scala) ((-),/ (-))
| constant Pair ⇀
    (SML) !((-),/ (-))
    and (OCaml) !((-),/ (-))
    and (Haskell) !((-),/ (-))
    and (Scala) !((-),/ (-))
| class-instance prod :: equal ⇀
    (Haskell) -
| constant HOL.equal :: 'a × 'b ⇒ 'a × 'b ⇒ bool ⇀
    (Haskell) infix 4 ==
| constant fst ⇀ (Haskell) fst
| constant snd ⇀ (Haskell) snd
```

14.3.4 Fundamental operations and properties

```
lemma Pair-inject: (a, b) = (a', b') ⟹ (a = a' ⟹ b = b' ⟹ R) ⟹ R
  by simp

lemma surj-pair [simp]: ∃x y. p = (x, y)
  by (cases p) simp

lemma fst-eqD: fst (x, y) = a ⟹ x = a
  by simp

lemma snd-eqD: snd (x, y) = a ⟹ y = a
  by simp
```

lemma *case-prod-unfold* [*nitpick-unfold*]: *case-prod* = (λ*c p*. *c* (*fst p*) (*snd p*))
by (*simp add*: *fun-eq-iff split*: *prod.split*)

lemma *case-prod-conv* [*simp*, *code*]: (*case* (*a*, *b*) *of* (*c*, *d*) ⇒ *f c d*) = *f a b*
by (*fact prod.case*)

lemmas *surjective-pairing* = *prod.collapse* [*symmetric*]

lemma *prod-eq-iff*: *s* = *t* ⟷ *fst s* = *fst t* ∧ *snd s* = *snd t*
by (*cases s*, *cases t*) *simp*

lemma *prod-eqI* [*intro?*]: *fst p* = *fst q* ⟹ *snd p* = *snd q* ⟹ *p* = *q*
by (*simp add*: *prod-eq-iff*)

lemma *case-prodI*: *f a b* ⟹ *case* (*a*, *b*) *of* (*c*, *d*) ⇒ *f c d*
by (*rule prod.case* [*THEN iffD2*])

lemma *case-prodD*: (*case* (*a*, *b*) *of* (*c*, *d*) ⇒ *f c d*) ⟹ *f a b*
by (*rule prod.case* [*THEN iffD1*])

lemma *case-prod-Pair* [*simp*]: *case-prod Pair* = *id*
by (*simp add*: *fun-eq-iff split*: *prod.split*)

lemma *case-prod-eta*: (λ(*x*, *y*). *f* (*x*, *y*)) = *f*
— Subsumes the old *split-Pair* when *f* is the identity function.
by (*simp add*: *fun-eq-iff split*: *prod.split*)

lemma *case-prod-comp*: (*case x of* (*a*, *b*) ⇒ (*f* ∘ *g*) *a b*) = *f* (*g* (*fst x*)) (*snd x*)
by (*cases x*) *simp*

lemma *The-case-prod*: *The* (*case-prod P*) = (*THE xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *case-prod-unfold*)

lemma *cond-case-prod-eta*: (⋀*x y*. *f x y* = *g* (*x*, *y*)) ⟹ (λ(*x*, *y*). *f x y*) = *g*
by (*simp add*: *case-prod-eta*)

lemma *split-paired-all* [*no-atp*]: (⋀*x*. *PROP P x*) ≡ (⋀*a b*. *PROP P* (*a*, *b*))
proof
fix *a b*
assume ⋀*x*. *PROP P x*
then show *PROP P* (*a*, *b*) **.**
next
fix *x*
assume ⋀*a b*. *PROP P* (*a*, *b*)
from ‹*PROP P* (*fst x*, *snd x*)› **show** *PROP P x* **by** *simp*
qed

The rule *split-paired-all* does not work with the Simplifier because it also affects premises in congrence rules, where this can lead to premises of the form $\bigwedge a\ b.\ \ldots\ =\ ?P(a,\ b)$ which cannot be solved by reflexivity.

lemmas *split-tupled-all* = *split-paired-all unit-all-eq2*

ML ‹
```
(* replace parameters of product type by individual component parameters *)
local (* filtering with exists-paired-all is an essential optimization *)
  fun exists-paired-all (Const (const-name ‹Pure.all›, -) $ Abs (-, T, t)) =
        can HOLogic.dest-prodT T orelse exists-paired-all t
    | exists-paired-all (t $ u) = exists-paired-all t orelse exists-paired-all u
    | exists-paired-all (Abs (-, -, t)) = exists-paired-all t
    | exists-paired-all - = false;
  val ss =
    HOL-basic-ss
    |> Simplifier.simpset-map context (
      Simplifier.add-simps @{thms split-paired-all unit-all-eq2 unit-abs-eta-conv}
      #> Simplifier.add-proc simproc ‹unit-eq›);
in
  fun split-all-tac ctxt = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then safe-full-simp-tac (put-simpset ss ctxt) i else no-tac);

  fun unsafe-split-all-tac ctxt = SUBGOAL (fn (t, i) =>
    if exists-paired-all t then full-simp-tac (put-simpset ss ctxt) i else no-tac);

  fun split-all ctxt th =
    if exists-paired-all (Thm.prop-of th)
    then full-simplify (put-simpset ss ctxt) th else th;
end;
```
›

setup ‹*map-theory-claset* (*fn ctxt => ctxt addSbefore* (*split-all-tac*, *split-all-tac*))›

lemma *split-paired-All* [*simp*, *no-atp*]: $(\forall x.\ P\ x) \longleftrightarrow (\forall a\ b.\ P\ (a,\ b))$
— [*iff*] is not a good idea because it makes *blast* loop
by *fast*

lemma *split-paired-Ex* [*simp*, *no-atp*]: $(\exists x.\ P\ x) \longleftrightarrow (\exists a\ b.\ P\ (a,\ b))$
by *fast*

lemma *split-paired-The* [*no-atp*]: $(THE\ x.\ P\ x) = (THE\ (a,\ b).\ P\ (a,\ b))$
— Can't be added to simpset: loops!
by (*simp add*: *case-prod-eta*)

Simplification procedure for *cond-case-prod-eta*. Using *case-prod-eta* as a rewrite rule is not general enough, and using *cond-case-prod-eta* directly would render some existing proofs very inefficient; similarly for *prod.case-eq-if*.

ML ‹

```
local
  val cond-case-prod-eta-ss =
    HOL-basic-ss
   |> Simplifier.simpset-map context (Simplifier.add-simps @{thms cond-case-prod-eta});
  fun Pair-pat k 0 (Bound m) = (m = k)
    | Pair-pat k i (Const (const-name ‹Pair›,  -) $ Bound m $ t) =
        i > 0 andalso m = k + i andalso Pair-pat k (i − 1) t
    | Pair-pat - - - = false;
  fun no-args k i (Abs (-, -, t)) = no-args (k + 1) i t
    | no-args k i (t $ u) = no-args k i t andalso no-args k i u
    | no-args k i (Bound m) = m < k orelse m > k + i
    | no-args - - - = true;
  fun split-pat tp i (Abs  (-, -, t)) = if tp 0 i t then SOME (i, t) else NONE
    | split-pat tp i (Const (const-name ‹case-prod›, -) $ Abs (-, -, t)) = split-pat
tp (i + 1) t
    | split-pat tp i - = NONE;
  fun metaeq ctxt lhs rhs = mk-meta-eq (Goal.prove ctxt [] []
        (HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs)))
        (K (simp-tac (put-simpset cond-case-prod-eta-ss ctxt) 1)));

  fun beta-term-pat k i (Abs (-, -, t)) = beta-term-pat (k + 1) i t
    | beta-term-pat k i (t $ u) =
        Pair-pat k i (t $ u) orelse (beta-term-pat k i t andalso beta-term-pat k i u)
    | beta-term-pat k i t = no-args k i t;
  fun eta-term-pat k i (f $ arg) = no-args k i f andalso Pair-pat k i arg
    | eta-term-pat - - - = false;
  fun subst arg k i (Abs (x, T, t)) = Abs (x, T, subst arg (k+1) i t)
    | subst arg k i (t $ u) =
        if Pair-pat k i (t $ u) then incr-boundvars k arg
        else (subst arg k i t $ subst arg k i u)
    | subst arg k i t = t;
in
  fun beta-proc ctxt (s as Const (const-name ‹case-prod›, -) $ Abs (-, -, t) $ arg)
=
        (case split-pat beta-term-pat 1 t of
          SOME (i, f) => SOME (metaeq ctxt s (subst arg 0 i f))
        | NONE => NONE)
    | beta-proc - - = NONE;
  fun eta-proc ctxt (s as Const (const-name ‹case-prod›, -) $ Abs (-, -, t)) =
        (case split-pat eta-term-pat 1 t of
          SOME (-, ft) => SOME (metaeq ctxt s (let val f $ - = ft in f end))
        | NONE => NONE)
    | eta-proc - - = NONE;
end;
›
simproc-setup case-prod-beta (case-prod f z) =
  ‹K (fn ctxt => fn ct => beta-proc ctxt (Thm.term-of ct))›
simproc-setup case-prod-eta (case-prod f) =
  ‹K (fn ctxt => fn ct => eta-proc ctxt (Thm.term-of ct))›
```

lemma *case-prod-beta'*: $(\lambda(x,y).\ f\ x\ y) = (\lambda x.\ f\ (fst\ x)\ (snd\ x))$
by (*auto simp*: *fun-eq-iff*)

case-prod used as a logical connective or set former.

These rules are for use with *blast*; could instead call *simp* using *prod.split* as rewrite.

lemma *case-prodI2*:
$\bigwedge p.\ (\bigwedge a\ b.\ p = (a,\ b) \Longrightarrow c\ a\ b) \Longrightarrow case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b$
by (*simp add*: *split-tupled-all*)

lemma *case-prodI2'*:
$\bigwedge p.\ (\bigwedge a\ b.\ (a,\ b) = p \Longrightarrow c\ a\ b\ x) \Longrightarrow (case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)\ x$
by (*simp add*: *split-tupled-all*)

lemma *case-prodE* [*elim*!]:
$(case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b) \Longrightarrow (\bigwedge x\ y.\ p = (x,\ y) \Longrightarrow c\ x\ y \Longrightarrow Q) \Longrightarrow Q$
by (*induct p*) *simp*

lemma *case-prodE'* [*elim*!]:
$(case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)\ z \Longrightarrow (\bigwedge x\ y.\ p = (x,\ y) \Longrightarrow c\ x\ y\ z \Longrightarrow Q) \Longrightarrow Q$
by (*induct p*) *simp*

lemma *case-prodE2*:
assumes *q*: $Q\ (case\ z\ of\ (a,\ b) \Rightarrow P\ a\ b)$
and *r*: $\bigwedge x\ y.\ z = (x,\ y) \Longrightarrow Q\ (P\ x\ y) \Longrightarrow R$
shows *R*
proof (*rule r*)
show $z = (fst\ z,\ snd\ z)$ **by** *simp*
then show $Q\ (P\ (fst\ z)\ (snd\ z))$
using *q* **by** (*simp add*: *case-prod-unfold*)
qed

lemma *case-prodD'*: $(case\ (a,\ b)\ of\ (c,\ d) \Rightarrow R\ c\ d)\ c \Longrightarrow R\ a\ b\ c$
by *simp*

lemma *mem-case-prodI*: $z \in c\ a\ b \Longrightarrow z \in (case\ (a,\ b)\ of\ (d,\ e) \Rightarrow c\ d\ e)$
by *simp*

lemma *mem-case-prodI2* [*intro*!]:
$\bigwedge p.\ (\bigwedge a\ b.\ p = (a,\ b) \Longrightarrow z \in c\ a\ b) \Longrightarrow z \in (case\ p\ of\ (a,\ b) \Rightarrow c\ a\ b)$
by (*simp only*: *split-tupled-all*) *simp*

declare *mem-case-prodI* [*intro*!] — postponed to maintain traditional declaration order!
declare *case-prodI2'* [*intro*!] — postponed to maintain traditional declaration order!
declare *case-prodI2* [*intro*!] — postponed to maintain traditional declaration order!

declare *case-prodI* [*intro*!] — postponed to maintain traditional declaration order!

lemma *mem-case-prodE* [*elim*!]:
assumes $z \in$ *case-prod c p*
obtains *x y* **where** $p = (x, y)$ **and** $z \in c\ x\ y$
using *assms* **by** (*rule case-prodE2*)

ML ‹
local (∗ *filtering with exists-p-split is an essential optimization* ∗)
fun exists-p-split (*Const* (***const-name*** ‹*case-prod*›,-) $ - $ (*Const* (***const-name*** ‹*Pair*›,-)$-$-))
= *true*
| *exists-p-split* (*t* $ *u*) = *exists-p-split t orelse exists-p-split u*
| *exists-p-split* (*Abs* (-, -, *t*)) = *exists-p-split t*
| *exists-p-split* - = *false*;
in
fun split-conv-tac ctxt = *SUBGOAL* (*fn* (*t*, *i*) =>
if exists-p-split t
then
safe-full-simp-tac
(*put-simpset HOL-basic-ss ctxt* |> *Simplifier.add-simps* @{*thms case-prod-conv*})
i
else no-tac);
end;
›

setup ‹*map-theory-claset* (*fn ctxt* => *ctxt addSbefore* (*split-conv-tac*, *split-conv-tac*))›

lemma *split-eta-SetCompr* [*simp*, *no-atp*]: $(\lambda u.\ \exists x\ y.\ u = (x, y) \wedge P\ (x, y)) = P$
by (*rule ext*) *fast*

lemma *split-eta-SetCompr2* [*simp*, *no-atp*]: $(\lambda u.\ \exists x\ y.\ u = (x, y) \wedge P\ x\ y) =$ *case-prod P*
by (*rule ext*) *fast*

lemma *split-part* [*simp*]: $(\lambda(a,b).\ P \wedge Q\ a\ b) = (\lambda ab.\ P \wedge$ *case-prod Q ab*)
— Allows simplifications of nested splits in case of independent predicates.
by (*rule ext*) *blast*

lemma *split-comp-eq*:
fixes *f* :: $'a \Rightarrow 'b \Rightarrow 'c$
and *g* :: $'d \Rightarrow 'a$
shows $(\lambda u.\ f\ (g\ (fst\ u))\ (snd\ u)) =$ *case-prod* $(\lambda x.\ f\ (g\ x))$
by (*rule ext*) *auto*

lemma *pair-imageI* [*intro*]: $(a, b) \in A \Longrightarrow f\ a\ b \in (\lambda(a, b).\ f\ a\ b)$ ‘ *A*
by (*rule image-eqI* [**where** $x = (a, b)$]) *auto*

lemma *Collect-const-case-prod*[*simp*]: $\{(a,b).\ P\} = (\mathit{if}\ P\ \mathit{then}\ \mathit{UNIV}\ \mathit{else}\ \{\})$
by *auto*

lemma *The-split-eq* [*simp*]: $(\mathit{THE}\ (x',y').\ x = x' \wedge y = y') = (x,\ y)$
by *blast*

lemma *case-prod-beta*: $\mathit{case\text{-}prod}\ f\ p = f\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)$
by (*fact prod.case-eq-if*)

lemma *prod-cases3* [*cases type*]:
obtains (*fields*) $a\ b\ c$ **where** $y = (a,\ b,\ c)$
proof (*cases y*)
case (*Pair a b*)
with *that* **show** *?thesis*
by (*cases b*) *blast*
qed

lemma *prod-induct3* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c.\ P\ (a,\ b,\ c)) \Longrightarrow P\ x$
by (*cases x*) *blast*

lemma *prod-cases4* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d$ **where** $y = (a,\ b,\ c,\ d)$
proof (*cases y*)
case (*fields a b c*)
with *that* **show** *?thesis*
by (*cases c*) *blast*
qed

lemma *prod-induct4* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c\ d.\ P\ (a,\ b,\ c,\ d)) \Longrightarrow P\ x$
by (*cases x*) *blast*

lemma *prod-cases5* [*cases type*]:
obtains (*fields*) $a\ b\ c\ d\ e$ **where** $y = (a,\ b,\ c,\ d,\ e)$
proof (*cases y*)
case (*fields a b c d*)
with *that* **show** *?thesis*
by (*cases d*) *blast*
qed

lemma *prod-induct5* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c\ d\ e.\ P\ (a,\ b,\ c,\ d,\ e)) \Longrightarrow P\ x$
by (*cases x*) *blast*

lemma *prod-cases6* [*cases type*]:

obtains (*fields*) *a b c d e f* **where** $y = (a, b, c, d, e, f)$
proof (*cases y*)
case (*fields a b c d e*)
with *that* **show** *?thesis*
by (*cases e*) *blast*
qed

lemma *prod-induct6* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c\ d\ e\ f.\ P\ (a, b, c, d, e, f)) \Longrightarrow P\ x$
by (*cases x*) *blast*

lemma *prod-cases7* [*cases type*]:
obtains (*fields*) *a b c d e f g* **where** $y = (a, b, c, d, e, f, g)$
proof (*cases y*)
case (*fields a b c d e f*)
with *that* **show** *?thesis*
by (*cases f*) *blast*
qed

lemma *prod-induct7* [*case-names fields*, *induct type*]:
$(\bigwedge a\ b\ c\ d\ e\ f\ g.\ P\ (a, b, c, d, e, f, g)) \Longrightarrow P\ x$
by (*cases x*) *blast*

definition *internal-case-prod* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'c$
where *internal-case-prod* $\equiv$ *case-prod*

lemma *internal-case-prod-conv*: *internal-case-prod c* $(a, b) = c\ a\ b$
by (*simp only*: *internal-case-prod-def case-prod-conv*)

ML-file ‹*Tools/split-rule.ML*›

hide-const *internal-case-prod*

14.3.5 Derived operations

definition *curry* :: $('a \times 'b \Rightarrow 'c) \Rightarrow 'a \Rightarrow 'b \Rightarrow 'c$
where *curry* $= (\lambda c\ x\ y.\ c\ (x, y))$

lemma *curry-conv* [*simp*, *code*]: *curry f a b* $= f\ (a, b)$
by (*simp add*: *curry-def*)

lemma *curryI* [*intro*!]: $f\ (a, b) \Longrightarrow$ *curry f a b*
by (*simp add*: *curry-def*)

lemma *curryD* [*dest*!]: *curry f a b* $\Longrightarrow f\ (a, b)$
by (*simp add*: *curry-def*)

lemma *curryE*: *curry f a b* $\Longrightarrow (f\ (a, b) \Longrightarrow Q) \Longrightarrow Q$

by (*simp add*: *curry-def*)

lemma *curry-case-prod* [*simp*]: *curry* (*case-prod f*) = *f*
by (*simp add*: *curry-def case-prod-unfold*)

lemma *case-prod-curry* [*simp*]: *case-prod* (*curry f*) = *f*
by (*simp add*: *curry-def case-prod-unfold*)

lemma *curry-K*: *curry* (λ*x*. *c*) = (λ*x y*. *c*)
by (*simp add*: *fun-eq-iff*)

The composition-uncurry combinator.

definition *scomp* :: ('*a* ⇒ '*b* × '*c*) ⇒ ('*b* ⇒ '*c* ⇒ '*d*) ⇒ '*a* ⇒ '*d* (**infixl** ‹○→› *60*)
where *f* ○→ *g* = (λ*x*. *case-prod g* (*f x*))

no-notation *scomp* (**infixl** ‹○→› *60*)

bundle *state-combinator-syntax*
begin
notation *fcomp* (**infixl** ‹○>› *60*)
notation *scomp* (**infixl** ‹○→› *60*)
end

context
includes *state-combinator-syntax*
begin

lemma *scomp-unfold*: (○→) = (λ*f g x*. *g* (*fst* (*f x*)) (*snd* (*f x*)))
by (*simp add*: *fun-eq-iff scomp-def case-prod-unfold*)

lemma *scomp-apply* [*simp*]: (*f* ○→ *g*) *x* = *case-prod g* (*f x*)
by (*simp add*: *scomp-unfold case-prod-unfold*)

lemma *Pair-scomp*: *Pair x* ○→ *f* = *f x*
by (*simp add*: *fun-eq-iff*)

lemma *scomp-Pair*: *x* ○→ *Pair* = *x*
by (*simp add*: *fun-eq-iff*)

lemma *scomp-scomp*: (*f* ○→ *g*) ○→ *h* = *f* ○→ (λ*x*. *g x* ○→ *h*)
by (*simp add*: *fun-eq-iff scomp-unfold*)

lemma *scomp-fcomp*: (*f* ○→ *g*) ○> *h* = *f* ○→ (λ*x*. *g x* ○> *h*)
by (*simp add*: *fun-eq-iff scomp-unfold fcomp-def*)

lemma *fcomp-scomp*: (*f* ○> *g*) ○→ *h* = *f* ○> (*g* ○→ *h*)
by (*simp add*: *fun-eq-iff scomp-unfold*)

end

code-printing
  constant *scomp* ⇀ (*Eval*) **infixl** *3* #−>

map-prod — action of the product functor upon functions.

definition *map-prod* :: ('*a* ⇒ '*c*) ⇒ ('*b* ⇒ '*d*) ⇒ '*a* × '*b* ⇒ '*c* × '*d*
  where *map-prod f g* = (λ(*x*, *y*). (*f x*, *g y*))

lemma *map-prod-simp* [*simp*, *code*]: *map-prod f g* (*a*, *b*) = (*f a*, *g b*)
  by (*simp add*: *map-prod-def*)

functor *map-prod*: *map-prod*
  by (*auto simp add*: *split-paired-all*)

lemma *fst-map-prod* [*simp*]: *fst* (*map-prod f g x*) = *f* (*fst x*)
  by (*cases x*) *simp-all*

lemma *snd-map-prod* [*simp*]: *snd* (*map-prod f g x*) = *g* (*snd x*)
  by (*cases x*) *simp-all*

lemma *fst-comp-map-prod* [*simp*]: *fst* ∘ *map-prod f g* = *f* ∘ *fst*
  by (*rule ext*) *simp-all*

lemma *snd-comp-map-prod* [*simp*]: *snd* ∘ *map-prod f g* = *g* ∘ *snd*
  by (*rule ext*) *simp-all*

lemma *map-prod-compose*: *map-prod* (*f1* ∘ *f2*) (*g1* ∘ *g2*) = (*map-prod f1 g1* ∘ *map-prod f2 g2*)
  by (*rule ext*) (*simp add*: *map-prod.compositionality comp-def*)

lemma *map-prod-ident* [*simp*]: *map-prod* (λ*x*. *x*) (λ*y*. *y*) = (λ*z*. *z*)
  by (*rule ext*) (*simp add*: *map-prod.identity*)

lemma *map-prod-imageI* [*intro*]: (*a*, *b*) ∈ *R* ⟹ (*f a*, *g b*) ∈ *map-prod f g* ‘ *R*
  by (*rule image-eqI*) *simp-all*

lemma *prod-fun-imageE* [*elim*!]:
  assumes *major*: *c* ∈ *map-prod f g* ‘ *R*
    and *cases*: ⋀*x y*. *c* = (*f x*, *g y*) ⟹ (*x*, *y*) ∈ *R* ⟹ *P*
  shows *P*
proof (*rule major* [*THEN imageE*])
  fix *x*
  assume *c* = *map-prod f g x x* ∈ *R*
  then show *P*
    using *cases* **by** (*cases x*) *simp*
qed

definition *apfst* :: ('*a* ⇒ '*c*) ⇒ '*a* × '*b* ⇒ '*c* × '*b*

where *apfst f = map-prod f id*

definition *apsnd* :: $('b \Rightarrow 'c) \Rightarrow 'a \times 'b \Rightarrow 'a \times 'c$
where *apsnd f = map-prod id f*

lemma *apfst-conv* [*simp*, *code*]: *apfst f (x, y) = (f x, y)*
by (*simp add*: *apfst-def*)

lemma *apsnd-conv* [*simp*, *code*]: *apsnd f (x, y) = (x, f y)*
by (*simp add*: *apsnd-def*)

lemma *fst-apfst* [*simp*]: *fst (apfst f x) = f (fst x)*
by (*cases x*) *simp*

lemma *fst-comp-apfst* [*simp*]: *fst ∘ apfst f = f ∘ fst*
by (*simp add*: *fun-eq-iff*)

lemma *fst-apsnd* [*simp*]: *fst (apsnd f x) = fst x*
by (*cases x*) *simp*

lemma *fst-comp-apsnd* [*simp*]: *fst ∘ apsnd f = fst*
by (*simp add*: *fun-eq-iff*)

lemma *snd-apfst* [*simp*]: *snd (apfst f x) = snd x*
by (*cases x*) *simp*

lemma *snd-comp-apfst* [*simp*]: *snd ∘ apfst f = snd*
by (*simp add*: *fun-eq-iff*)

lemma *snd-apsnd* [*simp*]: *snd (apsnd f x) = f (snd x)*
by (*cases x*) *simp*

lemma *snd-comp-apsnd* [*simp*]: *snd ∘ apsnd f = f ∘ snd*
by (*simp add*: *fun-eq-iff*)

lemma *apfst-compose*: *apfst f (apfst g x) = apfst (f ∘ g) x*
by (*cases x*) *simp*

lemma *apsnd-compose*: *apsnd f (apsnd g x) = apsnd (f ∘ g) x*
by (*cases x*) *simp*

lemma *apfst-apsnd* [*simp*]: *apfst f (apsnd g x) = (f (fst x), g (snd x))*
by (*cases x*) *simp*

lemma *apsnd-apfst* [*simp*]: *apsnd f (apfst g x) = (g (fst x), f (snd x))*
by (*cases x*) *simp*

lemma *apfst-id* [*simp*]: *apfst id = id*
by (*simp add*: *fun-eq-iff*)

lemma *apsnd-id* [*simp*]: *apsnd id = id*
by (*simp add*: *fun-eq-iff*)

lemma *apfst-eq-conv* [*simp*]: *apfst f x = apfst g x ⟷ f (fst x) = g (fst x)*
by (*cases x*) *simp*

lemma *apsnd-eq-conv* [*simp*]: *apsnd f x = apsnd g x ⟷ f (snd x) = g (snd x)*
by (*cases x*) *simp*

lemma *apsnd-apfst-commute*: *apsnd f (apfst g p) = apfst g (apsnd f p)*
by *simp*

context
begin

local-setup ‹*Local-Theory.map-background-naming* (*Name-Space.mandatory-path prod*)›

definition *swap* :: *'a × 'b ⇒ 'b × 'a*
where *swap p* = (*snd p*, *fst p*)

end

lemma *swap-simp* [*simp*]: *prod.swap* (*x*, *y*) = (*y*, *x*)
by (*simp add*: *prod.swap-def*)

lemma *swap-swap* [*simp*]: *prod.swap* (*prod.swap p*) = *p*
by (*cases p*) *simp*

lemma *swap-comp-swap* [*simp*]: *prod.swap ∘ prod.swap = id*
by (*simp add*: *fun-eq-iff*)

lemma *pair-in-swap-image* [*simp*]: (*y*, *x*) ∈ *prod.swap ' A ⟷* (*x*, *y*) ∈ *A*
by (*auto intro*!: *image-eqI*)

lemma *inj-swap* [*simp*]: *inj-on prod.swap A*
by (*rule inj-onI*) *auto*

lemma *swap-inj-on*: *inj-on* (λ(*i*, *j*). (*j*, *i*)) *A*
by (*rule inj-onI*) *auto*

lemma *surj-swap* [*simp*]: *surj prod.swap*
by (*rule surjI* [*of - prod.swap*]) *simp*

lemma *bij-swap* [*simp*]: *bij prod.swap*
by (*simp add*: *bij-def*)

lemma *case-swap* [*simp*]: (*case prod.swap p of* (*y*, *x*) ⇒ *f x y*) = (*case p of* (*x*, *y*)

$\Rightarrow f\ x\ y$)
by (*cases p*) *simp*

lemma *fst-swap* [*simp*]: *fst* (*prod.swap x*) = *snd x*
by (*cases x*) *simp*

lemma *snd-swap* [*simp*]: *snd* (*prod.swap x*) = *fst x*
by (*cases x*) *simp*

lemma *split-pairs*: (*A*,*B*) = *X* $\longleftrightarrow$ *fst X* = *A* $\wedge$ *snd X* = *B*
and *split-pairs2*: *X* = (*A*,*B*) $\longleftrightarrow$ *fst X* = *A* $\wedge$ *snd X* = *B*
by *auto*

Disjoint union of a family of sets – Sigma.

definition *Sigma* :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b set*) $\Rightarrow$ (*'a* $\times$ *'b*) *set*
where *Sigma A B* $\equiv \bigcup x{\in}A.\ \bigcup y{\in}B\ x.\ \{Pair\ x\ y\}$

context
begin
qualified abbreviation *Times* :: *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ (*'a* $\times$ *'b*) *set* (**infixr** ‹×› *80*)
where *A* $\times$ *B* $\equiv$ *Sigma A* (λ-. *B*)
end

bundle *set-product-syntax*
begin
notation *Product-Type.Times* (**infixr** ‹×› *80*)
end

syntax
-Sigma :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ (*'a* $\times$ *'b*) *set*
(‹(‹*indent=3 notation=‹binder SIGMA››SIGMA* -:-./ -)› [*0*, *0*, *10*] *10*)
syntax-consts
-Sigma $\rightleftharpoons$ *Sigma*
translations
SIGMA x:A. B $\rightleftharpoons$ *CONST Sigma A* (λ*x*. *B*)

lemma *SigmaI* [*intro*!]: *a* $\in$ *A* $\Longrightarrow$ *b* $\in$ *B a* $\Longrightarrow$ (*a*, *b*) $\in$ *Sigma A B*
unfolding *Sigma-def* **by** *blast*

lemma *SigmaE* [*elim*!]: *c* $\in$ *Sigma A B* $\Longrightarrow$ ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in B\ x \Longrightarrow c$ = (*x*, *y*) $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
— The general elimination rule.
unfolding *Sigma-def* **by** *blast*

Elimination of (*a*, *b*) $\in$ *A* $\times$ *B* – introduces no eigenvariables.

lemma *SigmaD1*: (*a*, *b*) $\in$ *Sigma A B* $\Longrightarrow$ *a* $\in$ *A*
by *blast*

lemma *SigmaD2*: (*a*, *b*) $\in$ *Sigma A B* $\Longrightarrow$ *b* $\in$ *B a*

by *blast*

lemma *SigmaE2*: $(a, b) \in Sigma\ A\ B \Longrightarrow (a \in A \Longrightarrow b \in B\ a \Longrightarrow P) \Longrightarrow P$
by *blast*

lemma *Sigma-cong*: $A = B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow C\ x = D\ x) \Longrightarrow (SIGMA\ x{:}A.\ C\ x) = (SIGMA\ x{:}B.\ D\ x)$
by *auto*

lemma *Sigma-mono*: $A \subseteq C \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq D\ x) \Longrightarrow Sigma\ A\ B \subseteq Sigma\ C\ D$
by *blast*

lemma *Sigma-empty1* [*simp*]: $Sigma\ \{\}\ B = \{\}$
by *blast*

lemma *Sigma-empty2* [*simp*]: $A \times \{\} = \{\}$
by *blast*

lemma *Sigma-insert*: $Sigma\ (insert\ x\ A)\ B = (Pair\ x)\ `\ B\ x \cup Sigma\ A\ B$
by *auto*

lemma *UNIV-Times-UNIV* [*simp*]: $UNIV \times UNIV = UNIV$
by *auto*

lemma *Compl-Times-UNIV1* [*simp*]: $-\ (UNIV \times A) = UNIV \times (-A)$
by *auto*

lemma *Compl-Times-UNIV2* [*simp*]: $-\ (A \times UNIV) = (-A) \times UNIV$
by *auto*

lemma *mem-Sigma-iff* [*iff*]: $(a, b) \in Sigma\ A\ B \longleftrightarrow a \in A \wedge b \in B\ a$
by *blast*

lemma *mem-Times-iff*: $x \in A \times B \longleftrightarrow fst\ x \in A \wedge snd\ x \in B$
by (*induct x*) *simp*

lemma *Sigma-empty-iff*: $(SIGMA\ i{:}I.\ X\ i) = \{\} \longleftrightarrow (\forall i{\in}I.\ X\ i = \{\})$
by *auto*

lemma *Times-subset-cancel2*: $x \in C \Longrightarrow A \times C \subseteq B \times C \longleftrightarrow A \subseteq B$
by *blast*

lemma *Times-eq-cancel2*: $x \in C \Longrightarrow A \times C = B \times C \longleftrightarrow A = B$
by (*blast elim*: *equalityE*)

lemma *Collect-case-prod-Sigma*: $\{(x, y).\ P\ x \wedge Q\ x\ y\} = (SIGMA\ x{:}Collect\ P.\ Collect\ (Q\ x))$
by *blast*

lemma *Collect-case-prod* [*simp*]: $\{(a,\ b).\ P\ a \wedge Q\ b\} = Collect\ P \times Collect\ Q$
by (*fact Collect-case-prod-Sigma*)

lemma *Collect-case-prodD*: $x \in Collect\ (case\text{-}prod\ A) \Longrightarrow A\ (fst\ x)\ (snd\ x)$
by *auto*

lemma *Collect-case-prod-mono*: $A \leq B \Longrightarrow Collect\ (case\text{-}prod\ A) \subseteq Collect\ (case\text{-}prod\ B)$
by *auto* (*auto elim*!: *le-funE*)

lemma *Collect-split-mono-strong*:
$X = fst\ `\ A \Longrightarrow Y = snd\ `\ A \Longrightarrow \forall\, a{\in}X.\ \forall\, b \in Y.\ P\ a\ b \longrightarrow Q\ a\ b$
$\Longrightarrow A \subseteq Collect\ (case\text{-}prod\ P) \Longrightarrow A \subseteq Collect\ (case\text{-}prod\ Q)$
by *fastforce*

lemma *UN-Times-distrib*: $(\bigcup (a,\ b){\in}A \times B.\ E\ a \times F\ b) = \bigcup (E\ `\ A) \times \bigcup (F\ `\ B)$
— Suggested by Pierre Chartier
by *blast*

lemma *split-paired-Ball-Sigma* [*simp*, *no-atp*]: $(\forall\, z{\in}Sigma\ A\ B.\ P\ z) \longleftrightarrow (\forall\, x{\in}A.\ \forall\, y{\in}B\ x.\ P\ (x,\ y))$
by *blast*

lemma *split-paired-Bex-Sigma* [*simp*, *no-atp*]: $(\exists\, z{\in}Sigma\ A\ B.\ P\ z) \longleftrightarrow (\exists\, x{\in}A.\ \exists\, y{\in}B\ x.\ P\ (x,\ y))$
by *blast*

lemma *Sigma-Un-distrib1*: $Sigma\ (I \cup J)\ C = Sigma\ I\ C \cup Sigma\ J\ C$
by *blast*

lemma *Sigma-Un-distrib2*: $(SIGMA\ i{:}I.\ A\ i \cup B\ i) = Sigma\ I\ A \cup Sigma\ I\ B$
by *blast*

lemma *Sigma-Int-distrib1*: $Sigma\ (I \cap J)\ C = Sigma\ I\ C \cap Sigma\ J\ C$
by *blast*

lemma *Sigma-Int-distrib2*: $(SIGMA\ i{:}I.\ A\ i \cap B\ i) = Sigma\ I\ A \cap Sigma\ I\ B$
by *blast*

lemma *Sigma-Diff-distrib1*: $Sigma\ (I - J)\ C = Sigma\ I\ C - Sigma\ J\ C$
by *blast*

lemma *Sigma-Diff-distrib2*: $(SIGMA\ i{:}I.\ A\ i - B\ i) = Sigma\ I\ A - Sigma\ I\ B$
by *blast*

lemma *Sigma-Union*: $Sigma\ (\bigcup X)\ B = (\bigcup A{\in}X.\ Sigma\ A\ B)$
by *blast*

lemma *Pair-vimage-Sigma*: *Pair x −‘ Sigma A f* = (*if x* ∈ *A then f x else* {})
by *auto*

Non-dependent versions are needed to avoid the need for higher-order matching, especially when the rules are re-oriented.

lemma *Times-Un-distrib1*: (*A* ∪ *B*) × *C* = *A* × *C* ∪ *B* × *C*
by (*fact Sigma-Un-distrib1*)

lemma *Times-Int-distrib1*: (*A* ∩ *B*) × *C* = *A* × *C* ∩ *B* × *C*
by (*fact Sigma-Int-distrib1*)

lemma *Times-Diff-distrib1*: (*A* − *B*) × *C* = *A* × *C* − *B* × *C*
by (*fact Sigma-Diff-distrib1*)

lemma *Times-empty* [*simp*]: *A* × *B* = {} ⟷ *A* = {} ∨ *B* = {}
by *auto*

lemma *times-subset-iff*: *A* × *C* ⊆ *B* × *D* ⟷ *A*={} ∨ *C*={} ∨ *A* ⊆ *B* ∧ *C* ⊆ *D*
by *blast*

lemma *times-eq-iff*: *A* × *B* = *C* × *D* ⟷ *A* = *C* ∧ *B* = *D* ∨ (*A* = {} ∨ *B* = {}) ∧ (*C* = {} ∨ *D* = {})
by *auto*

lemma *fst-image-times* [*simp*]: *fst* ‘ (*A* × *B*) = (*if B* = {} *then* {} *else A*)
by *force*

lemma *snd-image-times* [*simp*]: *snd* ‘ (*A* × *B*) = (*if A* = {} *then* {} *else B*)
by *force*

lemma *fst-image-Sigma*: *fst* ‘ (*Sigma A B*) = {*x* ∈ *A*. *B*(*x*) ≠ {}}
by *force*

lemma *snd-image-Sigma*: *snd* ‘ (*Sigma A B*) = (⋃ *x* ∈ *A*. *B x*)
by *force*

lemma *vimage-fst*: *fst* −‘ *A* = *A* × *UNIV*
by *auto*

lemma *vimage-snd*: *snd* −‘ *A* = *UNIV* × *A*
by *auto*

lemma *insert-Times-insert* [*simp*]:
insert a A × *insert b B* = *insert* (*a*,*b*) (*A* × *insert b B* ∪ {*a*} × *B*)
by *blast*

lemma *sing-Times-sing*: {*x*}×{*y*} = {(*x*,*y*)}
by *simp*

lemma *vimage-Times*: $f -` (A \times B) = (fst \circ f) -` A \cap (snd \circ f) -` B$
proof (*rule set-eqI*)
  show $x \in f -` (A \times B) \longleftrightarrow x \in (fst \circ f) -` A \cap (snd \circ f) -` B$ **for** x
    by (*cases f x*) (*auto split*: *prod.split*)
qed

lemma *Times-Int-Times*: $A \times B \cap C \times D = (A \cap C) \times (B \cap D)$
  by *auto*

lemma *image-paired-Times*:
  $(\lambda(x,y).\ (f\ x,\ g\ y))\ `\ (A \times B) = (f\ `\ A) \times (g\ `\ B)$
  by *auto*

lemma *Times-insert-right*: $A \times insert\ y\ B = (\lambda x.\ (x,\ y))\ `\ A \cup A \times B$
  by *auto*

lemma *Times-insert-left*: $insert\ x\ A \times B = (\lambda y.\ (x,\ y))\ `\ B \cup A \times B$
  by *auto*

lemma *product-swap*: $prod.swap\ `\ (A \times B) = B \times A$
  by (*auto simp add*: *set-eq-iff*)

lemma *swap-product*: $(\lambda(i,\ j).\ (j,\ i))\ `\ (A \times B) = B \times A$
  by (*auto simp add*: *set-eq-iff*)

lemma *image-split-eq-Sigma*: $(\lambda x.\ (f\ x,\ g\ x))\ `\ A = Sigma\ (f\ `\ A)\ (\lambda x.\ g\ `\ (f -` \{x\} \cap A))$
proof (*safe intro*!: *imageI*)
  fix *a b*
  assume $*$: $a \in A$ $b \in A$ **and** *eq*: $f\ a = f\ b$
  show $(f\ b,\ g\ a) \in (\lambda x.\ (f\ x,\ g\ x))\ `\ A$
    using $*$ *eq*[*symmetric*] **by** *auto*
qed *simp-all*

lemma *subset-fst-snd*: $A \subseteq (fst\ `\ A \times snd\ `\ A)$
  by *force*

lemma *inj-on-apfst* [*simp*]: $inj\text{-}on\ (apfst\ f)\ (A \times UNIV) \longleftrightarrow inj\text{-}on\ f\ A$
  by (*auto simp add*: *inj-on-def*)

lemma *inj-apfst* [*simp*]: $inj\ (apfst\ f) \longleftrightarrow inj\ f$
  using *inj-on-apfst*[*of f UNIV*] **by** *simp*

lemma *inj-on-apsnd* [*simp*]: $inj\text{-}on\ (apsnd\ f)\ (UNIV \times A) \longleftrightarrow inj\text{-}on\ f\ A$
  by (*auto simp add*: *inj-on-def*)

lemma *inj-apsnd* [*simp*]: $inj\ (apsnd\ f) \longleftrightarrow inj\ f$
  using *inj-on-apsnd*[*of f UNIV*] **by** *simp*

context
begin

qualified definition *product* :: *'a set* ⇒ *'b set* ⇒ (*'a* × *'b*) *set*
  where [*code-abbrev*]: *product A B* = *A* × *B*

lemma *member-product*: *x* ∈ *Product-Type.product A B* ⟷ *x* ∈ *A* × *B*
  by (*simp add*: *product-def*)

end

The following *map-prod* lemmas are due to Joachim Breitner:

lemma *map-prod-inj-on*:
  assumes *inj-on f A*
    and *inj-on g B*
  shows *inj-on* (*map-prod f g*) (*A* × *B*)
proof (*rule inj-onI*)
  fix *x* :: *'a* × *'c*
  fix *y* :: *'a* × *'c*
  assume *x* ∈ *A* × *B*
  then have *fst x* ∈ *A* **and** *snd x* ∈ *B* **by** *auto*
  assume *y* ∈ *A* × *B*
  then have *fst y* ∈ *A* **and** *snd y* ∈ *B* **by** *auto*
  assume *map-prod f g x* = *map-prod f g y*
  then have *fst* (*map-prod f g x*) = *fst* (*map-prod f g y*) **by** *auto*
  then have *f* (*fst x*) = *f* (*fst y*) **by** (*cases x*, *cases y*) *auto*
  with ‹*inj-on f A*› **and** ‹*fst x* ∈ *A*› **and** ‹*fst y* ∈ *A*› **have** *fst x* = *fst y*
    by (*auto dest*: *inj-onD*)
  moreover from ‹*map-prod f g x* = *map-prod f g y*›
  have *snd* (*map-prod f g x*) = *snd* (*map-prod f g y*) **by** *auto*
  then have *g* (*snd x*) = *g* (*snd y*) **by** (*cases x*, *cases y*) *auto*
  with ‹*inj-on g B*› **and** ‹*snd x* ∈ *B*› **and** ‹*snd y* ∈ *B*› **have** *snd x* = *snd y*
    by (*auto dest*: *inj-onD*)
  ultimately show *x* = *y* **by** (*rule prod-eqI*)
qed

lemma *map-prod-surj*:
  fixes *f* :: *'a* ⇒ *'b*
    and *g* :: *'c* ⇒ *'d*
  assumes *surj f* **and** *surj g*
  shows *surj* (*map-prod f g*)
  unfolding *surj-def*
proof
  fix *y* :: *'b* × *'d*
  from ‹*surj f*› **obtain** *a* **where** *fst y* = *f a*
    by (*auto elim*: *surjE*)
  moreover
  from ‹*surj g*› **obtain** *b* **where** *snd y* = *g b*
    by (*auto elim*: *surjE*)

ultimately have (*fst y*, *snd y*) = *map-prod f g* (*a*,*b*)
by *auto*
then show ∃ *x*. *y* = *map-prod f g x*
by *auto*
qed

lemma *map-prod-surj-on*:
assumes *f* ‘ *A* = *A'* **and** *g* ‘ *B* = *B'*
shows *map-prod f g* ‘ (*A* × *B*) = *A'* × *B'*
unfolding *image-def*
proof (*rule set-eqI*, *rule iffI*)
fix *x* :: *'a* × *'c*
assume *x* ∈ {*y*::*'a* × *'c*. ∃ *x*::*'b* × *'d*∈*A* × *B*. *y* = *map-prod f g x*}
then obtain *y* **where** *y* ∈ *A* × *B* **and** *x* = *map-prod f g y*
by *blast*
from ‹*image f A* = *A'*› **and** ‹*y* ∈ *A* × *B*› **have** *f* (*fst y*) ∈ *A'*
by *auto*
moreover from ‹*image g B* = *B'*› **and** ‹*y* ∈ *A* × *B*› **have** *g* (*snd y*) ∈ *B'*
by *auto*
ultimately have (*f* (*fst y*), *g* (*snd y*)) ∈ (*A'* × *B'*)
by *auto*
with ‹*x* = *map-prod f g y*› **show** *x* ∈ *A'* × *B'*
by (*cases y*) *auto*
next
fix *x* :: *'a* × *'c*
assume *x* ∈ *A'* × *B'*
then have *fst x* ∈ *A'* **and** *snd x* ∈ *B'*
by *auto*
from ‹*image f A* = *A'*› **and** ‹*fst x* ∈ *A'*› **have** *fst x* ∈ *image f A*
by *auto*
then obtain *a* **where** *a* ∈ *A* **and** *fst x* = *f a*
by (*rule imageE*)
moreover from ‹*image g B* = *B'*› **and** ‹*snd x* ∈ *B'*› **obtain** *b* **where** *b* ∈ *B* **and** *snd x* = *g b*
by *auto*
ultimately have (*fst x*, *snd x*) = *map-prod f g* (*a*, *b*)
by *auto*
moreover from ‹*a* ∈ *A*› **and** ‹*b* ∈ *B*› **have** (*a* , *b*) ∈ *A* × *B*
by *auto*
ultimately have ∃ *y* ∈ *A* × *B*. *x* = *map-prod f g y*
by *auto*
then show *x* ∈ {*x*. ∃ *y* ∈ *A* × *B*. *x* = *map-prod f g y*}
by *auto*
qed

lemma *bij-betw-map-prod*:
assumes *bij-betw f A C bij-betw g B D*
shows *bij-betw* (*map-prod f g*) (*A* × *B*) (*C* × *D*)
using *assms* **unfolding** *bij-betw-def inj-on-def* **by** *auto*

14.4 Code generator setup for paired and tripled bounded set comprehension

context
begin

qualified lemma *paired-bounded-Collect-eq-filter* [*code-unfold*, *no-atp*]:
‹{(*x*, *y*). (*x*, *y*) ∈ *A* ∧ *P x y*} = *Set.filter* (λ(*x*, *y*). *P x y*) *A›*
by *auto*

qualified lemma *tripled-bounded-Collect-eq-filter* [*code-unfold*, *no-atp*]:
‹{(*x*, *y*, *z*). (*x*, *y*, *z*) ∈ *A* ∧ *P x y z*} = *Set.filter* (λ(*x*, *y*, *z*). *P x y z*) *A›*
by *auto*

end

14.5 Simproc for rewriting a set comprehension into a point-free expression

ML-file ‹*Tools/set-comprehension-pointfree.ML*›

simproc-setup passive *set-comprehension* (*Collect P*) =
‹*K Set-Comprehension-Pointfree.code-proc*›

setup ‹*Code-Preproc.map-pre* (*Simplifier.add-proc* ***simproc*** ‹*set-comprehension*›)›

14.6 Lemmas about disjointness

lemma *disjnt-Times1-iff* [*simp*]: *disjnt* (*C* × *A*) (*C* × *B*) ⟷ *C* = {} ∨ *disjnt A B*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Times2-iff* [*simp*]: *disjnt* (*A* × *C*) (*B* × *C*) ⟷ *C* = {} ∨ *disjnt A B*
by (*auto simp*: *disjnt-def*)

lemma *disjnt-Sigma-iff*: *disjnt* (*Sigma A C*) (*Sigma B C*) ⟷ (∀ *i* ∈ *A*∩*B*. *C i* = {}) ∨ *disjnt A B*
by (*auto simp*: *disjnt-def*)

14.7 Inductively defined sets

simproc-setup *Collect-mem* (*Collect t*) = ‹
K (*fn ctxt => fn ct =>*
(*case Thm.term-of ct of*
S as ***Const-*** ‹*Collect A for t*› =>
let val (*u*, -, *ps*) = *HOLogic.strip-ptupleabs t in*
(*case u of*
(*c as* ***Const-*** ‹*Set.member - for q S'*›) =>

```
            (case try (HOLogic.strip-ptuple ps) q of
              NONE => NONE
            | SOME ts =>
                if not (Term.is-open S') andalso
                  ts = map Bound (length ps downto 0)
                then
                  let
                    val simp =
                      full-simp-tac (put-simpset HOL-basic-ss ctxt
                            |> Simplifier.add-simps [@{thm split-paired-all}, @{thm
case-prod-conv}]) 1
                  in
                    SOME (Goal.prove ctxt [] [] Const‹Pure.eq Type‹set A› for S
S'›
                     (K (EVERY
                      [resolve-tac ctxt [eq-reflection] 1,
                       resolve-tac ctxt @{thms subset-antisym} 1,
                       resolve-tac ctxt @{thms subsetI} 1,
                       dresolve-tac ctxt @{thms CollectD} 1, simp,
                       resolve-tac ctxt @{thms subsetI} 1,
                       resolve-tac ctxt @{thms CollectI} 1, simp])))
                  end
                else NONE)
          | - => NONE)
        end
    | - => NONE))
›
```

ML-file ‹*Tools/inductive-set.ML*›

14.8 Legacy theorem bindings and duplicates

lemmas *fst-conv* = *prod.sel*(*1*)
lemmas *snd-conv* = *prod.sel*(*2*)
lemmas *split-def* = *case-prod-unfold*
lemmas *split-beta'* = *case-prod-beta'*
lemmas *split-beta* = *prod.case-eq-if*
lemmas *split-conv* = *case-prod-conv*
lemmas *split* = *case-prod-conv*

hide-const (**open**) *prod*

end

15 The Disjoint Sum of Two Types

theory *Sum-Type*
  imports *Typedef Inductive Fun*
begin

15.1 Construction of the sum type and its basic abstract operations

definition *Inl-Rep* :: $'a \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$
where $Inl\text{-}Rep\ a\ x\ y\ p \longleftrightarrow x = a \wedge p$

definition *Inr-Rep* :: $'b \Rightarrow 'a \Rightarrow 'b \Rightarrow bool \Rightarrow bool$
where $Inr\text{-}Rep\ b\ x\ y\ p \longleftrightarrow y = b \wedge \neg\ p$

definition $sum = \{f.\ (\exists\, a.\ f = Inl\text{-}Rep\ (a{::}'a)) \vee (\exists\, b.\ f = Inr\text{-}Rep\ (b{::}'b))\}$

typedef $('a, 'b)\ sum$ (**infixr** ‹+› *10*) $= sum :: ('a \Rightarrow 'b \Rightarrow bool \Rightarrow bool)\ set$
unfolding *sum-def* **by** *auto*

lemma *Inl-RepI*: $Inl\text{-}Rep\ a \in sum$
by (*auto simp add*: *sum-def*)

lemma *Inr-RepI*: $Inr\text{-}Rep\ b \in sum$
by (*auto simp add*: *sum-def*)

lemma *inj-on-Abs-sum*: $A \subseteq sum \Longrightarrow inj\text{-}on\ Abs\text{-}sum\ A$
by (*rule inj-on-inverseI*, *rule Abs-sum-inverse*) *auto*

lemma *Inl-Rep-inject*: *inj-on Inl-Rep A*
proof (*rule inj-onI*)
show $\bigwedge a\ c.\ Inl\text{-}Rep\ a = Inl\text{-}Rep\ c \Longrightarrow a = c$
by (*auto simp add*: *Inl-Rep-def fun-eq-iff*)
qed

lemma *Inr-Rep-inject*: *inj-on Inr-Rep A*
proof (*rule inj-onI*)
show $\bigwedge b\ d.\ Inr\text{-}Rep\ b = Inr\text{-}Rep\ d \Longrightarrow b = d$
by (*auto simp add*: *Inr-Rep-def fun-eq-iff*)
qed

lemma *Inl-Rep-not-Inr-Rep*: $Inl\text{-}Rep\ a \neq Inr\text{-}Rep\ b$
by (*auto simp add*: *Inl-Rep-def Inr-Rep-def fun-eq-iff*)

definition *Inl* :: $'a \Rightarrow 'a + 'b$
where $Inl = Abs\text{-}sum \circ Inl\text{-}Rep$

definition *Inr* :: $'b \Rightarrow 'a + 'b$
where $Inr = Abs\text{-}sum \circ Inr\text{-}Rep$

lemma *inj-Inl* [*simp*]: *inj-on Inl A*
by (*auto simp add*: *Inl-def intro*!: *comp-inj-on Inl-Rep-inject inj-on-Abs-sum Inl-RepI*)

lemma *Inl-inject*: $Inl\ x = Inl\ y \Longrightarrow x = y$
using *inj-Inl* **by** (*rule injD*)

lemma *inj-Inr* [*simp*]: *inj-on Inr A*
by (*auto simp add*: *Inr-def intro*!: *comp-inj-on Inr-Rep-inject inj-on-Abs-sum Inr-RepI*)

lemma *Inr-inject*: *Inr x* = *Inr y* ⟹ *x* = *y*
using *inj-Inr* **by** (*rule injD*)

lemma *Inl-not-Inr*: *Inl a* ≠ *Inr b*
proof −
have {*Inl-Rep a*, *Inr-Rep b*} ⊆ *sum*
using *Inl-RepI* [*of a*] *Inr-RepI* [*of b*] **by** *auto*
with *inj-on-Abs-sum* **have** *inj-on Abs-sum* {*Inl-Rep a*, *Inr-Rep b*} **.**
with *Inl-Rep-not-Inr-Rep* [*of a b*] *inj-on-contraD* **have** *Abs-sum* (*Inl-Rep a*) ≠ *Abs-sum* (*Inr-Rep b*)
by *auto*
then show *?thesis*
by (*simp add*: *Inl-def Inr-def*)
qed

lemma *Inr-not-Inl*: *Inr b* ≠ *Inl a*
using *Inl-not-Inr* **by** (*rule not-sym*)

lemma *sumE*:
assumes ⋀*x*::′*a*. *s* = *Inl x* ⟹ *P*
and ⋀*y*::′*b*. *s* = *Inr y* ⟹ *P*
shows *P*
proof (*rule Abs-sum-cases* [*of s*])
fix *f*
assume *s* = *Abs-sum f* **and** *f* ∈ *sum*
with *assms* **show** *P*
by (*auto simp add*: *sum-def Inl-def Inr-def*)
qed

free-constructors *case-sum* **for**
isl: *Inl projl*
| *Inr projr*
by (*erule sumE*, *assumption*) (*auto dest*: *Inl-inject Inr-inject simp add*: *Inl-not-Inr*)

Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.

setup ‹*Sign.mandatory-path old*›

old-rep-datatype *Inl Inr*
proof −
fix *P*
fix *s* :: ′*a* + ′*b*
assume *x*: ⋀*x*::′*a*. *P* (*Inl x*) **and** *y*: ⋀*y*::′*b*. *P* (*Inr y*)
then show *P s* **by** (*auto intro*: *sumE* [*of s*])
qed (*auto dest*: *Inl-inject Inr-inject simp add*: *Inl-not-Inr*)

setup ‹*Sign.parent-path*›

But erase the prefix for properties that are not generated by *free-constructors*.

setup ‹*Sign.mandatory-path sum*›

declare
old.sum.inject[*iff del*]
old.sum.distinct(*1*)[*simp del*, *induct-simp del*]

lemmas *induct* = *old.sum.induct*
lemmas *inducts* = *old.sum.inducts*
lemmas *rec* = *old.sum.rec*
lemmas *simps* = *sum.inject sum.distinct sum.case sum.rec*

setup ‹*Sign.parent-path*›

primrec *map-sum* :: ('*a* ⇒ '*c*) ⇒ ('*b* ⇒ '*d*) ⇒ '*a* + '*b* ⇒ '*c* + '*d*
where
map-sum f1 f2 (*Inl a*) = *Inl* (*f1 a*)
| *map-sum f1 f2* (*Inr a*) = *Inr* (*f2 a*)

functor *map-sum*: *map-sum*
proof −
show *map-sum f g* ∘ *map-sum h i* = *map-sum* (*f* ∘ *h*) (*g* ∘ *i*) **for** *f g h i*
proof
show (*map-sum f g* ∘ *map-sum h i*) *s* = *map-sum* (*f* ∘ *h*) (*g* ∘ *i*) *s* **for** *s*
by (*cases s*) *simp-all*
qed
show *map-sum id id* = *id*
proof
show *map-sum id id s* = *id s* **for** *s*
by (*cases s*) *simp-all*
qed
qed

lemma *split-sum-all*: (∀ *x*. *P x*) ⟷ (∀ *x*. *P* (*Inl x*)) ∧ (∀ *x*. *P* (*Inr x*))
by (*auto intro*: *sum.induct*)

lemma *split-sum-ex*: (∃ *x*. *P x*) ⟷ (∃ *x*. *P* (*Inl x*)) ∨ (∃ *x*. *P* (*Inr x*))
using *split-sum-all*[*of* λ*x*. ¬*P x*] **by** *blast*

15.2 Projections

lemma *case-sum-KK* [*simp*]: *case-sum* (λ*x*. *a*) (λ*x*. *a*) = (λ*x*. *a*)
by (*rule ext*) (*simp split*: *sum.split*)

lemma *surjective-sum*: *case-sum* (λ*x*::'*a*. *f* (*Inl x*)) (λ*y*::'*b*. *f* (*Inr y*)) = *f*
proof

```
  fix s :: 'a + 'b
  show (case s of Inl (x::'a) ⇒ f (Inl x) | Inr (y::'b) ⇒ f (Inr y)) = f s
    by (cases s) simp-all
qed

lemma case-sum-inject:
  assumes a: case-sum f1 f2 = case-sum g1 g2
    and r: f1 = g1 ⟹ f2 = g2 ⟹ P
  shows P
proof (rule r)
  show f1 = g1
  proof
    fix x :: 'a
    from a have case-sum f1 f2 (Inl x) = case-sum g1 g2 (Inl x) by simp
    then show f1 x = g1 x by simp
  qed
  show f2 = g2
  proof
    fix y :: 'b
    from a have case-sum f1 f2 (Inr y) = case-sum g1 g2 (Inr y) by simp
    then show f2 y = g2 y by simp
  qed
qed

primrec Suml :: ('a ⇒ 'c) ⇒ 'a + 'b ⇒ 'c
  where Suml f (Inl x) = f x

primrec Sumr :: ('b ⇒ 'c) ⇒ 'a + 'b ⇒ 'c
  where Sumr f (Inr x) = f x

lemma Suml-inject:
  assumes Suml f = Suml g
  shows f = g
proof
  fix x :: 'a
  let ?s = Inl x :: 'a + 'b
  from assms have Suml f ?s = Suml g ?s by simp
  then show f x = g x by simp
qed

lemma Sumr-inject:
  assumes Sumr f = Sumr g
  shows f = g
proof
  fix x :: 'b
  let ?s = Inr x :: 'a + 'b
  from assms have Sumr f ?s = Sumr g ?s by simp
  then show f x = g x by simp
qed
```

15.3 The Disjoint Sum of Sets

definition *Plus* :: *'a set* ⇒ *'b set* ⇒ (*'a* + *'b*) *set* (**infixr** ‹<+>› *65*)
where *A* <+> *B* = *Inl* ‘ *A* ∪ *Inr* ‘ *B*

hide-const (**open**) *Plus* — Valuable identifier

lemma *InlI* [*intro*!]: *a* ∈ *A* ⟹ *Inl a* ∈ *A* <+> *B*
by (*simp add*: *Plus-def*)

lemma *InrI* [*intro*!]: *b* ∈ *B* ⟹ *Inr b* ∈ *A* <+> *B*
by (*simp add*: *Plus-def*)

Exhaustion rule for sums, a degenerate form of induction

lemma *PlusE* [*elim*!]:
u ∈ *A* <+> *B* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *u* = *Inl x* ⟹ *P*) ⟹ (⋀*y*. *y* ∈ *B* ⟹ *u* = *Inr y* ⟹ *P*) ⟹ *P*
by (*auto simp add*: *Plus-def*)

lemma *Plus-eq-empty-conv* [*simp*]: *A* <+> *B* = {} ⟷ *A* = {} ∧ *B* = {}
by *auto*

lemma *UNIV-Plus-UNIV* [*simp*]: *UNIV* <+> *UNIV* = *UNIV*
proof (*rule set-eqI*)
fix *u* :: *'a* + *'b*
show *u* ∈ *UNIV* <+> *UNIV* ⟷ *u* ∈ *UNIV* **by** (*cases u*) *auto*
qed

lemma *UNIV-sum*: *UNIV* = *Inl* ‘ *UNIV* ∪ *Inr* ‘ *UNIV*
proof −
have *x* ∈ *range Inl* **if** *x* ∉ *range Inr* **for** *x* :: *'a* + *'b*
using *that* **by** (*cases x*) *simp-all*
then show *?thesis* **by** *auto*
qed

hide-const (**open**) *Suml Sumr sum*

end

16 Rings

theory *Rings*
imports *Groups Set Fun*
begin

16.1 Semirings and rings

class *semiring* = *ab-semigroup-add* + *semigroup-mult* +

assumes *distrib-right* [*algebra-simps*, *algebra-split-simps*]: $(a + b) * c = a * c + b * c$
assumes *distrib-left* [*algebra-simps*, *algebra-split-simps*]: $a * (b + c) = a * b + a * c$
begin

For the *combine-numerals* simproc

lemma *combine-common-factor*: $a * e + (b * e + c) = (a + b) * e + c$
by (*simp add*: *distrib-right ac-simps*)

end

class *mult-zero* = *times* + *zero* +
assumes *mult-zero-left* [*simp*]: $0 * a = 0$
assumes *mult-zero-right* [*simp*]: $a * 0 = 0$
begin

lemma *mult-not-zero*: $a * b \neq 0 \Longrightarrow a \neq 0 \wedge b \neq 0$
by *auto*

end

class *semiring-0* = *semiring* + *comm-monoid-add* + *mult-zero*

class *semiring-0-cancel* = *semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0*
proof
fix $a :: 'a$
have $0 * a + 0 * a = 0 * a + 0$
by (*simp add*: *distrib-right* [*symmetric*])
then show $0 * a = 0$
by (*simp only*: *add-left-cancel*)
have $a * 0 + a * 0 = a * 0 + 0$
by (*simp add*: *distrib-left* [*symmetric*])
then show $a * 0 = 0$
by (*simp only*: *add-left-cancel*)
qed

end

class *comm-semiring* = *ab-semigroup-add* + *ab-semigroup-mult* +
assumes *distrib*: $(a + b) * c = a * c + b * c$
begin

subclass *semiring*
proof
fix $a\ b\ c :: 'a$

show $(a + b) * c = a * c + b * c$
by (*simp add*: *distrib*)
have $a * (b + c) = (b + c) * a$
by (*simp add*: *ac-simps*)
also have $\ldots = b * a + c * a$
by (*simp only*: *distrib*)
also have $\ldots = a * b + a * c$
by (*simp add*: *ac-simps*)
finally show $a * (b + c) = a * b + a * c$
by *blast*
qed

end

class *comm-semiring-0* = *comm-semiring* + *comm-monoid-add* + *mult-zero*
begin

subclass *semiring-0* **..**

end

class *comm-semiring-0-cancel* = *comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**

subclass *comm-semiring-0* **..**

end

class *zero-neq-one* = *zero* + *one* +
assumes *zero-neq-one* [*simp*]: $0 \neq 1$
begin

lemma *one-neq-zero* [*simp*]: $1 \neq 0$
by (*rule not-sym*) (*rule zero-neq-one*)

definition *of-bool* :: *bool* ⇒ *'a*
where *of-bool p* = (*if p then 1 else 0*)

lemma *of-bool-eq* [*simp*, *code*]:
of-bool False = *0*
of-bool True = *1*
by (*simp-all add*: *of-bool-def*)

lemma *of-bool-eq-iff*: *of-bool p* = *of-bool q* ⟷ *p* = *q*
by (*simp add*: *of-bool-def*)

lemma *split-of-bool* [*split*]: *P* (*of-bool p*) ⟷ (*p* ⟶ *P 1*) ∧ (¬ *p* ⟶ *P 0*)

by (*cases p*) *simp-all*

lemma *split-of-bool-asm*: P (*of-bool p*) $\longleftrightarrow \neg$ ($p \wedge \neg$ *P 1* $\vee \neg$ *p* $\wedge \neg$ *P 0*)
by (*cases p*) *simp-all*

lemma *of-bool-eq-0-iff* [*simp*]:
‹*of-bool P = 0* $\longleftrightarrow \neg$ *P*›
by *simp*

lemma *of-bool-eq-1-iff* [*simp*]:
‹*of-bool P = 1* $\longleftrightarrow$ *P*›
by *simp*

end

class *semiring-1* = *zero-neq-one* + *semiring-0* + *monoid-mult*
begin

lemma *of-bool-conj*:
of-bool ($P \wedge Q$) = *of-bool P* $*$ *of-bool Q*
by *auto*

end

lemma *lambda-zero*: (λh::'*a*::*mult-zero*. *0*) = ($*$) *0*
by *auto*

lemma *lambda-one*: (λx::'*a*::*monoid-mult*. *x*) = ($*$) *1*
by *auto*

16.2 Abstract divisibility

class *dvd* = *times*
begin

definition *dvd* :: '*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool* (**infix** ‹*dvd*› *50*)
where *b dvd a* $\longleftrightarrow$ ($\exists$ *k*. *a* = *b* $*$ *k*)

lemma *dvdI* [*intro?*]: *a* = *b* $*$ *k* $\Longrightarrow$ *b dvd a*
unfolding *dvd-def* **..**

lemma *dvdE* [*elim*]: *b dvd a* $\Longrightarrow$ ($\bigwedge$*k*. *a* = *b* $*$ *k* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
unfolding *dvd-def* **by** *blast*

end

context *comm-monoid-mult*
begin

subclass *dvd* **.**

lemma *dvd-refl* [*simp*]: *a dvd a*
proof
  show *a = a * 1* **by** *simp*
qed

lemma *dvd-trans* [*trans*]:
  assumes *a dvd b* **and** *b dvd c*
  shows *a dvd c*
proof −
  from *assms* **obtain** *v* **where** *b = a * v*
    by *auto*
  moreover from *assms* **obtain** *w* **where** *c = b * w*
    by *auto*
  ultimately have *c = a * (v * w)*
    by (*simp add*: *mult.assoc*)
  then show *?thesis* **..**
qed

lemma *subset-divisors-dvd*: {*c*. *c dvd a*} ⊆ {*c*. *c dvd b*} ⟷ *a dvd b*
  by (*auto simp add*: *subset-iff intro*: *dvd-trans*)

lemma *strict-subset-divisors-dvd*: {*c*. *c dvd a*} ⊂ {*c*. *c dvd b*} ⟷ *a dvd b* ∧ ¬ *b dvd a*
  by (*auto simp add*: *subset-iff intro*: *dvd-trans*)

lemma *one-dvd* [*simp*]: *1 dvd a*
  by (*auto intro*: *dvdI*)

lemma *dvd-mult* [*simp*]: *a dvd* (*b * c*) **if** *a dvd c*
  using *that* **by** (*auto intro*: *mult.left-commute dvdI*)

lemma *dvd-mult2* [*simp*]: *a dvd* (*b * c*) **if** *a dvd b*
  using *that dvd-mult* [*of a b c*] **by** (*simp add*: *ac-simps*)

lemma *dvd-triv-right* [*simp*]: *a dvd b * a*
  by (*rule dvd-mult*) (*rule dvd-refl*)

lemma *dvd-triv-left* [*simp*]: *a dvd a * b*
  by (*rule dvd-mult2*) (*rule dvd-refl*)

lemma *mult-dvd-mono*:
  assumes *a dvd b*
    and *c dvd d*
  shows *a * c dvd b * d*
proof −
  from ‹*a dvd b*› **obtain** *b′* **where** *b = a * b′* **..**
  moreover from ‹*c dvd d*› **obtain** *d′* **where** *d = c * d′* **..**

ultimately have $b * d = (a * c) * (b' * d')$
by (*simp add*: *ac-simps*)
then show *?thesis* **..**
qed

lemma *dvd-mult-left*: $a * b$ *dvd* $c \Longrightarrow a$ *dvd* c
by (*simp add*: *dvd-def mult.assoc*) *blast*

lemma *dvd-mult-right*: $a * b$ *dvd* $c \Longrightarrow b$ *dvd* c
using *dvd-mult-left* [*of b a c*] **by** (*simp add*: *ac-simps*)

end

class *comm-semiring-1* = *zero-neq-one* + *comm-semiring-0* + *comm-monoid-mult*
begin

subclass *semiring-1* **..**

lemma *dvd-0-left-iff* [*simp*]: *0 dvd a* $\longleftrightarrow a = 0$
by *auto*

lemma *dvd-0-right* [*iff*]: *a dvd 0*
proof
show $0 = a * 0$ **by** *simp*
qed

lemma *dvd-0-left*: *0 dvd a* $\Longrightarrow a = 0$
by *simp*

lemma *dvd-add* [*simp*]:
assumes *a dvd b* **and** *a dvd c*
shows *a dvd* $(b + c)$
proof −
from ‹*a dvd b*› **obtain** b' **where** $b = a * b'$ **..**
moreover from ‹*a dvd c*› **obtain** c' **where** $c = a * c'$ **..**
ultimately have $b + c = a * (b' + c')$
by (*simp add*: *distrib-left*)
then show *?thesis* **..**
qed

end

class *semiring-1-cancel* = *semiring* + *cancel-comm-monoid-add*
+ *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-0-cancel* **..**

subclass *semiring-1* **..**

end

class *comm-semiring-1-cancel* =
comm-semiring + *cancel-comm-monoid-add* + *zero-neq-one* + *comm-monoid-mult*
+
assumes *right-diff-distrib'* [*algebra-simps*, *algebra-split-simps*]:
$a * (b - c) = a * b - a * c$
begin

subclass *semiring-1-cancel* **..**
subclass *comm-semiring-0-cancel* **..**
subclass *comm-semiring-1* **..**

lemma *left-diff-distrib'* [*algebra-simps*, *algebra-split-simps*]:
$(b - c) * a = b * a - c * a$
by (*simp add*: *algebra-simps*)

lemma *dvd-add-times-triv-left-iff* [*simp*]: *a dvd c* $*$ *a* + *b* $\longleftrightarrow$ *a dvd b*
proof −
have *a dvd a* $*$ *c* + *b* $\longleftrightarrow$ *a dvd b* (**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?Q*
then show *?P* **by** *simp*
next
assume *?P*
then obtain *d* **where** $a * c + b = a * d$ **..**
then have $a * c + b - a * c = a * d - a * c$ **by** *simp*
then have $b = a * d - a * c$ **by** *simp*
then have $b = a * (d - c)$ **by** (*simp add*: *algebra-simps*)
then show *?Q* **..**
qed
then show *a dvd c* $*$ *a* + *b* $\longleftrightarrow$ *a dvd b* **by** (*simp add*: *ac-simps*)
qed

lemma *dvd-add-times-triv-right-iff* [*simp*]: *a dvd b* + *c* $*$ *a* $\longleftrightarrow$ *a dvd b*
using *dvd-add-times-triv-left-iff* [*of a c b*] **by** (*simp add*: *ac-simps*)

lemma *dvd-add-triv-left-iff* [*simp*]: *a dvd a* + *b* $\longleftrightarrow$ *a dvd b*
using *dvd-add-times-triv-left-iff* [*of a 1 b*] **by** *simp*

lemma *dvd-add-triv-right-iff* [*simp*]: *a dvd b* + *a* $\longleftrightarrow$ *a dvd b*
using *dvd-add-times-triv-right-iff* [*of a b 1*] **by** *simp*

lemma *dvd-add-right-iff*:
assumes *a dvd b*
shows *a dvd b* + *c* $\longleftrightarrow$ *a dvd c* (**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?P*

then obtain d **where** $b + c = a * d$ **..**
moreover from ‹a *dvd* b› **obtain** e **where** $b = a * e$ **..**
ultimately have $a * e + c = a * d$ **by** *simp*
then have $a * e + c - a * e = a * d - a * e$ **by** *simp*
then have $c = a * d - a * e$ **by** *simp*
then have $c = a * (d - e)$ **by** (*simp add*: *algebra-simps*)
then show *?Q* **..**
next
assume *?Q*
with *assms* **show** *?P* **by** *simp*
qed

lemma *dvd-add-left-iff*: a *dvd* $c \Longrightarrow a$ *dvd* $b + c \longleftrightarrow a$ *dvd* b
using *dvd-add-right-iff* [*of* a c b] **by** (*simp add*: *ac-simps*)

end

class *ring* = *semiring* + *ab-group-add*
begin

subclass *semiring-0-cancel* **..**

Distribution rules

lemma *minus-mult-left*: $-(a * b) = - a * b$
by (*rule minus-unique*) (*simp add*: *distrib-right* [*symmetric*])

lemma *minus-mult-right*: $-(a * b) = a * - b$
by (*rule minus-unique*) (*simp add*: *distrib-left* [*symmetric*])

Extract signs from products

lemmas *mult-minus-left* [*simp*] = *minus-mult-left* [*symmetric*]
lemmas *mult-minus-right* [*simp*] = *minus-mult-right* [*symmetric*]

lemma *minus-mult-minus* [*simp*]: $- a * - b = a * b$
by *simp*

lemma *minus-mult-commute*: $- a * b = a * - b$
by *simp*

lemma *right-diff-distrib* [*algebra-simps*, *algebra-split-simps*]:
$a * (b - c) = a * b - a * c$
using *distrib-left* [*of* a b $-c$] **by** *simp*

lemma *left-diff-distrib* [*algebra-simps*, *algebra-split-simps*]:
$(a - b) * c = a * c - b * c$
using *distrib-right* [*of* a $-$ b c] **by** *simp*

lemmas *ring-distribs* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

lemma *eq-add-iff1*: $a * e + c = b * e + d \longleftrightarrow (a - b) * e + c = d$
by (*simp add*: *algebra-simps*)

lemma *eq-add-iff2*: $a * e + c = b * e + d \longleftrightarrow c = (b - a) * e + d$
by (*simp add*: *algebra-simps*)

end

lemmas *ring-distribs* = *distrib-left distrib-right left-diff-distrib right-diff-distrib*

class *comm-ring* = *comm-semiring* + *ab-group-add*
begin

subclass *ring* **..**
subclass *comm-semiring-0-cancel* **..**

lemma *square-diff-square-factored*: $x * x - y * y = (x + y) * (x - y)$
by (*simp add*: *algebra-simps*)

end

class *ring-1* = *ring* + *zero-neq-one* + *monoid-mult*
begin

subclass *semiring-1-cancel* **..**

lemma *of-bool-not-iff*:
‹*of-bool* (¬ *P*) = *1* − *of-bool P*›
by *simp*

lemma *square-diff-one-factored*: $x * x - 1 = (x + 1) * (x - 1)$
by (*simp add*: *algebra-simps*)

end

class *comm-ring-1* = *comm-ring* + *zero-neq-one* + *comm-monoid-mult*
begin

subclass *ring-1* **..**
subclass *comm-semiring-1-cancel*
by *standard* (*simp add*: *algebra-simps*)

lemma *dvd-minus-iff* [*simp*]: *x dvd* − *y* ⟷ *x dvd y*
proof
assume *x dvd* − *y*
then have *x dvd* − *1* ∗ − *y* **by** (*rule dvd-mult*)
then show *x dvd y* **by** *simp*
next
assume *x dvd y*

then have *x dvd − 1 ∗ y* **by** (*rule dvd-mult*)
then show *x dvd − y* **by** *simp*
qed

lemma *minus-dvd-iff* [*simp*]: *− x dvd y ⟷ x dvd y*
proof
assume *− x dvd y*
then obtain *k* **where** *y = − x ∗ k* **..**
then have *y = x ∗ − k* **by** *simp*
then show *x dvd y* **..**
next
assume *x dvd y*
then obtain *k* **where** *y = x ∗ k* **..**
then have *y = − x ∗ − k* **by** *simp*
then show *− x dvd y* **..**
qed

lemma *dvd-diff-right-iff*:
assumes *a dvd b*
shows *a dvd b − c ⟷ a dvd c* (**is** *?P ⟷ ?Q*)
using *dvd-add-right-iff*[*of a b −c*] *assms* **by** *auto*

lemma *dvd-diff-left-iff*:
shows *a dvd c ⟹ a dvd b − c ⟷ a dvd b*
using *dvd-add-left-iff*[*of a −c b*] **by** *auto*

lemma *dvd-diff* [*simp*]: *x dvd y ⟹ x dvd z ⟹ x dvd* (*y − z*)
using *dvd-add* [*of x y − z*] **by** *simp*

end

16.3 Towards integral domains

class *semiring-no-zero-divisors = semiring-0 +*
assumes *no-zero-divisors*: *a ≠ 0 ⟹ b ≠ 0 ⟹ a ∗ b ≠ 0*
begin

lemma *divisors-zero*:
assumes *a ∗ b = 0*
shows *a = 0 ∨ b = 0*
proof (*rule classical*)
assume *¬ ?thesis*
then have *a ≠ 0* **and** *b ≠ 0* **by** *auto*
with *no-zero-divisors* **have** *a ∗ b ≠ 0* **by** *blast*
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *mult-eq-0-iff* [*simp*]: *a ∗ b = 0 ⟷ a = 0 ∨ b = 0*
proof (*cases a = 0 ∨ b = 0*)

case *False*
then have $a \neq 0$ **and** $b \neq 0$ **by** *auto*
then show *?thesis* **using** *no-zero-divisors* **by** *simp*
next
case *True*
then show *?thesis* **by** *auto*
qed

end

class *semiring-1-no-zero-divisors* = *semiring-1* + *semiring-no-zero-divisors*

class *semiring-no-zero-divisors-cancel* = *semiring-no-zero-divisors* +
assumes *mult-cancel-right* [*simp*]: $a * c = b * c \longleftrightarrow c = 0 \vee a = b$
and *mult-cancel-left* [*simp*]: $c * a = c * b \longleftrightarrow c = 0 \vee a = b$
begin

lemma *mult-left-cancel*: $c \neq 0 \Longrightarrow c * a = c * b \longleftrightarrow a = b$
by *simp*

lemma *mult-right-cancel*: $c \neq 0 \Longrightarrow a * c = b * c \longleftrightarrow a = b$
by *simp*

end

class *ring-no-zero-divisors* = *ring* + *semiring-no-zero-divisors*
begin

subclass *semiring-no-zero-divisors-cancel*
proof
fix *a b c*
have $a * c = b * c \longleftrightarrow (a - b) * c = 0$
by (*simp add*: *algebra-simps*)
also have $\ldots \longleftrightarrow c = 0 \vee a = b$
by *auto*
finally show $a * c = b * c \longleftrightarrow c = 0 \vee a = b$ **.**
have $c * a = c * b \longleftrightarrow c * (a - b) = 0$
by (*simp add*: *algebra-simps*)
also have $\ldots \longleftrightarrow c = 0 \vee a = b$
by *auto*
finally show $c * a = c * b \longleftrightarrow c = 0 \vee a = b$ **.**
qed

end

class *ring-1-no-zero-divisors* = *ring-1* + *ring-no-zero-divisors*
begin

subclass *semiring-1-no-zero-divisors* **..**

lemma *square-eq-1-iff*: $x * x = 1 \longleftrightarrow x = 1 \vee x = -1$
proof −
  have $(x - 1) * (x + 1) = x * x - 1$
    by (*simp add*: *algebra-simps*)
  then have $x * x = 1 \longleftrightarrow (x - 1) * (x + 1) = 0$
    by *simp*
  then show *?thesis*
    by (*simp add*: *eq-neg-iff-add-eq-0*)
qed

lemma *mult-cancel-right1* [*simp*]: $c = b * c \longleftrightarrow c = 0 \vee b = 1$
  using *mult-cancel-right* [*of 1 c b*] **by** *auto*

lemma *mult-cancel-right2* [*simp*]: $a * c = c \longleftrightarrow c = 0 \vee a = 1$
  using *mult-cancel-right* [*of a c 1*] **by** *simp*

lemma *mult-cancel-left1* [*simp*]: $c = c * b \longleftrightarrow c = 0 \vee b = 1$
  using *mult-cancel-left* [*of c 1 b*] **by** *force*

lemma *mult-cancel-left2* [*simp*]: $c * a = c \longleftrightarrow c = 0 \vee a = 1$
  using *mult-cancel-left* [*of c a 1*] **by** *simp*

end

class *semidom* = *comm-semiring-1-cancel* + *semiring-no-zero-divisors*
begin

subclass *semiring-1-no-zero-divisors* **..**

end

class *idom* = *comm-ring-1* + *semiring-no-zero-divisors*
begin

subclass *semidom* **..**

subclass *ring-1-no-zero-divisors* **..**

lemma *dvd-mult-cancel-right* [*simp*]:
  $a * c \text{ dvd } b * c \longleftrightarrow c = 0 \vee a \text{ dvd } b$
proof −
  have $a * c \text{ dvd } b * c \longleftrightarrow (\exists k.\ b * c = (a * k) * c)$
    by (*auto simp add*: *ac-simps*)
  also have $(\exists k.\ b * c = (a * k) * c) \longleftrightarrow c = 0 \vee a \text{ dvd } b$
    by *auto*
  finally show *?thesis* **.**
qed

lemma *dvd-mult-cancel-left* [*simp*]:
  $c * a$ *dvd* $c * b \longleftrightarrow c = 0 \vee a$ *dvd* b
  using *dvd-mult-cancel-right* [*of* a c b] **by** (*simp add*: *ac-simps*)

lemma *square-eq-iff*: $a * a = b * b \longleftrightarrow a = b \vee a = - b$
proof
  assume $a * a = b * b$
  then have $(a - b) * (a + b) = 0$
    by (*simp add*: *algebra-simps*)
  then show $a = b \vee a = - b$
    by (*simp add*: *eq-neg-iff-add-eq-0*)
next
  assume $a = b \vee a = - b$
  then show $a * a = b * b$ **by** *auto*
qed

lemma *inj-mult-left* [*simp*]: ‹*inj* ((∗) *a*) ⟷ *a* ≠ *0*› (**is** ‹*?P* ⟷ *?Q*›)
proof
  assume *?P*
  show *?Q*
  proof
    assume ‹*a* = *0*›
    with ‹*?P*› **have** *inj* ((∗) *0*)
      by *simp*
    moreover have *0* ∗ *0* = *0* ∗ *1*
      by *simp*
    ultimately have *0* = *1*
      by (*rule injD*)
    then show *False*
      by *simp*
  qed
next
  assume *?Q* **then show** *?P*
    by (*auto intro*: *injI*)
qed

end

class *idom-abs-sgn* = *idom* + *abs* + *sgn* +
  assumes *sgn-mult-abs*: *sgn a* ∗ |*a*| = *a*
    and *sgn-sgn* [*simp*]: *sgn* (*sgn a*) = *sgn a*
    and *abs-abs* [*simp*]: ||*a*|| = |*a*|
    and *abs-0* [*simp*]: |*0*| = *0*
    and *sgn-0* [*simp*]: *sgn 0* = *0*
    and *sgn-1* [*simp*]: *sgn 1* = *1*
    and *sgn-minus-1*: *sgn* (− *1*) = − *1*
    and *sgn-mult*: *sgn* (*a* ∗ *b*) = *sgn a* ∗ *sgn b*
begin

lemma *sgn-eq-0-iff*:
sgn a = 0 ⟷ a = 0
proof −
{ assume *sgn a = 0*
then have *sgn a * |a| = 0*
by *simp*
then have *a = 0*
by (*simp add*: *sgn-mult-abs*)
} then show *?thesis*
by *auto*
qed

lemma *abs-eq-0-iff*:
|a| = 0 ⟷ a = 0
proof −
{ assume *|a| = 0*
then have *sgn a * |a| = 0*
by *simp*
then have *a = 0*
by (*simp add*: *sgn-mult-abs*)
} then show *?thesis*
by *auto*
qed

lemma *abs-mult-sgn*:
*|a| * sgn a = a*
using *sgn-mult-abs* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *abs-1* [*simp*]:
|1| = 1
using *sgn-mult-abs* [*of 1*] **by** *simp*

lemma *sgn-abs* [*simp*]:
|sgn a| = of-bool (a ≠ 0)
using *sgn-mult-abs* [*of sgn a*] *mult-cancel-left* [*of sgn a |sgn a| 1*]
by (*auto simp add*: *sgn-eq-0-iff*)

lemma *abs-sgn* [*simp*]:
sgn |a| = of-bool (a ≠ 0)
using *sgn-mult-abs* [*of |a|*] *mult-cancel-right* [*of sgn |a| |a| 1*]
by (*auto simp add*: *abs-eq-0-iff*)

lemma *abs-mult*:
*|a * b| = |a| * |b|*
proof (*cases a = 0 ∨ b = 0*)
case *True*
then show *?thesis*
by *auto*
next

case *False*
then have $*$: *sgn* $(a * b) \neq 0$
by (*simp add*: *sgn-eq-0-iff*)
from *abs-mult-sgn* [*of* $a * b$] *abs-mult-sgn* [*of* a] *abs-mult-sgn* [*of* b]
have $|a * b| * sgn\ (a * b) = |a| * sgn\ a * |b| * sgn\ b$
by (*simp add*: *ac-simps*)
then have $|a * b| * sgn\ (a * b) = |a| * |b| * sgn\ (a * b)$
by (*simp add*: *sgn-mult ac-simps*)
with $*$ **show** *?thesis*
by *simp*
qed

lemma *sgn-minus* [*simp*]:
$sgn\ (-\ a) = -\ sgn\ a$
proof −
from *sgn-minus-1* **have** $sgn\ (-\ 1 * a) = -\ 1 * sgn\ a$
by (*simp only*: *sgn-mult*)
then show *?thesis*
by *simp*
qed

lemma *abs-minus* [*simp*]:
$|-\ a| = |a|$
proof −
have [*simp*]: $|-\ 1| = 1$
using *sgn-mult-abs* [*of* $-\ 1$] **by** *simp*
then have $|-\ 1 * a| = 1 * |a|$
by (*simp only*: *abs-mult*)
then show *?thesis*
by *simp*
qed

end

16.4 (Partial) Division

class *divide* =
fixes *divide* :: $'a \Rightarrow 'a \Rightarrow 'a$ (**infixl** ‹*div*› *70*)

setup ‹*Sign.add-const-constraint* (***const-name***‹*divide*›, *SOME* ***typ***‹$'a \Rightarrow 'a \Rightarrow 'a$›)›

context *semiring*
begin

lemma [*field-simps*, *field-split-simps*]:
shows *distrib-left-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ a \Longrightarrow a * (b + c) = a * b + a * c$
and *distrib-right-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ c \Longrightarrow (a + b) * c = a$

$* c + b * c$
by (*rule distrib-left distrib-right*)+

end

context *ring*
begin

lemma [*field-simps*, *field-split-simps*]:
shows *left-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ c \Longrightarrow (a - b) * c = a * c - b * c$
and *right-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ div\ y)\ a \Longrightarrow a * (b - c) = a * b - a * c$
by (*rule left-diff-distrib right-diff-distrib*)+

end

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*divide*›, *SOME* ***typ*** ‹'*a*::*divide* ⇒ '*a* ⇒ '*a*›)›

class *divide-trivial* = *zero* + *one* + *divide* +
assumes *div-by-0* [*simp*]: ‹*a div 0* = *0*›
and *div-by-1* [*simp*]: ‹*a div 1* = *a*›
and *div-0* [*simp*]: ‹*0 div a* = *0*›

Algebraic classes with division

class *semidom-divide* = *semidom* + *divide* +
assumes *nonzero-mult-div-cancel-right* [*simp*]: ‹$b \neq 0 \Longrightarrow (a * b)\ div\ b = a$›
assumes *semidom-div-by-0*: ‹*a div 0* = *0*›
begin

lemma *nonzero-mult-div-cancel-left* [*simp*]: ‹$a \neq 0 \Longrightarrow (a * b)\ div\ a = b$›
using *nonzero-mult-div-cancel-right* [*of a b*] **by** (*simp add*: *ac-simps*)

subclass *divide-trivial*
proof
show [*simp*]: ‹*a div 0* = *0*› **for** *a*
by (*fact semidom-div-by-0*)
show ‹*a div 1* = *a*› **for** *a*
using *nonzero-mult-div-cancel-right* [*of 1 a*] **by** *simp*
show ‹*0 div a* = *0*› **for** *a*
using *nonzero-mult-div-cancel-right* [*of a 0*] **by** (*cases* ‹*a* = *0*›) *simp-all*
qed

subclass *semiring-no-zero-divisors-cancel*
proof
show *: $a * c = b * c \longleftrightarrow c = 0 \vee a = b$ **for** *a b c*
proof (*cases c* = *0*)
case *True*

```
    then show ?thesis by simp
  next
    case False
    have a = b if a * c = b * c
    proof −
      from that have a * c div c = b * c div c
        by simp
      with False show ?thesis
        by simp
    qed
    then show ?thesis by auto
  qed
  show c * a = c * b ⟷ c = 0 ∨ a = b for a b c
    using * [of a c b] by (simp add: ac-simps)
qed

lemma div-self [simp]: a ≠ 0 ⟹ a div a = 1
  using nonzero-mult-div-cancel-left [of a 1] by simp

lemma dvd-div-eq-0-iff:
  assumes b dvd a
  shows a div b = 0 ⟷ a = 0
  using assms by (elim dvdE, cases b = 0) simp-all

lemma dvd-div-eq-cancel:
  a div c = b div c ⟹ c dvd a ⟹ c dvd b ⟹ a = b
  by (elim dvdE, cases c = 0) simp-all

lemma dvd-div-eq-iff:
  c dvd a ⟹ c dvd b ⟹ a div c = b div c ⟷ a = b
  by (elim dvdE, cases c = 0) simp-all

lemma inj-on-mult:
  inj-on ((*) a) A if a ≠ 0
proof (rule inj-onI)
  fix b c
  assume a * b = a * c
  then have a * b div a = a * c div a
    by (simp only:)
  with that show b = c
    by simp
qed

end

class idom-divide = idom + semidom-divide
begin

lemma dvd-neg-div:
```

assumes *b dvd a*
shows $- a \mathit{div} b = - (a \mathit{div} b)$
proof (*cases* $b = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
from *assms* **obtain** *c* **where** $a = b * c$ **..**
then have $- a \mathit{div} b = (b * - c) \mathit{div} b$
by *simp*
from *False* **also have** $\ldots = - c$
by (*rule nonzero-mult-div-cancel-left*)
with *False* ‹$a = b * c$› **show** *?thesis*
by *simp*
qed

lemma *dvd-div-neg*:
assumes *b dvd a*
shows $a \mathit{div} - b = - (a \mathit{div} b)$
proof (*cases* $b = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $- b \neq 0$
by *simp*
from *assms* **obtain** *c* **where** $a = b * c$ **..**
then have $a \mathit{div} - b = (- b * - c) \mathit{div} - b$
by *simp*
from ‹$- b \neq 0$› **also have** $\ldots = - c$
by (*rule nonzero-mult-div-cancel-left*)
with *False* ‹$a = b * c$› **show** *?thesis*
by *simp*
qed

end

class *algebraic-semidom* = *semidom-divide*
begin

Class *algebraic-semidom* enriches a integral domain by notions from algebra, like units in a ring. It is a separate class to avoid spoiling fields with notions which are degenerated there.

lemma *dvd-times-left-cancel-iff* [*simp*]:
assumes $a \neq 0$
shows $a * b \mathit{dvd} a * c \longleftrightarrow b \mathit{dvd} c$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*

```
  then obtain d where a * c = a * b * d ..
  with assms have c = b * d by (simp add: ac-simps)
  then show ?rhs ..
next
  assume ?rhs
  then obtain d where c = b * d ..
  then have a * c = a * b * d by (simp add: ac-simps)
  then show ?lhs ..
qed

lemma dvd-times-right-cancel-iff [simp]:
  assumes a ≠ 0
  shows b * a dvd c * a ⟷ b dvd c
  using dvd-times-left-cancel-iff [of a b c] assms by (simp add: ac-simps)

lemma div-dvd-iff-mult:
  assumes b ≠ 0 and b dvd a
  shows a div b dvd c ⟷ a dvd c * b
proof −
  from ‹b dvd a› obtain d where a = b * d ..
  with ‹b ≠ 0› show ?thesis by (simp add: ac-simps)
qed

lemma dvd-div-iff-mult:
  assumes c ≠ 0 and c dvd b
  shows a dvd b div c ⟷ a * c dvd b
proof −
  from ‹c dvd b› obtain d where b = c * d ..
  with ‹c ≠ 0› show ?thesis by (simp add: mult.commute [of a])
qed

lemma div-dvd-div [simp]:
  assumes a dvd b and a dvd c
  shows b div a dvd c div a ⟷ b dvd c
proof (cases a = 0)
  case True
  with assms show ?thesis by simp
next
  case False
  moreover from assms obtain k l where b = a * k and c = a * l
    by blast
  ultimately show ?thesis by simp
qed

lemma div-add [simp]:
  assumes c dvd a and c dvd b
  shows (a + b) div c = a div c + b div c
proof (cases c = 0)
  case True
```

then show *?thesis* **by** *simp*
next
case *False*
moreover from *assms* **obtain** $k\ l$ **where** $a = c * k$ **and** $b = c * l$
by *blast*
moreover have $c * k + c * l = c * (k + l)$
by (*simp add*: *algebra-simps*)
ultimately show *?thesis*
by *simp*
qed

lemma *div-mult-div-if-dvd*:
assumes *b dvd a* **and** *d dvd c*
shows $(a\ div\ b) * (c\ div\ d) = (a * c)\ div\ (b * d)$
proof (*cases* $b = 0 \vee c = 0$)
case *True*
with *assms* **show** *?thesis* **by** *auto*
next
case *False*
moreover from *assms* **obtain** $k\ l$ **where** $a = b * k$ **and** $c = d * l$
by *blast*
moreover have $b * k * (d * l)\ div\ (b * d) = (b * d) * (k * l)\ div\ (b * d)$
by (*simp add*: *ac-simps*)
ultimately show *?thesis* **by** *simp*
qed

lemma *dvd-div-eq-mult*:
assumes $a \neq 0$ **and** *a dvd b*
shows $b\ div\ a = c \longleftrightarrow b = c * a$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
then show *?lhs* **by** (*simp add*: *assms*)
next
assume *?lhs*
then have $b\ div\ a * a = c * a$ **by** *simp*
moreover from *assms* **have** $b\ div\ a * a = b$
by (*auto simp add*: *ac-simps*)
ultimately show *?rhs* **by** *simp*
qed

lemma *dvd-div-mult-self* [*simp*]: *a dvd b* $\Longrightarrow$ $b\ div\ a * a = b$
by (*cases* $a = 0$) (*auto simp add*: *ac-simps*)

lemma *dvd-mult-div-cancel* [*simp*]: *a dvd b* $\Longrightarrow$ $a * (b\ div\ a) = b$
using *dvd-div-mult-self* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *div-mult-swap*:
assumes *c dvd b*

```
  shows a * (b div c) = (a * b) div c
proof (cases c = 0)
  case True
  then show ?thesis by simp
next
  case False
  from assms obtain d where b = c * d ..
  moreover from False have a * divide (d * c) c = ((a * d) * c) div c
    by simp
  ultimately show ?thesis by (simp add: ac-simps)
qed

lemma dvd-div-mult: c dvd b ⟹ b div c * a = (b * a) div c
  using div-mult-swap [of c b a] by (simp add: ac-simps)

lemma dvd-div-mult2-eq:
  assumes b * c dvd a
  shows a div (b * c) = a div b div c
proof −
  from assms obtain k where a = b * c * k ..
  then show ?thesis
    by (cases b = 0 ∨ c = 0) (auto, simp add: ac-simps)
qed

lemma dvd-div-div-eq-mult:
  assumes a ≠ 0 c ≠ 0 and a dvd b c dvd d
  shows b div a = d div c ⟷ b * c = a * d
    (is ?lhs ⟷ ?rhs)
proof −
  from assms have a * c ≠ 0 by simp
  then have ?lhs ⟷ b div a * (a * c) = d div c * (a * c)
    by simp
  also have ... ⟷ (a * (b div a)) * c = (c * (d div c)) * a
    by (simp add: ac-simps)
  also have ... ⟷ (a * b div a) * c = (c * d div c) * a
    using assms by (simp add: div-mult-swap)
  also have ... ⟷ ?rhs
    using assms by (simp add: ac-simps)
  finally show ?thesis .
qed

lemma dvd-mult-imp-div:
  assumes a * c dvd b
  shows a dvd b div c
proof (cases c = 0)
  case True then show ?thesis by simp
next
  case False
  from ‹a * c dvd b› obtain d where b = a * c * d ..
```

with *False* **show** *?thesis*
by (*simp add*: *mult.commute* [*of a*] *mult.assoc*)
qed

lemma *div-div-eq-right*:
assumes *c dvd b b dvd a*
shows *a div* (*b div c*) = *a div b* ∗ *c*
proof (*cases c* = *0* ∨ *b* = *0*)
case *True*
then show *?thesis*
by *auto*
next
case *False*
from *assms* **obtain** *r s* **where** *b* = *c* ∗ *r* **and** *a* = *c* ∗ *r* ∗ *s*
by *blast*
moreover with *False* **have** *r* ≠ *0*
by *auto*
ultimately show *?thesis* **using** *False*
by *simp* (*simp add*: *mult.commute* [*of* - *r*] *mult.assoc mult.commute* [*of c*])
qed

lemma *div-div-div-same*:
assumes *d dvd b b dvd a*
shows (*a div d*) *div* (*b div d*) = *a div b*
proof (*cases b* = *0* ∨ *d* = *0*)
case *True*
with *assms* **show** *?thesis*
by *auto*
next
case *False*
from *assms* **obtain** *r s*
where *a* = *d* ∗ *r* ∗ *s* **and** *b* = *d* ∗ *r*
by *blast*
with *False* **show** *?thesis*
by *simp* (*simp add*: *ac-simps*)
qed

Units: invertible elements in a ring

abbreviation *is-unit* :: *'a* ⇒ *bool*
where *is-unit a* ≡ *a dvd 1*

lemma *not-is-unit-0* [*simp*]: ¬ *is-unit 0*
by *simp*

lemma *unit-imp-dvd* [*dest*]: *is-unit b* ⟹ *b dvd a*
by (*rule dvd-trans* [*of* - *1*]) *simp-all*

lemma *unit-dvdE*:
assumes *is-unit a*

```
  obtains c where a ≠ 0 and b = a * c
proof −
  from assms have a dvd b by auto
  then obtain c where b = a * c ..
  moreover from assms have a ≠ 0 by auto
  ultimately show thesis using that by blast
qed

lemma dvd-unit-imp-unit: a dvd b ⟹ is-unit b ⟹ is-unit a
  by (rule dvd-trans)

lemma unit-div-1-unit [simp, intro]:
  assumes is-unit a
  shows is-unit (1 div a)
proof −
  from assms have 1 = 1 div a * a by simp
  then show is-unit (1 div a) by (rule dvdI)
qed

lemma is-unitE [elim?]:
  assumes is-unit a
  obtains b where a ≠ 0 and b ≠ 0
    and is-unit b and 1 div a = b and 1 div b = a
    and a * b = 1 and c div a = c * b
proof (rule that)
  define b where b = 1 div a
  then show 1 div a = b by simp
  from assms b-def show is-unit b by simp
  with assms show a ≠ 0 and b ≠ 0 by auto
  from assms b-def show a * b = 1 by simp
  then have 1 = a * b ..
  with b-def ‹b ≠ 0› show 1 div b = a by simp
  from assms have a dvd c ..
  then obtain d where c = a * d ..
  with ‹a ≠ 0› ‹a * b = 1› show c div a = c * b
    by (simp add: mult.assoc mult.left-commute [of a])
qed

lemma unit-prod [intro]: is-unit a ⟹ is-unit b ⟹ is-unit (a * b)
  by (subst mult-1-left [of 1, symmetric]) (rule mult-dvd-mono)

lemma is-unit-mult-iff: is-unit (a * b) ⟷ is-unit a ∧ is-unit b
  by (auto dest: dvd-mult-left dvd-mult-right)

lemma unit-div [intro]: is-unit a ⟹ is-unit b ⟹ is-unit (a div b)
  by (erule is-unitE [of b a]) (simp add: ac-simps unit-prod)

lemma mult-unit-dvd-iff:
  assumes is-unit b
```

shows $a * b$ *dvd* $c \longleftrightarrow a$ *dvd* c
proof
assume $a * b$ *dvd* c
with *assms* **show** a *dvd* c
by (*simp add*: *dvd-mult-left*)
next
assume a *dvd* c
then obtain k **where** $c = a * k$ **..**
with *assms* **have** $c = (a * b) * (1$ *div* $b * k)$
by (*simp add*: *mult-ac*)
then show $a * b$ *dvd* c **by** (*rule dvdI*)
qed

lemma *mult-unit-dvd-iff'*: *is-unit* $a \Longrightarrow (a * b)$ *dvd* $c \longleftrightarrow b$ *dvd* c
using *mult-unit-dvd-iff* [*of* a b c] **by** (*simp add*: *ac-simps*)

lemma *dvd-mult-unit-iff*:
assumes *is-unit* b
shows a *dvd* $c * b \longleftrightarrow a$ *dvd* c
proof
assume a *dvd* $c * b$
with *assms* **have** $c * b$ *dvd* $c * (b * (1$ *div* $b))$
by (*subst mult-assoc* [*symmetric*]) *simp*
also from *assms* **have** $b * (1$ *div* $b) = 1$
by (*rule is-unitE*) *simp*
finally have $c * b$ *dvd* c **by** *simp*
with ‹a *dvd* $c * b$› **show** a *dvd* c **by** (*rule dvd-trans*)
next
assume a *dvd* c
then show a *dvd* $c * b$ **by** *simp*
qed

lemma *dvd-mult-unit-iff'*: *is-unit* $b \Longrightarrow a$ *dvd* $b * c \longleftrightarrow a$ *dvd* c
using *dvd-mult-unit-iff* [*of* b a c] **by** (*simp add*: *ac-simps*)

lemma *div-unit-dvd-iff*: *is-unit* $b \Longrightarrow a$ *div* b *dvd* $c \longleftrightarrow a$ *dvd* c
by (*erule is-unitE* [*of* - a]) (*auto simp add*: *mult-unit-dvd-iff*)

lemma *dvd-div-unit-iff*: *is-unit* $b \Longrightarrow a$ *dvd* c *div* $b \longleftrightarrow a$ *dvd* c
by (*erule is-unitE* [*of* - c]) (*simp add*: *dvd-mult-unit-iff*)

lemmas *unit-dvd-iff* = *mult-unit-dvd-iff mult-unit-dvd-iff'*
dvd-mult-unit-iff dvd-mult-unit-iff'
div-unit-dvd-iff dvd-div-unit-iff

lemma *unit-mult-div-div* [*simp*]: *is-unit* $a \Longrightarrow b * (1$ *div* $a) = b$ *div* a
by (*erule is-unitE* [*of* - b]) *simp*

lemma *unit-div-mult-self* [*simp*]: *is-unit* $a \Longrightarrow b$ *div* $a * a = b$

by (*rule dvd-div-mult-self*) *auto*

lemma *unit-div-1-div-1* [*simp*]: *is-unit a* $\Longrightarrow$ *1 div* (*1 div a*) = *a*
by (*erule is-unitE*) *simp*

lemma *unit-div-mult-swap*: *is-unit c* $\Longrightarrow$ *a* $*$ (*b div c*) = (*a* $*$ *b*) *div c*
by (*erule unit-dvdE* [*of* - *b*]) (*simp add*: *mult.left-commute* [*of* - *c*])

lemma *unit-div-commute*: *is-unit b* $\Longrightarrow$ (*a div b*) $*$ *c* = (*a* $*$ *c*) *div b*
using *unit-div-mult-swap* [*of b c a*] **by** (*simp add*: *ac-simps*)

lemma *unit-eq-div1*: *is-unit b* $\Longrightarrow$ *a div b* = *c* $\longleftrightarrow$ *a* = *c* $*$ *b*
by (*auto elim*: *is-unitE*)

lemma *unit-eq-div2*: *is-unit b* $\Longrightarrow$ *a* = *c div b* $\longleftrightarrow$ *a* $*$ *b* = *c*
using *unit-eq-div1* [*of b c a*] **by** *auto*

lemma *unit-mult-left-cancel*: *is-unit a* $\Longrightarrow$ *a* $*$ *b* = *a* $*$ *c* $\longleftrightarrow$ *b* = *c*
using *mult-cancel-left* [*of a b c*] **by** *auto*

lemma *unit-mult-right-cancel*: *is-unit a* $\Longrightarrow$ *b* $*$ *a* = *c* $*$ *a* $\longleftrightarrow$ *b* = *c*
using *unit-mult-left-cancel* [*of a b c*] **by** (*auto simp add*: *ac-simps*)

lemma *unit-div-cancel*:
assumes *is-unit a*
shows *b div a* = *c div a* $\longleftrightarrow$ *b* = *c*
proof −
from *assms* **have** *is-unit* (*1 div a*) **by** *simp*
then have *b* $*$ (*1 div a*) = *c* $*$ (*1 div a*) $\longleftrightarrow$ *b* = *c*
by (*rule unit-mult-right-cancel*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *is-unit-div-mult2-eq*:
assumes *is-unit b* **and** *is-unit c*
shows *a div* (*b* $*$ *c*) = *a div b div c*
proof −
from *assms* **have** *is-unit* (*b* $*$ *c*)
by (*simp add*: *unit-prod*)
then have *b* $*$ *c dvd a*
by (*rule unit-imp-dvd*)
then show *?thesis*
by (*rule dvd-div-mult2-eq*)
qed

lemma *is-unit-div-mult-cancel-left*:
assumes *a* $\neq$ *0* **and** *is-unit b*
shows *a div* (*a* $*$ *b*) = *1 div b*
proof −

from *assms* **have** *a div* $(a * b)$ $=$ *a div a div b*
by (*simp add*: *mult-unit-dvd-iff dvd-div-mult2-eq*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *is-unit-div-mult-cancel-right*:
assumes $a \neq 0$ **and** *is-unit b*
shows *a div* $(b * a)$ $=$ *1 div b*
using *assms is-unit-div-mult-cancel-left* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *unit-div-eq-0-iff*:
assumes *is-unit b*
shows *a div b* $= 0 \longleftrightarrow$ *a* $= 0$
using *assms* **by** (*simp add*: *dvd-div-eq-0-iff unit-imp-dvd*)

lemma *div-mult-unit2*:
is-unit c $\Longrightarrow$ *b dvd a* $\Longrightarrow$ *a div* $(b * c)$ $=$ *a div b div c*
by (*rule dvd-div-mult2-eq*) (*simp-all add*: *mult-unit-dvd-iff*)

Coprimality

definition *coprime* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
where *coprime a b* $\longleftrightarrow$ $(\forall$ *c. c dvd a* $\longrightarrow$ *c dvd b* $\longrightarrow$ *is-unit c*$)$

lemma *coprimeI*:
assumes $\bigwedge$*c. c dvd a* $\Longrightarrow$ *c dvd b* $\Longrightarrow$ *is-unit c*
shows *coprime a b*
using *assms* **by** (*auto simp*: *coprime-def*)

lemma *not-coprimeI*:
assumes *c dvd a* **and** *c dvd b* **and** $\neg$ *is-unit c*
shows $\neg$ *coprime a b*
using *assms* **by** (*auto simp*: *coprime-def*)

lemma *coprime-common-divisor*:
is-unit c **if** *coprime a b* **and** *c dvd a* **and** *c dvd b*
using *that* **by** (*auto simp*: *coprime-def*)

lemma *not-coprimeE*:
assumes $\neg$ *coprime a b*
obtains *c* **where** *c dvd a* **and** *c dvd b* **and** $\neg$ *is-unit c*
using *assms* **by** (*auto simp*: *coprime-def*)

lemma *coprime-imp-coprime*:
coprime a b **if** *coprime c d*
and $\bigwedge$*e.* $\neg$ *is-unit e* $\Longrightarrow$ *e dvd a* $\Longrightarrow$ *e dvd b* $\Longrightarrow$ *e dvd c*
and $\bigwedge$*e.* $\neg$ *is-unit e* $\Longrightarrow$ *e dvd a* $\Longrightarrow$ *e dvd b* $\Longrightarrow$ *e dvd d*
proof (*rule coprimeI*)
fix *e*
assume *e dvd a* **and** *e dvd b*

with *that* **have** *e dvd c* **and** *e dvd d*
by (*auto intro*: *dvd-trans*)
with ‹*coprime c d*› **show** *is-unit e*
by (*rule coprime-common-divisor*)
qed

lemma *coprime-divisors*:
coprime a b **if** *a dvd c b dvd d* **and** *coprime c d*
using ‹*coprime c d*› **proof** (*rule coprime-imp-coprime*)
fix *e*
assume *e dvd a* **then show** *e dvd c*
using ‹*a dvd c*› **by** (*rule dvd-trans*)
assume *e dvd b* **then show** *e dvd d*
using ‹*b dvd d*› **by** (*rule dvd-trans*)
qed

lemma *coprime-self* [*simp*]:
coprime a a ⟷ *is-unit a* (**is** *?P* ⟷ *?Q*)
proof
assume *?P*
then show *?Q*
by (*rule coprime-common-divisor*) *simp-all*
next
assume *?Q*
show *?P*
by (*rule coprimeI*) (*erule dvd-unit-imp-unit*, *rule* ‹*?Q*›)
qed

lemma *coprime-commute* [*ac-simps*]:
coprime b a ⟷ *coprime a b*
unfolding *coprime-def* **by** *auto*

lemma *is-unit-left-imp-coprime*:
coprime a b **if** *is-unit a*
proof (*rule coprimeI*)
fix *c*
assume *c dvd a*
with *that* **show** *is-unit c*
by (*auto intro*: *dvd-unit-imp-unit*)
qed

lemma *is-unit-right-imp-coprime*:
coprime a b **if** *is-unit b*
using *that is-unit-left-imp-coprime* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *coprime-1-left* [*simp*]:
coprime 1 a
by (*rule coprimeI*)

lemma *coprime-1-right* [*simp*]:
coprime a 1
by (*rule coprimeI*)

lemma *coprime-0-left-iff* [*simp*]:
coprime 0 a ⟷ *is-unit a*
by (*auto intro*: *coprimeI dvd-unit-imp-unit coprime-common-divisor* [*of 0 a a*])

lemma *coprime-0-right-iff* [*simp*]:
coprime a 0 ⟷ *is-unit a*
using *coprime-0-left-iff* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *coprime-mult-self-left-iff* [*simp*]:
coprime (*c* ∗ *a*) (*c* ∗ *b*) ⟷ *is-unit c* ∧ *coprime a b*
by (*auto intro*: *coprime-common-divisor*)
(*rule coprimeI*, *auto intro*: *coprime-common-divisor simp add*: *dvd-mult-unit-iff'*)+

lemma *coprime-mult-self-right-iff* [*simp*]:
coprime (*a* ∗ *c*) (*b* ∗ *c*) ⟷ *is-unit c* ∧ *coprime a b*
using *coprime-mult-self-left-iff* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *coprime-absorb-left*:
assumes *x dvd y*
shows *coprime x y* ⟷ *is-unit x*
using *assms coprime-common-divisor is-unit-left-imp-coprime* **by** *auto*

lemma *coprime-absorb-right*:
assumes *y dvd x*
shows *coprime x y* ⟷ *is-unit y*
using *assms coprime-common-divisor is-unit-right-imp-coprime* **by** *auto*

end

class *unit-factor* =
fixes *unit-factor* :: *'a* ⇒ *'a*

class *semidom-divide-unit-factor* = *semidom-divide* + *unit-factor* +
assumes *unit-factor-0* [*simp*]: *unit-factor 0* = *0*
and *is-unit-unit-factor*: *a dvd 1* ⟹ *unit-factor a* = *a*
and *unit-factor-is-unit*: *a* ≠ *0* ⟹ *unit-factor a dvd 1*
and *unit-factor-mult-unit-left*: *a dvd 1* ⟹ *unit-factor* (*a* ∗ *b*) = *a* ∗ *unit-factor b*
— This fine-grained hierarchy will later on allow lean normalization of polynomials
begin

lemma *unit-factor-mult-unit-right*: *a dvd 1* ⟹ *unit-factor* (*b* ∗ *a*) = *unit-factor b* ∗ *a*
using *unit-factor-mult-unit-left*[*of a b*] **by** (*simp add*: *mult-ac*)

lemmas [*simp*] = *unit-factor-mult-unit-left unit-factor-mult-unit-right*

end

class *normalization-semidom* = *algebraic-semidom* + *semidom-divide-unit-factor* +
  fixes *normalize* :: *'a* ⇒ *'a*
  assumes *unit-factor-mult-normalize* [*simp*]: *unit-factor a* ∗ *normalize a* = *a*
    and *normalize-0* [*simp*]: *normalize 0* = *0*
begin

Class *normalization-semidom* cultivates the idea that each integral domain can be split into equivalence classes whose representants are associated, i.e. divide each other. *normalize* specifies a canonical representant for each equivalence class. The rationale behind this is that it is easier to reason about equality than equivalences, hence we prefer to think about equality of normalized values rather than associated elements.

declare *unit-factor-is-unit* [*iff*]

lemma *unit-factor-dvd* [*simp*]: *a* ≠ *0* ⟹ *unit-factor a dvd b*
  by (*rule unit-imp-dvd*) *simp*

lemma *unit-factor-self* [*simp*]: *unit-factor a dvd a*
  by (*cases a* = *0*) *simp-all*

lemma *normalize-mult-unit-factor* [*simp*]: *normalize a* ∗ *unit-factor a* = *a*
  using *unit-factor-mult-normalize* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *normalize-eq-0-iff* [*simp*]: *normalize a* = *0* ⟷ *a* = *0*
  (**is** *?lhs* ⟷ *?rhs*)
proof
  assume *?lhs*
  moreover have *unit-factor a* ∗ *normalize a* = *a* **by** *simp*
  ultimately show *?rhs* **by** *simp*
next
  assume *?rhs*
  then show *?lhs* **by** *simp*
qed

lemma *unit-factor-eq-0-iff* [*simp*]: *unit-factor a* = *0* ⟷ *a* = *0*
  (**is** *?lhs* ⟷ *?rhs*)
proof
  assume *?lhs*
  moreover have *unit-factor a* ∗ *normalize a* = *a* **by** *simp*
  ultimately show *?rhs* **by** *simp*
next
  assume *?rhs*
  then show *?lhs* **by** *simp*
qed

lemma *div-unit-factor* [*simp*]: *a div unit-factor a = normalize a*
proof (*cases a = 0*)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  then have *unit-factor a ≠ 0*
    by *simp*
  with *nonzero-mult-div-cancel-left*
  have *unit-factor a * normalize a div unit-factor a = normalize a*
    by *blast*
  then show *?thesis* **by** *simp*
qed

lemma *normalize-div* [*simp*]: *normalize a div a = 1 div unit-factor a*
proof (*cases a = 0*)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  have *normalize a div a = normalize a div* (*unit-factor a * normalize a*)
    by *simp*
  also have *... = 1 div unit-factor a*
    using *False* **by** (*subst is-unit-div-mult-cancel-right*) *simp-all*
  finally show *?thesis* **.**
qed

lemma *is-unit-normalize*:
  assumes *is-unit a*
  shows *normalize a = 1*
proof −
  from *assms* **have** *unit-factor a = a*
    by (*rule is-unit-unit-factor*)
  moreover from *assms* **have** *a ≠ 0*
    by *auto*
  moreover have *normalize a = a div unit-factor a*
    by *simp*
  ultimately show *?thesis*
    by *simp*
qed

lemma *unit-factor-1* [*simp*]: *unit-factor 1 = 1*
  by (*rule is-unit-unit-factor*) *simp*

lemma *normalize-1* [*simp*]: *normalize 1 = 1*
  by (*rule is-unit-normalize*) *simp*

lemma *normalize-1-iff*: *normalize a = 1 ⟷ is-unit a*

(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
then show *?lhs* **by** (*rule is-unit-normalize*)
next
assume *?lhs*
then have *unit-factor a* $*$ *normalize a* $=$ *unit-factor a* $*$ *1*
by *simp*
then have *unit-factor a* $=$ *a*
by *simp*
moreover
from ‹*?lhs*› **have** *a* $\neq$ *0* **by** *auto*
then have *is-unit* (*unit-factor a*) **by** *simp*
ultimately show *?rhs* **by** *simp*
qed

lemma *div-normalize* [*simp*]: *a div normalize a* $=$ *unit-factor a*
proof (*cases a* $=$ *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *normalize a* $\neq$ *0* **by** *simp*
with *nonzero-mult-div-cancel-right*
have *unit-factor a* $*$ *normalize a div normalize a* $=$ *unit-factor a* **by** *blast*
then show *?thesis* **by** *simp*
qed

lemma *mult-one-div-unit-factor* [*simp*]: *a* $*$ (*1 div unit-factor b*) $=$ *a div unit-factor b*
by (*cases b* $=$ *0*) *simp-all*

lemma *inv-unit-factor-eq-0-iff* [*simp*]:
1 div unit-factor a $=$ *0* $\longleftrightarrow$ *a* $=$ *0*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have *a* $*$ (*1 div unit-factor a*) $=$ *a* $*$ *0*
by *simp*
then show *?rhs*
by *simp*
next
assume *?rhs*
then show *?lhs* **by** *simp*
qed

lemma *unit-factor-idem* [*simp*]: *unit-factor* (*unit-factor a*) $=$ *unit-factor a*
by (*cases a* $=$ *0*) (*auto intro*: *is-unit-unit-factor*)

lemma *normalize-unit-factor* [*simp*]: $a \neq 0 \Longrightarrow$ *normalize* (*unit-factor a*) = *1*
by (*rule is-unit-normalize*) *simp*

lemma *normalize-mult-unit-left* [*simp*]:
assumes *a dvd 1*
shows *normalize* (*a* ∗ *b*) = *normalize b*
proof (*cases b* = *0*)
case *False*
have *a* ∗ *unit-factor b* ∗ *normalize* (*a* ∗ *b*) = *unit-factor* (*a* ∗ *b*) ∗ *normalize* (*a* ∗ *b*)
using *assms* **by** (*subst unit-factor-mult-unit-left*) *auto*
also have ... = *a* ∗ *b* **by** *simp*
also have *b* = *unit-factor b* ∗ *normalize b* **by** *simp*
hence *a* ∗ *b* = *a* ∗ *unit-factor b* ∗ *normalize b*
by (*simp only*: *mult-ac*)
finally show *?thesis*
using *assms False* **by** *auto*
qed *auto*

lemma *normalize-mult-unit-right* [*simp*]:
assumes *b dvd 1*
shows *normalize* (*a* ∗ *b*) = *normalize a*
using *assms* **by** (*subst mult.commute*) *auto*

lemma *normalize-idem* [*simp*]: *normalize* (*normalize a*) = *normalize a*
proof (*cases a* = *0*)
case *False*
have *normalize a* = *normalize* (*unit-factor a* ∗ *normalize a*)
by *simp*
also from *False* **have** ... = *normalize* (*normalize a*)
by (*subst normalize-mult-unit-left*) *auto*
finally show *?thesis* **..**
qed *auto*

lemma *unit-factor-normalize* [*simp*]:
assumes *a* ≠ *0*
shows *unit-factor* (*normalize a*) = *1*
proof −
from *assms* **have** ∗: *normalize a* ≠ *0*
by *simp*
have *unit-factor* (*normalize a*) ∗ *normalize* (*normalize a*) = *normalize a*
by (*simp only*: *unit-factor-mult-normalize*)
then have *unit-factor* (*normalize a*) ∗ *normalize a* = *normalize a*
by *simp*
with ∗ **have** *unit-factor* (*normalize a*) ∗ *normalize a div normalize a* = *normalize a div normalize a*
by *simp*
with ∗ **show** *?thesis*
by *simp*

qed

lemma *normalize-dvd-iff* [*simp*]: *normalize a dvd b* ⟷ *a dvd b*
proof −
  have *normalize a dvd b* ⟷ *unit-factor a* ∗ *normalize a dvd b*
    using *mult-unit-dvd-iff* [*of unit-factor a normalize a b*]
      by (*cases a* = *0*) *simp-all*
  then show *?thesis* **by** *simp*
qed

lemma *dvd-normalize-iff* [*simp*]: *a dvd normalize b* ⟷ *a dvd b*
proof −
  have *a dvd normalize b* ⟷ *a dvd normalize b* ∗ *unit-factor b*
    using *dvd-mult-unit-iff* [*of unit-factor b a normalize b*]
      by (*cases b* = *0*) *simp-all*
  then show *?thesis* **by** *simp*
qed

lemma *normalize-idem-imp-unit-factor-eq*:
  assumes *normalize a* = *a*
  shows *unit-factor a* = *of-bool* (*a* ≠ *0*)
proof (*cases a* = *0*)
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  then show *?thesis*
    using *assms unit-factor-normalize* [*of a*] **by** *simp*
qed

lemma *normalize-idem-imp-is-unit-iff*:
  assumes *normalize a* = *a*
  shows *is-unit a* ⟷ *a* = *1*
  using *assms* **by** (*cases a* = *0*) (*auto dest*: *is-unit-normalize*)

lemma *coprime-normalize-left-iff* [*simp*]:
  coprime (*normalize a*) *b* ⟷ *coprime a b*
  by (*rule iffI*; *rule coprimeI*) (*auto intro*: *coprime-common-divisor*)

lemma *coprime-normalize-right-iff* [*simp*]:
  coprime a (*normalize b*) ⟷ *coprime a b*
  using *coprime-normalize-left-iff* [*of b a*] **by** (*simp add*: *ac-simps*)

We avoid an explicit definition of associated elements but prefer explicit normalisation instead. In theory we could define an abbreviation like *associated a b* = (*normalize a* = *normalize b*) but this is counterproductive without suggestive infix syntax, which we do not want to sacrifice for this purpose here.

lemma *associatedI*:
assumes *a dvd b* **and** *b dvd a*
shows *normalize a = normalize b*
proof (*cases a = 0 ∨ b = 0*)
case *True*
with *assms* **show** *?thesis* **by** *auto*
next
case *False*
from ‹*a dvd b*› **obtain** *c* **where** *b*: *b = a * c* **..**
moreover from ‹*b dvd a*› **obtain** *d* **where** *a*: *a = b * d* **..**
ultimately have *b * 1 = b * (c * d)*
by (*simp add*: *ac-simps*)
with *False* **have** *1 = c * d*
unfolding *mult-cancel-left* **by** *simp*
then have *is-unit c* **and** *is-unit d*
by *auto*
with *a b* **show** *?thesis*
by (*simp add*: *is-unit-normalize*)
qed

lemma *associatedD1*: *normalize a = normalize b ⟹ a dvd b*
using *dvd-normalize-iff* [*of - b*, *symmetric*] *normalize-dvd-iff* [*of a -*, *symmetric*]
by *simp*

lemma *associatedD2*: *normalize a = normalize b ⟹ b dvd a*
using *dvd-normalize-iff* [*of - a*, *symmetric*] *normalize-dvd-iff* [*of b -*, *symmetric*]
by *simp*

lemma *associated-unit*: *normalize a = normalize b ⟹ is-unit a ⟹ is-unit b*
using *dvd-unit-imp-unit* **by** (*auto dest*!: *associatedD1 associatedD2*)

lemma *associated-iff-dvd*: *normalize a = normalize b ⟷ a dvd b ∧ b dvd a*
(**is** *?lhs ⟷ ?rhs*)
proof
assume *?rhs*
then show *?lhs* **by** (*auto intro*!: *associatedI*)
next
assume *?lhs*
then have *unit-factor a * normalize a = unit-factor a * normalize b*
by *simp*
then have *: *normalize b * unit-factor a = a*
by (*simp add*: *ac-simps*)
show *?rhs*
proof (*cases a = 0 ∨ b = 0*)
case *True*
with ‹*?lhs*› **show** *?thesis* **by** *auto*
next
case *False*
then have *b dvd normalize b * unit-factor a* **and** *normalize b * unit-factor a*

dvd b
by (*simp-all add*: *mult-unit-dvd-iff dvd-mult-unit-iff*)
with ∗ **show** *?thesis* **by** *simp*
qed
qed

lemma *associated-eqI*:
assumes *a dvd b* **and** *b dvd a*
assumes *normalize a* = *a* **and** *normalize b* = *b*
shows *a* = *b*
proof −
from *assms* **have** *normalize a* = *normalize b*
unfolding *associated-iff-dvd* **by** *simp*
with ‹*normalize a* = *a*› **have** *a* = *normalize b*
by *simp*
with ‹*normalize b* = *b*› **show** *a* = *b*
by *simp*
qed

lemma *normalize-unit-factor-eqI*:
assumes *normalize a* = *normalize b*
and *unit-factor a* = *unit-factor b*
shows *a* = *b*
proof −
from *assms* **have** *unit-factor a* ∗ *normalize a* = *unit-factor b* ∗ *normalize b*
by *simp*
then show *?thesis*
by *simp*
qed

lemma *normalize-mult-normalize-left* [*simp*]: *normalize* (*normalize a* ∗ *b*) = *normalize* (*a* ∗ *b*)
by (*rule associated-eqI*) (*auto intro*!: *mult-dvd-mono*)

lemma *normalize-mult-normalize-right* [*simp*]: *normalize* (*a* ∗ *normalize b*) = *normalize* (*a* ∗ *b*)
by (*rule associated-eqI*) (*auto intro*!: *mult-dvd-mono*)

end

class *normalization-semidom-multiplicative* = *normalization-semidom* +
assumes *unit-factor-mult*: *unit-factor* (*a* ∗ *b*) = *unit-factor a* ∗ *unit-factor b*
begin

lemma *normalize-mult*: *normalize* (*a* ∗ *b*) = *normalize a* ∗ *normalize b*
proof (*cases a* = *0* ∨ *b* = *0*)
case *True*
then show *?thesis* **by** *auto*

next
case *False*
have *unit-factor* (*a* ∗ *b*) ∗ *normalize* (*a* ∗ *b*) = *a* ∗ *b*
by (*rule unit-factor-mult-normalize*)
then have *normalize* (*a* ∗ *b*) = *a* ∗ *b div unit-factor* (*a* ∗ *b*)
by *simp*
also have ... = *a* ∗ *b div unit-factor* (*b* ∗ *a*)
by (*simp add*: *ac-simps*)
also have ... = *a* ∗ *b div unit-factor b div unit-factor a*
using *False* **by** (*simp add*: *unit-factor-mult is-unit-div-mult2-eq* [*symmetric*])
also have ... = *a* ∗ (*b div unit-factor b*) *div unit-factor a*
using *False* **by** (*subst unit-div-mult-swap*) *simp-all*
also have ... = *normalize a* ∗ *normalize b*
using *False*
by (*simp add*: *mult.commute* [*of a*] *mult.commute* [*of normalize a*] *unit-div-mult-swap* [*symmetric*])
finally show *?thesis* **.**
qed

lemma *dvd-unit-factor-div*:
assumes *b dvd a*
shows *unit-factor* (*a div b*) = *unit-factor a div unit-factor b*
proof −
from *assms* **have** *a* = *a div b* ∗ *b*
by *simp*
then have *unit-factor a* = *unit-factor* (*a div b* ∗ *b*)
by *simp*
then show *?thesis*
by (*cases b* = *0*) (*simp-all add*: *unit-factor-mult*)
qed

lemma *dvd-normalize-div*:
assumes *b dvd a*
shows *normalize* (*a div b*) = *normalize a div normalize b*
proof −
from *assms* **have** *a* = *a div b* ∗ *b*
by *simp*
then have *normalize a* = *normalize* (*a div b* ∗ *b*)
by *simp*
then show *?thesis*
by (*cases b* = *0*) (*simp-all add*: *normalize-mult*)
qed

end

Syntactic division remainder operator

class *modulo* = *dvd* + *divide* +
fixes *modulo* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ‹*mod*› *70*)

Arbitrary quotient and remainder partitions

class *semiring-modulo* = *comm-semiring-1-cancel* + *divide* + *modulo* +
  assumes *div-mult-mod-eq*: ‹*a div b* * *b* + *a mod b* = *a*›
begin

lemma *mod-div-decomp*:
  fixes *a b*
  obtains *q r* **where** *q* = *a div b* **and** *r* = *a mod b*
    and *a* = *q* * *b* + *r*
proof −
  from *div-mult-mod-eq* **have** *a* = *a div b* * *b* + *a mod b* **by** *simp*
  moreover have *a div b* = *a div b* **..**
  moreover have *a mod b* = *a mod b* **..**
  note *that* **ultimately show** *thesis* **by** *blast*
qed

lemma *mult-div-mod-eq*: *b* * (*a div b*) + *a mod b* = *a*
  using *div-mult-mod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *mod-div-mult-eq*: *a mod b* + *a div b* * *b* = *a*
  using *div-mult-mod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *mod-mult-div-eq*: *a mod b* + *b* * (*a div b*) = *a*
  using *div-mult-mod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *minus-div-mult-eq-mod*: *a* − *a div b* * *b* = *a mod b*
  by (*rule add-implies-diff* [*symmetric*]) (*fact mod-div-mult-eq*)

lemma *minus-mult-div-eq-mod*: *a* − *b* * (*a div b*) = *a mod b*
  by (*rule add-implies-diff* [*symmetric*]) (*fact mod-mult-div-eq*)

lemma *minus-mod-eq-div-mult*: *a* − *a mod b* = *a div b* * *b*
  by (*rule add-implies-diff* [*symmetric*]) (*fact div-mult-mod-eq*)

lemma *minus-mod-eq-mult-div*: *a* − *a mod b* = *b* * (*a div b*)
  by (*rule add-implies-diff* [*symmetric*]) (*fact mult-div-mod-eq*)

lemma *mod-0-imp-dvd* [*dest*!]:
  b dvd a **if** *a mod b* = *0*
proof −
  have *b dvd* (*a div b*) * *b* **by** *simp*
  also have (*a div b*) * *b* = *a*
    using *div-mult-mod-eq* [*of a b*] **by** (*simp add*: *that*)
  finally show *?thesis* **.**
qed

lemma [*nitpick-unfold*]:
  a mod b = *a* − *a div b* * *b*
  by (*fact minus-div-mult-eq-mod* [*symmetric*])

end

class *semiring-modulo-trivial* = *semiring-modulo* + *divide-trivial*
begin

lemma *mod-0* [*simp*]:
‹*0 mod a = 0*›
using *div-mult-mod-eq* [*of 0 a*] **by** *simp*

lemma *mod-by-0* [*simp*]:
‹*a mod 0 = a*›
using *div-mult-mod-eq* [*of a 0*] **by** *simp*

lemma *mod-by-1* [*simp*]:
‹*a mod 1 = 0*›
proof −
have ‹*a + a mod 1 = a*›
using *div-mult-mod-eq* [*of a 1*] **by** *simp*
then have ‹*a + a mod 1 = a + 0*›
by *simp*
then show *?thesis*
by (*rule add-left-imp-eq*)
qed

end

16.5 Quotient and remainder in integral domains

class *semidom-modulo* = *algebraic-semidom* + *semiring-modulo*
begin

subclass *semiring-modulo-trivial* **..**

lemma *mod-self* [*simp*]:
a mod a = 0
using *div-mult-mod-eq* [*of a a*] **by** *simp*

lemma *dvd-imp-mod-0* [*simp*]:
b mod a = 0 **if** *a dvd b*
using *that minus-div-mult-eq-mod* [*of b a*] **by** *simp*

lemma *mod-eq-0-iff-dvd*:
a mod b = 0 ⟷ *b dvd a*
by (*auto intro*: *mod-0-imp-dvd*)

lemma *dvd-eq-mod-eq-0* [*nitpick-unfold*, *code*]:
a dvd b ⟷ *b mod a = 0*
by (*simp add*: *mod-eq-0-iff-dvd*)

lemma *dvd-mod-iff*:
assumes *c dvd b*
shows *c dvd a mod b* ⟷ *c dvd a*
proof −
from *assms* **have** (*c dvd a mod b*) ⟷ (*c dvd* ((*a div b*) ∗ *b* + *a mod b*))
by (*simp add*: *dvd-add-right-iff*)
also have (*a div b*) ∗ *b* + *a mod b* = *a*
using *div-mult-mod-eq* [*of a b*] **by** *simp*
finally show *?thesis* **.**
qed

lemma *dvd-mod-imp-dvd*:
assumes *c dvd a mod b* **and** *c dvd b*
shows *c dvd a*
using *assms dvd-mod-iff* [*of c b a*] **by** *simp*

lemma *dvd-minus-mod* [*simp*]:
b dvd a − *a mod b*
by (*simp add*: *minus-mod-eq-div-mult*)

lemma *cancel-div-mod-rules*:
((*a div b*) ∗ *b* + *a mod b*) + *c* = *a* + *c*
(*b* ∗ (*a div b*) + *a mod b*) + *c* = *a* + *c*
by (*simp-all add*: *div-mult-mod-eq mult-div-mod-eq*)

end

class *idom-modulo* = *idom* + *semidom-modulo*
begin

subclass *idom-divide* **..**

lemma *div-diff* [*simp*]:
c dvd a ⟹ *c dvd b* ⟹ (*a* − *b*) *div c* = *a div c* − *b div c*
using *div-add* [*of* - - − *b*] **by** (*simp add*: *dvd-neg-div*)

end

16.6 Interlude: basic tool support for algebraic and arithmetic calculations

named-theorems *arith arith facts* −− *only ground formulas*
ML-file ‹*Tools/arith-data.ML*›

ML-file ‹~~/*src/Provers/Arith/cancel-div-mod.ML*›

ML ‹
structure Cancel-Div-Mod-Ring = *Cancel-Div-Mod*
(

```
  val div-name = const-name ‹divide›;
  val mod-name = const-name ‹modulo›;
  val mk-binop = HOLogic.mk-binop;
  val mk-sum = Arith-Data.mk-sum;
  val dest-sum = Arith-Data.dest-sum;

  val div-mod-eqs = map mk-meta-eq @{thms cancel-div-mod-rules};

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac (Arith-Data.simp-all-tac
    @{thms diff-conv-add-uminus add-0-left add-0-right ac-simps})
)
›

simproc-setup cancel-div-mod-int ((a::'a::semidom-modulo) + b) =
  ‹K Cancel-Div-Mod-Ring.proc›
```

16.7 Ordered semirings and rings

The theory of partially ordered rings is taken from the books:

- *Lattice Theory* by Garret Birkhoff, American Mathematical Society, 1979
- *Partially Ordered Algebraic Systems*, Pergamon Press, 1963

Most of the used notions can also be looked up in

- http://www.mathworld.com by Eric Weisstein et. al.
- *Algebra I* by van der Waerden, Springer

```
class ordered-semiring = semiring + ordered-comm-monoid-add +
  assumes mult-left-mono: a ≤ b ⟹ 0 ≤ c ⟹ c * a ≤ c * b
  assumes mult-right-mono: a ≤ b ⟹ 0 ≤ c ⟹ a * c ≤ b * c
begin

lemma mult-mono: a ≤ b ⟹ c ≤ d ⟹ 0 ≤ b ⟹ 0 ≤ c ⟹ a * c ≤ b * d
  apply (erule (1) mult-right-mono [THEN order-trans])
  apply (erule (1) mult-left-mono)
  done

lemma mult-mono': a ≤ b ⟹ c ≤ d ⟹ 0 ≤ a ⟹ 0 ≤ c ⟹ a * c ≤ b * d
  by (rule mult-mono) (fast intro: order-trans)+

end

lemma mono-mult:
  fixes a :: 'a::ordered-semiring
```

shows $a \geq 0 \Longrightarrow \mathit{mono}\ ((*)\ a)$
by (*simp add*: *mono-def mult-left-mono*)

class *ordered-semiring-0* = *semiring-0* + *ordered-semiring*
begin

lemma *mult-nonneg-nonneg* [*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * b$
using *mult-left-mono* [*of 0 b a*] **by** *simp*

lemma *mult-nonneg-nonpos*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow a * b \leq 0$
using *mult-left-mono* [*of b 0 a*] **by** *simp*

lemma *mult-nonpos-nonneg*: $a \leq 0 \Longrightarrow 0 \leq b \Longrightarrow a * b \leq 0$
using *mult-right-mono* [*of a 0 b*] **by** *simp*

Legacy – use *mult-nonpos-nonneg*.

lemma *mult-nonneg-nonpos2*: $0 \leq a \Longrightarrow b \leq 0 \Longrightarrow b * a \leq 0$
by (*drule mult-right-mono* [*of b 0*]) *auto*

lemma *split-mult-neg-le*: $(0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b) \Longrightarrow a * b \leq 0$
by (*auto simp add*: *mult-nonneg-nonpos mult-nonneg-nonpos2*)

end

class *zero-less-one* = *order* + *zero* + *one* +
assumes *zero-less-one* [*simp*]: $0 < 1$
begin

subclass *zero-neq-one*
by *standard* (*simp add*: *less-imp-neq*)

lemma *zero-le-one* [*simp*]:
‹$0 \leq 1$› **by** (*rule less-imp-le*) *simp*

end

class *ordered-semiring-1* = *ordered-semiring-0* + *semiring-1* + *zero-less-one*
begin

lemma *convex-bound-le*:
assumes $x \leq a$ **and** $y \leq a$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
shows $u * x + v * y \leq a$
proof−
from *assms* **have** $u * x + v * y \leq u * a + v * a$
by (*simp add*: *add-mono mult-left-mono*)
with *assms* **show** *?thesis*
unfolding *distrib-right*[*symmetric*] **by** *simp*
qed

end

class *ordered-cancel-semiring* = *ordered-semiring* + *cancel-comm-monoid-add*
begin

subclass *semiring-0-cancel* **..**

subclass *ordered-semiring-0* **..**

subclass *ordered-cancel-ab-semigroup-add* **..**

end

class *linordered-semiring* = *ordered-semiring* + *linordered-cancel-ab-semigroup-add*
begin

subclass *ordered-cancel-semiring* **..**

subclass *ordered-cancel-comm-monoid-add* **..**

subclass *ordered-ab-semigroup-monoid-add-imp-le* **..**

lemma *mult-left-less-imp-less*: $c * a < c * b \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-left-mono not-le* [*symmetric*])

lemma *mult-right-less-imp-less*: $a * c < b * c \Longrightarrow 0 \leq c \Longrightarrow a < b$
by (*force simp add*: *mult-right-mono not-le* [*symmetric*])

end

class *ordered-semiring-strict* = *semiring* + *comm-monoid-add* + *ordered-cancel-ab-semigroup-add*
+
assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* **..**

subclass *ordered-semiring*
proof
fix *a b c* :: *'a*
assume §: $a \leq b$ $0 \leq c$
thus $c * a \leq c * b$
unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
show $a * c \leq b * c$
using § **by** (*force simp*: *le-less intro*: *mult-strict-right-mono dest*: *sym*)
qed

lemma *mult-pos-pos*[*simp*]: $0 < a \Longrightarrow 0 < b \Longrightarrow 0 < a * b$
using *mult-strict-left-mono* [*of 0 b a*] **by** *simp*

lemma *mult-pos-neg*: $0 < a \Longrightarrow b < 0 \Longrightarrow a * b < 0$
using *mult-strict-left-mono* [*of b 0 a*] **by** *simp*

lemma *mult-neg-pos*: $a < 0 \Longrightarrow 0 < b \Longrightarrow a * b < 0$
using *mult-strict-right-mono* [*of a 0 b*] **by** *simp*

Strict monotonicity in both arguments

lemma *mult-strict-mono*:
assumes $a < b$ $c < d$ $0 < b$ $0 \leq c$
shows $a * c < b * d$
proof−
have $a * c \leq b * c$
using *assms* **by** (*intro mult-right-mono*) *auto*
also have ... $< b * d$
using *assms* **by** (*intro mult-strict-left-mono*) *auto*
finally show *?thesis* **.**
qed

This weaker variant has more natural premises

lemma *mult-strict-mono′*:
assumes $a < b$ **and** $c < d$ **and** $0 \leq a$ **and** $0 \leq c$
shows $a * c < b * d$
using *assms* **by** (*intro mult-strict-mono*) *auto*

lemma *mult-less-le-imp-less*:
assumes $a < b$ $c \leq d$ $0 \leq a$ $0 < c$
shows $a * c < b * d$
proof−
have $a * c < b * c$
using *assms* **by** (*intro mult-strict-right-mono*) *auto*
also have ... $\leq b * d$
using *assms* **by** (*intro mult-left-mono*) *auto*
finally show *?thesis* **.**
qed

lemma *mult-le-less-imp-less*:
assumes $a \leq b$ **and** $c < d$ **and** $0 < a$ **and** $0 \leq c$
shows $a * c < b * d$
proof−
have $a * c \leq b * c$
using *assms* **by** (*intro mult-right-mono*) *auto*
also have ... $< b * d$
using *assms* **by** (*intro mult-strict-left-mono*) *auto*
finally show *?thesis* **.**
qed

end

class *linordered-semiring-1* = *linordered-semiring* + *semiring-1* + *zero-less-one*
begin

lemma *convex-bound-le*:
  assumes $x \leq a$ $y \leq a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
  shows $u * x + v * y \leq a$
proof−
  from *assms* **have** $u * x + v * y \leq u * a + v * a$
    by (*simp add*: *add-mono mult-left-mono*)
  with *assms* **show** *?thesis*
    unfolding *distrib-right*[*symmetric*] **by** *simp*
qed

end

subclass (**in** *linordered-semiring-1*) *ordered-semiring-1* **..**

class *linordered-semiring-strict* = *semiring* + *comm-monoid-add* + *linordered-cancel-ab-semigroup-add*
+
  assumes *mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
  assumes *mult-strict-right-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow a * c < b * c$
begin

subclass *semiring-0-cancel* **..**

subclass *linordered-semiring*
proof
  fix a b c :: $'a$
  assume $*$: $a \leq b$ $0 \leq c$
  then show $c * a \leq c * b$
    unfolding *le-less*
    using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
  from $*$ **show** $a * c \leq b * c$
    unfolding *le-less*
    using *mult-strict-right-mono* **by** (*cases* $c = 0$) *auto*
qed

subclass (**in** *linordered-semiring-strict*) *ordered-semiring-strict*
proof qed (*auto simp*: *mult-strict-left-mono mult-strict-right-mono*)

lemma *mult-left-le-imp-le*: $c * a \leq c * b \Longrightarrow 0 < c \Longrightarrow a \leq b$
  by (*auto simp add*: *mult-strict-left-mono simp flip*: *not-less*)

lemma *mult-right-le-imp-le*: $a * c \leq b * c \Longrightarrow 0 < c \Longrightarrow a \leq b$
  by (*auto simp add*: *mult-strict-right-mono not-less* [*symmetric*])

lemma *zero-less-mult-pos*:
assumes $0 < a * b$ $0 < a$ **shows** $0 < b$
proof (*cases* $b \leq 0$)
case *True*
then show *?thesis*
using *assms* **by** (*auto simp*: *le-less dest*: *less-not-sym mult-pos-neg* [*of a b*])
qed (*auto simp add*: *le-less not-less*)

lemma *zero-less-mult-pos2*:
assumes $0 < b * a$ $0 < a$ **shows** $0 < b$
proof (*cases* $b \leq 0$)
case *True*
then show *?thesis*
using *assms* **by** (*auto simp*: *le-less dest*: *less-not-sym mult-neg-pos*)
qed (*auto simp add*: *le-less not-less*)

end

class *linordered-semiring-1-strict* = *linordered-semiring-strict* + *semiring-1* + *zero-less-one*
begin

subclass *linordered-semiring-1* **..**

lemma *convex-bound-lt*:
assumes $x < a$ $y < a$ $0 \leq u$ $0 \leq v$ $u + v = 1$
shows $u * x + v * y < a$
proof −
from *assms* **have** $u * x + v * y < u * a + v * a$
by (*cases* $u = 0$) (*auto intro*!: *add-less-le-mono mult-strict-left-mono mult-left-mono*)
with *assms* **show** *?thesis*
unfolding *distrib-right*[*symmetric*] **by** *simp*
qed

end

class *ordered-semiring-1-strict* = *ordered-semiring-strict* + *semiring-1* + *zero-less-one*
— analogous to *linordered-semiring-1-strict* not requiring a total order
begin

subclass *ordered-semiring-1* **..**

lemma *convex-bound-lt*:
assumes $x < a$ **and** $y < a$ **and** $0 \leq u$ **and** $0 \leq v$ **and** $u + v = 1$
shows $u * x + v * y < a$
proof −
from *assms* **have** $u * x + v * y < u * a + v * a$
by (*cases* $u = 0$) (*auto intro*!: *add-less-le-mono mult-strict-left-mono mult-left-mono*)
with *assms* **show** *?thesis*
unfolding *distrib-right*[*symmetric*] **by** *simp*

qed

end

subclass (**in** *linordered-semiring-1-strict*) *ordered-semiring-1-strict* **..**

class *ordered-comm-semiring* = *comm-semiring-0* + *ordered-ab-semigroup-add* +
  assumes *comm-mult-left-mono*: $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
begin

subclass *ordered-semiring*
proof
  fix a b c :: $'a$
  assume $a \leq b$ $0 \leq c$
  then show $c * a \leq c * b$ **by** (*rule comm-mult-left-mono*)
  then show $a * c \leq b * c$ **by** (*simp only*: *mult.commute*)
qed

end

class *ordered-cancel-comm-semiring* = *ordered-comm-semiring* + *cancel-comm-monoid-add*
begin

subclass *comm-semiring-0-cancel* **..**
subclass *ordered-comm-semiring* **..**
subclass *ordered-cancel-semiring* **..**

end

class *linordered-comm-semiring-strict* = *comm-semiring-0* + *linordered-cancel-ab-semigroup-add*
+
  assumes *comm-mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *linordered-semiring-strict*
proof
  fix a b c :: $'a$
  assume $a < b$ $0 < c$
  then show $c * a < c * b$
    by (*rule comm-mult-strict-left-mono*)
  then show $a * c < b * c$
    by (*simp only*: *mult.commute*)
qed

subclass *ordered-cancel-comm-semiring*
proof
  fix a b c :: $'a$
  assume $a \leq b$ $0 \leq c$
  then show $c * a \leq c * b$

unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
qed

end

class *ordered-comm-semiring-strict* = *comm-semiring-0* + *ordered-cancel-ab-semigroup-add*
+
— analogous to *linordered-comm-semiring-strict* not requiring a total order
assumes *comm-mult-strict-left-mono*: $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
begin

subclass *ordered-semiring-strict*
proof
fix $a\ b\ c :: 'a$
assume $a < b$ **and** $0 < c$
thus $c * a < c * b$
by (*rule comm-mult-strict-left-mono*)
thus $a * c < b * c$
by (*simp only*: *mult.commute*)
qed

subclass *ordered-cancel-comm-semiring*
proof
fix $a\ b\ c :: 'a$
assume $a \leq b$ **and** $0 \leq c$
thus $c * a \leq c * b$
unfolding *le-less*
using *mult-strict-left-mono* **by** (*cases* $c = 0$) *auto*
qed

end

subclass (**in** *linordered-comm-semiring-strict*) *ordered-comm-semiring-strict*
proof qed (*simp add*: *local.mult-strict-left-mono*)

class *ordered-ring* = *ring* + *ordered-cancel-semiring*
begin

subclass *ordered-ab-group-add* **..**

lemma *less-add-iff1*: $a * e + c < b * e + d \longleftrightarrow (a - b) * e + c < d$
by (*simp add*: *algebra-simps*)

lemma *less-add-iff2*: $a * e + c < b * e + d \longleftrightarrow c < (b - a) * e + d$
by (*simp add*: *algebra-simps*)

lemma *le-add-iff1*: $a * e + c \leq b * e + d \longleftrightarrow (a - b) * e + c \leq d$
by (*simp add*: *algebra-simps*)

lemma *le-add-iff2*: $a * e + c \leq b * e + d \longleftrightarrow c \leq (b - a) * e + d$
by (*simp add*: *algebra-simps*)

lemma *mult-left-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow c * a \leq c * b$
by (*auto dest*: *mult-left-mono* [*of* - - − *c*])

lemma *mult-right-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a * c \leq b * c$
by (*auto dest*: *mult-right-mono* [*of* - - − *c*])

lemma *mult-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a * b$
using *mult-right-mono-neg* [*of a 0 b*] **by** *simp*

lemma *split-mult-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a * b$
by (*auto simp add*: *mult-nonpos-nonpos*)

end

class *abs-if* = *minus* + *uminus* + *ord* + *zero* + *abs* +
assumes *abs-if*: $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$

class *linordered-ring* = *ring* + *linordered-semiring* + *linordered-ab-group-add* + *abs-if*
begin

subclass *ordered-ring* **..**

subclass *ordered-ab-group-add-abs*
proof
fix *a b*
show $|a + b| \leq |a| + |b|$
by (*auto simp add*: *abs-if not-le not-less algebra-simps*
simp del: *add.commute dest*: *add-neg-neg add-nonneg-nonneg*)
qed (*auto simp*: *abs-if*)

lemma *zero-le-square* [*simp*]: $0 \leq a * a$
using *linear* [*of 0 a*] **by** (*auto simp add*: *mult-nonpos-nonpos*)

lemma *not-square-less-zero* [*simp*]: $\neg\ (a * a < 0)$
by (*simp add*: *not-less*)

proposition *abs-eq-iff*: $|x| = |y| \longleftrightarrow x = y \vee x = -y$
by (*auto simp add*: *abs-if split*: *if-split-asm*)

lemma *abs-eq-iff'*:
$|a| = b \longleftrightarrow b \geq 0 \wedge (a = b \vee a = - b)$
by (*cases a* $\geq$ *0*) *auto*

lemma *eq-abs-iff'*:

```
  a = |b| ⟷ a ≥ 0 ∧ (b = a ∨ b = - a)
  using abs-eq-iff' [of b a] by auto

lemma sum-squares-ge-zero: 0 ≤ x * x + y * y
  by (intro add-nonneg-nonneg zero-le-square)

lemma not-sum-squares-lt-zero: ¬ x * x + y * y < 0
  by (simp add: not-less sum-squares-ge-zero)

end

class linordered-ring-strict = ring + linordered-semiring-strict
  + ordered-ab-group-add + abs-if
begin

subclass linordered-ring ..

lemma mult-strict-left-mono-neg: b < a ⟹ c < 0 ⟹ c * a < c * b
  using mult-strict-left-mono [of b a - c] by simp

lemma mult-strict-right-mono-neg: b < a ⟹ c < 0 ⟹ a * c < b * c
  using mult-strict-right-mono [of b a - c] by simp

lemma mult-neg-neg: a < 0 ⟹ b < 0 ⟹ 0 < a * b
  using mult-strict-right-mono-neg [of a 0 b] by simp

subclass ring-no-zero-divisors
proof
  fix a b
  assume a ≠ 0
  then have a: a < 0 ∨ 0 < a by (simp add: neq-iff)
  assume b ≠ 0
  then have b: b < 0 ∨ 0 < b by (simp add: neq-iff)
  have a * b < 0 ∨ 0 < a * b
  proof (cases a < 0)
    case True
    show ?thesis
    proof (cases b < 0)
      case True
      with ‹a < 0› show ?thesis by (auto dest: mult-neg-neg)
    next
      case False
      with b have 0 < b by auto
      with ‹a < 0› show ?thesis by (auto dest: mult-strict-right-mono)
    qed
  next
    case False
    with a have 0 < a by auto
    show ?thesis
```

```
    proof (cases b < 0)
      case True
      with ‹0 < a› show ?thesis
        by (auto dest: mult-strict-right-mono-neg)
    next
      case False
      with b have 0 < b by auto
      with ‹0 < a› show ?thesis by auto
    qed
  qed
  then show a * b ≠ 0
    by (simp add: neq-iff)
qed

lemma zero-less-mult-iff [algebra-split-simps, field-split-simps]:
  0 < a * b ⟷ 0 < a ∧ 0 < b ∨ a < 0 ∧ b < 0
  by (cases a 0 b 0 rule: linorder-cases[case-product linorder-cases])
    (auto simp add: mult-neg-neg not-less le-less dest: zero-less-mult-pos zero-less-mult-pos2)

lemma zero-le-mult-iff [algebra-split-simps, field-split-simps]:
  0 ≤ a * b ⟷ 0 ≤ a ∧ 0 ≤ b ∨ a ≤ 0 ∧ b ≤ 0
  by (auto simp add: eq-commute [of 0] le-less not-less zero-less-mult-iff)

lemma mult-less-0-iff [algebra-split-simps, field-split-simps]:
  a * b < 0 ⟷ 0 < a ∧ b < 0 ∨ a < 0 ∧ 0 < b
  using zero-less-mult-iff [of − a b] by auto

lemma mult-le-0-iff [algebra-split-simps, field-split-simps]:
  a * b ≤ 0 ⟷ 0 ≤ a ∧ b ≤ 0 ∨ a ≤ 0 ∧ 0 ≤ b
  using zero-le-mult-iff [of − a b] by auto
```

Cancellation laws for $c * a < c * b$ and $a * c < b * c$, also with the relations $\leq$ and equality.

These "disjunction" versions produce two cases when the comparison is an assumption, but effectively four when the comparison is a goal.

```
lemma mult-less-cancel-right-disj: a * c < b * c ⟷ 0 < c ∧ a < b ∨ c < 0 ∧
b < a
proof (cases c = 0)
  case False
  show ?thesis (is ?lhs ⟷ ?rhs)
  proof
    assume ?lhs
    then have c < 0 ⟹ b < a c > 0 ⟹ b > a
      by (auto simp flip: not-le intro: mult-right-mono mult-right-mono-neg)
    with False show ?rhs
      by (auto simp add: neq-iff)
  next
    assume ?rhs
```

with *False* **show** *?lhs*
by (*auto simp add*: *mult-strict-right-mono mult-strict-right-mono-neg*)
qed
qed *auto*

lemma *mult-less-cancel-left-disj*: $c * a < c * b \longleftrightarrow 0 < c \wedge a < b \vee c < 0 \wedge b < a$
proof (*cases* $c = 0$)
case *False*
show *?thesis* (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have $c < 0 \Longrightarrow b < a$ $c > 0 \Longrightarrow b > a$
by (*auto simp flip*: *not-le intro*: *mult-left-mono mult-left-mono-neg*)
with *False* **show** *?rhs*
by (*auto simp add*: *neq-iff*)
next
assume *?rhs*
with *False* **show** *?lhs*
by (*auto simp add*: *mult-strict-left-mono mult-strict-left-mono-neg*)
qed
qed *auto*

The "conjunction of implication" lemmas produce two cases when the comparison is a goal, but give four when the comparison is an assumption.

lemma *mult-less-cancel-right*: $a * c < b * c \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-right-disj* [*of a c b*] **by** *auto*

lemma *mult-less-cancel-left*: $c * a < c * b \longleftrightarrow (0 \leq c \longrightarrow a < b) \wedge (c \leq 0 \longrightarrow b < a)$
using *mult-less-cancel-left-disj* [*of c a b*] **by** *auto*

lemma *mult-le-cancel-right*: $a * c \leq b * c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-right-disj*)

lemma *mult-le-cancel-left*: $c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *not-less* [*symmetric*] *mult-less-cancel-left-disj*)

lemma *mult-le-cancel-left-pos*: $0 < c \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq b$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-le-cancel-left-neg*: $c < 0 \Longrightarrow c * a \leq c * b \longleftrightarrow b \leq a$
by (*auto simp*: *mult-le-cancel-left*)

lemma *mult-less-cancel-left-pos*: $0 < c \Longrightarrow c * a < c * b \longleftrightarrow a < b$
by (*auto simp*: *mult-less-cancel-left*)

lemma *mult-less-cancel-left-neg*: $c < 0 \Longrightarrow c * a < c * b \longleftrightarrow b < a$
by (*auto simp*: *mult-less-cancel-left*)

lemma *mult-le-cancel-right-pos*: $0 < c \Longrightarrow a * c \leq b * c \longleftrightarrow a \leq b$
by (*auto simp*: *mult-le-cancel-right*)

lemma *mult-le-cancel-right-neg*: $c < 0 \Longrightarrow a * c \leq b * c \longleftrightarrow b \leq a$
by (*auto simp*: *mult-le-cancel-right*)

lemma *mult-less-cancel-right-pos*: $0 < c \Longrightarrow a * c < b * c \longleftrightarrow a < b$
by (*auto simp*: *mult-less-cancel-right*)

lemma *mult-less-cancel-right-neg*: $c < 0 \Longrightarrow a * c < b * c \longleftrightarrow b < a$
by (*auto simp*: *mult-less-cancel-right*)

end

lemmas *mult-sign-intros* =
mult-nonneg-nonneg mult-nonneg-nonpos
mult-nonpos-nonneg mult-nonpos-nonpos
mult-pos-pos mult-pos-neg
mult-neg-pos mult-neg-neg

class *ordered-comm-ring* = *comm-ring* + *ordered-comm-semiring*
begin

subclass *ordered-ring* ..
subclass *ordered-cancel-comm-semiring* ..

end

class *linordered-nonzero-semiring* = *ordered-comm-semiring* + *monoid-mult* + *linorder*
+ *zero-less-one* +
assumes *add-mono1*: $a < b \Longrightarrow a + 1 < b + 1$
begin

subclass *zero-neq-one* ..

subclass *comm-semiring-1*
by *standard* (*rule mult-1-left*)

subclass *ordered-semiring-1* ..

lemma *not-one-le-zero* [*simp*]: $\neg\ 1 \leq 0$
by (*simp add*: *not-le*)

lemma *not-one-less-zero* [*simp*]: $\neg\ 1 < 0$
by (*simp add*: *not-less*)

lemma *of-bool-less-eq-iff* [*simp*]:
‹*of-bool P* ≤ *of-bool Q* ⟷ (*P* ⟶ *Q*)›
by *auto*

lemma *of-bool-less-iff* [*simp*]:
‹*of-bool P* < *of-bool Q* ⟷ ¬ *P* ∧ *Q*›
by *auto*

lemma *mult-left-le*: *c* ≤ *1* ⟹ *0* ≤ *a* ⟹ *a* ∗ *c* ≤ *a*
using *mult-left-mono*[*of c 1 a*] **by** *simp*

lemma *mult-le-one*: *a* ≤ *1* ⟹ *0* ≤ *b* ⟹ *b* ≤ *1* ⟹ *a* ∗ *b* ≤ *1*
using *mult-mono*[*of a 1 b 1*] **by** *simp*

lemma *zero-less-two*: *0* < *1* + *1*
using *add-pos-pos*[*OF zero-less-one zero-less-one*] **.**

end

class *linordered-semidom* = *semidom* + *linordered-comm-semiring-strict* + *zero-less-one*
+
assumes *le-add-diff-inverse2* [*simp*]: *b* ≤ *a* ⟹ *a* − *b* + *b* = *a*
begin

subclass *linordered-nonzero-semiring*
proof
show *a* + *1* < *b* + *1* **if** *a* < *b* **for** *a b*
proof (*rule ccontr*)
assume ¬ *a* + *1* < *b* + *1*
moreover with *that* **have** *a* + *1* < *b* + *1*
by *simp*
ultimately show *False*
by *contradiction*
qed
qed

lemma *zero-less-eq-of-bool* [*simp*]:
‹*0* ≤ *of-bool P*›
by *simp*

lemma *zero-less-of-bool-iff* [*simp*]:
‹*0* < *of-bool P* ⟷ *P*›
by *simp*

lemma *of-bool-less-eq-one* [*simp*]:
‹*of-bool P* ≤ *1*›
by *simp*

lemma *of-bool-less-one-iff* [*simp*]:
‹*of-bool P* < *1* ⟷ ¬ *P*›
by *simp*

lemma *of-bool-or-iff* [*simp*]:
‹*of-bool* (*P* ∨ *Q*) = *max* (*of-bool P*) (*of-bool Q*)›
by (*simp add*: *max-def*)

Addition is the inverse of subtraction.

lemma *le-add-diff-inverse* [*simp*]: *b* ≤ *a* ⟹ *b* + (*a* − *b*) = *a*
by (*frule le-add-diff-inverse2*) (*simp add*: *add.commute*)

lemma *add-diff-inverse*: ¬ *a* < *b* ⟹ *b* + (*a* − *b*) = *a*
by *simp*

lemma *add-le-imp-le-diff*:
assumes *i* + *k* ≤ *n* **shows** *i* ≤ *n* − *k*
proof −
have *n* − (*i* + *k*) + *i* + *k* = *n*
by (*simp add*: *assms add.assoc*)
with *assms add-implies-diff* **have** *i* + *k* ≤ *n* − *k* + *k*
by *fastforce*
then show *?thesis*
by *simp*
qed

lemma *add-le-add-imp-diff-le*:
assumes *1*: *i* + *k* ≤ *n*
and *2*: *n* ≤ *j* + *k*
shows *i* + *k* ≤ *n* ⟹ *n* ≤ *j* + *k* ⟹ *n* − *k* ≤ *j*
proof −
have *n* − (*i* + *k*) + *i* + *k* = *n*
using *1* **by** (*simp add*: *add.assoc*)
moreover have *n* − *k* = *n* − *k* − *i* + *i*
using *1* **by** (*simp add*: *add-le-imp-le-diff*)
ultimately show *?thesis*
using *2 add-le-imp-le-diff* [*of n*−*k k j* + *k*]
by (*simp add*: *add.commute diff-diff-add*)
qed

lemma *less-1-mult*: *1* < *m* ⟹ *1* < *n* ⟹ *1* < *m* ∗ *n*
using *mult-strict-mono* [*of 1 m 1 n*] **by** (*simp add*: *less-trans* [*OF zero-less-one*])

lemma *less-1-mult′*:
shows *1* < *a* ⟹ *1* ≤ *b* ⟹ *1* < *a* ∗ *b*
by (*cases b=1*) (*auto simp*: *le-less less-1-mult*)

end

class *linordered-idom* = *comm-ring-1* + *linordered-comm-semiring-strict* +
ordered-ab-group-add + *abs-if* + *sgn* +
assumes *sgn-if*: *sgn x* = (*if x* = *0 then 0 else if 0* < *x then 1 else* − *1*)
begin

subclass *linordered-ring-strict* **..**

subclass *linordered-semiring-1-strict*
proof
have *0* ≤ *1* ∗ *1*
by (*fact zero-le-square*)
then show *0* < *1*
by (*simp add*: *le-less*)
qed

subclass *ordered-comm-ring* **..**
subclass *idom* **..**

subclass *linordered-semidom*
by *standard simp*

subclass *idom-abs-sgn*
by *standard*
(*auto simp add*: *sgn-if abs-if zero-less-mult-iff*)

lemma *abs-bool-eq* [*simp*]:
‹|*of-bool P*| = *of-bool P*›
by *simp*

lemma *linorder-neqE-linordered-idom*:
assumes *x* ≠ *y*
obtains *x* < *y* | *y* < *x*
using *assms* **by** (*rule neqE*)

These cancellation simp rules also produce two cases when the comparison is a goal.

lemma *mult-le-cancel-right1*: *c* ≤ *b* ∗ *c* ⟷ (*0* < *c* ⟶ *1* ≤ *b*) ∧ (*c* < *0* ⟶ *b* ≤ *1*)
using *mult-le-cancel-right* [*of 1 c b*] **by** *simp*

lemma *mult-le-cancel-right2*: *a* ∗ *c* ≤ *c* ⟷ (*0* < *c* ⟶ *a* ≤ *1*) ∧ (*c* < *0* ⟶ *1* ≤ *a*)
using *mult-le-cancel-right* [*of a c 1*] **by** *simp*

lemma *mult-le-cancel-left1*: *c* ≤ *c* ∗ *b* ⟷ (*0* < *c* ⟶ *1* ≤ *b*) ∧ (*c* < *0* ⟶ *b* ≤ *1*)
using *mult-le-cancel-left* [*of c 1 b*] **by** *simp*

lemma *mult-le-cancel-left2*: *c* ∗ *a* ≤ *c* ⟷ (*0* < *c* ⟶ *a* ≤ *1*) ∧ (*c* < *0* ⟶ *1* ≤

a)
using *mult-le-cancel-left* [*of c a 1*] **by** *simp*

lemma *mult-less-cancel-right1*: $c < b * c \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
using *mult-less-cancel-right* [*of 1 c b*] **by** *simp*

lemma *mult-less-cancel-right2*: $a * c < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
using *mult-less-cancel-right* [*of a c 1*] **by** *simp*

lemma *mult-less-cancel-left1*: $c < c * b \longleftrightarrow (0 \leq c \longrightarrow 1 < b) \wedge (c \leq 0 \longrightarrow b < 1)$
using *mult-less-cancel-left* [*of c 1 b*] **by** *simp*

lemma *mult-less-cancel-left2*: $c * a < c \longleftrightarrow (0 \leq c \longrightarrow a < 1) \wedge (c \leq 0 \longrightarrow 1 < a)$
using *mult-less-cancel-left* [*of c a 1*] **by** *simp*

lemma *sgn-0-0*: $sgn\ a = 0 \longleftrightarrow a = 0$
by (*fact sgn-eq-0-iff*)

lemma *sgn-1-pos*: $sgn\ a = 1 \longleftrightarrow a > 0$
unfolding *sgn-if* **by** *simp*

lemma *sgn-1-neg*: $sgn\ a = -1 \longleftrightarrow a < 0$
unfolding *sgn-if* **by** *auto*

lemma *sgn-pos* [*simp*]: $0 < a \Longrightarrow sgn\ a = 1$
by (*simp only*: *sgn-1-pos*)

lemma *sgn-neg* [*simp*]: $a < 0 \Longrightarrow sgn\ a = -1$
by (*simp only*: *sgn-1-neg*)

lemma *abs-sgn*: $|k| = k * sgn\ k$
unfolding *sgn-if abs-if* **by** *auto*

lemma *sgn-greater* [*simp*]: $0 < sgn\ a \longleftrightarrow 0 < a$
unfolding *sgn-if* **by** *auto*

lemma *sgn-less* [*simp*]: $sgn\ a < 0 \longleftrightarrow a < 0$
unfolding *sgn-if* **by** *auto*

lemma *abs-sgn-eq-1* [*simp*]:
$a \neq 0 \Longrightarrow |sgn\ a| = 1$
by *simp*

lemma *abs-sgn-eq*: $|sgn\ a| = (if\ a = 0\ then\ 0\ else\ 1)$
by (*simp add*: *sgn-if*)

lemma *sgn-mult-self-eq* [*simp*]:
  *sgn a * sgn a = of-bool (a ≠ 0)*
  by (*cases a > 0*) *simp-all*

lemma *left-sgn-mult-self-eq* [*simp*]:
  ‹*sgn a * (sgn a * b) = of-bool (a ≠ 0) * b*›
  by (*simp flip*: *mult.assoc*)

lemma *abs-mult-self-eq* [*simp*]:
  *|a| * |a| = a * a*
  by (*cases a > 0*) *simp-all*

lemma *same-sgn-sgn-add*:
  sgn (a + b) = sgn a **if** *sgn b = sgn a*
proof (*cases a 0 rule*: *linorder-cases*)
  case *equal*
  with *that* **show** *?thesis*
    by *simp*
next
  case *less*
  with *that* **have** *b < 0*
    by (*simp add*: *sgn-1-neg*)
  with ‹*a < 0*› **have** *a + b < 0*
    by (*rule add-neg-neg*)
  with ‹*a < 0*› **show** *?thesis*
    by *simp*
next
  case *greater*
  with *that* **have** *b > 0*
    by (*simp add*: *sgn-1-pos*)
  with ‹*a > 0*› **have** *a + b > 0*
    by (*rule add-pos-pos*)
  with ‹*a > 0*› **show** *?thesis*
    by *simp*
qed

lemma *same-sgn-abs-add*:
  |a + b| = |a| + |b| **if** *sgn b = sgn a*
proof −
  have *a + b = sgn a * |a| + sgn b * |b|*
    by (*simp add*: *sgn-mult-abs*)
  also have *... = sgn a * (|a| + |b|)*
    using *that* **by** (*simp add*: *algebra-simps*)
  finally show *?thesis*
    by (*auto simp add*: *abs-mult*)
qed

lemma *sgn-not-eq-imp*:

sgn a = − sgn b **if** *sgn b ≠ sgn a* **and** *sgn a ≠ 0* **and** *sgn b ≠ 0*
using *that* **by** (*cases a < 0*) (*auto simp add*: *sgn-0-0 sgn-1-pos sgn-1-neg*)

lemma *abs-dvd-iff* [*simp*]: *|m| dvd k ⟷ m dvd k*
by (*simp add*: *abs-if*)

lemma *dvd-abs-iff* [*simp*]: *m dvd |k| ⟷ m dvd k*
by (*simp add*: *abs-if*)

lemma *dvd-if-abs-eq*: *|l| = |k| ⟹ l dvd k*
by (*subst abs-dvd-iff* [*symmetric*]) *simp*

The following lemmas can be proven in more general structures, but are dangerous as simp rules in absence of *(− ?a = ?a) = (?a = 0)*, *(− ?a < ?a) = (0 < ?a)*, *(− ?a ≤ ?a) = (0 ≤ ?a)*.

lemma *equation-minus-iff-1* [*simp*, *no-atp*]: *1 = − a ⟷ a = − 1*
by (*fact equation-minus-iff*)

lemma *minus-equation-iff-1* [*simp*, *no-atp*]: *− a = 1 ⟷ a = − 1*
by (*subst minus-equation-iff*, *auto*)

lemma *le-minus-iff-1* [*simp*, *no-atp*]: *1 ≤ − b ⟷ b ≤ − 1*
by (*fact le-minus-iff*)

lemma *minus-le-iff-1* [*simp*, *no-atp*]: *− a ≤ 1 ⟷ − 1 ≤ a*
by (*fact minus-le-iff*)

lemma *less-minus-iff-1* [*simp*, *no-atp*]: *1 < − b ⟷ b < − 1*
by (*fact less-minus-iff*)

lemma *minus-less-iff-1* [*simp*, *no-atp*]: *− a < 1 ⟷ − 1 < a*
by (*fact minus-less-iff*)

lemma *add-less-zeroD*:
shows *x+y < 0 ⟹ x<0 ∨ y<0*
by (*auto simp*: *not-less intro*: *le-less-trans* [*of - x+y*])

Is this really better than just rewriting with *abs-if*?

lemma *abs-split* [*no-atp*]: *‹P |a| ⟷ (0 ≤ a ⟶ P a) ∧ (a < 0 ⟶ P (− a))›*
by (*force dest*: *order-less-le-trans simp add*: *abs-if linorder-not-less*)

end

class *discrete-linordered-semidom = linordered-semidom +*
assumes *less-iff-succ-less-eq*: *‹a < b ⟷ a + 1 ≤ b›*
begin

lemma *less-eq-iff-succ-less*:
‹a ≤ b ⟷ a < b + 1›

using *less-iff-succ-less-eq* [*of* a ‹$b + 1$›] **by** *simp*

end

Reasoning about inequalities with division

context *linordered-semidom*
begin

lemma *less-add-one*: $a < a + 1$
proof −
have $a + 0 < a + 1$
by (*blast intro*: *zero-less-one add-strict-left-mono*)
then show *?thesis* **by** *simp*
qed

end

context *linordered-idom*
begin

lemma *mult-right-le-one-le*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow x * y \leq x$
by (*rule mult-left-le*)

lemma *mult-left-le-one-le*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow y \leq 1 \Longrightarrow y * x \leq x$
by (*auto simp add*: *mult-le-cancel-right2*)

lemma *affine-ineq*:
assumes $x \leq 1$ $v \leq u$
shows $v + x * u \leq u + x * v$
proof −
have $(1-x)*(u-v) \geq 0$
using *assms* **by** *auto*
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

end

Absolute Value

context *linordered-idom*
begin

lemma *mult-sgn-abs*: *sgn* $x * |x| = x$
by (*fact sgn-mult-abs*)

lemma *abs-one*: $|1| = 1$
by (*fact abs-1*)

end

class *ordered-ring-abs* = *ordered-ring* + *ordered-ab-group-add-abs* +
  assumes *abs-eq-mult*:
    $(0 \leq a \vee a \leq 0) \wedge (0 \leq b \vee b \leq 0) \Longrightarrow |a * b| = |a| * |b|$

context *linordered-idom*
begin

subclass *ordered-ring-abs*
  by *standard* (*auto simp*: *abs-if not-less mult-less-0-iff*)

lemma *abs-mult-self*: $|a| * |a| = a * a$
  by (*fact abs-mult-self-eq*)

lemma *abs-mult-less*:
  assumes *ac*: $|a| < c$
    and *bd*: $|b| < d$
  shows $|a| * |b| < c * d$
proof −
  from *ac* **have** $0 < c$
    by (*blast intro*: *le-less-trans abs-ge-zero*)
  with *bd* **show** *?thesis* **by** (*simp add*: *ac mult-strict-mono*)
qed

lemma *abs-less-iff*: $|a| < b \longleftrightarrow a < b \wedge - a < b$
  by (*simp add*: *less-le abs-le-iff*) (*auto simp add*: *abs-if*)

lemma *abs-mult-pos*: $0 \leq x \Longrightarrow |y| * x = |y * x|$
  by (*simp add*: *abs-mult*)

lemma *abs-mult-pos'*: $0 \leq x \Longrightarrow x * |y| = |x * y|$
  by (*simp add*: *abs-mult*)

lemma *abs-diff-less-iff*: $|x - a| < r \longleftrightarrow a - r < x \wedge x < a + r$
  by (*auto simp add*: *diff-less-eq ac-simps abs-less-iff*)

lemma *abs-diff-le-iff*: $|x - a| \leq r \longleftrightarrow a - r \leq x \wedge x \leq a + r$
  by (*auto simp add*: *diff-le-eq ac-simps abs-le-iff*)

lemma *abs-add-one-gt-zero*: $0 < 1 + |x|$
  by (*auto simp*: *abs-if not-less intro*: *zero-less-one add-strict-increasing less-trans*)

end

16.8 Dioids

Dioids are the alternative extensions of semirings, a semiring can either be a ring or a dioid but never both.

class *dioid* = *semiring-1* + *canonically-ordered-monoid-add*

begin

subclass *ordered-semiring*
  by *standard* (*auto simp*: *le-iff-add distrib-left distrib-right*)

end

hide-fact (**open**) *comm-mult-left-mono comm-mult-strict-left-mono distrib*

code-identifier
  code-module *Rings* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

17 Natural numbers

theory *Nat*
imports *Inductive Typedef Fun Rings*
begin

17.1 Type *ind*

typedecl *ind*

axiomatization *Zero-Rep* :: *ind* **and** *Suc-Rep* :: *ind* ⇒ *ind*
  — The axiom of infinity in 2 parts:
  where *Suc-Rep-inject*: *Suc-Rep x* = *Suc-Rep y* ⟹ *x* = *y*
    and *Suc-Rep-not-Zero-Rep*: *Suc-Rep x* ≠ *Zero-Rep*

17.2 Type nat

Type definition

inductive *Nat* :: *ind* ⇒ *bool*
  where
    Zero-RepI: *Nat Zero-Rep*
  | *Suc-RepI*: *Nat i* ⟹ *Nat* (*Suc-Rep i*)

typedef *nat* = {*n*. *Nat n*}
  morphisms *Rep-Nat Abs-Nat*
  using *Nat.Zero-RepI* **by** *auto*

lemma *Nat-Rep-Nat*: *Nat* (*Rep-Nat n*)
  using *Rep-Nat* **by** *simp*

lemma *Nat-Abs-Nat-inverse*: *Nat n* ⟹ *Rep-Nat* (*Abs-Nat n*) = *n*
  using *Abs-Nat-inverse* **by** *simp*

lemma *Nat-Abs-Nat-inject*: *Nat n* $\Longrightarrow$ *Nat m* $\Longrightarrow$ *Abs-Nat n = Abs-Nat m* $\longleftrightarrow$ *n = m*
using *Abs-Nat-inject* **by** *simp*

instantiation *nat* :: *zero*
begin

definition *Zero-nat-def*: *0 = Abs-Nat Zero-Rep*

instance ..

end

definition *Suc* :: *nat* $\Rightarrow$ *nat*
where *Suc n = Abs-Nat* (*Suc-Rep* (*Rep-Nat n*))

lemma *Suc-not-Zero*: *Suc m* $\neq$ *0*
by (*simp add*: *Zero-nat-def Suc-def Suc-RepI Zero-RepI*
Nat-Abs-Nat-inject Suc-Rep-not-Zero-Rep Nat-Rep-Nat)

lemma *Zero-not-Suc*: *0* $\neq$ *Suc m*
by (*rule not-sym*) (*rule Suc-not-Zero*)

lemma *Suc-Rep-inject'*: *Suc-Rep x = Suc-Rep y* $\longleftrightarrow$ *x = y*
by (*rule iffI*, *rule Suc-Rep-inject*) *simp-all*

lemma *nat-induct0*:
assumes *P 0* **and** $\bigwedge$*n*. *P n* $\Longrightarrow$ *P* (*Suc n*)
shows *P n*
proof −
have *P* (*Abs-Nat* (*Rep-Nat n*))
using *assms* **unfolding** *Zero-nat-def Suc-def*
by (*iprover intro*: *Nat-Rep-Nat* [*THEN Nat.induct*] *elim*: *Nat-Abs-Nat-inverse* [*THEN subst*])
then show *?thesis*
by (*simp add*: *Rep-Nat-inverse*)
qed

free-constructors *case-nat* **for** *0* :: *nat* | *Suc pred*
where *pred* (*0* :: *nat*) = (*0* :: *nat*)
proof *atomize-elim*
fix *n*
show *n = 0* $\vee$ ($\exists$ *m*. *n = Suc m*)
by (*induction n rule*: *nat-induct0*) *auto*
next
fix *n m*
show (*Suc n = Suc m*) = (*n = m*)
by (*simp add*: *Suc-def Nat-Abs-Nat-inject Nat-Rep-Nat Suc-RepI Suc-Rep-inject' Rep-Nat-inject*)

next
  fix *n*
  show *0 ≠ Suc n*
    by (*simp add*: *Suc-not-Zero*)
qed

— Avoid name clashes by prefixing the output of *old-rep-datatype* with *old*.
setup ‹*Sign.mandatory-path old*›

old-rep-datatype *0* :: *nat Suc*
  by (*erule nat-induct0*) *auto*

setup ‹*Sign.parent-path*›

— But erase the prefix for properties that are not generated by *free-constructors*.
setup ‹*Sign.mandatory-path nat*›

declare *old.nat.inject*[*iff del*]
  and *old.nat.distinct*(*1*)[*simp del*, *induct-simp del*]

lemmas *induct* = *old.nat.induct*
lemmas *inducts* = *old.nat.inducts*
lemmas *rec* = *old.nat.rec*
lemmas *simps* = *nat.inject nat.distinct nat.case nat.rec*

setup ‹*Sign.parent-path*›

abbreviation *rec-nat* :: *'a* ⇒ (*nat* ⇒ *'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a*
  where *rec-nat* ≡ *old.rec-nat*

hide-const (**open**) *Nat.pred* — hide everything related to the selector

lemma *nat-exhaust* [*case-names 0 Suc*, *cases type*: *nat*]:
  (*y* = *0* ⟹ *P*) ⟹ (⋀*nat*. *y* = *Suc nat* ⟹ *P*) ⟹ *P*
  — for backward compatibility – names of variables differ
  by (*rule old.nat.exhaust*)

lemma *nat-induct* [*case-names 0 Suc*, *induct type*: *nat*]:
  fixes *n*
  assumes *P 0* **and** ⋀*n*. *P n* ⟹ *P* (*Suc n*)
  shows *P n*
  — for backward compatibility – names of variables differ
  using *assms* **by** (*rule nat.induct*)

hide-fact
  nat-exhaust
  nat-induct0

ML ‹
val nat-basic-lfp-sugar =
let
val ctr-sugar = the (Ctr-Sugar.ctr-sugar-of-global ***theory type-name*** ‹*nat*›);
val recx = Logic.varify-types-global ***term*** ‹*rec-nat*›;
val C = body-type (fastype-of recx);
in
{*T = HOLogic.natT*, *fp-res-index = 0*, *C = C*, *fun-arg-Tsss = [[], [[HOLogic.natT*, *C]]]*,
ctr-sugar = ctr-sugar, *recx = recx*, *rec-thms = @{thms nat.rec}*}
end;
›

setup ‹
let
*fun basic-lfp-sugars-of - [*** typ ‹*nat*›*] - - ctxt =*
([], [0], [nat-basic-lfp-sugar], [], [], [], TrueI (∗dummy∗), [], false, ctxt)
| basic-lfp-sugars-of bs arg-Ts callers callssss ctxt =
BNF-LFP-Rec-Sugar.default-basic-lfp-sugars-of bs arg-Ts callers callssss ctxt;
in
BNF-LFP-Rec-Sugar.register-lfp-rec-extension
{*nested-simps = []*, *special-endgame-tac = K (K (K (K no-tac)))*, *is-new-datatype = K (K true)*,
basic-lfp-sugars-of = basic-lfp-sugars-of, *rewrite-nested-rec-call = NONE*}
end
›

Injectiveness and distinctness lemmas

lemma *inj-Suc* [*simp*]:
inj-on Suc N
by (*simp add*: *inj-on-def*)

lemma *bij-betw-Suc* [*simp*]:
bij-betw Suc M N ⟷ *Suc ‘ M = N*
by (*simp add*: *bij-betw-def*)

lemma *Suc-neq-Zero*: *Suc m = 0* ⟹ *R*
by (*rule notE*) (*rule Suc-not-Zero*)

lemma *Zero-neq-Suc*: *0 = Suc m* ⟹ *R*
by (*rule Suc-neq-Zero*) (*erule sym*)

lemma *Suc-inject*: *Suc x = Suc y* ⟹ *x = y*
by (*rule inj-Suc* [*THEN injD*])

lemma *n-not-Suc-n*: *n ≠ Suc n*
by (*induct n*) *simp-all*

lemma *Suc-n-not-n*: *Suc n ≠ n*

by (*rule not-sym*) (*rule n-not-Suc-n*)

A special form of induction for reasoning about $m < n$ and $m - n$.

lemma *diff-induct*:
assumes $\bigwedge x.\ P\ x\ 0$
and $\bigwedge y.\ P\ 0\ (Suc\ y)$
and $\bigwedge x\ y.\ P\ x\ y \Longrightarrow P\ (Suc\ x)\ (Suc\ y)$
shows $P\ m\ n$
proof (*induct n arbitrary*: *m*)
case *0*
show *?case* **by** (*rule assms*(*1*))
next
case (*Suc n*)
show *?case*
proof (*induct m*)
case *0*
show *?case* **by** (*rule assms*(*2*))
next
case (*Suc m*)
from ‹*P m n*› **show** *?case* **by** (*rule assms*(*3*))
qed
qed

17.3 Arithmetic operators

instantiation *nat* :: *comm-monoid-diff*
begin

primrec *plus-nat*
where
add-0 [*code*]: $0 + n = (n::nat)$
| *add-Suc*: $Suc\ m + n = Suc\ (m + n)$

lemma *add-0-right* [*simp*]: $m + 0 = m$
for $m :: nat$
by (*induct m*) *simp-all*

lemma *add-Suc-right* [*simp*]: $m + Suc\ n = Suc\ (m + n)$
by (*induct m*) *simp-all*

lemma *add-Suc-shift* [*code*]: $Suc\ m + n = m + Suc\ n$
by *simp*

primrec *minus-nat*
where
diff-0 [*code*]: $m - 0 = (m::nat)$
| *diff-Suc*: $m - Suc\ n = (case\ m - n\ of\ 0 \Rightarrow 0 \mid Suc\ k \Rightarrow k)$

declare *diff-Suc* [*simp del*]

lemma *diff-0-eq-0* [*simp*, *code*]: *0 − n = 0*
  for *n* :: *nat*
  by (*induct n*) (*simp-all add*: *diff-Suc*)

lemma *diff-Suc-Suc* [*simp*, *code*]: *Suc m − Suc n = m − n*
  by (*induct n*) (*simp-all add*: *diff-Suc*)

instance
proof
  fix *n m q* :: *nat*
  show *(n + m) + q = n + (m + q)* **by** (*induct n*) *simp-all*
  show *n + m = m + n* **by** (*induct n*) *simp-all*
  show *m + n − m = n* **by** (*induct m*) *simp-all*
  show *n − m − q = n − (m + q)* **by** (*induct q*) (*simp-all add*: *diff-Suc*)
  show *0 + n = n* **by** *simp*
  show *0 − n = 0* **by** *simp*
qed

end

hide-fact (**open**) *add-0 add-0-right diff-0*

instantiation *nat* :: *comm-semiring-1-cancel*
begin

definition *One-nat-def* [*simp*]: *1 = Suc 0*

primrec *times-nat*
  where
    mult-0: *0 * n = (0::nat)*
  | *mult-Suc*: *Suc m * n = n + (m * n)*

lemma *mult-0-right* [*simp*]: *m * 0 = 0*
  for *m* :: *nat*
  by (*induct m*) *simp-all*

lemma *mult-Suc-right* [*simp*]: *m * Suc n = m + (m * n)*
  by (*induct m*) (*simp-all add*: *add.left-commute*)

lemma *add-mult-distrib*: *(m + n) * k = (m * k) + (n * k)*
  for *m n k* :: *nat*
  by (*induct m*) (*simp-all add*: *add.assoc*)

instance
proof
  fix *k n m q* :: *nat*
  show *0 ≠ (1::nat)*
    by *simp*

show $1 * n = n$
by *simp*
show $n * m = m * n$
by (*induct n*) *simp-all*
show $(n * m) * q = n * (m * q)$
by (*induct n*) (*simp-all add*: *add-mult-distrib*)
show $(n + m) * q = n * q + m * q$
by (*rule add-mult-distrib*)
show $k * (m - n) = (k * m) - (k * n)$
by (*induct m n rule*: *diff-induct*) *simp-all*
qed

end

17.3.1 Addition

Reasoning about $m + 0 = 0$, etc.

lemma *add-is-0* [*iff*]: $m + n = 0 \longleftrightarrow m = 0 \wedge n = 0$
for $m\ n$ :: *nat*
by (*cases m*) *simp-all*

lemma *add-is-1*: $m + n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \wedge n = 0 \vee m = 0 \wedge n = Suc\ 0$
by (*cases m*) *simp-all*

lemma *one-is-add*: $Suc\ 0 = m + n \longleftrightarrow m = Suc\ 0 \wedge n = 0 \vee m = 0 \wedge n = Suc\ 0$
by (*rule trans*, *rule eq-commute*, *rule add-is-1*)

lemma *add-eq-self-zero*: $m + n = m \Longrightarrow n = 0$
for $m\ n$ :: *nat*
by (*induct m*) *simp-all*

lemma *plus-1-eq-Suc*:
$plus\ 1 = Suc$
by (*simp add*: *fun-eq-iff*)

lemma *Suc-eq-plus1*: $Suc\ n = n + 1$
by *simp*

lemma *Suc-eq-plus1-left*: $Suc\ n = 1 + n$
by *simp*

17.3.2 Difference

lemma *Suc-diff-diff* [*simp*]: $(Suc\ m - n) - Suc\ k = m - n - k$
by (*simp add*: *diff-diff-add*)

lemma *diff-Suc-1*: $Suc\ n - 1 = n$
by *simp*

lemma *diff-Suc-1'* [*simp*]: *Suc n − Suc 0 = n*
by *simp*

17.3.3 Multiplication

lemma *mult-is-0* [*simp*]: *m * n = 0 ⟷ m = 0 ∨ n = 0* **for** *m n :: nat*
by (*induct m*) *auto*

lemma *mult-eq-1-iff* [*simp*]: *m * n = Suc 0 ⟷ m = Suc 0 ∧ n = Suc 0*
proof (*induct m*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
then show *?case* **by** (*induct n*) *auto*
qed

lemma *one-eq-mult-iff* [*simp*]: *Suc 0 = m * n ⟷ m = Suc 0 ∧ n = Suc 0*
by (*simp add*: *eq-commute flip*: *mult-eq-1-iff*)

lemma *nat-mult-eq-1-iff* [*simp*]: *m * n = 1 ⟷ m = 1 ∧ n = 1*
and *nat-1-eq-mult-iff* [*simp*]: *1 = m * n ⟷ m = 1 ∧ n = 1* **for** *m n :: nat*
by *auto*

lemma *mult-cancel1* [*simp*]: *k * m = k * n ⟷ m = n ∨ k = 0*
for *k m n :: nat*
proof −
have *k ≠ 0 ⟹ k * m = k * n ⟹ m = n*
proof (*induct n arbitrary*: *m*)
case *0*
then show *m = 0* **by** *simp*
next
case (*Suc n*)
then show *m = Suc n*
by (*cases m*) (*simp-all add*: *eq-commute* [*of 0*])
qed
then show *?thesis* **by** *auto*
qed

lemma *mult-cancel2* [*simp*]: *m * k = n * k ⟷ m = n ∨ k = 0*
for *k m n :: nat*
by (*simp add*: *mult.commute*)

lemma *Suc-mult-cancel1*: *Suc k * m = Suc k * n ⟷ m = n*
by (*subst mult-cancel1*) *simp*

17.4 Orders on *nat*

17.4.1 Operation definition

instantiation *nat* :: *linorder*
begin

primrec *less-eq-nat*
  where
    (*0*::*nat*) ≤ *n* ⟷ *True*
  | *Suc m* ≤ *n* ⟷ (*case n of 0* ⇒ *False* | *Suc n* ⇒ *m* ≤ *n*)

declare *less-eq-nat.simps* [*simp del*]

lemma *le0* [*iff*]: *0* ≤ *n* **for**
  n :: *nat*
  by (*simp add*: *less-eq-nat.simps*)

lemma [*code*]: *0* ≤ *n* ⟷ *True*
  for *n* :: *nat*
  by *simp*

definition *less-nat*
  where *less-eq-Suc-le*: *n* < *m* ⟷ *Suc n* ≤ *m*

lemma *Suc-le-mono* [*iff*]: *Suc n* ≤ *Suc m* ⟷ *n* ≤ *m*
  by (*simp add*: *less-eq-nat.simps*(*2*))

lemma *Suc-le-eq* [*code*]: *Suc m* ≤ *n* ⟷ *m* < *n*
  unfolding *less-eq-Suc-le* **..**

lemma *le-0-eq* [*iff*]: *n* ≤ *0* ⟷ *n* = *0*
  for *n* :: *nat*
  by (*induct n*) (*simp-all add*: *less-eq-nat.simps*(*2*))

lemma *not-less0* [*iff*]: ¬ *n* < *0*
  for *n* :: *nat*
  by (*simp add*: *less-eq-Suc-le*)

lemma *less-nat-zero-code* [*code*]: *n* < *0* ⟷ *False*
  for *n* :: *nat*
  by *simp*

lemma *Suc-less-eq* [*iff*]: *Suc m* < *Suc n* ⟷ *m* < *n*
  by (*simp add*: *less-eq-Suc-le*)

lemma *less-Suc-eq-le* [*code*]: *m* < *Suc n* ⟷ *m* ≤ *n*
  by (*simp add*: *less-eq-Suc-le*)

lemma *Suc-less-eq2*: *Suc n* < *m* ⟷ (∃ *m'*. *m* = *Suc m'* ∧ *n* < *m'*)

```
  by (cases m) auto

lemma le-SucI: m ≤ n ⟹ m ≤ Suc n
  by (induct m arbitrary: n) (simp-all add: less-eq-nat.simps(2) split: nat.splits)

lemma Suc-leD: Suc m ≤ n ⟹ m ≤ n
  by (cases n) (auto intro: le-SucI)

lemma less-SucI: m < n ⟹ m < Suc n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

lemma Suc-lessD: Suc m < n ⟹ m < n
  by (simp add: less-eq-Suc-le) (erule Suc-leD)

instance
proof
  fix n m q :: nat
  show n < m ⟷ n ≤ m ∧ ¬ m ≤ n
  proof (induct n arbitrary: m)
    case 0
    then show ?case
      by (cases m) (simp-all add: less-eq-Suc-le)
  next
    case (Suc n)
    then show ?case
      by (cases m) (simp-all add: less-eq-Suc-le)
  qed
  show n ≤ n
    by (induct n) simp-all
  then show n = m if n ≤ m and m ≤ n
    using that by (induct n arbitrary: m)
      (simp-all add: less-eq-nat.simps(2) split: nat.splits)
  show n ≤ q if n ≤ m and m ≤ q
    using that
  proof (induct n arbitrary: m q)
    case 0
    show ?case by simp
  next
    case (Suc n)
    then show ?case
      by (simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits, clarify,
        simp-all (no-asm-use) add: less-eq-nat.simps(2) split: nat.splits)
  qed
  show n ≤ m ∨ m ≤ n
    by (induct n arbitrary: m)
      (simp-all add: less-eq-nat.simps(2) split: nat.splits)
qed
```

end

instantiation *nat* :: *order-bot*
begin

definition *bot-nat* :: *nat*
  where *bot-nat* = *0*

instance
  by *standard* (*simp add*: *bot-nat-def*)

end

instance *nat* :: *no-top*
  by *standard* (*auto intro*: *less-Suc-eq-le* [*THEN iffD2*])

17.4.2 Introduction properties

lemma *lessI* [*iff*]: $n < Suc\ n$
  by (*simp add*: *less-Suc-eq-le*)

lemma *zero-less-Suc* [*iff*]: $0 < Suc\ n$
  by (*simp add*: *less-Suc-eq-le*)

17.4.3 Elimination properties

lemma *less-not-refl*: $\neg\ n < n$
  for *n* :: *nat*
  by (*rule order-less-irrefl*)

lemma *less-not-refl2*: $n < m \Longrightarrow m \neq n$
  for *m n* :: *nat*
  by (*rule not-sym*) (*rule less-imp-neq*)

lemma *less-not-refl3*: $s < t \Longrightarrow s \neq t$
  for *s t* :: *nat*
  by (*rule less-imp-neq*)

lemma *less-irrefl-nat*: $n < n \Longrightarrow R$
  for *n* :: *nat*
  by (*rule notE*, *rule less-not-refl*)

lemma *less-zeroE*: $n < 0 \Longrightarrow R$
  for *n* :: *nat*
  by (*rule notE*) (*rule not-less0*)

lemma *less-Suc-eq*: $m < Suc\ n \longleftrightarrow m < n \vee m = n$
  unfolding *less-Suc-eq-le le-less* **..**

lemma *less-Suc0* [*iff*]: $(n < Suc\ 0) = (n = 0)$

by (*simp add*: *less-Suc-eq*)

lemma *less-one* [*iff*]: $n < 1 \longleftrightarrow n = 0$
for n :: *nat*
unfolding *One-nat-def* **by** (*rule less-Suc0*)

lemma *Suc-mono*: $m < n \Longrightarrow Suc\ m < Suc\ n$
by *simp*

"Less than" is antisymmetric, sort of.

lemma *less-antisym*: $\neg\ n < m \Longrightarrow n < Suc\ m \Longrightarrow m = n$
unfolding *not-less less-Suc-eq-le* **by** (*rule antisym*)

lemma *nat-neq-iff*: $m \neq n \longleftrightarrow m < n \vee n < m$
for $m\ n$ :: *nat*
by (*rule linorder-neq-iff*)

17.4.4 Inductive (?) properties

lemma *Suc-lessI*: $m < n \Longrightarrow Suc\ m \neq n \Longrightarrow Suc\ m < n$
unfolding *less-eq-Suc-le* [*of m*] *le-less* **by** *simp*

lemma *lessE*:
assumes *major*: $i < k$
and *1*: $k = Suc\ i \Longrightarrow P$
and *2*: $\bigwedge j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
shows P
proof −
from *major* **have** $\exists j.\ i \leq j \wedge k = Suc\ j$
unfolding *less-eq-Suc-le* **by** (*induct k*) *simp-all*
then have $(\exists j.\ i < j \wedge k = Suc\ j) \vee k = Suc\ i$
by (*auto simp add*: *less-le*)
with *1 2* **show** P **by** *auto*
qed

lemma *less-SucE*:
assumes *major*: $m < Suc\ n$
and *less*: $m < n \Longrightarrow P$
and *eq*: $m = n \Longrightarrow P$
shows P
proof (*rule major* [*THEN lessE*])
show $Suc\ n = Suc\ m \Longrightarrow P$
using *eq* **by** *blast*
show $\bigwedge j.\ [\![m < j;\ Suc\ n = Suc\ j]\!] \Longrightarrow P$
by (*blast intro*: *less*)
qed

lemma *Suc-lessE*:
assumes *major*: $Suc\ i < k$

and *minor*: $\bigwedge j.\ i < j \Longrightarrow k = Suc\ j \Longrightarrow P$
shows P
proof (*rule major* [*THEN lessE*])
show $k = Suc\ (Suc\ i) \Longrightarrow P$
using *lessI minor* **by** *iprover*
show $\bigwedge j.\ [\![Suc\ i < j;\ k = Suc\ j]\!] \Longrightarrow P$
using *Suc-lessD minor* **by** *iprover*
qed

lemma *Suc-less-SucD*: $Suc\ m < Suc\ n \Longrightarrow m < n$
by *simp*

lemma *less-trans-Suc*:
assumes *le*: $i < j$
shows $j < k \Longrightarrow Suc\ i < k$
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
with *le* **show** *?case*
by *simp* (*auto simp add*: *less-Suc-eq dest*: *Suc-lessD*)
qed

Can be used with *less-Suc-eq* to get $n = m \vee n < m$.

lemma *not-less-eq*: $\neg\ m < n \longleftrightarrow n < Suc\ m$
by (*simp only*: *not-less less-Suc-eq-le*)

lemma *not-less-eq-eq*: $\neg\ m \leq n \longleftrightarrow Suc\ n \leq m$
by (*simp only*: *not-le Suc-le-eq*)

Properties of "less than or equal".

lemma *le-imp-less-Suc*: $m \leq n \Longrightarrow m < Suc\ n$
by (*simp only*: *less-Suc-eq-le*)

lemma *Suc-n-not-le-n*: $\neg\ Suc\ n \leq n$
by (*simp add*: *not-le less-Suc-eq-le*)

lemma *le-Suc-eq*: $m \leq Suc\ n \longleftrightarrow m \leq n \vee m = Suc\ n$
by (*simp add*: *less-Suc-eq-le* [*symmetric*] *less-Suc-eq*)

lemma *le-SucE*: $m \leq Suc\ n \Longrightarrow (m \leq n \Longrightarrow R) \Longrightarrow (m = Suc\ n \Longrightarrow R) \Longrightarrow R$
by (*drule le-Suc-eq* [*THEN iffD1*], *iprover*+)

lemma *Suc-leI*: $m < n \Longrightarrow Suc\ m \leq n$
by (*simp only*: *Suc-le-eq*)

Stronger version of *Suc-leD*.

lemma *Suc-le-lessD*: $Suc\ m \leq n \Longrightarrow m < n$

by (*simp only*: *Suc-le-eq*)

lemma *less-imp-le-nat*: $m < n \Longrightarrow m \leq n$ **for** $m\ n$ :: *nat*
unfolding *less-eq-Suc-le* **by** (*rule Suc-leD*)

For instance, $(Suc\ m < Suc\ n) = (Suc\ m \leq n) = (m < n)$

lemmas *le-simps* = *less-imp-le-nat less-Suc-eq-le Suc-le-eq*

Equivalence of $m \leq n$ and $m < n \vee m = n$

lemma *less-or-eq-imp-le*: $m < n \vee m = n \Longrightarrow m \leq n$
for $m\ n$ :: *nat*
unfolding *le-less* **.**

lemma *le-eq-less-or-eq*: $m \leq n \longleftrightarrow m < n \vee m = n$
for $m\ n$ :: *nat*
by (*rule le-less*)

Useful with *blast*.

lemma *eq-imp-le*: $m = n \Longrightarrow m \leq n$
for $m\ n$ :: *nat*
by *auto*

lemma *le-refl*: $n \leq n$
for n :: *nat*
by *simp*

lemma *le-trans*: $i \leq j \Longrightarrow j \leq k \Longrightarrow i \leq k$
for $i\ j\ k$ :: *nat*
by (*rule order-trans*)

lemma *le-antisym*: $m \leq n \Longrightarrow n \leq m \Longrightarrow m = n$
for $m\ n$ :: *nat*
by (*rule antisym*)

lemma *nat-less-le*: $m < n \longleftrightarrow m \leq n \wedge m \neq n$
for $m\ n$ :: *nat*
by (*rule less-le*)

lemma *le-neq-implies-less*: $m \leq n \Longrightarrow m \neq n \Longrightarrow m < n$
for $m\ n$ :: *nat*
unfolding *less-le* **..**

lemma *nat-le-linear*: $m \leq n \vee n \leq m$
for $m\ n$ :: *nat*
by (*rule linear*)

lemmas *linorder-neqE-nat* = *linorder-neqE* [**where** $'a = nat$]

lemma *le-less-Suc-eq*: $m \leq n \Longrightarrow n < Suc\ m \longleftrightarrow n = m$

unfolding *less-Suc-eq-le* **by** *auto*

lemma *not-less-less-Suc-eq*: $\neg\ n < m \Longrightarrow n < Suc\ m \longleftrightarrow n = m$
unfolding *not-less* **by** (*rule le-less-Suc-eq*)

lemmas *not-less-simps* = *not-less-less-Suc-eq le-less-Suc-eq*

lemma *not0-implies-Suc*: $n \neq 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr0-implies-Suc*: $n > 0 \Longrightarrow \exists\, m.\ n = Suc\ m$
by (*cases n*) *simp-all*

lemma *gr-implies-not0*: $m < n \Longrightarrow n \neq 0$
for *m n* :: *nat*
by (*cases n*) *simp-all*

lemma *neq0-conv*[*iff*]: $n \neq 0 \longleftrightarrow 0 < n$
for *n* :: *nat*
by (*cases n*) *simp-all*

This theorem is useful with *blast*

lemma *gr0I*: $(n = 0 \Longrightarrow False) \Longrightarrow 0 < n$
for *n* :: *nat*
by (*rule neq0-conv*[*THEN iffD1*]) *iprover*

lemma *gr0-conv-Suc*: $0 < n \longleftrightarrow (\exists\, m.\ n = Suc\ m)$
by (*fast intro*: *not0-implies-Suc*)

lemma *not-gr0* [*iff*]: $\neg\ 0 < n \longleftrightarrow n = 0$
for *n* :: *nat*
using *neq0-conv* **by** *blast*

lemma *Suc-le-D*: $Suc\ n \leq m' \Longrightarrow \exists\, m.\ m' = Suc\ m$
by (*induct m'*) *simp-all*

Useful in certain inductive arguments

lemma *less-Suc-eq-0-disj*: $m < Suc\ n \longleftrightarrow m = 0 \vee (\exists\, j.\ m = Suc\ j \wedge j < n)$
by (*cases m*) *simp-all*

lemma *All-less-Suc*: $(\forall\, i < Suc\ n.\ P\ i) = (P\ n \wedge (\forall\, i < n.\ P\ i))$
by (*auto simp*: *less-Suc-eq*)

lemma *All-less-Suc2*: $(\forall\, i < Suc\ n.\ P\ i) = (P\ 0 \wedge (\forall\, i < n.\ P(Suc\ i)))$
by (*auto simp*: *less-Suc-eq-0-disj*)

lemma *Ex-less-Suc*: $(\exists\, i < Suc\ n.\ P\ i) = (P\ n \vee (\exists\, i < n.\ P\ i))$
by (*auto simp*: *less-Suc-eq*)

lemma *Ex-less-Suc2*: $(\exists\, i < \mathit{Suc}\ n.\ P\ i) = (P\ 0 \vee (\exists\, i < n.\ P(\mathit{Suc}\ i)))$
by (*auto simp*: *less-Suc-eq-0-disj*)

mono (non-strict) doesn't imply increasing, as the function could be constant

lemma *strict-mono-imp-increasing*:
fixes *n*::*nat*
assumes *strict-mono f* **shows** $f\ n \geq n$
proof (*induction n*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
then show *?case*
unfolding *not-less-eq-eq* [*symmetric*]
using *Suc-n-not-le-n assms order-trans strict-mono-less-eq* **by** *blast*
qed

17.4.5 Monotonicity of Addition

lemma *Suc-pred* [*simp*]: $n > 0 \Longrightarrow \mathit{Suc}\ (n - \mathit{Suc}\ 0) = n$
by (*simp add*: *diff-Suc split*: *nat.split*)

lemma *Suc-diff-1* [*simp*]: $0 < n \Longrightarrow \mathit{Suc}\ (n - 1) = n$
unfolding *One-nat-def* **by** (*rule Suc-pred*)

lemma *nat-add-left-cancel-le* [*simp*]: $k + m \leq k + n \longleftrightarrow m \leq n$
for *k m n* :: *nat*
by (*induct k*) *simp-all*

lemma *nat-add-left-cancel-less* [*simp*]: $k + m < k + n \longleftrightarrow m < n$
for *k m n* :: *nat*
by (*induct k*) *simp-all*

lemma *add-gr-0* [*iff*]: $m + n > 0 \longleftrightarrow m > 0 \vee n > 0$
for *m n* :: *nat*
by (*auto dest*: *gr0-implies-Suc*)

strict, in 1st argument

lemma *add-less-mono1*: $i < j \Longrightarrow i + k < j + k$
for *i j k* :: *nat*
by (*induct k*) *simp-all*

strict, in both arguments

lemma *add-less-mono*:
fixes *i j k l* :: *nat*
assumes $i < j$ $k < l$ **shows** $i + k < j + l$
proof −
have $i + k < j + k$

by (*simp add*: *add-less-mono1 assms*)
also have ... $< j + l$
using ‹$i < j$› **by** (*induction j*) (*auto simp*: *assms*)
finally show *?thesis* **.**
qed

lemma *less-imp-Suc-add*: $m < n \Longrightarrow \exists\, k.\ n = Suc\ (m + k)$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case*
by (*simp add*: *order-le-less*)
(*blast elim*!: *less-SucE intro*!: *Nat.add-0-right* [*symmetric*] *add-Suc-right* [*symmetric*])
qed

lemma *le-Suc-ex*: $k \leq l \Longrightarrow (\exists\, n.\ l = k + n)$
for $k\ l$ :: *nat*
by (*auto simp*: *less-Suc-eq-le*[*symmetric*] *dest*: *less-imp-Suc-add*)

lemma *less-natE*:
assumes ‹$m < n$›
obtains q **where** ‹$n = Suc\ (m + q)$›
using *assms* **by** (*auto dest*: *less-imp-Suc-add intro*: *that*)

strict, in 1st argument; proof is by induction on $k > 0$

lemma *mult-less-mono2*:
fixes $i\ j$ :: *nat*
assumes $i < j$ **and** $0 < k$
shows $k * i < k * j$
using ‹$0 < k$›
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
with ‹$i < j$› **show** *?case*
by (*cases k*) (*simp-all add*: *add-less-mono*)
qed

Addition is the inverse of subtraction: if $n \leq m$ then $n + (m - n) = m$.

lemma *add-diff-inverse-nat*: $\neg\ m < n \Longrightarrow n + (m - n) = m$
for $m\ n$ :: *nat*
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *nat-le-iff-add*: $m \leq n \longleftrightarrow (\exists\, k.\ n = m + k)$
for $m\ n$ :: *nat*
using *nat-add-left-cancel-le*[*of m 0*] **by** (*auto dest*: *le-Suc-ex*)

The naturals form an ordered *semidom* and a *dioid*.

instance *nat* :: *discrete-linordered-semidom*
proof
  fix *m n q* :: *nat*
  show ‹*0* < (*1*::*nat*)›
    by *simp*
  show ‹*m* ≤ *n* ⟹ *q* + *m* ≤ *q* + *n*›
    by *simp*
  show ‹*m* < *n* ⟹ *0* < *q* ⟹ *q* * *m* < *q* * *n*›
    by (*simp add*: *mult-less-mono2*)
  show ‹*m* ≠ *0* ⟹ *n* ≠ *0* ⟹ *m* * *n* ≠ *0*›
    by *simp*
  show ‹*n* ≤ *m* ⟹ (*m* − *n*) + *n* = *m*›
    by (*simp add*: *add-diff-inverse-nat add.commute linorder-not-less*)
  show ‹*m* < *n* ⟷ *m* + *1* ≤ *n*›
    by (*simp add*: *Suc-le-eq*)
qed

instance *nat* :: *dioid*
  by *standard* (*rule nat-le-iff-add*)

declare *le0*[*simp del*] — This is now *0* ≤ *?x*
declare *le-0-eq*[*simp del*] — This is now (*?n* ≤ *0*) = (*?n* = *0*)
declare *not-less0*[*simp del*] — This is now ¬ *?n* < *0*
declare *not-gr0*[*simp del*] — This is now (¬ *0* < *?n*) = (*?n* = *0*)

instance *nat* :: *ordered-cancel-comm-monoid-add* **..**
instance *nat* :: *ordered-cancel-comm-monoid-diff* **..**

17.4.6 *min* and *max*

global-interpretation *bot-nat-0*: *ordering-top* ‹(≥)› ‹(>)› ‹*0*::*nat*›
  by *standard simp*

global-interpretation *max-nat*: *semilattice-neutr-order max* ‹*0*::*nat*› ‹(≥)› ‹(>)›
  by *standard* (*simp add*: *max-def*)

lemma *mono-Suc*: *mono Suc*
  by (*rule monoI*) *simp*

lemma *min-0L* [*simp*]: *min 0 n* = *0*
  for *n* :: *nat*
  by (*rule min-absorb1*) *simp*

lemma *min-0R* [*simp*]: *min n 0* = *0*
  for *n* :: *nat*
  by (*rule min-absorb2*) *simp*

lemma *min-Suc-Suc* [*simp*]: *min* (*Suc m*) (*Suc n*) = *Suc* (*min m n*)

by (*simp add*: *mono-Suc min-of-mono*)

lemma *min-Suc1*: *min* (*Suc n*) *m* = (*case m of 0* ⇒ *0* | *Suc m′* ⇒ *Suc*(*min n m′*))
by (*simp split*: *nat.split*)

lemma *min-Suc2*: *min m* (*Suc n*) = (*case m of 0* ⇒ *0* | *Suc m′* ⇒ *Suc*(*min m′ n*))
by (*simp split*: *nat.split*)

lemma *max-0L* [*simp*]: *max 0 n* = *n*
for *n* :: *nat*
by (*fact max-nat.left-neutral*)

lemma *max-0R* [*simp*]: *max n 0* = *n*
for *n* :: *nat*
by (*fact max-nat.right-neutral*)

lemma *max-Suc-Suc* [*simp*]: *max* (*Suc m*) (*Suc n*) = *Suc* (*max m n*)
by (*simp add*: *mono-Suc max-of-mono*)

lemma *max-Suc1*: *max* (*Suc n*) *m* = (*case m of 0* ⇒ *Suc n* | *Suc m′* ⇒ *Suc* (*max n m′*))
by (*simp split*: *nat.split*)

lemma *max-Suc2*: *max m* (*Suc n*) = (*case m of 0* ⇒ *Suc n* | *Suc m′* ⇒ *Suc* (*max m′ n*))
by (*simp split*: *nat.split*)

lemma *nat-mult-min-left*: *min m n* ∗ *q* = *min* (*m* ∗ *q*) (*n* ∗ *q*)
for *m n q* :: *nat*
by (*simp add*: *min-def not-le*)
(*auto dest*: *mult-right-le-imp-le mult-right-less-imp-less le-less-trans*)

lemma *nat-mult-min-right*: *m* ∗ *min n q* = *min* (*m* ∗ *n*) (*m* ∗ *q*)
for *m n q* :: *nat*
by (*simp add*: *min-def not-le*)
(*auto dest*: *mult-left-le-imp-le mult-left-less-imp-less le-less-trans*)

lemma *nat-add-max-left*: *max m n* + *q* = *max* (*m* + *q*) (*n* + *q*)
for *m n q* :: *nat*
by (*simp add*: *max-def*)

lemma *nat-add-max-right*: *m* + *max n q* = *max* (*m* + *n*) (*m* + *q*)
for *m n q* :: *nat*
by (*simp add*: *max-def*)

lemma *nat-mult-max-left*: *max m n* ∗ *q* = *max* (*m* ∗ *q*) (*n* ∗ *q*)
for *m n q* :: *nat*

by (*simp add*: *max-def not-le*)
(*auto dest*: *mult-right-le-imp-le mult-right-less-imp-less le-less-trans*)

lemma *nat-mult-max-right*: $m * max\ n\ q = max\ (m * n)\ (m * q)$
for $m\ n\ q :: nat$
by (*simp add*: *max-def not-le*)
(*auto dest*: *mult-left-le-imp-le mult-left-less-imp-less le-less-trans*)

17.4.7 Additional theorems about $(\leq)$

Complete induction, aka course-of-values induction

instance *nat* :: *wellorder*
proof
fix P **and** $n :: nat$
assume *step*: $(\bigwedge m.\ m < n \Longrightarrow P\ m) \Longrightarrow P\ n$ **for** $n :: nat$
have $\bigwedge q.\ q \leq n \Longrightarrow P\ q$
proof (*induct n*)
case (*0 n*)
have *P 0* **by** (*rule step*) *auto*
with *0* **show** *?case* **by** *auto*
next
case (*Suc m n*)
then have $n \leq m \vee n = Suc\ m$
by (*simp add*: *le-Suc-eq*)
then show *?case*
proof
assume $n \leq m$
then show $P\ n$ **by** (*rule Suc*(*1*))
next
assume *n*: $n = Suc\ m$
show $P\ n$ **by** (*rule step*) (*rule Suc*(*1*), *simp add*: *n le-simps*)
qed
qed
then show $P\ n$ **by** *auto*
qed

lemma *Least-eq-0*[*simp*]: $P\ 0 \Longrightarrow Least\ P = 0$
for $P :: nat \Rightarrow bool$
by (*rule Least-equality*[*OF - le0*])

lemma *Least-Suc*:
assumes $P\ n \neg P\ 0$
shows $(LEAST\ n.\ P\ n) = Suc\ (LEAST\ m.\ P\ (Suc\ m))$
proof (*cases n*)
case (*Suc m*)
show *?thesis*
proof (*rule antisym*)
show $(LEAST\ x.\ P\ x) \leq Suc\ (LEAST\ x.\ P\ (Suc\ x))$

```
    using assms Suc by (force intro: LeastI Least-le)
  have §: P (LEAST x. P x)
    by (blast intro: LeastI assms)
  show Suc (LEAST m. P (Suc m)) ≤ (LEAST n. P n)
  proof (cases (LEAST n. P n))
    case 0
    then show ?thesis
      using § by (simp add: assms)
  next
    case Suc
    with § show ?thesis
      by (auto simp: Least-le)
  qed
 qed
qed (use assms in auto)

lemma Least-Suc2: P n ⟹ Q m ⟹ ¬ P 0 ⟹ ∀ k. P (Suc k) = Q k ⟹ Least
P = Suc (Least Q)
 by (erule (1) Least-Suc [THEN ssubst]) simp

lemma ex-least-nat-le:
 fixes P :: nat ⇒ bool
 assumes P n
 shows ∃ k≤n. (∀ i<k. ¬ P i) ∧ P k
proof (cases n)
 case (Suc m)
 with assms show ?thesis
   by (blast intro: Least-le LeastI-ex dest: not-less-Least)
qed (use assms in auto)

lemma ex-least-nat-less:
 fixes P :: nat ⇒ bool
 assumes P n ¬ P 0
 shows ∃ k<n. (∀ i≤k. ¬ P i) ∧ P (Suc k)
proof (cases n)
 case (Suc m)
 then obtain k where k: k ≤ n ∀ i<k. ¬ P i P k
   using ex-least-nat-le ‹P n› by blast
 show ?thesis
   by (cases k) (use assms k less-eq-Suc-le in auto)
qed (use assms in auto)


lemma nat-less-induct:
 fixes P :: nat ⇒ bool
 assumes ⋀n. ∀ m. m < n ⟶ P m ⟹ P n
 shows P n
 using assms less-induct by blast
```

lemma *measure-induct-rule* [*case-names less*]:
fixes $f :: 'a \Rightarrow 'b$::*wellorder*
assumes *step*: $\bigwedge x.\ (\bigwedge y.\ f\ y < f\ x \Longrightarrow P\ y) \Longrightarrow P\ x$
shows $P\ a$
by (*induct* $m \equiv f\ a$ *arbitrary*: a *rule*: *less-induct*) (*auto intro*: *step*)

old style induction rules:

lemma *measure-induct*:
fixes $f :: 'a \Rightarrow 'b$::*wellorder*
shows $(\bigwedge x.\ \forall y.\ f\ y < f\ x \longrightarrow P\ y \Longrightarrow P\ x) \Longrightarrow P\ a$
by (*rule measure-induct-rule* [*of f P a*]) *iprover*

lemma *full-nat-induct*:
assumes *step*: $\bigwedge n.\ (\forall m.\ Suc\ m \leq n \longrightarrow P\ m) \Longrightarrow P\ n$
shows $P\ n$
by (*rule less-induct*) (*auto intro*: *step simp*:*le-simps*)

An induction rule for establishing binary relations

lemma *less-Suc-induct* [*consumes 1*]:
assumes *less*: $i < j$
and *step*: $\bigwedge i.\ P\ i\ (Suc\ i)$
and *trans*: $\bigwedge i\ j\ k.\ i < j \Longrightarrow j < k \Longrightarrow P\ i\ j \Longrightarrow P\ j\ k \Longrightarrow P\ i\ k$
shows $P\ i\ j$
proof −
from *less* **obtain** k **where** *j*: $j = Suc\ (i + k)$
by (*auto dest*: *less-imp-Suc-add*)
have $P\ i\ (Suc\ (i + k))$
proof (*induct k*)
case *0*
show *?case* **by** (*simp add*: *step*)
next
case (*Suc k*)
have $0 + i < Suc\ k + i$ **by** (*rule add-less-mono1*) *simp*
then have $i < Suc\ (i + k)$ **by** (*simp add*: *add.commute*)
from *trans*[*OF this lessI Suc step*]
show *?case* **by** *simp*
qed
then show $P\ i\ j$ **by** (*simp add*: *j*)
qed

The method of infinite descent, frequently used in number theory. Provided by Roelof Oosterhuis. $P\ n$ is true for all natural numbers if

- case "0": given $n = 0$ prove $P\ n$
- case "smaller": given $n > 0$ and $\neg\ P\ n$ prove there exists a smaller natural number m such that $\neg\ P\ m$.

lemma *infinite-descent*: $(\bigwedge n.\ \neg\ P\ n \Longrightarrow \exists\, m{<}n.\ \neg\ P\ m) \Longrightarrow P\ n$ **for** $P :: nat \Rightarrow bool$

— compact version without explicit base case
by (*induct n rule*: *less-induct*) *auto*

lemma *infinite-descent0* [*case-names 0 smaller*]:
fixes $P :: nat \Rightarrow bool$
assumes $P\ 0$
and $\bigwedge n.\ n > 0 \Longrightarrow \neg\ P\ n \Longrightarrow \exists\, m.\ m < n \wedge \neg\ P\ m$
shows $P\ n$
proof (*rule infinite-descent*)
fix n
show $\neg\ P\ n \Longrightarrow \exists\, m{<}n.\ \neg\ P\ m$
using *assms* **by** (*cases* $n > 0$) *auto*
qed

Infinite descent using a mapping to *nat*: $P\ x$ is true for all $x \in D$ if there exists a $V \in D \Rightarrow nat$ and

- case "0": given $V\ x = 0$ prove $P\ x$
- "smaller": given $V\ x > 0$ and $\neg\ P\ x$ prove there exists a $y \in D$ such that $V\ y < V\ x$ and $\neg\ P\ y$.

corollary *infinite-descent0-measure* [*case-names 0 smaller*]:
fixes $V :: 'a \Rightarrow nat$
assumes *1*: $\bigwedge x.\ V\ x = 0 \Longrightarrow P\ x$
and *2*: $\bigwedge x.\ V\ x > 0 \Longrightarrow \neg\ P\ x \Longrightarrow \exists\, y.\ V\ y < V\ x \wedge \neg\ P\ y$
shows $P\ x$
proof −
obtain n **where** $n = V\ x$ **by** *auto*
moreover have $\bigwedge x.\ V\ x = n \Longrightarrow P\ x$
proof (*induct n rule*: *infinite-descent0*)
case *0*
with *1* **show** $P\ x$ **by** *auto*
next
case (*smaller n*)
then obtain x **where** $*$: $V\ x = n$ **and** $V\ x > 0 \wedge \neg\ P\ x$ **by** *auto*
with *2* **obtain** y **where** $V\ y < V\ x \wedge \neg\ P\ y$ **by** *auto*
with $*$ **obtain** m **where** $m = V\ y \wedge m < n \wedge \neg\ P\ y$ **by** *auto*
then show *?case* **by** *auto*
qed
ultimately show $P\ x$ **by** *auto*
qed

Again, without explicit base case:

lemma *infinite-descent-measure*:
fixes $V :: 'a \Rightarrow nat$
assumes $\bigwedge x.\ \neg\ P\ x \Longrightarrow \exists\, y.\ V\ y < V\ x \wedge \neg\ P\ y$
shows $P\ x$
proof −

from *assms* **obtain** n **where** $n = V\ x$ **by** *auto*
moreover have $\bigwedge x.\ V\ x = n \Longrightarrow P\ x$
proof −
have $\exists\, m < V\ x.\ \exists\, y.\ V\ y = m \wedge \neg\ P\ y$ **if** $\neg\ P\ x$ **for** x
using *assms* **and** *that* **by** *auto*
then show $\bigwedge x.\ V\ x = n \Longrightarrow P\ x$
by (*induct n rule*: *infinite-descent*, *auto*)
qed
ultimately show $P\ x$ **by** *auto*
qed

A (clumsy) way of lifting $<$ monotonicity to $\leq$ monotonicity

lemma *less-mono-imp-le-mono*:
fixes $f :: nat \Rightarrow nat$
and $i\ j :: nat$
assumes $\bigwedge i\ j{::}nat.\ i < j \Longrightarrow f\ i < f\ j$
and $i \leq j$
shows $f\ i \leq f\ j$
using *assms* **by** (*auto simp add*: *order-le-less*)

non-strict, in 1st argument

lemma *add-le-mono1*: $i \leq j \Longrightarrow i + k \leq j + k$
for $i\ j\ k :: nat$
by (*rule add-right-mono*)

non-strict, in both arguments

lemma *add-le-mono*: $i \leq j \Longrightarrow k \leq l \Longrightarrow i + k \leq j + l$
for $i\ j\ k\ l :: nat$
by (*rule add-mono*)

lemma *le-add2*: $n \leq m + n$
for $m\ n :: nat$
by *simp*

lemma *le-add1*: $n \leq n + m$
for $m\ n :: nat$
by *simp*

lemma *less-add-Suc1*: $i < Suc\ (i + m)$
by (*rule le-less-trans*, *rule le-add1*, *rule lessI*)

lemma *less-add-Suc2*: $i < Suc\ (m + i)$
by (*rule le-less-trans*, *rule le-add2*, *rule lessI*)

lemma *less-iff-Suc-add*: $m < n \longleftrightarrow (\exists\, k.\ n = Suc\ (m + k))$
by (*iprover intro*!: *less-add-Suc1 less-imp-Suc-add*)

lemma *trans-le-add1*: $i \leq j \Longrightarrow i \leq j + m$
for $i\ j\ m :: nat$

by (*rule le-trans*, *assumption*, *rule le-add1*)

lemma *trans-le-add2*: $i \leq j \Longrightarrow i \leq m + j$
for $i\ j\ m$ :: *nat*
by (*rule le-trans*, *assumption*, *rule le-add2*)

lemma *trans-less-add1*: $i < j \Longrightarrow i < j + m$
for $i\ j\ m$ :: *nat*
by (*rule less-le-trans*, *assumption*, *rule le-add1*)

lemma *trans-less-add2*: $i < j \Longrightarrow i < m + j$
for $i\ j\ m$ :: *nat*
by (*rule less-le-trans*, *assumption*, *rule le-add2*)

lemma *add-lessD1*: $i + j < k \Longrightarrow i < k$
for $i\ j\ k$ :: *nat*
by (*rule le-less-trans* [*of* - *i*+*j*]) (*simp-all add*: *le-add1*)

lemma *not-add-less1* [*iff*]: $\neg\ i + j < i$
for $i\ j$ :: *nat*
by *simp*

lemma *not-add-less2* [*iff*]: $\neg\ j + i < i$
for $i\ j$ :: *nat*
by *simp*

lemma *add-leD1*: $m + k \leq n \Longrightarrow m \leq n$
for $k\ m\ n$ :: *nat*
by (*rule order-trans* [*of* - $m + k$]) (*simp-all add*: *le-add1*)

lemma *add-leD2*: $m + k \leq n \Longrightarrow k \leq n$
for $k\ m\ n$ :: *nat*
by (*force simp add*: *add.commute dest*: *add-leD1*)

lemma *add-leE*: $m + k \leq n \Longrightarrow (m \leq n \Longrightarrow k \leq n \Longrightarrow R) \Longrightarrow R$
for $k\ m\ n$ :: *nat*
by (*blast dest*: *add-leD1 add-leD2*)

needs $\bigwedge k$ for *ac-simps* to work

lemma *less-add-eq-less*: $\bigwedge k.\ k < l \Longrightarrow m + l = k + n \Longrightarrow m < n$
for $l\ m\ n$ :: *nat*
by (*force simp del*: *add-Suc-right simp add*: *less-iff-Suc-add add-Suc-right* [*symmetric*] *ac-simps*)

17.4.8 More results about difference

lemma *Suc-diff-le*: $n \leq m \Longrightarrow Suc\ m - n = Suc\ (m - n)$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-less-Suc*: $m - n < Suc\ m$
by (*induct m n rule*: *diff-induct*) (*auto simp*: *less-Suc-eq*)

lemma *diff-le-self* [*simp*]: $m - n \leq m$
for $m\ n :: nat$
by (*induct m n rule*: *diff-induct*) (*simp-all add*: *le-SucI*)

lemma *less-imp-diff-less*: $j < k \Longrightarrow j - n < k$
for $j\ k\ n :: nat$
by (*rule le-less-trans*, *rule diff-le-self*)

lemma *diff-Suc-less* [*simp*]: $0 < n \Longrightarrow n - Suc\ i < n$
by (*cases n*) (*auto simp add*: *le-simps*)

lemma *diff-add-assoc*: $k \leq j \Longrightarrow (i + j) - k = i + (j - k)$
for $i\ j\ k :: nat$
by (*fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc*)

lemma *add-diff-assoc* [*simp*]: $k \leq j \Longrightarrow i + (j - k) = i + j - k$
for $i\ j\ k :: nat$
by (*fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc*)

lemma *diff-add-assoc2*: $k \leq j \Longrightarrow (j + i) - k = (j - k) + i$
for $i\ j\ k :: nat$
by (*fact ordered-cancel-comm-monoid-diff-class.diff-add-assoc2*)

lemma *add-diff-assoc2* [*simp*]: $k \leq j \Longrightarrow j - k + i = j + i - k$
for $i\ j\ k :: nat$
by (*fact ordered-cancel-comm-monoid-diff-class.add-diff-assoc2*)

lemma *le-imp-diff-is-add*: $i \leq j \Longrightarrow (j - i = k) = (j = k + i)$
for $i\ j\ k :: nat$
by *auto*

lemma *diff-is-0-eq* [*simp*]: $m - n = 0 \longleftrightarrow m \leq n$
for $m\ n :: nat$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *diff-is-0-eq'* [*simp*]: $m \leq n \Longrightarrow m - n = 0$
for $m\ n :: nat$
by *simp*

lemma *zero-less-diff* [*simp*]: $0 < n - m \longleftrightarrow m < n$
for $m\ n :: nat$
by (*induct m n rule*: *diff-induct*) *simp-all*

lemma *less-imp-add-positive*:
assumes $i < j$
shows $\exists k{::}nat.\ 0 < k \wedge i + k = j$

proof
from *assms* **show** $0 < j - i \wedge i + (j - i) = j$
by (*simp add*: *order-less-imp-le*)
qed

a nice rewrite for bounded subtraction

lemma *nat-minus-add-max*: $n - m + m = max\ n\ m$
for $m\ n$ :: *nat*
by (*simp add*: *max-def not-le order-less-imp-le*)

lemma *nat-diff-split*: $P\ (a - b) \longleftrightarrow (a < b \longrightarrow P\ 0) \wedge (\forall d.\ a = b + d \longrightarrow P\ d)$
for $a\ b$ :: *nat*
— elimination of $-$ on *nat*
by (*cases* $a < b$) (*auto simp add*: *not-less le-less dest*!: *add-eq-self-zero* [*OF sym*])

lemma *nat-diff-split-asm*: $P\ (a - b) \longleftrightarrow \neg\ (a < b \wedge \neg\ P\ 0 \vee (\exists d.\ a = b + d \wedge \neg\ P\ d))$
for $a\ b$ :: *nat*
— elimination of $-$ on *nat* in assumptions
by (*auto split*: *nat-diff-split*)

lemmas *nat-diff-splits* = *nat-diff-split nat-diff-split-asm*

lemma *Suc-pred'*: $0 < n \Longrightarrow n = Suc(n - 1)$
by *simp*

lemma *add-eq-if*: $m + n = (if\ m = 0\ then\ n\ else\ Suc\ ((m - 1) + n))$
unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma *mult-eq-if*: $m * n = (if\ m = 0\ then\ 0\ else\ n + ((m - 1) * n))$
for $m\ n$ :: *nat*
by (*cases m*) *simp-all*

lemma *Suc-diff-eq-diff-pred*: $0 < n \Longrightarrow Suc\ m - n = m - (n - 1)$
by (*cases n*) *simp-all*

lemma *diff-Suc-eq-diff-pred*: $m - Suc\ n = (m - 1) - n$
by (*cases m*) *simp-all*

lemma *Let-Suc* [*simp*]: $Let\ (Suc\ n)\ f \equiv f\ (Suc\ n)$
by (*fact Let-def*)

17.4.9 Monotonicity of multiplication

lemma *mult-le-mono1*: $i \leq j \Longrightarrow i * k \leq j * k$
for $i\ j\ k$ :: *nat*
by (*simp add*: *mult-right-mono*)

lemma *mult-le-mono2*: $i \leq j \Longrightarrow k * i \leq k * j$

for *i j k* :: *nat*
by (*simp add*: *mult-left-mono*)

$\leq$ monotonicity, BOTH arguments

lemma *mult-le-mono*: $i \leq j \Longrightarrow k \leq l \Longrightarrow i * k \leq j * l$
for *i j k l* :: *nat*
by (*simp add*: *mult-mono*)

lemma *mult-less-mono1*: $i < j \Longrightarrow 0 < k \Longrightarrow i * k < j * k$
for *i j k* :: *nat*
by (*simp add*: *mult-strict-right-mono*)

Differs from the standard *zero-less-mult-iff* in that there are no negative numbers.

lemma *nat-0-less-mult-iff* [*simp*]: $0 < m * n \longleftrightarrow 0 < m \wedge 0 < n$
for *m n* :: *nat*
proof (*induct m*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
then show *?case* **by** (*cases n*) *simp-all*
qed

lemma *one-le-mult-iff* [*simp*]: $Suc\ 0 \leq m * n \longleftrightarrow Suc\ 0 \leq m \wedge Suc\ 0 \leq n$
proof (*induct m*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
then show *?case* **by** (*cases n*) *simp-all*
qed

lemma *mult-less-cancel2* [*simp*]: $m * k < n * k \longleftrightarrow 0 < k \wedge m < n$
for *k m n* :: *nat*
proof (*intro iffI conjI*)
assume *m*: $m * k < n * k$
then show $0 < k$
by (*cases k*) *auto*
show $m < n$
proof (*cases k*)
case *0*
then show *?thesis*
using *m* **by** *auto*
next
case (*Suc k'*)
then show *?thesis*
using *m*
by (*simp flip*: *linorder-not-le*) (*blast intro*: *add-mono mult-le-mono1*)

qed
next
assume $0 < k \wedge m < n$
then show $m * k < n * k$
by (*blast intro*: *mult-less-mono1*)
qed

lemma *mult-less-cancel1* [*simp*]: $k * m < k * n \longleftrightarrow 0 < k \wedge m < n$
for $k\ m\ n$ :: *nat*
by (*simp add*: *mult.commute* [*of k*])

lemma *mult-le-cancel1* [*simp*]: $k * m \leq k * n \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n$ :: *nat*
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *mult-le-cancel2* [*simp*]: $m * k \leq n * k \longleftrightarrow (0 < k \longrightarrow m \leq n)$
for $k\ m\ n$ :: *nat*
by (*simp add*: *linorder-not-less* [*symmetric*], *auto*)

lemma *Suc-mult-less-cancel1*: $Suc\ k * m < Suc\ k * n \longleftrightarrow m < n$
by (*subst mult-less-cancel1*) *simp*

lemma *Suc-mult-le-cancel1*: $Suc\ k * m \leq Suc\ k * n \longleftrightarrow m \leq n$
by (*subst mult-le-cancel1*) *simp*

lemma *le-square*: $m \leq m * m$
for m :: *nat*
by (*cases m*) (*auto intro*: *le-add1*)

lemma *le-cube*: $m \leq m * (m * m)$
for m :: *nat*
by (*cases m*) (*auto intro*: *le-add1*)

Lemma for *gcd*

lemma *mult-eq-self-implies-10*:
fixes $m\ n$ :: *nat*
assumes $m = m * n$ **shows** $n = 1 \vee m = 0$
proof (*rule disjCI*)
assume $m \neq 0$
show $n = 1$
proof (*cases n 1*::*nat rule*: *linorder-cases*)
case *greater*
show *?thesis*
using *assms mult-less-mono2* [*OF greater*, *of m*] ‹$m \neq 0$› **by** *auto*
qed (*use assms* ‹$m \neq 0$› **in** *auto*)
qed

lemma *mono-times-nat*:
fixes n :: *nat*

assumes $n > 0$
shows *mono* (*times* n)
proof
fix m q :: *nat*
assume $m \leq q$
with *assms* **show** $n * m \leq n * q$ **by** *simp*
qed

The lattice order on *nat*.

instantiation *nat* :: *distrib-lattice*
begin

definition (*inf* :: *nat* ⇒ *nat* ⇒ *nat*) = *min*

definition (*sup* :: *nat* ⇒ *nat* ⇒ *nat*) = *max*

instance
by *intro-classes*
(*auto simp add*: *inf-nat-def sup-nat-def max-def not-le min-def*
intro: *order-less-imp-le antisym elim*!: *order-trans order-less-trans*)

end

17.5 Natural operation of natural numbers on functions

We use the same logical constant for the power operations on functions and relations, in order to share the same syntax.

consts *compow* :: *nat* ⇒ 'a ⇒ 'a

abbreviation *compower* :: 'a ⇒ *nat* ⇒ 'a (**infixr** ‹^^› *80*)
where $f \hat{} n \equiv compow\ n\ f$

notation (*latex* **output**)
compower (‹(-^)›) [*1000*] *1000*)

$f \hat{} n = f \circ \ldots \circ f$, the n-fold composition of f

overloading
funpow ≡ *compow* :: *nat* ⇒ ('a ⇒ 'a) ⇒ ('a ⇒ 'a)
begin

primrec *funpow* :: *nat* ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where
funpow 0 f = *id*
| *funpow* (*Suc n*) *f* = *f* ∘ *funpow n f*

end

lemma *funpow-0* [*simp*]: ($f \hat{} 0$) $x = x$

by *simp*

lemma *funpow-Suc-right*: $f \mathbin{\frown\frown} Suc\ n = f \mathbin{\frown\frown} n \circ f$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
fix *n*
assume $f \mathbin{\frown\frown} Suc\ n = f \mathbin{\frown\frown} n \circ f$
then show $f \mathbin{\frown\frown} Suc\ (Suc\ n) = f \mathbin{\frown\frown} Suc\ n \circ f$
by (*simp add*: *o-assoc*)
qed

lemmas *funpow-simps-right* = *funpow.simps*(*1*) *funpow-Suc-right*

For code generation.

context
begin

qualified definition *funpow* :: *nat* ⇒ ('*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a*
where *funpow-code-def* [*code-abbrev*]: *funpow* = *compow*

lemma [*code*]:
funpow 0 f = *id*
funpow (*Suc n*) *f* = *f* ∘ *funpow n f*
by (*simp-all add*: *funpow-code-def*)

end

lemma *funpow-add*: $f \mathbin{\frown\frown} (m + n) = f \mathbin{\frown\frown} m \circ f \mathbin{\frown\frown} n$
by (*induct m*) *simp-all*

lemma *funpow-mult*: $(f \mathbin{\frown\frown} m) \mathbin{\frown\frown} n = f \mathbin{\frown\frown} (m * n)$
for *f* :: '*a* ⇒ '*a*
by (*induct n*) (*simp-all add*: *funpow-add*)

lemma *funpow-swap1*: $f\ ((f \mathbin{\frown\frown} n)\ x) = (f \mathbin{\frown\frown} n)\ (f\ x)$
proof −
have $f\ ((f \mathbin{\frown\frown} n)\ x) = (f \mathbin{\frown\frown} (n + 1))\ x$ **by** *simp*
also have ... $= (f \mathbin{\frown\frown} n \circ f \mathbin{\frown\frown} 1)\ x$ **by** (*simp only*: *funpow-add*)
also have ... $= (f \mathbin{\frown\frown} n)\ (f\ x)$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *comp-funpow*: *comp f* $\mathbin{\frown\frown} n$ = *comp* $(f \mathbin{\frown\frown} n)$
for *f* :: '*a* ⇒ '*a*
by (*induct n*) *simp-all*

lemma *Suc-funpow*[*simp*]: *Suc* $\mathbin{\frown\frown} n = ((+)\ n)$

by (*induct n*) *simp-all*

lemma *id-funpow*[*simp*]: *id* ^^ *n* = *id*
by (*induct n*) *simp-all*

lemma *funpow-mono*: *mono-on UNIV f* ⟹ *A* ≤ *B* ⟹ (*f* ^^ *n*) *A* ≤ (*f* ^^ *n*) *B*
for *f* :: *'a* ⇒ (*'a*::*preorder*)
by (*induct n*) (*auto simp*: *mono-on-def*)

lemma *funpow-mono2*:
fixes *f* :: *'a* ⇒ *'a* :: *preorder*
assumes *mono-on UNIV f*
and *i* ≤ *j*
and *x* ≤ *y*
and *x* ≤ *f x*
shows (*f* ^^ *i*) *x* ≤ (*f* ^^ *j*) *y*
using *assms*(*2*,*3*)
proof (*induct j arbitrary*: *y*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc j*)
show *?case*
proof(*cases i* = *Suc j*)
case *True*
with *assms*(*1*) *Suc* **show** *?thesis*
by (*simp del*: *funpow.simps add*: *funpow-simps-right mono-onD funpow-mono*)
next
case *False*
have *x* ≤ *f y*
using *assms*(*4*) *Suc*(*3*) *mono-onD*[*OF assms*(*1*)] *order.trans*[*of x f x f y*] **by** *simp*
with *False Suc*(*1*,*2*) **show** *?thesis*
by (*simp del*: *funpow.simps add*: *funpow-simps-right le-eq-less-or-eq less-Suc-eq-le*)
qed
qed

lemma *inj-fn*[*simp*]:
fixes *f*::*'a* ⇒ *'a*
assumes *inj f*
shows *inj* (*f*^^*n*)
proof (*induction n*)
case *Suc* **thus** *?case* **using** *inj-compose*[*OF assms Suc.IH*] **by** (*simp del*: *comp-apply*)
qed *simp*

lemma *surj-fn*[*simp*]:
fixes *f*::*'a* ⇒ *'a*
assumes *surj f*
shows *surj* (*f*^^*n*)

proof (*induction n*)
case *Suc* **thus** *?case* **by** (*simp add*: *comp-surj*[*OF Suc.IH assms*] *del*: *comp-apply*)
qed *simp*

lemma *bij-fn*[*simp*]:
fixes $f::'a \Rightarrow 'a$
assumes *bij f*
shows *bij* $(f \frown\frown n)$
by (*rule bijI*[*OF inj-fn*[*OF bij-is-inj*[*OF assms*]] *surj-fn*[*OF bij-is-surj*[*OF assms*]]])

lemma *bij-betw-funpow*:
assumes *bij-betw f S S* **shows** *bij-betw* $(f \frown\frown n)$ *S S*
proof (*induct n*)
case *0* **then show** *?case* **by** (*auto simp*: *id-def*[*symmetric*])
next
case (*Suc n*)
then show *?case* **unfolding** *funpow.simps* **using** *assms* **by** (*rule bij-betw-trans*)
qed

17.6 Kleene iteration

lemma *Kleene-iter-lpfp*:
fixes $f :: 'a::order\text{-}bot \Rightarrow 'a$
assumes *mono f*
and $f\ p \leq p$
shows $(f \frown\frown k)\ bot \leq p$
proof (*induct k*)
case *0*
show *?case* **by** *simp*
next
case *Suc*
show *?case*
using *monoD*[*OF assms*(*1*) *Suc*] *assms*(*2*) **by** *simp*
qed

lemma *lfp-Kleene-iter*:
assumes *mono f*
and $(f \frown\frown Suc\ k)\ bot = (f \frown\frown k)\ bot$
shows $lfp\ f = (f \frown\frown k)\ bot$
proof (*rule antisym*)
show $lfp\ f \leq (f \frown\frown k)\ bot$
proof (*rule lfp-lowerbound*)
show $f\ ((f \frown\frown k)\ bot) \leq (f \frown\frown k)\ bot$
using *assms*(*2*) **by** *simp*
qed
show $(f \frown\frown k)\ bot \leq lfp\ f$
using *Kleene-iter-lpfp*[*OF assms*(*1*)] *lfp-unfold*[*OF assms*(*1*)] **by** *simp*
qed

lemma *mono-pow*: *mono-on UNIV f* $\Longrightarrow$ *mono-on UNIV* (*f* $\widehat{\ }\widehat{\ }$ *n*)
for *f* :: *'a* $\Rightarrow$ *'a*::*preorder*
by (*induct n*) (*auto simp*: *mono-on-def*)

lemma *lfp-funpow*:
assumes *f*: *mono f*
shows *lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*) = *lfp f*
proof (*rule antisym*)
show *lfp f* $\leq$ *lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)
proof (*rule lfp-lowerbound*)
have *f* (*lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)) = *lfp* ($\lambda x.$ *f* ((*f* $\widehat{\ }\widehat{\ }$ *n*) *x*))
unfolding *funpow-Suc-right* **by** (*simp add*: *lfp-rolling f mono-pow comp-def*)
then show *f* (*lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)) $\leq$ *lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)
by (*simp add*: *comp-def*)
qed
have (*f* $\widehat{\ }\widehat{\ }$ *n*) (*lfp f*) = *lfp f* **for** *n*
by (*induct n*) (*auto intro*: *f lfp-fixpoint*)
then show *lfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*) $\leq$ *lfp f*
by (*intro lfp-lowerbound*) (*simp del*: *funpow.simps*)
qed

lemma *gfp-funpow*:
assumes *f*: *mono f*
shows *gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*) = *gfp f*
proof (*rule antisym*)
show *gfp f* $\geq$ *gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)
proof (*rule gfp-upperbound*)
have *f* (*gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)) = *gfp* ($\lambda x.$ *f* ((*f* $\widehat{\ }\widehat{\ }$ *n*) *x*))
unfolding *funpow-Suc-right* **by** (*simp add*: *gfp-rolling f mono-pow comp-def*)
then show *f* (*gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)) $\geq$ *gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*)
by (*simp add*: *comp-def*)
qed
have (*f* $\widehat{\ }\widehat{\ }$ *n*) (*gfp f*) = *gfp f* **for** *n*
by (*induct n*) (*auto intro*: *f gfp-fixpoint*)
then show *gfp* (*f* $\widehat{\ }\widehat{\ }$ *Suc n*) $\geq$ *gfp f*
by (*intro gfp-upperbound*) (*simp del*: *funpow.simps*)
qed

lemma *Kleene-iter-gpfp*:
fixes *f* :: *'a*::*order-top* $\Rightarrow$ *'a*
assumes *mono f*
and *p* $\leq$ *f p*
shows *p* $\leq$ (*f* $\widehat{\ }\widehat{\ }$ *k*) *top*
proof (*induct k*)
case *0*
show *?case* **by** *simp*
next
case *Suc*
show *?case*

using *monoD*[*OF assms*(*1*) *Suc*] *assms*(*2*) **by** *simp*
qed

lemma *gfp-Kleene-iter*:
assumes *mono f*
and (*f* ^^ *Suc k*) *top* = (*f* ^^ *k*) *top*
shows *gfp f* = (*f* ^^ *k*) *top*
(**is** *?lhs* = *?rhs*)
proof (*rule antisym*)
have *?rhs* ≤ *f ?rhs*
using *assms*(*2*) **by** *simp*
then show *?rhs* ≤ *?lhs*
by (*rule gfp-upperbound*)
show *?lhs* ≤ *?rhs*
using *Kleene-iter-gpfp*[*OF assms*(*1*)] *gfp-unfold*[*OF assms*(*1*)] **by** *simp*
qed

17.7 Embedding of the naturals into any *semiring-1*: *of-nat*

context *semiring-1*
begin

definition *of-nat* :: *nat* ⇒ *'a*
where *of-nat n* = (*plus 1* ^^ *n*) *0*

lemma *of-nat-simps* [*simp*]:
shows *of-nat-0*: *of-nat 0* = *0*
and *of-nat-Suc*: *of-nat* (*Suc m*) = *1* + *of-nat m*
by (*simp-all add*: *of-nat-def*)

lemma *of-nat-1* [*simp*]: *of-nat 1* = *1*
by (*simp add*: *of-nat-def*)

lemma *of-nat-add* [*simp*]: *of-nat* (*m* + *n*) = *of-nat m* + *of-nat n*
by (*induct m*) (*simp-all add*: *ac-simps*)

lemma *of-nat-mult* [*simp*]: *of-nat* (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
by (*induct m*) (*simp-all add*: *ac-simps distrib-right*)

lemma *mult-of-nat-commute*: *of-nat x* ∗ *y* = *y* ∗ *of-nat x*
by (*induct x*) (*simp-all add*: *algebra-simps*)

primrec *of-nat-aux* :: (*'a* ⇒ *'a*) ⇒ *nat* ⇒ *'a* ⇒ *'a*
where
of-nat-aux inc 0 i = *i*
| *of-nat-aux inc* (*Suc n*) *i* = *of-nat-aux inc n* (*inc i*) — tail recursive

lemma *of-nat-code*: *of-nat n* = *of-nat-aux* (λ*i*. *i* + *1*) *n 0*
proof (*induct n*)

case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have $\bigwedge i$. *of-nat-aux* (λi. *i + 1*) *n* (*i + 1*) = *of-nat-aux* (λi. *i + 1*) *n i + 1*
by (*induct n*) *simp-all*
from *this* [*of 0*] **have** *of-nat-aux* (λi. *i + 1*) *n 1* = *of-nat-aux* (λi. *i + 1*) *n 0 + 1*
by *simp*
with *Suc* **show** *?case*
by (*simp add*: *add.commute*)
qed

lemma *of-nat-of-bool* [*simp*]:
of-nat (*of-bool P*) = *of-bool P*
by *auto*

end

declare *of-nat-code* [*code*]

context *semiring-1-cancel*
begin

lemma *of-nat-diff* [*simp*]:
‹*of-nat* (*m − n*) = *of-nat m − of-nat n*› **if** ‹*n ≤ m*›
proof −
from *that* **obtain** *q* **where** ‹*m* = *n* + *q*›
by (*blast dest*: *le-Suc-ex*)
then show *?thesis*
by *simp*
qed

lemma *of-nat-diff-if*: ‹*of-nat* (*m − n*) = (*if n≤m then of-nat m − of-nat n else 0*)›
by (*simp add*: *not-le less-imp-le*)

end

Class for unital semirings with characteristic zero. Includes non-ordered rings like the complex numbers.

class *semiring-char-0* = *semiring-1* +
assumes *inj-of-nat*: *inj of-nat*
begin

lemma *of-nat-eq-iff* [*simp*]: *of-nat m* = *of-nat n* ⟷ *m* = *n*
by (*auto intro*: *inj-of-nat injD*)

Special cases where either operand is zero

lemma *of-nat-0-eq-iff* [*simp*]: *0 = of-nat n ⟷ 0 = n*
by (*fact of-nat-eq-iff* [*of 0 n*, *unfolded of-nat-0*])

lemma *of-nat-eq-0-iff* [*simp*]: *of-nat m = 0 ⟷ m = 0*
by (*fact of-nat-eq-iff* [*of m 0*, *unfolded of-nat-0*])

lemma *of-nat-1-eq-iff* [*simp*]: *1 = of-nat n ⟷ n=1*
using *of-nat-eq-iff* **by** *fastforce*

lemma *of-nat-eq-1-iff* [*simp*]: *of-nat n = 1 ⟷ n=1*
using *of-nat-eq-iff* **by** *fastforce*

lemma *of-nat-neq-0* [*simp*]: *of-nat* (*Suc n*) ≠ *0*
unfolding *of-nat-eq-0-iff* **by** *simp*

lemma *of-nat-0-neq* [*simp*]: *0* ≠ *of-nat* (*Suc n*)
unfolding *of-nat-0-eq-iff* **by** *simp*

end

class *ring-char-0 = ring-1 + semiring-char-0*

lemma (**in** *ordered-semiring-1*) *of-nat-0-le-iff* [*simp*]: *0* ≤ *of-nat n*
by (*induct n*) *simp-all*

context *linordered-nonzero-semiring*
begin

lemma *of-nat-less-0-iff* [*simp*]: ¬ *of-nat m < 0*
by (*simp add*: *not-less*)

lemma *of-nat-mono*[*simp*]: *i* ≤ *j* ⟹ *of-nat i* ≤ *of-nat j*
by (*auto simp*: *le-iff-add intro*!: *add-increasing2*)

lemma *of-nat-less-iff* [*simp*]: *of-nat m < of-nat n ⟷ m < n*
proof(*induct m n rule*: *diff-induct*)
case (*1 m*) **then show** *?case*
by *auto*
next
case (*2 n*) **then show** *?case*
by (*simp add*: *add-pos-nonneg*)
next
case (*3 m n*)
then show *?case*
by (*auto simp*: *add-commute* [*of 1*] *add-mono1 not-less add-right-mono leD*)
qed

lemma *of-nat-le-iff* [*simp*]: *of-nat m* ≤ *of-nat n ⟷ m* ≤ *n*
by (*simp add*: *not-less* [*symmetric*] *linorder-not-less* [*symmetric*])

lemma *less-imp-of-nat-less*: $m < n \Longrightarrow$ *of-nat* $m <$ *of-nat* n
by *simp*

lemma *of-nat-less-imp-less*: *of-nat* $m <$ *of-nat* $n \Longrightarrow m < n$
by *simp*

Every *linordered-nonzero-semiring* has characteristic zero.

subclass *semiring-char-0*
by *standard* (*auto intro*!: *injI simp add*: *order.eq-iff*)

Special cases where either operand is zero

lemma *of-nat-le-0-iff* [*simp*]: *of-nat* $m \leq 0 \longleftrightarrow m = 0$
by (*rule of-nat-le-iff* [*of - 0*, *simplified*])

lemma *of-nat-0-less-iff* [*simp*]: $0 <$ *of-nat* $n \longleftrightarrow 0 < n$
by (*rule of-nat-less-iff* [*of 0*, *simplified*])

end

context *linordered-nonzero-semiring*
begin

lemma *of-nat-max*: *of-nat* (*max x y*) = *max* (*of-nat x*) (*of-nat y*)
by (*auto simp*: *max-def ord-class.max-def*)

lemma *of-nat-min*: *of-nat* (*min x y*) = *min* (*of-nat x*) (*of-nat y*)
by (*auto simp*: *min-def ord-class.min-def*)

end

context *linordered-semidom*
begin

subclass *linordered-nonzero-semiring* **..**

subclass *semiring-char-0* **..**

end

context *linordered-idom*
begin

lemma *abs-of-nat* [*simp*]:
$|$*of-nat* $n|$ = *of-nat* n
by (*simp add*: *abs-if*)

lemma *sgn-of-nat* [*simp*]:
sgn (*of-nat* n) = *of-bool* ($n > 0$)

by *simp*

end

lemma *of-nat-id* [*simp*]: *of-nat n = n*
by (*induct n*) *simp-all*

lemma *of-nat-eq-id* [*simp*]: *of-nat = id*
by (*auto simp add*: *fun-eq-iff*)

17.8 The set of natural numbers

context *semiring-1*
begin

definition *Nats* :: *'a set* (‹ℕ›)
where ℕ = *range of-nat*

lemma *of-nat-in-Nats* [*simp*]: *of-nat n* ∈ ℕ
by (*simp add*: *Nats-def*)

lemma *Nats-0* [*simp*]: *0* ∈ ℕ
using *of-nat-0* [*symmetric*] **unfolding** *Nats-def*
by (*rule range-eqI*)

lemma *Nats-1* [*simp*]: *1* ∈ ℕ
using *of-nat-1* [*symmetric*] **unfolding** *Nats-def*
by (*rule range-eqI*)

lemma *Nats-add* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a + b* ∈ ℕ
unfolding *Nats-def* **using** *of-nat-add* [*symmetric*]
by (*blast intro*: *range-eqI*)

lemma *Nats-mult* [*simp*]: *a* ∈ ℕ ⟹ *b* ∈ ℕ ⟹ *a * b* ∈ ℕ
unfolding *Nats-def* **using** *of-nat-mult* [*symmetric*]
by (*blast intro*: *range-eqI*)

lemma *Nats-cases* [*cases set*: *Nats*]:
assumes *x* ∈ ℕ
obtains (*of-nat*) *n* **where** *x = of-nat n*
unfolding *Nats-def*
proof −
from ‹*x* ∈ ℕ› **have** *x* ∈ *range of-nat* **unfolding** *Nats-def* **.**
then obtain *n* **where** *x = of-nat n* **..**
then show *thesis* **..**
qed

lemma *Nats-induct* [*case-names of-nat*, *induct set*: *Nats*]: *x* ∈ ℕ ⟹ (⋀*n*. *P* (*of-nat n*)) ⟹ *P x*

by (*rule Nats-cases*) *auto*

lemma *Nats-nonempty* [*simp*]: $\mathbb{N} \neq \{\}$
unfolding *Nats-def* **by** *auto*

end

lemma *Nats-diff* [*simp*]:
fixes *a*:: *'a*::*linordered-idom*
assumes $a \in \mathbb{N}$ $b \in \mathbb{N}$ $b \leq a$ **shows** $a - b \in \mathbb{N}$
proof −
obtain *i* **where** *i*: $a =$ *of-nat i*
using *Nats-cases assms* **by** *blast*
obtain *j* **where** *j*: $b =$ *of-nat j*
using *Nats-cases assms* **by** *blast*
have $j \leq i$
using ‹$b \leq a$› *i j of-nat-le-iff* **by** *blast*
then have ∗: *of-nat i* − *of-nat j* = (*of-nat* (*i*−*j*) :: *'a*)
by (*simp add*: *of-nat-diff*)
then show *?thesis*
by (*simp add*: ∗ *i j*)
qed

17.9 Further arithmetic facts concerning the natural numbers

lemma *subst-equals*:
assumes $t = s$ **and** $u = t$
shows $u = s$
using *assms*(*2*,*1*) **by** (*rule trans*)

locale *nat-arith*
begin

lemma *add1*: (*A*::*'a*::*comm-monoid-add*) ≡ $k + a \Longrightarrow A + b \equiv k + (a + b)$
by (*simp only*: *ac-simps*)

lemma *add2*: (*B*::*'a*::*comm-monoid-add*) ≡ $k + b \Longrightarrow a + B \equiv k + (a + b)$
by (*simp only*: *ac-simps*)

lemma *suc1*: $A == k + a \Longrightarrow Suc\ A \equiv k + Suc\ a$
by (*simp only*: *add-Suc-right*)

lemma *rule0*: (*a*::*'a*::*comm-monoid-add*) ≡ $a + 0$
by (*simp only*: *add-0-right*)

end

ML-file ‹*Tools/nat-arith.ML*›

simproc-setup *nateq-cancel-sums*
((*l*::*nat*) + *m* = *n* | (*l*::*nat*) = *m* + *n* | *Suc m* = *n* | *m* = *Suc n*) =
‹*K* (*try o Nat-Arith.cancel-eq-conv*)›

simproc-setup *natless-cancel-sums*
((*l*::*nat*) + *m* < *n* | (*l*::*nat*) < *m* + *n* | *Suc m* < *n* | *m* < *Suc n*) =
‹*K* (*try o Nat-Arith.cancel-less-conv*)›

simproc-setup *natle-cancel-sums*
((*l*::*nat*) + *m* ≤ *n* | (*l*::*nat*) ≤ *m* + *n* | *Suc m* ≤ *n* | *m* ≤ *Suc n*) =
‹*K* (*try o Nat-Arith.cancel-le-conv*)›

simproc-setup *natdiff-cancel-sums*
((*l*::*nat*) + *m* − *n* | (*l*::*nat*) − (*m* + *n*) | *Suc m* − *n* | *m* − *Suc n*) =
‹*K* (*try o Nat-Arith.cancel-diff-conv*)›

context *preorder*
begin

lemma *lift-Suc-mono-le*:
assumes *mono*: ⋀*n*. *f n* ≤ *f* (*Suc n*)
and *n* ≤ *n'*
shows *f n* ≤ *f n'*
proof (*cases n* < *n'*)
case *True*
then show *?thesis*
by (*induct n n' rule*: *less-Suc-induct*) (*auto intro*: *mono order.trans*)
next
case *False*
with ‹*n* ≤ *n'*› **show** *?thesis* **by** *auto*
qed

lemma *lift-Suc-antimono-le*:
assumes *mono*: ⋀*n*. *f n* ≥ *f* (*Suc n*)
and *n* ≤ *n'*
shows *f n* ≥ *f n'*
proof (*cases n* < *n'*)
case *True*
then show *?thesis*
by (*induct n n' rule*: *less-Suc-induct*) (*auto intro*: *mono order.trans*)
next
case *False*
with ‹*n* ≤ *n'*› **show** *?thesis* **by** *auto*
qed

lemma *lift-Suc-mono-less*:
assumes *mono*: ⋀*n*. *f n* < *f* (*Suc n*)
and *n* < *n'*

shows $f\ n < f\ n'$
using ‹$n < n'$› **by** (*induct n n' rule*: *less-Suc-induct*) (*auto intro*: *mono order.strict-trans*)

lemma *lift-Suc-mono-less-iff*: $(\bigwedge n.\ f\ n < f\ (Suc\ n)) \Longrightarrow f\ n < f\ m \longleftrightarrow n < m$
by (*blast intro*: *less-asym'* *lift-Suc-mono-less* [*of f*]
dest: *linorder-not-less*[*THEN iffD1*] *le-eq-less-or-eq* [*THEN iffD1*])

end

lemma *mono-iff-le-Suc*: *mono-on UNIV f* $\longleftrightarrow$ $(\forall n.\ f\ n \leq f\ (Suc\ n))$
for $f :: nat \Rightarrow 'a$::*preorder*
unfolding *mono-on-def* **by** (*auto intro*: *lift-Suc-mono-le* [*of f*])

lemma *antimono-iff-le-Suc*: *antimono-on UNIV f* $\longleftrightarrow$ $(\forall n.\ f\ (Suc\ n) \leq f\ n)$
for $f :: nat \Rightarrow 'a$::*preorder*
unfolding *monotone-on-def* **by** (*auto intro*: *lift-Suc-antimono-le* [*of f*])

lemma *strict-mono-Suc-iff*: *strict-mono-on UNIV f* $\longleftrightarrow$ $(\forall n.\ f\ n < f\ (Suc\ n))$
for $f :: nat \Rightarrow 'a$::*preorder*
proof (*intro iffI strict-mono-onI*)
assume $*$: $\forall n.\ f\ n < f\ (Suc\ n)$
fix $m\ n :: nat$ **assume** $m < n$
thus $f\ m < f\ n$
by (*induction rule*: *less-Suc-induct*) (*use* $*$ **in** ‹*auto intro*: *order.strict-trans*›)
qed (*auto simp*: *strict-mono-on-def*)

lemma *strict-mono-add*: *strict-mono* $(\lambda n::'a$::*linordered-semidom*. $n + k)$
by (*auto simp*: *strict-mono-def*)

lemma *mono-nat-linear-lb*:
fixes $f :: nat \Rightarrow nat$
assumes $\bigwedge m\ n.\ m < n \Longrightarrow f\ m < f\ n$
shows $f\ m + k \leq f\ (m + k)$
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
then have *Suc* $(f\ m + k) \leq$ *Suc* $(f\ (m + k))$ **by** *simp*
also from *assms* [*of* $m + k$ *Suc* $(m + k)$] **have** *Suc* $(f\ (m + k)) \leq f\ ($*Suc* $(m + k))$
by (*simp add*: *Suc-le-eq*)
finally show *?case* **by** *simp*
qed

lemma *bex-const1-if-mono-below-diag*: **fixes** $f :: nat \Rightarrow nat$ **assumes** *mono f*
shows $f\ n < n \Longrightarrow \exists i{<}n.\ f(Suc\ i) = f\ i$
proof(*induction n*)

```
  case 0
  then show ?case by simp
next
  case (Suc n)
  have *: f n ≤ f(Suc n) using assms[simplified mono-iff-le-Suc] by blast
  from Suc.prems[simplified less-Suc-eq]
  show ?case
  proof
    assume f(Suc n) < n
    from order.strict-trans1[OF * this]
    show ?thesis using Suc.IH less-SucI by blast
  next
    assume f(Suc n) = n
    from order.strict-trans1[OF * Suc.prems, simplified less-Suc-eq]
    show ?case
    proof
      assume f n < n
      thus ?thesis using Suc.IH less-SucI by blast
    next
      assume f n = n
      with ‹f(Suc n) = n› show ?thesis by auto
    qed
  qed
qed

lemma bex-const1-if-mono-below-diag-Suc:
fixes f :: nat ⇒ nat assumes mono f f(Suc m) ≤ m
shows ∃ i≤m. f (Suc i) = f i
using bex-const1-if-mono-below-diag[OF assms(1), of Suc m] assms(2) less-Suc-eq-le
by blast
```

Subtraction laws, mostly by Clemens Ballarin

```
lemma diff-less-mono:
  fixes a b c :: nat
  assumes a < b and c ≤ a
  shows a − c < b − c
proof −
  from assms obtain d e where b = c + (d + e) and a = c + e and d > 0
    by (auto dest!: le-Suc-ex less-imp-Suc-add simp add: ac-simps)
  then show ?thesis by simp
qed

lemma less-diff-conv: i < j − k ⟷ i + k < j
  for i j k :: nat
  by (cases k ≤ j) (auto simp add: not-le dest: less-imp-Suc-add le-Suc-ex)

lemma less-diff-conv2: k ≤ j ⟹ j − k < i ⟷ j < i + k
  for j k i :: nat
  by (auto dest: le-Suc-ex)
```

lemma *le-diff-conv*: $j - k \leq i \longleftrightarrow j \leq i + k$
for *j k i* :: *nat*
by (*cases* $k \leq j$) (*auto simp add*: *not-le dest*!: *less-imp-Suc-add le-Suc-ex*)

lemma *diff-diff-cancel* [*simp*]: $i \leq n \Longrightarrow n - (n - i) = i$
for *i n* :: *nat*
by (*auto dest*: *le-Suc-ex*)

lemma *diff-less* [*simp*]: $0 < n \Longrightarrow 0 < m \Longrightarrow m - n < m$
for *i n* :: *nat*
by (*auto dest*: *less-imp-Suc-add*)

Simplification of relational expressions involving subtraction

lemma *diff-diff-eq*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k - (n - k) = m - n$
for *m n k* :: *nat*
by (*auto dest*!: *le-Suc-ex*)

hide-fact (**open**) *diff-diff-eq*

lemma *eq-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k = n - k \longleftrightarrow m = n$
for *m n k* :: *nat*
by (*auto dest*: *le-Suc-ex*)

lemma *less-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k < n - k \longleftrightarrow m < n$
for *m n k* :: *nat*
by (*auto dest*!: *le-Suc-ex*)

lemma *le-diff-iff*: $k \leq m \Longrightarrow k \leq n \Longrightarrow m - k \leq n - k \longleftrightarrow m \leq n$
for *m n k* :: *nat*
by (*auto dest*!: *le-Suc-ex*)

lemma *le-diff-iff'*: $a \leq c \Longrightarrow b \leq c \Longrightarrow c - a \leq c - b \longleftrightarrow b \leq a$
for *a b c* :: *nat*
by (*force dest*: *le-Suc-ex*)

(Anti)Monotonicity of subtraction – by Stephan Merz

lemma *diff-le-mono*: $m \leq n \Longrightarrow m - l \leq n - l$
for *m n l* :: *nat*
by (*auto dest*: *less-imp-le less-imp-Suc-add split*: *nat-diff-split*)

lemma *diff-le-mono2*: $m \leq n \Longrightarrow l - n \leq l - m$
for *m n l* :: *nat*
by (*auto dest*: *less-imp-le le-Suc-ex less-imp-Suc-add less-le-trans split*: *nat-diff-split*)

lemma *diff-less-mono2*: $m < n \Longrightarrow m < l \Longrightarrow l - n < l - m$
for *m n l* :: *nat*
by (*auto dest*: *less-imp-Suc-add split*: *nat-diff-split*)

lemma *diffs0-imp-equal*: $m - n = 0 \Longrightarrow n - m = 0 \Longrightarrow m = n$
for $m\ n$ :: *nat*
by (*simp split*: *nat-diff-split*)

lemma *min-diff*: $min\ (m - i)\ (n - i) = min\ m\ n - i$
for $m\ n\ i$ :: *nat*
by (*cases m n rule*: *le-cases*)
(*auto simp add*: *not-le min.absorb1 min.absorb2 min.absorb-iff1* [*symmetric*] *diff-le-mono*)

lemma *inj-on-diff-nat*:
fixes k :: *nat*
assumes $\bigwedge n.\ n \in N \Longrightarrow k \leq n$
shows *inj-on* $(\lambda n.\ n - k)\ N$
proof (*rule inj-onI*)
fix $x\ y$
assume *a*: $x \in N\ y \in N\ x - k = y - k$
with *assms* **have** $x - k + k = y - k + k$ **by** *auto*
with *a assms* **show** $x = y$ **by** (*auto simp add*: *eq-diff-iff*)
qed

Rewriting to pull differences out

lemma *diff-diff-right* [*simp*]: $k \leq j \Longrightarrow i - (j - k) = i + k - j$
for $i\ j\ k$ :: *nat*
by (*fact diff-diff-right*)

lemma *diff-Suc-diff-eq1* [*simp*]:
assumes $k \leq j$
shows $i - Suc\ (j - k) = i + k - Suc\ j$
proof −
from *assms* **have** ∗: $Suc\ (j - k) = Suc\ j - k$
by (*simp add*: *Suc-diff-le*)
from *assms* **have** $k \leq Suc\ j$
by (*rule order-trans*) *simp*
with *diff-diff-right* [*of k Suc j i*] ∗ **show** *?thesis*
by *simp*
qed

lemma *diff-Suc-diff-eq2* [*simp*]:
assumes $k \leq j$
shows $Suc\ (j - k) - i = Suc\ j - (k + i)$
proof −
from *assms* **obtain** n **where** $j = k + n$
by (*auto dest*: *le-Suc-ex*)
moreover have $Suc\ n - i = (k + Suc\ n) - (k + i)$
using *add-diff-cancel-left* [*of k Suc n i*] **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *Suc-diff-Suc*:
assumes $n < m$
shows $Suc\ (m - Suc\ n) = m - n$
proof −
from *assms* **obtain** q **where** $m = n + Suc\ q$
by (*auto dest*: *less-imp-Suc-add*)
moreover define r **where** $r = Suc\ q$
ultimately have $Suc\ (m - Suc\ n) = r$ **and** $m = n + r$
by *simp-all*
then show *?thesis* **by** *simp*
qed

lemma *one-less-mult*: $Suc\ 0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow Suc\ 0 < m * n$
using *less-1-mult* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *n-less-m-mult-n*: $0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow n < m * n$
using *mult-strict-right-mono* [*of 1 m n*] **by** *simp*

lemma *n-less-n-mult-m*: $0 < n \Longrightarrow Suc\ 0 < m \Longrightarrow n < n * m$
using *mult-strict-left-mono* [*of 1 m n*] **by** *simp*

Induction starting beyond zero

lemma *nat-induct-at-least* [*consumes 1*, *case-names base Suc*]:
$P\ n$ **if** $n \geq m\ P\ m\ \bigwedge n.\ n \geq m \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)$
proof −
define q **where** $q = n - m$
with ‹$n \geq m$› **have** $n = m + q$
by *simp*
moreover have $P\ (m + q)$
by (*induction q*) (*use that* **in** *simp-all*)
ultimately show $P\ n$
by *simp*
qed

lemma *nat-induct-non-zero* [*consumes 1*, *case-names 1 Suc*]:
$P\ n$ **if** $n > 0\ P\ 1\ \bigwedge n.\ n > 0 \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)$
proof −
from ‹$n > 0$› **have** $n \geq 1$
by (*cases n*) *simp-all*
moreover note ‹$P\ 1$›
moreover have $\bigwedge n.\ n \geq 1 \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)$
using ‹$\bigwedge n.\ n > 0 \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)$›
by (*simp add*: *Suc-le-eq*)
ultimately show $P\ n$
by (*rule nat-induct-at-least*)
qed

Specialized induction principles that work "backwards":

lemma *inc-induct* [*consumes 1*, *case-names base step*]:

```
  assumes less: i ≤ j
    and base: P j
    and step: ⋀n. i ≤ n ⟹ n < j ⟹ P (Suc n) ⟹ P n
  shows P i
  using less step
proof (induct j − i arbitrary: i)
  case (0 i)
  then have i = j by simp
  with base show ?case by simp
next
  case (Suc d n)
  from Suc.hyps have n ≠ j by auto
  with Suc have n < j by (simp add: less-le)
  from ‹Suc d = j − n› have d + 1 = j − n by simp
  then have d + 1 − 1 = j − n − 1 by simp
  then have d = j − n − 1 by simp
  then have d = j − (n + 1) by (simp add: diff-diff-eq)
  then have d = j − Suc n by simp
  moreover from ‹n < j› have Suc n ≤ j by (simp add: Suc-le-eq)
  ultimately have P (Suc n)
  proof (rule Suc.hyps)
    fix q
    assume Suc n ≤ q
    then have n ≤ q by (simp add: Suc-le-eq less-imp-le)
    moreover assume q < j
    moreover assume P (Suc q)
    ultimately show P q by (rule Suc.prems)
  qed
  with order-refl ‹n < j› show P n by (rule Suc.prems)
qed

lemma strict-inc-induct [consumes 1, case-names base step]:
  assumes less: i < j
    and base: ⋀i. j = Suc i ⟹ P i
    and step: ⋀i. Suc i < j ⟹ P (Suc i) ⟹ P i
  shows P i
using less proof (induct j − i − 1 arbitrary: i)
  case (0 i)
  from ‹i < j› obtain n where j = i + n and n > 0
    by (auto dest!: less-imp-Suc-add)
  with 0 have j = Suc i
    by (auto intro: order-antisym simp add: Suc-le-eq)
  with base show ?case by simp
next
  case (Suc d i)
  from ‹Suc d = j − i − 1› have ∗: Suc d = j − Suc i
    by (simp add: diff-diff-add)
  then have Suc d − 1 = j − Suc i − 1 by simp
  then have d = j − Suc i − 1 by simp
```

moreover from $*$ **have** $j - Suc\ i \neq 0$ **by** *auto*
then have $Suc\ i < j$ **by** (*simp add*: *not-le*)
ultimately have $P\ (Suc\ i)$ **by** (*rule Suc.hyps*)
with ‹$Suc\ i < j$› **show** $P\ i$ **by** (*rule step*)
qed

lemma *zero-induct-lemma*: $P\ k \Longrightarrow (\bigwedge n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ (k - i)$
using *inc-induct*[*of* $k - i\ k\ P$, *simplified*] **by** *blast*

lemma *zero-induct*: $P\ k \Longrightarrow (\bigwedge n.\ P\ (Suc\ n) \Longrightarrow P\ n) \Longrightarrow P\ 0$
using *inc-induct*[*of* $0\ k\ P$] **by** *blast*

Further induction rule similar to $[\![?i \leq ?j;\ ?P\ ?j;\ \bigwedge n.\ [\![?i \leq n;\ n < ?j;\ ?P\ (Suc\ n)]\!] \Longrightarrow ?P\ n]\!] \Longrightarrow ?P\ ?i$.

lemma *dec-induct* [*consumes 1*, *case-names base step*]:
$i \leq j \Longrightarrow P\ i \Longrightarrow (\bigwedge n.\ i \leq n \Longrightarrow n < j \Longrightarrow P\ n \Longrightarrow P\ (Suc\ n)) \Longrightarrow P\ j$
proof (*induct j arbitrary*: *i*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc j*)
from *Suc.prems* **consider** $i \leq j \mid i = Suc\ j$
by (*auto simp add*: *le-Suc-eq*)
then show *?case*
proof *cases*
case *1*
moreover have $j < Suc\ j$ **by** *simp*
moreover have $P\ j$ **using** ‹$i \leq j$› ‹$P\ i$›
proof (*rule Suc.hyps*)
fix *q*
assume $i \leq q$
moreover assume $q < j$ **then have** $q < Suc\ j$
by (*simp add*: *less-Suc-eq*)
moreover assume $P\ q$
ultimately show $P\ (Suc\ q)$ **by** (*rule Suc.prems*)
qed
ultimately show $P\ (Suc\ j)$ **by** (*rule Suc.prems*)
next
case *2*
with ‹$P\ i$› **show** $P\ (Suc\ j)$ **by** *simp*
qed
qed

lemma *transitive-stepwise-le*:
assumes $m \leq n$ $\bigwedge x.\ R\ x\ x$ $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$ **and** $\bigwedge n.\ R\ n\ (Suc\ n)$
shows $R\ m\ n$
using ‹$m \leq n$›
by (*induction rule*: *dec-induct*) (*use assms* **in** *blast*)+

17.9.1 Greatest operator

lemma *ex-has-greatest-nat*:
  $P\ (k{::}nat) \Longrightarrow \forall y.\ P\ y \longrightarrow y \leq b \Longrightarrow \exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow y \leq x)$
proof (*induction b−k arbitrary*: *b k rule*: *less-induct*)
  case *less*
  show *?case*
  proof *cases*
    assume $\exists n{>}k.\ P\ n$
    then obtain *n* **where** $n{>}k\ P\ n$ **by** *blast*
    have $n \leq b$ **using** ‹*P n*› *less.prems*(*2*) **by** *auto*
    hence $b{-}n < b{-}k$
      by(*rule diff-less-mono2*[*OF* ‹*k*<*n*› *less-le-trans*[*OF* ‹*k*<*n*›]])
    from *less.hyps*[*OF this* ‹*P n*› *less.prems*(*2*)]
    show *?thesis* **.**
  next
    assume $\neg\ (\exists n{>}k.\ P\ n)$
    hence $\forall y.\ P\ y \longrightarrow y \leq k$ **by** (*auto simp*: *not-less*)
    thus *?thesis* **using** *less.prems*(*1*) **by** *auto*
  qed
qed

lemma
  fixes *k*::*nat*
  assumes *P k* **and** *minor*: $\bigwedge y.\ P\ y \Longrightarrow y \leq b$
  shows *GreatestI-nat*: *P* (*Greatest P*)
    and *Greatest-le-nat*: $k \leq$ *Greatest P*
proof −
  obtain *x* **where** *P x* $\bigwedge y.\ P\ y \Longrightarrow y \leq x$
    using *assms ex-has-greatest-nat* **by** *blast*
  with ‹*P k*› **show** *P* (*Greatest P*) $k \leq$ *Greatest P*
    using *GreatestI2-order* **by** *blast*+
qed

lemma *GreatestI-ex-nat*:
  $[\![\ \exists k{::}nat.\ P\ k;\ \bigwedge y.\ P\ y \Longrightarrow y \leq b\ ]\!] \Longrightarrow P\ (Greatest\ P)$
  by (*blast intro*: *GreatestI-nat*)

17.10 Monotonicity of *funpow*

lemma *funpow-increasing*: $m \leq n \Longrightarrow mono\ f \Longrightarrow (f \frown\frown n)\ \top \leq (f \frown\frown m)\ \top$
  for $f :: 'a{::}order\text{-}top \Rightarrow 'a$
  by (*induct rule*: *inc-induct*)
    (*auto simp del*: *funpow.simps*(*2*) *simp add*: *funpow-Suc-right*
      intro: *order-trans*[*OF - funpow-mono*])

lemma *funpow-decreasing*: $m \leq n \Longrightarrow mono\ f \Longrightarrow (f \frown\frown m)\ \bot \leq (f \frown\frown n)\ \bot$
  for $f :: 'a{::}order\text{-}bot \Rightarrow 'a$
  by (*induct rule*: *dec-induct*)
    (*auto simp del*: *funpow.simps*(*2*) *simp add*: *funpow-Suc-right*

intro: *order-trans*[*OF - funpow-mono*])

lemma *mono-funpow*: *mono Q* ⟹ *mono* (λ*i*. (*Q* ^^ *i*) ⊥)
for *Q* :: *'a*::*order-bot* ⇒ *'a*
by (*auto intro*!: *funpow-decreasing simp*: *mono-def*)

lemma *antimono-funpow*: *mono Q* ⟹ *antimono* (λ*i*. (*Q* ^^ *i*) ⊤)
for *Q* :: *'a*::*order-top* ⇒ *'a*
by (*auto intro*!: *funpow-increasing simp*: *antimono-def*)

17.11 Kleene's fixed point theorem for continuous functions

Kleene's fixed point theorem shows that the *lfp* of a omega-continuous function can be obtained as the supremum of an omega chain. It only requires an omega-complete partial order. We prove it here for complete lattices because the latter structures are not defined in Main but the theorem is also useful for complete lattices.

definition *omega-chain* :: (*nat* ⇒ (*'a*::*complete-lattice*)) ⇒ *bool* **where**
omega-chain C = (∀ *i*. *C i* ≤ *C*(*Suc i*))

definition *omega-cont* :: ((*'a*::*complete-lattice*) ⇒ (*'b*::*complete-lattice*)) ⇒ *bool*
where
omega-cont f = (∀ *C*. *omega-chain C* ⟶ *f*(*SUP n*. *C n*) = (*SUP n*. *f*(*C n*)))

lemma *omega-chain-mono*: *omega-chain C* ⟹ *i* ≤ *j* ⟹ *C i* ≤ *C j*
unfolding *omega-chain-def* **using** *lift-Suc-mono-le*[*of C*]
by(*induction j*−*i arbitrary*: *i j*)*auto*

lemma *mono-if-omega-cont*: **fixes** *f* :: (*'a*::*complete-lattice*) ⇒ (*'b*::*complete-lattice*)
assumes *omega-cont f* **shows** *mono f*
proof
fix *a b* :: *'a* **assume** *a* ≤ *b*
let *?C* = λ*n*::*nat*. *if n*=*0 then a else b*
have ∗: *omega-chain ?C* **using** ‹*a* ≤ *b*› **by**(*auto simp*: *omega-chain-def*)
have *f a* ≤ *sup* (*f a*) (*SUP n*. *f*(*?C n*)) **by**(*rule sup.cobounded1*)
also have ... = *sup* (*f*(*?C 0*)) (*SUP n*. *f*(*?C n*)) **by** (*simp*)
also have ... = (*SUP n*. *f* (*?C n*)) **using** *SUP-absorb*[*OF UNIV-I*] **.**
also have ... = *f* (*SUP n*. *?C n*)
using *assms* ∗ **by** (*simp add*: *omega-cont-def del*: *if-image-distrib*)
also have *f* (*SUP n*. *?C n*) = *f b*
using ‹*a* ≤ *b*› **by** (*auto simp add*: *gt-ex sup.absorb2 split*: *if-splits*)
finally show *f a* ≤ *f b* **.**
qed

lemma *omega-chain-iterates*: **fixes** *f* :: (*'a*::*complete-lattice*) ⇒ *'a*
assumes *mono f* **shows** *omega-chain*(λ*n*. (*f*^^*n*) *bot*)
proof−
have (*f* ^^ *n*) *bot* ≤ (*f* ^^ *Suc n*) *bot* **for** *n*

```
  proof (induction n)
    case 0 show ?case by simp
  next
    case (Suc n) thus ?case using assms by (auto simp: mono-def)
  qed
  thus ?thesis by(auto simp: omega-chain-def assms)
qed

theorem Kleene-lfp:
  assumes omega-cont f shows lfp f = (SUP n. (f^^n) bot) (is - = ?U)
proof(rule Orderings.antisym)
  from assms mono-if-omega-cont
  have mono: (f ^^ n) bot ≤ (f ^^ Suc n) bot for n
    using funpow-decreasing [of n Suc n] by auto
  show lfp f ≤ ?U
  proof (rule lfp-lowerbound)
    have f ?U = (SUP n. (f^^Suc n) bot)
      using omega-chain-iterates[OF mono-if-omega-cont[OF assms]] assms
      by(simp add: omega-cont-def)
    also have ... = ?U using mono by(blast intro: SUP-eq)
    finally show f ?U ≤ ?U by simp
  qed
next
  have (f^^n) bot ≤ p if f p ≤ p for n p
  proof −
    show ?thesis
    proof(induction n)
      case 0 show ?case by simp
    next
      case Suc
      from monoD[OF mono-if-omega-cont[OF assms] Suc] ‹f p ≤ p›
      show ?case by simp
    qed
  qed
  thus ?U ≤ lfp f
    using lfp-unfold[OF mono-if-omega-cont[OF assms]]
    by (simp add: SUP-le-iff)
qed
```

17.12 The divides relation on *nat*

```
lemma dvd-1-left [iff]: Suc 0 dvd k
  by (simp add: dvd-def)

lemma dvd-1-iff-1 [simp]: m dvd Suc 0 ⟷ m = Suc 0
  by (simp add: dvd-def)

lemma nat-dvd-1-iff-1 [simp]: m dvd 1 ⟷ m = 1
  for m :: nat
```

by (*simp add*: *dvd-def*)

lemma *dvd-antisym*: m *dvd* $n \Longrightarrow n$ *dvd* $m \Longrightarrow m = n$
for m n :: *nat*
unfolding *dvd-def* **by** (*force dest*: *mult-eq-self-implies-10 simp add*: *mult.assoc*)

lemma *dvd-diff-nat* [*simp*]: k *dvd* $m \Longrightarrow k$ *dvd* $n \Longrightarrow k$ *dvd* $(m - n)$
for k m n :: *nat*
unfolding *dvd-def* **by** (*blast intro*: *right-diff-distrib'* [*symmetric*])

lemma *dvd-diffD*:
fixes k m n :: *nat*
assumes k *dvd* $m - n$ k *dvd* n $n \leq m$
shows k *dvd* m
proof −
have k *dvd* $n + (m - n)$
using *assms* **by** (*blast intro*: *dvd-add*)
with *assms* **show** *?thesis*
by *simp*
qed

lemma *dvd-diffD1*: k *dvd* $m - n \Longrightarrow k$ *dvd* $m \Longrightarrow n \leq m \Longrightarrow k$ *dvd* n
for k m n :: *nat*
by (*drule-tac* $m = m$ **in** *dvd-diff-nat*) *auto*

lemma *dvd-mult-cancel*:
fixes m n k :: *nat*
assumes $k * m$ *dvd* $k * n$ **and** $0 < k$
shows m *dvd* n
proof −
from *assms*(*1*) **obtain** q **where** $k * n = (k * m) * q$ **..**
then have $k * n = k * (m * q)$ **by** (*simp add*: *ac-simps*)
with ‹$0 < k$› **have** $n = m * q$ **by** (*auto simp add*: *mult-left-cancel*)
then show *?thesis* **..**
qed

lemma *dvd-mult-cancel1*:
fixes m n :: *nat*
assumes $0 < m$
shows $m * n$ *dvd* $m \longleftrightarrow n = 1$
proof
assume $m * n$ *dvd* m
then have $m * n$ *dvd* $m * 1$
by *simp*
then have n *dvd* *1*
by (*iprover intro*: *assms dvd-mult-cancel*)
then show $n = 1$
by *auto*
qed *auto*

lemma *dvd-mult-cancel2*: $0 < m \Longrightarrow n * m \ dvd \ m \longleftrightarrow n = 1$
for $m \ n :: nat$
using *dvd-mult-cancel1* [*of m n*] **by** (*simp add*: *ac-simps*)

lemma *dvd-imp-le*: $k \ dvd \ n \Longrightarrow 0 < n \Longrightarrow k \leq n$
for $k \ n :: nat$
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *nat-dvd-not-less*: $0 < m \Longrightarrow m < n \Longrightarrow \neg \ n \ dvd \ m$
for $m \ n :: nat$
by (*auto elim*!: *dvdE*) (*auto simp add*: *gr0-conv-Suc*)

lemma *less-eq-dvd-minus*:
fixes $m \ n :: nat$
assumes $m \leq n$
shows $m \ dvd \ n \longleftrightarrow m \ dvd \ n - m$
proof −
from *assms* **have** $n = m + (n - m)$ **by** *simp*
then obtain q **where** $n = m + q$ **..**
then show *?thesis* **by** (*simp add*: *add.commute* [*of m*])
qed

lemma *dvd-minus-self*: $m \ dvd \ n - m \longleftrightarrow n < m \vee m \ dvd \ n$
for $m \ n :: nat$
by (*cases* $n < m$) (*auto elim*!: *dvdE simp add*: *not-less le-imp-diff-is-add dest*: *less-imp-le*)

lemma *dvd-minus-add*:
fixes $m \ n \ q \ r :: nat$
assumes $q \leq n \ q \leq r * m$
shows $m \ dvd \ n - q \longleftrightarrow m \ dvd \ n + (r * m - q)$
proof −
have $m \ dvd \ n - q \longleftrightarrow m \ dvd \ r * m + (n - q)$
using *dvd-add-times-triv-left-iff* [*of m r*] **by** *simp*
also from *assms* **have** $\ldots \longleftrightarrow m \ dvd \ r * m + n - q$ **by** *simp*
also from *assms* **have** $\ldots \longleftrightarrow m \ dvd \ (r * m - q) + n$ **by** *simp*
also have $\ldots \longleftrightarrow m \ dvd \ n + (r * m - q)$ **by** (*simp add*: *add.commute*)
finally show *?thesis* **.**
qed

17.13 Aliasses

lemma *nat-mult-1*: $1 * n = n$
for $n :: nat$
by (*fact mult-1-left*)

lemma *nat-mult-1-right*: $n * 1 = n$
for $n :: nat$

by (*fact mult-1-right*)

lemma *diff-mult-distrib*: $(m - n) * k = (m * k) - (n * k)$
for $k\ m\ n$ :: *nat*
by (*fact left-diff-distrib'*)

lemma *diff-mult-distrib2*: $k * (m - n) = (k * m) - (k * n)$
for $k\ m\ n$ :: *nat*
by (*fact right-diff-distrib'*)

lemma *le-diff-conv2*: $k \leq j \Longrightarrow (i \leq j - k) = (i + k \leq j)$
for $i\ j\ k$ :: *nat*
by (*fact le-diff-conv2*)

lemma *diff-self-eq-0* [*simp*]: $m - m = 0$
for m :: *nat*
by (*fact diff-cancel*)

lemma *diff-diff-left* [*simp*]: $i - j - k = i - (j + k)$
for $i\ j\ k$ :: *nat*
by (*fact diff-diff-add*)

lemma *diff-commute*: $i - j - k = i - k - j$
for $i\ j\ k$ :: *nat*
by (*fact diff-right-commute*)

lemma *diff-add-inverse*: $(n + m) - n = m$
for $m\ n$ :: *nat*
by (*fact add-diff-cancel-left'*)

lemma *diff-add-inverse2*: $(m + n) - n = m$
for $m\ n$ :: *nat*
by (*fact add-diff-cancel-right'*)

lemma *diff-cancel*: $(k + m) - (k + n) = m - n$
for $k\ m\ n$ :: *nat*
by (*fact add-diff-cancel-left*)

lemma *diff-cancel2*: $(m + k) - (n + k) = m - n$
for $k\ m\ n$ :: *nat*
by (*fact add-diff-cancel-right*)

lemma *diff-add-0*: $n - (n + m) = 0$
for $m\ n$ :: *nat*
by (*fact diff-add-zero*)

lemma *add-mult-distrib2*: $k * (m + n) = (k * m) + (k * n)$
for $k\ m\ n$ :: *nat*

by (*fact distrib-left*)

lemmas *nat-distrib* =
add-mult-distrib distrib-left diff-mult-distrib diff-mult-distrib2

17.14 Size of a datatype value

class *size* =
fixes *size* :: *'a* ⇒ *nat* — see further theory *Wellfounded*

instantiation *nat* :: *size*
begin

definition *size-nat* **where** [*simp*, *code*]: *size* (*n*::*nat*) = *n*

instance ..

end

lemmas *size-nat* = *size-nat-def*

lemma *size-neq-size-imp-neq*: *size x* ≠ *size y* ⟹ *x* ≠ *y*
by (*erule contrapos-nn*) (*rule arg-cong*)

17.15 Code module namespace

code-identifier
code-module *Nat* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

hide-const (**open**) *of-nat-aux*

end

18 Fields

theory *Fields*
imports *Nat*
begin

18.1 Division rings

A division ring is like a field, but without the commutativity requirement.

class *inverse* = *divide* +
fixes *inverse* :: *'a* ⇒ *'a*
begin

abbreviation *inverse-divide* :: *'a* ⇒ *'a* ⇒ *'a* (**infixl** ‹'/› *70*)
where

inverse-divide ≡ *divide*

end

Setup for linear arithmetic prover

ML-file ‹~~/*src/Provers/Arith/fast-lin-arith.ML*›
ML-file ‹*Tools/lin-arith.ML*›
setup ‹*Lin-Arith.global-setup*›
declaration ‹*K* (
Lin-Arith.init-arith-data
#> *Lin-Arith.add-discrete-type* ***type-name*** ‹*nat*›
#> *Lin-Arith.add-lessD* @{*thm Suc-leI*}
#> *Lin-Arith.add-simps* @{*thms simp-thms ring-distribs if-True if-False*
minus-diff-eq
add-0-left add-0-right order-less-irrefl
zero-neq-one zero-less-one zero-le-one
zero-neq-one [*THEN not-sym*] *not-one-le-zero not-one-less-zero*
add-Suc add-Suc-right nat.inject
Suc-le-mono Suc-less-eq Zero-not-Suc
Suc-not-Zero le-0-eq One-nat-def}
#> *Lin-Arith.add-simprocs* [***simproc*** ‹*group-cancel-add*›, ***simproc*** ‹*group-cancel-diff*›,
simproc ‹*group-cancel-eq*›, ***simproc*** ‹*group-cancel-le*›,
simproc ‹*group-cancel-less*›,
simproc ‹*nateq-cancel-sums*›,***simproc*** ‹*natless-cancel-sums*›,
simproc ‹*natle-cancel-sums*›])›

simproc-setup *fast-arith-nat* ((*m*::*nat*) < *n* | (*m*::*nat*) ≤ *n* | (*m*::*nat*) = *n*) =
‹*K Lin-Arith.simproc*› — Because of this simproc, the arithmetic solver is really only useful to detect inconsistencies among the premises for subgoals which are *not* themselves (in)equalities, because the latter activate *fast-nat-arith-simproc* anyway. However, it seems cheaper to activate the solver all the time rather than add the additional check.

lemmas [*linarith-split*] = *nat-diff-split split-min split-max abs-split*

Lemmas *divide-simps* move division to the outside and eliminates them on (in)equalities.

named-theorems *divide-simps rewrite rules to eliminate divisions*

class *division-ring* = *ring-1* + *inverse* +
assumes *left-inverse* [*simp*]: *a* ≠ *0* ⟹ *inverse a* ∗ *a* = *1*
assumes *right-inverse* [*simp*]: *a* ≠ *0* ⟹ *a* ∗ *inverse a* = *1*
assumes *divide-inverse*: *a* / *b* = *a* ∗ *inverse b*
assumes *inverse-zero* [*simp*]: *inverse 0* = *0*
begin

subclass *ring-1-no-zero-divisors*
proof
fix *a b* :: *'a*

```
  assume a: a ≠ 0 and b: b ≠ 0
  show a * b ≠ 0
  proof
    assume ab: a * b = 0
    hence 0 = inverse a * (a * b) * inverse b by simp
    also have ... = (inverse a * a) * (b * inverse b)
      by (simp only: mult.assoc)
    also have ... = 1 using a b by simp
    finally show False by simp
  qed
qed

lemma nonzero-imp-inverse-nonzero:
  a ≠ 0 ⟹ inverse a ≠ 0
proof
  assume ianz: inverse a = 0
  assume a ≠ 0
  hence 1 = a * inverse a by simp
  also have ... = 0 by (simp add: ianz)
  finally have 1 = 0 .
  thus False by (simp add: eq-commute)
qed

lemma inverse-zero-imp-zero:
  assumes inverse a = 0 shows a = 0
proof (rule ccontr)
  assume a ≠ 0
  then have inverse a ≠ 0
    by (simp add: nonzero-imp-inverse-nonzero)
  with assms show False
    by auto
qed

lemma inverse-unique:
  assumes ab: a * b = 1
  shows inverse a = b
proof −
  have a ≠ 0 using ab by (cases a = 0) simp-all
  moreover have inverse a * (a * b) = inverse a by (simp add: ab)
  ultimately show ?thesis by (simp add: mult.assoc [symmetric])
qed

lemma nonzero-inverse-minus-eq:
  a ≠ 0 ⟹ inverse (− a) = − inverse a
by (rule inverse-unique) simp

lemma nonzero-inverse-inverse-eq:
  a ≠ 0 ⟹ inverse (inverse a) = a
by (rule inverse-unique) simp
```

lemma *nonzero-inverse-eq-imp-eq*:
assumes *inverse a* = *inverse b* **and** *a* ≠ *0* **and** *b* ≠ *0*
shows *a* = *b*
proof −
from ‹*inverse a* = *inverse b*›
have *inverse* (*inverse a*) = *inverse* (*inverse b*) **by** (*rule arg-cong*)
with ‹*a* ≠ *0*› **and** ‹*b* ≠ *0*› **show** *a* = *b*
by (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-1* [*simp*]: *inverse 1* = *1*
by (*rule inverse-unique*) *simp*

subclass *divide-trivial*
by *standard* (*simp-all add*: *divide-inverse*)

lemma *nonzero-inverse-mult-distrib*:
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *inverse* (*a* ∗ *b*) = *inverse b* ∗ *inverse a*
proof −
have *a* ∗ (*b* ∗ *inverse b*) ∗ *inverse a* = *1* **using** *assms* **by** *simp*
hence *a* ∗ *b* ∗ (*inverse b* ∗ *inverse a*) = *1* **by** (*simp only*: *mult.assoc*)
thus *?thesis* **by** (*rule inverse-unique*)
qed

lemma *division-ring-inverse-add*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* + *inverse b* = *inverse a* ∗ (*a* + *b*) ∗ *inverse b*
by (*simp add*: *algebra-simps*)

lemma *division-ring-inverse-diff*:
a ≠ *0* ⟹ *b* ≠ *0* ⟹ *inverse a* − *inverse b* = *inverse a* ∗ (*b* − *a*) ∗ *inverse b*
by (*simp add*: *algebra-simps*)

lemma *right-inverse-eq*: *b* ≠ *0* ⟹ *a* / *b* = *1* ⟷ *a* = *b*
proof
assume *neq*: *b* ≠ *0*
{
hence *a* = (*a* / *b*) ∗ *b* **by** (*simp add*: *divide-inverse mult.assoc*)
also assume *a* / *b* = *1*
finally show *a* = *b* **by** *simp*
next
assume *a* = *b*
with *neq* **show** *a* / *b* = *1* **by** (*simp add*: *divide-inverse*)
}
qed

lemma *nonzero-inverse-eq-divide*: *a* ≠ *0* ⟹ *inverse a* = *1* / *a*
by (*simp add*: *divide-inverse*)

lemma *divide-self* [*simp*]: $a \neq 0 \Longrightarrow a \mathbin{/} a = 1$
by (*simp add*: *divide-inverse*)

lemma *inverse-eq-divide* [*field-simps*, *field-split-simps*, *divide-simps*]: *inverse* $a = 1 \mathbin{/} a$
by (*simp add*: *divide-inverse*)

lemma *add-divide-distrib*: $(a+b) \mathbin{/} c = a/c + b/c$
by (*simp add*: *divide-inverse algebra-simps*)

lemma *times-divide-eq-right* [*simp*]: $a * (b \mathbin{/} c) = (a * b) \mathbin{/} c$
by (*simp add*: *divide-inverse mult.assoc*)

lemma *minus-divide-left*: $- (a \mathbin{/} b) = (-a) \mathbin{/} b$
by (*simp add*: *divide-inverse*)

lemma *nonzero-minus-divide-right*: $b \neq 0 \Longrightarrow - (a \mathbin{/} b) = a \mathbin{/} (- b)$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *nonzero-minus-divide-divide*: $b \neq 0 \Longrightarrow (-a) \mathbin{/} (-b) = a \mathbin{/} b$
by (*simp add*: *divide-inverse nonzero-inverse-minus-eq*)

lemma *divide-minus-left* [*simp*]: $(-a) \mathbin{/} b = - (a \mathbin{/} b)$
by (*simp add*: *divide-inverse*)

lemma *diff-divide-distrib*: $(a - b) \mathbin{/} c = a \mathbin{/} c - b \mathbin{/} c$
using *add-divide-distrib* [*of* $a - b$ c] **by** *simp*

lemma *nonzero-eq-divide-eq* [*field-simps*]: $c \neq 0 \Longrightarrow a = b \mathbin{/} c \longleftrightarrow a * c = b$
proof −
assume [*simp*]: $c \neq 0$
have $a = b \mathbin{/} c \longleftrightarrow a * c = (b \mathbin{/} c) * c$ **by** *simp*
also have ... $\longleftrightarrow a * c = b$ **by** (*simp add*: *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *nonzero-divide-eq-eq* [*field-simps*]: $c \neq 0 \Longrightarrow b \mathbin{/} c = a \longleftrightarrow b = a * c$
proof −
assume [*simp*]: $c \neq 0$
have $b \mathbin{/} c = a \longleftrightarrow (b \mathbin{/} c) * c = a * c$ **by** *simp*
also have ... $\longleftrightarrow b = a * c$ **by** (*simp add*: *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *nonzero-neg-divide-eq-eq* [*field-simps*]: $b \neq 0 \Longrightarrow - (a \mathbin{/} b) = c \longleftrightarrow - a = c * b$
using *nonzero-divide-eq-eq*[*of* b $-a$ c] **by** *simp*

lemma *nonzero-neg-divide-eq-eq2* [*field-simps*]: $b \neq 0 \Longrightarrow c = - (a \mathbin{/} b) \longleftrightarrow c * b = - a$
using *nonzero-neg-divide-eq-eq*[*of b a c*] **by** *auto*

lemma *divide-eq-imp*: $c \neq 0 \Longrightarrow b = a * c \Longrightarrow b \mathbin{/} c = a$
by (*simp add*: *divide-inverse mult.assoc*)

lemma *eq-divide-imp*: $c \neq 0 \Longrightarrow a * c = b \Longrightarrow a = b \mathbin{/} c$
by (*drule sym*) (*simp add*: *divide-inverse mult.assoc*)

lemma *add-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x + y \mathbin{/} z = (x * z + y) \mathbin{/} z$
by (*simp add*: *add-divide-distrib nonzero-eq-divide-eq*)

lemma *divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z + y = (x + y * z) \mathbin{/} z$
by (*simp add*: *add-divide-distrib nonzero-eq-divide-eq*)

lemma *diff-divide-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x - y \mathbin{/} z = (x * z - y) \mathbin{/} z$
by (*simp add*: *diff-divide-distrib nonzero-eq-divide-eq eq-diff-eq*)

lemma *minus-divide-add-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow - (x \mathbin{/} z) + y = (- x + y * z) \mathbin{/} z$
by (*simp add*: *add-divide-distrib diff-divide-eq-iff*)

lemma *divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow x \mathbin{/} z - y = (x - y * z) \mathbin{/} z$
by (*simp add*: *field-simps*)

lemma *minus-divide-diff-eq-iff* [*field-simps*]:
$z \neq 0 \Longrightarrow - (x \mathbin{/} z) - y = (- x - y * z) \mathbin{/} z$
by (*simp add*: *divide-diff-eq-iff*[*symmetric*])

lemma *division-ring-divide-zero*:
$a \mathbin{/} 0 = 0$
by (*fact div-by-0*)

lemma *divide-self-if* [*simp*]:
$a \mathbin{/} a = (\textit{if } a = 0 \textit{ then } 0 \textit{ else } 1)$
by *simp*

lemma *inverse-nonzero-iff-nonzero* [*simp*]:
$\textit{inverse } a = 0 \longleftrightarrow a = 0$
by (*rule iffI*) (*fact inverse-zero-imp-zero*, *simp*)

lemma *inverse-minus-eq* [*simp*]:
$\textit{inverse } (- a) = - \textit{ inverse } a$
proof *cases*

assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *a≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *inverse-inverse-eq* [*simp*]:
inverse (*inverse a*) = *a*
proof *cases*
assume *a=0* **thus** *?thesis* **by** *simp*
next
assume *a≠0*
thus *?thesis* **by** (*simp add*: *nonzero-inverse-inverse-eq*)
qed

lemma *inverse-eq-imp-eq*:
inverse a = *inverse b* ⟹ *a* = *b*
by (*drule arg-cong* [**where** *f*=*inverse*], *simp*)

lemma *inverse-eq-iff-eq* [*simp*]:
inverse a = *inverse b* ⟷ *a* = *b*
by (*force dest*!: *inverse-eq-imp-eq*)

lemma *mult-commute-imp-mult-inverse-commute*:
assumes *y* ∗ *x* = *x* ∗ *y*
shows *inverse y* ∗ *x* = *x* ∗ *inverse y*
proof (*cases y=0*)
case *False*
hence *x* ∗ *inverse y* = *inverse y* ∗ *y* ∗ *x* ∗ *inverse y*
by *simp*
also have ... = *inverse y* ∗ (*x* ∗ *y* ∗ *inverse y*)
by (*simp add*: *mult.assoc assms*)
finally show *?thesis* **by** (*simp add*: *mult.assoc False*)
qed *simp*

lemmas *mult-inverse-of-nat-commute* =
mult-commute-imp-mult-inverse-commute[*OF mult-of-nat-commute*]

lemma *divide-divide-eq-left'*:
(*a* / *b*) / *c* = *a* / (*c* ∗ *b*)
by (*cases b* = *0* ∨ *c* = *0*)
(*auto simp*: *divide-inverse mult.assoc nonzero-inverse-mult-distrib*)

lemma *add-divide-eq-if-simps* [*field-split-simps*, *divide-simps*]:
a + *b* / *z* = (*if z* = *0 then a else* (*a* ∗ *z* + *b*) / *z*)
a / *z* + *b* = (*if z* = *0 then b else* (*a* + *b* ∗ *z*) / *z*)
− (*a* / *z*) + *b* = (*if z* = *0 then b else* (−*a* + *b* ∗ *z*) / *z*)
a − *b* / *z* = (*if z* = *0 then a else* (*a* ∗ *z* − *b*) / *z*)
a / *z* − *b* = (*if z* = *0 then* −*b else* (*a* − *b* ∗ *z*) / *z*)

$- (a \mathbin{/} z) - b = (\mathit{if}\ z = 0\ \mathit{then}\ -b\ \mathit{else}\ (- a - b * z) \mathbin{/} z)$
by (*simp-all add*: *add-divide-eq-iff divide-add-eq-iff diff-divide-eq-iff divide-diff-eq-iff minus-divide-diff-eq-iff*)

lemma [*field-split-simps*, *divide-simps*]:
shows *divide-eq-eq*: $b \mathbin{/} c = a \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then}\ b = a * c\ \mathit{else}\ a = 0)$
and *eq-divide-eq*: $a = b \mathbin{/} c \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then}\ a * c = b\ \mathit{else}\ a = 0)$
and *minus-divide-eq-eq*: $- (b \mathbin{/} c) = a \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then} - b = a * c\ \mathit{else}\ a = 0)$
and *eq-minus-divide-eq*: $a = - (b \mathbin{/} c) \longleftrightarrow (\mathit{if}\ c \neq 0\ \mathit{then}\ a * c = - b\ \mathit{else}\ a = 0)$
by (*auto simp add*: *field-simps*)

end

18.2 Fields

class *field* = *comm-ring-1* + *inverse* +
assumes *field-inverse*: $a \neq 0 \Longrightarrow \mathit{inverse}\ a * a = 1$
assumes *field-divide-inverse*: $a \mathbin{/} b = a * \mathit{inverse}\ b$
assumes *field-inverse-zero*: $\mathit{inverse}\ 0 = 0$
begin

subclass *division-ring*
proof
fix $a :: 'a$
assume $a \neq 0$
thus $\mathit{inverse}\ a * a = 1$ **by** (*rule field-inverse*)
thus $a * \mathit{inverse}\ a = 1$ **by** (*simp only*: *mult.commute*)
next
fix $a\ b :: 'a$
show $a \mathbin{/} b = a * \mathit{inverse}\ b$ **by** (*rule field-divide-inverse*)
next
show $\mathit{inverse}\ 0 = 0$
by (*fact field-inverse-zero*)
qed

subclass *idom-divide*
proof
fix $b\ a$
assume $b \neq 0$
then show $a * b \mathbin{/} b = a$
by (*simp add*: *divide-inverse ac-simps*)
next
fix a
show $a \mathbin{/} 0 = 0$
by (*simp add*: *divide-inverse*)
qed

There is no slick version using division by zero.

lemma *inverse-add*:
  $a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ a + inverse\ b = (a + b) * inverse\ a * inverse\ b$
  by (*simp add*: *division-ring-inverse-add ac-simps*)

lemma *nonzero-mult-divide-mult-cancel-left* [*simp*]:
  assumes [*simp*]: $c \neq 0$
  shows $(c * a) / (c * b) = a / b$
proof (*cases* $b = 0$)
  case *True* **then show** *?thesis* **by** *simp*
next
  case *False*
  then have $(c*a)/(c*b) = c * a * (inverse\ b * inverse\ c)$
    by (*simp add*: *divide-inverse nonzero-inverse-mult-distrib*)
  also have ... = $a * inverse\ b * (inverse\ c * c)$
    by (*simp only*: *ac-simps*)
  also have ... = $a * inverse\ b$ **by** *simp*
    finally show *?thesis* **by** (*simp add*: *divide-inverse*)
qed

lemma *nonzero-mult-divide-mult-cancel-right* [*simp*]:
  $c \neq 0 \Longrightarrow (a * c) / (b * c) = a / b$
  using *nonzero-mult-divide-mult-cancel-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *times-divide-eq-left* [*simp*]: $(b / c) * a = (b * a) / c$
  by (*simp add*: *divide-inverse ac-simps*)

lemma *divide-inverse-commute*: $a / b = inverse\ b * a$
  by (*simp add*: *divide-inverse mult.commute*)

lemma *add-frac-eq*:
  assumes $y \neq 0$ **and** $z \neq 0$
  shows $x / y + w / z = (x * z + w * y) / (y * z)$
proof −
  have $x / y + w / z = (x * z) / (y * z) + (y * w) / (y * z)$
    using *assms* **by** *simp*
  also have ... $= (x * z + y * w) / (y * z)$
    by (*simp only*: *add-divide-distrib*)
  finally show *?thesis*
    by (*simp only*: *mult.commute*)
qed

Special Cancellation Simprules for Division

lemma *nonzero-divide-mult-cancel-right* [*simp*]:
  $b \neq 0 \Longrightarrow b / (a * b) = 1 / a$
  using *nonzero-mult-divide-mult-cancel-right* [*of b 1 a*] **by** *simp*

lemma *nonzero-divide-mult-cancel-left* [*simp*]:
  $a \neq 0 \Longrightarrow a / (a * b) = 1 / b$
  using *nonzero-mult-divide-mult-cancel-left* [*of a 1 b*] **by** *simp*

lemma *nonzero-mult-divide-mult-cancel-left2* [*simp*]:
$c \neq 0 \Longrightarrow (c * a) \mathbin{/} (b * c) = a \mathbin{/} b$
using *nonzero-mult-divide-mult-cancel-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *nonzero-mult-divide-mult-cancel-right2* [*simp*]:
$c \neq 0 \Longrightarrow (a * c) \mathbin{/} (c * b) = a \mathbin{/} b$
using *nonzero-mult-divide-mult-cancel-right* [*of b c a*] **by** (*simp add*: *ac-simps*)

lemma *diff-frac-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y - w \mathbin{/} z = (x * z - w * y) \mathbin{/} (y * z)$
by (*simp add*: *field-simps*)

lemma *frac-eq-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow (x \mathbin{/} y = w \mathbin{/} z) = (x * z = w * y)$
by (*simp add*: *field-simps*)

lemma *divide-minus1* [*simp*]: $x \mathbin{/} -1 = -x$
using *nonzero-minus-divide-right* [*of 1 x*] **by** *simp*

This version builds in division by zero while also re-orienting the right-hand side.

lemma *inverse-mult-distrib* [*simp*]:
inverse $(a * b) =$ *inverse* $a *$ *inverse* b
proof *cases*
assume $a \neq 0 \wedge b \neq 0$
thus *?thesis* **by** (*simp add*: *nonzero-inverse-mult-distrib ac-simps*)
next
assume $\neg (a \neq 0 \wedge b \neq 0)$
thus *?thesis* **by** *force*
qed

lemma *inverse-divide* [*simp*]:
inverse $(a \mathbin{/} b) = b \mathbin{/} a$
by (*simp add*: *divide-inverse mult.commute*)

Calculations with fractions

There is a whole bunch of simp-rules just for class *field* but none for class *field* and *nonzero-divides* because the latter are covered by a simproc.

lemmas *mult-divide-mult-cancel-left* = *nonzero-mult-divide-mult-cancel-left*

lemmas *mult-divide-mult-cancel-right* = *nonzero-mult-divide-mult-cancel-right*

lemma *divide-divide-eq-right* [*simp*]:
$a \mathbin{/} (b \mathbin{/} c) = (a * c) \mathbin{/} b$
by (*simp add*: *divide-inverse ac-simps*)

lemma *divide-divide-eq-left* [*simp*]:

$(a\ /\ b)\ /\ c = a\ /\ (b * c)$
by (*simp add*: *divide-inverse mult.assoc*)

lemma *divide-divide-times-eq*:
$(x\ /\ y)\ /\ (z\ /\ w) = (x * w)\ /\ (y * z)$
by *simp*

Special Cancellation Simprules for Division

lemma *mult-divide-mult-cancel-left-if* [*simp*]:
shows $(c * a)\ /\ (c * b) = (\textit{if } c = 0 \textit{ then } 0 \textit{ else } a\ /\ b)$
by *simp*

Division and Unary Minus

lemma *minus-divide-right*:
$-\ (a\ /\ b) = a\ /\ -\ b$
by (*simp add*: *divide-inverse*)

lemma *divide-minus-right* [*simp*]:
$a\ /\ -\ b = -\ (a\ /\ b)$
by (*simp add*: *divide-inverse*)

lemma *minus-divide-divide*:
$(-\ a)\ /\ (-\ b) = a\ /\ b$
by (*cases b=0*) (*simp-all add*: *nonzero-minus-divide-divide*)

lemma *inverse-eq-1-iff* [*simp*]:
$\textit{inverse } x = 1 \longleftrightarrow x = 1$
using *inverse-eq-iff-eq* [*of x 1*] **by** *simp*

lemma *divide-eq-0-iff* [*simp*]:
$a\ /\ b = 0 \longleftrightarrow a = 0 \vee b = 0$
by (*simp add*: *divide-inverse*)

lemma *divide-cancel-right* [*simp*]:
$a\ /\ c = b\ /\ c \longleftrightarrow c = 0 \vee a = b$
by (*cases c=0*) (*simp-all add*: *divide-inverse*)

lemma *divide-cancel-left* [*simp*]:
$c\ /\ a = c\ /\ b \longleftrightarrow c = 0 \vee a = b$
by (*cases c=0*) (*simp-all add*: *divide-inverse*)

lemma *divide-eq-1-iff* [*simp*]:
$a\ /\ b = 1 \longleftrightarrow b \neq 0 \wedge a = b$
by (*cases b=0*) (*simp-all add*: *right-inverse-eq*)

lemma *one-eq-divide-iff* [*simp*]:
$1 = a\ /\ b \longleftrightarrow b \neq 0 \wedge a = b$
by (*simp add*: *eq-commute* [*of 1*])

lemma *divide-eq-minus-1-iff*:
  $(a \mathbin{/} b = -1) \longleftrightarrow b \neq 0 \wedge a = -b$
using *divide-eq-1-iff* **by** *fastforce*

lemma *times-divide-times-eq*:
  $(x \mathbin{/} y) * (z \mathbin{/} w) = (x * z) \mathbin{/} (y * w)$
  by *simp*

lemma *add-frac-num*:
  $y \neq 0 \Longrightarrow x \mathbin{/} y + z = (x + z * y) \mathbin{/} y$
  by (*simp add*: *add-divide-distrib*)

lemma *add-num-frac*:
  $y \neq 0 \Longrightarrow z + x \mathbin{/} y = (x + z * y) \mathbin{/} y$
  by (*simp add*: *add-divide-distrib add.commute*)

lemma *dvd-field-iff*:
  a dvd b $\longleftrightarrow$ $(a = 0 \longrightarrow b = 0)$
proof (*cases a* $= 0$)
  case *False*
  then have $b = a * (b \mathbin{/} a)$
    by (*simp add*: *field-simps*)
  then have *a dvd b* **..**
  with *False* **show** *?thesis*
    by *simp*
qed *simp*

lemma *inj-divide-right* [*simp*]:
  inj $(\lambda b.\ b \mathbin{/} a) \longleftrightarrow a \neq 0$
proof −
  have $(\lambda b.\ b \mathbin{/} a) = (*)\ (inverse\ a)$
    by (*simp add*: *field-simps fun-eq-iff*)
  then have *inj* $(\lambda y.\ y \mathbin{/} a) \longleftrightarrow$ *inj* $((*)\ (inverse\ a))$
    by *simp*
  also have $\ldots \longleftrightarrow$ *inverse a* $\neq 0$
    by *simp*
  also have $\ldots \longleftrightarrow a \neq 0$
    by *simp*
  finally show *?thesis*
    by *simp*
qed

end

class *field-char-0* = *field* + *ring-char-0*

18.3 Ordered fields

class *field-abs-sgn* = *field* + *idom-abs-sgn*

begin

lemma *sgn-inverse* [*simp*]:
  sgn (*inverse a*) = *inverse* (*sgn a*)
proof (*cases a* = *0*)
  case *True* **then show** *?thesis* **by** *simp*
next
  case *False*
  then have *a* ∗ *inverse a* = *1*
    by *simp*
  then have *sgn* (*a* ∗ *inverse a*) = *sgn 1*
    by *simp*
  then have *sgn a* ∗ *sgn* (*inverse a*) = *1*
    by (*simp add*: *sgn-mult*)
  then have *inverse* (*sgn a*) ∗ (*sgn a* ∗ *sgn* (*inverse a*)) = *inverse* (*sgn a*) ∗ *1*
    by *simp*
  then have (*inverse* (*sgn a*) ∗ *sgn a*) ∗ *sgn* (*inverse a*) = *inverse* (*sgn a*)
    by (*simp add*: *ac-simps*)
  with *False* **show** *?thesis*
    by (*simp add*: *sgn-eq-0-iff*)
qed

lemma *abs-inverse* [*simp*]:
  |*inverse a*| = *inverse* |*a*|
proof −
  from *sgn-mult-abs* [*of inverse a*] *sgn-mult-abs* [*of a*]
  have *inverse* (*sgn a*) ∗ |*inverse a*| = *inverse* (*sgn a* ∗ |*a*|)
    by *simp*
  then show *?thesis* **by** (*auto simp add*: *sgn-eq-0-iff*)
qed

lemma *sgn-divide* [*simp*]:
  sgn (*a* / *b*) = *sgn a* / *sgn b*
  unfolding *divide-inverse sgn-mult* **by** *simp*

lemma *abs-divide* [*simp*]:
  |*a* / *b*| = |*a*| / |*b*|
  unfolding *divide-inverse abs-mult* **by** *simp*

end

class *linordered-field* = *field* + *linordered-idom*
begin

lemma *positive-imp-inverse-positive*:
  assumes *a-gt-0*: *0* < *a*
  shows *0* < *inverse a*
proof −
  have *0* < *a* ∗ *inverse a*

```
    by (simp add: a-gt-0 [THEN less-imp-not-eq2])
  thus 0 < inverse a
    by (simp add: a-gt-0 [THEN less-not-sym] zero-less-mult-iff)
qed

lemma negative-imp-inverse-negative:
  a < 0 ⟹ inverse a < 0
  using positive-imp-inverse-positive [of −a]
  by (simp add: nonzero-inverse-minus-eq less-imp-not-eq)

lemma inverse-le-imp-le:
  assumes invle: inverse a ≤ inverse b and apos: 0 < a
  shows b ≤ a
proof (rule classical)
  assume ¬ b ≤ a
  hence a < b  by (simp add: linorder-not-le)
  hence bpos: 0 < b  by (blast intro: apos less-trans)
  hence a * inverse a ≤ a * inverse b
    by (simp add: apos invle less-imp-le mult-left-mono)
  hence (a * inverse a) * b ≤ (a * inverse b) * b
    by (simp add: bpos less-imp-le mult-right-mono)
  thus b ≤ a  by (simp add: mult.assoc apos bpos less-imp-not-eq2)
qed

lemma inverse-positive-imp-positive:
  assumes inv-gt-0: 0 < inverse a and nz: a ≠ 0
  shows 0 < a
proof −
  have 0 < inverse (inverse a)
    using inv-gt-0 by (rule positive-imp-inverse-positive)
  thus 0 < a
    using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma inverse-negative-imp-negative:
  assumes inv-less-0: inverse a < 0 and nz: a ≠ 0
  shows a < 0
proof −
  have inverse (inverse a) < 0
    using inv-less-0 by (rule negative-imp-inverse-negative)
  thus a < 0 using nz by (simp add: nonzero-inverse-inverse-eq)
qed

lemma linordered-field-no-lb:
  ∀ x. ∃ y. y < x
proof
  fix x::'a
  have m1: − (1::'a) < 0 by simp
  from add-strict-right-mono[OF m1, where c=x]
```

have $(-1) + x < x$ **by** *simp*
thus $\exists\, y.\ y < x$ **by** *blast*
qed

lemma *linordered-field-no-ub*:
$\forall\ x.\ \exists\, y.\ y > x$
proof
fix $x::'a$
have *m1*: $(1::'a) > 0$ **by** *simp*
from *add-strict-right-mono*[*OF m1*, **where** $c{=}x$]
have $1 + x > x$ **by** *simp*
thus $\exists\, y.\ y > x$ **by** *blast*
qed

lemma *less-imp-inverse-less*:
assumes *less*: $a < b$ **and** *apos*: $0 < a$
shows $inverse\ b < inverse\ a$
proof (*rule ccontr*)
assume $\neg\ inverse\ b < inverse\ a$
hence $inverse\ a \leq inverse\ b$ **by** *simp*
hence $\neg\ (a < b)$
by (*simp add*: *not-less inverse-le-imp-le* [*OF - apos*])
thus *False* **by** (*rule notE* [*OF - less*])
qed

lemma *inverse-less-imp-less*:
assumes $inverse\ a < inverse\ b\ 0 < a$
shows $b < a$
proof −
have $a \neq b$
using *assms* **by** (*simp add*: *less-le*)
moreover have $b \leq a$
using *assms* **by** (*force simp*: *less-le dest*: *inverse-le-imp-le*)
ultimately show *?thesis*
by (*simp add*: *less-le*)
qed

Both premises are essential. Consider -1 and 1.

lemma *inverse-less-iff-less* [*simp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a < inverse\ b \longleftrightarrow b < a$
by (*blast intro*: *less-imp-inverse-less dest*: *inverse-less-imp-less*)

lemma *le-imp-inverse-le*:
$a \leq b \Longrightarrow 0 < a \Longrightarrow inverse\ b \leq inverse\ a$
by (*force simp add*: *le-less less-imp-inverse-less*)

lemma *inverse-le-iff-le* [*simp*]:
$0 < a \Longrightarrow 0 < b \Longrightarrow inverse\ a \leq inverse\ b \longleftrightarrow b \leq a$
by (*blast intro*: *le-imp-inverse-le dest*: *inverse-le-imp-le*)

These results refer to both operands being negative. The opposite-sign case is trivial, since inverse preserves signs.

lemma *inverse-le-imp-le-neg*:
assumes *inverse a* ≤ *inverse b b* < *0*
shows *b* ≤ *a*
proof (*rule classical*)
assume ¬ *b* ≤ *a*
with ‹*b* < *0*› **have** *a* < *0*
by *force*
with *assms* **show** *b* ≤ *a*
using *inverse-le-imp-le* [*of* −*b* −*a*] **by** (*simp add*: *nonzero-inverse-minus-eq*)
qed

lemma *less-imp-inverse-less-neg*:
assumes *a* < *b b* < *0*
shows *inverse b* < *inverse a*
proof −
have *a* < *0*
using *assms* **by** (*blast intro*: *less-trans*)
with *less-imp-inverse-less* [*of* −*b* −*a*] **show** *?thesis*
by (*simp add*: *nonzero-inverse-minus-eq assms*)
qed

lemma *inverse-less-imp-less-neg*:
assumes *inverse a* < *inverse b b* < *0*
shows *b* < *a*
proof (*rule classical*)
assume ¬ *b* < *a*
with ‹*b* < *0*› **have** *a* < *0*
by *force*
with *inverse-less-imp-less* [*of* −*b* −*a*] **show** *?thesis*
by (*simp add*: *nonzero-inverse-minus-eq assms*)
qed

lemma *inverse-less-iff-less-neg* [*simp*]:
a < *0* ⟹ *b* < *0* ⟹ *inverse a* < *inverse b* ⟷ *b* < *a*
using *inverse-less-iff-less* [*of* −*b* −*a*]
by (*simp del*: *inverse-less-iff-less add*: *nonzero-inverse-minus-eq*)

lemma *le-imp-inverse-le-neg*:
a ≤ *b* ⟹ *b* < *0* ⟹ *inverse b* ≤ *inverse a*
by (*force simp add*: *le-less less-imp-inverse-less-neg*)

lemma *inverse-le-iff-le-neg* [*simp*]:
a < *0* ⟹ *b* < *0* ⟹ *inverse a* ≤ *inverse b* ⟷ *b* ≤ *a*
by (*blast intro*: *le-imp-inverse-le-neg dest*: *inverse-le-imp-le-neg*)

lemma *one-less-inverse*:
0 < *a* ⟹ *a* < *1* ⟹ *1* < *inverse a*

using *less-imp-inverse-less* [*of a 1*, *unfolded inverse-1*] **.**

lemma *one-le-inverse*:
0 < a $\Longrightarrow$ *a* $\leq$ *1* $\Longrightarrow$ *1* $\leq$ *inverse a*
using *le-imp-inverse-le* [*of a 1*, *unfolded inverse-1*] **.**

lemma *pos-le-divide-eq* [*field-simps*]:
assumes *0 < c*
shows *a* $\leq$ *b / c* $\longleftrightarrow$ *a * c* $\leq$ *b*
proof −
from *assms* **have** *a* $\leq$ *b / c* $\longleftrightarrow$ *a * c* $\leq$ (*b / c*) ** c*
using *mult-le-cancel-right* [*of a c b * inverse c*] **by** (*auto simp add*: *field-simps*)
also have ... $\longleftrightarrow$ *a * c* $\leq$ *b*
by (*simp add*: *less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *pos-less-divide-eq* [*field-simps*]:
assumes *0 < c*
shows *a < b / c* $\longleftrightarrow$ *a * c < b*
proof −
from *assms* **have** *a < b / c* $\longleftrightarrow$ *a * c <* (*b / c*) ** c*
using *mult-less-cancel-right* [*of a c b / c*] **by** *auto*
also have ... = (*a*c < b*)
by (*simp add*: *less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *neg-less-divide-eq* [*field-simps*]:
assumes *c < 0*
shows *a < b / c* $\longleftrightarrow$ *b < a * c*
proof −
from *assms* **have** *a < b / c* $\longleftrightarrow$ (*b / c*) ** c < a * c*
using *mult-less-cancel-right* [*of b / c c a*] **by** *auto*
also have ... $\longleftrightarrow$ *b < a * c*
by (*simp add*: *less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *neg-le-divide-eq* [*field-simps*]:
assumes *c < 0*
shows *a* $\leq$ *b / c* $\longleftrightarrow$ *b* $\leq$ *a * c*
proof −
from *assms* **have** *a* $\leq$ *b / c* $\longleftrightarrow$ (*b / c*) ** c* $\leq$ *a * c*
using *mult-le-cancel-right* [*of b * inverse c c a*] **by** (*auto simp add*: *field-simps*)
also have ... $\longleftrightarrow$ *b* $\leq$ *a * c*
by (*simp add*: *less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *pos-divide-le-eq* [*field-simps*]:
assumes $0 < c$
shows $b \mathbin{/} c \leq a \longleftrightarrow b \leq a * c$
proof −
from *assms* **have** $b \mathbin{/} c \leq a \longleftrightarrow (b \mathbin{/} c) * c \leq a * c$
using *mult-le-cancel-right* [*of* $b \mathbin{/} c$ c a] **by** *auto*
also have ... $\longleftrightarrow b \leq a * c$
by (*simp add*: *less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *pos-divide-less-eq* [*field-simps*]:
assumes $0 < c$
shows $b \mathbin{/} c < a \longleftrightarrow b < a * c$
proof −
from *assms* **have** $b \mathbin{/} c < a \longleftrightarrow (b \mathbin{/} c) * c < a * c$
using *mult-less-cancel-right* [*of* $b \mathbin{/} c$ c a] **by** *auto*
also have ... $\longleftrightarrow b < a * c$
by (*simp add*: *less-imp-not-eq2* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-le-eq* [*field-simps*]:
assumes $c < 0$
shows $b \mathbin{/} c \leq a \longleftrightarrow a * c \leq b$
proof −
from *assms* **have** $b \mathbin{/} c \leq a \longleftrightarrow a * c \leq (b \mathbin{/} c) * c$
using *mult-le-cancel-right* [*of* a c $b \mathbin{/} c$] **by** *auto*
also have ... $\longleftrightarrow a * c \leq b$
by (*simp add*: *less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

lemma *neg-divide-less-eq* [*field-simps*]:
assumes $c < 0$
shows $b \mathbin{/} c < a \longleftrightarrow a * c < b$
proof −
from *assms* **have** $b \mathbin{/} c < a \longleftrightarrow a * c < b \mathbin{/} c * c$
using *mult-less-cancel-right* [*of* a c $b \mathbin{/} c$] **by** *auto*
also have ... $\longleftrightarrow a * c < b$
by (*simp add*: *less-imp-not-eq* [*OF assms*] *divide-inverse mult.assoc*)
finally show *?thesis* **.**
qed

The following *field-simps* rules are necessary, as minus is always moved atop of division but we want to get rid of division.

lemma *pos-le-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a \leq - (b \mathbin{/} c) \longleftrightarrow a * c \leq - b$

unfolding *minus-divide-left* **by** (*rule pos-le-divide-eq*)

lemma *neg-le-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a \leq - (b \mathbin{/} c) \longleftrightarrow - b \leq a * c$
unfolding *minus-divide-left* **by** (*rule neg-le-divide-eq*)

lemma *pos-less-minus-divide-eq* [*field-simps*]: $0 < c \Longrightarrow a < - (b \mathbin{/} c) \longleftrightarrow a * c < - b$
unfolding *minus-divide-left* **by** (*rule pos-less-divide-eq*)

lemma *neg-less-minus-divide-eq* [*field-simps*]: $c < 0 \Longrightarrow a < - (b \mathbin{/} c) \longleftrightarrow - b < a * c$
unfolding *minus-divide-left* **by** (*rule neg-less-divide-eq*)

lemma *pos-minus-divide-less-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \mathbin{/} c) < a \longleftrightarrow - b < a * c$
unfolding *minus-divide-left* **by** (*rule pos-divide-less-eq*)

lemma *neg-minus-divide-less-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \mathbin{/} c) < a \longleftrightarrow a * c < - b$
unfolding *minus-divide-left* **by** (*rule neg-divide-less-eq*)

lemma *pos-minus-divide-le-eq* [*field-simps*]: $0 < c \Longrightarrow - (b \mathbin{/} c) \leq a \longleftrightarrow - b \leq a * c$
unfolding *minus-divide-left* **by** (*rule pos-divide-le-eq*)

lemma *neg-minus-divide-le-eq* [*field-simps*]: $c < 0 \Longrightarrow - (b \mathbin{/} c) \leq a \longleftrightarrow a * c \leq - b$
unfolding *minus-divide-left* **by** (*rule neg-divide-le-eq*)

lemma *frac-less-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y < w \mathbin{/} z \longleftrightarrow (x * z - w * y) \mathbin{/} (y * z) < 0$
by (*subst less-iff-diff-less-0*) (*simp add*: *diff-frac-eq*)

lemma *frac-le-eq*:
$y \neq 0 \Longrightarrow z \neq 0 \Longrightarrow x \mathbin{/} y \leq w \mathbin{/} z \longleftrightarrow (x * z - w * y) \mathbin{/} (y * z) \leq 0$
by (*subst le-iff-diff-le-0*) (*simp add*: *diff-frac-eq*)

lemma *divide-pos-pos*[*simp*]:
$0 < x \Longrightarrow 0 < y \Longrightarrow 0 < x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-pos*:
$0 \leq x \Longrightarrow 0 < y \Longrightarrow 0 \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-neg-pos*:
$x < 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-pos*:
$x \leq 0 \Longrightarrow 0 < y \Longrightarrow x \mathbin{/} y \leq 0$
by(*simp add*:*field-simps*)

lemma *divide-pos-neg*:
$0 < x \Longrightarrow y < 0 \Longrightarrow x \mathbin{/} y < 0$
by(*simp add*:*field-simps*)

lemma *divide-nonneg-neg*:
$0 \leq x \Longrightarrow y < 0 \Longrightarrow x \mathbin{/} y \leq 0$
by(*simp add*:*field-simps*)

lemma *divide-neg-neg*:
$x < 0 \Longrightarrow y < 0 \Longrightarrow 0 < x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-nonpos-neg*:
$x \leq 0 \Longrightarrow y < 0 \Longrightarrow 0 \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *divide-strict-right-mono*:
$[\![a < b;\ 0 < c]\!] \Longrightarrow a \mathbin{/} c < b \mathbin{/} c$
by (*simp add*: *less-imp-not-eq2 divide-inverse mult-strict-right-mono positive-imp-inverse-positive*)

lemma *divide-strict-right-mono-neg*:
assumes $b < a$ $c < 0$ **shows** $a \mathbin{/} c < b \mathbin{/} c$
proof −
have $b \mathbin{/} - c < a \mathbin{/} - c$
by (*rule divide-strict-right-mono*) (*use assms* **in** *auto*)
then show *?thesis*
by (*simp add*: *less-imp-not-eq*)
qed

The last premise ensures that a and b have the same sign

lemma *divide-strict-left-mono*:
$[\![b < a;\ 0 < c;\ 0 < a*b]\!] \Longrightarrow c \mathbin{/} a < c \mathbin{/} b$
by (*auto simp*: *field-simps zero-less-mult-iff mult-strict-right-mono*)

lemma *divide-left-mono*:
$[\![b \leq a;\ 0 \leq c;\ 0 < a*b]\!] \Longrightarrow c \mathbin{/} a \leq c \mathbin{/} b$
by (*auto simp*: *field-simps zero-less-mult-iff mult-right-mono*)

lemma *divide-strict-left-mono-neg*:
$[\![a < b;\ c < 0;\ 0 < a*b]\!] \Longrightarrow c \mathbin{/} a < c \mathbin{/} b$
by (*auto simp*: *field-simps zero-less-mult-iff mult-strict-right-mono-neg*)

lemma *mult-imp-div-pos-le*: $0 < y \Longrightarrow x \leq z * y \Longrightarrow x \mathbin{/} y \leq z$
by (*subst pos-divide-le-eq*, *assumption*+)

lemma *mult-imp-le-div-pos*: $0 < y \Longrightarrow z * y \leq x \Longrightarrow z \leq x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *mult-imp-div-pos-less*: $0 < y \Longrightarrow x < z * y \Longrightarrow x \mathbin{/} y < z$
by(*simp add*:*field-simps*)

lemma *mult-imp-less-div-pos*: $0 < y \Longrightarrow z * y < x \Longrightarrow z < x \mathbin{/} y$
by(*simp add*:*field-simps*)

lemma *frac-le*:
assumes $0 \leq y$ $x \leq y$ $0 < w$ $w \leq z$
shows $x \mathbin{/} z \leq y \mathbin{/} w$
proof (*rule mult-imp-div-pos-le*)
show $z > 0$
using *assms* **by** *simp*
have $x \leq y * z \mathbin{/} w$
proof (*rule mult-imp-le-div-pos* [*OF* ‹$0 < w$›])
show $x * w \leq y * z$
using *assms* **by** (*auto intro*: *mult-mono*)
qed
also have ... $= y \mathbin{/} w * z$
by *simp*
finally show $x \leq y \mathbin{/} w * z$ **.**
qed

lemma *frac-less*:
assumes $0 \leq x$ $x < y$ $0 < w$ $w \leq z$
shows $x \mathbin{/} z < y \mathbin{/} w$
proof (*rule mult-imp-div-pos-less*)
show $z > 0$
using *assms* **by** *simp*
have $x < y * z \mathbin{/} w$
proof (*rule mult-imp-less-div-pos* [*OF* ‹$0 < w$›])
show $x * w < y * z$
using *assms* **by** (*auto intro*: *mult-less-le-imp-less*)
qed
also have ... $= y \mathbin{/} w * z$
by *simp*
finally show $x < y \mathbin{/} w * z$ **.**
qed

lemma *frac-less2*:
assumes $0 < x$ $x \leq y$ $0 < w$ $w < z$
shows $x \mathbin{/} z < y \mathbin{/} w$
proof (*rule mult-imp-div-pos-less*)
show $z > 0$

using *assms* **by** *simp*
show $x < y \,/\, w * z$
using *assms* **by** (*force intro*: *mult-imp-less-div-pos mult-le-less-imp-less*)
qed

As above, with a better name

lemma *divide-mono*:
$[\![b \le a;\ c \le d;\ 0 < b;\ 0 \le c]\!] \Longrightarrow c \,/\, a \le d \,/\, b$
by (*simp add*: *frac-le*)

lemma *less-half-sum*: $a < b \Longrightarrow a < (a+b) \,/\, (1+1)$
by (*simp add*: *field-simps zero-less-two*)

lemma *gt-half-sum*: $a < b \Longrightarrow (a+b)/(1+1) < b$
by (*simp add*: *field-simps zero-less-two*)

subclass *unbounded-dense-linorder*
proof
fix $x\ y :: 'a$
from *less-add-one* **show** $\exists\, y.\ x < y$ **..**
from *less-add-one* **have** $x + (-1) < (x + 1) + (-1)$ **by** (*rule add-strict-right-mono*)
then have $x - 1 < x + 1 - 1$ **by** *simp*
then have $x - 1 < x$ **by** (*simp add*: *algebra-simps*)
then show $\exists\, y.\ y < x$ **..**
show $x < y \Longrightarrow \exists\, z{>}x.\ z < y$ **by** (*blast intro*!: *less-half-sum gt-half-sum*)
qed

subclass *field-abs-sgn* **..**

lemma *inverse-sgn* [*simp*]:
$inverse\ (sgn\ a) = sgn\ a$
by (*cases a 0 rule*: *linorder-cases*) *simp-all*

lemma *divide-sgn* [*simp*]:
$a \,/\, sgn\ b = a * sgn\ b$
by (*cases b 0 rule*: *linorder-cases*) *simp-all*

lemma *nonzero-abs-inverse*:
$a \neq 0 \Longrightarrow |inverse\ a| = inverse\ |a|$
by (*rule abs-inverse*)

lemma *nonzero-abs-divide*:
$b \neq 0 \Longrightarrow |a \,/\, b| = |a| \,/\, |b|$
by (*rule abs-divide*)

lemma *field-le-epsilon*:
assumes *e*: $\bigwedge e.\ 0 < e \Longrightarrow x \le y + e$
shows $x \le y$
proof (*rule dense-le*)

fix t **assume** $t < x$
hence $0 < x - t$ **by** (*simp add*: *less-diff-eq*)
from e [*OF this*] **have** $x + 0 \leq x + (y - t)$ **by** (*simp add*: *algebra-simps*)
then have $0 \leq y - t$ **by** (*simp only*: *add-le-cancel-left*)
then show $t \leq y$ **by** (*simp add*: *algebra-simps*)
qed

lemma *inverse-positive-iff-positive* [*simp*]: $(0 < inverse\ a) = (0 < a)$
proof (*cases* $a = 0$)
case *False*
then show *?thesis*
by (*blast intro*: *inverse-positive-imp-positive positive-imp-inverse-positive*)
qed *auto*

lemma *inverse-negative-iff-negative* [*simp*]: $(inverse\ a < 0) = (a < 0)$
proof (*cases* $a = 0$)
case *False*
then show *?thesis*
by (*blast intro*: *inverse-negative-imp-negative negative-imp-inverse-negative*)
qed *auto*

lemma *inverse-nonnegative-iff-nonnegative* [*simp*]: $0 \leq inverse\ a \longleftrightarrow 0 \leq a$
by (*simp add*: *not-less* [*symmetric*])

lemma *inverse-nonpositive-iff-nonpositive* [*simp*]: $inverse\ a \leq 0 \longleftrightarrow a \leq 0$
by (*simp add*: *not-less* [*symmetric*])

lemma *one-less-inverse-iff*: $1 < inverse\ x \longleftrightarrow 0 < x \wedge x < 1$
using *less-trans*[*of 1 x 0* **for** *x*]
by (*cases x 0 rule*: *linorder-cases*) (*auto simp add*: *field-simps*)

lemma *one-le-inverse-iff*: $1 \leq inverse\ x \longleftrightarrow 0 < x \wedge x \leq 1$
proof (*cases* $x = 1$)
case *True* **then show** *?thesis* **by** *simp*
next
case *False* **then have** $inverse\ x \neq 1$ **by** *simp*
then have $1 \neq inverse\ x$ **by** *blast*
then have $1 \leq inverse\ x \longleftrightarrow 1 < inverse\ x$ **by** (*simp add*: *le-less*)
with *False* **show** *?thesis* **by** (*auto simp add*: *one-less-inverse-iff*)
qed

lemma *inverse-less-1-iff*: $inverse\ x < 1 \longleftrightarrow x \leq 0 \vee 1 < x$
by (*simp add*: *not-le* [*symmetric*] *one-le-inverse-iff*)

lemma *inverse-le-1-iff*: $inverse\ x \leq 1 \longleftrightarrow x \leq 0 \vee 1 \leq x$
by (*simp add*: *not-less* [*symmetric*] *one-less-inverse-iff*)

lemma [*field-split-simps*, *divide-simps*]:
shows *le-divide-eq*: $a \leq b\ /\ c \longleftrightarrow (if\ 0 < c\ then\ a * c \leq b\ else\ if\ c < 0\ then\ b$

$\leq a * c$ *else* $a \leq 0$)
and *divide-le-eq*: $b \mathbin{/} c \leq a \longleftrightarrow$ (*if* $0 < c$ *then* $b \leq a * c$ *else if* $c < 0$ *then* $a * c \leq b$ *else* $0 \leq a$)
and *less-divide-eq*: $a < b \mathbin{/} c \longleftrightarrow$ (*if* $0 < c$ *then* $a * c < b$ *else if* $c < 0$ *then* $b < a * c$ *else* $a < 0$)
and *divide-less-eq*: $b \mathbin{/} c < a \longleftrightarrow$ (*if* $0 < c$ *then* $b < a * c$ *else if* $c < 0$ *then* $a * c < b$ *else* $0 < a$)
and *le-minus-divide-eq*: $a \leq - (b \mathbin{/} c) \longleftrightarrow$ (*if* $0 < c$ *then* $a * c \leq - b$ *else if* $c < 0$ *then* $- b \leq a * c$ *else* $a \leq 0$)
and *minus-divide-le-eq*: $- (b \mathbin{/} c) \leq a \longleftrightarrow$ (*if* $0 < c$ *then* $- b \leq a * c$ *else if* $c < 0$ *then* $a * c \leq - b$ *else* $0 \leq a$)
and *less-minus-divide-eq*: $a < - (b \mathbin{/} c) \longleftrightarrow$ (*if* $0 < c$ *then* $a * c < - b$ *else if* $c < 0$ *then* $- b < a * c$ *else* $a < 0$)
and *minus-divide-less-eq*: $- (b \mathbin{/} c) < a \longleftrightarrow$ (*if* $0 < c$ *then* $- b < a * c$ *else if* $c < 0$ *then* $a * c < - b$ *else* $0 < a$)
by (*auto simp*: *field-simps not-less dest*: *order.antisym*)

Division and Signs

lemma
shows *zero-less-divide-iff*: $0 < a \mathbin{/} b \longleftrightarrow 0 < a \wedge 0 < b \vee a < 0 \wedge b < 0$
and *divide-less-0-iff*: $a \mathbin{/} b < 0 \longleftrightarrow 0 < a \wedge b < 0 \vee a < 0 \wedge 0 < b$
and *zero-le-divide-iff*: $0 \leq a \mathbin{/} b \longleftrightarrow 0 \leq a \wedge 0 \leq b \vee a \leq 0 \wedge b \leq 0$
and *divide-le-0-iff*: $a \mathbin{/} b \leq 0 \longleftrightarrow 0 \leq a \wedge b \leq 0 \vee a \leq 0 \wedge 0 \leq b$
by (*auto simp add*: *field-split-simps*)

Division and the Number One

Simplify expressions equated with 1

lemma *zero-eq-1-divide-iff* [*simp*]: $0 = 1 \mathbin{/} a \longleftrightarrow a = 0$
by (*cases* $a = 0$) (*auto simp*: *field-simps*)

lemma *one-divide-eq-0-iff* [*simp*]: $1 \mathbin{/} a = 0 \longleftrightarrow a = 0$
using *zero-eq-1-divide-iff*[*of a*] **by** *simp*

Simplify expressions such as $0 < 1/x$ to $0 < x$

lemma *zero-le-divide-1-iff* [*simp*]:
$0 \leq 1 \mathbin{/} a \longleftrightarrow 0 \leq a$
by (*simp add*: *zero-le-divide-iff*)

lemma *zero-less-divide-1-iff* [*simp*]:
$0 < 1 \mathbin{/} a \longleftrightarrow 0 < a$
by (*simp add*: *zero-less-divide-iff*)

lemma *divide-le-0-1-iff* [*simp*]:
$1 \mathbin{/} a \leq 0 \longleftrightarrow a \leq 0$
by (*simp add*: *divide-le-0-iff*)

lemma *divide-less-0-1-iff* [*simp*]:
$1 \mathbin{/} a < 0 \longleftrightarrow a < 0$

by (*simp add*: *divide-less-0-iff*)

lemma *divide-right-mono*:
$[\![a \leq b;\ 0 \leq c]\!] \Longrightarrow a/c \leq b/c$
by (*force simp add*: *divide-strict-right-mono le-less*)

lemma *divide-right-mono-neg*: $a \leq b \Longrightarrow c \leq 0 \Longrightarrow b\ /\ c \leq a\ /\ c$
by (*auto dest*: *divide-right-mono* [*of* - - − *c*])

lemma *divide-left-mono-neg*: $a \leq b \Longrightarrow c \leq 0 \Longrightarrow 0 < a * b \Longrightarrow c\ /\ a \leq c\ /\ b$
by (*auto simp add*: *mult.commute dest*: *divide-left-mono* [*of* - - − *c*])

lemma *inverse-le-iff*: *inverse* $a \leq$ *inverse* $b \longleftrightarrow (0 < a * b \longrightarrow b \leq a) \wedge (a * b \leq 0 \longrightarrow a \leq b)$
by (*cases a 0 b 0 rule*: *linorder-cases*[*case-product linorder-cases*])
(*auto simp add*: *field-simps zero-less-mult-iff mult-le-0-iff*)

lemma *inverse-less-iff*: *inverse* $a <$ *inverse* $b \longleftrightarrow (0 < a * b \longrightarrow b < a) \wedge (a * b \leq 0 \longrightarrow a < b)$
by (*subst less-le*) (*auto simp*: *inverse-le-iff*)

lemma *divide-le-cancel*: $a\ /\ c \leq b\ /\ c \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*simp add*: *divide-inverse mult-le-cancel-right*)

lemma *divide-less-cancel*: $a\ /\ c < b\ /\ c \longleftrightarrow (0 < c \longrightarrow a < b) \wedge (c < 0 \longrightarrow b < a) \wedge c \neq 0$
by (*auto simp add*: *divide-inverse mult-less-cancel-right*)

Simplify quotients that are compared with the value 1.

lemma *le-divide-eq-1*:
$(1 \leq b\ /\ a) = ((0 < a \wedge a \leq b) \vee (a < 0 \wedge b \leq a))$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1*:
$(b\ /\ a \leq 1) = ((0 < a \wedge b \leq a) \vee (a < 0 \wedge a \leq b) \vee a{=}0)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1*:
$(1 < b\ /\ a) = ((0 < a \wedge a < b) \vee (a < 0 \wedge b < a))$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1*:
$(b\ /\ a < 1) = ((0 < a \wedge b < a) \vee (a < 0 \wedge a < b) \vee a{=}0)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-nonneg-nonneg* [*simp*]:
$0 \leq x \Longrightarrow 0 \leq y \Longrightarrow 0 \leq x\ /\ y$
by (*auto simp add*: *field-split-simps*)

lemma *divide-nonpos-nonpos*:
$x \leq 0 \Longrightarrow y \leq 0 \Longrightarrow 0 \leq x \mathbin{/} y$
by (*auto simp add*: *field-split-simps*)

lemma *divide-nonneg-nonpos*:
$0 \leq x \Longrightarrow y \leq 0 \Longrightarrow x \mathbin{/} y \leq 0$
by (*auto simp add*: *field-split-simps*)

lemma *divide-nonpos-nonneg*:
$x \leq 0 \Longrightarrow 0 \leq y \Longrightarrow x \mathbin{/} y \leq 0$
by (*auto simp add*: *field-split-simps*)

Conditional Simplification Rules: No Case Splits

lemma *le-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 \leq b/a) = (a \leq b)$
by (*auto simp add*: *le-divide-eq*)

lemma *le-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 \leq b/a) = (b \leq a)$
by (*auto simp add*: *le-divide-eq*)

lemma *divide-le-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a \leq 1) = (b \leq a)$
by (*auto simp add*: *divide-le-eq*)

lemma *divide-le-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (b/a \leq 1) = (a \leq b)$
by (*auto simp add*: *divide-le-eq*)

lemma *less-divide-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (1 < b/a) = (a < b)$
by (*auto simp add*: *less-divide-eq*)

lemma *less-divide-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow (1 < b/a) = (b < a)$
by (*auto simp add*: *less-divide-eq*)

lemma *divide-less-eq-1-pos* [*simp*]:
$0 < a \Longrightarrow (b/a < 1) = (b < a)$
by (*auto simp add*: *divide-less-eq*)

lemma *divide-less-eq-1-neg* [*simp*]:
$a < 0 \Longrightarrow b/a < 1 \longleftrightarrow a < b$
by (*auto simp add*: *divide-less-eq*)

lemma *eq-divide-eq-1* [*simp*]:
$(1 = b/a) = ((a \neq 0 \wedge a = b))$
by (*auto simp add*: *eq-divide-eq*)

lemma *divide-eq-eq-1* [*simp*]:
(*b*/*a* = *1*) = ((*a* ≠ *0* ∧ *a* = *b*))
by (*auto simp add*: *divide-eq-eq*)

lemma *abs-div-pos*: *0* < *y* ⟹ |*x*| / *y* = |*x* / *y*|
by (*simp add*: *order-less-imp-le*)

lemma *zero-le-divide-abs-iff* [*simp*]: (*0* ≤ *a* / |*b*|) = (*0* ≤ *a* ∨ *b* = *0*)
by (*auto simp*: *zero-le-divide-iff*)

lemma *divide-le-0-abs-iff* [*simp*]: (*a* / |*b*| ≤ *0*) = (*a* ≤ *0* ∨ *b* = *0*)
by (*auto simp*: *divide-le-0-iff*)

lemma *field-le-mult-one-interval*:
assumes ∗: ⋀*z*. ⟦ *0* < *z* ; *z* < *1* ⟧ ⟹ *z* ∗ *x* ≤ *y*
shows *x* ≤ *y*
proof (*cases 0* < *x*)
assume *0* < *x*
thus *?thesis*
using *dense-le-bounded*[*of 0 1 y*/*x*] ∗
unfolding *le-divide-eq if-P*[*OF* ‹*0* < *x*›] **by** *simp*
next
assume ¬*0* < *x* **hence** *x* ≤ *0* **by** *simp*
obtain *s*::′*a* **where** *s*: *0* < *s s* < *1* **using** *dense*[*of 0 1*::′*a*] **by** *auto*
hence *x* ≤ *s* ∗ *x* **using** *mult-le-cancel-right*[*of 1 x s*] ‹*x* ≤ *0*› **by** *auto*
also note ∗[*OF s*]
finally show *?thesis* **.**
qed

For creating values between *u* and *v*.

lemma *scaling-mono*:
assumes *u* ≤ *v* *0* ≤ *r r* ≤ *s*
shows *u* + *r* ∗ (*v* − *u*) / *s* ≤ *v*
proof −
have *r*/*s* ≤ *1* **using** *assms*
using *divide-le-eq-1* **by** *fastforce*
moreover have *0* ≤ *v* − *u*
using *assms* **by** *simp*
ultimately have (*r*/*s*) ∗ (*v* − *u*) ≤ *1* ∗ (*v* − *u*)
by (*rule mult-right-mono*)
then show *?thesis*
by (*simp add*: *field-simps*)
qed

end

Min/max Simplification Rules

lemma *min-mult-distrib-left*:

fixes *x*::*'a*::*linordered-idom*
shows *p* ∗ *min x y* = (*if* 0 ≤ *p then min* (*p*∗*x*) (*p*∗*y*) *else max* (*p*∗*x*) (*p*∗*y*))
by (*auto simp add*: *min-def max-def mult-le-cancel-left*)

lemma *min-mult-distrib-right*:
fixes *x*::*'a*::*linordered-idom*
shows *min x y* ∗ *p* = (*if* 0 ≤ *p then min* (*x*∗*p*) (*y*∗*p*) *else max* (*x*∗*p*) (*y*∗*p*))
by (*auto simp add*: *min-def max-def mult-le-cancel-right*)

lemma *min-divide-distrib-right*:
fixes *x*::*'a*::*linordered-field*
shows *min x y* / *p* = (*if* 0 ≤ *p then min* (*x*/*p*) (*y*/*p*) *else max* (*x*/*p*) (*y*/*p*))
by (*simp add*: *min-mult-distrib-right divide-inverse*)

lemma *max-mult-distrib-left*:
fixes *x*::*'a*::*linordered-idom*
shows *p* ∗ *max x y* = (*if* 0 ≤ *p then max* (*p*∗*x*) (*p*∗*y*) *else min* (*p*∗*x*) (*p*∗*y*))
by (*auto simp add*: *min-def max-def mult-le-cancel-left*)

lemma *max-mult-distrib-right*:
fixes *x*::*'a*::*linordered-idom*
shows *max x y* ∗ *p* = (*if* 0 ≤ *p then max* (*x*∗*p*) (*y*∗*p*) *else min* (*x*∗*p*) (*y*∗*p*))
by (*auto simp add*: *min-def max-def mult-le-cancel-right*)

lemma *max-divide-distrib-right*:
fixes *x*::*'a*::*linordered-field*
shows *max x y* / *p* = (*if* 0 ≤ *p then max* (*x*/*p*) (*y*/*p*) *else min* (*x*/*p*) (*y*/*p*))
by (*simp add*: *max-mult-distrib-right divide-inverse*)

hide-fact (**open**) *field-inverse field-divide-inverse field-inverse-zero*

code-identifier
code-module *Fields* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

19 Relations – as sets of pairs, and binary predicates

theory *Relation*
imports *Product-Type Sum-Type Fields*
begin

A preliminary: classical rules for reasoning on predicates

declare *predicate1I* [*Pure.intro*!, *intro*!]
declare *predicate1D* [*Pure.dest*, *dest*]
declare *predicate2I* [*Pure.intro*!, *intro*!]
declare *predicate2D* [*Pure.dest*, *dest*]

declare *bot1E* [*elim*!]
declare *bot2E* [*elim*!]
declare *top1I* [*intro*!]
declare *top2I* [*intro*!]
declare *inf1I* [*intro*!]
declare *inf2I* [*intro*!]
declare *inf1E* [*elim*!]
declare *inf2E* [*elim*!]
declare *sup1I1* [*intro?*]
declare *sup2I1* [*intro?*]
declare *sup1I2* [*intro?*]
declare *sup2I2* [*intro?*]
declare *sup1E* [*elim*!]
declare *sup2E* [*elim*!]
declare *sup1CI* [*intro*!]
declare *sup2CI* [*intro*!]
declare *Inf1-I* [*intro*!]
declare *INF1-I* [*intro*!]
declare *Inf2-I* [*intro*!]
declare *INF2-I* [*intro*!]
declare *Inf1-D* [*elim*]
declare *INF1-D* [*elim*]
declare *Inf2-D* [*elim*]
declare *INF2-D* [*elim*]
declare *Inf1-E* [*elim*]
declare *INF1-E* [*elim*]
declare *Inf2-E* [*elim*]
declare *INF2-E* [*elim*]
declare *Sup1-I* [*intro*]
declare *SUP1-I* [*intro*]
declare *Sup2-I* [*intro*]
declare *SUP2-I* [*intro*]
declare *Sup1-E* [*elim*!]
declare *SUP1-E* [*elim*!]
declare *Sup2-E* [*elim*!]
declare *SUP2-E* [*elim*!]

19.1 Fundamental

19.1.1 Relations as sets of pairs

type-synonym $'a\ rel = ('a \times 'a)\ set$

lemma *subrelI*: $(\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (x,\ y) \in s) \Longrightarrow r \subseteq s$
— Version of *subsetI* for binary relations
by *auto*

lemma *lfp-induct2*:
$(a,\ b) \in lfp\ f \Longrightarrow mono\ f \Longrightarrow$
$(\bigwedge a\ b.\ (a,\ b) \in f\ (lfp\ f \cap \{(x,\ y).\ P\ x\ y\}) \Longrightarrow P\ a\ b) \Longrightarrow P\ a\ b$

— Version of *lfp-induct* for binary relations
using *lfp-induct-set* [*of* (a, b) *f case-prod P*] **by** *auto*

19.1.2 Conversions between set and predicate relations

lemma *pred-equals-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) = (\lambda x.\ x \in S) \longleftrightarrow R = S$
by (*simp add*: *set-eq-iff fun-eq-iff*)

lemma *pred-equals-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) = (\lambda x\ y.\ (x, y) \in S) \longleftrightarrow R = S$
by (*simp add*: *set-eq-iff fun-eq-iff*)

lemma *pred-subset-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \leq (\lambda x.\ x \in S) \longleftrightarrow R \subseteq S$
by (*simp add*: *subset-iff le-fun-def*)

lemma *pred-subset-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \leq (\lambda x\ y.\ (x, y) \in S) \longleftrightarrow R \subseteq S$
by (*simp add*: *subset-iff le-fun-def*)

lemma *bot-empty-eq* [*pred-set-conv*]: $\bot = (\lambda x.\ x \in \{\})$
by (*auto simp add*: *fun-eq-iff*)

lemma *bot-empty-eq2* [*pred-set-conv*]: $\bot = (\lambda x\ y.\ (x, y) \in \{\})$
by (*auto simp add*: *fun-eq-iff*)

lemma *top-empty-eq*: $\top = (\lambda x.\ x \in UNIV)$
by (*auto simp add*: *fun-eq-iff*)

lemma *top-empty-eq2*: $\top = (\lambda x\ y.\ (x, y) \in UNIV)$
by (*auto simp add*: *fun-eq-iff*)

lemma *inf-Int-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \sqcap (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cap S)$
by (*simp add*: *inf-fun-def*)

lemma *inf-Int-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \sqcap (\lambda x\ y.\ (x, y) \in S) = (\lambda x\ y.\ (x, y) \in R \cap S)$
by (*simp add*: *inf-fun-def*)

lemma *sup-Un-eq* [*pred-set-conv*]: $(\lambda x.\ x \in R) \sqcup (\lambda x.\ x \in S) = (\lambda x.\ x \in R \cup S)$
by (*simp add*: *sup-fun-def*)

lemma *sup-Un-eq2* [*pred-set-conv*]: $(\lambda x\ y.\ (x, y) \in R) \sqcup (\lambda x\ y.\ (x, y) \in S) = (\lambda x\ y.\ (x, y) \in R \cup S)$
by (*simp add*: *sup-fun-def*)

lemma *INF-INT-eq* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (\bigcap i{\in}S.\ r\ i))$
by (*simp add*: *fun-eq-iff*)

lemma *INF-INT-eq2* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (\bigcap i{\in}S.\ r\ i))$
by (*simp add*: *fun-eq-iff*)

lemma *SUP-UN-eq* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x.\ x \in r\ i)) = (\lambda x.\ x \in (\bigcup i{\in}S.\ r\ i))$
by (*simp add*: *fun-eq-iff*)

lemma *SUP-UN-eq2* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in r\ i)) = (\lambda x\ y.\ (x,\ y) \in (\bigcup i{\in}S.\ r\ i))$
by (*simp add*: *fun-eq-iff*)

lemma *Inf-INT-eq* [*pred-set-conv*]: $\bigsqcap S = (\lambda x.\ x \in (\bigcap (Collect\ `\ S)))$
by (*simp add*: *fun-eq-iff*)

lemma *INF-Int-eq* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x.\ x \in i)) = (\lambda x.\ x \in \bigcap S)$
by (*simp add*: *fun-eq-iff*)

lemma *Inf-INT-eq2* [*pred-set-conv*]: $\bigsqcap S = (\lambda x\ y.\ (x,\ y) \in (\bigcap (Collect\ `\ case\text{-}prod\ `\ S)))$
by (*simp add*: *fun-eq-iff*)

lemma *INF-Int-eq2* [*pred-set-conv*]: $(\bigsqcap i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in i)) = (\lambda x\ y.\ (x,\ y) \in \bigcap S)$
by (*simp add*: *fun-eq-iff*)

lemma *Sup-SUP-eq* [*pred-set-conv*]: $\bigsqcup S = (\lambda x.\ x \in \bigcup (Collect\ `\ S))$
by (*simp add*: *fun-eq-iff*)

lemma *SUP-Sup-eq* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x.\ x \in i)) = (\lambda x.\ x \in \bigcup S)$
by (*simp add*: *fun-eq-iff*)

lemma *Sup-SUP-eq2* [*pred-set-conv*]: $\bigsqcup S = (\lambda x\ y.\ (x,\ y) \in (\bigcup (Collect\ `\ case\text{-}prod\ `\ S)))$
by (*simp add*: *fun-eq-iff*)

lemma *SUP-Sup-eq2* [*pred-set-conv*]: $(\bigsqcup i{\in}S.\ (\lambda x\ y.\ (x,\ y) \in i)) = (\lambda x\ y.\ (x,\ y) \in \bigcup S)$
by (*simp add*: *fun-eq-iff*)

19.2 Properties of relations

19.2.1 Reflexivity

definition *refl-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
where *refl-on A r* ⟷ $(\forall x{\in}A.\ (x,\ x) \in r)$

abbreviation *refl* :: *'a rel* ⇒ *bool* — reflexivity over a type
where *refl* ≡ *refl-on UNIV*

definition *reflp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *reflp-on A R* ⟷ (∀ *x*∈*A*. *R x x*)

abbreviation *reflp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool*
where *reflp* ≡ *reflp-on UNIV*

lemma *reflp-def*[*no-atp*]: *reflp R* ⟷ (∀ *x*. *R x x*)
by (*simp add*: *reflp-on-def*)

reflp-def is for backward compatibility.

lemma *reflp-on-refl-on-eq*[*pred-set-conv*]: *reflp-on A* (λ*a b*. (*a*, *b*) ∈ *r*) ⟷ *refl-on A r*
by (*simp add*: *refl-on-def reflp-on-def*)

lemmas *reflp-refl-eq* = *reflp-on-refl-on-eq*[*of UNIV*]

lemma *refl-onI* [*intro?*]: (⋀*x*. *x* ∈ *A* ⟹ (*x*, *x*) ∈ *r*) ⟹ *refl-on A r*
unfolding *refl-on-def* **by** (*iprover intro*!: *ballI*)

lemma *reflI*: (⋀*x*. (*x*, *x*) ∈ *r*) ⟹ *refl r*
by (*auto intro*: *refl-onI*)

lemma *reflp-onI*:
(⋀*x*. *x* ∈ *A* ⟹ *R x x*) ⟹ *reflp-on A R*
by (*simp add*: *reflp-on-def*)

lemma *reflpI*[*intro?*]: (⋀*x*. *R x x*) ⟹ *reflp R*
by (*rule reflp-onI*)

lemma *refl-onD*: *refl-on A r* ⟹ *a* ∈ *A* ⟹ (*a*, *a*) ∈ *r*
unfolding *refl-on-def* **by** *blast*

lemma *reflD*: *refl r* ⟹ (*a*, *a*) ∈ *r*
unfolding *refl-on-def* **by** *blast*

lemma *reflp-onD*:
reflp-on A R ⟹ *x* ∈ *A* ⟹ *R x x*
by (*simp add*: *reflp-on-def*)

lemma *reflpD*[*dest?*]: *reflp R* ⟹ *R x x*
by (*simp add*: *reflp-onD*)

lemma *reflpE*:
assumes *reflp r*
obtains *r x x*
using *assms* **by** (*auto dest*: *refl-onD simp add*: *reflp-def*)

lemma *refl-on-top*[*simp*]: *refl-on A* ⊤
by (*simp add*: *refl-on-def*)

lemma *reflp-on-top*[*simp*]: *reflp-on A* ⊤
by (*simp add*: *reflp-on-def*)

lemma *reflp-on-mono-strong*:
reflp-on B R ⟹ *A* ⊆ *B* ⟹ (⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *R x y* ⟹ *Q x y*) ⟹ *reflp-on A Q*
by (*rule reflp-onI*) (*auto dest*: *reflp-onD*)

lemma *reflp-on-mono*[*mono*]: *A* ⊆ *B* ⟹ *R* ≤ *Q* ⟹ *reflp-on B R* ≤ *reflp-on A Q*
by (*simp add*: *reflp-on-mono-strong le-fun-def*)

lemma *reflp-on-subset*: *reflp-on B R* ⟹ *A* ⊆ *B* ⟹ *reflp-on A R*
using *reflp-on-mono-strong* **.**

lemma *reflp-on-image*: *reflp-on* (*f* ‘ *A*) *R* ⟷ *reflp-on A* (λ*a b*. *R* (*f a*) (*f b*))
by (*simp add*: *reflp-on-def*)

lemma *refl-on-Int*: *refl-on A r* ⟹ *refl-on B s* ⟹ *refl-on* (*A* ∩ *B*) (*r* ∩ *s*)
unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-inf*: *reflp-on A R* ⟹ *reflp-on B S* ⟹ *reflp-on* (*A* ∩ *B*) (*R* ⊓ *S*)
by (*auto intro*: *reflp-onI dest*: *reflp-onD*)

lemma *reflp-inf*: *reflp r* ⟹ *reflp s* ⟹ *reflp* (*r* ⊓ *s*)
by (*rule reflp-on-inf*[*of UNIV - UNIV*, *unfolded Int-absorb*])

lemma *refl-on-Un*: *refl-on A r* ⟹ *refl-on B s* ⟹ *refl-on* (*A* ∪ *B*) (*r* ∪ *s*)
unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-sup*: *reflp-on A R* ⟹ *reflp-on B S* ⟹ *reflp-on* (*A* ∪ *B*) (*R* ⊔ *S*)
by (*auto intro*: *reflp-onI dest*: *reflp-onD*)

lemma *reflp-sup*: *reflp r* ⟹ *reflp s* ⟹ *reflp* (*r* ⊔ *s*)
by (*rule reflp-on-sup*[*of UNIV - UNIV*, *unfolded Un-absorb*])

lemma *refl-on-INTER*: ∀ *x*∈*S*. *refl-on* (*A x*) (*r x*) ⟹ *refl-on* (⋂(*A* ‘ *S*)) (⋂(*r* ‘ *S*))
unfolding *refl-on-def* **by** *fast*

lemma *reflp-on-Inf*: ∀ *x*∈*S*. *reflp-on* (*A x*) (*R x*) ⟹ *reflp-on* (⋂(*A* ‘ *S*)) (⨅(*R* ‘ *S*))
by (*auto intro*: *reflp-onI dest*: *reflp-onD*)

lemma *refl-on-UNION*: ∀ *x*∈*S*. *refl-on* (*A x*) (*r x*) ⟹ *refl-on* (⋃(*A* ‘ *S*)) (⋃(*r* ‘ *S*))
unfolding *refl-on-def* **by** *blast*

lemma *reflp-on-Sup*: $\forall x{\in}S.$ *reflp-on* $(A\ x)\ (R\ x) \Longrightarrow$ *reflp-on* $(\bigcup(A\ `\ S))\ (\bigsqcup(R\ `\ S))$
by (*auto intro*: *reflp-onI dest*: *reflp-onD*)

lemma *refl-on-empty* [*simp*]: *refl-on* $\{\}\ r$
by (*simp add*: *refl-on-def*)

lemma *reflp-on-empty* [*simp*]: *reflp-on* $\{\}\ R$
by (*auto intro*: *reflp-onI*)

lemma *refl-on-singleton* [*simp*]: *refl-on* $\{x\}\ \{(x,\ x)\}$
by (*blast intro*: *refl-onI*)

lemma *reflp-on-equality* [*simp*]: *reflp-on* $A\ (=)$
by (*simp add*: *reflp-on-def*)

lemma (**in** *preorder*) *reflp-on-le*[*simp*]: *reflp-on* $A\ (\leq)$
by (*simp add*: *reflp-onI*)

lemma (**in** *preorder*) *reflp-on-ge*[*simp*]: *reflp-on* $A\ (\geq)$
by (*simp add*: *reflp-onI*)

19.2.2 Irreflexivity

definition *irrefl-on* :: *'a set* ⇒ *'a rel* ⇒ *bool* **where**
$\textit{irrefl-on}\ A\ r \longleftrightarrow (\forall a \in A.\ (a,\ a) \notin r)$

abbreviation *irrefl* :: *'a rel* ⇒ *bool* **where**
$\textit{irrefl} \equiv \textit{irrefl-on}\ \textit{UNIV}$

definition *irreflp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
$\textit{irreflp-on}\ A\ R \longleftrightarrow (\forall a \in A.\ \neg\ R\ a\ a)$

abbreviation *irreflp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
$\textit{irreflp} \equiv \textit{irreflp-on}\ \textit{UNIV}$

lemma *irrefl-def*[*no-atp*]: $\textit{irrefl}\ r \longleftrightarrow (\forall a.\ (a,\ a) \notin r)$
by (*simp add*: *irrefl-on-def*)

lemma *irreflp-def*[*no-atp*]: $\textit{irreflp}\ R \longleftrightarrow (\forall a.\ \neg\ R\ a\ a)$
by (*simp add*: *irreflp-on-def*)

irrefl-def and *irreflp-def* are for backward compatibility.

lemma *irreflp-on-irrefl-on-eq* [*pred-set-conv*]: $\textit{irreflp-on}\ A\ (\lambda a\ b.\ (a,\ b) \in r) \longleftrightarrow \textit{irrefl-on}\ A\ r$
by (*simp add*: *irrefl-on-def irreflp-on-def*)

lemmas *irreflp-irrefl-eq* = *irreflp-on-irrefl-on-eq*[*of UNIV*]

lemma *irrefl-onI*: $(\bigwedge a.\ a \in A \Longrightarrow (a,\ a) \notin r) \Longrightarrow$ *irrefl-on A r*
by (*simp add*: *irrefl-on-def*)

lemma *irreflI*[*intro?*]: $(\bigwedge a.\ (a,\ a) \notin r) \Longrightarrow$ *irrefl r*
by (*rule irrefl-onI*[*of UNIV*, *simplified*])

lemma *irreflp-onI*: $(\bigwedge a.\ a \in A \Longrightarrow \neg\ R\ a\ a) \Longrightarrow$ *irreflp-on A R*
by (*rule irrefl-onI*[*to-pred*])

lemma *irreflpI*[*intro?*]: $(\bigwedge a.\ \neg\ R\ a\ a) \Longrightarrow$ *irreflp R*
by (*rule irreflI*[*to-pred*])

lemma *irrefl-onD*: *irrefl-on A r* $\Longrightarrow a \in A \Longrightarrow (a,\ a) \notin r$
by (*simp add*: *irrefl-on-def*)

lemma *irreflD*: *irrefl r* $\Longrightarrow (x,\ x) \notin r$
by (*rule irrefl-onD*[*of UNIV*, *simplified*])

lemma *irreflp-onD*: *irreflp-on A R* $\Longrightarrow a \in A \Longrightarrow \neg\ R\ a\ a$
by (*rule irrefl-onD*[*to-pred*])

lemma *irreflpD*: *irreflp R* $\Longrightarrow \neg\ R\ x\ x$
by (*rule irreflD*[*to-pred*])

lemma *irrefl-on-bot*[*simp*]: *irrefl-on A* $\bot$
by (*simp add*: *irrefl-on-def*)

lemma *irreflp-on-bot*[*simp*]: *irreflp-on A* $\bot$
using *irrefl-on-bot*[*to-pred*] **.**

lemma *irrefl-on-distinct* [*code*]: *irrefl-on A r* $\longleftrightarrow$ $(\forall (a,\ b) \in r.\ a \in A \longrightarrow b \in A \longrightarrow a \neq b)$
by (*auto simp add*: *irrefl-on-def*)

lemmas *irrefl-distinct* = *irrefl-on-distinct* — For backward compatibility

lemma *irreflp-on-mono-strong*:
irreflp-on B Q $\Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow$ *irreflp-on A R*
by (*rule irreflp-onI*) (*auto dest*: *irreflp-onD*)

lemma *irreflp-on-mono*[*mono*]: $A \subseteq B \Longrightarrow R \leq Q \Longrightarrow$ *irreflp-on B Q* $\leq$ *irreflp-on A R*
by (*simp add*: *irreflp-on-mono-strong le-fun-def*)

lemma *irrefl-on-subset*: *irrefl-on B r* $\Longrightarrow A \subseteq B \Longrightarrow$ *irrefl-on A r*
by (*auto simp*: *irrefl-on-def*)

lemma *irreflp-on-subset*: *irreflp-on B R* $\Longrightarrow A \subseteq B \Longrightarrow$ *irreflp-on A R*

using *irreflp-on-mono-strong* **.**

lemma *irreflp-on-image*: *irreflp-on* $(f \text{ ` } A)$ $R \longleftrightarrow$ *irreflp-on* A $(\lambda a\ b.\ R\ (f\ a)\ (f\ b))$
by (*simp add*: *irreflp-on-def*)

lemma (**in** *preorder*) *irreflp-on-less*[*simp*]: *irreflp-on* A $(<)$
by (*simp add*: *irreflp-onI*)

lemma (**in** *preorder*) *irreflp-on-greater*[*simp*]: *irreflp-on* A $(>)$
by (*simp add*: *irreflp-onI*)

19.2.3 Asymmetry

definition *asym-on* :: *'a set* ⇒ *'a rel* ⇒ *bool* **where**
asym-on A $r \longleftrightarrow (\forall x \in A.\ \forall y \in A.\ (x, y) \in r \longrightarrow (y, x) \notin r)$

abbreviation *asym* :: *'a rel* ⇒ *bool* **where**
asym ≡ *asym-on UNIV*

definition *asymp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
asymp-on A $R \longleftrightarrow (\forall x \in A.\ \forall y \in A.\ R\ x\ y \longrightarrow \neg\ R\ y\ x)$

abbreviation *asymp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
asymp ≡ *asymp-on UNIV*

lemma *asymp-on-asym-on-eq*[*pred-set-conv*]: *asymp-on* A $(\lambda x\ y.\ (x, y) \in r) \longleftrightarrow$ *asym-on* A r
by (*simp add*: *asymp-on-def asym-on-def*)

lemmas *asymp-asym-eq* = *asymp-on-asym-on-eq*[*of UNIV*] — For backward compatibility

lemma *asym-onI*[*intro*]:
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in r \Longrightarrow (y, x) \notin r) \Longrightarrow$ *asym-on* A r
by (*simp add*: *asym-on-def*)

lemma *asymI*[*intro*]: $(\bigwedge x\ y.\ (x, y) \in r \Longrightarrow (y, x) \notin r) \Longrightarrow$ *asym* r
by (*simp add*: *asym-onI*)

lemma *asymp-onI*[*intro*]:
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow \neg\ R\ y\ x) \Longrightarrow$ *asymp-on* A R
by (*rule asym-onI*[*to-pred*])

lemma *asympI*[*intro*]: $(\bigwedge x\ y.\ R\ x\ y \Longrightarrow \neg\ R\ y\ x) \Longrightarrow$ *asymp* R
by (*rule asymI*[*to-pred*])

lemma *asym-onD*: *asym-on* A $r \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in r \Longrightarrow (y, x) \notin r$
by (*simp add*: *asym-on-def*)

lemma *asymD*: *asym r* $\Longrightarrow$ $(x, y) \in r \Longrightarrow (y, x) \notin r$
by (*simp add*: *asym-onD*)

lemma *asymp-onD*: *asymp-on A R* $\Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow \neg\ R\ y\ x$
by (*rule asym-onD*[*to-pred*])

lemma *asympD*: *asymp R* $\Longrightarrow R\ x\ y \Longrightarrow \neg\ R\ y\ x$
by (*rule asymD*[*to-pred*])

lemma *asym-on-bot*[*simp*]: *asym-on A* $\bot$
by (*simp add*: *asym-on-def*)

lemma *asymp-on-bot*[*simp*]: *asymp-on A* $\bot$
using *asym-on-bot*[*to-pred*] **.**

lemma *asym-iff*: *asym r* $\longleftrightarrow$ $(\forall x\ y.\ (x,y) \in r \longrightarrow (y,x) \notin r)$
by (*blast dest*: *asymD*)

lemma *asymp-on-mono-strong*:
asymp-on B Q $\Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow Q\ x\ y)$
$\Longrightarrow$ *asymp-on A R*
by (*rule asymp-onI*) (*auto dest*: *asymp-onD*)

lemma *asymp-on-mono*[*mono*]: $A \subseteq B \Longrightarrow R \leq Q \Longrightarrow$ *asymp-on B Q* $\leq$ *asymp-on A R*
by (*simp add*: *asymp-on-mono-strong le-fun-def*)

lemma *asym-on-subset*: *asym-on B r* $\Longrightarrow A \subseteq B \Longrightarrow$ *asym-on A r*
by (*auto simp*: *asym-on-def*)

lemma *asymp-on-subset*: *asymp-on B R* $\Longrightarrow A \subseteq B \Longrightarrow$ *asymp-on A R*
using *asymp-on-mono-strong* **.**

lemma *asymp-on-image*: *asymp-on* $(f\ `\ A)\ R \longleftrightarrow$ *asymp-on A* $(\lambda a\ b.\ R\ (f\ a)\ (f\ b))$
by (*simp add*: *asymp-on-def*)

lemma *irrefl-on-if-asym-on*[*simp*]: *asym-on A r* $\Longrightarrow$ *irrefl-on A r*
by (*auto intro*: *irrefl-onI dest*: *asym-onD*)

lemma *irreflp-on-if-asymp-on*[*simp*]: *asymp-on A r* $\Longrightarrow$ *irreflp-on A r*
by (*rule irrefl-on-if-asym-on*[*to-pred*])

lemma (**in** *preorder*) *asymp-on-less*[*simp*]: *asymp-on A* $(<)$
by (*auto intro*: *dual-order.asym*)

lemma (**in** *preorder*) *asymp-on-greater*[*simp*]: *asymp-on A* $(>)$
by (*auto intro*: *dual-order.asym*)

19.2.4 Symmetry

definition *sym-on* :: *'a set* ⇒ *'a rel* ⇒ *bool* **where**
sym-on A r ⟷ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. (*x*, *y*) ∈ *r* ⟶ (*y*, *x*) ∈ *r*)

abbreviation *sym* :: *'a rel* ⇒ *bool* **where**
sym ≡ *sym-on UNIV*

definition *symp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
symp-on A R ⟷ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. *R x y* ⟶ *R y x*)

abbreviation *symp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
symp ≡ *symp-on UNIV*

lemma *sym-def*[*no-atp*]: *sym r* ⟷ (∀ *x y*. (*x*, *y*) ∈ *r* ⟶ (*y*, *x*) ∈ *r*)
by (*simp add*: *sym-on-def*)

lemma *symp-def*[*no-atp*]: *symp R* ⟷ (∀ *x y*. *R x y* ⟶ *R y x*)
by (*simp add*: *symp-on-def*)

sym-def and *symp-def* are for backward compatibility.

lemma *symp-on-sym-on-eq*[*pred-set-conv*]: *symp-on A* (λ*x y*. (*x*, *y*) ∈ *r*) ⟷ *sym-on A r*
by (*simp add*: *sym-on-def symp-on-def*)

lemmas *symp-sym-eq* = *symp-on-sym-on-eq*[*of UNIV*] — For backward compatibility

lemma *sym-onI*: (⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ (*x*, *y*) ∈ *r* ⟹ (*y*, *x*) ∈ *r*) ⟹ *sym-on A r*
by (*simp add*: *sym-on-def*)

lemma *symI* [*intro?*]: (⋀*x y*. (*x*, *y*) ∈ *r* ⟹ (*y*, *x*) ∈ *r*) ⟹ *sym r*
by (*simp add*: *sym-onI*)

lemma *symp-onI*: (⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *R x y* ⟹ *R y x*) ⟹ *symp-on A R*
by (*rule sym-onI*[*to-pred*])

lemma *sympI* [*intro?*]: (⋀*x y*. *R x y* ⟹ *R y x*) ⟹ *symp R*
by (*rule symI*[*to-pred*])

lemma *symE*:
assumes *sym r* **and** (*b*, *a*) ∈ *r*
obtains (*a*, *b*) ∈ *r*
using *assms* **by** (*simp add*: *sym-def*)

lemma *sympE*:
assumes *symp r* **and** *r b a*
obtains *r a b*

using *assms* **by** (*rule symE* [*to-pred*])

lemma *sym-onD*: *sym-on A r* $\Longrightarrow$ $x \in A \Longrightarrow y \in A \Longrightarrow (x, y) \in r \Longrightarrow (y, x) \in r$
by (*simp add*: *sym-on-def*)

lemma *symD* [*dest?*]: *sym r* $\Longrightarrow$ $(x, y) \in r \Longrightarrow (y, x) \in r$
by (*simp add*: *sym-onD*)

lemma *symp-onD*: *symp-on A R* $\Longrightarrow$ $x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x$
by (*rule sym-onD*[*to-pred*])

lemma *sympD* [*dest?*]: *symp R* $\Longrightarrow$ $R\ x\ y \Longrightarrow R\ y\ x$
by (*rule symD*[*to-pred*])

lemma *sym-on-bot*[*simp*]: *sym-on A* $\bot$
by (*simp add*: *sym-on-def*)

lemma *symp-on-bot*[*simp*]: *symp-on A* $\bot$
using *sym-on-bot*[*to-pred*] **.**

lemma *sym-on-top*[*simp*]: *sym-on A* $\top$
by (*simp add*: *sym-on-def*)

lemma *symp-on-top*[*simp*]: *symp-on A* $\top$
by (*simp add*: *symp-on-def*)

lemma *sym-on-subset*: *sym-on B r* $\Longrightarrow$ $A \subseteq B \Longrightarrow$ *sym-on A r*
by (*auto simp*: *sym-on-def*)

lemma *symp-on-subset*: *symp-on B R* $\Longrightarrow$ $A \subseteq B \Longrightarrow$ *symp-on A R*
by (*auto simp*: *symp-on-def*)

lemma *symp-on-image*: *symp-on* $(f\ `\ A)\ R \longleftrightarrow$ *symp-on A* $(\lambda a\ b.\ R\ (f\ a)\ (f\ b))$
by (*simp add*: *symp-on-def*)

lemma *symp-on-equality*[*simp*]: *symp-on A* $(=)$
by (*simp add*: *symp-on-def*)

lemma *sym-Int*: *sym r* $\Longrightarrow$ *sym s* $\Longrightarrow$ *sym* $(r \cap s)$
by (*fast intro*: *symI elim*: *symE*)

lemma *symp-inf*: *symp r* $\Longrightarrow$ *symp s* $\Longrightarrow$ *symp* $(r \sqcap s)$
by (*fact sym-Int* [*to-pred*])

lemma *sym-Un*: *sym r* $\Longrightarrow$ *sym s* $\Longrightarrow$ *sym* $(r \cup s)$
by (*fast intro*: *symI elim*: *symE*)

lemma *symp-sup*: *symp r* $\Longrightarrow$ *symp s* $\Longrightarrow$ *symp* $(r \sqcup s)$
by (*fact sym-Un* [*to-pred*])

lemma *sym-INTER*: $\forall x{\in}S.\ sym\ (r\ x) \Longrightarrow sym\ (\bigcap (r\ `\ S))$
by (*fast intro*: *symI elim*: *symE*)

lemma *symp-INF*: $\forall x{\in}S.\ symp\ (r\ x) \Longrightarrow symp\ (\bigsqcap (r\ `\ S))$
by (*fact sym-INTER* [*to-pred*])

lemma *sym-UNION*: $\forall x{\in}S.\ sym\ (r\ x) \Longrightarrow sym\ (\bigcup (r\ `\ S))$
by (*fast intro*: *symI elim*: *symE*)

lemma *symp-SUP*: $\forall x{\in}S.\ symp\ (r\ x) \Longrightarrow symp\ (\bigsqcup (r\ `\ S))$
by (*fact sym-UNION* [*to-pred*])

19.2.5 Antisymmetry

definition *antisym-on* :: $'a\ set \Rightarrow 'a\ rel \Rightarrow bool$ **where**
$antisym\text{-}on\ A\ r \longleftrightarrow (\forall x \in A.\ \forall y \in A.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r \longrightarrow x = y)$

abbreviation *antisym* :: $'a\ rel \Rightarrow bool$ **where**
$antisym \equiv antisym\text{-}on\ UNIV$

definition *antisymp-on* :: $'a\ set \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$antisymp\text{-}on\ A\ R \longleftrightarrow (\forall x \in A.\ \forall y \in A.\ R\ x\ y \longrightarrow R\ y\ x \longrightarrow x = y)$

abbreviation *antisymp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$antisymp \equiv antisymp\text{-}on\ UNIV$

lemma *antisym-def*[*no-atp*]: $antisym\ r \longleftrightarrow (\forall x\ y.\ (x,\ y) \in r \longrightarrow (y,\ x) \in r \longrightarrow x = y)$
by (*simp add*: *antisym-on-def*)

lemma *antisymp-def*[*no-atp*]: $antisymp\ R \longleftrightarrow (\forall x\ y.\ R\ x\ y \longrightarrow R\ y\ x \longrightarrow x = y)$
by (*simp add*: *antisymp-on-def*)

antisym-def and *antisymp-def* are for backward compatibility.

lemma *antisymp-on-antisym-on-eq*[*pred-set-conv*]:
$antisymp\text{-}on\ A\ (\lambda x\ y.\ (x,\ y) \in r) \longleftrightarrow antisym\text{-}on\ A\ r$
by (*simp add*: *antisym-on-def antisymp-on-def*)

lemmas *antisymp-antisym-eq* = *antisymp-on-antisym-on-eq*[*of UNIV*] — For backward compatibility

lemma *antisym-onI*:
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y) \Longrightarrow antisym\text{-}on\ A\ r$
unfolding *antisym-on-def* **by** *simp*

lemma *antisymI* [*intro?*]:

$(\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y) \Longrightarrow antisym\ r$
by (*simp add*: *antisym-onI*)

lemma *antisymp-onI*:
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y) \Longrightarrow antisymp\text{-}on\ A\ R$
by (*rule antisym-onI*[*to-pred*])

lemma *antisympI* [*intro?*]:
$(\bigwedge x\ y.\ R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y) \Longrightarrow antisymp\ R$
by (*rule antisymI*[*to-pred*])

lemma *antisym-onD*:
$antisym\text{-}on\ A\ r \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y$
by (*simp add*: *antisym-on-def*)

lemma *antisymD* [*dest?*]:
$antisym\ r \Longrightarrow (x,\ y) \in r \Longrightarrow (y,\ x) \in r \Longrightarrow x = y$
by (*simp add*: *antisym-onD*)

lemma *antisymp-onD*:
$antisymp\text{-}on\ A\ R \Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y$
by (*rule antisym-onD*[*to-pred*])

lemma *antisympD* [*dest?*]:
$antisymp\ R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x \Longrightarrow x = y$
by (*rule antisymD*[*to-pred*])

lemma *antisym-on-bot*[*simp*]: $antisym\text{-}on\ A\ \bot$
by (*simp add*: *antisym-on-def*)

lemma *antisymp-on-bot*[*simp*]: $antisymp\text{-}on\ A\ \bot$
using *antisym-on-bot*[*to-pred*] **.**

lemma *antisymp-on-mono-stronger*:
fixes
$A :: 'a\ set$ **and** $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and**
$B :: 'b\ set$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow bool$ **and**
$f :: 'a \Rightarrow 'b$
assumes $antisymp\text{-}on\ B\ Q$ **and** $f\ `\ A \subseteq B$ **and**
Q-imp-R: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow Q\ (f\ x)\ (f\ y)$ **and**
inj-f: $inj\text{-}on\ f\ A$
shows $antisymp\text{-}on\ A\ R$
proof (*rule antisymp-onI*)
fix $x\ y :: 'a$
assume $x \in A$ **and** $y \in A$ **and** $R\ x\ y$ **and** $R\ y\ x$
hence $Q\ (f\ x)\ (f\ y)$ **and** $Q\ (f\ y)\ (f\ x)$
using *Q-imp-R* **by** *iprover+*
moreover have $f\ x \in B$ **and** $f\ y \in B$
using $\langle f\ `\ A \subseteq B\rangle$ $\langle x \in A\rangle$ $\langle y \in A\rangle$ **by** *blast+*

ultimately have $f\ x = f\ y$
using ‹*antisymp-on B Q*›[*THEN antisymp-onD*] **by** *iprover*
thus $x = y$
using *inj-f*[*THEN inj-onD*] ‹$x \in A$› ‹$y \in A$› **by** *iprover*
qed

lemma *antisymp-on-mono-strong*:
antisymp-on B Q $\Longrightarrow$ $A \subseteq B \Longrightarrow$ ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow Q\ x\ y$) $\Longrightarrow$ *antisymp-on A R*
using *antisymp-on-mono-stronger*[*of B Q* $\lambda x.\ x\ A\ R$, *OF - - - inj-on-id2*, *unfolded image-ident*] **.**

lemma *antisymp-on-mono*[*mono*]: $A \subseteq B \Longrightarrow R \leq Q \Longrightarrow$ *antisymp-on B Q* $\leq$ *antisymp-on A R*
by (*simp add*: *antisymp-on-mono-strong le-fun-def*)

lemma *antisym-on-subset*: *antisym-on B r* $\Longrightarrow$ $A \subseteq B \Longrightarrow$ *antisym-on A r*
by (*auto simp*: *antisym-on-def*)

lemma *antisymp-on-subset*: *antisymp-on B R* $\Longrightarrow$ $A \subseteq B \Longrightarrow$ *antisymp-on A R*
using *antisymp-on-mono-strong* **.**

lemma *antisymp-on-image*:
assumes *inj-on f A*
shows *antisymp-on* (*f* ‘ *A*) *R* $\longleftrightarrow$ *antisymp-on A* ($\lambda a\ b.\ R\ (f\ a)\ (f\ b)$)
using *assms* **by** (*auto simp*: *antisymp-on-def inj-on-def*)

lemma *antisym-subset*:
$r \subseteq s \Longrightarrow$ *antisym s* $\Longrightarrow$ *antisym r*
unfolding *antisym-def* **by** *blast*

lemma *antisymp-less-eq*:
$r \leq s \Longrightarrow$ *antisymp s* $\Longrightarrow$ *antisymp r*
by (*fact antisym-subset* [*to-pred*])

lemma *antisymp-on-equality*[*simp*]: *antisymp-on A* (=)
by (*auto intro*: *antisymp-onI*)

lemma *antisym-singleton* [*simp*]:
antisym $\{x\}$
by (*blast intro*: *antisymI*)

lemma *antisym-on-if-asym-on*: *asym-on A r* $\Longrightarrow$ *antisym-on A r*
by (*auto intro*: *antisym-onI dest*: *asym-onD*)

lemma *antisymp-on-if-asymp-on*: *asymp-on A R* $\Longrightarrow$ *antisymp-on A R*
by (*rule antisym-on-if-asym-on*[*to-pred*])

lemma (**in** *preorder*) *antisymp-on-less*[*simp*]: *antisymp-on A* (<)

by (*rule antisymp-on-if-asymp-on*[*OF asymp-on-less*])

lemma (**in** *preorder*) *antisymp-on-greater*[*simp*]: *antisymp-on A* (>)
by (*rule antisymp-on-if-asymp-on*[*OF asymp-on-greater*])

lemma (**in** *order*) *antisymp-on-le*[*simp*]: *antisymp-on A* (≤)
by (*simp add*: *antisymp-onI*)

lemma (**in** *order*) *antisymp-on-ge*[*simp*]: *antisymp-on A* (≥)
by (*simp add*: *antisymp-onI*)

19.2.6 Transitivity

definition *trans-on* :: *'a set* ⇒ *'a rel* ⇒ *bool* **where**
trans-on A r ⟷ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. ∀ *z* ∈ *A*. (*x*, *y*) ∈ *r* ⟶ (*y*, *z*) ∈ *r* ⟶ (*x*, *z*) ∈ *r*)

abbreviation *trans* :: *'a rel* ⇒ *bool* **where**
trans ≡ *trans-on UNIV*

definition *transp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
transp-on A R ⟷ (∀ *x* ∈ *A*. ∀ *y* ∈ *A*. ∀ *z* ∈ *A*. *R x y* ⟶ *R y z* ⟶ *R x z*)

abbreviation *transp* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
transp ≡ *transp-on UNIV*

lemma *trans-def*[*no-atp*]: *trans r* ⟷ (∀ *x y z*. (*x*, *y*) ∈ *r* ⟶ (*y*, *z*) ∈ *r* ⟶ (*x*, *z*) ∈ *r*)
by (*simp add*: *trans-on-def*)

lemma *transp-def*: *transp R* ⟷ (∀ *x y z*. *R x y* ⟶ *R y z* ⟶ *R x z*)
by (*simp add*: *transp-on-def*)

trans-def and *transp-def* are for backward compatibility.

lemma *transp-on-trans-on-eq*[*pred-set-conv*]: *transp-on A* (λ*x y*. (*x*, *y*) ∈ *r*) ⟷ *trans-on A r*
by (*simp add*: *trans-on-def transp-on-def*)

lemmas *transp-trans-eq* = *transp-on-trans-on-eq*[*of UNIV*] — For backward compatibility

lemma *trans-onI*:
(⋀*x y z*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *z* ∈ *A* ⟹ (*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*) ⟹
trans-on A r
unfolding *trans-on-def*
by (*intro ballI*) *iprover*

lemma *transI* [*intro?*]: (⋀*x y z*. (*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*) ⟹

trans r
by (*rule trans-onI*)

lemma *transp-onI*:
(⋀*x y z*. *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *z* ∈ *A* ⟹ *R x y* ⟹ *R y z* ⟹ *R x z*) ⟹ *transp-on A R*
by (*rule trans-onI*[*to-pred*])

lemma *transpI* [*intro?*]: (⋀*x y z*. *R x y* ⟹ *R y z* ⟹ *R x z*) ⟹ *transp R*
by (*rule transI*[*to-pred*])

lemma *transE*:
assumes *trans r* **and** (*x*, *y*) ∈ *r* **and** (*y*, *z*) ∈ *r*
obtains (*x*, *z*) ∈ *r*
using *assms* **by** (*unfold trans-def*) *iprover*

lemma *transpE*:
assumes *transp r* **and** *r x y* **and** *r y z*
obtains *r x z*
using *assms* **by** (*rule transE* [*to-pred*])

lemma *trans-onD*:
trans-on A r ⟹ *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *z* ∈ *A* ⟹ (*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*
unfolding *trans-on-def*
by (*elim ballE*) *iprover*+

lemma *transD*[*dest?*]: *trans r* ⟹ (*x*, *y*) ∈ *r* ⟹ (*y*, *z*) ∈ *r* ⟹ (*x*, *z*) ∈ *r*
by (*simp add*: *trans-onD*[*of UNIV r x y z*])

lemma *transp-onD*: *transp-on A R* ⟹ *x* ∈ *A* ⟹ *y* ∈ *A* ⟹ *z* ∈ *A* ⟹ *R x y* ⟹ *R y z* ⟹ *R x z*
by (*rule trans-onD*[*to-pred*])

lemma *transpD*[*dest?*]: *transp R* ⟹ *R x y* ⟹ *R y z* ⟹ *R x z*
by (*rule transD*[*to-pred*])

lemma *trans-on-subset*: *trans-on B r* ⟹ *A* ⊆ *B* ⟹ *trans-on A r*
by (*auto simp*: *trans-on-def*)

lemma *transp-on-subset*: *transp-on B R* ⟹ *A* ⊆ *B* ⟹ *transp-on A R*
by (*auto simp*: *transp-on-def*)

lemma *transp-on-image*: *transp-on* (*f* ‘ *A*) *R* ⟷ *transp-on A* (λ*a b*. *R* (*f a*) (*f b*))
by (*simp add*: *transp-on-def*)

lemma *trans-Int*: *trans r* ⟹ *trans s* ⟹ *trans* (*r* ∩ *s*)
by (*fast intro*: *transI elim*: *transE*)

lemma *transp-inf*: *transp r* $\Longrightarrow$ *transp s* $\Longrightarrow$ *transp* $(r \sqcap s)$
by (*fact trans-Int* [*to-pred*])

lemma *trans-INTER*: $\forall x{\in}S.$ *trans* $(r\ x) \Longrightarrow$ *trans* $(\bigcap(r\ `\ S))$
by (*fast intro*: *transI elim*: *transD*)

lemma *transp-INF*: $\forall x{\in}S.$ *transp* $(r\ x) \Longrightarrow$ *transp* $(\bigsqcap(r\ `\ S))$
by (*fact trans-INTER* [*to-pred*])

lemma *trans-on-join* [*code*]:
trans-on A r $\longleftrightarrow$ $(\forall(x,\ y1) \in r.\ x \in A \longrightarrow y1 \in A \longrightarrow$
$(\forall(y2,\ z) \in r.\ y1 = y2 \longrightarrow z \in A \longrightarrow (x,\ z) \in r))$
by (*auto simp*: *trans-on-def*)

lemma *trans-join*: *trans r* $\longleftrightarrow$ $(\forall(x,\ y1) \in r.\ \forall(y2,\ z) \in r.\ y1 = y2 \longrightarrow (x,\ z) \in r)$
by (*auto simp add*: *trans-def*)

lemma *transp-trans*: *transp r* $\longleftrightarrow$ *trans* $\{(x,\ y).\ r\ x\ y\}$
by (*simp add*: *trans-def transp-def*)

lemma *transp-on-equality*[*simp*]: *transp-on A* $(=)$
by (*auto intro*: *transp-onI*)

lemma *trans-on-bot*[*simp*]: *trans-on A* $\bot$
by (*simp add*: *trans-on-def*)

lemma *transp-on-bot*[*simp*]: *transp-on A* $\bot$
using *trans-on-bot*[*to-pred*] **.**

lemma *trans-on-top*[*simp*]: *trans-on A* $\top$
by (*simp add*: *trans-on-def*)

lemma *transp-on-top*[*simp*]: *transp-on A* $\top$
by (*simp add*: *transp-on-def*)

lemma *transp-empty* [*simp*]: *transp* $(\lambda x\ y.\ False)$
using *transp-on-bot* **unfolding** *bot-fun-def bot-bool-def* **.**

lemma *trans-singleton* [*simp*]: *trans* $\{(a,\ a)\}$
by (*blast intro*: *transI*)

lemma *transp-singleton* [*simp*]: *transp* $(\lambda x\ y.\ x = a \wedge y = a)$
by (*simp add*: *transp-def*)

lemma *asym-on-iff-irrefl-on-if-trans-on*: *trans-on A r* $\Longrightarrow$ *asym-on A r* $\longleftrightarrow$ *irrefl-on A r*
by (*auto intro*: *irrefl-on-if-asym-on dest*: *trans-onD irrefl-onD*)

lemma *asymp-on-iff-irreflp-on-if-transp-on*: *transp-on A R* $\Longrightarrow$ *asymp-on A R* $\longleftrightarrow$ *irreflp-on A R*
by (*rule asym-on-iff-irrefl-on-if-trans-on*[*to-pred*])

lemma (**in** *preorder*) *transp-on-le*[*simp*]: *transp-on A* ($\leq$)
by (*auto intro*: *transp-onI order-trans*)

lemma (**in** *preorder*) *transp-on-less*[*simp*]: *transp-on A* ($<$)
by (*auto intro*: *transp-onI less-trans*)

lemma (**in** *preorder*) *transp-on-ge*[*simp*]: *transp-on A* ($\geq$)
by (*auto intro*: *transp-onI order-trans*)

lemma (**in** *preorder*) *transp-on-greater*[*simp*]: *transp-on A* ($>$)
by (*auto intro*: *transp-onI less-trans*)

19.2.7 Totality

definition *total-on* :: *'a set* $\Rightarrow$ *'a rel* $\Rightarrow$ *bool* **where**
total-on A r $\longleftrightarrow$ ($\forall x{\in}A.\ \forall y{\in}A.\ x \neq y \longrightarrow (x,\ y) \in r \vee (y,\ x) \in r$)

abbreviation *total* :: *'a rel* $\Rightarrow$ *bool* **where**
total $\equiv$ *total-on UNIV*

definition *totalp-on* :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
totalp-on A R $\longleftrightarrow$ ($\forall x \in A.\ \forall y \in A.\ x \neq y \longrightarrow R\ x\ y \vee R\ y\ x$)

abbreviation *totalp* :: (*'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
totalp $\equiv$ *totalp-on UNIV*

lemma *totalp-on-total-on-eq*[*pred-set-conv*]: *totalp-on A* ($\lambda x\ y.\ (x,\ y) \in r$) $\longleftrightarrow$ *total-on A r*
by (*simp add*: *totalp-on-def total-on-def*)

lemma *total-onI* [*intro?*]:
($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow (x,\ y) \in r \vee (y,\ x) \in r$) $\Longrightarrow$ *total-on A r*
unfolding *total-on-def* **by** *blast*

lemma *totalI*: ($\bigwedge x\ y.\ x \neq y \Longrightarrow (x,\ y) \in r \vee (y,\ x) \in r$) $\Longrightarrow$ *total r*
by (*rule total-onI*)

lemma *totalp-onI*: ($\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow R\ x\ y \vee R\ y\ x$) $\Longrightarrow$ *totalp-on A R*
by (*rule total-onI*[*to-pred*])

lemma *totalpI*: ($\bigwedge x\ y.\ x \neq y \Longrightarrow R\ x\ y \vee R\ y\ x$) $\Longrightarrow$ *totalp R*
by (*rule totalI*[*to-pred*])

lemma *totalp-onD*:
totalp-on A R $\Longrightarrow$ $x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow R\ x\ y \vee R\ y\ x$
by (*simp add*: *totalp-on-def*)

lemma *totalpD*: *totalp R* $\Longrightarrow$ $x \neq y \Longrightarrow R\ x\ y \vee R\ y\ x$
by (*simp add*: *totalp-onD*)

lemma *total-on-top*[*simp*]: *total-on A* $\top$
by (*simp add*: *total-on-def*)

lemma *totalp-on-top*[*simp*]: *totalp-on A* $\top$
by (*simp add*: *totalp-on-def*)

lemma *totalp-on-mono-stronger*:
fixes
$A :: 'a\ set$ **and** $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and**
$B :: 'b\ set$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow bool$ **and**
$f :: 'a \Rightarrow 'b$
assumes *totalp-on B Q* **and** $f\ `\ A \subseteq B$ **and**
Q-imp-R: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow Q\ (f\ x)\ (f\ y) \Longrightarrow R\ x\ y$ **and**
inj-f: *inj-on f A*
shows *totalp-on A R*
proof (*rule totalp-onI*)
fix $x\ y :: 'a$
assume $x \in A$ **and** $y \in A$ **and** $x \neq y$
hence $f\ x \in B$ **and** $f\ y \in B$ **and** $f\ x \neq f\ y$
using ‹$f\ `\ A \subseteq B$› *inj-f* **by** (*auto dest*: *inj-onD*)
hence $Q\ (f\ x)\ (f\ y) \vee Q\ (f\ y)\ (f\ x)$
using ‹*totalp-on B Q*› **by** (*iprover dest*: *totalp-onD*)
thus $R\ x\ y \vee R\ y\ x$
using *Q-imp-R* ‹$x \in A$› ‹$y \in A$› **by** *iprover*
qed

lemma *totalp-on-mono-stronger-alt*:
fixes
$A :: 'a\ set$ **and** $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and**
$B :: 'b\ set$ **and** $Q :: 'b \Rightarrow 'b \Rightarrow bool$ **and**
$f :: 'b \Rightarrow 'a$
assumes *totalp-on B Q* **and** $A \subseteq f\ `\ B$ **and**
Q-imp-R: $\bigwedge x\ y.\ x \in B \Longrightarrow y \in B \Longrightarrow Q\ x\ y \Longrightarrow R\ (f\ x)\ (f\ y)$
shows *totalp-on A R*
proof (*rule totalp-onI*)
fix $x\ y :: 'a$
assume $x \in A$ **and** $y \in A$ **and** $x \neq y$
then obtain $x'\ y'$ **where** $x' \in B$ **and** $x = f\ x'$ **and** $y' \in B$ **and** $y = f\ y'$ **and** $x' \neq y'$
using ‹$A \subseteq f\ `\ B$› **by** *blast*
hence $Q\ x'\ y' \vee Q\ y'\ x'$
using ‹*totalp-on B Q*›[*THEN totalp-onD*] **by** *blast*

hence $R\ (f\ x')\ (f\ y') \vee R\ (f\ y')\ (f\ x')$
using *Q-imp-R* ‹$x' \in B$› ‹$y' \in B$› **by** *blast*
thus $R\ x\ y \vee R\ y\ x$
using ‹$x = f\ x'$› ‹$y = f\ y'$› **by** *blast*
qed

lemma *totalp-on-mono-strong*:
totalp-on B Q $\Longrightarrow$ $A \subseteq B \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow Q\ x\ y \Longrightarrow R\ x\ y) \Longrightarrow$ *totalp-on A R*
using *totalp-on-mono-stronger*[*of B Q* $\lambda x.\ x\ A\ R$, *simplified*] **.**

lemma *totalp-on-mono*[*mono*]: $A \subseteq B \Longrightarrow Q \leq R \Longrightarrow$ *totalp-on B Q* $\leq$ *totalp-on A R*
by (*auto intro*: *totalp-on-mono-strong*)

lemma *total-on-subset*: *total-on B r* $\Longrightarrow A \subseteq B \Longrightarrow$ *total-on A r*
by (*auto simp*: *total-on-def*)

lemma *totalp-on-subset*: *totalp-on B R* $\Longrightarrow A \subseteq B \Longrightarrow$ *totalp-on A R*
using *totalp-on-mono-strong* **.**

lemma *totalp-on-image*:
assumes *inj-on f A*
shows *totalp-on* (*f* ‘ *A*) *R* $\longleftrightarrow$ *totalp-on A* ($\lambda a\ b.\ R\ (f\ a)\ (f\ b)$)
using *assms* **by** (*auto simp*: *totalp-on-def inj-on-def*)

lemma *total-on-empty* [*simp*]: *total-on* {} *r*
by (*simp add*: *total-on-def*)

lemma *totalp-on-empty* [*simp*]: *totalp-on* {} *R*
by (*simp add*: *totalp-on-def*)

lemma *total-on-singleton* [*simp*]: *total-on* {*x*} *r*
by (*simp add*: *total-on-def*)

lemma *totalp-on-singleton* [*simp*]: *totalp-on* {*x*} *R*
by (*simp add*: *totalp-on-def*)

lemma (**in** *linorder*) *totalp-on-less*[*simp*]: *totalp-on A* ($<$)
by (*auto intro*: *totalp-onI*)

lemma (**in** *linorder*) *totalp-on-greater*[*simp*]: *totalp-on A* ($>$)
by (*auto intro*: *totalp-onI*)

lemma (**in** *linorder*) *totalp-on-le*[*simp*]: *totalp-on A* ($\leq$)
by (*rule totalp-onI*, *rule linear*)

lemma (**in** *linorder*) *totalp-on-ge*[*simp*]: *totalp-on A* ($\geq$)
by (*rule totalp-onI*, *rule linear*)

19.2.8 Left uniqueness

definition *left-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
$left\text{-}unique\ R \longleftrightarrow (\forall x\ y\ z.\ R\ x\ z \longrightarrow R\ y\ z \longrightarrow x = y)$

lemma *left-uniqueI*: $(\bigwedge x\ y\ z.\ A\ x\ z \Longrightarrow A\ y\ z \Longrightarrow x = y) \Longrightarrow left\text{-}unique\ A$
unfolding *left-unique-def* **by** *blast*

lemma *left-uniqueD*: $left\text{-}unique\ A \Longrightarrow A\ x\ z \Longrightarrow A\ y\ z \Longrightarrow x = y$
unfolding *left-unique-def* **by** *blast*

lemma *left-unique-iff-Uniq*: $left\text{-}unique\ r \longleftrightarrow (\forall y.\ \exists_{\leq 1} x.\ r\ x\ y)$
unfolding *Uniq-def left-unique-def* **by** *blast*

lemma *left-unique-bot*[*simp*]: $left\text{-}unique\ \bot$
by (*simp add*: *left-unique-def*)

lemma *left-unique-mono-strong*:
$left\text{-}unique\ Q \Longrightarrow (\bigwedge x\ y.\ R\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow left\text{-}unique\ R$
by (*rule left-uniqueI*) (*auto dest*: *left-uniqueD*)

lemma *left-unique-mono*[*mono*]: $R \leq Q \Longrightarrow left\text{-}unique\ Q \leq left\text{-}unique\ R$
using *left-unique-mono-strong*[*of Q R*]
by (*simp add*: *le-fun-def*)

19.2.9 Right uniqueness

definition *single-valued* :: $('a \times 'b)\ set \Rightarrow bool$
where $single\text{-}valued\ r \longleftrightarrow (\forall x\ y.\ (x, y) \in r \longrightarrow (\forall z.\ (x, z) \in r \longrightarrow y = z))$

definition *right-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
$right\text{-}unique\ R \longleftrightarrow (\forall x\ y\ z.\ R\ x\ y \longrightarrow R\ x\ z \longrightarrow y = z)$

lemma *right-unique-single-valued-eq* [*pred-set-conv*]:
$right\text{-}unique\ (\lambda x\ y.\ (x, y) \in r) \longleftrightarrow single\text{-}valued\ r$
by (*simp add*: *single-valued-def right-unique-def*)

lemma *right-unique-iff-Uniq*:
$right\text{-}unique\ r \longleftrightarrow (\forall x.\ \exists_{\leq 1} y.\ r\ x\ y)$
unfolding *Uniq-def right-unique-def* **by** *auto*

lemma *single-valuedI*:
$(\bigwedge x\ y.\ (x, y) \in r \Longrightarrow (\bigwedge z.\ (x, z) \in r \Longrightarrow y = z)) \Longrightarrow single\text{-}valued\ r$
unfolding *single-valued-def* **by** *blast*

lemma *right-uniqueI*: $(\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ x\ z \Longrightarrow y = z) \Longrightarrow right\text{-}unique\ R$
unfolding *right-unique-def* **by** *fast*

lemma *single-valuedD*:
$single\text{-}valued\ r \Longrightarrow (x, y) \in r \Longrightarrow (x, z) \in r \Longrightarrow y = z$

by (*simp add*: *single-valued-def*)

lemma *right-uniqueD*: *right-unique* $R \Longrightarrow R\ x\ y \Longrightarrow R\ x\ z \Longrightarrow y = z$
unfolding *right-unique-def* **by** *fast*

lemma *single-valued-empty* [*simp*]:
single-valued {}
by (*simp add*: *single-valued-def*)

lemma *right-unique-bot*[*simp*]: *right-unique* $\bot$
by (*fact single-valued-empty*[*to-pred*])

lemma *right-unique-mono-strong*:
right-unique $Q \Longrightarrow (\bigwedge x\ y.\ R\ x\ y \Longrightarrow Q\ x\ y) \Longrightarrow$ *right-unique* R
by (*rule right-uniqueI*) (*auto dest*: *right-uniqueD*)

lemma *right-unique-mono*[*mono*]: $R \leq Q \Longrightarrow$ *right-unique* $Q \leq$ *right-unique* R
using *right-unique-mono-strong*[*of Q R*]
by (*simp add*: *le-fun-def*)

lemma *single-valued-subset*:
$r \subseteq s \Longrightarrow$ *single-valued* $s \Longrightarrow$ *single-valued* r
unfolding *single-valued-def* **by** *blast*

lemma *right-unique-less-eq*: $r \leq s \Longrightarrow$ *right-unique* $s \Longrightarrow$ *right-unique* r
by (*fact single-valued-subset* [*to-pred*])

19.3 Relation operations

19.3.1 The identity relation

definition *Id* :: $'a$ *rel*
where $Id = \{p.\ \exists x.\ p = (x, x)\}$

lemma *IdI* [*intro*]: $(a, a) \in Id$
by (*simp add*: *Id-def*)

lemma *IdE* [*elim*!]: $p \in Id \Longrightarrow (\bigwedge x.\ p = (x, x) \Longrightarrow P) \Longrightarrow P$
unfolding *Id-def* **by** (*iprover elim*: *CollectE*)

lemma *pair-in-Id-conv* [*iff*]: $(a, b) \in Id \longleftrightarrow a = b$
unfolding *Id-def* **by** *blast*

lemma *refl-Id*: *refl Id*
by (*simp add*: *refl-on-def*)

lemma *antisym-Id*: *antisym Id*
— A strange result, since *Id* is also symmetric.
by (*simp add*: *antisym-def*)

lemma *sym-Id*: *sym Id*
by (*simp add*: *sym-def*)

lemma *trans-Id*: *trans Id*
by (*simp add*: *trans-def*)

lemma *single-valued-Id* [*simp*]: *single-valued Id*
by (*unfold single-valued-def*) *blast*

lemma *irrefl-diff-Id* [*simp*]: *irrefl* (*r* − *Id*)
by (*simp add*: *irrefl-def*)

lemma *trans-on-diff-Id*: *trans-on A r* ⟹ *antisym-on A r* ⟹ *trans-on A* (*r* − *Id*)
by (*blast intro*: *trans-onI dest*: *trans-onD antisym-onD*)

lemma *trans-diff-Id*[*no-atp*]: *trans r* ⟹ *antisym r* ⟹ *trans* (*r* − *Id*)
using *trans-on-diff-Id* **.**

lemma *total-on-diff-Id* [*simp*]: *total-on A* (*r* − *Id*) = *total-on A r*
by (*simp add*: *total-on-def*)

lemma *Id-fstsnd-eq*: *Id* = {*x*. *fst x* = *snd x*}
by *force*

19.3.2 Diagonal: identity over a set

definition *Id-on* :: *'a set* ⇒ *'a rel*
where *Id-on A* = (⋃*x*∈*A*. {(*x*, *x*)})

lemma *Id-on-empty* [*simp*]: *Id-on* {} = {}
by (*simp add*: *Id-on-def*)

lemma *Id-on-eqI*: *a* = *b* ⟹ *a* ∈ *A* ⟹ (*a*, *b*) ∈ *Id-on A*
by (*simp add*: *Id-on-def*)

lemma *Id-onI* [*intro*!]: *a* ∈ *A* ⟹ (*a*, *a*) ∈ *Id-on A*
by (*rule Id-on-eqI*) (*rule refl*)

lemma *Id-onE* [*elim*!]: *c* ∈ *Id-on A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *c* = (*x*, *x*) ⟹ *P*) ⟹ *P*
— The general elimination rule.
unfolding *Id-on-def* **by** (*iprover elim*!: *UN-E singletonE*)

lemma *Id-on-iff*: (*x*, *y*) ∈ *Id-on A* ⟷ *x* = *y* ∧ *x* ∈ *A*
by *blast*

lemma *Id-on-def'* [*nitpick-unfold*]: *Id-on* {*x*. *A x*} = *Collect* (λ(*x*, *y*). *x* = *y* ∧ *A x*)
by *auto*

lemma *Id-on-subset-Times*: *Id-on A* $\subseteq$ *A* $\times$ *A*
by *blast*

lemma *refl-on-Id-on*: *refl-on A* (*Id-on A*)
by (*rule refl-onI*[*OF Id-onI*])

lemma *antisym-Id-on* [*simp*]: *antisym* (*Id-on A*)
unfolding *antisym-def* **by** *blast*

lemma *sym-Id-on* [*simp*]: *sym* (*Id-on A*)
by (*rule symI*) *clarify*

lemma *trans-Id-on* [*simp*]: *trans* (*Id-on A*)
by (*fast intro*: *transI elim*: *transD*)

lemma *single-valued-Id-on* [*simp*]: *single-valued* (*Id-on A*)
unfolding *single-valued-def* **by** *blast*

19.3.3 Composition

inductive-set *relcomp* :: ('*a* $\times$ '*b*) *set* $\Rightarrow$ ('*b* $\times$ '*c*) *set* $\Rightarrow$ ('*a* $\times$ '*c*) *set*
for *r* :: ('*a* $\times$ '*b*) *set* **and** *s* :: ('*b* $\times$ '*c*) *set*
where *relcompI* [*intro*]: (*a*, *b*) $\in$ *r* $\Longrightarrow$ (*b*, *c*) $\in$ *s* $\Longrightarrow$ (*a*, *c*) $\in$ *relcomp r s*

open-bundle *relcomp-syntax*
begin
notation *relcomp* (**infixr** ‹*O*› *75*) **and** *relcompp* (**infixr** ‹*OO*› *75*)
end

lemmas *relcomppI* = *relcompp.intros*

For historic reasons, the elimination rules are not wholly corresponding. Feel free to consolidate this.

inductive-cases *relcompEpair*: (*a*, *c*) $\in$ *r O s*
inductive-cases *relcomppE* [*elim*!]: (*r OO s*) *a c*

lemma *relcompE* [*elim*!]: *xz* $\in$ *r O s* $\Longrightarrow$
(⋀*x y z*. *xz* = (*x*, *z*) $\Longrightarrow$ (*x*, *y*) $\in$ *r* $\Longrightarrow$ (*y*, *z*) $\in$ *s* $\Longrightarrow$ *P*) $\Longrightarrow$ *P*
apply (*cases xz*)
apply *simp*
apply (*erule relcompEpair*)
apply *iprover*
done

lemma *R-O-Id* [*simp*]: *R O Id* = *R*
by *fast*

lemma *Id-O-R* [*simp*]: *Id O R* = *R*

by *fast*

lemma *relcomp-empty1* [*simp*]: $\{\}\ O\ R = \{\}$
by *blast*

lemma *relcompp-bot1* [*simp*]: $\bot\ OO\ R = \bot$
by (*fact relcomp-empty1* [*to-pred*])

lemma *relcomp-empty2* [*simp*]: $R\ O\ \{\} = \{\}$
by *blast*

lemma *relcompp-bot2* [*simp*]: $R\ OO\ \bot = \bot$
by (*fact relcomp-empty2* [*to-pred*])

lemma *O-assoc*: $(R\ O\ S)\ O\ T = R\ O\ (S\ O\ T)$
by *blast*

lemma *relcompp-assoc*: $(r\ OO\ s)\ OO\ t = r\ OO\ (s\ OO\ t)$
by (*fact O-assoc* [*to-pred*])

lemma *trans-O-subset*: *trans* $r \Longrightarrow r\ O\ r \subseteq r$
by (*unfold trans-def*) *blast*

lemma *transp-relcompp-less-eq*: *transp* $r \Longrightarrow r\ OO\ r \leq r$
by (*fact trans-O-subset* [*to-pred*])

lemma *relcomp-mono*: $r' \subseteq r \Longrightarrow s' \subseteq s \Longrightarrow r'\ O\ s' \subseteq r\ O\ s$
by *blast*

lemma *relcompp-mono*: $r' \leq r \Longrightarrow s' \leq s \Longrightarrow r'\ OO\ s' \leq r\ OO\ s$
by (*fact relcomp-mono* [*to-pred*])

lemma *relcomp-subset-Sigma*: $r \subseteq A \times B \Longrightarrow s \subseteq B \times C \Longrightarrow r\ O\ s \subseteq A \times C$
by *blast*

lemma *relcomp-distrib* [*simp*]: $R\ O\ (S \cup T) = (R\ O\ S) \cup (R\ O\ T)$
by *auto*

lemma *relcompp-distrib* [*simp*]: $R\ OO\ (S \sqcup T) = R\ OO\ S \sqcup R\ OO\ T$
by (*fact relcomp-distrib* [*to-pred*])

lemma *relcomp-distrib2* [*simp*]: $(S \cup T)\ O\ R = (S\ O\ R) \cup (T\ O\ R)$
by *auto*

lemma *relcompp-distrib2* [*simp*]: $(S \sqcup T)\ OO\ R = S\ OO\ R \sqcup T\ OO\ R$
by (*fact relcomp-distrib2* [*to-pred*])

lemma *relcomp-UNION-distrib*: $s\ O\ \bigcup(r\ `\ I) = (\bigcup i{\in}I.\ s\ O\ r\ i)$
by *auto*

lemma *relcompp-SUP-distrib*: $s\ OO\ \bigsqcup (r\ `\ I) = (\bigsqcup i{\in}I.\ s\ OO\ r\ i)$
by (*fact relcomp-UNION-distrib* [*to-pred*])

lemma *relcomp-UNION-distrib2*: $\bigcup (r\ `\ I)\ O\ s = (\bigcup i{\in}I.\ r\ i\ O\ s)$
by *auto*

lemma *relcompp-SUP-distrib2*: $\bigsqcup (r\ `\ I)\ OO\ s = (\bigsqcup i{\in}I.\ r\ i\ OO\ s)$
by (*fact relcomp-UNION-distrib2* [*to-pred*])

lemma *single-valued-relcomp*: *single-valued* $r \Longrightarrow$ *single-valued* $s \Longrightarrow$ *single-valued* $(r\ O\ s)$
unfolding *single-valued-def* **by** *blast*

lemma *relcomp-unfold*: $r\ O\ s = \{(x,\ z).\ \exists y.\ (x,\ y) \in r \wedge (y,\ z) \in s\}$
by (*auto simp add*: *set-eq-iff*)

lemma *relcompp-apply*: $(R\ OO\ S)\ a\ c \longleftrightarrow (\exists b.\ R\ a\ b \wedge S\ b\ c)$
unfolding *relcomp-unfold* [*to-pred*] **..**

lemma *eq-OO*: $(=)\ OO\ R = R$
by *blast*

lemma *OO-eq*: $R\ OO\ (=) = R$
by *blast*

19.3.4 Converse

inductive-set *converse* :: $('a \times 'b)\ set \Rightarrow ('b \times 'a)\ set$
for r :: $('a \times 'b)\ set$
where $(a,\ b) \in r \Longrightarrow (b,\ a) \in converse\ r$

open-bundle *converse-syntax*
begin
notation
converse (‹(‹*notation*=‹*postfix* −1››-$^{-1}$)› [*1000*] *999*) **and**
conversep (‹(‹*notation*=‹*postfix* −1−1››-$^{-1-1}$)› [*1000*] *1000*)
notation (*ASCII*)
converse (‹(‹*notation*=‹*postfix* −1››-^−1)› [*1000*] *999*) **and**
conversep (‹(‹*notation*=‹*postfix* −1−1››-^−−1)› [*1000*] *1000*)
end

lemma *converseI* [*sym*]: $(a,\ b) \in r \Longrightarrow (b,\ a) \in r^{-1}$
by (*fact converse.intros*)

lemma *conversepI* : $r\ a\ b \Longrightarrow r^{-1-1}\ b\ a$
by (*fact conversep.intros*)

lemma *converseD* [*sym*]: $(a,\ b) \in r^{-1} \Longrightarrow (b,\ a) \in r$

by (*erule converse.cases*) *iprover*

lemma *conversepD* : $r^{-1-1}\ b\ a \Longrightarrow r\ a\ b$
by (*fact converseD* [*to-pred*])

lemma *converseE* [*elim*!]: $yx \in r^{-1} \Longrightarrow (\bigwedge x\ y.\ yx = (y,\ x) \Longrightarrow (x,\ y) \in r \Longrightarrow P) \Longrightarrow P$
— More general than *converseD*, as it "splits" the member of the relation.
apply (*cases yx*)
apply *simp*
apply (*erule converse.cases*)
apply *iprover*
done

lemmas *conversepE* [*elim*!] = *conversep.cases*

lemma *converse-iff* [*iff*]: $(a,\ b) \in r^{-1} \longleftrightarrow (b,\ a) \in r$
by (*auto intro*: *converseI*)

lemma *conversep-iff* [*iff*]: $r^{-1-1}\ a\ b = r\ b\ a$
by (*fact converse-iff* [*to-pred*])

lemma *converse-converse* [*simp*]: $(r^{-1})^{-1} = r$
by (*simp add*: *set-eq-iff*)

lemma *conversep-conversep* [*simp*]: $(r^{-1-1})^{-1-1} = r$
by (*fact converse-converse* [*to-pred*])

lemma *converse-empty*[*simp*]: $\{\}^{-1} = \{\}$
by *auto*

lemma *converse-UNIV*[*simp*]: $UNIV^{-1} = UNIV$
by *auto*

lemma *converse-relcomp*: $(r\ O\ s)^{-1} = s^{-1}\ O\ r^{-1}$
by *blast*

lemma *converse-relcompp*: $(r\ OO\ s)^{-1-1} = s^{-1-1}\ OO\ r^{-1-1}$
by (*iprover intro*: *order-antisym conversepI relcomppI elim*: *relcomppE dest*: *conversepD*)

lemma *converse-Int*: $(r \cap s)^{-1} = r^{-1} \cap s^{-1}$
by *blast*

lemma *converse-meet*: $(r \sqcap s)^{-1-1} = r^{-1-1} \sqcap s^{-1-1}$
by (*simp add*: *inf-fun-def*) (*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-Un*: $(r \cup s)^{-1} = r^{-1} \cup s^{-1}$
by *blast*

lemma *converse-join*: $(r \sqcup s)^{-1-1} = r^{-1-1} \sqcup s^{-1-1}$
by (*simp add*: *sup-fun-def*) (*iprover intro*: *conversepI ext dest*: *conversepD*)

lemma *converse-INTER*: $(\bigcap (r \mathbin{`} S))^{-1} = (\bigcap x{\in}S.\ (r\ x)^{-1})$
by *fast*

lemma *converse-UNION*: $(\bigcup (r \mathbin{`} S))^{-1} = (\bigcup x{\in}S.\ (r\ x)^{-1})$
by *blast*

lemma *converse-mono*[*simp*]: $r^{-1} \subseteq s^{-1} \longleftrightarrow r \subseteq s$
by *auto*

lemma *conversep-mono*[*simp*]: $r^{-1-1} \leq s^{-1-1} \longleftrightarrow r \leq s$
by (*fact converse-mono*[*to-pred*])

lemma *converse-inject*[*simp*]: $r^{-1} = s^{-1} \longleftrightarrow r = s$
by *auto*

lemma *conversep-inject*[*simp*]: $r^{-1-1} = s^{-1-1} \longleftrightarrow r = s$
by (*fact converse-inject*[*to-pred*])

lemma *converse-subset-swap*: $r \subseteq s^{-1} \longleftrightarrow r^{-1} \subseteq s$
by *auto*

lemma *conversep-le-swap*: $r \leq s^{-1-1} \longleftrightarrow r^{-1-1} \leq s$
by (*fact converse-subset-swap*[*to-pred*])

lemma *converse-Id* [*simp*]: $Id^{-1} = Id$
by *blast*

lemma *converse-Id-on* [*simp*]: $(Id\text{-}on\ A)^{-1} = Id\text{-}on\ A$
by *blast*

lemma *refl-on-converse* [*simp*]: *refl-on A* (r^{-1}) = *refl-on A r*
by (*auto simp*: *refl-on-def*)

lemma *reflp-on-conversp* [*simp*]: *reflp-on A* $R^{-1-1} \longleftrightarrow$ *reflp-on A R*
by (*auto simp*: *reflp-on-def*)

lemma *irrefl-on-converse* [*simp*]: *irrefl-on A* (r^{-1}) = *irrefl-on A r*
by (*simp add*: *irrefl-on-def*)

lemma *irreflp-on-converse* [*simp*]: *irreflp-on A* (r^{-1-1}) = *irreflp-on A r*
by (*rule irrefl-on-converse*[*to-pred*])

lemma *sym-on-converse* [*simp*]: *sym-on A* (r^{-1}) = *sym-on A r*
by (*auto intro*: *sym-onI dest*: *sym-onD*)

lemma *symp-on-conversep* [*simp*]: *symp-on A* R^{-1-1} = *symp-on A R*
by (*rule sym-on-converse*[*to-pred*])

lemma *asym-on-converse* [*simp*]: *asym-on A* (r^{-1}) = *asym-on A r*
by (*auto dest*: *asym-onD*)

lemma *asymp-on-conversep* [*simp*]: *asymp-on A* R^{-1-1} = *asymp-on A R*
by (*rule asym-on-converse*[*to-pred*])

lemma *antisym-on-converse* [*simp*]: *antisym-on A* (r^{-1}) = *antisym-on A r*
by (*auto intro*: *antisym-onI dest*: *antisym-onD*)

lemma *antisymp-on-conversep* [*simp*]: *antisymp-on A* R^{-1-1} = *antisymp-on A R*
by (*rule antisym-on-converse*[*to-pred*])

lemma *trans-on-converse* [*simp*]: *trans-on A* (r^{-1}) = *trans-on A r*
by (*auto intro*: *trans-onI dest*: *trans-onD*)

lemma *transp-on-conversep* [*simp*]: *transp-on A* R^{-1-1} = *transp-on A R*
by (*rule trans-on-converse*[*to-pred*])

lemma *sym-conv-converse-eq*: *sym r* $\longleftrightarrow$ $r^{-1} = r$
unfolding *sym-def* **by** *fast*

lemma *sym-Un-converse*: *sym* $(r \cup r^{-1})$
unfolding *sym-def* **by** *blast*

lemma *sym-Int-converse*: *sym* $(r \cap r^{-1})$
unfolding *sym-def* **by** *blast*

lemma *total-on-converse* [*simp*]: *total-on A* (r^{-1}) = *total-on A r*
by (*auto simp*: *total-on-def*)

lemma *totalp-on-converse* [*simp*]: *totalp-on A* R^{-1-1} = *totalp-on A R*
by (*rule total-on-converse*[*to-pred*])

lemma *left-unique-conversep*[*simp*]: *left-unique* A^{-1-1} $\longleftrightarrow$ *right-unique A*
by (*auto simp add*: *left-unique-def right-unique-def*)

lemma *right-unique-conversep*[*simp*]: *right-unique* A^{-1-1} $\longleftrightarrow$ *left-unique A*
by (*auto simp add*: *left-unique-def right-unique-def*)

lemma *conversep-noteq* [*simp*]: $(\neq)^{-1-1} = (\neq)$
by (*auto simp add*: *fun-eq-iff*)

lemma *conversep-eq* [*simp*]: $(=)^{-1-1} = (=)$
by (*auto simp add*: *fun-eq-iff*)

lemma *converse-unfold* [*code*]: $r^{-1} = \{(y, x).\ (x, y) \in r\}$

by (*simp add*: *set-eq-iff*)

19.3.5 Domain, range and field

inductive-set *Domain* :: ('*a* × '*b*) *set* ⇒ '*a set* **for** *r* :: ('*a* × '*b*) *set*
where *DomainI* [*intro*]: (*a*, *b*) ∈ *r* ⟹ *a* ∈ *Domain r*

lemmas *DomainPI* = *Domainp.DomainI*

inductive-cases *DomainE* [*elim*!]: *a* ∈ *Domain r*
inductive-cases *DomainpE* [*elim*!]: *Domainp r a*

inductive-set *Range* :: ('*a* × '*b*) *set* ⇒ '*b set* **for** *r* :: ('*a* × '*b*) *set*
where *RangeI* [*intro*]: (*a*, *b*) ∈ *r* ⟹ *b* ∈ *Range r*

lemmas *RangePI* = *Rangep.RangeI*

inductive-cases *RangeE* [*elim*!]: *b* ∈ *Range r*
inductive-cases *RangepE* [*elim*!]: *Rangep r b*

definition *Field* :: '*a rel* ⇒ '*a set*
where *Field r* = *Domain r* ∪ *Range r*

lemma *Field-iff*: *x* ∈ *Field r* ⟷ (∃ *y*. (*x*,*y*) ∈ *r* ∨ (*y*,*x*) ∈ *r*)
by (*auto simp*: *Field-def*)

lemma *FieldI1*: (*i*, *j*) ∈ *R* ⟹ *i* ∈ *Field R*
unfolding *Field-def* **by** *blast*

lemma *FieldI2*: (*i*, *j*) ∈ *R* ⟹ *j* ∈ *Field R*
unfolding *Field-def* **by** *auto*

lemma *Domain-fst* [*code*]: *Domain r* = *fst* ‘ *r*
by *force*

lemma *Range-snd* [*code*]: *Range r* = *snd* ‘ *r*
by *force*

lemma *fst-eq-Domain*: *fst* ‘ *R* = *Domain R*
by *force*

lemma *snd-eq-Range*: *snd* ‘ *R* = *Range R*
by *force*

lemma *range-fst* [*simp*]: *range fst* = *UNIV*
by (*auto simp*: *fst-eq-Domain*)

lemma *range-snd* [*simp*]: *range snd* = *UNIV*
by (*auto simp*: *snd-eq-Range*)

lemma *Domain-empty* [*simp*]: *Domain* {} = {}
by *auto*

lemma *Range-empty* [*simp*]: *Range* {} = {}
by *auto*

lemma *Field-empty* [*simp*]: *Field* {} = {}
by (*simp add*: *Field-def*)

lemma *Domain-empty-iff*: *Domain r* = {} ⟷ *r* = {}
by *auto*

lemma *Range-empty-iff*: *Range r* = {} ⟷ *r* = {}
by *auto*

lemma *Domain-insert* [*simp*]: *Domain* (*insert* (*a*, *b*) *r*) = *insert a* (*Domain r*)
by *blast*

lemma *Range-insert* [*simp*]: *Range* (*insert* (*a*, *b*) *r*) = *insert b* (*Range r*)
by *blast*

lemma *Field-insert* [*simp*]: *Field* (*insert* (*a*, *b*) *r*) = {*a*, *b*} ∪ *Field r*
by (*auto simp add*: *Field-def*)

lemma *Domain-iff*: *a* ∈ *Domain r* ⟷ (∃ *y*. (*a*, *y*) ∈ *r*)
by *blast*

lemma *Range-iff*: *a* ∈ *Range r* ⟷ (∃ *y*. (*y*, *a*) ∈ *r*)
by *blast*

lemma *Domain-Id* [*simp*]: *Domain Id* = *UNIV*
by *blast*

lemma *Range-Id* [*simp*]: *Range Id* = *UNIV*
by *blast*

lemma *Domain-Id-on* [*simp*]: *Domain* (*Id-on A*) = *A*
by *blast*

lemma *Range-Id-on* [*simp*]: *Range* (*Id-on A*) = *A*
by *blast*

lemma *Domain-Un-eq*: *Domain* (*A* ∪ *B*) = *Domain A* ∪ *Domain B*
by *blast*

lemma *Range-Un-eq*: *Range* (*A* ∪ *B*) = *Range A* ∪ *Range B*
by *blast*

lemma *Field-Un* [*simp*]: *Field* $(r \cup s) =$ *Field* $r \cup$ *Field* s
by (*auto simp*: *Field-def*)

lemma *Domain-Int-subset*: *Domain* $(A \cap B) \subseteq$ *Domain* $A \cap$ *Domain* B
by *blast*

lemma *Range-Int-subset*: *Range* $(A \cap B) \subseteq$ *Range* $A \cap$ *Range* B
by *blast*

lemma *Domain-Diff-subset*: *Domain* $A -$ *Domain* $B \subseteq$ *Domain* $(A - B)$
by *blast*

lemma *Range-Diff-subset*: *Range* $A -$ *Range* $B \subseteq$ *Range* $(A - B)$
by *blast*

lemma *Domain-Union*: *Domain* $(\bigcup S) = (\bigcup A{\in}S.$ *Domain* $A)$
by *blast*

lemma *Range-Union*: *Range* $(\bigcup S) = (\bigcup A{\in}S.$ *Range* $A)$
by *blast*

lemma *Field-Union* [*simp*]: *Field* $(\bigcup R) = \bigcup$ (*Field* ' R)
by (*auto simp*: *Field-def*)

lemma *Domain-converse* [*simp*]: *Domain* $(r^{-1}) =$ *Range* r
by *auto*

lemma *Range-converse* [*simp*]: *Range* $(r^{-1}) =$ *Domain* r
by *blast*

lemma *Field-converse* [*simp*]: *Field* $(r^{-1}) =$ *Field* r
by (*auto simp*: *Field-def*)

lemma *Domain-Collect-case-prod* [*simp*]: *Domain* $\{(x, y).\ P\ x\ y\} = \{x.\ \exists\, y.\ P\ x\ y\}$
by *auto*

lemma *Range-Collect-case-prod* [*simp*]: *Range* $\{(x, y).\ P\ x\ y\} = \{y.\ \exists\, x.\ P\ x\ y\}$
by *auto*

lemma *Domain-mono*: $r \subseteq s \Longrightarrow$ *Domain* $r \subseteq$ *Domain* s
by *blast*

lemma *Range-mono*: $r \subseteq s \Longrightarrow$ *Range* $r \subseteq$ *Range* s
by *blast*

lemma *mono-Field*: $r \subseteq s \Longrightarrow$ *Field* $r \subseteq$ *Field* s
by (*auto simp*: *Field-def Domain-def Range-def*)

lemma *Domain-unfold*: $Domain\ r = \{x.\ \exists y.\ (x,\ y) \in r\}$
by *blast*

lemma *Field-square* [*simp*]: $Field\ (x \times x) = x$
unfolding *Field-def* **by** *blast*

19.3.6 Image of a set under a relation

definition *Image* :: $('a \times 'b)\ set \Rightarrow 'a\ set \Rightarrow 'b\ set$ (**infixr** ‹‘›$\ 90$)
where $r\ ``\ s = \{y.\ \exists x{\in}s.\ (x,\ y) \in r\}$

lemma *Image-iff*: $b \in r``A \longleftrightarrow (\exists x{\in}A.\ (x,\ b) \in r)$
by (*simp add*: *Image-def*)

lemma *Image-singleton*: $r``\{a\} = \{b.\ (a,\ b) \in r\}$
by (*simp add*: *Image-def*)

lemma *Image-singleton-iff* [*iff*]: $b \in r``\{a\} \longleftrightarrow (a,\ b) \in r$
by (*rule Image-iff* [*THEN trans*]) *simp*

lemma *ImageI* [*intro*]: $(a,\ b) \in r \Longrightarrow a \in A \Longrightarrow b \in r``A$
unfolding *Image-def* **by** *blast*

lemma *ImageE* [*elim*!]: $b \in r\ ``\ A \Longrightarrow (\bigwedge x.\ (x,\ b) \in r \Longrightarrow x \in A \Longrightarrow P) \Longrightarrow P$
unfolding *Image-def* **by** (*iprover elim*!: *CollectE bexE*)

lemma *rev-ImageI*: $a \in A \Longrightarrow (a,\ b) \in r \Longrightarrow b \in r\ ``\ A$
— This version's more effective when we already have the required a
by *blast*

lemma *Image-empty1* [*simp*]: $\{\}\ ``\ X = \{\}$
by *auto*

lemma *Image-empty2* [*simp*]: $R``\{\} = \{\}$
by *blast*

lemma *Image-Id* [*simp*]: $Id\ ``\ A = A$
by *blast*

lemma *Image-Id-on* [*simp*]: $Id\text{-}on\ A\ ``\ B = A \cap B$
by *blast*

lemma *Image-Int-subset*: $R\ ``\ (A \cap B) \subseteq R\ ``\ A \cap R\ ``\ B$
by *blast*

lemma *Image-Int-eq*: $single\text{-}valued\ (converse\ R) \Longrightarrow R\ ``\ (A \cap B) = R\ ``\ A \cap R\ ``\ B$
by (*auto simp*: *single-valued-def*)

lemma *Image-Un*: R `` $(A \cup B) = R$ `` $A \cup R$ `` B
by *blast*

lemma *Un-Image*: $(R \cup S)$ `` $A = R$ `` $A \cup S$ `` A
by *blast*

lemma *Image-subset*: $r \subseteq A \times B \Longrightarrow r``C \subseteq B$
by (*iprover intro*!: *subsetI elim*!: *ImageE dest*!: *subsetD SigmaD2*)

lemma *Image-eq-UN*: $r``B = (\bigcup y{\in}\, B.\ r``\{y\})$
— NOT suitable for rewriting
by *blast*

lemma *Image-mono*: $r' \subseteq r \Longrightarrow A' \subseteq A \Longrightarrow (r' `` A') \subseteq (r `` A)$
by *blast*

lemma *Image-UN*: r `` $(\bigcup (B ` A)) = (\bigcup x{\in}A.\ r$ `` $(B\ x))$
by *blast*

lemma *UN-Image*: $(\bigcup i{\in}I.\ X\ i)$ `` $S = (\bigcup i{\in}I.\ X\ i$ `` $S)$
by *auto*

lemma *Image-INT-subset*: $(r$ `` $(\bigcap (B ` A))) \subseteq (\bigcap x{\in}A.\ r$ `` $(B\ x))$
by *blast*

Converse inclusion requires some assumptions

lemma *Image-INT-eq*:
assumes *single-valued* (r^{-1})
and $A \neq \{\}$
shows r `` $(\bigcap (B ` A)) = (\bigcap x{\in}A.\ r$ `` $B\ x)$
proof(*rule equalityI*, *rule Image-INT-subset*)
show $(\bigcap x{\in}A.\ r$ `` $B\ x) \subseteq r$ `` $\bigcap\ (B ` A)$
proof
fix x
assume $x \in (\bigcap x{\in}A.\ r$ `` $B\ x)$
then show $x \in r$ `` $\bigcap\ (B ` A)$
using *assms* **unfolding** *single-valued-def* **by** *simp blast*
qed
qed

lemma *Image-subset-eq*: $r``A \subseteq B \longleftrightarrow A \subseteq -\ ((r^{-1})$ `` $(-\ B))$
by *blast*

lemma *Image-Collect-case-prod* [*simp*]: $\{(x,\ y).\ P\ x\ y\}$ `` $A = \{y.\ \exists x{\in}A.\ P\ x\ y\}$
by *auto*

lemma *Sigma-Image*: $(SIGMA\ x{:}A.\ B\ x)$ `` $X = (\bigcup x{\in}X \cap A.\ B\ x)$
by *auto*

lemma *relcomp-Image*: $(X\ O\ Y)\ ``\ Z = Y\ ``\ (X\ ``\ Z)$
by *auto*

19.3.7 Inverse image

definition *inv-image* :: $'b\ rel \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a\ rel$
where $inv\text{-}image\ r\ f = \{(x,\ y).\ (f\ x,\ f\ y) \in r\}$

definition *inv-imagep* :: $('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where $inv\text{-}imagep\ r\ f = (\lambda x\ y.\ r\ (f\ x)\ (f\ y))$

lemma [*pred-set-conv*]: $inv\text{-}imagep\ (\lambda x\ y.\ (x,\ y) \in r)\ f = (\lambda x\ y.\ (x,\ y) \in inv\text{-}image\ r\ f)$
by (*simp add*: *inv-image-def inv-imagep-def*)

lemma *sym-inv-image*: $sym\ r \Longrightarrow sym\ (inv\text{-}image\ r\ f)$
unfolding *sym-def inv-image-def* **by** *blast*

lemma *trans-inv-image*: $trans\ r \Longrightarrow trans\ (inv\text{-}image\ r\ f)$
unfolding *trans-def inv-image-def*
by (*simp* (*no-asm*)) *blast*

lemma *total-inv-image*: $[\![inj\ f;\ total\ r]\!] \Longrightarrow total\ (inv\text{-}image\ r\ f)$
unfolding *inv-image-def total-on-def* **by** (*auto simp*: *inj-eq*)

lemma *asym-inv-image*: $asym\ R \Longrightarrow asym\ (inv\text{-}image\ R\ f)$
by (*simp add*: *inv-image-def asym-iff*)

lemma *in-inv-image*[*simp*]: $(x,\ y) \in inv\text{-}image\ r\ f \longleftrightarrow (f\ x,\ f\ y) \in r$
by (*auto simp*: *inv-image-def*)

lemma *converse-inv-image*[*simp*]: $(inv\text{-}image\ R\ f)^{-1} = inv\text{-}image\ (R^{-1})\ f$
unfolding *inv-image-def converse-unfold* **by** *auto*

lemma *in-inv-imagep* [*simp*]: $inv\text{-}imagep\ r\ f\ x\ y = r\ (f\ x)\ (f\ y)$
by (*simp add*: *inv-imagep-def*)

19.3.8 Powerset

definition *Powp* :: $('a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where $Powp\ A = (\lambda B.\ \forall x \in B.\ A\ x)$

lemma *Powp-Pow-eq* [*pred-set-conv*]: $Powp\ (\lambda x.\ x \in A) = (\lambda x.\ x \in Pow\ A)$
by (*auto simp add*: *Powp-def fun-eq-iff*)

lemmas *Powp-mono* [*mono*] = *Pow-mono* [*to-pred*]

end

20 Finite sets

theory *Finite-Set*
  imports *Product-Type Sum-Type Fields Relation*
begin

20.1 Predicate for finite sets

context notes [[*inductive-internals*]]
begin

inductive *finite* :: *'a set* ⇒ *bool*
  where
    emptyI [*simp*, *intro*!]: *finite* {}
  | *insertI* [*simp*, *intro*!]: *finite A* ⟹ *finite* (*insert a A*)

end

simproc-setup *finite-Collect* (*finite* (*Collect P*)) = ‹*K Set-Comprehension-Pointfree.proc*›

declare [[*simproc del*: *finite-Collect*]]

lemma *finite-induct* [*case-names empty insert*, *induct set*: *finite*]:
  — Discharging $x \notin F$ entails extra work.
  assumes *finite F*
  assumes *P* {}
    and *insert*: ⋀*x F*. *finite F* ⟹ *x* ∉ *F* ⟹ *P F* ⟹ *P* (*insert x F*)
  shows *P F*
  using ‹*finite F*›
proof *induct*
  show *P* {} **by** *fact*
next
  fix *x F*
  assume *F*: *finite F* **and** *P*: *P F*
  show *P* (*insert x F*)
  proof *cases*
    assume *x* ∈ *F*
    then have *insert x F* = *F* **by** (*rule insert-absorb*)
    with *P* **show** *?thesis* **by** (*simp only*:)
  next
    assume *x* ∉ *F*
    from *F this P* **show** *?thesis* **by** (*rule insert*)
  qed
qed

lemma *infinite-finite-induct* [*case-names infinite empty insert*]:
  assumes *infinite*: ⋀*A*. ¬ *finite A* ⟹ *P A*
    and *empty*: *P* {}
    and *insert*: ⋀*x F*. *finite F* ⟹ *x* ∉ *F* ⟹ *P F* ⟹ *P* (*insert x F*)
  shows *P A*

proof (*cases finite A*)
case *False*
with *infinite* **show** *?thesis* **.**
next
case *True*
then show *?thesis* **by** (*induct A*) (*fact empty insert*)+
qed

20.1.1 Choice principles

lemma *ex-new-if-finite*: — does not depend on def of finite at all
assumes ¬ *finite* (*UNIV* :: *'a set*) **and** *finite A*
shows ∃ *a*::*'a*. *a* ∉ *A*
proof −
from *assms* **have** *A* ≠ *UNIV* **by** *blast*
then show *?thesis* **by** *blast*
qed

A finite choice principle. Does not need the SOME choice operator.

lemma *finite-set-choice*: *finite A* ⟹ ∀ *x*∈*A*. ∃ *y*. *P x y* ⟹ ∃ *f*. ∀ *x*∈*A*. *P x* (*f x*)
proof (*induct rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
then obtain *f b* **where** *f*: ∀ *x*∈*A*. *P x* (*f x*) **and** *ab*: *P a b*
by *auto*
show *?case* (**is** ∃ *f*. *?P f*)
proof
show *?P* (λ*x*. *if x* = *a then b else f x*)
using *f ab* **by** *auto*
qed
qed

20.1.2 Finite sets are the images of initial segments of natural numbers

lemma *finite-imp-nat-seg-image-inj-on*:
assumes *finite A*
shows ∃ (*n*::*nat*) *f*. *A* = *f* ' {*i*. *i* < *n*} ∧ *inj-on f* {*i*. *i* < *n*}
using *assms*
proof *induct*
case *empty*
show *?case*
proof
show ∃ *f*. {} = *f* ' {*i*::*nat*. *i* < *0*} ∧ *inj-on f* {*i*. *i* < *0*}
by *simp*
qed
next

case (*insert a A*)
have *notinA*: $a \notin A$ **by** *fact*
from *insert.hyps* **obtain** *n f* **where** $A = f$ ' $\{i$::*nat*. $i < n\}$ *inj-on f* $\{i.\ i < n\}$
by *blast*
then have *insert a A* = $f(n:=a)$ ' $\{i.\ i <$ *Suc n*$\}$ **and** *inj-on* $(f(n:=a))$ $\{i.\ i <$ *Suc n*$\}$
using *notinA* **by** (*auto simp add*: *image-def Ball-def inj-on-def less-Suc-eq*)
then show *?case* **by** *blast*
qed

lemma *nat-seg-image-imp-finite*: $A = f$ ' $\{i$::*nat*. $i < n\} \Longrightarrow$ *finite A*
proof (*induct n arbitrary*: *A*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
let *?B* = f ' $\{i.\ i < n\}$
have *finB*: *finite ?B* **by** (*rule Suc.hyps*[*OF refl*])
show *?case*
proof (*cases* $\exists\, k{<}n.\ f\ n = f\ k$)
case *True*
then have *A* = *?B*
using *Suc.prems* **by** (*auto simp*:*less-Suc-eq*)
then show *?thesis*
using *finB* **by** *simp*
next
case *False*
then have *A* = *insert* (*f n*) *?B*
using *Suc.prems* **by** (*auto simp*:*less-Suc-eq*)
then show *?thesis* **using** *finB* **by** *simp*
qed
qed

lemma *finite-conv-nat-seg-image*: *finite A* $\longleftrightarrow$ $(\exists\, n\ f.\ A = f$ ' $\{i$::*nat*. $i < n\})$
by (*blast intro*: *nat-seg-image-imp-finite dest*: *finite-imp-nat-seg-image-inj-on*)

lemma *finite-imp-inj-to-nat-seg*:
assumes *finite A*
shows $\exists f\ n.\ f$ ' $A = \{i$::*nat*. $i < n\} \wedge$ *inj-on f A*
proof −
from *finite-imp-nat-seg-image-inj-on* [*OF* ‹*finite A*›]
obtain *f* **and** *n* :: *nat* **where** *bij*: *bij-betw f* $\{i.\ i{<}n\}$ *A*
by (*auto simp*: *bij-betw-def*)
let *?f* = *the-inv-into* $\{i.\ i{<}n\}$ *f*
have *inj-on ?f A* $\wedge$ *?f* ' *A* = $\{i.\ i{<}n\}$
by (*fold bij-betw-def*) (*rule bij-betw-the-inv-into*[*OF bij*])
then show *?thesis* **by** *blast*
qed

lemma *finite-Collect-less-nat* [*iff*]: *finite* {*n*::*nat*. *n* < *k*}
by (*fastforce simp*: *finite-conv-nat-seg-image*)

lemma *finite-Collect-le-nat* [*iff*]: *finite* {*n*::*nat*. *n* ≤ *k*}
by (*simp add*: *le-eq-less-or-eq Collect-disj-eq*)

20.2 Finiteness and common set operations

lemma *rev-finite-subset*: *finite B* ⟹ *A* ⊆ *B* ⟹ *finite A*
proof (*induct arbitrary*: *A rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F A*)
have *A*: *A* ⊆ *insert x F* **and** *r*: *A* − {*x*} ⊆ *F* ⟹ *finite* (*A* − {*x*})
by *fact+*
show *finite A*
proof *cases*
assume *x*: *x* ∈ *A*
with *A* **have** *A* − {*x*} ⊆ *F* **by** (*simp add*: *subset-insert-iff*)
with *r* **have** *finite* (*A* − {*x*}) **.**
then have *finite* (*insert x* (*A* − {*x*})) **..**
also have *insert x* (*A* − {*x*}) = *A*
using *x* **by** (*rule insert-Diff*)
finally show *?thesis* **.**
next
show *?thesis* **when** *A* ⊆ *F*
using *that* **by** *fact*
assume *x* ∉ *A*
with *A* **show** *A* ⊆ *F*
by (*simp add*: *subset-insert-iff*)
qed
qed

lemma *finite-subset*: *A* ⊆ *B* ⟹ *finite B* ⟹ *finite A*
by (*rule rev-finite-subset*)

simproc-setup *finite* (*finite A*) = ‹
let
val finite-subset = @{*thm finite-subset*}
val Eq-TrueI = @{*thm Eq-TrueI*}

fun is-subset A th = *case Thm.prop-of th of*
(- $ ***Const-*** ‹*less-eq* ***Type*** ‹*set* -› *for A′ B*›)
=> *if A aconv A′ then SOME*(*B*,*th*) *else NONE*
| - => *NONE*;

fun is-finite th = *case Thm.prop-of th of*
(- $ ***Const-*** ‹*finite* - *for A*›) => *SOME*(*A*,*th*)

```
    | - => NONE;

 fun comb (A,sub-th) (A',fin-th) ths = if A aconv A' then (sub-th,fin-th) :: ths else
ths

 fun proc ctxt ct =
   (let
      val - $ A = Thm.term-of ct
      val prems = Simplifier.prems-of ctxt
      val fins = map-filter is-finite prems
      val subsets = map-filter (is-subset A) prems
    in case fold-product comb subsets fins [] of
          (sub-th,fin-th) :: - => SOME((fin-th RS (sub-th RS finite-subset)) RS
Eq-TrueI)
       | - => NONE
    end)
in K proc end
›
```

declare [[*simproc del*: *finite*]]

lemma *finite-UnI*:
  assumes *finite F* **and** *finite G*
  shows *finite* (*F* ∪ *G*)
  using *assms* **by** *induct simp-all*

lemma *finite-Un* [*iff*]: *finite* (*F* ∪ *G*) ⟷ *finite F* ∧ *finite G*
  by (*blast intro*: *finite-UnI finite-subset* [*of* - *F* ∪ *G*])

lemma *finite-insert* [*simp*]: *finite* (*insert a A*) ⟷ *finite A*
proof −
  have *finite* {*a*} ∧ *finite A* ⟷ *finite A* **by** *simp*
  then have *finite* ({*a*} ∪ *A*) ⟷ *finite A* **by** (*simp only*: *finite-Un*)
  then show *?thesis* **by** *simp*
qed

lemma *finite-Int* [*simp*, *intro*]: *finite F* ∨ *finite G* ⟹ *finite* (*F* ∩ *G*)
  by (*blast intro*: *finite-subset*)

lemma *finite-Collect-conjI* [*simp*, *intro*]:
  finite {*x*. *P x*} ∨ *finite* {*x*. *Q x*} ⟹ *finite* {*x*. *P x* ∧ *Q x*}
  by (*simp add*: *Collect-conj-eq*)

lemma *finite-Collect-disjI* [*simp*]:
  finite {*x*. *P x* ∨ *Q x*} ⟷ *finite* {*x*. *P x*} ∧ *finite* {*x*. *Q x*}
  by (*simp add*: *Collect-disj-eq*)

lemma *finite-Diff* [*simp*, *intro*]: *finite A* ⟹ *finite* (*A* − *B*)

by (*rule finite-subset*, *rule Diff-subset*)

lemma *finite-Diff2* [*simp*]:
assumes *finite B*
shows *finite* $(A - B) \longleftrightarrow$ *finite A*
proof −
have *finite A* $\longleftrightarrow$ *finite* $((A - B) \cup (A \cap B))$
by (*simp add*: *Un-Diff-Int*)
also have ... $\longleftrightarrow$ *finite* $(A - B)$
using ‹*finite B*› **by** *simp*
finally show *?thesis* **..**
qed

lemma *finite-Diff-insert* [*iff*]: *finite* $(A - \mathit{insert}\ a\ B) \longleftrightarrow$ *finite* $(A - B)$
proof −
have *finite* $(A - B) \longleftrightarrow$ *finite* $(A - B - \{a\})$ **by** *simp*
moreover have $A - \mathit{insert}\ a\ B = A - B - \{a\}$ **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *finite-compl* [*simp*]:
finite $(A :: 'a\ \mathit{set}) \Longrightarrow$ *finite* $(-A) \longleftrightarrow$ *finite* $(\mathit{UNIV} :: 'a\ \mathit{set})$
by (*simp add*: *Compl-eq-Diff-UNIV*)

lemma *finite-Collect-not* [*simp*]:
finite $\{x :: 'a.\ P\ x\} \Longrightarrow$ *finite* $\{x.\ \neg\ P\ x\} \longleftrightarrow$ *finite* $(\mathit{UNIV} :: 'a\ \mathit{set})$
by (*simp add*: *Collect-neg-eq*)

lemma *finite-Union* [*simp*, *intro*]:
finite A $\Longrightarrow (\bigwedge M.\ M \in A \Longrightarrow$ *finite M*$) \Longrightarrow$ *finite* $(\bigcup A)$
by (*induct rule*: *finite-induct*) *simp-all*

lemma *finite-UN-I* [*intro*]:
finite A $\Longrightarrow (\bigwedge a.\ a \in A \Longrightarrow$ *finite* $(B\ a)) \Longrightarrow$ *finite* $(\bigcup a{\in}A.\ B\ a)$
by (*induct rule*: *finite-induct*) *simp-all*

lemma *finite-UN* [*simp*]: *finite A* $\Longrightarrow$ *finite* $(\bigcup (B\ `\ A)) \longleftrightarrow (\forall x{\in}A.$ *finite* $(B\ x))$
by (*blast intro*: *finite-subset*)

lemma *finite-Inter* [*intro*]: $\exists A{\in}M.$ *finite A* $\Longrightarrow$ *finite* $(\bigcap M)$
by (*blast intro*: *Inter-lower finite-subset*)

lemma *finite-INT* [*intro*]: $\exists x{\in}I.$ *finite* $(A\ x) \Longrightarrow$ *finite* $(\bigcap x{\in}I.\ A\ x)$
by (*blast intro*: *INT-lower finite-subset*)

lemma *finite-imageI* [*simp*, *intro*]: *finite F* $\Longrightarrow$ *finite* $(h\ `\ F)$
by (*induct rule*: *finite-induct*) *simp-all*

lemma *finite-image-set* [*simp*]: *finite* $\{x.\ P\ x\} \Longrightarrow$ *finite* $\{f\ x\ |x.\ P\ x\}$

by (*simp add*: *image-Collect* [*symmetric*])

lemma *finite-image-set2*:
finite {*x*. *P x*} ⟹ *finite* {*y*. *Q y*} ⟹ *finite* {*f x y* |*x y*. *P x* ∧ *Q y*}
by (*rule finite-subset* [**where** *B* = ⋃*x* ∈ {*x*. *P x*}. ⋃*y* ∈ {*y*. *Q y*}. {*f x y*}]) *auto*

lemma *finite-imageD*:
assumes *finite* (*f* ‘ *A*) **and** *inj-on f A*
shows *finite A*
using *assms*
proof (*induct f* ‘ *A arbitrary*: *A*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x B*)
then have *B-A*: *insert x B* = *f* ‘ *A*
by *simp*
then obtain *y* **where** *x* = *f y* **and** *y* ∈ *A*
by *blast*
from *B-A* ‹*x* ∉ *B*› **have** *B* = *f* ‘ *A* − {*x*}
by *blast*
with *B-A* ‹*x* ∉ *B*› ‹*x* = *f y*› ‹*inj-on f A*› ‹*y* ∈ *A*› **have** *B* = *f* ‘ (*A* − {*y*})
by (*simp add*: *inj-on-image-set-diff*)
moreover from ‹*inj-on f A*› **have** *inj-on f* (*A* − {*y*})
by (*rule inj-on-diff*)
ultimately have *finite* (*A* − {*y*})
by (*rule insert.hyps*)
then show *finite A*
by *simp*
qed

lemma *finite-image-iff*: *inj-on f A* ⟹ *finite* (*f* ‘ *A*) ⟷ *finite A*
using *finite-imageD* **by** *blast*

lemma *finite-surj*: *finite A* ⟹ *B* ⊆ *f* ‘ *A* ⟹ *finite B*
by (*erule finite-subset*) (*rule finite-imageI*)

lemma *finite-range-imageI*: *finite* (*range g*) ⟹ *finite* (*range* (λ*x*. *f* (*g x*)))
by (*drule finite-imageI*) (*simp add*: *range-composition*)

lemma *finite-subset-image*:
assumes *finite B*
shows *B* ⊆ *f* ‘ *A* ⟹ ∃ *C*⊆*A*. *finite C* ∧ *B* = *f* ‘ *C*
using *assms*
proof (*induct B rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert b B*)

then have $\exists\, C{\subseteq}A.$ *finite* $C \wedge B = f\ `\ C$
by *auto*
then show *?case*
using *insert.prems* **by** *auto*
qed

lemma *all-subset-image*: $(\forall\, B.\ B \subseteq f\ `\ A \longrightarrow P\ B) \longleftrightarrow (\forall\, B.\ B \subseteq A \longrightarrow P(f\ `\ B))$
by (*auto simp*: *subset-image-iff dest*: *image-mono*)

lemma *all-finite-subset-image*:
$(\forall\, B.$ *finite* $B \wedge B \subseteq f\ `\ A \longrightarrow P\ B) \longleftrightarrow (\forall\, B.$ *finite* $B \wedge B \subseteq A \longrightarrow P\ (f\ `\ B))$
proof *safe*
fix $B :: 'a$ *set*
assume B: *finite* B $B \subseteq f\ `\ A$ **and** $P$: $\forall\, B.$ *finite* $B \wedge B \subseteq A \longrightarrow P\ (f\ `\ B)$
show $P\ B$
using *finite-subset-image* [*OF* B] P **by** *blast*
qed *blast*

lemma *ex-finite-subset-image*:
$(\exists\, B.$ *finite* $B \wedge B \subseteq f\ `\ A \wedge P\ B) \longleftrightarrow (\exists\, B.$ *finite* $B \wedge B \subseteq A \wedge P\ (f\ `\ B))$
proof *safe*
fix $B :: 'a$ *set*
assume B: *finite* B $B \subseteq f\ `\ A$ **and** $P\ B$
show $\exists\, B.$ *finite* $B \wedge B \subseteq A \wedge P\ (f\ `\ B)$
using *finite-subset-image* [*OF* B] ‹$P\ B$› **by** *blast*
qed *blast*

lemma *finite-vimage-IntI*: *finite* $F \Longrightarrow$ *inj-on* $h\ A \Longrightarrow$ *finite* $(h -` F \cap A)$
proof (*induct rule*: *finite-induct*)
case (*insert* $x\ F$)
then show *?case*
by (*simp add*: *vimage-insert* [*of* $h\ x\ F$] *finite-subset* [*OF inj-on-vimage-singleton*] *Int-Un-distrib2*)
qed *simp*

lemma *finite-finite-vimage-IntI*:
assumes *finite* F
and $\bigwedge y.\ y \in F \Longrightarrow$ *finite* $((h -` \{y\}) \cap A)$
shows *finite* $(h -` F \cap A)$
proof −
have $*$: $h -` F \cap A = (\bigcup\ y{\in}F.\ (h -` \{y\}) \cap A)$
by *blast*
show *?thesis*
by (*simp only*: $*$ *assms finite-UN-I*)
qed

lemma *finite-vimageI*: *finite* $F \Longrightarrow$ *inj* $h \Longrightarrow$ *finite* $(h -` F)$
using *finite-vimage-IntI*[*of* F h *UNIV*] **by** *auto*

lemma *finite-vimageD'*: *finite* $(f -` A) \Longrightarrow A \subseteq \mathit{range}\ f \Longrightarrow \mathit{finite}\ A$
by (*auto simp add*: *subset-image-iff intro*: *finite-subset*[*rotated*])

lemma *finite-vimageD*: *finite* $(h -` F) \Longrightarrow \mathit{surj}\ h \Longrightarrow \mathit{finite}\ F$
by (*auto dest*: *finite-vimageD'*)

lemma *finite-vimage-iff*: *bij h* $\Longrightarrow$ *finite* $(h -` F) \longleftrightarrow$ *finite F*
unfolding *bij-def* **by** (*auto elim*: *finite-vimageD finite-vimageI*)

lemma *finite-inverse-image-gen*:
assumes *finite A inj-on f D*
shows *finite* $\{j{\in}D.\ f\ j \in A\}$
using *finite-vimage-IntI* [*OF assms*]
by (*simp add*: *Collect-conj-eq inf-commute vimage-def*)

lemma *finite-inverse-image*:
assumes *finite A inj f*
shows *finite* $\{j.\ f\ j \in A\}$
using *finite-inverse-image-gen* [*OF assms*] **by** *simp*

lemma *finite-Collect-bex* [*simp*]:
assumes *finite A*
shows *finite* $\{x.\ \exists\, y{\in}A.\ Q\ x\ y\} \longleftrightarrow (\forall\, y{\in}A.\ \mathit{finite}\ \{x.\ Q\ x\ y\})$
proof −
have $\{x.\ \exists\, y{\in}A.\ Q\ x\ y\} = (\bigcup y{\in}A.\ \{x.\ Q\ x\ y\})$ **by** *auto*
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *finite-Collect-bounded-ex* [*simp*]:
assumes *finite* $\{y.\ P\ y\}$
shows *finite* $\{x.\ \exists\, y.\ P\ y \wedge Q\ x\ y\} \longleftrightarrow (\forall\, y.\ P\ y \longrightarrow \mathit{finite}\ \{x.\ Q\ x\ y\})$
proof −
have $\{x.\ \exists\, y.\ P\ y \wedge Q\ x\ y\} = (\bigcup y{\in}\{y.\ P\ y\}.\ \{x.\ Q\ x\ y\})$
by *auto*
with *assms* **show** *?thesis*
by *simp*
qed

lemma *finite-Plus*: *finite A* $\Longrightarrow$ *finite B* $\Longrightarrow$ *finite* $(A <+> B)$
by (*simp add*: *Plus-def*)

lemma *finite-PlusD*:
fixes $A :: 'a\ \mathit{set}$ **and** $B :: 'b\ \mathit{set}$
assumes *fin*: *finite* $(A <+> B)$
shows *finite A finite B*
proof −
have *Inl* ` $A \subseteq A <+> B$
by *auto*

then have *finite* (*Inl* ' *A* :: ('*a* + '*b*) *set*)
using *fin* **by** (*rule finite-subset*)
then show *finite A*
by (*rule finite-imageD*) (*auto intro*: *inj-onI*)
next
have *Inr* ' *B* ⊆ *A* <+> *B*
by *auto*
then have *finite* (*Inr* ' *B* :: ('*a* + '*b*) *set*)
using *fin* **by** (*rule finite-subset*)
then show *finite B*
by (*rule finite-imageD*) (*auto intro*: *inj-onI*)
qed

lemma *finite-Plus-iff* [*simp*]: *finite* (*A* <+> *B*) ⟷ *finite A* ∧ *finite B*
by (*auto intro*: *finite-PlusD finite-Plus*)

lemma *finite-Plus-UNIV-iff* [*simp*]:
finite (*UNIV* :: ('*a* + '*b*) *set*) ⟷ *finite* (*UNIV* :: '*a set*) ∧ *finite* (*UNIV* :: '*b set*)
by (*subst UNIV-Plus-UNIV* [*symmetric*]) (*rule finite-Plus-iff*)

lemma *finite-SigmaI* [*simp*, *intro*]:
finite A ⟹ (⋀*a*. *a*∈*A* ⟹ *finite* (*B a*)) ⟹ *finite* (*SIGMA a*:*A*. *B a*)
unfolding *Sigma-def* **by** *blast*

lemma *finite-SigmaI2*:
assumes *finite* {*x*∈*A*. *B x* ≠ {}}
and ⋀*a*. *a* ∈ *A* ⟹ *finite* (*B a*)
shows *finite* (*Sigma A B*)
proof −
from *assms* **have** *finite* (*Sigma* {*x*∈*A*. *B x* ≠ {}} *B*)
by *auto*
also have *Sigma* {*x*:*A*. *B x* ≠ {}} *B* = *Sigma A B*
by *auto*
finally show *?thesis* **.**
qed

lemma *finite-cartesian-product*: *finite A* ⟹ *finite B* ⟹ *finite* (*A* × *B*)
by (*rule finite-SigmaI*)

lemma *finite-Prod-UNIV*:
finite (*UNIV* :: '*a set*) ⟹ *finite* (*UNIV* :: '*b set*) ⟹ *finite* (*UNIV* :: ('*a* × '*b*) *set*)
by (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product*)

lemma *finite-cartesian-productD1*:
assumes *finite* (*A* × *B*) **and** *B* ≠ {}
shows *finite A*
proof −

from *assms* **obtain** *n f* **where** $A \times B = f \mathbin{`} \{i::nat.\ i < n\}$
by (*auto simp add*: *finite-conv-nat-seg-image*)
then have $fst \mathbin{`} (A \times B) = fst \mathbin{`} f \mathbin{`} \{i::nat.\ i < n\}$
by *simp*
with ‹$B \neq \{\}$› **have** $A = (fst \circ f) \mathbin{`} \{i::nat.\ i < n\}$
by (*simp add*: *image-comp*)
then have $\exists\, n\, f.\ A = f \mathbin{`} \{i::nat.\ i < n\}$
by *blast*
then show *?thesis*
by (*auto simp add*: *finite-conv-nat-seg-image*)
qed

lemma *finite-cartesian-productD2*:
assumes *finite* $(A \times B)$ **and** $A \neq \{\}$
shows *finite B*
proof −
from *assms* **obtain** *n f* **where** $A \times B = f \mathbin{`} \{i::nat.\ i < n\}$
by (*auto simp add*: *finite-conv-nat-seg-image*)
then have $snd \mathbin{`} (A \times B) = snd \mathbin{`} f \mathbin{`} \{i::nat.\ i < n\}$
by *simp*
with ‹$A \neq \{\}$› **have** $B = (snd \circ f) \mathbin{`} \{i::nat.\ i < n\}$
by (*simp add*: *image-comp*)
then have $\exists\, n\, f.\ B = f \mathbin{`} \{i::nat.\ i < n\}$
by *blast*
then show *?thesis*
by (*auto simp add*: *finite-conv-nat-seg-image*)
qed

lemma *finite-cartesian-product-iff*:
finite $(A \times B) \longleftrightarrow (A = \{\} \vee B = \{\} \vee (finite\ A \wedge finite\ B))$
by (*auto dest*: *finite-cartesian-productD1 finite-cartesian-productD2 finite-cartesian-product*)

lemma *finite-prod*:
finite $(UNIV :: ('a \times 'b)\ set) \longleftrightarrow finite\ (UNIV :: 'a\ set) \wedge finite\ (UNIV :: 'b\ set)$
using *finite-cartesian-product-iff*[*of UNIV UNIV*] **by** *simp*

lemma *finite-Pow-iff* [*iff*]: *finite* $(Pow\ A) \longleftrightarrow finite\ A$
proof
assume *finite* $(Pow\ A)$
then have *finite* $((\lambda x.\ \{x\}) \mathbin{`} A)$
by (*blast intro*: *finite-subset*)
then show *finite A*
by (*rule finite-imageD* [*unfolded inj-on-def*]) *simp*
next
assume *finite A*
then show *finite* $(Pow\ A)$
by *induct* (*simp-all add*: *Pow-insert*)
qed

corollary *finite-Collect-subsets* [*simp*, *intro*]: *finite A* $\Longrightarrow$ *finite* $\{B.\ B \subseteq A\}$
by (*simp add*: *Pow-def* [*symmetric*])

lemma *finite-set*: *finite* (*UNIV* :: *'a set set*) $\longleftrightarrow$ *finite* (*UNIV* :: *'a set*)
by (*simp only*: *finite-Pow-iff Pow-UNIV*[*symmetric*])

lemma *finite-UnionD*: *finite* $(\bigcup A)$ $\Longrightarrow$ *finite A*
by (*blast intro*: *finite-subset* [*OF subset-Pow-Union*])

lemma *finite-bind*:
assumes *finite S*
assumes $\forall x \in S.$ *finite* $(f\ x)$
shows *finite* (*Set.bind S f*)
using *assms* **by** (*simp add*: *bind-UNION*)

lemma *finite-filter* [*simp*]: *finite S* $\Longrightarrow$ *finite* (*Set.filter P S*)
by (*simp add*:)

lemma *finite-set-of-finite-funs*:
assumes *finite A finite B*
shows *finite* $\{f.\ \forall x.\ (x \in A \longrightarrow f\ x \in B) \wedge (x \notin A \longrightarrow f\ x = d)\}$ (**is** *finite ?S*)
proof −
let *?F* = $\lambda f.\ \{(a,b).\ a \in A \wedge b = f\ a\}$
have *?F* ‘ *?S* $\subseteq$ *Pow*$(A \times B)$
by *auto*
from *finite-subset*[*OF this*] *assms* **have** *1*: *finite* (*?F* ‘ *?S*)
by *simp*
have *2*: *inj-on ?F ?S*
by (*fastforce simp add*: *inj-on-def set-eq-iff fun-eq-iff*)
show *?thesis*
by (*rule finite-imageD* [*OF 1 2*])
qed

lemma *not-finite-existsD*:
assumes ¬ *finite* $\{a.\ P\ a\}$
shows $\exists a.\ P\ a$
proof (*rule classical*)
assume ¬ *?thesis*
with *assms* **show** *?thesis* **by** *auto*
qed

lemma *finite-converse* [*iff*]: *finite* (r^{-1}) $\longleftrightarrow$ *finite r*
unfolding *converse-def conversep-iff*
using [[*simproc add*: *finite-Collect*]]
by (*auto elim*: *finite-imageD simp*: *inj-on-def*)

lemma *finite-Domain*: *finite r* $\Longrightarrow$ *finite* (*Domain r*)
by (*induct set*: *finite*) *auto*

lemma *finite-Range*: *finite r* $\Longrightarrow$ *finite* (*Range r*)
by (*induct set*: *finite*) *auto*

lemma *finite-Field*: *finite r* $\Longrightarrow$ *finite* (*Field r*)
by (*simp add*: *Field-def finite-Domain finite-Range*)

lemma *finite-Image*[*simp*]: *finite R* $\Longrightarrow$ *finite* (*R* `` *A*)
by(*rule finite-subset*[*OF - finite-Range*]) *auto*

20.3 Further induction rules on finite sets

lemma *finite-ne-induct* [*case-names singleton insert*, *consumes 2*]:
assumes *finite F* **and** $F \neq \{\}$
assumes $\bigwedge x.\ P\ \{x\}$
and $\bigwedge x\ F.$ *finite* $F \Longrightarrow F \neq \{\} \Longrightarrow x \notin F \Longrightarrow P\ F \Longrightarrow P$ (*insert x F*)
shows *P F*
using *assms*
proof *induct*
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then show *?case* **by** *cases auto*
qed

lemma *finite-subset-induct* [*consumes 2*, *case-names empty insert*]:
assumes *finite F* **and** $F \subseteq A$
and *empty*: *P* $\{\}$
and *insert*: $\bigwedge a\ F.$ *finite* $F \Longrightarrow a \in A \Longrightarrow a \notin F \Longrightarrow P\ F \Longrightarrow P$ (*insert a F*)
shows *P F*
using ‹*finite F*› ‹$F \subseteq A$›
proof *induct*
show *P* $\{\}$ **by** *fact*
next
fix *x F*
assume *finite F* **and** $x \notin F$ **and** *P*: $F \subseteq A \Longrightarrow P\ F$ **and** *i*: *insert x F* $\subseteq A$
show *P* (*insert x F*)
proof (*rule insert*)
from *i* **show** $x \in A$ **by** *blast*
from *i* **have** $F \subseteq A$ **by** *blast*
with *P* **show** *P F* **.**
show *finite F* **by** *fact*
show $x \notin F$ **by** *fact*
qed
qed

lemma *finite-empty-induct*:
assumes *finite A*

```
    and P A
    and remove: ⋀a A. finite A ⟹ a ∈ A ⟹ P A ⟹ P (A − {a})
  shows P {}
proof −
  have P (A − B) if B ⊆ A for B :: 'a set
  proof −
    from ‹finite A› that have finite B
      by (rule rev-finite-subset)
    from this ‹B ⊆ A› show P (A − B)
    proof induct
      case empty
      from ‹P A› show ?case by simp
    next
      case (insert b B)
      have P (A − B − {b})
      proof (rule remove)
        from ‹finite A› show finite (A − B)
          by induct auto
        from insert show b ∈ A − B
          by simp
        from insert show P (A − B)
          by simp
      qed
      also have A − B − {b} = A − insert b B
        by (rule Diff-insert [symmetric])
      finally show ?case .
    qed
  qed
  then have P (A − A) by blast
  then show ?thesis by simp
qed

lemma finite-update-induct [consumes 1, case-names const update]:
  assumes finite: finite {a. f a ≠ c}
    and const: P (λa. c)
    and update: ⋀a b f. finite {a. f a ≠ c} ⟹ f a = c ⟹ b ≠ c ⟹ P f ⟹ P
(f(a := b))
  shows P f
  using finite
proof (induct {a. f a ≠ c} arbitrary: f)
  case empty
  with const show ?case by simp
next
  case (insert a A)
  then have A = {a'. (f(a := c)) a' ≠ c} and f a ≠ c
    by auto
  with ‹finite A› have finite {a'. (f(a := c)) a' ≠ c}
    by simp
  have (f(a := c)) a = c
```

by *simp*
from *insert* ‹$A = \{a'.\ (f(a := c))\ a' \neq c\}$› **have** $P\ (f(a := c))$
by *simp*
with ‹*finite* $\{a'.\ (f(a := c))\ a' \neq c\}$› ‹$(f(a := c))\ a = c$› ‹$f\ a \neq c$›
have $P\ ((f(a := c))(a := f\ a))$
by (*rule update*)
then show *?case* **by** *simp*
qed

lemma *finite-subset-induct'* [*consumes 2*, *case-names empty insert*]:
assumes *finite* F **and** $F \subseteq A$
and *empty*: $P\ \{\}$
and *insert*: $\bigwedge a\ F.\ [\![\mathit{finite}\ F;\ a \in A;\ F \subseteq A;\ a \notin F;\ P\ F\ ]\!] \Longrightarrow P\ (\mathit{insert}\ a\ F)$
shows $P\ F$
using *assms*(*1*,*2*)
proof *induct*
show $P\ \{\}$ **by** *fact*
next
fix $x\ F$
assume *finite* F **and** $x \notin F$ **and**
P: $F \subseteq A \Longrightarrow P\ F$ **and** i: *insert* $x\ F \subseteq A$
show $P\ (\mathit{insert}\ x\ F)$
proof (*rule insert*)
from i **show** $x \in A$ **by** *blast*
from i **have** $F \subseteq A$ **by** *blast*
with P **show** $P\ F$ **.**
show *finite* F **by** *fact*
show $x \notin F$ **by** *fact*
show $F \subseteq A$ **by** *fact*
qed
qed

20.4 Class *finite*

class *finite* =
assumes *finite-UNIV*: *finite* $(UNIV :: 'a\ set)$
begin

lemma *finite* [*simp*]: *finite* $(A :: 'a\ set)$
by (*rule subset-UNIV finite-UNIV finite-subset*)+

lemma *finite-code* [*code*]: *finite* $(A :: 'a\ set) \longleftrightarrow True$
by *simp*

end

instance *prod* :: (*finite*, *finite*) *finite*
by *standard* (*simp only*: *UNIV-Times-UNIV* [*symmetric*] *finite-cartesian-product finite*)

lemma *inj-graph*: *inj* (λ*f*. {(*x*, *y*). *y* = *f x*})
by (*rule inj-onI*) (*auto simp add*: *set-eq-iff fun-eq-iff*)

instance *fun* :: (*finite*, *finite*) *finite*
proof
show *finite* (*UNIV* :: ('*a* ⇒ '*b*) *set*)
proof (*rule finite-imageD*)
let *?graph* = λ*f*::'*a* ⇒ '*b*. {(*x*, *y*). *y* = *f x*}
have *range ?graph* ⊆ *Pow UNIV*
by *simp*
moreover have *finite* (*Pow* (*UNIV* :: ('*a* ∗ '*b*) *set*))
by (*simp only*: *finite-Pow-iff finite*)
ultimately show *finite* (*range ?graph*)
by (*rule finite-subset*)
show *inj ?graph*
by (*rule inj-graph*)
qed
qed

instance *bool* :: *finite*
by *standard* (*simp add*: *UNIV-bool*)

instance *set* :: (*finite*) *finite*
by *standard* (*simp only*: *Pow-UNIV* [*symmetric*] *finite-Pow-iff finite*)

instance *unit* :: *finite*
by *standard* (*simp add*: *UNIV-unit*)

instance *sum* :: (*finite*, *finite*) *finite*
by *standard* (*simp only*: *UNIV-Plus-UNIV* [*symmetric*] *finite-Plus finite*)

20.5 A basic fold functional for finite sets

The intended behaviour is *fold f z* {$x_1, \ldots, x_n$} = $f\ x_1\ (\ldots\ (f\ x_n\ z)\ldots)$ if f is "left-commutative". The commutativity requirement is relativised to the carrier set S:

locale *comp-fun-commute-on* =
fixes *S* :: '*a set*
fixes *f* :: '*a* ⇒ '*b* ⇒ '*b*
assumes *comp-fun-commute-on*: *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *f y* ∘ *f x* = *f x* ∘ *f y*
begin

lemma *fun-left-comm*: *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *f y* (*f x z*) = *f x* (*f y z*)
using *comp-fun-commute-on* **by** (*simp add*: *fun-eq-iff*)

lemma *commute-left-comp*: *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *f y* ∘ (*f x* ∘ *g*) = *f x* ∘ (*f y* ∘ *g*)
by (*simp add*: *o-assoc comp-fun-commute-on*)

end

inductive *fold-graph* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$
  for $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$
  where
    emptyI [*intro*]: *fold-graph f z* $\{\}$ *z*
  | *insertI* [*intro*]: $x \notin A \Longrightarrow$ *fold-graph f z A y* $\Longrightarrow$ *fold-graph f z* (*insert x A*) (*f x y*)

inductive-cases *empty-fold-graphE* [*elim*!]: *fold-graph f z* $\{\}$ *x*

lemma *fold-graph-closed-lemma*:
  fold-graph f z A x $\wedge$ $x \in B$
  if *fold-graph g z A x*
    $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow f\ a\ b = g\ a\ b$
    $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow g\ a\ b \in B$
    $z \in B$
  using *that*(*1*−*3*)
proof (*induction rule*: *fold-graph.induct*)
  case (*insertI x A y*)
  have *fold-graph f z A y y* $\in B$
    unfolding *atomize-conj*
    by (*rule insertI.IH*) (*auto intro*: *insertI.prems*)
  then have *g x y* $\in B$ **and** *f-eq*: *f x y* = *g x y*
    by (*auto simp*: *insertI.prems*)
  moreover have *fold-graph f z* (*insert x A*) (*f x y*)
    by (*rule fold-graph.insertI*; *fact*)
  ultimately
  show *?case*
    by (*simp add*: *f-eq*)
qed (*auto intro*!: *that*)

lemma *fold-graph-closed-eq*:
  fold-graph f z A = *fold-graph g z A*
  if $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow f\ a\ b = g\ a\ b$
    $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow g\ a\ b \in B$
    $z \in B$
  using *fold-graph-closed-lemma*[*of f z A - B g*] *fold-graph-closed-lemma*[*of g z A - B f*] *that*
  by *auto*

definition *fold* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b$
  where *fold f z A* = (*if finite A then* (*THE y. fold-graph f z A y*) *else z*)

lemma *fold-closed-eq*: *fold f z A* = *fold g z A*
  if $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow f\ a\ b = g\ a\ b$
    $\bigwedge a\ b.\ a \in A \Longrightarrow b \in B \Longrightarrow g\ a\ b \in B$
    $z \in B$
  unfolding *Finite-Set.fold-def*

by (*subst fold-graph-closed-eq*[**where** *B*=*B* **and** *g*=*g*]) (*auto simp*: *that*)

A tempting alternative for the definition is *if finite A then THE y. fold-graph f z A y else e*. It allows the removal of finiteness assumptions from the theorems *fold-comm*, *fold-reindex* and *fold-distrib*. The proofs become ugly. It is not worth the effort. (???)

lemma *finite-imp-fold-graph*: *finite A* $\Longrightarrow \exists x.$ *fold-graph f z A x*
by (*induct rule*: *finite-induct*) *auto*

20.5.1 From *fold-graph* to *fold*

context *comp-fun-commute-on*
begin

lemma *fold-graph-finite*:
assumes *fold-graph f z A y*
shows *finite A*
using *assms* **by** *induct simp-all*

lemma *fold-graph-insertE-aux*:
assumes $A \subseteq S$
assumes *fold-graph f z A y a* $\in A$
shows $\exists y'.$ $y = f\ a\ y' \wedge$ *fold-graph f z* $(A - \{a\})\ y'$
using *assms*(*2*−,*1*)
proof (*induct set*: *fold-graph*)
case *emptyI*
then show *?case* **by** *simp*
next
case (*insertI x A y*)
show *?case*
proof (*cases x* = *a*)
case *True*
with *insertI* **show** *?thesis* **by** *auto*
next
case *False*
then obtain y' **where** *y*: $y = f\ a\ y'$ **and** y': *fold-graph f z* $(A - \{a\})\ y'$
using *insertI* **by** *auto*
from *insertI* **have** $x \in S\ a \in S$ **by** *auto*
then have $f\ x\ y = f\ a\ (f\ x\ y')$
unfolding *y* **by** (*intro fun-left-comm*; *simp*)
moreover have *fold-graph f z* (*insert x A* − $\{a\}$) $(f\ x\ y')$
using y' **and** ‹$x \neq a$› **and** ‹$x \notin A$›
by (*simp add*: *insert-Diff-if fold-graph.insertI*)
ultimately show *?thesis*
by *fast*
qed
qed

lemma *fold-graph-insertE*:

assumes *insert x A* ⊆ *S*
assumes *fold-graph f z* (*insert x A*) *v* **and** *x* ∉ *A*
obtains *y* **where** *v* = *f x y* **and** *fold-graph f z A y*
using *assms* **by** (*auto dest*: *fold-graph-insertE-aux*[*OF* ‹*insert x A* ⊆ *S*› - *insertI1*])

lemma *fold-graph-determ*:
assumes *A* ⊆ *S*
assumes *fold-graph f z A x fold-graph f z A y*
shows *y* = *x*
using *assms*(*2*−,*1*)
proof (*induct arbitrary*: *y set*: *fold-graph*)
case *emptyI*
then show *?case* **by** *fast*
next
case (*insertI x A y v*)
from ‹*insert x A* ⊆ *S*› **and** ‹*fold-graph f z* (*insert x A*) *v*› **and** ‹*x* ∉ *A*›
obtain *y′* **where** *v* = *f x y′* **and** *fold-graph f z A y′*
by (*rule fold-graph-insertE*)
from ‹*fold-graph f z A y′*› *insertI* **have** *y′* = *y*
by *simp*
with ‹*v* = *f x y′*› **show** *v* = *f x y*
by *simp*
qed

lemma *fold-equality*: *A* ⊆ *S* ⟹ *fold-graph f z A y* ⟹ *fold f z A* = *y*
by (*cases finite A*) (*auto simp add*: *fold-def intro*: *fold-graph-determ dest*: *fold-graph-finite*)

lemma *fold-graph-fold*:
assumes *A* ⊆ *S*
assumes *finite A*
shows *fold-graph f z A* (*fold f z A*)
proof −
from ‹*finite A*› **have** ∃ *x*. *fold-graph f z A x*
by (*rule finite-imp-fold-graph*)
moreover note *fold-graph-determ*[*OF* ‹*A* ⊆ *S*›]
ultimately have ∃!*x*. *fold-graph f z A x*
by (*rule ex-ex1I*)
then have *fold-graph f z A* (*The* (*fold-graph f z A*))
by (*rule theI′*)
with *assms* **show** *?thesis*
by (*simp add*: *fold-def*)
qed

The base case for *fold*:

lemma (**in** −) *fold-infinite* [*simp*]: ¬ *finite A* ⟹ *fold f z A* = *z*
by (*auto simp*: *fold-def*)

lemma (**in** −) *fold-empty* [*simp*]: *fold f z* {} = *z*

by (*auto simp*: *fold-def*)

The various recursion equations for *fold*:

lemma *fold-insert* [*simp*]:
assumes *insert x A* ⊆ *S*
assumes *finite A* **and** *x* ∉ *A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
proof (*rule fold-equality*[*OF* ‹*insert x A* ⊆ *S*›])
fix *z*
from ‹*insert x A* ⊆ *S*› ‹*finite A*› **have** *fold-graph f z A* (*fold f z A*)
by (*blast intro*: *fold-graph-fold*)
with ‹*x* ∉ *A*› **have** *fold-graph f z* (*insert x A*) (*f x* (*fold f z A*))
by (*rule fold-graph.insertI*)
then show *fold-graph f z* (*insert x A*) (*f x* (*fold f z A*))
by *simp*
qed

declare (**in** −) *empty-fold-graphE* [*rule del*] *fold-graph.intros* [*rule del*]
— No more proofs involve these.

lemma *fold-fun-left-comm*:
assumes *insert x A* ⊆ *S finite A*
shows *f x* (*fold f z A*) = *fold f* (*f x z*) *A*
using *assms*(*2*,*1*)
proof (*induct rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert y F*)
then have *fold f* (*f x z*) (*insert y F*) = *f y* (*fold f* (*f x z*) *F*)
by *simp*
also have ... = *f x* (*f y* (*fold f z F*))
using *insert* **by** (*simp add*: *fun-left-comm*[**where** *?y*=*x*])
also have ... = *f x* (*fold f z* (*insert y F*))
proof −
from *insert* **have** *insert y F* ⊆ *S* **by** *simp*
from *fold-insert*[*OF this*] *insert* **show** *?thesis* **by** *simp*
qed
finally show *?case* **..**
qed

lemma *fold-insert2*:
insert x A ⊆ *S* ⟹ *finite A* ⟹ *x* ∉ *A* ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by (*simp add*: *fold-fun-left-comm*)

lemma *fold-rec*:
assumes *A* ⊆ *S*
assumes *finite A* **and** *x* ∈ *A*

shows *fold f z A = f x (fold f z (A − {x}))*
proof −
have *A*: *A = insert x (A − {x})*
using ‹*x ∈ A*› **by** *blast*
then have *fold f z A = fold f z (insert x (A − {x}))*
by *simp*
also have *... = f x (fold f z (A − {x}))*
by (*rule fold-insert*) (*use assms* **in** ‹*auto*›)
finally show *?thesis* **.**
qed

lemma *fold-insert-remove*:
assumes *insert x A ⊆ S*
assumes *finite A*
shows *fold f z (insert x A) = f x (fold f z (A − {x}))*
proof −
from ‹*finite A*› **have** *finite (insert x A)*
by *auto*
moreover have *x ∈ insert x A*
by *auto*
ultimately have *fold f z (insert x A) = f x (fold f z (insert x A − {x}))*
using ‹*insert x A ⊆ S*› **by** (*blast intro*: *fold-rec*)
then show *?thesis*
by *simp*
qed

lemma *fold-set-union-disj*:
assumes *A ⊆ S B ⊆ S*
assumes *finite A finite B A ∩ B = {}*
shows *Finite-Set.fold f z (A ∪ B) = Finite-Set.fold f (Finite-Set.fold f z A) B*
using ‹*finite B*› *assms*(*1*,*2*,*3*,*5*)
proof *induct*
case (*insert x F*)
have *fold f z (A ∪ insert x F) = f x (fold f (fold f z A) F)*
using *insert* **by** *auto*
also have *... = fold f (fold f z A) (insert x F)*
using *insert* **by** (*blast intro*: *fold-insert*[*symmetric*])
finally show *?case* **.**
qed *simp*

end

Other properties of *fold*:

lemma *finite-set-fold-single* [*simp*]: *Finite-Set.fold f z {x} = f x z*
proof −
have *fold-graph f z {x} (f x z)*
by (*auto intro*: *fold-graph.intros*)
moreover

```
  {
    fix X y
    have fold-graph f z X y ⟹ (X = {} ⟶ y = z) ∧ (X = {x} ⟶ y = f x z)
      by (induct rule: fold-graph.induct) auto
  }
  ultimately have (THE y. fold-graph f z {x} y) = f x z
    by blast
  thus ?thesis
    by (simp add: Finite-Set.fold-def)
qed

lemma fold-graph-image:
  assumes inj-on g A
  shows fold-graph f z (g ‘ A) = fold-graph (f ∘ g) z A
proof
  fix w
  show fold-graph f z (g ‘ A) w = fold-graph (f o g) z A w
  proof
    assume fold-graph f z (g ‘ A) w
    then show fold-graph (f ∘ g) z A w
      using assms
    proof (induct g ‘ A w arbitrary: A)
      case emptyI
      then show ?case by (auto intro: fold-graph.emptyI)
    next
      case (insertI x A r B)
      from ‹inj-on g B› ‹x ∉ A› ‹insert x A = image g B› obtain x' A'
        where x' ∉ A' and [simp]: B = insert x' A' x = g x' A = g ‘ A'
        by (rule inj-img-insertE)
      from insertI.prems have fold-graph (f ∘ g) z A' r
        by (auto intro: insertI.hyps)
      with ‹x' ∉ A'› have fold-graph (f ∘ g) z (insert x' A') ((f ∘ g) x' r)
        by (rule fold-graph.insertI)
      then show ?case
        by simp
    qed
  next
    assume fold-graph (f ∘ g) z A w
    then show fold-graph f z (g ‘ A) w
      using assms
    proof induct
      case emptyI
      then show ?case
        by (auto intro: fold-graph.emptyI)
    next
      case (insertI x A r)
      from ‹x ∉ A› insertI.prems have g x ∉ g ‘ A
        by auto
      moreover from insertI have fold-graph f z (g ‘ A) r
```

```
      by simp
    ultimately have fold-graph f z (insert (g x) (g ‘ A)) (f (g x) r)
      by (rule fold-graph.insertI)
    then show ?case
      by simp
  qed
 qed
qed

lemma fold-image:
 assumes inj-on g A
 shows fold f z (g ‘ A) = fold (f ∘ g) z A
proof (cases finite A)
 case False
 with assms show ?thesis
  by (auto dest: finite-imageD simp add: fold-def)
next
 case True
 then show ?thesis
  by (auto simp add: fold-def fold-graph-image[OF assms])
qed

lemma fold-cong:
 assumes comp-fun-commute-on S f comp-fun-commute-on S g
  and A ⊆ S finite A
  and cong: ⋀x. x ∈ A ⟹ f x = g x
  and s = t and A = B
 shows fold f s A = fold g t B
proof −
 have fold f s A = fold g s A
  using ‹finite A› ‹A ⊆ S› cong
 proof (induct A)
  case empty
  then show ?case by simp
 next
  case insert
  interpret f: comp-fun-commute-on S f by (fact ‹comp-fun-commute-on S f›)
  interpret g: comp-fun-commute-on S g by (fact ‹comp-fun-commute-on S g›)
  from insert show ?case by simp
 qed
 with assms show ?thesis by simp
qed
```

A simplified version for idempotent functions:

```
locale comp-fun-idem-on = comp-fun-commute-on +
 assumes comp-fun-idem-on: x ∈ S ⟹ f x ∘ f x = f x
begin

lemma fun-left-idem: x ∈ S ⟹ f x (f x z) = f x z
```

using *comp-fun-idem-on* **by** (*simp add*: *fun-eq-iff*)

lemma *fold-insert-idem*:
assumes *insert x A* ⊆ *S*
assumes *fin*: *finite A*
shows *fold f z* (*insert x A*) = *f x* (*fold f z A*)
proof *cases*
assume *x* ∈ *A*
then obtain *B* **where** *A* = *insert x B* **and** *x* ∉ *B*
by (*rule set-insert*)
then show *?thesis*
using *assms* **by** (*simp add*: *comp-fun-idem-on fun-left-idem*)
next
assume *x* ∉ *A*
then show *?thesis*
using *assms* **by** *auto*
qed

declare *fold-insert* [*simp del*] *fold-insert-idem* [*simp*]

lemma *fold-insert-idem2*: *insert x A* ⊆ *S* ⟹ *finite A* ⟹ *fold f z* (*insert x A*) = *fold f* (*f x z*) *A*
by (*simp add*: *fold-fun-left-comm*)

end

20.5.2 Liftings to *comp-fun-commute-on* etc.

lemma (**in** *comp-fun-commute-on*) *comp-comp-fun-commute-on*:
range g ⊆ *S* ⟹ *comp-fun-commute-on R* (*f* ∘ *g*)
by *standard* (*force intro*: *comp-fun-commute-on*)

lemma (**in** *comp-fun-idem-on*) *comp-comp-fun-idem-on*:
assumes *range g* ⊆ *S*
shows *comp-fun-idem-on R* (*f* ∘ *g*)
proof
interpret *f-g*: *comp-fun-commute-on R f o g*
by (*fact comp-comp-fun-commute-on*[*OF* ‹*range g* ⊆ *S*›])
show *x* ∈ *R* ⟹ *y* ∈ *R* ⟹ (*f* ∘ *g*) *y* ∘ (*f* ∘ *g*) *x* = (*f* ∘ *g*) *x* ∘ (*f* ∘ *g*) *y* **for** *x y*
by (*fact f-g.comp-fun-commute-on*)
qed (*use* ‹*range g* ⊆ *S*› **in** ‹*force intro*: *comp-fun-idem-on*›)

lemma (**in** *comp-fun-commute-on*) *comp-fun-commute-on-funpow*:
comp-fun-commute-on S (λ*x*. *f x* ^^ *g x*)
proof
fix *x y* **assume** *x* ∈ *S y* ∈ *S*
show *f y* ^^ *g y* ∘ *f x* ^^ *g x* = *f x* ^^ *g x* ∘ *f y* ^^ *g y*
proof (*cases x* = *y*)
case *True*

```
    then show ?thesis by simp
  next
    case False
    show ?thesis
    proof (induct g x arbitrary: g)
      case 0
      then show ?case by simp
    next
      case (Suc n g)
      have hyp1: f y ^^ g y ∘ f x = f x ∘ f y ^^ g y
      proof (induct g y arbitrary: g)
        case 0
        then show ?case by simp
      next
        case (Suc n g)
        define h where h z = g z − 1 for z
        with Suc have n = h y
          by simp
        with Suc have hyp: f y ^^ h y ∘ f x = f x ∘ f y ^^ h y
          by auto
        from Suc h-def have g y = Suc (h y)
          by simp
        with ‹x ∈ S› ‹y ∈ S› show ?case
          by (simp add: comp-assoc hyp) (simp add: o-assoc comp-fun-commute-on)
      qed
      define h where h z = (if z = x then g x − 1 else g z) for z
      with Suc have n = h x
        by simp
      with Suc have f y ^^ h y ∘ f x ^^ h x = f x ^^ h x ∘ f y ^^ h y
        by auto
      with False h-def have hyp2: f y ^^ g y ∘ f x ^^ h x = f x ^^ h x ∘ f y ^^ g y
        by simp
      from Suc h-def have g x = Suc (h x)
        by simp
      then show ?case
        by (simp del: funpow.simps add: funpow-Suc-right o-assoc hyp2) (simp add:
comp-assoc hyp1)
    qed
  qed
qed
```

20.5.3 *UNIV* as carrier set

```
locale comp-fun-commute =
  fixes f :: 'a ⇒ 'b ⇒ 'b
  assumes comp-fun-commute: f y ∘ f x = f x ∘ f y
begin

lemma (in −) comp-fun-commute-def': comp-fun-commute f = comp-fun-commute-on
```

UNIV f
unfolding *comp-fun-commute-def comp-fun-commute-on-def* **by** *blast*

We abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *comp-fun-commute-on UNIV f*
rewrites $\bigwedge X.\ (X \subseteq UNIV) \equiv True$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
and $\bigwedge P\ Q.\ (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
proof −
show *comp-fun-commute-on UNIV f*
by *standard* (*simp add*: *comp-fun-commute*)
qed *simp-all*

end

lemma (**in** *comp-fun-commute*) *comp-comp-fun-commute*: *comp-fun-commute* (*f o g*)
unfolding *comp-fun-commute-def'* **by** (*fact comp-comp-fun-commute-on*)

lemma (**in** *comp-fun-commute*) *comp-fun-commute-funpow*: *comp-fun-commute* ($\lambda x.$ *f x* ^^ *g x*)
unfolding *comp-fun-commute-def'* **by** (*fact comp-fun-commute-on-funpow*)

locale *comp-fun-idem* = *comp-fun-commute* +
assumes *comp-fun-idem*: *f x o f x* = *f x*
begin

lemma (**in** −) *comp-fun-idem-def'*: *comp-fun-idem f* = *comp-fun-idem-on UNIV f*
unfolding *comp-fun-idem-on-def comp-fun-idem-def comp-fun-commute-def'*
unfolding *comp-fun-idem-axioms-def comp-fun-idem-on-axioms-def*
by *blast*

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *comp-fun-idem-on UNIV f*
rewrites $\bigwedge X.\ (X \subseteq UNIV) \equiv True$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
and $\bigwedge P\ Q.\ (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
proof −
show *comp-fun-idem-on UNIV f*
by *standard* (*simp-all add*: *comp-fun-idem comp-fun-commute*)
qed *simp-all*

end

lemma (**in** *comp-fun-idem*) *comp-comp-fun-idem*: *comp-fun-idem* (*f o g*)
unfolding *comp-fun-idem-def'* **by** (*fact comp-comp-fun-idem-on*)

20.5.4 Expressing set operations via *fold*

lemma *comp-fun-commute-const*: *comp-fun-commute* (λ-. *f*)
by *standard* (*rule refl*)

lemma *comp-fun-idem-insert*: *comp-fun-idem insert*
by *standard auto*

lemma *comp-fun-idem-remove*: *comp-fun-idem Set.remove*
by *standard auto*

lemma (**in** *semilattice-inf*) *comp-fun-idem-inf*: *comp-fun-idem inf*
by *standard* (*auto simp add*: *inf-left-commute*)

lemma (**in** *semilattice-sup*) *comp-fun-idem-sup*: *comp-fun-idem sup*
by *standard* (*auto simp add*: *sup-left-commute*)

lemma *union-fold-insert*:
assumes *finite A*
shows *A* ∪ *B* = *fold insert B A*
proof −
interpret *comp-fun-idem insert*
by (*fact comp-fun-idem-insert*)
from ‹*finite A*› **show** *?thesis*
by (*induct A arbitrary*: *B*) *simp-all*
qed

lemma *minus-fold-remove*:
assumes *finite A*
shows *B* − *A* = *fold Set.remove B A*
proof −
interpret *comp-fun-idem Set.remove*
by (*fact comp-fun-idem-remove*)
from ‹*finite A*› **have** *fold Set.remove B A* = *B* − *A*
by (*induct A arbitrary*: *B*) *auto*
then show *?thesis* **..**
qed

lemma *comp-fun-commute-filter-fold*:
comp-fun-commute (λ*x A'*. *if P x then Set.insert x A' else A'*)
proof −
interpret *comp-fun-idem Set.insert* **by** (*fact comp-fun-idem-insert*)
show *?thesis* **by** *standard* (*auto simp*: *fun-eq-iff*)
qed

lemma *Set-filter-fold*:
assumes *finite A*
shows *Set.filter P A = fold* (λ*x A'*. *if P x then Set.insert x A' else A'*) {} *A*
using *assms*
proof −
interpret *commute-insert*: *comp-fun-commute* (λ*x A'*. *if P x then Set.insert x A' else A'*)
by (*fact comp-fun-commute-filter-fold*)
from ‹*finite A*› **show** *?thesis*
by *induct* (*auto simp add*: *set-eq-iff*)
qed

lemma *inter-Set-filter*:
assumes *finite B*
shows *A* ∩ *B = Set.filter* (λ*x*. *x* ∈ *A*) *B*
using *assms* **by** (*simp add*: *set-eq-iff ac-simps*)

lemma *image-fold-insert*:
assumes *finite A*
shows *image f A = fold* (λ*k A*. *Set.insert* (*f k*) *A*) {} *A*
proof −
interpret *comp-fun-commute* λ*k A*. *Set.insert* (*f k*) *A*
by *standard auto*
show *?thesis*
using *assms* **by** (*induct A*) *auto*
qed

lemma *Ball-fold*:
assumes *finite A*
shows *Ball A P = fold* (λ*k s*. *s* ∧ *P k*) *True A*
proof −
interpret *comp-fun-commute* λ*k s*. *s* ∧ *P k*
by *standard auto*
show *?thesis*
using *assms* **by** (*induct A*) *auto*
qed

lemma *Bex-fold*:
assumes *finite A*
shows *Bex A P = fold* (λ*k s*. *s* ∨ *P k*) *False A*
proof −
interpret *comp-fun-commute* λ*k s*. *s* ∨ *P k*
by *standard auto*
show *?thesis*
using *assms* **by** (*induct A*) *auto*
qed

lemma *comp-fun-commute-Pow-fold*: *comp-fun-commute* (λ*x A*. *A* ∪ *Set.insert x*

‘ *A*)
by (*clarsimp simp*: *fun-eq-iff comp-fun-commute-def*) *blast*

lemma *Pow-fold*:
assumes *finite A*
shows *Pow A* = *fold* (λ*x A*. *A* ∪ *Set.insert x* ‘ *A*) {{}} *A*
proof −
interpret *comp-fun-commute* λ*x A*. *A* ∪ *Set.insert x* ‘ *A*
by (*rule comp-fun-commute-Pow-fold*)
show *?thesis*
using *assms* **by** (*induct A*) (*auto simp*: *Pow-insert*)
qed

lemma *fold-union-pair*:
assumes *finite B*
shows (⋃ *y*∈*B*. {(*x*, *y*)}) ∪ *A* = *fold* (λ*y*. *Set.insert* (*x*, *y*)) *A B*
proof −
interpret *comp-fun-commute* λ*y*. *Set.insert* (*x*, *y*)
by *standard auto*
show *?thesis*
using *assms* **by** (*induct arbitrary*: *A*) *simp-all*
qed

lemma *comp-fun-commute-product-fold*:
finite B ⟹ *comp-fun-commute* (λ*x z*. *fold* (λ*y*. *Set.insert* (*x*, *y*)) *z B*)
by *standard* (*auto simp*: *fold-union-pair* [*symmetric*])

lemma *product-fold*:
assumes *finite A finite B*
shows *A* × *B* = *fold* (λ*x z*. *fold* (λ*y*. *Set.insert* (*x*, *y*)) *z B*) {} *A*
proof −
interpret *commute-product*: *comp-fun-commute* (λ*x z*. *fold* (λ*y*. *Set.insert* (*x*, *y*)) *z B*)
by (*fact comp-fun-commute-product-fold*[*OF* ‹*finite B*›])
from *assms* **show** *?thesis* **unfolding** *Sigma-def*
by (*induct A*) (*simp-all add*: *fold-union-pair*)
qed

context *complete-lattice*
begin

lemma *inf-Inf-fold-inf*:
assumes *finite A*
shows *inf* (*Inf A*) *B* = *fold inf B A*
proof −
interpret *comp-fun-idem inf*
by (*fact comp-fun-idem-inf*)
from ‹*finite A*› *fold-fun-left-comm* **show** *?thesis*
by (*induct A arbitrary*: *B*) (*simp-all add*: *inf-commute fun-eq-iff*)

qed

lemma *sup-Sup-fold-sup*:
assumes *finite A*
shows *sup* (*Sup A*) *B* = *fold sup B A*
proof −
interpret *comp-fun-idem sup*
by (*fact comp-fun-idem-sup*)
from ‹*finite A*› *fold-fun-left-comm* **show** *?thesis*
by (*induct A arbitrary*: *B*) (*simp-all add*: *sup-commute fun-eq-iff*)
qed

lemma *Inf-fold-inf*: *finite A* ⟹ *Inf A* = *fold inf top A*
using *inf-Inf-fold-inf* [*of A top*] **by** (*simp add*: *inf-absorb2*)

lemma *Sup-fold-sup*: *finite A* ⟹ *Sup A* = *fold sup bot A*
using *sup-Sup-fold-sup* [*of A bot*] **by** (*simp add*: *sup-absorb2*)

lemma *inf-INF-fold-inf*:
assumes *finite A*
shows *inf B* (⨅(*f* ‘ *A*)) = *fold* (*inf* ∘ *f*) *B A* (**is** *?inf* = *?fold*)
proof −
interpret *comp-fun-idem inf* **by** (*fact comp-fun-idem-inf*)
interpret *comp-fun-idem inf* ∘ *f* **by** (*fact comp-comp-fun-idem*)
from ‹*finite A*› **have** *?fold* = *?inf*
by (*induct A arbitrary*: *B*) (*simp-all add*: *inf-left-commute*)
then show *?thesis* **..**
qed

lemma *sup-SUP-fold-sup*:
assumes *finite A*
shows *sup B* (⨆(*f* ‘ *A*)) = *fold* (*sup* ∘ *f*) *B A* (**is** *?sup* = *?fold*)
proof −
interpret *comp-fun-idem sup* **by** (*fact comp-fun-idem-sup*)
interpret *comp-fun-idem sup* ∘ *f* **by** (*fact comp-comp-fun-idem*)
from ‹*finite A*› **have** *?fold* = *?sup*
by (*induct A arbitrary*: *B*) (*simp-all add*: *sup-left-commute*)
then show *?thesis* **..**
qed

lemma *INF-fold-inf*: *finite A* ⟹ ⨅(*f* ‘ *A*) = *fold* (*inf* ∘ *f*) *top A*
using *inf-INF-fold-inf* [*of A top*] **by** *simp*

lemma *SUP-fold-sup*: *finite A* ⟹ ⨆(*f* ‘ *A*) = *fold* (*sup* ∘ *f*) *bot A*
using *sup-SUP-fold-sup* [*of A bot*] **by** *simp*

lemma *finite-Inf-in*:
assumes *finite A A*≠{} **and** *inf*: ⋀*x y*. ⟦*x* ∈ *A*; *y* ∈ *A*⟧ ⟹ *inf x y* ∈ *A*
shows *Inf A* ∈ *A*

proof −
have *Inf* $B \in A$ **if** $B \leq A$ $B \neq \{\}$ **for** B
using *finite-subset* [*OF* ‹$B \subseteq A$› ‹*finite* A›] *that*
by (*induction* B) (*use inf* **in** ‹*force*+›)
then show *?thesis*
by (*simp add*: *assms*)
qed

lemma *finite-Sup-in*:
assumes *finite* A $A \neq \{\}$ **and** *sup*: $\bigwedge x\ y.\ [\![x \in A;\ y \in A]\!] \Longrightarrow sup\ x\ y \in A$
shows *Sup* $A \in A$
proof −
have *Sup* $B \in A$ **if** $B \leq A$ $B \neq \{\}$ **for** B
using *finite-subset* [*OF* ‹$B \subseteq A$› ‹*finite* A›] *that*
by (*induction* B) (*use sup* **in** ‹*force*+›)
then show *?thesis*
by (*simp add*: *assms*)
qed

end

20.5.5 Expressing relation operations via *fold*

lemma *Id-on-fold*:
assumes *finite* A
shows *Id-on* A = *Finite-Set.fold* ($\lambda x.$ *Set.insert* (*Pair* x x)) $\{\}$ A
proof −
interpret *comp-fun-commute* $\lambda x.$ *Set.insert* (*Pair* x x)
by *standard auto*
from *assms* **show** *?thesis*
unfolding *Id-on-def* **by** (*induct* A) *simp-all*
qed

lemma *comp-fun-commute-Image-fold*:
comp-fun-commute ($\lambda(x,y)$ A. *if* $x \in S$ *then Set.insert* y A *else* A)
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
show *?thesis*
by *standard* (*auto simp*: *fun-eq-iff comp-fun-commute split*: *prod.split*)
qed

lemma *Image-fold*:
assumes *finite* R
shows R `` S = *Finite-Set.fold* ($\lambda(x,y)$ A. *if* $x \in S$ *then Set.insert* y A *else* A)
$\{\}$ R
proof −
interpret *comp-fun-commute* ($\lambda(x,y)$ A. *if* $x \in S$ *then Set.insert* y A *else* A)
by (*rule comp-fun-commute-Image-fold*)

have $*$: $\bigwedge x\ F.$ *Set.insert x F `` S = (if fst x ∈ S then Set.insert (snd x) (F `` S) else (F `` S))*
by (*force intro*: *rev-ImageI*)
show *?thesis*
using *assms* **by** (*induct R*) (*auto simp*: $*$)
qed

lemma *insert-relcomp-union-fold*:
assumes *finite S*
shows {*x*} *O S* ∪ *X* = *Finite-Set.fold* (λ(*w*,*z*) *A'*. *if snd x* = *w then Set.insert* (*fst x*,*z*) *A' else A'*) *X S*
proof −
interpret *comp-fun-commute* λ(*w*,*z*) *A'*. *if snd x* = *w then Set.insert* (*fst x*,*z*) *A' else A'*
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
show *comp-fun-commute* (λ(*w*,*z*) *A'*. *if snd x* = *w then Set.insert* (*fst x*,*z*) *A' else A'*)
by *standard* (*auto simp add*: *fun-eq-iff split*: *prod.split*)
qed
have $*$: {*x*} *O S* = {(*x'*, *z*). *x'* = *fst x* ∧ (*snd x*, *z*) ∈ *S*}
by (*auto simp*: *relcomp-unfold intro*!: *exI*)
show *?thesis*
unfolding $*$ **using** ‹*finite S*› **by** (*induct S*) (*auto split*: *prod.split*)
qed

lemma *insert-relcomp-fold*:
assumes *finite S*
shows *Set.insert x R O S* =
Finite-Set.fold (λ(*w*,*z*) *A'*. *if snd x* = *w then Set.insert* (*fst x*,*z*) *A' else A'*) (*R O S*) *S*
proof −
have *Set.insert x R O S* = ({*x*} *O S*) ∪ (*R O S*)
by *auto*
then show *?thesis*
by (*auto simp*: *insert-relcomp-union-fold* [*OF assms*])
qed

lemma *comp-fun-commute-relcomp-fold*:
assumes *finite S*
shows *comp-fun-commute* (λ(*x*,*y*) *A*.
Finite-Set.fold (λ(*w*,*z*) *A'*. *if y* = *w then Set.insert* (*x*,*z*) *A' else A'*) *A S*)
proof −
have $*$: $\bigwedge a\ b\ A.$
Finite-Set.fold (λ(*w*, *z*) *A'*. *if b* = *w then Set.insert* (*a*, *z*) *A' else A'*) *A S* = {(*a*,*b*)} *O S* ∪ *A*
by (*auto simp*: *insert-relcomp-union-fold*[*OF assms*] *cong*: *if-cong*)
show *?thesis*

by *standard* (*auto simp*: ∗)
qed

lemma *relcomp-fold*:
assumes *finite R finite S*
shows *R O S = Finite-Set.fold*
(λ(*x*,*y*) *A*. *Finite-Set.fold* (λ(*w*,*z*) *A*′. *if y = w then Set.insert* (*x*,*z*) *A*′ *else A*′)
A S) {} *R*
proof −
interpret *commute-relcomp-fold*: *comp-fun-commute*
(λ(*x*, *y*) *A*. *Finite-Set.fold* (λ(*w*, *z*) *A*′. *if y = w then insert* (*x*, *z*) *A*′ *else A*′)
A S)
by (*fact comp-fun-commute-relcomp-fold*[*OF* ‹*finite S*›])
from *assms* **show** *?thesis*
by (*induct R*) (*auto simp*: *comp-fun-commute-relcomp-fold insert-relcomp-fold cong*: *if-cong*)
qed

20.6 Locales as mini-packages for fold operations

20.6.1 The natural case

locale *folding-on* =
fixes *S* :: *′a set*
fixes *f* :: *′a* ⇒ *′b* ⇒ *′b* **and** *z* :: *′b*
assumes *comp-fun-commute-on*: *x* ∈ *S* ⟹ *y* ∈ *S* ⟹ *f y o f x* = *f x o f y*
begin

interpretation *fold?*: *comp-fun-commute-on S f*
by *standard* (*simp add*: *comp-fun-commute-on*)

definition *F* :: *′a set* ⇒ *′b*
where *eq-fold*: *F A* = *Finite-Set.fold f z A*

lemma *empty* [*simp*]: *F* {} = *z*
by (*simp add*: *eq-fold*)

lemma *infinite* [*simp*]: ¬ *finite A* ⟹ *F A* = *z*
by (*simp add*: *eq-fold*)

lemma *insert* [*simp*]:
assumes *insert x A* ⊆ *S* **and** *finite A* **and** *x* ∉ *A*
shows *F* (*insert x A*) = *f x* (*F A*)
proof −
from *fold-insert assms*
have *Finite-Set.fold f z* (*insert x A*)
= *f x* (*Finite-Set.fold f z A*)
by *simp*
with ‹*finite A*› **show** *?thesis* **by** (*simp add*: *eq-fold fun-eq-iff*)
qed

lemma *remove*:
  assumes $A \subseteq S$ **and** *finite* A **and** $x \in A$
  shows $F\ A = f\ x\ (F\ (A - \{x\}))$
proof −
  from ‹$x \in A$› **obtain** B **where** A: $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
    by (*auto dest*: *mk-disjoint-insert*)
  moreover from ‹*finite* A› A **have** *finite* B **by** *simp*
  ultimately show *?thesis*
    using ‹$A \subseteq S$› **by** *auto*
qed

lemma *insert-remove*:
  assumes *insert* $x\ A \subseteq S$ **and** *finite* A
  shows $F\ (\mathit{insert}\ x\ A) = f\ x\ (F\ (A - \{x\}))$
  using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *remove insert-absorb*)

end

20.6.2 With idempotency

locale *folding-idem-on* = *folding-on* +
  assumes *comp-fun-idem-on*: $x \in S \Longrightarrow y \in S \Longrightarrow f\ x \circ f\ x = f\ x$
begin

declare *insert* [*simp del*]

interpretation *fold?*: *comp-fun-idem-on* $S\ f$
  by *standard* (*simp-all add*: *comp-fun-commute-on comp-fun-idem-on*)

lemma *insert-idem* [*simp*]:
  assumes *insert* $x\ A \subseteq S$ **and** *finite* A
  shows $F\ (\mathit{insert}\ x\ A) = f\ x\ (F\ A)$
proof −
  from *fold-insert-idem assms*
  have *fold* $f\ z\ (\mathit{insert}\ x\ A) = f\ x\ (\mathit{fold}\ f\ z\ A)$ **by** *simp*
  with ‹*finite* A› **show** *?thesis* **by** (*simp add*: *eq-fold fun-eq-iff*)
qed

end

20.6.3 *UNIV* as the carrier set

locale *folding* =
  fixes $f :: 'a \Rightarrow 'b \Rightarrow 'b$ **and** $z :: 'b$
  assumes *comp-fun-commute*: $f\ y \circ f\ x = f\ x \circ f\ y$
begin

lemma (**in** −) *folding-def'*: *folding* f = *folding-on UNIV* f
  unfolding *folding-def folding-on-def* **by** *blast*

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *folding-on UNIV f*
  rewrites $\bigwedge X.\ (X \subseteq UNIV) \equiv True$
      and $\bigwedge x.\ x \in UNIV \equiv True$
      and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
      and $\bigwedge P\ Q.\ (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
proof −
  show *folding-on UNIV f*
    by *standard* (*simp add*: *comp-fun-commute*)
qed *simp-all*

end

locale *folding-idem* = *folding* +
  assumes *comp-fun-idem*: $f\ x \circ f\ x = f\ x$
begin

lemma (**in** −) *folding-idem-def'*: *folding-idem f* = *folding-idem-on UNIV f*
  unfolding *folding-idem-def folding-def' folding-idem-on-def*
  unfolding *folding-idem-axioms-def folding-idem-on-axioms-def*
  by *blast*

Again, we abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the carrier set to *UNIV*.

sublocale *folding-idem-on UNIV f*
  rewrites $\bigwedge X.\ (X \subseteq UNIV) \equiv True$
      and $\bigwedge x.\ x \in UNIV \equiv True$
      and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
      and $\bigwedge P\ Q.\ (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
proof −
  show *folding-idem-on UNIV f*
    by *standard* (*simp add*: *comp-fun-idem*)
qed *simp-all*

end

20.7 Finite cardinality

The traditional definition $card\ A \equiv LEAST\ n.\ \exists f.\ A = \{f\ i\ |i.\ i < n\}$ is ugly to work with. But now that we have *fold* things are easy:

global-interpretation *card*: *folding* λ-. *Suc 0*
  defines *card* = *folding-on.F* (λ-. *Suc*) *0*
  by *standard* (*rule refl*)

lemma *card-insert-disjoint*: *finite A* $\Longrightarrow$ $x \notin A$ $\Longrightarrow$ *card* (*insert x A*) = *Suc* (*card A*)
by (*fact card.insert*)

lemma *card-insert-if*: *finite A* $\Longrightarrow$ *card* (*insert x A*) = (*if x* $\in$ *A then card A else Suc* (*card A*))
by *auto* (*simp add*: *card.insert-remove card.remove*)

lemma *card-Domain-le*: *finite A* $\Longrightarrow$ *card* (*Domain A*) $\leq$ *card A*
by (*induction rule*: *finite-induct*) (*simp-all flip*: *fst-eq-Domain add*: *card-insert-if*)

lemma *card-Range-le*: *finite A* $\Longrightarrow$ *card* (*Range A*) $\leq$ *card A*
by (*induction rule*: *finite-induct*) (*simp-all flip*: *snd-eq-Range add*: *card-insert-if*)

lemma *card-ge-0-finite*: *card A* > *0* $\Longrightarrow$ *finite A*
by (*rule ccontr*) *simp*

lemma *card-0-eq* [*simp*]: *finite A* $\Longrightarrow$ *card A* = *0* $\longleftrightarrow$ *A* = {}
by (*auto dest*: *mk-disjoint-insert*)

lemma *finite-UNIV-card-ge-0*: *finite* (*UNIV* :: *'a set*) $\Longrightarrow$ *card* (*UNIV* :: *'a set*) > *0*
by (*rule ccontr*) *simp*

lemma *card-eq-0-iff*: *card A* = *0* $\longleftrightarrow$ *A* = {} $\vee$ $\neg$ *finite A*
by *auto*

lemma *card-range-greater-zero*: *finite* (*range f*) $\Longrightarrow$ *card* (*range f*) > *0*
by (*rule ccontr*) (*simp add*: *card-eq-0-iff*)

lemma *card-gt-0-iff*: *0* < *card A* $\longleftrightarrow$ *A* $\neq$ {} $\wedge$ *finite A*
by (*simp add*: *neq0-conv* [*symmetric*] *card-eq-0-iff*)

lemma *card-Suc-Diff1*:
assumes *finite A x* $\in$ *A* **shows** *Suc* (*card* (*A* $-$ {*x*})) = *card A*
proof $-$
have *Suc* (*card* (*A* $-$ {*x*})) = *card* (*insert x* (*A* $-$ {*x*}))
using *assms* **by** (*simp add*: *card.insert-remove*)
also have ... = *card A*
using *assms* **by** (*simp add*: *card-insert-if*)
finally show *?thesis* **.**
qed

lemma *card-insert-le-m1*:
assumes *n* > *0 card y* $\leq$ *n* $-$ *1* **shows** *card* (*insert x y*) $\leq$ *n*
using *assms*
by (*cases finite y*) (*auto simp*: *card-insert-if*)

lemma *card-Diff-singleton*:

assumes $x \in A$ **shows** *card* $(A - \{x\}) =$ *card* $A - 1$
proof (*cases finite* A)
case *True*
with *assms* **show** *?thesis*
by (*simp add*: *card-Suc-Diff1* [*symmetric*])
qed *auto*

lemma *card-Diff-singleton-if*:
card $(A - \{x\}) =$ (*if* $x \in A$ *then card* $A - 1$ *else card* A)
by (*simp add*: *card-Diff-singleton*)

lemma *card-Diff-insert*[*simp*]:
assumes $a \in A$ **and** $a \notin B$
shows *card* $(A -$ *insert* $a\ B) =$ *card* $(A - B) - 1$
proof −
have $A -$ *insert* $a\ B = (A - B) - \{a\}$
using *assms* **by** *blast*
then show *?thesis*
using *assms* **by** (*simp add*: *card-Diff-singleton*)
qed

lemma *card-insert-le*: *card* $A \leq$ *card* (*insert* $x\ A$)
proof (*cases finite* A)
case *True*
then show *?thesis* **by** (*simp add*: *card-insert-if*)
qed *auto*

lemma *card-Collect-less-nat*[*simp*]: *card* $\{i$::*nat*. $i < n\} = n$
by (*induct* n) (*simp-all add*:*less-Suc-eq Collect-disj-eq*)

lemma *card-Collect-le-nat*[*simp*]: *card* $\{i$::*nat*. $i \leq n\} =$ *Suc* n
using *card-Collect-less-nat*[*of Suc* n] **by** (*simp add*: *less-Suc-eq-le*)

lemma *card-mono*:
assumes *finite* B **and** $A \subseteq B$
shows *card* $A \leq$ *card* B
proof −
from *assms* **have** *finite* A
by (*auto intro*: *finite-subset*)
then show *?thesis*
using *assms*
proof (*induct* A *arbitrary*: B)
case *empty*
then show *?case* **by** *simp*
next
case (*insert* $x\ A$)
then have $x \in B$
by *simp*
from *insert* **have** $A \subseteq B - \{x\}$ **and** *finite* $(B - \{x\})$

```
    by auto
  with insert.hyps have card A ≤ card (B − {x})
    by auto
  with ‹finite A› ‹x ∉ A› ‹finite B› ‹x ∈ B› show ?case
    by simp (simp only: card.remove)
 qed
qed

lemma card-seteq:
 assumes finite B and A: A ⊆ B card B ≤ card A
 shows A = B
 using assms
proof (induction arbitrary: A rule: finite-induct)
 case (insert b B)
 then have A: finite A A − {b} ⊆ B
  by force+
 then have card B ≤ card (A − {b})
  using insert by (auto simp add: card-Diff-singleton-if)
 then have A − {b} = B
  using A insert.IH by auto
 then show ?case
  using insert.hyps insert.prems by auto
qed auto

lemma psubset-card-mono: finite B ⟹ A < B ⟹ card A < card B
 using card-seteq [of B A] by (auto simp add: psubset-eq)

lemma card-Un-Int:
 assumes finite A finite B
 shows card A + card B = card (A ∪ B) + card (A ∩ B)
 using assms
proof (induct A)
 case empty
 then show ?case by simp
next
 case insert
 then show ?case
  by (auto simp add: insert-absorb Int-insert-left)
qed

lemma card-Un-disjoint: finite A ⟹ finite B ⟹ A ∩ B = {} ⟹ card (A ∪ B)
= card A + card B
 using card-Un-Int [of A B] by simp

lemma card-Un-disjnt: ⟦finite A; finite B; disjnt A B⟧ ⟹ card (A ∪ B) = card
A + card B
 by (simp add: card-Un-disjoint disjnt-def)

lemma card-sym-diff-finite:
```

assumes *finite A finite B*
shows *card* (*sym-diff A B*) = *card* (*A*−*B*) + *card* (*B*−*A*)
by (*simp add*: *assms card-Un-disjnt disjnt-Diff2*)

lemma *card-Un-le*: *card* (*A* ∪ *B*) ≤ *card A* + *card B*
proof (*cases finite A* ∧ *finite B*)
case *True*
then show *?thesis*
using *le-iff-add card-Un-Int* [*of A B*] **by** *auto*
qed *auto*

lemma *card-Diff-subset*:
assumes *finite B*
and *B* ⊆ *A*
shows *card* (*A* − *B*) = *card A* − *card B*
using *assms*
proof (*cases finite A*)
case *False*
with *assms* **show** *?thesis*
by *simp*
next
case *True*
with *assms* **show** *?thesis*
by (*induct B arbitrary*: *A*) *simp-all*
qed

lemma *card-Diff-subset-Int*:
assumes *finite* (*A* ∩ *B*)
shows *card* (*A* − *B*) = *card A* − *card* (*A* ∩ *B*)
proof −
have *A* − *B* = *A* − *A* ∩ *B* **by** *auto*
with *assms* **show** *?thesis*
by (*simp add*: *card-Diff-subset*)
qed

lemma *card-Int-Diff*:
assumes *finite A*
shows *card A* = *card* (*A* ∩ *B*) + *card* (*A* − *B*)
by (*simp add*: *assms card-Diff-subset-Int card-mono*)

lemma *diff-card-le-card-Diff*:
assumes *finite B*
shows *card A* − *card B* ≤ *card* (*A* − *B*)
proof −
have *card A* − *card B* ≤ *card A* − *card* (*A* ∩ *B*)
using *card-mono*[*OF assms Int-lower2*, *of A*] **by** *arith*
also have ... = *card* (*A* − *B*)
using *assms* **by** (*simp add*: *card-Diff-subset-Int*)
finally show *?thesis* **.**

qed

lemma *card-le-sym-Diff*:
assumes *finite A finite B card A* ≤ *card B*
shows *card*(*A* − *B*) ≤ *card*(*B* − *A*)
proof −
have *card*(*A* − *B*) = *card A* − *card* (*A* ∩ *B*) **using** *assms*(*1*,*2*) **by**(*simp add*: *card-Diff-subset-Int*)
also have ... ≤ *card B* − *card* (*A* ∩ *B*) **using** *assms*(*3*) **by** *linarith*
also have ... = *card*(*B* − *A*) **using** *assms*(*1*,*2*) **by**(*simp add*: *card-Diff-subset-Int Int-commute*)
finally show *?thesis* **.**
qed

lemma *card-less-sym-Diff*:
assumes *finite A finite B card A* < *card B*
shows *card*(*A* − *B*) < *card*(*B* − *A*)
proof −
have *card*(*A* − *B*) = *card A* − *card* (*A* ∩ *B*) **using** *assms*(*1*,*2*) **by**(*simp add*: *card-Diff-subset-Int*)
also have ... < *card B* − *card* (*A* ∩ *B*) **using** *assms*(*1*,*3*) **by** (*simp add*: *card-mono diff-less-mono*)
also have ... = *card*(*B* − *A*) **using** *assms*(*1*,*2*) **by**(*simp add*: *card-Diff-subset-Int Int-commute*)
finally show *?thesis* **.**
qed

lemma *card-Diff1-less-iff*: *card* (*A* − {*x*}) < *card A* ⟷ *finite A* ∧ *x* ∈ *A*
proof (*cases finite A* ∧ *x* ∈ *A*)
case *True*
then show *?thesis*
by (*auto simp*: *card-gt-0-iff intro*: *diff-less*)
qed *auto*

lemma *card-Diff1-less*: *finite A* ⟹ *x* ∈ *A* ⟹ *card* (*A* − {*x*}) < *card A*
unfolding *card-Diff1-less-iff* **by** *auto*

lemma *card-Diff2-less*:
assumes *finite A x* ∈ *A y* ∈ *A* **shows** *card* (*A* − {*x*} − {*y*}) < *card A*
proof (*cases x* = *y*)
case *True*
with *assms* **show** *?thesis*
by (*simp add*: *card-Diff1-less del*: *card-Diff-insert*)
next
case *False*
then have *card* (*A* − {*x*} − {*y*}) < *card* (*A* − {*x*}) *card* (*A* − {*x*}) < *card A*
using *assms* **by** (*intro card-Diff1-less*; *simp*)+
then show *?thesis*
by (*blast intro*: *less-trans*)

qed

lemma *card-Diff1-le*: *card* $(A - \{x\}) \leq$ *card* A
proof (*cases finite A*)
  case *True*
  then show *?thesis*
    by (*cases* $x \in A$) (*simp-all add*: *card-Diff1-less less-imp-le*)
qed *auto*

lemma *card-psubset*: *finite* $B \Longrightarrow A \subseteq B \Longrightarrow$ *card* $A <$ *card* $B \Longrightarrow A < B$
  by (*erule psubsetI*) *blast*

lemma *card-add-diff-finite*:
  assumes *finite A finite B*
  shows *card* $A +$ *card* $(B-A) =$ *card* $B +$ *card* $(A-B)$
proof −
  from *assms* **have** *fi*: *finite* $(A \cap B)$
    by *simp*
  have *card* $(B-A) =$ *card* $B -$ *card* $(A \cap B)$
    using *fi card-Diff-subset-Int*[*of B A*] **by** (*simp add*: *Int-commute*)
  also have *card* $(A-B) =$ *card* $A -$ *card* $(A \cap B)$
    using *fi card-Diff-subset-Int* **by** *blast*
  moreover have *card* $A +$ (*card* $B -$ *card* $(A \cap B)) =$ *card* $B +$ (*card* $A -$ *card* $(A \cap B)$)
    using *assms card-mono* [*of concl*: $A \cap B$ B] *card-mono* [*of concl*: $A \cap B$ A]
    by (*simp flip*: *Nat.diff-add-assoc*)
  ultimately show *?thesis*
    by *simp*
qed

lemma *card-le-inj*:
  assumes *fA*: *finite A*
    and *fB*: *finite B*
    and *c*: *card* $A \leq$ *card* B
  shows $\exists f.\ f\ `\ A \subseteq B \wedge$ *inj-on f A*
  using *fA fB c*
proof (*induct arbitrary*: *B rule*: *finite-induct*)
  case *empty*
  then show *?case* **by** *simp*
next
  case (*insert x s t*)
  then show *?case*
  proof (*induct rule*: *finite-induct* [*OF insert.prems*(*1*)])
    case *1*
    then show *?case* **by** *simp*
  next
    case (*2 y t*)
    from *2.prems*(*1*,*2*,*5*) *2.hyps*(*1*,*2*) **have** *cst*: *card* $s \leq$ *card* t
      by *simp*

```
    from 2.prems(3) [OF 2.hyps(1) cst]
    obtain f where *: f ‘ s ⊆ t inj-on f s
      by blast
    let ?g = (λa. if a = x then y else f a)
    have ?g ‘ insert x s ⊆ insert y t ∧ inj-on ?g (insert x s)
      using * 2.prems(2) 2.hyps(2) unfolding inj-on-def by auto
    then show ?case by (rule exI[where ?x=?g])
  qed
qed

lemma card-subset-eq:
  assumes fB: finite B
    and AB: A ⊆ B
    and c: card A = card B
  shows A = B
proof −
  from fB AB have fA: finite A
    by (auto intro: finite-subset)
  from fA fB have fBA: finite (B − A)
    by auto
  have e: A ∩ (B − A) = {}
    by blast
  have eq: A ∪ (B − A) = B
    using AB by blast
  from card-Un-disjoint[OF fA fBA e, unfolded eq c] have card (B − A) = 0
    by arith
  then have B − A = {}
    unfolding card-eq-0-iff using fA fB by simp
  with AB show A = B
    by blast
qed

lemma insert-partition:
  x ∉ F ⟹ ∀ c1 ∈ insert x F. ∀ c2 ∈ insert x F. c1 ≠ c2 ⟶ c1 ∩ c2 = {} ⟹
x ∩ ⋃ F = {}
  by auto

lemma finite-psubset-induct [consumes 1, case-names psubset]:
  assumes finite: finite A
    and major: ⋀A. finite A ⟹ (⋀B. B ⊂ A ⟹ P B) ⟹ P A
  shows P A
  using finite
proof (induct A taking: card rule: measure-induct-rule)
  case (less A)
  have fin: finite A by fact
  have ih: card B < card A ⟹ finite B ⟹ P B for B by fact
  have P B if B ⊂ A for B
  proof −
    from that have card B < card A
```

```
    using psubset-card-mono fin by blast
   moreover
   from that have B ⊆ A
    by auto
   then have finite B
    using fin finite-subset by blast
   ultimately show ?thesis using ih by simp
  qed
  with fin show P A using major by blast
qed

lemma finite-induct-select [consumes 1, case-names empty select]:
  assumes finite S
   and P {}
   and select: ⋀T. T ⊂ S ⟹ P T ⟹ ∃s∈S − T. P (insert s T)
  shows P S
proof −
  have 0 ≤ card S by simp
  then have ∃T ⊆ S. card T = card S ∧ P T
  proof (induct rule: dec-induct)
   case base with ‹P {}›
   show ?case
    by (intro exI[of - {}]) auto
  next
   case (step n)
   then obtain T where T: T ⊆ S card T = n P T
    by auto
   with ‹n < card S› have T ⊂ S P T
    by auto
   with select[of T] obtain s where s ∈ S s ∉ T P (insert s T)
    by auto
   with step(2) T ‹finite S› show ?case
    by (intro exI[of - insert s T]) (auto dest: finite-subset)
  qed
  with ‹finite S› show P S
   by (auto dest: card-subset-eq)
qed

lemma remove-induct [case-names empty infinite remove]:
  assumes empty: P ({} :: 'a set)
   and infinite: ¬ finite B ⟹ P B
   and remove: ⋀A. finite A ⟹ A ≠ {} ⟹ A ⊆ B ⟹ (⋀x. x ∈ A ⟹ P (A
− {x})) ⟹ P A
  shows P B
proof (cases finite B)
  case False
  then show ?thesis by (rule infinite)
next
  case True
```

```
  define A where A = B
  with True have finite A A ⊆ B
    by simp-all
  then show P A
  proof (induct card A arbitrary: A)
    case 0
    then have A = {} by auto
    with empty show ?case by simp
  next
    case (Suc n A)
    from ‹A ⊆ B› and ‹finite B› have finite A
      by (rule finite-subset)
    moreover from Suc.hyps have A ≠ {} by auto
    moreover note ‹A ⊆ B›
    moreover have P (A − {x}) if x: x ∈ A for x
      using x Suc.prems ‹Suc n = card A› by (intro Suc) auto
    ultimately show ?case by (rule remove)
  qed
qed

lemma finite-remove-induct [consumes 1, case-names empty remove]:
  fixes P :: 'a set ⇒ bool
  assumes finite B
    and P {}
    and ⋀A. finite A ⟹ A ≠ {} ⟹ A ⊆ B ⟹ (⋀x. x ∈ A ⟹ P (A − {x}))
⟹ P A
  defines B' ≡ B
  shows P B'
  by (induct B' rule: remove-induct) (simp-all add: assms)
```

Main cardinality theorem.

```
lemma card-partition [rule-format]:
  finite C ⟹ finite (⋃ C) ⟹ (∀ c∈C. card c = k) ⟹
    (∀ c1 ∈ C. ∀ c2 ∈ C. c1 ≠ c2 ⟶ c1 ∩ c2 = {}) ⟹
    k * card C = card (⋃ C)
proof (induct rule: finite-induct)
  case empty
  then show ?case by simp
next
  case (insert x F)
  then show ?case
    by (simp add: card-Un-disjoint insert-partition finite-subset [of - ⋃(insert - -)])
qed

lemma card-eq-UNIV-imp-eq-UNIV:
  assumes fin: finite (UNIV :: 'a set)
    and card: card A = card (UNIV :: 'a set)
  shows A = (UNIV :: 'a set)
proof
```

```
  show A ⊆ UNIV by simp
  show UNIV ⊆ A
  proof
    show x ∈ A for x
    proof (rule ccontr)
      assume x ∉ A
      then have A ⊂ UNIV by auto
      with fin have card A < card (UNIV :: 'a set)
        by (fact psubset-card-mono)
      with card show False by simp
    qed
  qed
qed
```

The form of a finite set of given cardinality

```
lemma card-eq-SucD:
  assumes card A = Suc k
  shows ∃ b B. A = insert b B ∧ b ∉ B ∧ card B = k ∧ (k = 0 ⟶ B = {})
proof −
  have fin: finite A
    using assms by (auto intro: ccontr)
  moreover have card A ≠ 0
    using assms by auto
  ultimately obtain b where b: b ∈ A
    by auto
  show ?thesis
  proof (intro exI conjI)
    show A = insert b (A − {b})
      using b by blast
    show b ∉ A − {b}
      by blast
    show card (A − {b}) = k and k = 0 ⟶ A − {b} = {}
      using assms b fin by (fastforce dest: mk-disjoint-insert)+
  qed
qed

lemma card-Suc-eq:
  card A = Suc k ⟷
    (∃ b B. A = insert b B ∧ b ∉ B ∧ card B = k ∧ (k = 0 ⟶ B = {}))
  by (auto simp: card-insert-if card-gt-0-iff elim!: card-eq-SucD)

lemma card-Suc-eq-finite:
  card A = Suc k ⟷ (∃ b B. A = insert b B ∧ b ∉ B ∧ card B = k ∧ finite B)
  unfolding card-Suc-eq using card-gt-0-iff by fastforce

lemma card-1-singletonE:
  assumes card A = 1
  obtains x where A = {x}
  using assms by (auto simp: card-Suc-eq)
```

lemma *is-singleton-iff-card-eq-Suc-0* [*code*]:
  ‹*is-singleton A* ⟷ *card A* = *Suc 0*›
  by (*simp add*: *is-singleton-def card-Suc-eq*)

lemma *is-singleton-altdef*:
  ‹*is-singleton A* ⟷ *card A* = *1*›
  by (*simp add*: *is-singleton-iff-card-eq-Suc-0*)

lemma *card-eq-Suc-0-iff-is-singleton*:
  ‹*card A* = *Suc 0* ⟷ *is-singleton A*›
  by (*simp add*: *is-singleton-altdef*)

lemma *card-1-singleton-iff*:
  ‹*card A* = *Suc 0* ⟷ (∃ *x*. *A* = {*x*})›
  by (*simp add*: *card-eq-Suc-0-iff-is-singleton is-singleton-def*)

lemma *card-le-Suc0-iff-eq*:
  assumes *finite A*
  shows *card A* ≤ *Suc 0* ⟷ (∀ *a1* ∈ *A*. ∀ *a2* ∈ *A*. *a1* = *a2*) (**is** *?C* = *?A*)
proof
  assume *?C* **thus** *?A* **using** *assms* **by** (*auto simp*: *le-Suc-eq dest*: *card-eq-SucD*)
next
  assume *?A*
  show *?C*
  proof *cases*
    assume *A* = {} **thus** *?C* **using** ‹*?A*› **by** *simp*
  next
    assume *A* ≠ {}
    then obtain *a* **where** *A* = {*a*} **using** ‹*?A*› **by** *blast*
    thus *?C* **by** *simp*
  qed
qed

lemma *card-le-Suc-iff*:
  Suc n ≤ *card A* = (∃ *a B*. *A* = *insert a B* ∧ *a* ∉ *B* ∧ *n* ≤ *card B* ∧ *finite B*)
proof (*cases finite A*)
  case *True*
  then show *?thesis*
    by (*fastforce simp*: *card-Suc-eq less-eq-nat.simps split*: *nat.splits*)
qed *auto*

lemma *finite-fun-UNIVD2*:
  assumes *fin*: *finite* (*UNIV* :: (*'a* ⇒ *'b*) *set*)
  shows *finite* (*UNIV* :: *'b set*)
proof −
  from *fin* **have** *finite* (*range* (λ*f* :: *'a* ⇒ *'b*. *f arbitrary*)) **for** *arbitrary*
    by (*rule finite-imageI*)
  moreover have *UNIV* = *range* (λ*f* :: *'a* ⇒ *'b*. *f arbitrary*) **for** *arbitrary*

by (*rule UNIV-eq-I*) *auto*
ultimately show *finite* (*UNIV* :: *'b set*)
by *simp*
qed

lemma *card-UNIV-unit* [*simp*]: *card* (*UNIV* :: *unit set*) = *1*
unfolding *UNIV-unit* **by** *simp*

lemma *infinite-arbitrarily-large*:
assumes ¬ *finite A*
shows ∃ *B*. *finite B* ∧ *card B* = *n* ∧ *B* ⊆ *A*
proof (*induction n*)
case *0*
show *?case* **by** (*intro exI*[*of* - {}]) *auto*
next
case (*Suc n*)
then obtain *B* **where** *B*: *finite B* ∧ *card B* = *n* ∧ *B* ⊆ *A* **..**
with ‹¬ *finite A*› **have** *A* ≠ *B* **by** *auto*
with *B* **have** *B* ⊂ *A* **by** *auto*
then have ∃ *x*. *x* ∈ *A* − *B*
by (*elim psubset-imp-ex-mem*)
then obtain *x* **where** *x*: *x* ∈ *A* − *B* **..**
with *B* **have** *finite* (*insert x B*) ∧ *card* (*insert x B*) = *Suc n* ∧ *insert x B* ⊆ *A*
by *auto*
then show ∃ *B*. *finite B* ∧ *card B* = *Suc n* ∧ *B* ⊆ *A* **..**
qed

corollary *finite-arbitrarily-large-disj*:
⟦ ¬ *finite*(*UNIV*::*'a set*); *finite* (*A*::*'a set*) ⟧ ⟹ ∃ *B*. *finite B* ∧ *card B* = *n* ∧ *A* ∩ *B* = {}
using *infinite-arbitrarily-large*[*of UNIV* − *A*]
by *fastforce*

Sometimes, to prove that a set is finite, it is convenient to work with finite subsets and to show that their cardinalities are uniformly bounded. This possibility is formalized in the next criterion.

lemma *finite-if-finite-subsets-card-bdd*:
assumes ⋀*G*. *G* ⊆ *F* ⟹ *finite G* ⟹ *card G* ≤ *C*
shows *finite F* ∧ *card F* ≤ *C*
proof (*cases finite F*)
case *False*
obtain *n*::*nat* **where** *n*: *n* > *max C 0* **by** *auto*
obtain *G* **where** *G*: *G* ⊆ *F card G* = *n* **using** *infinite-arbitrarily-large*[*OF False*]
by *auto*
hence *finite G* **using** ‹*n* > *max C 0*› **using** *card.infinite gr-implies-not0* **by** *blast*
hence *False* **using** *assms G n not-less* **by** *auto*
thus *?thesis* **..**
next

```
  case True thus ?thesis using assms[of F] by auto
qed

lemma obtain-subset-with-card-n:
  assumes n ≤ card S
  obtains T where T ⊆ S card T = n finite T
proof −
  obtain n' where card S = n + n'
    using le-Suc-ex[OF assms] by blast
  with that show thesis
  proof (induct n' arbitrary: S)
    case 0
    thus ?case by (cases finite S) auto
  next
    case Suc
    thus ?case by (auto simp add: card-Suc-eq)
  qed
qed

lemma exists-subset-between:
  assumes
    card A ≤ n
    n ≤ card C
    A ⊆ C
    finite C
  shows ∃B. A ⊆ B ∧ B ⊆ C ∧ card B = n
  using assms
proof (induct n arbitrary: A C)
  case 0
  thus ?case using finite-subset[of A C] by (intro exI[of - {}], auto)
next
  case (Suc n A C)
  show ?case
  proof (cases A = {})
    case True
    from obtain-subset-with-card-n[OF Suc(3)]
    obtain B where B ⊆ C card B = Suc n .
    thus ?thesis unfolding True by blast
  next
    case False
    then obtain a where a: a ∈ A by auto
    let ?A = A − {a}
    let ?C = C − {a}
    have 1: card ?A ≤ n using Suc(2−) a
      using finite-subset by fastforce
    have 2: card ?C ≥ n using Suc(2−) a by auto
    from Suc(1)[OF 1 2 - finite-subset[OF - Suc(5)]] Suc(2−)
    obtain B where ?A ⊆ B B ⊆ ?C card B = n by blast
    thus ?thesis using a Suc(2−)
```

by (*intro exI*[*of* - *insert a B*], *auto intro*!: *card-insert-disjoint finite-subset*[*of B C*])
qed
qed

20.7.1 Cardinality of image

lemma *card-image-le*: *finite A* ⟹ *card* (*f* ‘ *A*) ≤ *card A*
by (*induct rule*: *finite-induct*) (*simp-all add*: *le-SucI card-insert-if*)

lemma *card-image*: *inj-on f A* ⟹ *card* (*f* ‘ *A*) = *card A*
proof (*induct A rule*: *infinite-finite-induct*)
case (*infinite A*)
then have ¬ *finite* (*f* ‘ *A*) **by** (*auto dest*: *finite-imageD*)
with *infinite* **show** *?case* **by** *simp*
qed *simp-all*

lemma *bij-betw-same-card*: *bij-betw f A B* ⟹ *card A* = *card B*
by (*auto simp*: *card-image bij-betw-def*)

lemma *endo-inj-surj*: *finite A* ⟹ *f* ‘ *A* ⊆ *A* ⟹ *inj-on f A* ⟹ *f* ‘ *A* = *A*
by (*simp add*: *card-seteq card-image*)

lemma *eq-card-imp-inj-on*:
assumes *finite A card*(*f* ‘ *A*) = *card A*
shows *inj-on f A*
using *assms*
proof (*induct rule*:*finite-induct*)
case *empty*
show *?case* **by** *simp*
next
case (*insert x A*)
then show *?case*
using *card-image-le* [*of A f*] **by** (*simp add*: *card-insert-if split*: *if-splits*)
qed

lemma *inj-on-iff-eq-card*: *finite A* ⟹ *inj-on f A* ⟷ *card* (*f* ‘ *A*) = *card A*
by (*blast intro*: *card-image eq-card-imp-inj-on*)

lemma *card-inj-on-le*:
assumes *inj-on f A f* ‘ *A* ⊆ *B finite B*
shows *card A* ≤ *card B*
proof −
have *finite A*
using *assms* **by** (*blast intro*: *finite-imageD dest*: *finite-subset*)
then show *?thesis*
using *assms* **by** (*force intro*: *card-mono simp*: *card-image* [*symmetric*])
qed

lemma *inj-on-iff-card-le*:
  ⟦ *finite A*; *finite B* ⟧ ⟹ (∃*f*. *inj-on f A* ∧ *f* ‘ *A* ≤ *B*) = (*card A* ≤ *card B*)
using *card-inj-on-le*[*of* - *A B*] *card-le-inj*[*of A B*] **by** *blast*

lemma *surj-card-le*: *finite A* ⟹ *B* ⊆ *f* ‘ *A* ⟹ *card B* ≤ *card A*
  by (*blast intro*: *card-image-le card-mono le-trans*)

lemma *card-le-card-if-mem-imp-ex-mem*:
  fixes *f* :: *'a* ⇒ *'b* ⇒ *'c* **and** 𝒳 :: *'a set* **and** 𝒴 :: *'c set*
  defines *XY* ≡ {(*x*, *y*). *x* ∈ 𝒳 ∧ *f x y* ∈ 𝒴}
  assumes *finite* 𝒳 **and** *finite* 𝒴 **and**
    f-inj: *inj-on* (λ(*x*, *y*). *f x y*) *XY* **and**
    ex-in-𝒴: ⋀*x*. *x* ∈ 𝒳 ⟹ ∃ *y*. *f x y* ∈ 𝒴
  shows *card* 𝒳 ≤ *card* 𝒴
proof −
  have *f-XY-subset*: (λ(*x*, *y*). *f x y*) ‘ *XY* ⊆ 𝒴
    using *XY-def* **by** *auto*

  then have *finite* ((λ(*x*, *y*). *f x y*) ‘ *XY*)
    using ‹*finite* 𝒴› **by** (*rule finite-subset*)

  then have *finite XY*
    by (*rule finite-image-iff*[*THEN iffD1*, *OF f-inj*])

  moreover have *Domain XY* = 𝒳
    unfolding *XY-def*
    using *ex-in-𝒴*
    by (*simp add*: *equalityI subsetI*)

  ultimately have *card* 𝒳 ≤ *card XY*
    using *card-Domain-le* **by** *blast*

  also have ... ≤ *card* 𝒴
    using *inj-on-iff-card-le*[*OF* ‹*finite XY*› ‹*finite* 𝒴›]
    using *f-XY-subset f-inj* **by** *blast*

  finally show *card* 𝒳 ≤ *card* 𝒴 **.**
qed

lemma *card-bij-eq*:
  inj-on f A ⟹ *f* ‘ *A* ⊆ *B* ⟹ *inj-on g B* ⟹ *g* ‘ *B* ⊆ *A* ⟹ *finite A* ⟹ *finite B*
    ⟹ *card A* = *card B*
  by (*auto intro*: *le-antisym card-inj-on-le*)

lemma *bij-betw-finite*: *bij-betw f A B* ⟹ *finite A* ⟷ *finite B*
  unfolding *bij-betw-def* **using** *finite-imageD* [*of f A*] **by** *auto*

lemma *inj-on-finite*: *inj-on f A* ⟹ *f* ‘ *A* ≤ *B* ⟹ *finite B* ⟹ *finite A*
  using *finite-imageD finite-subset* **by** *blast*

lemma *card-vimage-inj-on-le*:
assumes *inj-on f D finite A*
shows *card* $(f-`A \cap D) \leq$ *card A*
proof (*rule card-inj-on-le*)
show *inj-on f* $(f -` A \cap D)$
by (*blast intro*: *assms inj-on-subset*)
qed (*use assms* **in** *auto*)

lemma *card-vimage-inj*: *inj f* $\Longrightarrow$ $A \subseteq$ *range f* $\Longrightarrow$ *card* $(f -` A)$ = *card A*
by (*auto 4 3 simp*: *subset-image-iff inj-vimage-image-eq*
intro: *card-image*[*symmetric*, *OF inj-on-subset*])

lemma *card-inverse*[*simp*]: *card* (R^{-1}) = *card R*
proof −
have $*$: $\bigwedge R$. *prod.swap* ` *R* = R^{-1} **by** *auto*
{
assume ¬*finite R*
hence *?thesis*
by *auto*
} moreover {
assume *finite R*
with *card-image-le*[*of R prod.swap*] *card-image-le*[*of* R^{-1} *prod.swap*]
have *?thesis* **by** (*auto simp*: $*$)
} ultimately show *?thesis* **by** *blast*
qed

20.7.2 Pigeonhole Principles

lemma *pigeonhole*: *card A* > *card* $(f$ ` $A) \Longrightarrow \neg$ *inj-on f A*
by (*auto dest*: *card-image less-irrefl-nat*)

lemma *pigeonhole-infinite*:
assumes ¬ *finite A* **and** *finite* $(f`A)$
shows $\exists\, a0 \in A.$ ¬ *finite* $\{a \in A.\ f\ a = f\ a0\}$
using *assms*(*2*,*1*)
proof (*induct* $f`A$ *arbitrary*: *A rule*: *finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert b F*)
show *?case*
proof (*cases finite* $\{a \in A.\ f\ a = b\}$)
case *True*
with ‹¬ *finite A*› **have** ¬ *finite* $(A - \{a \in A.\ f\ a = b\})$
by *simp*
also have $A - \{a \in A.\ f\ a = b\} = \{a \in A.\ f\ a \neq b\}$
by *blast*
finally have ¬ *finite* $\{a \in A.\ f\ a \neq b\}$ **.**

from *insert(3)*[*OF - this*] *insert(2,4)* **show** *?thesis*
by *simp* (*blast intro*: *rev-finite-subset*)
next
case *False*
then have $\{a \in A.\ f\ a = b\} \neq \{\}$ **by** *force*
with *False* **show** *?thesis* **by** *blast*
qed
qed

lemma *pigeonhole-infinite-rel*:
assumes ¬ *finite A*
and *finite B*
and $\forall\, a{\in}A.\ \exists\, b{\in}B.\ R\ a\ b$
shows $\exists\, b{\in}B.$ ¬ *finite* $\{a{:}A.\ R\ a\ b\}$
proof −
let *?F* = $\lambda a.\ \{b{\in}B.\ R\ a\ b\}$
from *finite-Pow-iff*[*THEN iffD2*, *OF* ‹*finite B*›] **have** *finite* (*?F* ‘ *A*)
by (*blast intro*: *rev-finite-subset*)
from *pigeonhole-infinite* [**where** *f* = *?F*, *OF assms(1) this*]
obtain *a0* **where** *a0* ∈ *A* **and** *infinite*: ¬ *finite* $\{a{\in}A.$ *?F a* = *?F a0*$\}$ **..**
obtain *b0* **where** *b0* ∈ *B* **and** *R a0 b0*
using ‹*a0* ∈ *A*› *assms(3)* **by** *blast*
have *finite* $\{a{\in}A.$ *?F a* = *?F a0*$\}$ **if** *finite* $\{a{\in}A.\ R\ a\ b0\}$
using ‹*b0* ∈ *B*› ‹*R a0 b0*› *that* **by** (*blast intro*: *rev-finite-subset*)
with *infinite* ‹*b0* ∈ *B*› **show** *?thesis*
by *blast*
qed

20.7.3 Cardinality of sums

lemma *card-Plus*:
assumes *finite A finite B*
shows *card* (*A* <+> *B*) = *card A* + *card B*
proof −
have *Inl*‘*A* ∩ *Inr*‘*B* = {} **by** *fast*
with *assms* **show** *?thesis*
by (*simp add*: *Plus-def card-Un-disjoint card-image*)
qed

lemma *card-Plus-conv-if*:
card (*A* <+> *B*) = (*if finite A* ∧ *finite B then card A* + *card B else 0*)
by (*auto simp add*: *card-Plus*)

Relates to equivalence classes. Based on a theorem of F. Kammüller.

lemma *dvd-partition*:
assumes *f*: *finite* ($\bigcup C$)
and $\forall\, c{\in}C.\ k$ *dvd card c* $\forall\, c1{\in}C.\ \forall\, c2{\in}C.\ c1 \neq c2 \longrightarrow c1 \cap c2 = \{\}$
shows *k dvd card* ($\bigcup C$)
proof −

have *finite C*
by (*rule finite-UnionD* [*OF f*])
then show *?thesis*
using *assms*
proof (*induct rule*: *finite-induct*)
case *empty*
show *?case* **by** *simp*
next
case (*insert c C*)
then have $c \cap \bigcup C = \{\}$
by *auto*
with *insert* **show** *?case*
by (*simp add*: *card-Un-disjoint*)
qed
qed

20.8 Minimal and maximal elements of finite sets

context begin

qualified lemma
assumes *finite A* **and** *asymp-on A R* **and** *transp-on A R* **and** $\exists\, x \in A.\ P\ x$
shows
bex-min-element-with-property: $\exists\, x \in A.\ P\ x \wedge (\forall\, y \in A.\ R\ y\ x \longrightarrow \neg\ P\ y)$ **and**
bex-max-element-with-property: $\exists\, x \in A.\ P\ x \wedge (\forall\, y \in A.\ R\ x\ y \longrightarrow \neg\ P\ y)$
unfolding *atomize-conj*
using *assms*
proof (*induction A rule*: *finite-induct*)
case *empty*
hence *False*
by *simp-all*
thus *?case* **..**
next
case (*insert x F*)

from *insert.prems* **have** *asymp-on F R*
using *asymp-on-subset* **by** *blast*

from *insert.prems* **have** *transp-on F R*
using *transp-on-subset* **by** *blast*

show *?case*
proof (*cases P x*)
case *True*
show *?thesis*
proof (*cases* $\exists\, a{\in}F.\ P\ a$)
case *True*
with *insert.IH* **obtain** *min max* **where**
min $\in F$ **and** *P min* **and** $\forall\, z \in F.\ R\ z\ min \longrightarrow \neg\ P\ z$

```
      max ∈ F and P max and ∀ z ∈ F. R max z ⟶ ¬ P z
      using ‹asymp-on F R› ‹transp-on F R› by auto

    show ?thesis
    proof (rule conjI)
      show ∃ y ∈ insert x F. P y ∧ (∀ z ∈ insert x F. R y z ⟶ ¬ P z)
      proof (cases R max x)
        case True
        show ?thesis
        proof (intro bexI conjI ballI impI)
          show x ∈ insert x F
            by simp
        next
          show P x
            using ‹P x› by simp
        next
          fix z assume z ∈ insert x F and R x z
          hence z = x ∨ z ∈ F
            by simp
          thus ¬ P z
          proof (rule disjE)
            assume z = x
            hence R x x
              using ‹R x z› by simp
            moreover have ¬ R x x
            using ‹asymp-on (insert x F) R›[THEN irreflp-on-if-asymp-on, THEN
irreflp-onD]
              by simp
            ultimately have False
              by simp
            thus ?thesis ..
          next
            assume z ∈ F
            moreover have R max z
              using ‹R max x› ‹R x z›
              using ‹transp-on (insert x F) R›[THEN transp-onD, of max x z]
              using ‹max ∈ F› ‹z ∈ F› by simp
            ultimately show ?thesis
              using ‹∀ z ∈ F. R max z ⟶ ¬ P z› by simp
          qed
        qed
      next
        case False
        show ?thesis
        proof (intro bexI conjI ballI impI)
          show max ∈ insert x F
            using ‹max ∈ F› by simp
        next
          show P max
```

```
          using ‹P max› by simp
        next
          fix z assume z ∈ insert x F and R max z
          hence z = x ∨ z ∈ F
            by simp
          thus ¬ P z
          proof (rule disjE)
            assume z = x
            hence False
              using ‹¬ R max x› ‹R max z› by simp
            thus ?thesis ..
          next
            assume z ∈ F
            thus ?thesis
              using ‹R max z› ‹∀z∈F. R max z ⟶ ¬ P z› by simp
          qed
        qed
      qed
    next
      show ∃y ∈ insert x F. P y ∧ (∀z ∈ insert x F. R z y ⟶ ¬ P z)
      proof (cases R x min)
        case True
        show ?thesis
        proof (intro bexI conjI ballI impI)
          show x ∈ insert x F
            by simp
        next
          show P x
            using ‹P x› by simp
        next
          fix z assume z ∈ insert x F and R z x
          hence z = x ∨ z ∈ F
            by simp
          thus ¬ P z
          proof (rule disjE)
            assume z = x
            hence R x x
              using ‹R z x› by simp
            moreover have ¬ R x x
            using ‹asymp-on (insert x F) R›[THEN irreflp-on-if-asymp-on, THEN
irreflp-onD]
              by simp
            ultimately have False
              by simp
            thus ?thesis ..
          next
            assume z ∈ F
            moreover have R z min
              using ‹R z x› ‹R x min›
```

```
            using ‹transp-on (insert x F) R›[THEN transp-onD, of z x min]
            using ‹min ∈ F› ‹z ∈ F› by simp
          ultimately show ?thesis
            using ‹∀z∈F. R z min ⟶ ¬ P z› by simp
        qed
      qed
    next
      case False
      show ?thesis
      proof (intro bexI conjI ballI impI)
        show min ∈ insert x F
          using ‹min ∈ F› by simp
      next
        show P min
          using ‹P min› by simp
      next
        fix z assume z ∈ insert x F and R z min
        hence z = x ∨ z ∈ F
          by simp
        thus ¬ P z
        proof (rule disjE)
          assume z = x
          hence False
            using ‹¬ R x min› ‹R z min› by simp
          thus ?thesis ..
        next
          assume z ∈ F
          thus ?thesis
            using ‹R z min› ‹∀z∈F. R z min ⟶ ¬ P z› by simp
        qed
      qed
    qed
  next
    case False
    then show ?thesis
      using ‹∃a∈insert x F. P a›
      using ‹asymp-on (insert x F) R›[THEN asymp-onD, of x] insert-iff[of - x
F]
      by blast
  qed
next
  case False
  with insert.prems have ∃x∈F. P x
    by simp
  with insert.IH have
    ∃y∈F. P y ∧ (∀z∈F. R z y ⟶ ¬ P z)
    ∃y∈F. P y ∧ (∀z∈F. R y z ⟶ ¬ P z)
    using ‹asymp-on F R› ‹transp-on F R› by auto
```

```
    thus ?thesis
      using False by auto
  qed
qed

qualified lemma
  assumes finite A and asymp-on A R and transp-on A R and A ≠ {}
  shows
    bex-min-element: ∃ m ∈ A. ∀ x ∈ A. x ≠ m ⟶ ¬ R x m and
    bex-max-element: ∃ m ∈ A. ∀ x ∈ A. x ≠ m ⟶ ¬ R m x
  using ‹A ≠ {}›
    bex-min-element-with-property[OF assms(1,2,3), of λ-. True, simplified]
    bex-max-element-with-property[OF assms(1,2,3), of λ-. True, simplified]
  by blast+

end
```

The following alternative form might sometimes be easier to work with.

```
lemma is-min-element-in-set-iff:
  asymp-on A R ⟹ (∀ y ∈ A. y ≠ x ⟶ ¬ R y x) ⟷ (∀ y. R y x ⟶ y ∉ A)
  by (auto dest: asymp-onD)

lemma is-max-element-in-set-iff:
  asymp-on A R ⟹ (∀ y ∈ A. y ≠ x ⟶ ¬ R x y) ⟷ (∀ y. R x y ⟶ y ∉ A)
  by (auto dest: asymp-onD)

context begin

qualified lemma
  assumes finite A and A ≠ {} and transp-on A R and totalp-on A R
  shows
    bex-least-element: ∃ l ∈ A. ∀ x ∈ A. x ≠ l ⟶ R l x and
    bex-greatest-element: ∃ g ∈ A. ∀ x ∈ A. x ≠ g ⟶ R x g
  unfolding atomize-conj
  using assms
proof (induction A rule: finite-induct)
  case empty
  hence False by simp
  thus ?case ..
next
  case (insert a A')

  from insert.prems(2) have transp-on-A': transp-on A' R
    by (auto intro: transp-onI dest: transp-onD)

  from insert.prems(3) have
    totalp-on-a-A'-raw: ∀ y ∈ A'. a ≠ y ⟶ R a y ∨ R y a and
    totalp-on-A': totalp-on A' R
    by (simp-all add: totalp-on-def)
```

```
show ?case
proof (cases A' = {})
  case True
  thus ?thesis by simp
next
  case False
  then obtain least greatest where
    least ∈ A' and least-of-A': ∀x∈A'. x ≠ least ⟶ R least x and
    greatest ∈ A' and greatest-of-A': ∀x∈A'. x ≠ greatest ⟶ R x greatest
    using insert.IH[OF - transp-on-A' totalp-on-A'] by auto

  show ?thesis
  proof (rule conjI)
    show ∃l∈insert a A'. ∀x∈insert a A'. x ≠ l ⟶ R l x
    proof (cases R a least)
      case True
      show ?thesis
      proof (intro bexI ballI impI)
        show a ∈ insert a A'
          by simp
      next
        fix x
        show ⋀x. x ∈ insert a A' ⟹ x ≠ a ⟹ R a x
          using True ‹least ∈ A'› least-of-A'
          using insert.prems(2)[THEN transp-onD, of a least]
          by auto
      qed
    next
      case False
      show ?thesis
      proof (intro bexI ballI impI)
        show least ∈ insert a A'
          using ‹least ∈ A'› by simp
      next
        fix x
        show x ∈ insert a A' ⟹ x ≠ least ⟹ R least x
          using False ‹least ∈ A'› least-of-A' totalp-on-a-A'-raw
          by (cases x = a) auto
      qed
    qed
  next
    show ∃g ∈ insert a A'. ∀x ∈ insert a A'. x ≠ g ⟶ R x g
    proof (cases R greatest a)
      case True
      show ?thesis
      proof (intro bexI ballI impI)
        show a ∈ insert a A'
          by simp
```

```
      next
        fix x
        show ⋀x. x ∈ insert a A' ⟹ x ≠ a ⟹ R x a
          using True ‹greatest ∈ A'› greatest-of-A'
          using insert.prems(2)[THEN transp-onD, of - greatest a]
          by auto
      qed
    next
      case False
      show ?thesis
      proof (intro bexI ballI impI)
        show greatest ∈ insert a A'
          using ‹greatest ∈ A'› by simp
      next
        fix x
        show x ∈ insert a A' ⟹ x ≠ greatest ⟹ R x greatest
          using False ‹greatest ∈ A'› greatest-of-A' totalp-on-a-A'-raw
          by (cases x = a) auto
      qed
    qed
  qed
qed

end
```

20.8.1 Finite orders

```
context order
begin

lemma finite-has-maximal:
  assumes finite A and A ≠ {}
  shows ∃ m ∈ A. ∀ b ∈ A. m ≤ b ⟶ m = b
proof −
  obtain m where m ∈ A and m-is-max: ∀x∈A. x ≠ m ⟶ ¬ m < x
    using Finite-Set.bex-max-element[OF ‹finite A› - - ‹A ≠ {}›, of (<)] by auto
  moreover have ∀ b ∈ A. m ≤ b ⟶ m = b
    using m-is-max by (auto simp: le-less)
  ultimately show ?thesis
    by auto
qed

lemma finite-has-maximal2:
  ⟦ finite A; a ∈ A ⟧ ⟹ ∃ m ∈ A. a ≤ m ∧ (∀ b ∈ A. m ≤ b ⟶ m = b)
using finite-has-maximal[of {b ∈ A. a ≤ b}] by fastforce

lemma finite-has-minimal:
  assumes finite A and A ≠ {}
```

shows $\exists\ m \in A.\ \forall\ b \in A.\ b \leq m \longrightarrow m = b$
proof −
obtain m **where** $m \in A$ **and** *m-is-min*: $\forall x{\in}A.\ x \neq m \longrightarrow \neg\ x < m$
using *Finite-Set.bex-min-element*[*OF* ‹*finite A*› - - ‹$A \neq \{\}$›, *of* (<)] **by** *auto*
moreover have $\forall\, b \in A.\ b \leq m \longrightarrow m = b$
using *m-is-min* **by** (*auto simp*: *le-less*)
ultimately show *?thesis*
by *auto*
qed

lemma *finite-has-minimal2*:
⟦ *finite A*; $a \in A$ ⟧ $\Longrightarrow \exists\ m \in A.\ m \leq a \wedge (\forall\ b \in A.\ b \leq m \longrightarrow m = b)$
using *finite-has-minimal*[*of* $\{b \in A.\ b \leq a\}$] **by** *fastforce*

end

20.8.2 Relating injectivity and surjectivity

lemma *finite-surj-inj*:
assumes *finite A* $A \subseteq f\ `\ A$
shows *inj-on f A*
proof −
have $f\ `\ A = A$
by (*rule card-seteq* [*THEN sym*]) (*auto simp add*: *assms card-image-le*)
then show *?thesis* **using** *assms*
by (*simp add*: *eq-card-imp-inj-on*)
qed

lemma *finite-UNIV-surj-inj*: *finite*(*UNIV*:: *'a set*) $\Longrightarrow$ *surj f* $\Longrightarrow$ *inj f*
for f :: $'a \Rightarrow 'a$
by (*blast intro*: *finite-surj-inj subset-UNIV*)

lemma *finite-UNIV-inj-surj*: *finite*(*UNIV*:: *'a set*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *surj f*
for f :: $'a \Rightarrow 'a$
by (*fastforce simp*:*surj-def dest*!: *endo-inj-surj*)

lemma *surjective-iff-injective-gen*:
assumes *fS*: *finite S*
and *fT*: *finite T*
and *c*: *card S* = *card T*
and *ST*: $f\ `\ S \subseteq T$
shows $(\forall\, y \in T.\ \exists\, x \in S.\ f\, x = y) \longleftrightarrow$ *inj-on f S*
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *h*: *?lhs*
{
fix *x y*
assume *x*: $x \in S$
assume *y*: $y \in S$

```
    assume f: f x = f y
    from x fS have S0: card S ≠ 0
      by auto
    have x = y
    proof (rule ccontr)
      assume xy: ¬ ?thesis
      have th: card S ≤ card (f ‘ (S − {y}))
        unfolding c
      proof (rule card-mono)
        show finite (f ‘ (S − {y}))
          by (simp add: fS)
        have ⟦x ≠ y; x ∈ S; z ∈ S; f x = f y⟧
         ⟹ ∃ x ∈ S. x ≠ y ∧ f z = f x for z
          by (cases z = y ⟶ z = x) auto
        then show T ⊆ f ‘ (S − {y})
          using h xy x y f by fastforce
      qed
      also have  ... ≤ card (S − {y})
        by (simp add: card-image-le fS)
      also have ... ≤ card S − 1 using y fS by simp
      finally show False using S0 by arith
    qed
  }
  then show ?rhs
    unfolding inj-on-def by blast
next
  assume h: ?rhs
  have f ‘ S = T
    by (simp add: ST c card-image card-subset-eq fT h)
  then show ?lhs by blast
qed

hide-const (open) Finite-Set.fold
```

20.9 Infinite Sets

Some elementary facts about infinite sets, mostly by Stephan Merz. Beware! Because "infinite" merely abbreviates a negation, these lemmas may not work well with *blast*.

```
abbreviation infinite :: 'a set ⇒ bool
  where infinite S ≡ ¬ finite S
```

Infinite sets are non-empty, and if we remove some elements from an infinite set, the result is still infinite.

```
lemma infinite-UNIV-nat [iff]: infinite (UNIV :: nat set)
proof
  assume finite (UNIV :: nat set)
  with finite-UNIV-inj-surj [of Suc] show False
```

by *simp* (*blast dest*: *Suc-neq-Zero surjD*)
qed

lemma *infinite-UNIV-char-0*: *infinite* (*UNIV* :: *'a*::*semiring-char-0 set*)
proof
assume *finite* (*UNIV* :: *'a set*)
with *subset-UNIV* **have** *finite* (*range of-nat* :: *'a set*)
by (*rule finite-subset*)
moreover have *inj* (*of-nat* :: *nat* ⇒ *'a*)
by (*simp add*: *inj-on-def*)
ultimately have *finite* (*UNIV* :: *nat set*)
by (*rule finite-imageD*)
then show *False*
by *simp*
qed

lemma *infinite-imp-nonempty*: *infinite S* ⟹ *S* ≠ {}
by *auto*

lemma *infinite-remove*: *infinite S* ⟹ *infinite* (*S* − {*a*})
by *simp*

lemma *Diff-infinite-finite*:
assumes *finite T infinite S*
shows *infinite* (*S* − *T*)
using ‹*finite T*›
proof *induct*
from ‹*infinite S*› **show** *infinite* (*S* − {})
by *auto*
next
fix *T x*
assume *ih*: *infinite* (*S* − *T*)
have *S* − (*insert x T*) = (*S* − *T*) − {*x*}
by (*rule Diff-insert*)
with *ih* **show** *infinite* (*S* − (*insert x T*))
by (*simp add*: *infinite-remove*)
qed

lemma *Un-infinite*: *infinite S* ⟹ *infinite* (*S* ∪ *T*)
by *simp*

lemma *infinite-Un*: *infinite* (*S* ∪ *T*) ⟷ *infinite S* ∨ *infinite T*
by *simp*

lemma *infinite-super*:
assumes *S* ⊆ *T*
and *infinite S*
shows *infinite T*
proof

```
  assume finite T
  with ‹S ⊆ T› have finite S by (simp add: finite-subset)
  with ‹infinite S› show False by simp
qed

proposition infinite-coinduct [consumes 1, case-names infinite]:
  assumes X A
    and step: ⋀A. X A ⟹ ∃x∈A. X (A − {x}) ∨ infinite (A − {x})
  shows infinite A
proof
  assume finite A
  then show False
    using ‹X A›
  proof (induction rule: finite-psubset-induct)
    case (psubset A)
    then obtain x where x ∈ A X (A − {x}) ∨ infinite (A − {x})
      using local.step psubset.prems by blast
    then have X (A − {x})
      using psubset.hyps by blast
    show False
    proof (rule psubset.IH [where B = A − {x}])
      show A − {x} ⊂ A
        using ‹x ∈ A› by blast
    qed fact
  qed
qed
```

For any function with infinite domain and finite range there is some element that is the image of infinitely many domain elements. In particular, any infinite sequence of elements from a finite set contains some element that occurs infinitely often.

```
lemma inf-img-fin-dom′:
  assumes img: finite (f ‘ A)
    and dom: infinite A
  shows ∃y ∈ f ‘ A. infinite (f −‘ {y} ∩ A)
proof (rule ccontr)
  have A ⊆ (⋃y∈f ‘ A. f −‘ {y} ∩ A) by auto
  moreover assume ¬ ?thesis
  with img have finite (⋃y∈f ‘ A. f −‘ {y} ∩ A) by blast
  ultimately have finite A by (rule finite-subset)
  with dom show False by contradiction
qed

lemma inf-img-fin-domE′:
  assumes finite (f ‘ A) and infinite A
  obtains y where y ∈ f‘A and infinite (f −‘ {y} ∩ A)
  using assms by (blast dest: inf-img-fin-dom′)

lemma inf-img-fin-dom:
```

assumes *img*: *finite* (*f*‘*A*) **and** *dom*: *infinite A*
shows ∃ *y* ∈ *f*‘*A*. *infinite* (*f* −‘ {*y*})
using *inf-img-fin-dom'*[*OF assms*] **by** *auto*

lemma *inf-img-fin-domE*:
assumes *finite* (*f*‘*A*) **and** *infinite A*
obtains *y* **where** *y* ∈ *f*‘*A* **and** *infinite* (*f* −‘ {*y*})
using *assms* **by** (*blast dest*: *inf-img-fin-dom*)

proposition *finite-image-absD*: *finite* (*abs* ‘ *S*) ⟹ *finite S*
for *S* :: *'a*::*linordered-ring set*
by (*rule ccontr*) (*auto simp*: *abs-eq-iff vimage-def dest*: *inf-img-fin-dom*)

20.10 The finite powerset operator

definition *Fpow* :: *'a set* ⇒ *'a set set*
where *Fpow A* ≡ {*X*. *X* ⊆ *A* ∧ *finite X*}

lemma *Fpow-mono*: *A* ⊆ *B* ⟹ *Fpow A* ⊆ *Fpow B*
unfolding *Fpow-def* **by** *auto*

lemma *empty-in-Fpow*: {} ∈ *Fpow A*
unfolding *Fpow-def* **by** *auto*

lemma *Fpow-not-empty*: *Fpow A* ≠ {}
using *empty-in-Fpow* **by** *blast*

lemma *Fpow-subset-Pow*: *Fpow A* ⊆ *Pow A*
unfolding *Fpow-def* **by** *auto*

lemma *Fpow-Pow-finite*: *Fpow A* = *Pow A Int* {*A*. *finite A*}
unfolding *Fpow-def Pow-def* **by** *blast*

lemma *inj-on-image-Fpow*:
assumes *inj-on f A*
shows *inj-on* (*image f*) (*Fpow A*)
using *assms Fpow-subset-Pow*[*of A*] *inj-on-subset*[*of image f Pow A*]
inj-on-image-Pow **by** *blast*

lemma *image-Fpow-mono*:
assumes *f* ‘ *A* ⊆ *B*
shows (*image f*) ‘ (*Fpow A*) ⊆ *Fpow B*
using *assms* **by** (*auto simp*: *Fpow-def*)

end

21 Reflexive and Transitive closure of a relation

theory *Transitive-Closure*

imports *Finite-Set*
abbrevs $\hat{}* = {}^{*}\ {}^{**}$
and $\hat{}+ = {}^{+}\ {}^{++}$
and $\hat{}= = {}^{=}\ {}^{==}$
begin

ML-file ‹~~/*src/Provers/trancl.ML*›

rtrancl is reflexive/transitive closure, *trancl* is transitive closure, *reflcl* is reflexive closure.

These postfix operators have *maximum priority*, forcing their operands to be atomic.

context notes [[*inductive-internals*]]
begin

inductive-set *rtrancl* :: $('a \times 'a)$ *set* ⇒ $('a \times 'a)$ *set* (‹(‹*notation*=‹*postfix* ∗››-*)› [*1000*] *999*)
for r :: $('a \times 'a)$ *set*
where
rtrancl-refl [*intro*!, *Pure.intro*!, *simp*]: $(a, a) \in r^*$
| *rtrancl-into-rtrancl* [*Pure.intro*]: $(a, b) \in r^* \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^*$

inductive-set *trancl* :: $('a \times 'a)$ *set* ⇒ $('a \times 'a)$ *set* (‹(‹*notation*=‹*postfix* +››-$^{+}$)› [*1000*] *999*)
for r :: $('a \times 'a)$ *set*
where
r-into-trancl [*intro*, *Pure.intro*]: $(a, b) \in r \Longrightarrow (a, b) \in r^+$
| *trancl-into-trancl* [*Pure.intro*]: $(a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^+$

notation
rtranclp (‹(‹*notation*=‹*postfix* ∗∗››-**)› [*1000*] *1000*) **and**
tranclp (‹(‹*notation*=‹*postfix* ++››-$^{++}$)› [*1000*] *1000*)

declare
rtrancl-def [*nitpick-unfold del*]
rtranclp-def [*nitpick-unfold del*]
trancl-def [*nitpick-unfold del*]
tranclp-def [*nitpick-unfold del*]

end

lemma *trancl-incr*: $r \subseteq r^+$
by *auto*

abbreviation *reflcl* :: $('a \times 'a)$ *set* ⇒ $('a \times 'a)$ *set* (‹(‹*notation*=‹*postfix* =››-$^{=}$)› [*1000*] *999*)
where $r^= \equiv r \cup Id$

abbreviation *reflclp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ (‹(‹*notation*=‹*postfix*

$==\rangle\rangle\text{-}^{==})\rangle$ [*1000*] *1000*)
where $r^{==} \equiv sup\ r\ (=)$

notation (*ASCII*)
rtrancl (‹(‹*notation*=‹*postfix* $*\rangle\rangle\text{-}\hat{}*)\rangle$ [*1000*] *999*) **and**
trancl (‹(‹*notation*=‹*postfix* $+\rangle\rangle\text{-}\hat{}+)\rangle$ [*1000*] *999*) **and**
reflcl (‹(‹*notation*=‹*postfix* $=\rangle\rangle\text{-}\hat{}=)\rangle$ [*1000*] *999*) **and**
rtranclp (‹(‹*notation*=‹*postfix* $**\rangle\rangle\text{-}\hat{}**)\rangle$ [*1000*] *1000*) **and**
tranclp (‹(‹*notation*=‹*postfix* $++\rangle\rangle\text{-}\hat{}++)\rangle$ [*1000*] *1000*) **and**
reflclp (‹(‹*notation*=‹*postfix* $==\rangle\rangle\text{-}\hat{}==)\rangle$ [*1000*] *1000*)

bundle *rtrancl-syntax*
begin
notation
rtrancl (‹(‹*notation*=‹*postfix* $*\rangle\rangle\text{-}^{*})\rangle$ [*1000*] *999*) **and**
rtranclp (‹(‹*notation*=‹*postfix* $**\rangle\rangle\text{-}^{**})\rangle$ [*1000*] *1000*)
notation (*ASCII*)
rtrancl (‹(‹*notation*=‹*postfix* $*\rangle\rangle\text{-}\hat{}*)\rangle$ [*1000*] *999*) **and**
rtranclp (‹(‹*notation*=‹*postfix* $**\rangle\rangle\text{-}\hat{}**)\rangle$ [*1000*] *1000*)
end

bundle *trancl-syntax*
begin
notation
trancl (‹(‹*notation*=‹*postfix* $+\rangle\rangle\text{-}^{+})\rangle$ [*1000*] *999*) **and**
tranclp (‹(‹*notation*=‹*postfix* $++\rangle\rangle\text{-}^{++})\rangle$ [*1000*] *1000*)
notation (*ASCII*)
trancl (‹(‹*notation*=‹*postfix* $+\rangle\rangle\text{-}\hat{}+)\rangle$ [*1000*] *999*) **and**
tranclp (‹(‹*notation*=‹*postfix* $++\rangle\rangle\text{-}\hat{}++)\rangle$ [*1000*] *1000*)
end

bundle *reflcl-syntax*
begin
notation
reflcl (‹(‹*notation*=‹*postfix* $=\rangle\rangle\text{-}^{=})\rangle$ [*1000*] *999*) **and**
reflclp (‹(‹*notation*=‹*postfix* $==\rangle\rangle\text{-}^{==})\rangle$ [*1000*] *1000*)
notation (*ASCII*)
reflcl (‹(‹*notation*=‹*postfix* $=\rangle\rangle\text{-}\hat{}=)\rangle$ [*1000*] *999*) **and**
reflclp (‹(‹*notation*=‹*postfix* $==\rangle\rangle\text{-}\hat{}==)\rangle$ [*1000*] *1000*)
end

21.1 Reflexive closure

lemma *reflcl-set-eq* [*pred-set-conv*]: $(sup\ (\lambda x\ y.\ (x,\ y) \in r)\ (=)) = (\lambda x\ y.\ (x,\ y) \in r \cup Id)$
by (*auto simp*: *fun-eq-iff*)

lemma *refl-reflcl*[*simp*]: *refl* $(r^{=})$
by (*simp add*: *refl-on-def*)

lemma *reflp-on-reflclp*[*simp*]: *reflp-on A* $R^{==}$
by (*simp add*: *reflp-on-def*)

lemma *antisym-on-reflcl*[*simp*]: *antisym-on A* $(r^{=})$ $\longleftrightarrow$ *antisym-on A r*
by (*simp add*: *antisym-on-def*)

lemma *antisymp-on-reflclp*[*simp*]: *antisymp-on A* $R^{==}$ $\longleftrightarrow$ *antisymp-on A R*
by (*rule antisym-on-reflcl*[*to-pred*])

lemma *trans-on-reflcl*[*simp*]: *trans-on A r* $\Longrightarrow$ *trans-on A* $(r^{=})$
by (*auto intro*: *trans-onI dest*: *trans-onD*)

lemma *transp-on-reflclp*[*simp*]: *transp-on A R* $\Longrightarrow$ *transp-on A* $R^{==}$
by (*rule trans-on-reflcl*[*to-pred*])

lemma *antisymp-on-reflclp-if-asymp-on*:
assumes *asymp-on A R*
shows *antisymp-on A* $R^{==}$
unfolding *antisymp-on-reflclp*
using *antisymp-on-if-asymp-on*[*OF* ‹*asymp-on A R*›] **.**

lemma *antisym-on-reflcl-if-asym-on*: *asym-on A R* $\Longrightarrow$ *antisym-on A* $(R^{=})$
using *antisymp-on-reflclp-if-asymp-on*[*to-set*] **.**

lemma *reflclp-idemp* [*simp*]: $(P^{==})^{==} = P^{==}$
by *blast*

lemma *reflclp-ident-if-reflp*[*simp*]: *reflp R* $\Longrightarrow$ $R^{==} = R$
by (*auto dest*: *reflpD*)

The following are special cases of *reflclp-ident-if-reflp*, but they appear duplicated in multiple, independent theories, which causes name clashes.

lemma (**in** *preorder*) *reflclp-less-eq*[*simp*]: $(\leq)^{==} = (\leq)$
using *reflp-on-le* **by** (*simp only*: *reflclp-ident-if-reflp*)

lemma (**in** *preorder*) *reflclp-greater-eq*[*simp*]: $(\geq)^{==} = (\geq)$
using *reflp-on-ge* **by** (*simp only*: *reflclp-ident-if-reflp*)

lemma *order-reflclp-if-transp-and-asymp*:
assumes *transp R* **and** *asymp R*
shows *class.order* $R^{==}$ *R*
proof *unfold-locales*
show $\bigwedge x\ y.\ R\ x\ y = (R^{==}\ x\ y \wedge \neg\ R^{==}\ y\ x)$
using ‹*asymp R*› *asympD* **by** *fastforce*
next
show $\bigwedge x.\ R^{==}\ x\ x$
by *simp*
next

show $\bigwedge x\ y\ z.\ R^{==}\ x\ y \Longrightarrow R^{==}\ y\ z \Longrightarrow R^{==}\ x\ z$
using *transp-on-reflclp*[*OF* ‹*transp R*›, *THEN transpD*] .
next
show $\bigwedge x\ y.\ R^{==}\ x\ y \Longrightarrow R^{==}\ y\ x \Longrightarrow x = y$
using *antisymp-on-reflclp-if-asymp-on*[*OF* ‹*asymp R*›, *THEN antisympD*] .
qed

21.2 Reflexive-transitive closure

lemma *r-into-rtrancl* [*intro*]: $\bigwedge p.\ p \in r \Longrightarrow p \in r^*$
— *rtrancl* of *r* contains *r*
by (*simp add*: *split-tupled-all rtrancl-refl* [*THEN rtrancl-into-rtrancl*])

lemma *r-into-rtranclp* [*intro*]: $r\ x\ y \Longrightarrow r^{**}\ x\ y$
— *rtrancl* of *r* contains *r*
by (*erule rtranclp.rtrancl-refl* [*THEN rtranclp.rtrancl-into-rtrancl*])

lemma *rtranclp-mono*: $r \leq s \Longrightarrow r^{**} \leq s^{**}$
— monotonicity of *rtrancl*
proof (*rule predicate2I*)
show $s^{**}\ x\ y$ **if** $r \leq s\ r^{**}\ x\ y$ **for** $x\ y$
using ‹$r^{**}\ x\ y$› ‹$r \leq s$›
by (*induction rule*: *rtranclp.induct*) (*blast intro*: *rtranclp.rtrancl-into-rtrancl*)+
qed

lemma *mono-rtranclp*[*mono*]: $(\bigwedge a\ b.\ x\ a\ b \longrightarrow y\ a\ b) \Longrightarrow x^{**}\ a\ b \longrightarrow y^{**}\ a\ b$
using *rtranclp-mono*[*of x y*] **by** *auto*

lemmas *rtrancl-mono* = *rtranclp-mono* [*to-set*]

theorem *rtranclp-induct* [*consumes 1*, *case-names base step*, *induct set*: *rtranclp*]:
assumes *a*: $r^{**}\ a\ b$
and *cases*: $P\ a\ \bigwedge y\ z.\ r^{**}\ a\ y \Longrightarrow r\ y\ z \Longrightarrow P\ y \Longrightarrow P\ z$
shows $P\ b$
using *a* **by** (*induct x≡a b*) (*rule cases*)+

lemmas *rtrancl-induct* [*induct set*: *rtrancl*] = *rtranclp-induct* [*to-set*]

lemmas *rtranclp-induct2* =
rtranclp-induct[*of* - (*ax*,*ay*) (*bx*,*by*), *split-rule*, *consumes 1*, *case-names refl step*]

lemmas *rtrancl-induct2* =
rtrancl-induct[*of* (*ax*,*ay*) (*bx*,*by*), *split-format* (*complete*), *consumes 1*, *case-names refl step*]

lemma *refl-rtrancl*: *refl* (r^*)
unfolding *refl-on-def* **by** *fast*

Transitivity of transitive closure.

lemma *trans-rtrancl*: *trans* (r^*)
proof (*rule transI*)
  fix x y z
  assume $(x, y) \in r^*$
  assume $(y, z) \in r^*$
  then show $(x, z) \in r^*$
  proof *induct*
    case *base*
    show $(x, y) \in r^*$ **by** *fact*
  next
    case (*step* u v)
    from ‹$(x, u) \in r^*$› **and** ‹$(u, v) \in r$›
    show $(x, v) \in r^*$ **..**
  qed
qed

lemmas *rtrancl-trans* = *trans-rtrancl* [*THEN transD*]

lemma *rtranclp-trans*:
  assumes r^{**} x y
    and r^{**} y z
  shows r^{**} x z
  using *assms*(*2*,*1*) **by** *induct iprover+*

lemma *rtranclE* [*cases set*: *rtrancl*]:
  fixes a b :: $'a$
  assumes *major*: $(a, b) \in r^*$
  obtains
    (*base*) $a = b$
  | (*step*) y **where** $(a, y) \in r^*$ **and** $(y, b) \in r$
  — elimination of *rtrancl* – by induction on a special formula
proof −
  have $a = b \vee (\exists\, y.\ (a, y) \in r^* \wedge (y, b) \in r)$
    by (*rule major* [*THEN rtrancl-induct*]) *blast+*
  then show *?thesis*
    by (*auto intro*: *base step*)
qed

lemma *rtrancl-Int-subset*: $Id \subseteq s \Longrightarrow (r^* \cap s)\ O\ r \subseteq s \Longrightarrow r^* \subseteq s$
  by (*fastforce elim*: *rtrancl-induct*)

lemma *converse-rtranclp-into-rtranclp*: $r\ a\ b \Longrightarrow r^{**}\ b\ c \Longrightarrow r^{**}\ a\ c$
  by (*rule rtranclp-trans*) *iprover+*

lemmas *converse-rtrancl-into-rtrancl* = *converse-rtranclp-into-rtranclp* [*to-set*]

More r^* equations and inclusions.

lemma *rtranclp-idemp* [*simp*]: $(r^{**})^{**} = r^{**}$
proof −

have $r^{****}\ x\ y \Longrightarrow r^{**}\ x\ y$ **for** $x\ y$
by (*induction rule*: *rtranclp-induct*) (*blast intro*: *rtranclp-trans*)+
then show *?thesis*
by (*auto intro*!: *order-antisym*)
qed

lemmas *rtrancl-idemp* [*simp*] = *rtranclp-idemp* [*to-set*]

lemma *rtrancl-idemp-self-comp* [*simp*]: $R^*\ O\ R^* = R^*$
by (*force intro*: *rtrancl-trans*)

lemma *rtrancl-subset-rtrancl*: $r \subseteq s^* \Longrightarrow r^* \subseteq s^*$
by (*drule rtrancl-mono*, *simp*)

lemma *rtranclp-subset*: $R \leq S \Longrightarrow S \leq R^{**} \Longrightarrow S^{**} = R^{**}$
by (*fastforce dest*: *rtranclp-mono*)

lemmas *rtrancl-subset* = *rtranclp-subset* [*to-set*]

lemma *rtranclp-sup-rtranclp*: $(sup\ (R^{**})\ (S^{**}))^{**} = (sup\ R\ S)^{**}$
by (*blast intro*!: *rtranclp-subset intro*: *rtranclp-mono* [*THEN predicate2D*])

lemmas *rtrancl-Un-rtrancl* = *rtranclp-sup-rtranclp* [*to-set*]

lemma *rtranclp-reflclp* [*simp*]: $(R^{==})^{**} = R^{**}$
by (*blast intro*!: *rtranclp-subset*)

lemmas *rtrancl-reflcl* [*simp*] = *rtranclp-reflclp* [*to-set*]

lemma *rtrancl-r-diff-Id*: $(r - Id)^* = r^*$
by (*rule rtrancl-subset* [*symmetric*]) *auto*

lemma *rtranclp-r-diff-Id*: $(inf\ r\ (\neq))^{**} = r^{**}$
by (*rule rtranclp-subset* [*symmetric*]) *auto*

theorem *rtranclp-converseD*:
assumes $(r^{-1-1})^{**}\ x\ y$
shows $r^{**}\ y\ x$
using *assms* **by** *induct* (*iprover intro*: *rtranclp-trans dest*!: *conversepD*)+

lemmas *rtrancl-converseD* = *rtranclp-converseD* [*to-set*]

theorem *rtranclp-converseI*:
assumes $r^{**}\ y\ x$
shows $(r^{-1-1})^{**}\ x\ y$
using *assms* **by** *induct* (*iprover intro*: *rtranclp-trans conversepI*)+

lemmas *rtrancl-converseI* = *rtranclp-converseI* [*to-set*]

lemma *rtrancl-converse*: $(r^{-1})^* = (r^*)^{-1}$
by (*fast dest*!: *rtrancl-converseD intro*!: *rtrancl-converseI*)

lemma *sym-rtrancl*: *sym r* $\Longrightarrow$ *sym* (r^*)
by (*simp only*: *sym-conv-converse-eq rtrancl-converse* [*symmetric*])

theorem *converse-rtranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: r^{**} *a b*
and *cases*: *P b* $\bigwedge y$ *z*. *r y z* $\Longrightarrow$ r^{**} *z b* $\Longrightarrow$ *P z* $\Longrightarrow$ *P y*
shows *P a*
using *rtranclp-converseI* [*OF major*]
by *induct* (*iprover intro*: *cases dest*!: *conversepD rtranclp-converseD*)+

lemmas *converse-rtrancl-induct* = *converse-rtranclp-induct* [*to-set*]

lemmas *converse-rtranclp-induct2* =
converse-rtranclp-induct [*of* - (*ax*, *ay*) (*bx*, *by*), *split-rule*, *consumes 1*, *case-names refl step*]

lemmas *converse-rtrancl-induct2* =
converse-rtrancl-induct [*of* (*ax*, *ay*) (*bx*, *by*), *split-format* (*complete*),
consumes 1, *case-names refl step*]

lemma *converse-rtranclpE* [*consumes 1*, *case-names base step*]:
assumes *major*: r^{**} *x z*
and *cases*: *x* = *z* $\Longrightarrow$ *P* $\bigwedge y$. *r x y* $\Longrightarrow$ r^{**} *y z* $\Longrightarrow$ *P*
shows *P*
proof −
have *x* = *z* $\vee$ ($\exists$ *y*. *r x y* $\wedge$ r^{**} *y z*)
by (*rule major* [*THEN converse-rtranclp-induct*]) *iprover*+
then show *?thesis*
by (*auto intro*: *cases*)
qed

lemmas *converse-rtranclE* = *converse-rtranclpE* [*to-set*]

lemmas *converse-rtranclpE2* = *converse-rtranclpE* [*of* - (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemmas *converse-rtranclE2* = *converse-rtranclE* [*of* (*xa*,*xb*) (*za*,*zb*), *split-rule*]

lemma *r-comp-rtrancl-eq*: *r O* r^* = r^* *O r*
by (*blast elim*: *rtranclE converse-rtranclE*
intro: *rtrancl-into-rtrancl converse-rtrancl-into-rtrancl*)

lemma *rtrancl-unfold*: r^* = *Id* $\cup$ r^* *O r*
by (*auto intro*: *rtrancl-into-rtrancl elim*: *rtranclE*)

lemma *rtrancl-Un-separatorE*:
(*a*, *b*) $\in$ (*P* $\cup$ *Q*)* $\Longrightarrow$ $\forall$ *x y*. (*a*, *x*) $\in$ *P** $\longrightarrow$ (*x*, *y*) $\in$ *Q* $\longrightarrow$ *x* = *y* $\Longrightarrow$ (*a*, *b*)

$\in P^*$
proof (*induct rule*: *rtrancl.induct*)
case *rtrancl-refl*
then show *?case* **by** *blast*
next
case *rtrancl-into-rtrancl*
then show *?case* **by** (*blast intro*: *rtrancl-trans*)
qed

lemma *rtrancl-Un-separator-converseE*:
$(a, b) \in (P \cup Q)^* \Longrightarrow \forall x\ y.\ (x, b) \in P^* \longrightarrow (y, x) \in Q \longrightarrow y = x \Longrightarrow (a, b) \in P^*$
proof (*induct rule*: *converse-rtrancl-induct*)
case *base*
then show *?case* **by** *blast*
next
case *step*
then show *?case* **by** (*blast intro*: *rtrancl-trans*)
qed

lemma *Image-closed-trancl*:
assumes r `` $X \subseteq X$
shows r^* `` $X = X$
proof −
from *assms* **have** $**$: $\{y.\ \exists x{\in}X.\ (x, y) \in r\} \subseteq X$
by *auto*
have $x \in X$ **if** *1*: $(y, x) \in r^*$ **and** *2*: $y \in X$ **for** $x\ y$
proof −
from *1* **show** $x \in X$
proof *induct*
case *base*
show *?case* **by** (*fact 2*)
next
case *step*
with $**$ **show** *?case* **by** *auto*
qed
qed
then show *?thesis* **by** *auto*
qed

lemma *rtranclp-ident-if-reflp-and-transp*:
assumes *reflp* R **and** *transp* R
shows $R^{**} = R$
proof (*intro ext iffI*)
fix $x\ y$
show $R^{**}\ x\ y \Longrightarrow R\ x\ y$
proof (*induction y rule*: *rtranclp-induct*)
case *base*
show *?case*

```
    using ‹reflp R›[THEN reflpD] .
  next
    case (step y z)
    thus ?case
      using ‹transp R›[THEN transpD, of x y  z] by simp
  qed
next
  fix x y
  show R x y ⟹ R** x y
    using r-into-rtranclp .
qed
```

The following are special cases of *rtranclp-ident-if-reflp-and-transp*, but they appear duplicated in multiple, independent theories, which causes name clashes.

```
lemma (in preorder) rtranclp-less-eq[simp]: (≤)** = (≤)
  using reflp-on-le transp-on-le by (simp only: rtranclp-ident-if-reflp-and-transp)

lemma (in preorder) rtranclp-greater-eq[simp]: (≥)** = (≥)
  using reflp-on-ge transp-on-ge by (simp only: rtranclp-ident-if-reflp-and-transp)
```

21.3 Transitive closure

```
lemma totalp-on-tranclp: totalp-on A R ⟹ totalp-on A (tranclp R)
  by (auto intro: totalp-onI dest: totalp-onD)

lemma total-on-trancl: total-on A r ⟹ total-on A (trancl r)
  by (rule totalp-on-tranclp[to-set])

lemma trancl-mono:
  assumes p ∈ r⁺ r ⊆ s
  shows p ∈ s⁺
proof −
  have ⟦(a, b) ∈ r⁺; r ⊆ s⟧ ⟹ (a, b) ∈ s⁺ for a b
    by (induction rule: trancl.induct) (iprover dest: subsetD)+
  with assms show ?thesis
    by (cases p) force
qed

lemma trancl-mono-subset: A ⊆ B ⟹ A^+ ⊆ B^+
by (blast intro: trancl-mono)

lemma r-into-trancl': ⋀p. p ∈ r ⟹ p ∈ r⁺
  by (simp only: split-tupled-all) (erule r-into-trancl)
```

Conversions between *trancl* and *rtrancl*.

```
lemma tranclp-into-rtranclp: r⁺⁺ a b ⟹ r** a b
  by (erule tranclp.induct) iprover+
```

lemmas *trancl-into-rtrancl* = *tranclp-into-rtranclp* [*to-set*]

lemma *rtranclp-into-tranclp1*:
assumes r^{**} *a b*
shows *r b c* $\Longrightarrow$ r^{++} *a c*
using *assms* **by** (*induct arbitrary*: *c*) *iprover+*

lemmas *rtrancl-into-trancl1* = *rtranclp-into-tranclp1* [*to-set*]

lemma *rtranclp-into-tranclp2*:
assumes *r a b* r^{**} *b c* **shows** r^{++} *a c*
— intro rule from *r* and *rtrancl*
using ‹r^{**} *b c*›
proof (*cases rule*: *rtranclp.cases*)
case *rtrancl-refl*
with *assms* **show** *?thesis*
by *iprover*
next
case *rtrancl-into-rtrancl*
with *assms* **show** *?thesis*
by (*auto intro*: *rtranclp-trans* [*THEN rtranclp-into-tranclp1*])
qed

lemmas *rtrancl-into-trancl2* = *rtranclp-into-tranclp2* [*to-set*]

Nice induction rule for *trancl*

lemma *tranclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *tranclp*]:
assumes *a*: r^{++} *a b*
and *cases*: $\bigwedge y.$ *r a y* $\Longrightarrow$ *P y* $\bigwedge y\ z.$ r^{++} *a y* $\Longrightarrow$ *r y z* $\Longrightarrow$ *P y* $\Longrightarrow$ *P z*
shows *P b*
using *a* **by** (*induct x*≡*a b*) (*iprover intro*: *cases*)*+*

lemmas *trancl-induct* [*induct set*: *trancl*] = *tranclp-induct* [*to-set*]

lemmas *tranclp-induct2* =
tranclp-induct [*of* - (*ax*, *ay*) (*bx*, *by*), *split-rule*, *consumes 1*, *case-names base step*]

lemmas *trancl-induct2* =
trancl-induct [*of* (*ax*, *ay*) (*bx*, *by*), *split-format* (*complete*),
consumes 1, *case-names base step*]

lemma *tranclp-trans-induct*:
assumes *major*: r^{++} *x y*
and *cases*: $\bigwedge x\ y.$ *r x y* $\Longrightarrow$ *P x y* $\bigwedge x\ y\ z.$ r^{++} *x y* $\Longrightarrow$ *P x y* $\Longrightarrow$ r^{++} *y z* $\Longrightarrow$ *P y z* $\Longrightarrow$ *P x z*
shows *P x y*
— Another induction rule for trancl, incorporating transitivity

by (*iprover intro*: *major* [*THEN tranclp-induct*] *cases*)

lemmas *trancl-trans-induct* = *tranclp-trans-induct* [*to-set*]

lemma *tranclE* [*cases set*: *trancl*]:
assumes $(a, b) \in r^+$
obtains
(*base*) $(a, b) \in r$
| (*step*) c **where** $(a, c) \in r^+$ **and** $(c, b) \in r$
using *assms* **by** *cases simp-all*

lemma *trancl-Int-subset*: $r \subseteq s \Longrightarrow (r^+ \cap s)\ O\ r \subseteq s \Longrightarrow r^+ \subseteq s$
by (*fastforce simp add*: *elim*: *trancl-induct*)

lemma *trancl-unfold*: $r^+ = r \cup r^+\ O\ r$
by (*auto intro*: *trancl-into-trancl elim*: *tranclE*)

Transitivity of r^+

lemma *trans-trancl* [*simp*]: *trans* (r^+)
proof (*rule transI*)
fix *x y z*
assume $(x, y) \in r^+$
assume $(y, z) \in r^+$
then show $(x, z) \in r^+$
proof *induct*
case (*base u*)
from ‹$(x, y) \in r^+$› **and** ‹$(y, u) \in r$›
show $(x, u) \in r^+$ **..**
next
case (*step u v*)
from ‹$(x, u) \in r^+$› **and** ‹$(u, v) \in r$›
show $(x, v) \in r^+$ **..**
qed
qed

lemmas *trancl-trans* = *trans-trancl* [*THEN transD*]

lemma *tranclp-trans*:
assumes $r^{++}\ x\ y$
and $r^{++}\ y\ z$
shows $r^{++}\ x\ z$
using *assms*(*2*,*1*) **by** *induct iprover+*

lemma *trancl-id* [*simp*]: *trans* $r \Longrightarrow r^+ = r$
unfolding *trans-def* **by** (*fastforce simp add*: *elim*: *trancl-induct*)

lemma *rtranclp-tranclp-tranclp*:
assumes $r^{**}\ x\ y$
shows $\bigwedge z.\ r^{++}\ y\ z \Longrightarrow r^{++}\ x\ z$

using *assms* **by** *induct* (*iprover intro*: *tranclp-trans*)+

lemmas *rtrancl-trancl-trancl* = *rtranclp-tranclp-tranclp* [*to-set*]

lemma *tranclp-into-tranclp2*: *r a b* $\Longrightarrow$ r^{++} *b c* $\Longrightarrow$ r^{++} *a c*
by (*erule tranclp-trans* [*OF tranclp.r-into-trancl*])

lemmas *trancl-into-trancl2* = *tranclp-into-tranclp2* [*to-set*]

lemma *trancl-trancl-Un*: $(A\hat{}+ \cup B)\hat{}+ = (A \cup B)\hat{}+$
proof
show $(A^+ \cup B)^+ \subseteq (A \cup B)^+$
using *trancl-id*[*OF trans-trancl*] *trancl-incr*[*of* $A \cup B$]
trancl-mono-subset[*of* A $(A \cup B)^+$] *trancl-mono-subset*[*of* $A^+ \cup B$ $(A \cup B)^+$]
by *blast*
show $(A \cup B)^+ \subseteq (A^+ \cup B)^+$
using *trancl-incr*[*of* A] *trancl-mono-subset*[*OF sup-mono*] **by** *blast*
qed

lemma *trancl-absorb-subset-trancl*: $B \subseteq A\hat{}+ \Longrightarrow (A \cup B)\hat{}+ = A\hat{}+$
using *trancl-trancl-Un*[*of A B*] *sup.order-iff*[*of B* A^+] **by** *auto*

lemma *tranclp-converseI*:
assumes $(r^{++})^{-1-1}$ *x y* **shows** $(r^{-1-1})^{++}$ *x y*
using *conversepD* [*OF assms*]
proof (*induction rule*: *tranclp-induct*)
case (*base y*)
then show *?case*
by (*iprover intro*: *conversepI*)
next
case (*step y z*)
then show *?case*
by (*iprover intro*: *conversepI tranclp-trans*)
qed

lemmas *trancl-converseI* = *tranclp-converseI* [*to-set*]

lemma *tranclp-converseD*:
assumes $(r^{-1-1})^{++}$ *x y* **shows** $(r^{++})^{-1-1}$ *x y*
proof −
have r^{++} *y x*
using *assms*
by (*induction rule*: *tranclp-induct*) (*iprover dest*: *conversepD intro*: *tranclp-trans*)+
then show *?thesis*
by (*rule conversepI*)
qed

lemmas *trancl-converseD* = *tranclp-converseD* [*to-set*]

lemma *tranclp-converse*: $(r^{-1-1})^{++} = (r^{++})^{-1-1}$
by (*fastforce simp add*: *fun-eq-iff intro*!: *tranclp-converseI dest*!: *tranclp-converseD*)

lemmas *trancl-converse* = *tranclp-converse* [*to-set*]

lemma *sym-trancl*: *sym r* $\Longrightarrow$ *sym* (r^{+})
by (*simp only*: *sym-conv-converse-eq trancl-converse* [*symmetric*])

lemma *converse-tranclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: r^{++} *a b*
and *cases*: $\bigwedge y.$ *r y b* $\Longrightarrow$ *P y* $\bigwedge y\ z.$ *r y z* $\Longrightarrow$ r^{++} *z b* $\Longrightarrow$ *P z* $\Longrightarrow$ *P y*
shows *P a*
proof −
have r^{-1-1++} *b a*
by (*intro tranclp-converseI conversepI major*)
then show *?thesis*
by (*induction rule*: *tranclp-induct*) (*blast intro*: *cases dest*: *tranclp-converseD*)+
qed

lemmas *converse-trancl-induct* = *converse-tranclp-induct* [*to-set*]

lemma *tranclpD*: R^{++} *x y* $\Longrightarrow$ $\exists z.$ *R x z* $\wedge$ R^{**} *z y*
proof (*induction rule*: *converse-tranclp-induct*)
case (*step u v*)
then show *?case*
by (*blast intro*: *rtranclp-trans*)
qed *auto*

lemmas *tranclD* = *tranclpD* [*to-set*]

lemma *converse-tranclpE*:
assumes *major*: *tranclp r x z*
and *base*: *r x z* $\Longrightarrow$ *P*
and *step*: $\bigwedge y.$ *r x y* $\Longrightarrow$ *tranclp r y z* $\Longrightarrow$ *P*
shows *P*
proof −
from *tranclpD* [*OF major*] **obtain** *y* **where** *r x y* **and** *rtranclp r y z*
by *iprover*
from *this*(*2*) **show** *P*
proof (*cases rule*: *rtranclp.cases*)
case *rtrancl-refl*
with ‹*r x y*› *base* **show** *P*
by *iprover*
next
case *rtrancl-into-rtrancl*
then have *tranclp r y z*
by (*iprover intro*: *rtranclp-into-tranclp1*)
with ‹*r x y*› *step* **show** *P*
by *iprover*

qed
qed

lemmas *converse-tranclE* = *converse-tranclpE* [*to-set*]

lemma *tranclD2*: $(x, y) \in R^+ \Longrightarrow \exists z.\ (x, z) \in R^* \wedge (z, y) \in R$
by (*blast elim*: *tranclE intro*: *trancl-into-rtrancl*)

lemma *irrefl-tranclI*: $r^{-1} \cap r^* = \{\} \Longrightarrow (x, x) \notin r^+$
by (*blast elim*: *tranclE dest*: *trancl-into-rtrancl*)

lemma *irrefl-trancl-rD*: $\forall x.\ (x, x) \notin r^+ \Longrightarrow (x, y) \in r \Longrightarrow x \neq y$
by (*blast dest*: *r-into-trancl*)

lemma *trancl-subset-Sigma-aux*: $(a, b) \in r^* \Longrightarrow r \subseteq A \times A \Longrightarrow a = b \vee a \in A$
by (*induct rule*: *rtrancl-induct*) *auto*

lemma *trancl-subset-Sigma*:
assumes $r \subseteq A \times A$ **shows** $r^+ \subseteq A \times A$
proof (*rule trancl-Int-subset* [*OF assms*])
show $(r^+ \cap A \times A)\ O\ r \subseteq A \times A$
using *assms* **by** *auto*
qed

lemma *reflclp-tranclp* [*simp*]: $(r^{++})^{==} = r^{**}$
by (*fast elim*: *rtranclp.cases tranclp-into-rtranclp dest*: *rtranclp-into-tranclp1*)

lemmas *reflcl-trancl* [*simp*] = *reflclp-tranclp* [*to-set*]

lemma *trancl-reflcl* [*simp*]: $(r^=)^+ = r^*$
proof −
have $(a, b) \in (r^=)^+ \Longrightarrow (a, b) \in r^*$ **for** *a b*
by (*force dest*: *trancl-into-rtrancl*)
moreover have $(a, b) \in (r^=)^+$ **if** $(a, b) \in r^*$ **for** *a b*
using *that*
proof (*cases a b rule*: *rtranclE*)
case *step*
show *?thesis*
by (*rule rtrancl-into-trancl1*) (*use step* **in** *auto*)
qed *auto*
ultimately show *?thesis*
by *auto*
qed

lemma *rtrancl-trancl-reflcl* [*code*]: $r^* = (r^+)^=$
by *simp*

lemma *trancl-empty* [*simp*]: $\{\}^+ = \{\}$
by (*auto elim*: *trancl-induct*)

lemma *rtrancl-empty* [*simp*]: $\{\}^* = Id$
by (*rule subst* [*OF reflcl-trancl*]) *simp*

lemma *rtrancl--Id*[*simp*]: $Id^* = Id$
using *rtrancl-empty rtrancl-idemp*[*of* $\{\}$] **by** (*simp*)

lemma *rtranclpD*: $R^{**}\ a\ b \Longrightarrow a = b \vee a \neq b \wedge R^{++}\ a\ b$
by (*force simp*: *reflclp-tranclp* [*symmetric*] *simp del*: *reflclp-tranclp*)

lemmas *rtranclD* = *rtranclpD* [*to-set*]

lemma *rtrancl-eq-or-trancl*: $(x,y) \in R^* \longleftrightarrow x = y \vee x \neq y \wedge (x,\ y) \in R^+$
by (*fast elim*: *trancl-into-rtrancl dest*: *rtranclD*)

lemma *trancl-unfold-right*: $r^+ = r^*\ O\ r$
by (*auto dest*: *tranclD2 intro*: *rtrancl-into-trancl1*)

lemma *trancl-unfold-left*: $r^+ = r\ O\ r^*$
by (*auto dest*: *tranclD intro*: *rtrancl-into-trancl2*)

lemma *tranclp-unfold-left*: $r\hat{}{+}{+} = r\ OO\ r\hat{}{*}{*}$
by (*auto intro*!: *ext dest*: *tranclpD intro*: *rtranclp-into-tranclp2*)

lemma *trancl-insert*: $(insert\ (y,\ x)\ r)^+ = r^+ \cup \{(a,\ b).\ (a,\ y) \in r^* \wedge (x,\ b) \in r^*\}$
— primitive recursion for *trancl* over finite relations
proof −
have $\bigwedge a\ b.\ (a,\ b) \in (insert\ (y,\ x)\ r)^+ \Longrightarrow$
$(a,\ b) \in r^+ \cup \{(a,\ b).\ (a,\ y) \in r^* \wedge (x,\ b) \in r^*\}$
by (*erule trancl-induct*) (*blast intro*: *rtrancl-into-trancl1 trancl-into-rtrancl trancl-trans*)+
moreover have $r^+ \cup \{(a,\ b).\ (a,\ y) \in r^* \wedge (x,\ b) \in r^*\} \subseteq (insert\ (y,\ x)\ r)^+$
by (*blast intro*: *trancl-mono rtrancl-mono* [*THEN* [*2*] *rev-subsetD*]
rtrancl-trancl-trancl rtrancl-into-trancl2)
ultimately show *?thesis*
by *auto*
qed

lemma *trancl-insert2*:
$(insert\ (a,\ b)\ r)^+ = r^+ \cup \{(x,\ y).\ ((x,\ a) \in r^+ \vee x = a) \wedge ((b,\ y) \in r^+ \vee y = b)\}$
by (*auto simp*: *trancl-insert rtrancl-eq-or-trancl*)

lemma *rtrancl-insert*: $(insert\ (a,b)\ r)^* = r^* \cup \{(x,\ y).\ (x,\ a) \in r^* \wedge (b,\ y) \in r^*\}$
using *trancl-insert*[*of a b r*]
by (*simp add*: *rtrancl-trancl-reflcl del*: *reflcl-trancl*) *blast*

Simplifying nested closures

lemma *rtrancl-trancl-absorb*[*simp*]: $(R^*)^+ = R^*$

by (*simp add*: *trans-rtrancl*)

lemma *trancl-rtrancl-absorb*[*simp*]: $(R^+)^* = R^*$
by (*subst reflcl-trancl*[*symmetric*]) *simp*

lemma *rtrancl-reflcl-absorb*[*simp*]: $(R^*)^= = R^*$
by *auto*

Domain and *Range*

lemma *Domain-rtrancl* [*simp*]: *Domain* $(R^*) = UNIV$
by *blast*

lemma *Range-rtrancl* [*simp*]: *Range* $(R^*) = UNIV$
by *blast*

lemma *rtrancl-Un-subset*: $(R^* \cup S^*) \subseteq (R \cup S)^*$
by (*rule rtrancl-Un-rtrancl* [*THEN subst*]) *fast*

lemma *in-rtrancl-UnI*: $x \in R^* \vee x \in S^* \Longrightarrow x \in (R \cup S)^*$
by (*blast intro*: *subsetD* [*OF rtrancl-Un-subset*])

lemma *trancl-domain* [*simp*]: *Domain* $(r^+) =$ *Domain r*
by (*unfold Domain-unfold*) (*blast dest*: *tranclD*)

lemma *trancl-range* [*simp*]: *Range* $(r^+) =$ *Range r*
unfolding *Domain-converse* [*symmetric*] **by** (*simp add*: *trancl-converse* [*symmetric*])

lemma *Not-Domain-rtrancl*:
assumes $x \notin$ *Domain R* **shows** $(x, y) \in R^* \longleftrightarrow x = y$
proof −
have $(x, y) \in R^* \Longrightarrow x = y$
by (*erule rtrancl-induct*) (*use assms* **in** *auto*)
then show *?thesis*
by *auto*
qed

lemma *trancl-subset-Field2*: $r^+ \subseteq$ *Field r* × *Field r*
by (*rule trancl-Int-subset*) (*auto simp*: *Field-def*)

lemma *finite-trancl*[*simp*]: *finite* $(r^+) =$ *finite r*
proof
show *finite* $(r^+) \Longrightarrow$ *finite r*
by (*blast intro*: *r-into-trancl′ finite-subset*)
show *finite r* $\Longrightarrow$ *finite* (r^+)
by (*auto simp*: *finite-Field trancl-subset-Field2* [*THEN finite-subset*])
qed

lemma *finite-rtrancl-Image*[*simp*]: **assumes** *finite R finite A* **shows** *finite* $(R^* \text{ `` } A)$

proof (*rule ccontr*)
  assume *infinite* (R^* `` A)
  with *assms* **show** *False*
    by(*simp add*: *rtrancl-trancl-reflcl Un-Image del*: *reflcl-trancl*)
qed

More about converse *rtrancl* and *trancl*, should be merged with main body.

lemma *single-valued-confluent*:
  assumes *single-valued r* **and** *xy*: $(x, y) \in r^*$ **and** *xz*: $(x, z) \in r^*$
  shows $(y, z) \in r^* \vee (z, y) \in r^*$
  using *xy*
proof (*induction rule*: *rtrancl-induct*)
  case *base*
  show *?case*
    by (*simp add*: *assms*)
next
  case (*step y z*)
  with *xz* ‹*single-valued r*› **show** *?case*
    by (*auto elim*: *converse-rtranclE dest*: *single-valuedD intro*: *rtrancl-trans*)
qed

lemma *r-r-into-trancl*: $(a, b) \in R \Longrightarrow (b, c) \in R \Longrightarrow (a, c) \in R^+$
  by (*fast intro*: *trancl-trans*)

lemma *trancl-into-trancl*: $(a, b) \in r^+ \Longrightarrow (b, c) \in r \Longrightarrow (a, c) \in r^+$
  by (*induct rule*: *trancl-induct*) (*fast intro*: *r-r-into-trancl trancl-trans*)+

lemma *tranclp-rtranclp-tranclp*:
  assumes r^{++} *a b* r^{**} *b c* **shows** r^{++} *a c*
proof −
  obtain *z* **where** *r a z* r^{**} *z c*
    using *assms* **by** (*iprover dest*: *tranclpD rtranclp-trans*)
  then show *?thesis*
    by (*blast dest*: *rtranclp-into-tranclp2*)
qed

lemma *rtranclp-conversep*: $r^{-1-1**} = r^{**-1-1}$
  by(*auto simp add*: *fun-eq-iff intro*: *rtranclp-converseI rtranclp-converseD*)

lemmas *symp-rtranclp* = *sym-rtrancl*[*to-pred*]

lemmas *symp-conv-conversep-eq* = *sym-conv-converse-eq*[*to-pred*]

lemmas *rtranclp-tranclp-absorb* [*simp*] = *rtrancl-trancl-absorb*[*to-pred*]
lemmas *tranclp-rtranclp-absorb* [*simp*] = *trancl-rtrancl-absorb*[*to-pred*]
lemmas *rtranclp-reflclp-absorb* [*simp*] = *rtrancl-reflcl-absorb*[*to-pred*]

lemmas *trancl-rtrancl-trancl* = *tranclp-rtranclp-tranclp* [*to-set*]

lemmas *transitive-closure-trans* [*trans*] =
r-r-into-trancl trancl-trans rtrancl-trans
trancl.trancl-into-trancl trancl-into-trancl2
rtrancl.rtrancl-into-rtrancl converse-rtrancl-into-rtrancl
rtrancl-trancl-trancl trancl-rtrancl-trancl

lemmas *transitive-closurep-trans'* [*trans*] =
tranclp-trans rtranclp-trans
tranclp.trancl-into-trancl tranclp-into-tranclp2
rtranclp.rtrancl-into-rtrancl converse-rtranclp-into-rtranclp
rtranclp-tranclp-tranclp tranclp-rtranclp-tranclp

declare *trancl-into-rtrancl* [*elim*]

lemma *tranclp-ident-if-transp*:
assumes *transp R*
shows $R^{++} = R$
proof (*intro ext iffI*)
fix *x y*
show R^{++} *x y* $\Longrightarrow$ *R x y*
proof (*induction y rule*: *tranclp-induct*)
case (*base y*)
thus *?case*
by *simp*
next
case (*step y z*)
thus *?case*
using ‹*transp R*›[*THEN transpD*, *of x y z*] **by** *simp*
qed
next
fix *x y*
show *R x y* $\Longrightarrow$ R^{++} *x y*
using *tranclp.r-into-trancl* **.**
qed

The following are special cases of *tranclp-ident-if-transp*, but they appear duplicated in multiple, independent theories, which causes name clashes.

lemma (**in** *preorder*) *tranclp-less*[*simp*]: $(<)^{++} = (<)$
using *transp-on-less* **by** (*simp only*: *tranclp-ident-if-transp*)

lemma (**in** *preorder*) *tranclp-less-eq*[*simp*]: $(\leq)^{++} = (\leq)$
using *transp-on-le* **by** (*simp only*: *tranclp-ident-if-transp*)

lemma (**in** *preorder*) *tranclp-greater*[*simp*]: $(>)^{++} = (>)$
using *transp-on-greater* **by** (*simp only*: *tranclp-ident-if-transp*)

lemma (**in** *preorder*) *tranclp-greater-eq*[*simp*]: $(\geq)^{++} = (\geq)$
using *transp-on-ge* **by** (*simp only*: *tranclp-ident-if-transp*)

21.4 Symmetric closure

definition *symclp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$
where $symclp\ r\ x\ y \longleftrightarrow r\ x\ y \vee r\ y\ x$

lemma *symclpI* [*simp*, *intro?*]:
shows *symclpI1*: $r\ x\ y \Longrightarrow symclp\ r\ x\ y$
and *symclpI2*: $r\ y\ x \Longrightarrow symclp\ r\ x\ y$
by(*simp-all add*: *symclp-def*)

lemma *symclpE* [*consumes 1*, *cases pred*]:
assumes $symclp\ r\ x\ y$
obtains (*base*) $r\ x\ y$ | (*sym*) $r\ y\ x$
using *assms* **by**(*auto simp add*: *symclp-def*)

lemma *symclp-pointfree*: $symclp\ r = sup\ r\ r^{-1-1}$
by(*auto simp add*: *symclp-def fun-eq-iff*)

lemma *symclp-greater*: $r \leq symclp\ r$
by(*simp add*: *symclp-pointfree*)

lemma *symclp-conversep* [*simp*]: $symclp\ r^{-1-1} = symclp\ r$
by(*simp add*: *symclp-pointfree sup.commute*)

lemma *symp-on-symclp* [*simp*]: *symp-on A* (*symclp R*)
by(*auto simp add*: *symp-on-def elim*: *symclpE intro*: *symclpI*)

lemma *symp-symclp-eq*: $symp\ r \Longrightarrow symclp\ r = r$
by(*simp add*: *symclp-pointfree symp-conv-conversep-eq*)

lemma *symp-rtranclp-symclp* [*simp*]: $symp\ (symclp\ r)^{**}$
by(*simp add*: *symp-rtranclp*)

lemma *rtranclp-symclp-sym* [*sym*]: $(symclp\ r)^{**}\ x\ y \Longrightarrow (symclp\ r)^{**}\ y\ x$
by(*rule sympD*[*OF symp-rtranclp-symclp*])

lemma *symclp-idem* [*simp*]: $symclp\ (symclp\ r) = symclp\ r$
by(*simp add*: *symclp-pointfree sup-commute converse-join*)

lemma *reflp-on-rtranclp* [*simp*]: *reflp-on A* R^{**}
by (*simp add*: *reflp-on-def*)

21.5 The power operation on relations

$R \frown n = R\ O\ \ldots\ O\ R$, the n-fold composition of R

overloading
$relpow \equiv compow :: nat \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$
$relpowp \equiv compow :: nat \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
begin

primrec *relpow* :: *nat* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*
where
relpow 0 R = *Id*
| *relpow* (*Suc n*) *R* = (*R* ^^ *n*) *O R*

primrec *relpowp* :: *nat* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*)
where
relpowp 0 R = *HOL.eq*
| *relpowp* (*Suc n*) *R* = (*R* ^^ *n*) *OO R*

end

lemmas *relpowp-Suc-right* = *relpowp.simps*(*2*)

lemma *relpowp-relpow-eq* [*pred-set-conv*]:
(λ*x y*. (*x*, *y*) ∈ *R*) ^^ *n* = (λ*x y*. (*x*, *y*) ∈ *R* ^^ *n*) **for** *R* :: *'a rel*
by (*induct n*) (*simp-all add*: *relcompp-relcomp-eq*)

For code generation:

definition *relpow* :: *nat* ⇒ (*'a* × *'a*) *set* ⇒ (*'a* × *'a*) *set*
where *relpow-code-def* [*code-abbrev*]: *relpow* = *compow*

definition *relpowp* :: *nat* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *'a* ⇒ *bool*)
where *relpowp-code-def* [*code-abbrev*]: *relpowp* = *compow*

lemma [*code*]:
relpow 0 R = *Id*
relpow (*Suc n*) *R* = *relpow n R O R*
by (*simp-all add*: *relpow-code-def*)

lemma [*code*]:
relpowp 0 R = *HOL.eq*
relpowp (*Suc n*) *R* = *relpowp n R OO R*
by (*simp-all add*: *relpowp-code-def*)

hide-const (**open**) *relpow*
hide-const (**open**) *relpowp*

lemma *relpow-1* [*simp*]: *R* ^^ *1* = *R*
for *R* :: (*'a* × *'a*) *set*
by *simp*

lemma *relpowp-1* [*simp*]: *P* ^^ *1* = *P*
for *P* :: *'a* ⇒ *'a* ⇒ *bool*
by (*fact relpow-1* [*to-pred*])

lemma *relpowp-Suc-0* [*simp*]: *P* ^^ (*Suc 0*) = *P*
for *P* :: *'a* ⇒ *'a* ⇒ *bool*

by (*auto*)

lemma *relpow-0-I*: $(x, x) \in R \mathbin{\frown\frown} 0$
by *simp*

lemma *relpowp-0-I*: $(P \mathbin{\frown\frown} 0)\ x\ x$
by (*fact relpow-0-I* [*to-pred*])

lemma *relpow-Suc-I*: $(x, y) \in R \mathbin{\frown\frown} n \Longrightarrow (y, z) \in R \Longrightarrow (x, z) \in R \mathbin{\frown\frown} Suc\ n$
by *auto*

lemma *relpowp-Suc-I*[*trans*]: $(P \mathbin{\frown\frown} n)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow (P \mathbin{\frown\frown} Suc\ n)\ x\ z$
by (*fact relpow-Suc-I* [*to-pred*])

lemma *relpow-Suc-I2*: $(x, y) \in R \Longrightarrow (y, z) \in R \mathbin{\frown\frown} n \Longrightarrow (x, z) \in R \mathbin{\frown\frown} Suc\ n$
by (*induct n arbitrary*: *z*) (*simp*, *fastforce*)

lemma *relpowp-Suc-I2*[*trans*]: $P\ x\ y \Longrightarrow (P \mathbin{\frown\frown} n)\ y\ z \Longrightarrow (P \mathbin{\frown\frown} Suc\ n)\ x\ z$
by (*fact relpow-Suc-I2* [*to-pred*])

lemma *relpow-0-E*: $(x, y) \in R \mathbin{\frown\frown} 0 \Longrightarrow (x = y \Longrightarrow P) \Longrightarrow P$
by *simp*

lemma *relpowp-0-E*: $(P \mathbin{\frown\frown} 0)\ x\ y \Longrightarrow (x = y \Longrightarrow Q) \Longrightarrow Q$
by (*fact relpow-0-E* [*to-pred*])

lemma *relpow-Suc-E*: $(x, z) \in R \mathbin{\frown\frown} Suc\ n \Longrightarrow (\bigwedge y.\ (x, y) \in R \mathbin{\frown\frown} n \Longrightarrow (y, z) \in R \Longrightarrow P) \Longrightarrow P$
by *auto*

lemma *relpowp-Suc-E*: $(P \mathbin{\frown\frown} Suc\ n)\ x\ z \Longrightarrow (\bigwedge y.\ (P \mathbin{\frown\frown} n)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow Q) \Longrightarrow Q$
by (*fact relpow-Suc-E* [*to-pred*])

lemma *relpow-E*:
$(x, z) \in R \mathbin{\frown\frown} n \Longrightarrow$
$(n = 0 \Longrightarrow x = z \Longrightarrow P) \Longrightarrow$
$(\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (x, y) \in R \mathbin{\frown\frown} m \Longrightarrow (y, z) \in R \Longrightarrow P) \Longrightarrow P$
by (*cases n*) *auto*

lemma *relpowp-E*:
$(P \mathbin{\frown\frown} n)\ x\ z \Longrightarrow$
$(n = 0 \Longrightarrow x = z \Longrightarrow Q) \Longrightarrow$
$(\bigwedge y\ m.\ n = Suc\ m \Longrightarrow (P \mathbin{\frown\frown} m)\ x\ y \Longrightarrow P\ y\ z \Longrightarrow Q) \Longrightarrow Q$
by (*fact relpow-E* [*to-pred*])

lemma *relpow-Suc-D2*: $(x, z) \in R \mathbin{\frown\frown} Suc\ n \Longrightarrow (\exists y.\ (x, y) \in R \wedge (y, z) \in R \mathbin{\frown\frown} n)$
by (*induct n arbitrary*: *x z*)

(*blast intro*: *relpow-0-I relpow-Suc-I elim*: *relpow-0-E relpow-Suc-E*)+

lemma *relpowp-Suc-D2*: $(P \mathbin{\frown\frown} \mathit{Suc}\ n)\ x\ z \Longrightarrow \exists y.\ P\ x\ y \wedge (P \mathbin{\frown\frown} n)\ y\ z$
by (*fact relpow-Suc-D2* [*to-pred*])

lemma *relpow-Suc-E2*: $(x,\ z) \in R \mathbin{\frown\frown} \mathit{Suc}\ n \Longrightarrow (\bigwedge y.\ (x,\ y) \in R \Longrightarrow (y,\ z) \in R \mathbin{\frown\frown} n \Longrightarrow P) \Longrightarrow P$
by (*blast dest*: *relpow-Suc-D2*)

lemma *relpowp-Suc-E2*: $(P \mathbin{\frown\frown} \mathit{Suc}\ n)\ x\ z \Longrightarrow (\bigwedge y.\ P\ x\ y \Longrightarrow (P \mathbin{\frown\frown} n)\ y\ z \Longrightarrow Q) \Longrightarrow Q$
by (*fact relpow-Suc-E2* [*to-pred*])

lemma *relpow-Suc-D2'*: $\forall x\ y\ z.\ (x,\ y) \in R \mathbin{\frown\frown} n \wedge (y,\ z) \in R \longrightarrow (\exists w.\ (x,\ w) \in R \wedge (w,\ z) \in R \mathbin{\frown\frown} n)$
by (*induct n*) (*simp-all*, *blast*)

lemma *relpowp-Suc-D2'*: $\forall x\ y\ z.\ (P \mathbin{\frown\frown} n)\ x\ y \wedge P\ y\ z \longrightarrow (\exists w.\ P\ x\ w \wedge (P \mathbin{\frown\frown} n)\ w\ z)$
by (*fact relpow-Suc-D2'* [*to-pred*])

lemma *relpow-E2*:
assumes $(x,\ z) \in R \mathbin{\frown\frown} n$ $n = 0 \Longrightarrow x = z \Longrightarrow P$
$\bigwedge y\ m.\ n = \mathit{Suc}\ m \Longrightarrow (x,\ y) \in R \Longrightarrow (y,\ z) \in R \mathbin{\frown\frown} m \Longrightarrow P$
shows *P*
proof (*cases n*)
case *0*
with *assms* **show** *?thesis*
by *simp*
next
case (*Suc m*)
with *assms relpow-Suc-D2'* [*of m R*] **show** *?thesis*
by *force*
qed

lemma *relpowp-E2*:
$(P \mathbin{\frown\frown} n)\ x\ z \Longrightarrow$
$(n = 0 \Longrightarrow x = z \Longrightarrow Q) \Longrightarrow$
$(\bigwedge y\ m.\ n = \mathit{Suc}\ m \Longrightarrow P\ x\ y \Longrightarrow (P \mathbin{\frown\frown} m)\ y\ z \Longrightarrow Q) \Longrightarrow Q$
by (*fact relpow-E2* [*to-pred*])

lemma *relpowp-trans*[*trans*]: $(R \mathbin{\frown\frown} i)\ x\ y \Longrightarrow (R \mathbin{\frown\frown} j)\ y\ z \Longrightarrow (R \mathbin{\frown\frown} (i + j))\ x\ z$
proof (*induction i arbitrary*: *x*)
case *0*
thus *?case* **by** *simp*
next
case (*Suc i*)
obtain x' **where** $R\ x\ x'$ **and** $(R \mathbin{\frown\frown} i)\ x'\ y$
using ‹$(R \mathbin{\frown\frown} \mathit{Suc}\ i)\ x\ y$›[*THEN relpowp-Suc-D2*] **by** *auto*

```
  show (R ^^ (Suc i + j)) x z
    unfolding add-Suc
  proof (rule relpowp-Suc-I2)
    show R x x'
      using ‹R x x'› .
  next
    show (R ^^ (i + j)) x' z
      using Suc.IH[OF ‹(R ^^ i) x' y› ‹(R ^^ j) y z›] .
  qed
qed

lemma relpowp-mono:
  fixes x y :: 'a
  shows (⋀x y. R x y ⟹ S x y) ⟹ (R ^^ n) x y ⟹ (S ^^ n) x y
by (induction n arbitrary: y) auto

lemma relpow-trans[trans]: (x, y) ∈ R ^^ i ⟹ (y, z) ∈ R ^^ j ⟹ (x, z) ∈ R
^^ (i + j)
  using relpowp-trans[to-set] .

lemma relpowp-left-unique:
  fixes R :: 'a ⇒ 'a ⇒ bool and n :: nat and x y z :: 'a
  assumes lunique: ⋀x y z. R x z ⟹ R y z ⟹ x = y
  shows (R ^^ n) x z ⟹ (R ^^ n) y z ⟹ x = y
proof (induction n arbitrary: x y z)
  case 0
  thus ?case
    by simp
next
  case (Suc n')
  then obtain x' y' :: 'a where
    (R ^^ n') x x' and R x' z and
    (R ^^ n') y y' and R y' z
    by auto

  have x' = y'
    using lunique[OF ‹R x' z› ‹R y' z›] .

  show x = y
  proof (rule Suc.IH)
    show (R ^^ n') x x'
      using ‹(R ^^ n') x x'› .
  next
    show (R ^^ n') y x'
      using ‹(R ^^ n') y y'›
      unfolding ‹x' = y'› .
  qed
qed
```

lemma *relpow-left-unique*:
fixes $R :: ('a \times 'a)\ set$ **and** $n :: nat$ **and** $x\ y\ z :: 'a$
shows $(\bigwedge x\ y\ z.\ (x, z) \in R \Longrightarrow (y, z) \in R \Longrightarrow x = y) \Longrightarrow$
$(x, z) \in R \mathbin{\frown\frown} n \Longrightarrow (y, z) \in R \mathbin{\frown\frown} n \Longrightarrow x = y$
using *relpowp-left-unique*[*to-set*] **.**

lemma *relpowp-right-unique*:
fixes $R :: 'a \Rightarrow 'a \Rightarrow bool$ **and** $n :: nat$ **and** $x\ y\ z :: 'a$
assumes *runique*: $\bigwedge x\ y\ z.\ R\ x\ y \Longrightarrow R\ x\ z \Longrightarrow y = z$
shows $(R \mathbin{\frown\frown} n)\ x\ y \Longrightarrow (R \mathbin{\frown\frown} n)\ x\ z \Longrightarrow y = z$
proof (*induction n arbitrary*: *x y z*)
case *0*
thus *?case*
by *simp*
next
case (*Suc n'*)
then obtain $x' :: 'a$ **where**
$(R \mathbin{\frown\frown} n')\ x\ x'$ **and** $R\ x'\ y$ **and** $R\ x'\ z$
by *auto*
thus $y = z$
using *runique* **by** *simp*
qed

lemma *relpow-right-unique*:
fixes $R :: ('a \times 'a)\ set$ **and** $n :: nat$ **and** $x\ y\ z :: 'a$
shows $(\bigwedge x\ y\ z.\ (x, y) \in R \Longrightarrow (x, z) \in R \Longrightarrow y = z) \Longrightarrow$
$(x, y) \in (R \mathbin{\frown\frown} n) \Longrightarrow (x, z) \in (R \mathbin{\frown\frown} n) \Longrightarrow y = z$
using *relpowp-right-unique*[*to-set*] **.**

lemma *relpow-add*: $R \mathbin{\frown\frown} (m + n) = R \mathbin{\frown\frown} m\ O\ R \mathbin{\frown\frown} n$
by (*induct n*) *auto*

lemma *relpowp-add*: $P \mathbin{\frown\frown} (m + n) = P \mathbin{\frown\frown} m\ OO\ P \mathbin{\frown\frown} n$
by (*fact relpow-add* [*to-pred*])

lemma *relpow-commute*: $R\ O\ R \mathbin{\frown\frown} n = R \mathbin{\frown\frown} n\ O\ R$
by (*induct n*) (*simp-all add*: *O-assoc* [*symmetric*])

lemma *relpowp-commute*: $P\ OO\ P \mathbin{\frown\frown} n = P \mathbin{\frown\frown} n\ OO\ P$
by (*fact relpow-commute* [*to-pred*])

lemma *relpowp-Suc-left*: $R \mathbin{\frown\frown} Suc\ n = R\ OO\ (R \mathbin{\frown\frown} n)$
by (*simp add*: *relpowp-commute*)

lemma *relpow-empty*: $0 < n \Longrightarrow (\{\} :: ('a \times 'a)\ set) \mathbin{\frown\frown} n = \{\}$
by (*cases n*) *auto*

lemma *relpowp-bot*: $0 < n \Longrightarrow (\bot :: 'a \Rightarrow 'a \Rightarrow bool) \mathbin{\frown\frown} n = \bot$

by (*fact relpow-empty* [*to-pred*])

lemma *rtrancl-imp-UN-relpow*:
assumes $p \in R^*$
shows $p \in (\bigcup n.\ R \mathbin{^\frown\!^\frown} n)$
proof (*cases p*)
case (*Pair x y*)
with *assms* **have** $(x, y) \in R^*$ **by** *simp*
then have $(x, y) \in (\bigcup n.\ R \mathbin{^\frown\!^\frown} n)$
proof *induct*
case *base*
show *?case* **by** (*blast intro*: *relpow-0-I*)
next
case *step*
then show *?case* **by** (*blast intro*: *relpow-Suc-I*)
qed
with *Pair* **show** *?thesis* **by** *simp*
qed

lemma *rtranclp-imp-Sup-relpowp*:
assumes $(P^{**})\ x\ y$
shows $(\bigsqcup n.\ P \mathbin{^\frown\!^\frown} n)\ x\ y$
using *assms* **and** *rtrancl-imp-UN-relpow* [*of* (x, y), *to-pred*] **by** *simp*

lemma *relpow-imp-rtrancl*:
assumes $p \in R \mathbin{^\frown\!^\frown} n$
shows $p \in R^*$
proof (*cases p*)
case (*Pair x y*)
with *assms* **have** $(x, y) \in R \mathbin{^\frown\!^\frown} n$ **by** *simp*
then have $(x, y) \in R^*$
proof (*induct n arbitrary*: *x y*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case*
by (*blast elim*: *relpow-Suc-E intro*: *rtrancl-into-rtrancl*)
qed
with *Pair* **show** *?thesis* **by** *simp*
qed

lemma *relpowp-imp-rtranclp*: $(P \mathbin{^\frown\!^\frown} n)\ x\ y \Longrightarrow (P^{**})\ x\ y$
using *relpow-imp-rtrancl* [*of* (x, y), *to-pred*] **by** *simp*

lemma *rtrancl-is-UN-relpow*: $R^* = (\bigcup n.\ R \mathbin{^\frown\!^\frown} n)$
by (*blast intro*: *rtrancl-imp-UN-relpow relpow-imp-rtrancl*)

lemma *rtranclp-is-Sup-relpowp*: $P^{**} = (\bigsqcup n.\ P \mathbin{^\frown\!^\frown} n)$

using *rtrancl-is-UN-relpow* [*to-pred*, *of P*] **by** *auto*

lemma *rtrancl-power*: $p \in R^* \longleftrightarrow (\exists\, n.\ p \in R\ ^{\frown\frown}\ n)$
by (*simp add*: *rtrancl-is-UN-relpow*)

lemma *rtranclp-power*: $(P^{**})\ x\ y \longleftrightarrow (\exists\, n.\ (P\ ^{\frown\frown}\ n)\ x\ y)$
by (*simp add*: *rtranclp-is-Sup-relpowp*)

lemma *trancl-power*: $p \in R^+ \longleftrightarrow (\exists\, n > 0.\ p \in R\ ^{\frown\frown}\ n)$
proof −
have $(a,\ b) \in R^+ \longleftrightarrow (\exists\, n{>}0.\ (a,\ b) \in R\ ^{\frown\frown}\ n)$ **for** *a b*
proof *safe*
show $(a,\ b) \in R^+ \Longrightarrow \exists\, n{>}0.\ (a,\ b) \in R\ ^{\frown\frown}\ n$
by (*fastforce simp*: *rtrancl-is-UN-relpow relcomp-unfold dest*: *tranclD2*)
show $(a,\ b) \in R^+$ **if** $n > 0$ $(a,\ b) \in R\ ^{\frown\frown}\ n$ **for** *n*
proof (*cases n*)
case (*Suc m*)
with *that* **show** *?thesis*
by (*auto simp*: *dest*: *relpow-imp-rtrancl rtrancl-into-trancl1*)
qed (*use that* **in** *auto*)
qed
then show *?thesis*
by (*cases p*) *auto*
qed

lemma *tranclp-power*: $(P^{++})\ x\ y \longleftrightarrow (\exists\, n > 0.\ (P\ ^{\frown\frown}\ n)\ x\ y)$
using *trancl-power* [*to-pred*, *of P* $(x,\ y)$] **by** *simp*

lemma *rtrancl-imp-relpow*: $p \in R^* \Longrightarrow \exists\, n.\ p \in R\ ^{\frown\frown}\ n$
by (*auto dest*: *rtrancl-imp-UN-relpow*)

lemma *rtranclp-imp-relpowp*: $(P^{**})\ x\ y \Longrightarrow \exists\, n.\ (P\ ^{\frown\frown}\ n)\ x\ y$
by (*auto dest*: *rtranclp-imp-Sup-relpowp*)

By Sternagel/Thiemann:

lemma *relpow-fun-conv*: $(a,\ b) \in R\ ^{\frown\frown}\ n \longleftrightarrow (\exists f.\ f\ 0 = a \wedge f\ n = b \wedge (\forall\, i{<}n.\ (f\ i,\ f\ (Suc\ i)) \in R))$
proof (*induct n arbitrary*: *b*)
case *0*
show *?case* **by** *auto*
next
case (*Suc n*)
show *?case*
proof −
have $(\exists\, y.\ (\exists f.\ f\ 0 = a \wedge f\ n = y \wedge (\forall\, i{<}n.\ (f\ i, f(Suc\ i)) \in R)) \wedge (y,b) \in R)$
$\longleftrightarrow$
$(\exists f.\ f\ 0 = a \wedge f(Suc\ n) = b \wedge (\forall\, i{<}Suc\ n.\ (f\ i,\ f\ (Suc\ i)) \in R))$
(**is** *?l* $\longleftrightarrow$ *?r*)
proof

```
    assume ?l
    then obtain c f
      where 1: f 0 = a  f n = c  ⋀i. i < n ⟹ (f i, f (Suc i)) ∈ R  (c,b) ∈ R
      by auto
    let ?g = λ m. if m = Suc n then b else f m
    show ?r by (rule exI[of - ?g]) (simp add: 1)
  next
    assume ?r
    then obtain f where 1: f 0 = a  b = f (Suc n)  ⋀i. i < Suc n ⟹ (f i, f
(Suc i)) ∈ R
      by auto
    show ?l by (rule exI[of - f n], rule conjI, rule exI[of - f], auto simp add: 1)
  qed
  then show ?thesis by (simp add: relcomp-unfold Suc)
 qed
qed

lemma relpowp-fun-conv: (P ^^ n) x y ⟷ (∃f. f 0 = x ∧ f n = y ∧ (∀ i<n. P
(f i) (f (Suc i))))
 by (fact relpow-fun-conv [to-pred])

lemma relpow-finite-bounded1:
 fixes R :: ('a × 'a) set
 assumes finite R and k > 0
 shows R^^k ⊆ (⋃ n∈{n. 0 < n ∧ n ≤ card R}. R^^n)
   (is - ⊆ ?r)
proof −
 have (a, b) ∈ R^^(Suc k) ⟹ ∃ n. 0 < n ∧ n ≤ card R ∧ (a, b) ∈ R^^n for a
b k
 proof (induct k arbitrary: b)
   case 0
   then have R ≠ {} by auto
   with card-0-eq[OF ‹finite R›] have card R ≥ Suc 0 by auto
   then show ?case using 0 by force
 next
   case (Suc k)
   then obtain a' where (a, a') ∈ R^^(Suc k) and (a', b) ∈ R
     by auto
   from Suc(1)[OF ‹(a, a') ∈ R^^(Suc k)›] obtain n where n ≤ card R and (a,
a') ∈ R ^^ n
     by auto
   have (a, b) ∈ R^^(Suc n)
     using ‹(a, a') ∈ R^^n› and ‹(a', b)∈ R› by auto
   from ‹n ≤ card R› consider n < card R | n = card R by force
   then show ?case
   proof cases
     case 1
     then show ?thesis
       using ‹(a, b) ∈ R^^(Suc n)› Suc-leI[OF ‹n < card R›] by blast
```

```
  next
    case 2
    from ‹(a, b) ∈ R ^^ (Suc n)› [unfolded relpow-fun-conv]
    obtain f where f 0 = a and f (Suc n) = b
      and steps: ⋀i. i ≤ n ⟹ (f i, f (Suc i)) ∈ R by auto
    let ?p = λi. (f i, f(Suc i))
    let ?N = {i. i ≤ n}
    have ?p ‘ ?N ⊆ R
      using steps by auto
    from card-mono[OF assms(1) this] have card (?p ‘ ?N) ≤ card R .
    also have ... < card ?N
      using ‹n = card R› by simp
    finally have ¬ inj-on ?p ?N
      by (rule pigeonhole)
    then obtain i j where i: i ≤ n and j: j ≤ n and ij: i ≠ j and pij: ?p i =
?p j
      by (auto simp: inj-on-def)
    let ?i = min i j
    let ?j = max i j
    have i: ?i ≤ n and j: ?j ≤ n and pij: ?p ?i = ?p ?j and ij: ?i < ?j
      using i j ij pij unfolding min-def max-def by auto
    from i j pij ij obtain i j where i: i ≤ n and j: j ≤ n and ij: i < j
      and pij: ?p i = ?p j
      by blast
    let ?g = λl. if l ≤ i then f l else f (l + (j − i))
    let ?n = Suc (n − (j − i))
    have abl: (a, b) ∈ R ^^ ?n
      unfolding relpow-fun-conv
    proof (rule exI[of - ?g], intro conjI impI allI)
      show ?g ?n = b
        using ‹f(Suc n) = b› j ij by auto
    next
      fix k
      assume k < ?n
      show (?g k, ?g (Suc k)) ∈ R
      proof (cases k < i)
        case True
        with i have k ≤ n
          by auto
        from steps[OF this] show ?thesis
          using True by simp
      next
        case False
        then have i ≤ k by auto
        show ?thesis
        proof (cases k = i)
          case True
          then show ?thesis
            using ij pij steps[OF i] by simp
```

```
        next
          case False
          with ‹i ≤ k› have i < k by auto
          then have small: k + (j − i) ≤ n
            using ‹k<?n› by arith
          show ?thesis
            using steps[OF small] ‹i<k› by auto
        qed
      qed
    qed (simp add: ‹f 0 = a›)
    moreover have ?n ≤ n
      using i j ij by arith
    ultimately show ?thesis
      using ‹n = card R› by blast
  qed
 qed
 then show ?thesis
  using gr0-implies-Suc[OF ‹k > 0›] by auto
qed

lemma relpow-finite-bounded:
 fixes R :: ('a × 'a) set
 assumes finite R
 shows R^^k ⊆ (⋃n∈{n. n ≤ card R}. R^^n)
proof (cases k)
 case (Suc k')
 then show ?thesis
  using relpow-finite-bounded1[OF assms, of k] by auto
qed force

lemma rtrancl-finite-eq-relpow: finite R ⟹ R* = (⋃n∈{n. n ≤ card R}. R^^n)
 by (fastforce simp: rtrancl-power dest: relpow-finite-bounded)

lemma trancl-finite-eq-relpow:
 assumes finite R shows R⁺ = (⋃n∈{n. 0 < n ∧ n ≤ card R}. R^^n)
proof −
 have ⋀a b n. ⟦0 < n; (a, b) ∈ R ^^ n⟧ ⟹ ∃x>0. x ≤ card R ∧ (a, b) ∈ R ^^
x
  using assms by (auto dest: relpow-finite-bounded1)
 then show ?thesis
  by (auto simp: trancl-power)
qed

lemma finite-relcomp[simp,intro]:
 assumes finite R and finite S
 shows finite (R O S)
proof−
 have R O S = (⋃(x, y)∈R. ⋃(u, v)∈S. if u = y then {(x, v)} else {})
  by (force simp: split-def image-constant-conv split: if-splits)
```

then show *?thesis*
using *assms* **by** *clarsimp*
qed

lemma *finite-relpow* [*simp*, *intro*]:
fixes $R :: ('a \times 'a)$ *set*
assumes *finite R*
shows $n > 0 \Longrightarrow$ *finite* $(R \frown\frown n)$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **by** (*cases n*) (*use assms* **in** *simp-all*)
qed

lemma *single-valued-relpow*:
fixes $R :: ('a \times 'a)$ *set*
shows *single-valued R* $\Longrightarrow$ *single-valued* $(R \frown\frown n)$
proof (*induct n arbitrary*: *R*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
show *?case*
by (*rule single-valuedI*)
(*use Suc* **in** ‹*fast dest*: *single-valuedD elim*: *relpow-Suc-E*›)
qed

21.6 Bounded transitive closure

definition *ntrancl* :: *nat* $\Rightarrow$ $('a \times 'a)$ *set* $\Rightarrow$ $('a \times 'a)$ *set*
where *ntrancl n R* $= (\bigcup i \in \{i.\ 0 < i \wedge i \leq Suc\ n\}.\ R \frown\frown i)$

lemma *ntrancl-Zero* [*simp*, *code*]: *ntrancl 0 R* $= R$
proof
show $R \subseteq$ *ntrancl 0 R*
unfolding *ntrancl-def* **by** *fastforce*
have $0 < i \wedge i \leq Suc\ 0 \longleftrightarrow i = 1$ **for** *i*
by *auto*
then show *ntrancl 0 R* $\leq R$
unfolding *ntrancl-def* **by** *auto*
qed

lemma *ntrancl-Suc* [*simp*]: *ntrancl* (*Suc n*) *R* $=$ *ntrancl n R O* $(Id \cup R)$
proof
have $(a, b) \in$ *ntrancl n R O* $(Id \cup R)$ **if** $(a, b) \in$ *ntrancl* (*Suc n*) *R* **for** *a b*
proof −
from *that* **obtain** *i* **where** $0 < i\ i \leq Suc\ (Suc\ n)\ (a, b) \in R \frown\frown i$

```
    unfolding ntrancl-def by auto
  show ?thesis
  proof (cases i = 1)
    case True
    with ‹(a, b) ∈ R ^^ i› show ?thesis
      by (auto simp: ntrancl-def)
  next
    case False
    with ‹0 < i› obtain j where j: i = Suc j 0 < j
      by (cases i) auto
    with ‹(a, b) ∈ R ^^ i› obtain c where c1: (a, c) ∈ R ^^ j and c2: (c, b)
∈ R
      by auto
    from c1 j ‹i ≤ Suc (Suc n)› have (a, c) ∈ ntrancl n R
      by (fastforce simp: ntrancl-def)
    with c2 show ?thesis by fastforce
  qed
 qed
 then show ntrancl (Suc n) R ⊆ ntrancl n R O (Id ∪ R)
  by auto
 show ntrancl n R O (Id ∪ R) ⊆ ntrancl (Suc n) R
  by (fastforce simp: ntrancl-def)
qed

lemma [code]: ntrancl (Suc n) r = (let r' = ntrancl n r in r' ∪ r' O r)
 by (auto simp: Let-def)

lemma finite-trancl-ntranl: finite R ⟹ trancl R = ntrancl (card R - 1) R
 by (cases card R) (auto simp: trancl-finite-eq-relpow relpow-empty ntrancl-def)
```

The two lemmas below make the reflexive transitive executable in certain contexts.

```
lemma rtrancl-Image-code[code-unfold]: R^* `` A = R^+ `` A ∪ A
unfolding rtrancl-trancl-reflcl Un-Image Image-Id ..

lemma rtrancl-converse-code[code-unfold]: (R^*)^-1 `` A = (R^+)^-1 `` A ∪ A
by(simp flip:  rtrancl-converse trancl-converse add: rtrancl-Image-code)
```

21.7 Acyclic relations

```
definition acyclic :: ('a × 'a) set ⇒ bool
 where acyclic r ⟷ (∀ x. (x,x) ∉ r^+)

abbreviation acyclicP :: ('a ⇒ 'a ⇒ bool) ⇒ bool
 where acyclicP r ≡ acyclic {(x, y). r x y}

lemma acyclic-irrefl [code]: acyclic r ⟷ irrefl (r^+)
 by (simp add: acyclic-def irrefl-def)
```

lemma *acyclicI*: $\forall x.\ (x,\ x) \notin r^+ \Longrightarrow$ *acyclic r*
by (*simp add*: *acyclic-def*)

lemma (**in** *preorder*) *acyclicI-order*:
assumes $*$: $\bigwedge a\ b.\ (a,\ b) \in r \Longrightarrow f\ b < f\ a$
shows *acyclic r*
proof −
have $f\ b < f\ a$ **if** $(a,\ b) \in r^+$ **for** *a b*
using *that* **by** *induct* (*auto intro*: $*$ *less-trans*)
then show *?thesis*
by (*auto intro*!: *acyclicI*)
qed

lemma *acyclic-insert* [*iff*]: *acyclic* (*insert* $(y,\ x)$ *r*) $\longleftrightarrow$ *acyclic r* $\wedge$ $(x,\ y) \notin r^*$
by (*simp add*: *acyclic-def trancl-insert*) (*blast intro*: *rtrancl-trans*)

lemma *acyclic-converse* [*iff*]: *acyclic* (r^{-1}) $\longleftrightarrow$ *acyclic r*
by (*simp add*: *acyclic-def trancl-converse*)

lemmas *acyclicP-converse* [*iff*] = *acyclic-converse* [*to-pred*]

lemma *acyclic-impl-antisym-rtrancl*: *acyclic r* $\Longrightarrow$ *antisym* (r^*)
by (*simp add*: *acyclic-def antisym-def*)
(*blast elim*: *rtranclE intro*: *rtrancl-into-trancl1 rtrancl-trancl-trancl*)

lemma *acyclic-subset*: *acyclic s* $\Longrightarrow$ $r \subseteq s \Longrightarrow$ *acyclic r*
unfolding *acyclic-def* **by** (*blast intro*: *trancl-mono*)

21.8 Setup of transitivity reasoner

ML ‹
structure Trancl-Tac = Trancl-Tac
(
val r-into-trancl = @{thm trancl.r-into-trancl};
val trancl-trans = @{thm trancl-trans};
val rtrancl-refl = @{thm rtrancl.rtrancl-refl};
val r-into-rtrancl = @{thm r-into-rtrancl};
val trancl-into-rtrancl = @{thm trancl-into-rtrancl};
val rtrancl-trancl-trancl = @{thm rtrancl-trancl-trancl};
val trancl-rtrancl-trancl = @{thm trancl-rtrancl-trancl};
val rtrancl-trans = @{thm rtrancl-trans};

fun decomp ***Const-*** ‹*Trueprop for t*› =
let
fun dec ***Const-*** ‹*Set.member - for* ***Const-*** ‹*Pair - - for a b*› *rel*› =
let
fun decr ***Const-*** ‹*rtrancl - for r*› = (*r*,*r*$*$)

```
              | decr Const- ‹trancl - for r› = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr (Envir.beta-eta-contract rel);
          in SOME (a,b,rel,r) end
      | dec - =  NONE
    in dec t end
  | decomp - = NONE;
);

structure Tranclp-Tac = Trancl-Tac
(
  val r-into-trancl = @{thm tranclp.r-into-trancl};
  val trancl-trans  = @{thm tranclp-trans};
  val rtrancl-refl = @{thm rtranclp.rtrancl-refl};
  val r-into-rtrancl = @{thm r-into-rtranclp};
  val trancl-into-rtrancl = @{thm tranclp-into-rtranclp};
  val rtrancl-trancl-trancl = @{thm rtranclp-tranclp-tranclp};
  val trancl-rtrancl-trancl = @{thm tranclp-rtranclp-tranclp};
  val rtrancl-trans = @{thm rtranclp-trans};

  fun decomp Const- ‹Trueprop for t› =
      let
        fun dec (rel $ a $ b) =
          let
            fun decr Const- ‹rtranclp - for r› = (r,r*)
              | decr Const- ‹tranclp - for r› = (r,r+)
              | decr r = (r,r);
            val (rel,r) = decr rel;
          in SOME (a, b, rel, r) end
        | dec - =  NONE
      in dec t end
  | decomp - = NONE;
);
›

setup ‹
  Simplifier.map-theory-simpset
   (Simplifier.add-unsafe-solver (Simplifier.mk-solver Trancl Trancl-Tac.trancl-tac)
   #> Simplifier.add-unsafe-solver (Simplifier.mk-solver Rtrancl Trancl-Tac.rtrancl-tac)
   #> Simplifier.add-unsafe-solver (Simplifier.mk-solver Tranclp Tranclp-Tac.trancl-tac)
   #> Simplifier.add-unsafe-solver (Simplifier.mk-solver Rtranclp Tranclp-Tac.rtrancl-tac))
›

lemma transp-rtranclp [simp]: transp R**
  by(auto simp add: transp-def)
```

Optional methods.

```
method-setup trancl =
  ‹Scan.succeed (SIMPLE-METHOD' o Trancl-Tac.trancl-tac)›
```

‹*simple transitivity reasoner*›
method-setup *rtrancl* =
‹*Scan.succeed* (*SIMPLE-METHOD'* o *Trancl-Tac.rtrancl-tac*)›
‹*simple transitivity reasoner*›
method-setup *tranclp* =
‹*Scan.succeed* (*SIMPLE-METHOD'* o *Tranclp-Tac.trancl-tac*)›
‹*simple transitivity reasoner* (*predicate version*)›
method-setup *rtranclp* =
‹*Scan.succeed* (*SIMPLE-METHOD'* o *Tranclp-Tac.rtrancl-tac*)›
‹*simple transitivity reasoner* (*predicate version*)›

end

22 Well-founded Recursion

theory *Wellfounded*
imports *Transitive-Closure*
begin

22.1 Basic Definitions

definition *wf-on* :: *'a set* ⇒ *'a rel* ⇒ *bool* **where**
wf-on A r ⟷ (∀ *P*. (∀ *x* ∈ *A*. (∀ *y* ∈ *A*. (*y*, *x*) ∈ *r* ⟶ *P y*) ⟶ *P x*) ⟶ (∀ *x* ∈ *A*. *P x*))

abbreviation *wf* :: (*'a* × *'a*) *set* ⇒ *bool* **where**
wf ≡ *wf-on UNIV*

definition *wfp-on* :: *'a set* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
wfp-on A R ⟷ (∀ *P*. (∀ *x* ∈ *A*. (∀ *y* ∈ *A*. *R y x* ⟶ *P y*) ⟶ *P x*) ⟶ (∀ *x* ∈ *A*. *P x*))

abbreviation *wfP* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
wfP ≡ *wfp-on UNIV*

alias *wfp* = *wfP*

We keep old name *wfp* for backward compatibility, but offer new name *wfp* to be consistent with similar predicates, e.g., *asymp*, *transp*, *totalp*.

22.2 Equivalence of Definitions

lemma *wfp-on-wf-on-eq*[*pred-set-conv*]: *wfp-on A* (λ*x y*. (*x*, *y*) ∈ *r*) ⟷ *wf-on A r*
by (*simp add*: *wfp-on-def wf-on-def*)

lemma *wf-def*: *wf r* ⟷ (∀ *P*. (∀ *x*. (∀ *y*. (*y*, *x*) ∈ *r* ⟶ *P y*) ⟶ *P x*) ⟶ (∀ *x*. *P x*))
unfolding *wf-on-def* **by** *simp*

lemma *wfp-def*: *wfp r* ⟷ *wf* {(*x*, *y*). *r x y*}
unfolding *wf-def wfp-on-def* **by** *simp*

lemma *wfp-wf-eq*: *wfp* (λ*x y*. (*x*, *y*) ∈ *r*) = *wf r*
using *wfp-on-wf-on-eq* **.**

22.3 Induction Principles

lemma *wf-on-induct*[*consumes 1*, *case-names in-set less*, *induct set*: *wf-on*]:
assumes *wf-on A r* **and** *x* ∈ *A* **and** ⋀*x*. *x* ∈ *A* ⟹ (⋀*y*. *y* ∈ *A* ⟹ (*y*, *x*) ∈ *r* ⟹ *P y*) ⟹ *P x*
shows *P x*
using *assms*(*2*,*3*) **by** (*auto intro*: ‹*wf-on A r*›[*unfolded wf-on-def*, *rule-format*])

lemma *wfp-on-induct*[*consumes 1*, *case-names in-set less*, *induct pred*: *wfp-on*]:
assumes *wfp-on A r* **and** *x* ∈ *A* **and** ⋀*x*. *x* ∈ *A* ⟹ (⋀*y*. *y* ∈ *A* ⟹ *r y x* ⟹ *P y*) ⟹ *P x*
shows *P x*
using *assms* **by** (*fact wf-on-induct*[*to-pred*])

lemma *wf-induct*:
assumes *wf r*
and ⋀*x*. ∀ *y*. (*y*, *x*) ∈ *r* ⟶ *P y* ⟹ *P x*
shows *P a*
using *assms* **by** (*auto intro*: *wf-on-induct*[*of UNIV*])

lemmas *wfp-induct* = *wf-induct* [*to-pred*]

lemmas *wf-induct-rule* = *wf-induct* [*rule-format*, *consumes 1*, *case-names less*, *induct set*: *wf*]

lemmas *wfp-induct-rule* = *wf-induct-rule* [*to-pred*, *induct set*: *wfp*]

lemma *wf-on-iff-wf*: *wf-on A r* ⟷ *wf* {(*x*, *y*) ∈ *r*. *x* ∈ *A* ∧ *y* ∈ *A*}
proof (*rule iffI*)
assume *wf*: *wf-on A r*
show *wf* {(*x*, *y*) ∈ *r*. *x* ∈ *A* ∧ *y* ∈ *A*}
unfolding *wf-def*
proof (*intro allI impI ballI*)
fix *P x*
assume *IH*: ∀ *x*. (∀ *y*. (*y*, *x*) ∈ {(*x*, *y*). (*x*, *y*) ∈ *r* ∧ *x* ∈ *A* ∧ *y* ∈ *A*} ⟶ *P y*) ⟶ *P x*
show *P x*
proof (*cases x* ∈ *A*)
case *True*
show *?thesis*
using *wf*
proof (*induction x rule*: *wf-on-induct*)

```
      case in-set
      thus ?case
        using True .
    next
      case (less x)
      thus ?case
        by (auto intro: IH[rule-format])
    qed
  next
    case False
    then show ?thesis
      by (auto intro: IH[rule-format])
  qed
qed
next
  assume wf: wf {(x, y). (x, y) ∈ r ∧ x ∈ A ∧ y ∈ A}
  show wf-on A r
    unfolding wf-on-def
  proof (intro allI impI ballI)
    fix P x
    assume IH: ∀ x∈A. (∀ y∈A. (y, x) ∈ r ⟶ P y) ⟶ P x and x ∈ A
    show P x
      using wf ‹x ∈ A›
    proof (induction x rule: wf-on-induct)
      case in-set
      show ?case
        by simp
    next
      case (less y)
      hence ⋀z. (z, y) ∈ r ⟹ z ∈ A ⟹ P z
        by simp
      thus ?case
        using IH[rule-format, OF ‹y ∈ A›] by simp
    qed
  qed
qed
```

22.4 Introduction Rules

lemma *wfUNIVI*: (⋀P x. (∀ x. (∀ y. (y, x) ∈ r ⟶ P y) ⟶ P x) ⟹ P x) ⟹ *wf r*
  unfolding *wf-def* **by** *blast*

lemmas *wfpUNIVI* = *wfUNIVI* [*to-pred*]

Restriction to domain A and range B. If r is well-founded over their intersection, then *wf r*.

lemma *wfI*:
  assumes $r \subseteq A \times B$

and $\bigwedge x\ P.\ [\![\forall x.\ (\forall y.\ (y,\ x) \in r \longrightarrow P\ y) \longrightarrow P\ x;\ \ x \in A;\ x \in B]\!] \Longrightarrow P\ x$
shows *wf r*
using *assms* **unfolding** *wf-def* **by** *blast*

22.5 Ordering Properties

lemma *wf-not-sym*: *wf r* $\Longrightarrow$ $(a,\ x) \in r \Longrightarrow (x,\ a) \notin r$
by (*induct a arbitrary*: *x set*: *wf*) *blast*

lemma *wf-asym*:
assumes *wf r* $(a,\ x) \in r$
obtains $(x,\ a) \notin r$
by (*drule wf-not-sym*[*OF assms*])

lemma *wf-imp-asym*: *wf r* $\Longrightarrow$ *asym r*
by (*auto intro*: *asymI elim*: *wf-asym*)

lemma *wfp-imp-asymp*: *wfp r* $\Longrightarrow$ *asymp r*
by (*rule wf-imp-asym*[*to-pred*])

lemma *wf-not-refl* [*simp*]: *wf r* $\Longrightarrow$ $(a,\ a) \notin r$
by (*blast elim*: *wf-asym*)

lemma *wf-irefl*:
assumes *wf r*
obtains $(a,\ a) \notin r$
by (*drule wf-not-refl*[*OF assms*])

lemma *wf-imp-irefl*:
assumes *wf r* **shows** *irefl r*
using *wf-irefl* [*OF assms*] **by** (*auto simp add*: *irefl-def*)

lemma *wfp-imp-irreflp*: *wfp r* $\Longrightarrow$ *irreflp r*
by (*rule wf-imp-irefl*[*to-pred*])

lemma *wf-wellorderI*:
assumes *wf*: *wf* $\{(x::'a::ord,\ y).\ x < y\}$
and *lin*: *OFCLASS*(*'a*::*ord*, *linorder-class*)
shows *OFCLASS*(*'a*::*ord*, *wellorder-class*)
apply (*rule wellorder-class.intro* [*OF lin*])
apply (*simp add*: *wellorder-class.intro class.wellorder-axioms.intro wf-induct-rule* [*OF wf*])
done

lemma (**in** *wellorder*) *wf*: *wf* $\{(x,\ y).\ x < y\}$
unfolding *wf-def* **by** (*blast intro*: *less-induct*)

lemma (**in** *wellorder*) *wfp-on-less*[*simp*]: *wfp-on A* $(<)$
unfolding *wfp-on-def*

proof (*intro allI impI ballI*)
fix P x
assume *hyps*: $\forall x{\in}A.\ (\forall y{\in}A.\ y < x \longrightarrow P\ y) \longrightarrow P\ x$
show $x \in A \Longrightarrow P\ x$
proof (*induction x rule*: *less-induct*)
case (*less x*)
show *?case*
proof (*rule hyps*[*rule-format*])
show $x \in A$
using ‹$x \in A$› **.**
next
show $\bigwedge y.\ y \in A \Longrightarrow y < x \Longrightarrow P\ y$
using *less.IH* **.**
qed
qed
qed

22.6 Basic Results

Point-free characterization of well-foundedness

lemma *wf-onE-pf*:
assumes *wf*: *wf-on A r* **and** $B \subseteq A$ **and** $B \subseteq r\ ``\ B$
shows $B = \{\}$
proof −
have $x \notin B$ **if** $x \in A$ **for** x
using *wf*
proof (*induction x rule*: *wf-on-induct*)
case *in-set*
show *?case*
using *that* **.**
next
case (*less x*)
have $x \notin r\ ``\ B$
using *less.IH* ‹$B \subseteq A$› **by** *blast*
thus *?case*
using ‹$B \subseteq r\ ``\ B$› **by** *blast*
qed
with ‹$B \subseteq A$› **show** *?thesis*
by *blast*
qed

lemma *wfE-pf*: *wf R* $\Longrightarrow$ $A \subseteq R\ ``\ A \Longrightarrow A = \{\}$
using *wf-onE-pf*[*of UNIV*, *simplified*] **.**

lemma *wf-onI-pf*:
assumes $\bigwedge B.\ B \subseteq A \Longrightarrow B \subseteq R\ ``\ B \Longrightarrow B = \{\}$
shows *wf-on A R*
unfolding *wf-on-def*
proof (*intro allI impI ballI*)

fix $P :: 'a \Rightarrow bool$ **and** $x :: 'a$
let *?B* $= \{x \in A.\ \neg\ P\ x\}$
assume $\forall\, x{\in}A.\ (\forall\, y{\in}A.\ (y,\ x) \in R \longrightarrow P\ y) \longrightarrow P\ x$
hence *?B* $\subseteq R$ `` *?B* **by** *blast*
hence $\{x \in A.\ \neg\ P\ x\} = \{\}$
using *assms(1)*[*of ?B*] **by** *simp*
moreover assume $x \in A$
ultimately show $P\ x$
by *simp*
qed

lemma *wfI-pf*: $(\bigwedge A.\ A \subseteq R$ `` $A \Longrightarrow A = \{\}) \Longrightarrow wf\ R$
using *wf-onI-pf*[*of UNIV*, *simplified*] **.**

22.6.1 Minimal-element characterization of well-foundedness

lemma *wf-on-iff-ex-minimal*: *wf-on A R* $\longleftrightarrow$ $(\forall\, B \subseteq A.\ B \neq \{\} \longrightarrow (\exists\, z \in B.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin B))$
proof (*intro iffI allI impI*)
fix *B*
assume *wf-on A R* **and** $B \subseteq A$ **and** $B \neq \{\}$
show $\exists\, z \in B.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin B$
using *wf-onE-pf*[*OF* ‹*wf-on A R*› ‹$B \subseteq A$›] ‹$B \neq \{\}$› **by** *blast*
next
assume *ex-min*: $\forall\, B{\subseteq}A.\ B \neq \{\} \longrightarrow (\exists\, z{\in}B.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin B)$
show *wf-on A R*
proof (*rule wf-onI-pf*)
fix *B*
assume $B \subseteq A$ **and** $B \subseteq R$ `` *B*
have *False* **if** $B \neq \{\}$
using *ex-min*[*rule-format*, *OF* ‹$B \subseteq A$› ‹$B \neq \{\}$›]
using ‹$B \subseteq R$ `` *B*› **by** *blast*
thus $B = \{\}$
by *blast*
qed
qed

lemma *wf-iff-ex-minimal*: *wf R* $\longleftrightarrow$ $(\forall\, B.\ B \neq \{\} \longrightarrow (\exists\, z \in B.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin B))$
using *wf-on-iff-ex-minimal*[*of UNIV*, *simplified*] **.**

lemma *wfp-on-iff-ex-minimal*: *wfp-on A R* $\longleftrightarrow$ $(\forall\, B \subseteq A.\ B \neq \{\} \longrightarrow (\exists\, z \in B.\ \forall\, y.\ R\ y\ z \longrightarrow y \notin B))$
using *wf-on-iff-ex-minimal*[*of A*, *to-pred*] **by** *simp*

lemma *wfp-iff-ex-minimal*: *wfp R* $\longleftrightarrow$ $(\forall\, B.\ B \neq \{\} \longrightarrow (\exists\, z \in B.\ \forall\, y.\ R\ y\ z \longrightarrow y \notin B))$
using *wfp-on-iff-ex-minimal*[*of UNIV*, *simplified*] **.**

lemma *wfE-min*:
assumes *wf*: *wf R* **and** *Q*: $x \in Q$
obtains *z* **where** $z \in Q$ $\bigwedge y.\ (y,\ z) \in R \Longrightarrow y \notin Q$
using *Q wfE-pf*[*OF wf*, *of Q*] **by** *blast*

lemma *wfE-min'*:
wf R $\Longrightarrow$ $Q \neq \{\}$ $\Longrightarrow$ $(\bigwedge z.\ z \in Q \Longrightarrow (\bigwedge y.\ (y,\ z) \in R \Longrightarrow y \notin Q) \Longrightarrow thesis)$
$\Longrightarrow$ *thesis*
using *wfE-min*[*of R - Q*] **by** *blast*

lemma *wfI-min*:
assumes *a*: $\bigwedge x\ Q.\ x \in Q \Longrightarrow \exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in R \longrightarrow y \notin Q$
shows *wf R*
proof (*rule wfI-pf*)
fix *A*
assume *b*: $A \subseteq R$ `` *A*
have *False* **if** $x \in A$ **for** *x*
using *a*[*OF that*] *b* **by** *blast*
then show $A = \{\}$ **by** *blast*
qed

lemma *wf-eq-minimal*: *wf r* $\longleftrightarrow$ $(\forall\, Q\ x.\ x \in Q \longrightarrow (\exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in r \longrightarrow y \notin Q))$
unfolding *wf-iff-ex-minimal* **by** *blast*

lemmas *wfp-eq-minimal* = *wf-eq-minimal* [*to-pred*]

22.6.2 Finite characterization of well-foundedness

lemma *strict-partial-order-wfp-on-finite-set*:
assumes *transp-on* $\mathcal{X}$ *R* **and** *asymp-on* $\mathcal{X}$ *R* **and** *finite* $\mathcal{X}$
shows *wfp-on* $\mathcal{X}$ *R*
unfolding *Wellfounded.wfp-on-iff-ex-minimal*
proof (*intro allI impI*)
fix $\mathcal{W}$
assume $\mathcal{W} \subseteq \mathcal{X}$ **and** $\mathcal{W} \neq \{\}$

have *finite* $\mathcal{W}$
using *finite-subset*[*OF* ‹$\mathcal{W} \subseteq \mathcal{X}$› ‹*finite* $\mathcal{X}$›] **.**

moreover have *asymp-on* $\mathcal{W}$ *R*
using *asymp-on-subset*[*OF* ‹*asymp-on* $\mathcal{X}$ *R*› ‹$\mathcal{W} \subseteq \mathcal{X}$›] **.**

moreover have *transp-on* $\mathcal{W}$ *R*
using *transp-on-subset*[*OF* ‹*transp-on* $\mathcal{X}$ *R*› ‹$\mathcal{W} \subseteq \mathcal{X}$›] **.**

ultimately have $\exists\, m{\in}\mathcal{W}.\ \forall\, x{\in}\mathcal{W}.\ x \neq m \longrightarrow \neg\ R\ x\ m$
using ‹$\mathcal{W} \neq \{\}$› *Finite-Set.bex-min-element*[*of* $\mathcal{W}$ *R*] **by** *iprover*

thus $\exists z{\in}\mathcal{W}.\ \forall y.\ R\ y\ z \longrightarrow y \notin \mathcal{W}$
using *asymp-onD*[*OF* ‹*asymp-on* $\mathcal{W}$ R›] **by** *fast*
qed

22.6.3 Antimonotonicity

lemma *wfp-on-mono-stronger*:
fixes
$A :: 'a\ set$ **and** $B :: 'b\ set$ **and**
$f :: 'a \Rightarrow 'b$ **and**
$R :: 'b \Rightarrow 'b \Rightarrow bool$ **and** $Q :: 'a \Rightarrow 'a \Rightarrow bool$
assumes
wf: *wfp-on* B R **and**
sub: $f\ `\ A \subseteq B$ **and**
mono: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow Q\ x\ y \Longrightarrow R\ (f\ x)\ (f\ y)$
shows *wfp-on* A Q
unfolding *wfp-on-iff-ex-minimal*
proof (*intro allI impI*)
fix $A' :: 'a\ set$
assume $A' \subseteq A$ **and** $A' \neq \{\}$
have $f\ `\ A' \subseteq B$
using ‹$A' \subseteq A$› *sub* **by** *blast*
moreover have $f\ `\ A' \neq \{\}$
using ‹$A' \neq \{\}$› **by** *blast*
ultimately have $\exists z{\in}f\ `\ A'.\ \forall y.\ R\ y\ z \longrightarrow y \notin f\ `\ A'$
using *wf wfp-on-iff-ex-minimal* **by** *blast*
hence $\exists z{\in}A'.\ \forall y.\ R\ (f\ y)\ (f\ z) \longrightarrow y \notin A'$
by *blast*
thus $\exists z{\in}A'.\ \forall y.\ Q\ y\ z \longrightarrow y \notin A'$
using ‹$A' \subseteq A$› *mono* **by** *blast*
qed

lemma *wf-on-mono-stronger*:
assumes
wf-on B r **and**
$f\ `\ A \subseteq B$ **and**
$(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow (x,\ y) \in q \Longrightarrow (f\ x,\ f\ y) \in r)$
shows *wf-on* A q
using *assms wfp-on-mono-stronger*[*to-set*, *of* B r f A q] **by** *blast*

lemma *wf-on-mono-strong*:
assumes *wf-on* B r **and** $A \subseteq B$ **and** $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow (x,\ y) \in q \Longrightarrow (x,\ y) \in r)$
shows *wf-on* A q
using *assms wf-on-mono-stronger*[*of* B r $\lambda x.\ x$ A q] **by** *blast*

lemma *wfp-on-mono-strong*:
wfp-on B $R \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow Q\ x\ y \Longrightarrow R\ x\ y) \Longrightarrow$ *wfp-on* A Q

using *wf-on-mono-strong*[*of B - A, to-pred*] **.**

lemma *wf-on-mono*: $A \subseteq B \Longrightarrow q \subseteq r \Longrightarrow$ *wf-on B r* $\leq$ *wf-on A q*
using *wf-on-mono-strong*[*of B r A q*] **by** *auto*

lemma *wfp-on-mono*: $A \subseteq B \Longrightarrow Q \leq R \Longrightarrow$ *wfp-on B R* $\leq$ *wfp-on A Q*
using *wfp-on-mono-strong*[*of B R A Q*] **by** *auto*

lemma *wf-on-subset*: *wf-on B r* $\Longrightarrow A \subseteq B \Longrightarrow$ *wf-on A r*
using *wf-on-mono-strong* **.**

lemma *wfp-on-subset*: *wfp-on B R* $\Longrightarrow A \subseteq B \Longrightarrow$ *wfp-on A R*
using *wfp-on-mono-strong* **.**

22.6.4 Equivalence between *wfp-on* and *wfp*

lemma *wfp-on-iff-wfp*: *wfp-on A R* $\longleftrightarrow$ *wfp* $(\lambda x\ y.\ R\ x\ y \wedge\ x \in A \wedge y \in A)$
(**is** *?LHS* $\longleftrightarrow$ *?RHS*)
proof (*rule iffI*)
assume *?LHS*
then show *?RHS*
unfolding *wfp-on-iff-ex-minimal*
by *force*
next
assume *?RHS*
thus *?LHS*
proof (*rule wfp-on-mono-strong*)
show $A \subseteq$ *UNIV*
using *subset-UNIV* **.**
next
show $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow R\ x\ y \wedge x \in A \wedge y \in A$
by *iprover*
qed
qed

22.6.5 Well-foundedness of transitive closure

lemma *bex-rtrancl-min-element-if-wf-on*:
assumes *wf*: *wf-on A r* **and** *x-in*: $x \in A$
shows $\exists\, y \in A.\ (y,\ x) \in r^* \wedge \neg(\exists\, z \in A.\ (z,\ y) \in r)$
using *wf*
proof (*induction x rule*: *wf-on-induct*)
case *in-set*
thus *?case*
using *x-in* **.**
next
case (*less z*)
show *?case*
proof (*cases* $\exists\, y \in A.\ (y,\ z) \in r$)
case *True*

```
    then obtain y where y ∈ A and (y, z) ∈ r
      by blast
    then obtain x where x ∈ A and (x, y) ∈ r* and ¬ (∃ w∈A. (w, x) ∈ r)
      using less.IH by blast
    show ?thesis
    proof (intro bexI conjI)
      show (x, z) ∈ r*
        using rtrancl.rtrancl-into-rtrancl[of x y r z]
        using ‹(x, y) ∈ r*› ‹(y, z) ∈ r› by blast
    next
      show ¬ (∃ z∈A. (z, x) ∈ r)
        using ‹¬ (∃ w∈A. (w, x) ∈ r)› .
    next
      show x ∈ A
        using ‹x ∈ A› .
    qed
  next
    case False
    show ?thesis
    proof (intro bexI conjI)
      show (z, z) ∈ r*
        using rtrancl.rtrancl-refl .
    next
      show ¬ (∃ w∈A. (w, z) ∈ r)
        using False .
    next
      show z ∈ A
        using less.hyps .
    qed
  qed
qed

lemma bex-rtransclp-min-element-if-wfp-on: wfp-on A R ⟹ x ∈ A ⟹ ∃ y∈A.
R** y x ∧ ¬ (∃ z∈A. R z y)
  by (rule bex-rtrancl-min-element-if-wf-on[to-pred])

lemma ex-terminating-rtranclp-strong:
  assumes wf: wfp-on {x'. R** x x'} R^-1-1
  shows ∃ y. R** x y ∧ (∄ z. R y z)
proof −
  have x-in: x ∈ {x'. R** x x'}
    by simp

  show ?thesis
    using bex-rtransclp-min-element-if-wfp-on[OF wf x-in]
    using rtranclp.rtrancl-into-rtrancl[of R x] by blast
qed

lemma ex-terminating-rtranclp:
```

assumes *wf*: *wfp* R^{-1-1}
shows $\exists\, y.\ R^{**}\ x\ y \wedge (\nexists\, z.\ R\ y\ z)$
using *ex-terminating-rtranclp-strong*[*OF wfp-on-subset*[*OF wf subset-UNIV*]] **.**

lemma *wf-trancl*:
assumes *wf r*
shows *wf* (r^+)
proof −
have *P x* **if** *induct-step*: $\bigwedge x.\ (\bigwedge y.\ (y,\ x) \in r^+ \Longrightarrow P\ y) \Longrightarrow P\ x$ **for** *P x*
proof (*rule induct-step*)
show *P y* **if** $(y,\ x) \in r^+$ **for** *y*
using ‹*wf r*› **and** *that*
proof (*induct x arbitrary*: *y*)
case (*less x*)
note *hyp* = $\langle\bigwedge x'\ y'.\ (x',\ x) \in r \Longrightarrow (y',\ x') \in r^+ \Longrightarrow P\ y'\rangle$
from ‹$(y,\ x) \in r^+$› **show** *P y*
proof *cases*
case *base*
show *P y*
proof (*rule induct-step*)
fix y'
assume $(y',\ y) \in r^+$
with ‹$(y,\ x) \in r$› **show** $P\ y'$
by (*rule hyp* [*of y y'*])
qed
next
case *step*
then obtain x' **where** $(x',\ x) \in r$ **and** $(y,\ x') \in r^+$
by *simp*
then show *P y* **by** (*rule hyp* [*of* x' *y*])
qed
qed
qed
then show *?thesis* **unfolding** *wf-def* **by** *blast*
qed

lemmas *wfp-tranclp* = *wf-trancl* [*to-pred*]

lemma *wf-converse-trancl*: *wf* $(r^{-1}) \Longrightarrow$ *wf* $((r^+)^{-1})$
apply (*subst trancl-converse* [*symmetric*])
apply (*erule wf-trancl*)
done

Well-foundedness of subsets

lemma *wf-subset*: *wf* $r \Longrightarrow p \subseteq r \Longrightarrow$ *wf p*
using *wf-on-mono*[*OF subset-UNIV*, *unfolded le-bool-def*] **..**

lemmas *wfp-subset* = *wf-subset* [*to-pred*]

Well-foundedness of the empty relation

lemma *wf-on-bot*[*iff*]: *wf-on A* $\bot$
by (*simp add*: *wf-on-def*)

lemma *wfp-on-bot*[*iff*]: *wfp-on A* $\bot$
using *wf-on-bot*[*to-pred*] **.**

lemma *wfp-empty* [*iff*]: *wfp* ($\lambda x\ y.\ False$)
using *wfp-on-bot* **by** (*simp add*: *bot-fun-def*)

lemma *wf-Int1*: *wf r* $\Longrightarrow$ *wf* ($r \cap r'$)
by (*erule wf-subset*) (*rule Int-lower1*)

lemma *wf-Int2*: *wf r* $\Longrightarrow$ *wf* ($r' \cap r$)
by (*erule wf-subset*) (*rule Int-lower2*)

Exponentiation.

lemma *wf-exp*:
assumes *wf* (R $\widehat{\ }\widehat{\ }$ *n*)
shows *wf R*
proof (*rule wfI-pf*)
fix *A* **assume** $A \subseteq R$ `` A
then have $A \subseteq (R \mathbin{\widehat{\ }\widehat{\ }} n)$ `` A
by (*induct n*) *force*+
with ‹*wf* (R $\widehat{\ }\widehat{\ }$ *n*)› **show** $A = \{\}$
by (*rule wfE-pf*)
qed

Well-foundedness of *insert*.

lemma *wf-insert* [*iff*]: *wf* (*insert* (*y*,*x*) *r*) $\longleftrightarrow$ *wf r* $\wedge$ (*x*,*y*) $\notin$ r^* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **then show** *?rhs*
by (*blast elim*: *wf-trancl* [*THEN wf-irrefl*]
intro: *rtrancl-into-trancl1 wf-subset rtrancl-mono* [*THEN subsetD*])
next
assume *R*: *?rhs*
then have *R'*: $Q \neq \{\} \Longrightarrow (\exists z{\in}Q.\ \forall y.\ (y,\ z) \in r \longrightarrow y \notin Q)$ **for** *Q*
by (*auto simp*: *wf-eq-minimal*)
show *?lhs*
unfolding *wf-eq-minimal*
proof *clarify*
fix *Q* :: *'a set* **and** *q*
assume $q \in Q$
then obtain *a* **where** $a \in Q$ **and** *a*: $\bigwedge y.\ (y,\ a) \in r \Longrightarrow y \notin Q$
using *R* **by** (*auto simp*: *wf-eq-minimal*)
show $\exists z{\in}Q.\ \forall y'.\ (y',\ z) \in$ *insert* $(y,\ x)$ $r \longrightarrow y' \notin Q$
proof (*cases a*=*x*)
case *True*
show *?thesis*
proof (*cases* $y \in Q$)

```
      case True
      then obtain z where z ∈ Q (z, y) ∈ r*
                        ⋀z'. (z', z) ∈ r ⟶ z' ∈ Q ⟶ (z', y) ∉ r*
        using R' [of {z ∈ Q. (z,y) ∈ r*}] by auto
      then have ∀ y'. (y', z) ∈ insert (y, x) r ⟶ y' ∉ Q
        using R by(blast intro: rtrancl-trans)+
      then show ?thesis
        by (rule bexI) fact
    next
      case False
      then show ?thesis
        using a ‹a ∈ Q› by blast
    qed
  next
    case False
    with a ‹a ∈ Q› show ?thesis
      by blast
  qed
 qed
qed
```

22.6.6 Well-foundedness of image

```
lemma wf-map-prod-image-Dom-Ran:
 fixes r:: ('a × 'a) set
   and f:: 'a ⇒ 'b
 assumes wf-r: wf r
   and inj: ⋀ a a'. a ∈ Domain r ⟹ a' ∈ Range r ⟹ f a = f a' ⟹ a = a'
 shows wf (map-prod f f ‘ r)
proof (unfold wf-eq-minimal, clarify)
 fix B :: 'b set and b::'b
 assume b ∈ B
 define A where A = f −‘ B ∩ Domain r
 show ∃ z∈B. ∀ y. (y, z) ∈ map-prod f f ‘ r ⟶ y ∉ B
 proof (cases A = {})
   case False
   then obtain a0 where a0 ∈ A and ∀ a. (a, a0) ∈ r ⟶ a ∉ A
     using wfE-min[OF wf-r] by auto
   thus ?thesis
     using inj unfolding A-def
     by (intro bexI[of - f a0]) auto
 qed (use ‹b ∈ B› in ‹unfold A-def, auto›)
qed

lemma wf-map-prod-image: wf r ⟹ inj f ⟹ wf (map-prod f f ‘ r)
by(rule wf-map-prod-image-Dom-Ran) (auto dest: inj-onD)

lemma wfp-on-image: wfp-on (f ‘ A) R ⟷ wfp-on A (λa b. R (f a) (f b))
proof (rule iffI)
```

```
  assume hyp: wfp-on (f ‘ A) R
  show wfp-on A (λa b. R (f a) (f b))
    unfolding wfp-on-iff-ex-minimal
  proof (intro allI impI)
    fix B
    assume B ⊆ A and B ≠ {}
    hence f ‘ B ⊆ f ‘ A and f ‘ B ≠ {}
      unfolding atomize-conj image-is-empty
      using image-mono by iprover
    hence ∃ z∈f ‘ B. ∀ y. R y z ⟶ y ∉ f ‘ B
      using hyp[unfolded wfp-on-iff-ex-minimal, rule-format] by iprover
    then obtain fz where fz ∈ f ‘ B and fz-max: ∀ y. R y fz ⟶ y ∉ f ‘ B ..

    obtain z where z ∈ B and fz = f z
      using ‹fz ∈ f ‘ B› unfolding image-iff ..

    show ∃ z∈B. ∀ y. R (f y) (f z) ⟶ y ∉ B
    proof (intro bexI allI impI)
      show z ∈ B
        using ‹z ∈ B› .
    next
      fix y assume R (f y) (f z)
      hence f y ∉ f ‘ B
        using fz-max ‹fz = f z› by iprover
      thus y ∉ B
        by (rule contrapos-nn) (rule imageI)
    qed
  qed
next
  assume hyp: wfp-on A (λa b. R (f a) (f b))
  show wfp-on (f ‘ A) R
    unfolding wfp-on-iff-ex-minimal
  proof (intro allI impI)
    fix fA
    assume fA ⊆ f ‘ A and fA ≠ {}
    then obtain A' where A' ⊆ A and A' ≠ {} and fA = f ‘ A'
      by (auto simp only: subset-image-iff)

    obtain z where z ∈ A' and z-max: ∀ y. R (f y) (f z) ⟶ y ∉ A'
      using hyp[unfolded wfp-on-iff-ex-minimal, rule-format, OF ‹A' ⊆ A› ‹A' ≠
{}›] by blast

    show ∃ z∈fA. ∀ y. R y z ⟶ y ∉ fA
    proof (intro bexI allI impI)
      show f z ∈ fA
        unfolding ‹fA = f ‘ A'›
        using imageI[OF ‹z ∈ A'›] .
    next
      show ⋀y. R y (f z) ⟹ y ∉ fA
```

```
      unfolding ‹fA = f ‘ A'›
      using z-max by auto
    qed
  qed
qed
```

22.7 Well-Foundedness Results for Unions

```
lemma wf-union-compatible:
  assumes wf R wf S
  assumes R O S ⊆ R
  shows wf (R ∪ S)
proof (rule wfI-min)
  fix x :: 'a and Q
  let ?Q' = {x ∈ Q. ∀ y. (y, x) ∈ R ⟶ y ∉ Q}
  assume x ∈ Q
  obtain a where a ∈ ?Q'
    by (rule wfE-min [OF ‹wf R› ‹x ∈ Q›]) blast
  with ‹wf S› obtain z where z ∈ ?Q' and zmin: ⋀y. (y, z) ∈ S ⟹ y ∉ ?Q'
    by (erule wfE-min)
  have y ∉ Q if (y, z) ∈ S for y
  proof
    from that have y ∉ ?Q' by (rule zmin)
    assume y ∈ Q
    with ‹y ∉ ?Q'› obtain w where (w, y) ∈ R and w ∈ Q by auto
    from ‹(w, y) ∈ R› ‹(y, z) ∈ S› have (w, z) ∈ R O S by (rule relcompI)
    with ‹R O S ⊆ R› have (w, z) ∈ R ..
    with ‹z ∈ ?Q'› have w ∉ Q by blast
    with ‹w ∈ Q› show False by contradiction
  qed
  with ‹z ∈ ?Q'› show ∃ z∈Q. ∀ y. (y, z) ∈ R ∪ S ⟶ y ∉ Q by blast
qed
```

Well-foundedness of indexed union with disjoint domains and ranges.

```
lemma wf-UN:
  assumes r: ⋀i. i ∈ I ⟹ wf (r i)
    and disj: ⋀i j. ⟦i ∈ I; j ∈ I; r i ≠ r j⟧ ⟹ Domain (r i) ∩ Range (r j) = {}
  shows wf (⋃ i∈I. r i)
  unfolding wf-eq-minimal
proof clarify
  fix A and a :: 'b
  assume a ∈ A
  show ∃ z∈A. ∀ y. (y, z) ∈ ⋃(r ‘ I) ⟶ y ∉ A
  proof (cases ∃ i∈I. ∃ a∈A. ∃ b∈A. (b, a) ∈ r i)
    case True
    then obtain i b c where ibc: i ∈ I b ∈ A c ∈ A (c,b) ∈ r i
      by blast
    have ri: ⋀Q. Q ≠ {} ⟹ ∃ z∈Q. ∀ y. (y, z) ∈ r i ⟶ y ∉ Q
      using r [OF ‹i ∈ I›] unfolding wf-eq-minimal by auto
```

```
    show ?thesis
      using ri [of {a. a ∈ A ∧ (∃ b∈A. (b, a) ∈ r i) }] ibc disj
      by blast
  next
    case False
    with ‹a ∈ A› show ?thesis
      by blast
  qed
qed

lemma wfp-SUP:
  ∀ i. wfp (r i) ⟹ ∀ i j. r i ≠ r j ⟶ inf (Domainp (r i)) (Rangep (r j)) = bot
⟹
    wfp (⨆ (range r))
  by (rule wf-UN[to-pred]) simp-all

lemma wf-Union:
  assumes ∀ r∈R. wf r
    and ∀ r∈R. ∀ s∈R. r ≠ s ⟶ Domain r ∩ Range s = {}
  shows wf (⋃ R)
  using assms wf-UN[of R λi. i] by simp
```

Intuition: We find an $R \cup S$-min element of a nonempty subset A by case distinction.

1. There is a step $a\ {-R\rightarrow}\ b$ with $a, b \in A$. Pick an R-min element z of the (nonempty) set $\{a{\in}A \mid \exists\, b{\in}A.\ a\ {-R\rightarrow}\ b\}$. By definition, there is $z' \in A$ s.t. $z\ {-R\rightarrow}\ z'$. Because z is R-min in the subset, z' must be R-min in A. Because z' has an R-predecessor, it cannot have an S-successor and is thus S-min in A as well.

2. There is no such step. Pick an S-min element of A. In this case it must be an R-min element of A as well.

```
lemma wf-Un: wf r ⟹ wf s ⟹ Domain r ∩ Range s = {} ⟹ wf (r ∪ s)
  using wf-union-compatible[of s r]
  by (auto simp: Un-ac)

lemma wf-union-merge: wf (R ∪ S) = wf (R O R ∪ S O R ∪ S)
  (is wf ?A = wf ?B)
proof
  assume wf ?A
  with wf-trancl have wfT: wf (?A⁺) .
  moreover have ?B ⊆ ?A⁺
    by (subst trancl-unfold, subst trancl-unfold) blast
  ultimately show wf ?B by (rule wf-subset)
next
  assume wf ?B
```

```
  show wf ?A
  proof (rule wfI-min)
    fix Q :: 'a set and x
    assume x ∈ Q
    with ‹wf ?B› obtain z where z ∈ Q and ⋀y. (y, z) ∈ ?B ⟹ y ∉ Q
      by (erule wfE-min)
    then have 1: ⋀y. (y, z) ∈ R O R ⟹ y ∉ Q
      and 2: ⋀y. (y, z) ∈ S O R ⟹ y ∉ Q
      and 3: ⋀y. (y, z) ∈ S ⟹ y ∉ Q
      by auto
    show ∃z∈Q. ∀y. (y, z) ∈ ?A ⟶ y ∉ Q
    proof (cases ∀y. (y, z) ∈ R ⟶ y ∉ Q)
      case True
      with ‹z ∈ Q› 3 show ?thesis by blast
    next
      case False
      then obtain z' where z'∈Q (z', z) ∈ R by blast
      have ∀y. (y, z') ∈ ?A ⟶ y ∉ Q
      proof (intro allI impI)
        fix y assume (y, z') ∈ ?A
        then show y ∉ Q
        proof
          assume (y, z') ∈ R
          then have (y, z) ∈ R O R using ‹(z', z) ∈ R› ..
          with 1 show y ∉ Q .
        next
          assume (y, z') ∈ S
          then have (y, z) ∈ S O R using  ‹(z', z) ∈ R› ..
          with 2 show y ∉ Q .
        qed
      qed
      with ‹z' ∈ Q› show ?thesis ..
    qed
  qed
qed

lemma wf-comp-self: wf R ⟷ wf (R O R)  — special case
  by (rule wf-union-merge [where S = {}, simplified])
```

22.8 Well-Foundedness of Composition

Bachmair and Dershowitz 1986, Lemma 2. [Provided by Tjark Weber]

```
lemma qc-wf-relto-iff:
  assumes R O S ⊆ (R ∪ S)* O R — R quasi-commutes over S
  shows wf (S* O R O S*) ⟷ wf R
    (is wf ?S ⟷ -)
proof
  show wf R if wf ?S
  proof −
```

```
    have R ⊆ ?S by auto
    with wf-subset [of ?S] that show wf R
      by auto
  qed
next
  show wf ?S if wf R
  proof (rule wfI-pf)
    fix A
    assume A: A ⊆ ?S `` A
    let ?X = (R ∪ S)* `` A
    have ∗: R O (R ∪ S)* ⊆ (R ∪ S)* O R
    proof −
      have (x, z) ∈ (R ∪ S)* O R if (y, z) ∈ (R ∪ S)* and (x, y) ∈ R for x y z
        using that
      proof (induct y z)
        case rtrancl-refl
        then show ?case by auto
      next
        case (rtrancl-into-rtrancl a b c)
        then have (x, c) ∈ ((R ∪ S)* O (R ∪ S)*) O R
          using assms by blast
        then show ?case by simp
      qed
      then show ?thesis by auto
    qed
    then have R O S* ⊆ (R ∪ S)* O R
      using rtrancl-Un-subset by blast
    then have ?S ⊆ (R ∪ S)* O (R ∪ S)* O R
      by (simp add: relcomp-mono rtrancl-mono)
    also have ... = (R ∪ S)* O R
      by (simp add: O-assoc[symmetric])
    finally have ?S O (R ∪ S)* ⊆ (R ∪ S)* O R O (R ∪ S)*
      by (simp add: O-assoc[symmetric] relcomp-mono)
    also have ... ⊆ (R ∪ S)* O (R ∪ S)* O R
      using ∗ by (simp add: relcomp-mono)
    finally have ?S O (R ∪ S)* ⊆ (R ∪ S)* O R
      by (simp add: O-assoc[symmetric])
    then have (?S O (R ∪ S)*) `` A ⊆ ((R ∪ S)* O R) `` A
      by (simp add: Image-mono)
    moreover have ?X ⊆ (?S O (R ∪ S)*) `` A
      using A by (auto simp: relcomp-Image)
    ultimately have ?X ⊆ R `` ?X
      by (auto simp: relcomp-Image)
    then have ?X = {}
      using ‹wf R› by (simp add: wfE-pf)
    moreover have A ⊆ ?X by auto
    ultimately show A = {} by simp
  qed
qed
```

corollary *wf-relcomp-compatible*:
assumes *wf R* **and** $R\ O\ S \subseteq S\ O\ R$
shows *wf* $(S\ O\ R)$
proof −
have $R\ O\ S \subseteq (R \cup S)^*\ O\ R$
using *assms* **by** *blast*
then have *wf* $(S^*\ O\ R\ O\ S^*)$
by (*simp add*: *assms qc-wf-relto-iff*)
then show *?thesis*
by (*rule Wellfounded.wf-subset*) *blast*
qed

22.9 Acyclic relations

lemma *wf-acyclic*: *wf r* $\Longrightarrow$ *acyclic r*
by (*simp add*: *acyclic-def*) (*blast elim*: *wf-trancl* [*THEN wf-irrefl*])

lemmas *wfp-acyclicP* = *wf-acyclic* [*to-pred*]

22.9.1 Wellfoundedness of finite acyclic relations

lemma *finite-acyclic-wf*:
assumes *finite r acyclic r* **shows** *wf r*
using *assms*
proof (*induction r rule*: *finite-induct*)
case (*insert x r*)
then show *?case*
by (*cases x*) *simp*
qed *simp*

lemma *finite-acyclic-wf-converse*: *finite r* $\Longrightarrow$ *acyclic r* $\Longrightarrow$ *wf* (r^{-1})
apply (*erule finite-converse* [*THEN iffD2*, *THEN finite-acyclic-wf*])
apply (*erule acyclic-converse* [*THEN iffD2*])
done

Observe that the converse of an irreflexive, transitive, and finite relation is again well-founded. Thus, we may employ it for well-founded induction.

lemma *wf-converse*:
assumes *irrefl r* **and** *trans r* **and** *finite r*
shows *wf* (r^{-1})
proof −
have *acyclic r*
using ‹*irrefl r*› **and** ‹*trans r*›
by (*simp add*: *irrefl-def acyclic-irrefl*)
with ‹*finite r*› **show** *?thesis*
by (*rule finite-acyclic-wf-converse*)
qed

lemma *wf-iff-acyclic-if-finite*: *finite r* $\Longrightarrow$ *wf r* = *acyclic r*
by (*blast intro*: *finite-acyclic-wf wf-acyclic*)

22.10 *nat* is well-founded

lemma *less-nat-rel*: (<) = $(\lambda m\ n.\ n = Suc\ m)^{++}$
proof (*rule ext*, *rule ext*, *rule iffI*)
fix *n m* :: *nat*
show $(\lambda m\ n.\ n = Suc\ m)^{++}$ *m n* **if** *m* < *n*
using *that*
proof (*induct n*)
case *0*
then show *?case* **by** *auto*
next
case (*Suc n*)
then show *?case*
by (*auto simp add*: *less-Suc-eq-le le-less intro*: *tranclp.trancl-into-trancl*)
qed
show *m* < *n* **if** $(\lambda m\ n.\ n = Suc\ m)^{++}$ *m n*
using *that* **by** (*induct n*) (*simp-all add*: *less-Suc-eq-le reflexive le-less*)
qed

definition *pred-nat* :: (*nat* × *nat*) *set*
where *pred-nat* = {(*m*, *n*). *n* = *Suc m*}

definition *less-than* :: (*nat* × *nat*) *set*
where *less-than* = *pred-nat*$^+$

lemma *less-eq*: (*m*, *n*) ∈ *pred-nat*$^+$ ⟷ *m* < *n*
unfolding *less-nat-rel pred-nat-def trancl-def* **by** *simp*

lemma *pred-nat-trancl-eq-le*: (*m*, *n*) ∈ *pred-nat** ⟷ *m* ≤ *n*
unfolding *less-eq rtrancl-eq-or-trancl* **by** *auto*

lemma *wf-pred-nat*: *wf pred-nat*
unfolding *wf-def*
proof *clarify*
fix *P x*
assume ∀ *x′*. (∀ *y*. (*y*, *x′*) ∈ *pred-nat* ⟶ *P y*) ⟶ *P x′*
then show *P x*
unfolding *pred-nat-def* **by** (*induction x*) *blast*+
qed

lemma *wf-less-than* [*iff*]: *wf less-than*
by (*simp add*: *less-than-def wf-pred-nat* [*THEN wf-trancl*])

lemma *trans-less-than* [*iff*]: *trans less-than*
by (*simp add*: *less-than-def*)

lemma *less-than-iff* [*iff*]: $((x,y) \in \mathit{less\text{-}than}) = (x<y)$
by (*simp add*: *less-than-def less-eq*)

lemma *irrefl-less-than*: *irrefl less-than*
using *irrefl-def* **by** *blast*

lemma *asym-less-than*: *asym less-than*
by (*rule asymI*) *simp*

lemma *total-less-than*: *total less-than* **and** *total-on-less-than* [*simp*]: *total-on A less-than*
using *total-on-def* **by** *force+*

lemma *wf-less*: *wf* $\{(x, y{::}nat).\ x < y\}$
by (*rule Wellfounded.wellorder-class.wf*)

22.11 Accessible Part

Inductive definition of the accessible part *acc r* of a relation; see also [6].

inductive-set *acc* :: $('a \times 'a)\ set \Rightarrow 'a\ set$ **for** *r* :: $('a \times 'a)\ set$
where *accI*: $(\bigwedge y.\ (y, x) \in r \Longrightarrow y \in acc\ r) \Longrightarrow x \in acc\ r$

abbreviation *termip* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow bool$
where *termip r* $\equiv$ *accp* (r^{-1-1})

abbreviation *termi* :: $('a \times 'a)\ set \Rightarrow 'a\ set$
where *termi r* $\equiv$ *acc* (r^{-1})

lemmas *accpI* = *accp.accI*

lemma *accp-eq-acc* [*code*]: *accp r* = $(\lambda x.\ x \in \mathit{Wellfounded.acc}\ \{(x, y).\ r\ x\ y\})$
by (*simp add*: *acc-def*)

Induction rules

theorem *accp-induct*:
assumes *major*: *accp r a*
assumes *hyp*: $\bigwedge x.\ accp\ r\ x \Longrightarrow \forall y.\ r\ y\ x \longrightarrow P\ y \Longrightarrow P\ x$
shows *P a*
apply (*rule major* [*THEN accp.induct*])
apply (*rule hyp*)
apply (*rule accp.accI*)
apply *auto*
done

lemmas *accp-induct-rule* = *accp-induct* [*rule-format*, *induct set*: *accp*]

theorem *accp-downward*: *accp r b* $\Longrightarrow$ *r a b* $\Longrightarrow$ *accp r a*
by (*cases rule*: *accp.cases*)

lemma *not-accp-down*:
assumes *na*: ¬ *accp R x*
obtains *z* **where** *R z x* **and** ¬ *accp R z*
proof −
assume *a*: ⋀*z*. *R z x* ⟹ ¬ *accp R z* ⟹ *thesis*
show *thesis*
proof (*cases* ∀ *z*. *R z x* ⟶ *accp R z*)
case *True*
then have ⋀*z*. *R z x* ⟹ *accp R z* **by** *auto*
then have *accp R x* **by** (*rule accp.accI*)
with *na* **show** *thesis* **..**
next
case *False* **then obtain** *z* **where** *R z x* **and** ¬ *accp R z*
by *auto*
with *a* **show** *thesis* **.**
qed
qed

lemma *accp-downwards-aux*: *r*** *b a* ⟹ *accp r a* ⟶ *accp r b*
by (*erule rtranclp-induct*) (*blast dest*: *accp-downward*)+

theorem *accp-downwards*: *accp r a* ⟹ *r*** *b a* ⟹ *accp r b*
by (*blast dest*: *accp-downwards-aux*)

theorem *accp-wfpI*: ∀ *x*. *accp r x* ⟹ *wfp r*
proof (*rule wfpUNIVI*)
fix *P x*
assume ∀ *x*. *accp r x* ∀ *x*. (∀ *y*. *r y x* ⟶ *P y*) ⟶ *P x*
then show *P x*
using *accp-induct*[**where** *P* = *P*] **by** *blast*
qed

theorem *accp-wfpD*: *wfp r* ⟹ *accp r x*
apply (*erule wfp-induct-rule*)
apply (*rule accp.accI*)
apply *blast*
done

theorem *wfp-iff-accp*: *wfp r* = (∀ *x*. *accp r x*)
by (*blast intro*: *accp-wfpI dest*: *accp-wfpD*)

Smaller relations have bigger accessible parts:

lemma *accp-subset*:
assumes *R1* ≤ *R2*
shows *accp R2* ≤ *accp R1*
proof (*rule predicate1I*)
fix *x*
assume *accp R2 x*
then show *accp R1 x*

proof (*induct x*)
fix *x*
assume $\bigwedge y.$ *R2 y x* $\Longrightarrow$ *accp R1 y*
with *assms* **show** *accp R1 x*
by (*blast intro*: *accp.accI*)
qed
qed

This is a generalized induction theorem that works on subsets of the accessible part.

lemma *accp-subset-induct*:
assumes *subset*: $D \leq$ *accp R*
and *dcl*: $\bigwedge x\ z.$ *D x* $\Longrightarrow$ *R z x* $\Longrightarrow$ *D z*
and *D x*
and *istep*: $\bigwedge x.$ *D x* $\Longrightarrow$ $(\bigwedge z.$ *R z x* $\Longrightarrow$ *P z*$)$ $\Longrightarrow$ *P x*
shows *P x*
proof −
from *subset* **and** ‹*D x*›
have *accp R x* **..**
then show *P x* **using** ‹*D x*›
proof (*induct x*)
fix *x*
assume *D x* **and** $\bigwedge y.$ *R y x* $\Longrightarrow$ *D y* $\Longrightarrow$ *P y*
with *dcl* **and** *istep* **show** *P x* **by** *blast*
qed
qed

Set versions of the above theorems

lemmas *acc-induct* = *accp-induct* [*to-set*]
lemmas *acc-induct-rule* = *acc-induct* [*rule-format*, *induct set*: *acc*]
lemmas *acc-downward* = *accp-downward* [*to-set*]
lemmas *not-acc-down* = *not-accp-down* [*to-set*]
lemmas *acc-downwards-aux* = *accp-downwards-aux* [*to-set*]
lemmas *acc-downwards* = *accp-downwards* [*to-set*]
lemmas *acc-wfI* = *accp-wfpI* [*to-set*]
lemmas *acc-wfD* = *accp-wfpD* [*to-set*]
lemmas *wf-iff-acc* = *wfp-iff-accp* [*to-set*]
lemmas *acc-subset* = *accp-subset* [*to-set*]
lemmas *acc-subset-induct* = *accp-subset-induct* [*to-set*]

22.12 Tools for building wellfounded relations

Inverse Image

lemma *wf-inv-image* [*simp*,*intro*!]:
fixes *f* :: $'a \Rightarrow 'b$
assumes *wf r*
shows *wf* (*inv-image r f*)
proof −

have $\bigwedge x\ P.\ x \in P \Longrightarrow \exists\, z{\in}P.\ \forall\, y.\ (f\ y, f\ z) \in r \longrightarrow y \notin P$
proof −
fix P **and** $x{::}'a$
assume $x \in P$
then obtain w **where** $w{:}\ w \in \{w.\ \exists\, x{::}'a.\ x \in P \wedge f\ x = w\}$
by *auto*
have $*{:}\ \bigwedge Q\ u.\ u \in Q \Longrightarrow \exists\, z{\in}Q.\ \forall\, y.\ (y,\ z) \in r \longrightarrow y \notin Q$
using *assms* **by** (*auto simp add*: *wf-eq-minimal*)
show $\exists\, z{\in}P.\ \forall\, y.\ (f\ y, f\ z) \in r \longrightarrow y \notin P$
using $*\ [OF\ w]$ **by** *auto*
qed
then show *?thesis*
by (*clarsimp simp*: *inv-image-def wf-eq-minimal*)
qed

lemma *wfp-on-inv-imagep*:
assumes *wf*: *wfp-on* $(f\ `\ A)\ R$
shows *wfp-on* A (*inv-imagep* $R\ f$)
unfolding *wfp-on-iff-ex-minimal*
proof (*intro allI impI*)
fix B **assume** $B \subseteq A$ **and** $B \neq \{\}$
hence $\exists\, z{\in}f\ `\ B.\ \forall\, y.\ R\ y\ z \longrightarrow y \notin f\ `\ B$
using *wf*[*unfolded wfp-on-iff-ex-minimal*, *rule-format*, *of* $f\ `\ B$] **by** *blast*
thus $\exists\, z{\in}B.\ \forall\, y.\ inv\text{-}imagep\ R\ f\ y\ z \longrightarrow y \notin B$
unfolding *inv-imagep-def*
by *auto*
qed

22.12.1 Conversion to a known well-founded relation

lemma *wfp-on-if-convertible-to-wfp-on*:
assumes
wf: *wfp-on* $(f\ `\ A)\ Q$ **and**
convertible: $(\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow R\ x\ y \Longrightarrow Q\ (f\ x)\ (f\ y))$
shows *wfp-on* $A\ R$
unfolding *wfp-on-iff-ex-minimal*
proof (*intro allI impI*)
fix B **assume** $B \subseteq A$ **and** $B \neq \{\}$
moreover from *wf* **have** *wfp-on* A (*inv-imagep* $Q\ f$)
by (*rule wfp-on-inv-imagep*)
ultimately obtain y **where** $y \in B$ **and** $\bigwedge z.\ Q\ (f\ z)\ (f\ y) \Longrightarrow z \notin B$
unfolding *wfp-on-iff-ex-minimal in-inv-imagep*
by *blast*
thus $\exists\, z \in B.\ \forall\, y.\ R\ y\ z \longrightarrow y \notin B$
using ‹$B \subseteq A$› *convertible* **by** *blast*
qed

lemma *wf-on-if-convertible-to-wf-on*: *wf-on* $(f\ `\ A)\ Q \Longrightarrow (\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow (x,\ y) \in R \Longrightarrow (f\ x,\ f\ y) \in Q) \Longrightarrow$ *wf-on* $A\ R$

using *wfp-on-if-convertible-to-wfp-on*[*to-set*] **.**

lemma *wf-if-convertible-to-wf*:
fixes $r :: 'a\ rel$ **and** $s :: 'b\ rel$ **and** $f :: 'a \Rightarrow 'b$
assumes *wf s* **and** *convertible*: $\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (f\ x,\ f\ y) \in s$
shows *wf r*
proof (*rule wf-on-if-convertible-to-wf-on*)
show *wf-on* (*range f*) *s*
using *wf-on-subset*[*OF* ‹*wf s*› *subset-UNIV*] **.**
next
show $\bigwedge x\ y.\ (x,\ y) \in r \Longrightarrow (f\ x,\ f\ y) \in s$
using *convertible* **.**
qed

lemma *wfp-if-convertible-to-wfp*: *wfp S* $\Longrightarrow$ $(\bigwedge x\ y.\ R\ x\ y \Longrightarrow S\ (f\ x)\ (f\ y)) \Longrightarrow$ *wfp R*
using *wf-if-convertible-to-wf*[*to-pred*, *of S R f*] **by** *simp*

Converting to *nat* is a very common special case that might be found more easily by Sledgehammer.

lemma *wfp-if-convertible-to-nat*:
fixes $f :: - \Rightarrow nat$
shows $(\bigwedge x\ y.\ R\ x\ y \Longrightarrow f\ x < f\ y) \Longrightarrow$ *wfp R*
by (*rule wfp-if-convertible-to-wfp*[*of* $(<) :: nat \Rightarrow nat \Rightarrow bool$, *simplified*])

22.12.2 Measure functions into *nat*

definition *measure* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a)\ set$
where *measure* = *inv-image less-than*

lemma *in-measure*[*simp*, *code-unfold*]: $(x,\ y) \in measure\ f \longleftrightarrow f\ x < f\ y$
by (*simp add*:*measure-def*)

lemma *wf-measure* [*iff*]: *wf* (*measure f*)
unfolding *measure-def* **by** (*rule wf-less-than* [*THEN wf-inv-image*])

lemma *wf-if-measure*: $(\bigwedge x.\ P\ x \Longrightarrow f(g\ x) < f\ x) \Longrightarrow wf\ \{(y,x).\ P\ x \wedge y = g\ x\}$
for $f :: 'a \Rightarrow nat$
using *wf-measure*[*of f*] **unfolding** *measure-def inv-image-def less-than-def less-eq*
by (*rule wf-subset*) *auto*

22.12.3 Lexicographic combinations

definition *lex-prod* :: $('a \times 'a)\ set \Rightarrow ('b \times 'b)\ set \Rightarrow (('a \times 'b) \times ('a \times 'b))\ set$
(**infixr** ‹<*lex*>› *80*)
where *ra* <*lex*> *rb* = $\{((a,\ b),\ (a',\ b')).\ (a,\ a') \in ra \vee a = a' \wedge (b,\ b') \in rb\}$

lemma *in-lex-prod*[*simp*]: $((a,\ b),\ (a',\ b')) \in r$ <*lex*> $s \longleftrightarrow (a,\ a') \in r \vee a = a' \wedge (b,\ b') \in s$

by (*auto simp*:*lex-prod-def*)

lemma *wf-on-lex-prod*[*intro*]:
assumes *wfA*: *wf-on A* r_A **and** *wfB*: *wf-on B* r_B
shows *wf-on* $(A \times B)$ $(r_A <*lex*> r_B)$
unfolding *wf-on-iff-ex-minimal*
proof (*intro allI impI*)
fix *AB* **assume** $AB \subseteq A \times B$ **and** $AB \neq \{\}$
hence *fst* ' *AB* $\subseteq A$ **and** *snd* ' *AB* $\subseteq B$
by *auto*

from ‹*fst* ' *AB* $\subseteq A$› ‹$AB \neq \{\}$› **obtain** *a* **where**
a-in: $a \in$ *fst* ' *AB* **and**
a-minimal: $(\forall y.\ (y, a) \in r_A \longrightarrow y \notin$ *fst* ' *AB*)
using *wfA*[*unfolded wf-on-iff-ex-minimal*, *rule-format*, *of fst* ' *AB*]
by *auto*

from ‹*snd* ' *AB* $\subseteq B$› ‹$AB \neq \{\}$› *a-in* **obtain** *b* **where**
b-in: $b \in$ *snd* ' $\{p \in AB.$ *fst* $p = a\}$ **and**
b-minimal: $(\forall y.\ (y, b) \in r_B \longrightarrow y \notin$ *snd* ' $\{p \in AB.$ *fst* $p = a\})$
using *wfB*[*unfolded wf-on-iff-ex-minimal*, *rule-format*, *of snd* ' $\{p \in AB.$ *fst* $p = a\}$]
by *blast*

show $\exists z \in AB.\ \forall y.\ (y, z) \in r_A <*lex*> r_B \longrightarrow y \notin AB$
proof (*rule bexI*)
show $(a, b) \in AB$
using *b-in* **by** (*simp add*: *image-iff*)
next
show $\forall y.\ (y, (a, b)) \in r_A <*lex*> r_B \longrightarrow y \notin AB$
proof (*intro allI impI*)
fix *p* **assume** $(p, (a, b)) \in r_A <*lex*> r_B$
hence $(fst\ p, a) \in r_A \vee fst\ p = a \wedge (snd\ p, b) \in r_B$
unfolding *lex-prod-def* **by** *auto*
thus $p \notin AB$
proof (*elim disjE conjE*)
assume $(fst\ p, a) \in r_A$
hence *fst p* $\notin$ *fst* ' *AB*
using *a-minimal* **by** *simp*
thus *?thesis*
by (*rule contrapos-nn*) *simp*
next
assume *fst p* $= a$ **and** $(snd\ p, b) \in r_B$
hence *snd p* $\notin$ *snd* ' $\{p \in AB.$ *fst* $p = a\}$
using *b-minimal* **by** *simp*
thus $p \notin AB$
by (*rule contrapos-nn*) (*simp add*: ‹*fst p* $= a$›)
qed
qed

qed
qed

lemma *wf-lex-prod* [*intro*!]:
assumes *wf ra wf rb*
shows *wf* (*ra* <∗*lex*∗> *rb*)
using *wf-on-lex-prod*[*OF* ‹*wf ra*› ‹*wf rb*›, *unfolded UNIV-Times-UNIV*] **.**

lemma *refl-lex-prod*[*simp*]: *refl* $r_B \Longrightarrow$ *refl* (r_A <∗*lex*∗> r_B)
by (*auto intro*!: *reflI dest*: *refl-onD*)

lemma *irrefl-on-lex-prod*[*simp*]:
irrefl-on A $r_A \Longrightarrow$ *irrefl-on B* $r_B \Longrightarrow$ *irrefl-on* ($A \times B$) (r_A <∗*lex*∗> r_B)
by (*auto intro*!: *irrefl-onI dest*: *irrefl-onD*)

lemma *irrefl-lex-prod*[*simp*]: *irrefl* $r_A \Longrightarrow$ *irrefl* $r_B \Longrightarrow$ *irrefl* (r_A <∗*lex*∗> r_B)
by (*rule irrefl-on-lex-prod*[*of UNIV - UNIV*, *unfolded UNIV-Times-UNIV*])

lemma *sym-on-lex-prod*[*simp*]:
sym-on A $r_A \Longrightarrow$ *sym-on B* $r_B \Longrightarrow$ *sym-on* ($A \times B$) (r_A <∗*lex*∗> r_B)
by (*auto intro*!: *sym-onI dest*: *sym-onD*)

lemma *sym-lex-prod*[*simp*]:
sym $r_A \Longrightarrow$ *sym* $r_B \Longrightarrow$ *sym* (r_A <∗*lex*∗> r_B)
by (*rule sym-on-lex-prod*[*of UNIV - UNIV*, *unfolded UNIV-Times-UNIV*])

lemma *asym-on-lex-prod*[*simp*]:
asym-on A $r_A \Longrightarrow$ *asym-on B* $r_B \Longrightarrow$ *asym-on* ($A \times B$) (r_A <∗*lex*∗> r_B)
by (*auto intro*!: *asym-onI dest*: *asym-onD*)

lemma *asym-lex-prod*[*simp*]:
asym $r_A \Longrightarrow$ *asym* $r_B \Longrightarrow$ *asym* (r_A <∗*lex*∗> r_B)
by (*rule asym-on-lex-prod*[*of UNIV - UNIV*, *unfolded UNIV-Times-UNIV*])

lemma *trans-on-lex-prod*[*simp*]:
assumes *trans-on A* r_A **and** *trans-on B* r_B
shows *trans-on* ($A \times B$) (r_A <∗*lex*∗> r_B)
proof (*rule trans-onI*)
fix *x y z*
show $x \in A \times B \Longrightarrow y \in A \times B \Longrightarrow z \in A \times B \Longrightarrow$
$(x, y) \in r_A$ <∗*lex*∗> $r_B \Longrightarrow (y, z) \in r_A$ <∗*lex*∗> $r_B \Longrightarrow (x, z) \in r_A$ <∗*lex*∗> r_B
using *trans-onD*[*OF* ‹*trans-on A* r_A›, *of fst x fst y fst z*]
using *trans-onD*[*OF* ‹*trans-on B* r_B›, *of snd x snd y snd z*]
by *auto*
qed

lemma *trans-lex-prod* [*simp*,*intro*!]: *trans* $r_A \Longrightarrow$ *trans* $r_B \Longrightarrow$ *trans* (r_A <∗*lex*∗> r_B)

by (*rule trans-on-lex-prod*[*of UNIV - UNIV*, *unfolded UNIV-Times-UNIV*])

lemma *total-on-lex-prod*[*simp*]:
total-on A $r_A \Longrightarrow$ *total-on B* $r_B \Longrightarrow$ *total-on* $(A \times B)$ $(r_A <*lex*> r_B)$
by (*auto simp*: *total-on-def*)

lemma *total-lex-prod*[*simp*]: *total* $r_A \Longrightarrow$ *total* $r_B \Longrightarrow$ *total* $(r_A <*lex*> r_B)$
by (*rule total-on-lex-prod*[*of UNIV - UNIV*, *unfolded UNIV-Times-UNIV*])

lexicographic combinations with measure functions

definition *mlex-prod* :: $('a \Rightarrow nat) \Rightarrow ('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set$ (**infixr** ‹<*mlex*>› *80*)
where $f <*mlex*> R = inv\text{-}image\ (less\text{-}than <*lex*> R)\ (\lambda x.\ (f\ x,\ x))$

lemma
wf-mlex: *wf R* $\Longrightarrow$ *wf* $(f <*mlex*> R)$ **and**
mlex-less: $f\ x < f\ y \Longrightarrow (x,\ y) \in f <*mlex*> R$ **and**
mlex-leq: $f\ x \leq f\ y \Longrightarrow (x,\ y) \in R \Longrightarrow (x,\ y) \in f <*mlex*> R$ **and**
mlex-iff: $(x,\ y) \in f <*mlex*> R \longleftrightarrow f\ x < f\ y \vee f\ x = f\ y \wedge (x,\ y) \in R$
by (*auto simp*: *mlex-prod-def*)

Proper subset relation on finite sets.

definition *finite-psubset* :: $('a\ set \times 'a\ set)\ set$
where *finite-psubset* $= \{(A,\ B).\ A \subset B \wedge finite\ B\}$

lemma *wf-finite-psubset*[*simp*]: *wf finite-psubset*
apply (*unfold finite-psubset-def*)
apply (*rule wf-measure* [*THEN wf-subset*])
apply (*simp add*: *measure-def inv-image-def less-than-def less-eq*)
apply (*fast elim*!: *psubset-card-mono*)
done

lemma *trans-finite-psubset*: *trans finite-psubset*
by (*auto simp*: *finite-psubset-def less-le trans-def*)

lemma *in-finite-psubset*[*simp*]: $(A,\ B) \in finite\text{-}psubset \longleftrightarrow A \subset B \wedge finite\ B$
unfolding *finite-psubset-def* **by** *auto*

max- and min-extension of order to finite sets

inductive-set *max-ext* :: $('a \times 'a)\ set \Rightarrow ('a\ set \times 'a\ set)\ set$
for R :: $('a \times 'a)\ set$
where *max-extI*[*intro*]:
finite X $\Longrightarrow$ *finite Y* $\Longrightarrow$ $Y \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow \exists\, y{\in}Y.\ (x,\ y) \in R) \Longrightarrow (X,\ Y) \in max\text{-}ext\ R$

lemma *max-ext-wf*:
assumes *wf*: *wf r*
shows *wf* (*max-ext r*)
proof (*rule acc-wfI*, *intro allI*)

```
  show M ∈ acc (max-ext r) (is - ∈ ?W) for M
  proof (induct M rule: infinite-finite-induct)
    case empty
    show ?case
      by (rule accI) (auto elim: max-ext.cases)
  next
    case (insert a M)
    from wf ‹M ∈ ?W› ‹finite M› show insert a M ∈ ?W
    proof (induct arbitrary: M)
      fix M a
      assume M ∈ ?W
      assume [intro]: finite M
     assume hyp: ⋀b M. (b, a) ∈ r ⟹ M ∈ ?W ⟹ finite M ⟹ insert b M ∈
?W
      have add-less: M ∈ ?W ⟹ (⋀y. y ∈ N ⟹ (y, a) ∈ r) ⟹ N ∪ M ∈ ?W
        if finite N finite M for N M :: 'a set
        using that by (induct N arbitrary: M) (auto simp: hyp)
      show insert a M ∈ ?W
      proof (rule accI)
        fix N
        assume Nless: (N, insert a M) ∈ max-ext r
        then have ∗: ⋀x. x ∈ N ⟹ (x, a) ∈ r ∨ (∃ y ∈ M. (x, y) ∈ r)
          by (auto elim!: max-ext.cases)

        let ?N1 = {n ∈ N. (n, a) ∈ r}
        let ?N2 = {n ∈ N. (n, a) ∉ r}
        have N: ?N1 ∪ ?N2 = N by (rule set-eqI) auto
        from Nless have finite N by (auto elim: max-ext.cases)
        then have finites: finite ?N1 finite ?N2 by auto

        have ?N2 ∈ ?W
        proof (cases M = {})
          case [simp]: True
          have Mw: {} ∈ ?W by (rule accI) (auto elim: max-ext.cases)
          from ∗ have ?N2 = {} by auto
          with Mw show ?N2 ∈ ?W by (simp only:)
        next
          case False
          from ∗ finites have N2: (?N2, M) ∈ max-ext r
            using max-extI[OF - - ‹M ≠ {}›, where ?X = ?N2] by auto
          with ‹M ∈ ?W› show ?N2 ∈ ?W by (rule acc-downward)
        qed
        with finites have ?N1 ∪ ?N2 ∈ ?W
          by (rule add-less) simp
        then show N ∈ ?W by (simp only: N)
      qed
    qed
  next
    case infinite
```

```
    show ?case
      by (rule accI) (auto elim: max-ext.cases simp: infinite)
  qed
qed

lemma max-ext-additive: (A, B) ∈ max-ext R ⟹ (C, D) ∈ max-ext R ⟹ (A ∪
C, B ∪ D) ∈ max-ext R
  by (force elim!: max-ext.cases)

definition min-ext :: ('a × 'a) set ⇒ ('a set × 'a set) set
  where min-ext r = {(X, Y) | X Y. X ≠ {} ∧ (∀ y ∈ Y. (∃ x ∈ X. (x, y) ∈ r))}

lemma min-ext-wf:
  assumes wf r
  shows wf (min-ext r)
proof (rule wfI-min)
  show ∃ m ∈ Q. (∀ n. (n, m) ∈ min-ext r ⟶ n ∉ Q) if nonempty: x ∈ Q
    for Q :: 'a set set and x
  proof (cases Q = {{}})
    case True
    then show ?thesis by (simp add: min-ext-def)
  next
    case False
    with nonempty obtain e x where x ∈ Q e ∈ x by force
    then have eU: e ∈ ⋃ Q by auto
    with ‹wf r›
    obtain z where z: z ∈ ⋃ Q ⋀y. (y, z) ∈ r ⟹ y ∉ ⋃ Q
      by (erule wfE-min)
    from z obtain m where m ∈ Q z ∈ m by auto
    from ‹m ∈ Q› show ?thesis
    proof (intro rev-bexI allI impI)
      fix n
      assume smaller: (n, m) ∈ min-ext r
      with ‹z ∈ m› obtain y where y ∈ n (y, z) ∈ r
        by (auto simp: min-ext-def)
      with z(2) show n ∉ Q by auto
    qed
  qed
qed
```

22.12.4 Bounded increase must terminate

```
lemma wf-bounded-measure:
  fixes ub :: 'a ⇒ nat
    and f :: 'a ⇒ nat
  assumes ⋀a b. (b, a) ∈ r ⟹ ub b ≤ ub a ∧ ub a ≥ f b ∧ f b > f a
  shows wf r
  by (rule wf-subset[OF wf-measure[of λa. ub a − f a]]) (auto dest: assms)
```

lemma *wf-bounded-set*:
fixes *ub* :: *'a* ⇒ *'b set*
and *f* :: *'a* ⇒ *'b set*
assumes ⋀*a b*. (*b*,*a*) ∈ *r* ⟹ *finite* (*ub a*) ∧ *ub b* ⊆ *ub a* ∧ *ub a* ⊇ *f b* ∧ *f b* ⊃ *f a*
shows *wf r*
apply (*rule wf-bounded-measure*[*of r* λ*a*. *card* (*ub a*) λ*a*. *card* (*f a*)])
apply (*drule assms*)
apply (*blast intro*: *card-mono finite-subset psubset-card-mono dest*: *psubset-eq*[*THEN iffD2*])
done

lemma *finite-subset-wf*:
assumes *finite A*
shows *wf* {(*X*, *Y*). *X* ⊂ *Y* ∧ *Y* ⊆ *A*}
by (*rule wf-subset*[*OF wf-finite-psubset*[*unfolded finite-psubset-def*]])
(*auto intro*: *finite-subset*[*OF - assms*])

hide-const (**open**) *acc accp*

22.13 Code Generation Setup

Code equations with *wf* or *wfp* on the left-hand side are not supported by the code generation module because of the *UNIV* hidden behind the abbreviations. To sidestep this problem, we provide the following wrapper definitions and use *code-abbrev* to register the definitions with the pre- and post-processors of the code generator.

definition *wf-code* :: (*'a* × *'a*) *set* ⇒ *bool* **where**
[*code-abbrev*]: *wf-code r* ⟷ *wf r*

definition *wfp-code* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *bool* **where**
[*code-abbrev*]: *wfp-code R* ⟷ *wfp R*

end

23 Well-Founded Recursion Combinator

theory *Wfrec*
imports *Wellfounded*
begin

inductive *wfrec-rel* :: (*'a* × *'a*) *set* ⇒ ((*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*)) ⇒ *'a* ⇒ *'b* ⇒ *bool* **for** *R F*
where *wfrecI*: (⋀*z*. (*z*, *x*) ∈ *R* ⟹ *wfrec-rel R F z* (*g z*)) ⟹ *wfrec-rel R F x* (*F g x*)

definition *cut* :: (*'a* ⇒ *'b*) ⇒ (*'a* × *'a*) *set* ⇒ *'a* ⇒ *'a* ⇒ *'b*
where *cut f R x* = (λ*y*. *if* (*y*, *x*) ∈ *R then f y else undefined*)

definition *adm-wf* :: $('a \times 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow bool$
where *adm-wf R F* $\longleftrightarrow$ $(\forall f\ g\ x.\ (\forall z.\ (z, x) \in R \longrightarrow f\ z = g\ z) \longrightarrow F\ f\ x = F\ g\ x)$

definition *wfrec* :: $('a \times 'a)\ set \Rightarrow (('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)) \Rightarrow ('a \Rightarrow 'b)$
where *wfrec R F* $= (\lambda x.\ THE\ y.\ wfrec\text{-}rel\ R\ (\lambda f\ x.\ F\ (cut\ f\ R\ x)\ x)\ x\ y)$

lemma *cuts-eq*: $(cut\ f\ R\ x = cut\ g\ R\ x) \longleftrightarrow (\forall y.\ (y, x) \in R \longrightarrow f\ y = g\ y)$
by (*simp add*: *fun-eq-iff cut-def*)

lemma *cut-apply*: $(x, a) \in R \Longrightarrow cut\ f\ R\ a\ x = f\ x$
by (*simp add*: *cut-def*)

Inductive characterization of *wfrec* combinator; for details see: John Harrison, "Inductive definitions: automation and application".

lemma *theI-unique*: $\exists!x.\ P\ x \Longrightarrow P\ x \longleftrightarrow x = The\ P$
by (*auto intro*: *the-equality*[*symmetric*] *theI*)

lemma *wfrec-unique*:
assumes *adm-wf R F wf R*
shows $\exists!y.$ *wfrec-rel R F x y*
using ‹*wf R*›
proof *induct*
define *f* **where** *f y* = (*THE z. wfrec-rel R F y z*) **for** *y*
case (*less x*)
then have $\bigwedge y\ z.\ (y, x) \in R \Longrightarrow$ *wfrec-rel R F y z* $\longleftrightarrow$ *z* = *f y*
unfolding *f-def* **by** (*rule theI-unique*)
with ‹*adm-wf R F*› **show** *?case*
by (*subst wfrec-rel.simps*) (*auto simp*: *adm-wf-def*)
qed

lemma *adm-lemma*: *adm-wf R* $(\lambda f\ x.\ F\ (cut\ f\ R\ x)\ x)$
by (*auto simp*: *adm-wf-def intro*!: *arg-cong*[**where** *f*=$\lambda x.$ *F x y* **for** *y*] *cuts-eq*[*THEN iffD2*])

lemma *wfrec*: *wf R* $\Longrightarrow$ *wfrec R F a* = *F* (*cut* (*wfrec R F*) *R a*) *a*
apply (*simp add*: *wfrec-def*)
apply (*rule adm-lemma* [*THEN wfrec-unique*, *THEN the1-equality*])
apply *assumption*
apply (*rule wfrec-rel.wfrecI*)
apply (*erule adm-lemma* [*THEN wfrec-unique*, *THEN theI'*])
done

This form avoids giant explosions in proofs. NOTE USE OF $\equiv$.

lemma *def-wfrec*: *f* $\equiv$ *wfrec R F* $\Longrightarrow$ *wf R* $\Longrightarrow$ *f a* = *F* (*cut f R a*) *a*
by (*auto intro*: *wfrec*)

23.0.1 Well-founded recursion via genuine fixpoints

lemma *wfrec-fixpoint*:
assumes *wf*: *wf R*
and *adm*: *adm-wf R F*
shows *wfrec R F* = *F* (*wfrec R F*)
proof (*rule ext*)
fix *x*
have *wfrec R F x* = *F* (*cut* (*wfrec R F*) *R x*) *x*
using *wfrec*[*of R F*] *wf* **by** *simp*
also
have ⋀*y*. (*y*, *x*) ∈ *R* ⟹ *cut* (*wfrec R F*) *R x y* = *wfrec R F y*
by (*auto simp add*: *cut-apply*)
then have *F* (*cut* (*wfrec R F*) *R x*) *x* = *F* (*wfrec R F*) *x*
using *adm adm-wf-def*[*of R F*] **by** *auto*
finally show *wfrec R F x* = *F* (*wfrec R F*) *x* **.**
qed

lemma *wfrec-def-adm*: *f* ≡ *wfrec R F* ⟹ *wf R* ⟹ *adm-wf R F* ⟹ *f* = *F f*
using *wfrec-fixpoint* **by** *simp*

For completeness: uniqueness of fixed points

lemma *fixp-same-if-wf-adm-wf*:
assumes *wf R adm-wf R F f* = *F f g* = *F g*
shows *f* = *g*
proof
fix *x*
show *f x* = *g x*
using ‹*wf R*› **proof** *induction*
case (*less u*)
with ‹*adm-wf R F*›
have *F f u* = *F g u*
by (*simp add*: *adm-wf-def*)
with ‹*f* = *F f*› ‹*g* = *F g*›
show *f u* = *g u*
by *auto*
qed
qed

corollary *fixp-eq-wfrec-if-wf-adm-wf*:
⟦ *wf R*; *adm-wf R F*; *f* = *F f* ⟧ ⟹ *f* = *wfrec R F*
using *wfrec-def-adm fixp-same-if-wf-adm-wf* **by** *blast*

23.1 Wellfoundedness of *same-fst*

definition *same-fst* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ ('*b* × '*b*) *set*) ⇒ (('*a* × '*b*) × ('*a* × '*b*)) *set*
where *same-fst P R* = {((*x*′, *y*′), (*x*, *y*)) . *x*′ = *x* ∧ *P x* ∧ (*y*′,*y*) ∈ *R x*}
— For *wfrec* declarations where the first n parameters stay unchanged in the recursive call.

lemma *same-fstI* [*intro*!]: $P\ x \Longrightarrow (y',\ y) \in R\ x \Longrightarrow ((x,\ y'),\ (x,\ y)) \in$ *same-fst* $P\ R$
by (*simp add*: *same-fst-def*)

lemma *wf-same-fst*:
assumes $\bigwedge x.\ P\ x \Longrightarrow$ *wf* $(R\ x)$
shows *wf* (*same-fst* $P\ R$)
proof −
have $\bigwedge a\ b\ Q.\ \forall a\ b.\ (\forall x.\ P\ a \wedge (x,\ b) \in R\ a \longrightarrow Q\ (a,\ x)) \longrightarrow Q\ (a,\ b) \Longrightarrow Q\ (a,\ b)$
proof −
fix $Q\ a\ b$
assume ∗: $\forall a\ b.\ (\forall x.\ P\ a \wedge (x,b) \in R\ a \longrightarrow Q\ (a,x)) \longrightarrow Q\ (a,b)$
show $Q(a,b)$
proof (*cases wf* $(R\ a)$)
case *True*
then show *?thesis*
by (*induction b rule*: *wf-induct-rule*) (*use* ∗ **in** *blast*)
qed (*use* ∗ *assms* **in** *blast*)
qed
then show *?thesis*
by (*clarsimp simp add*: *wf-def same-fst-def*)
qed

hide-const (**open**) *cut*

end

24 Orders as Relations

theory *Order-Relation*
imports *Wfrec*
begin

24.1 Orders on a set

definition *preorder-on* $A\ r \equiv r \subseteq A \times A \wedge$ *refl-on* $A\ r \wedge$ *trans* r

definition *partial-order-on* $A\ r \equiv$ *preorder-on* $A\ r \wedge$ *antisym* r

definition *linear-order-on* $A\ r \equiv$ *partial-order-on* $A\ r \wedge$ *total-on* $A\ r$

definition *strict-linear-order-on* $A\ r \equiv$ *trans* $r \wedge$ *irrefl* $r \wedge$ *total-on* $A\ r$

definition *well-order-on* $A\ r \equiv$ *linear-order-on* $A\ r \wedge$ *wf*$(r - Id)$

lemmas *order-on-defs* =
preorder-on-def partial-order-on-def linear-order-on-def

strict-linear-order-on-def well-order-on-def

lemma *partial-order-onD*:
assumes *partial-order-on A r* **shows** *refl-on A r* **and** *trans r* **and** *antisym r* **and** $r \subseteq A \times A$
using *assms* **unfolding** *partial-order-on-def preorder-on-def* **by** *auto*

lemma *preorder-on-empty*[*simp*]: *preorder-on* {} {}
by (*simp add*: *preorder-on-def trans-def*)

lemma *partial-order-on-empty*[*simp*]: *partial-order-on* {} {}
by (*simp add*: *partial-order-on-def*)

lemma *lnear-order-on-empty*[*simp*]: *linear-order-on* {} {}
by (*simp add*: *linear-order-on-def*)

lemma *well-order-on-empty*[*simp*]: *well-order-on* {} {}
by (*simp add*: *well-order-on-def*)

lemma *preorder-on-converse*[*simp*]: *preorder-on A* (r^{-1}) = *preorder-on A r*
by (*auto simp add*: *preorder-on-def*)

lemma *partial-order-on-converse*[*simp*]: *partial-order-on A* (r^{-1}) = *partial-order-on A r*
by (*simp add*: *partial-order-on-def*)

lemma *linear-order-on-converse*[*simp*]: *linear-order-on A* (r^{-1}) = *linear-order-on A r*
by (*simp add*: *linear-order-on-def*)

lemma *partial-order-on-acyclic*:
partial-order-on A r $\Longrightarrow$ *acyclic* $(r - Id)$
by (*simp add*: *acyclic-irrefl partial-order-on-def preorder-on-def trans-diff-Id*)

lemma *partial-order-on-well-order-on*:
finite r $\Longrightarrow$ *partial-order-on A r* $\Longrightarrow$ *wf* $(r - Id)$
by (*simp add*: *finite-acyclic-wf partial-order-on-acyclic*)

lemma *strict-linear-order-on-diff-Id*: *linear-order-on A r* $\Longrightarrow$ *strict-linear-order-on A* $(r - Id)$
by (*simp add*: *order-on-defs trans-diff-Id*)

lemma *linear-order-on-singleton* [*simp*]: *linear-order-on* {*x*} {(*x*, *x*)}
by (*simp add*: *order-on-defs*)

lemma *linear-order-on-acyclic*:
assumes *linear-order-on A r*

shows *acyclic* $(r - Id)$
using *strict-linear-order-on-diff-Id*[*OF assms*]
by (*auto simp add*: *acyclic-irrefl strict-linear-order-on-def*)

lemma *linear-order-on-well-order-on*:
assumes *finite r*
shows *linear-order-on A r* $\longleftrightarrow$ *well-order-on A r*
unfolding *well-order-on-def*
using *assms finite-acyclic-wf*[*OF - linear-order-on-acyclic, of r*] **by** *blast*

24.2 Orders on the field

abbreviation *Refl r* $\equiv$ *refl-on* (*Field r*) *r*

abbreviation *Preorder r* $\equiv$ *preorder-on* (*Field r*) *r*

abbreviation *Partial-order r* $\equiv$ *partial-order-on* (*Field r*) *r*

abbreviation *Total r* $\equiv$ *total-on* (*Field r*) *r*

abbreviation *Linear-order r* $\equiv$ *linear-order-on* (*Field r*) *r*

abbreviation *Well-order r* $\equiv$ *well-order-on* (*Field r*) *r*

lemma *subset-Image-Image-iff*:
Preorder r $\Longrightarrow$ $A \subseteq$ *Field r* $\Longrightarrow$ $B \subseteq$ *Field r* $\Longrightarrow$
r `` $A \subseteq r$ `` $B \longleftrightarrow (\forall a{\in}A.\exists b{\in}B.\ (b,\ a) \in r)$
apply (*simp add*: *preorder-on-def refl-on-def Image-def subset-eq*)
apply (*simp only*: *trans-def*)
apply *fast*
done

lemma *subset-Image1-Image1-iff*:
Preorder r $\Longrightarrow$ $a \in$ *Field r* $\Longrightarrow$ $b \in$ *Field r* $\Longrightarrow$ r `` $\{a\} \subseteq r$ `` $\{b\} \longleftrightarrow (b,\ a) \in r$
by (*simp add*: *subset-Image-Image-iff*)

lemma *Refl-antisym-eq-Image1-Image1-iff*:
assumes *Refl r*
and *as*: *antisym r*
and *abf*: $a \in$ *Field r* $b \in$ *Field r*
shows r `` $\{a\} = r$ `` $\{b\} \longleftrightarrow a = b$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have $*$: $\bigwedge x.\ (a,\ x) \in r \longleftrightarrow (b,\ x) \in r$
by (*simp add*: *set-eq-iff*)
have $(a,\ a) \in r$ $(b,\ b) \in r$ **using** ‹*Refl r*› *abf* **by** (*simp-all add*: *refl-on-def*)

then have $(a, b) \in r$ $(b, a) \in r$ **using** $*[of\ a]$ $*[of\ b]$ **by** *simp-all*
then show *?rhs*
using ‹*antisym r*›[*unfolded antisym-def*] **by** *blast*
next
assume *?rhs*
then show *?lhs* **by** *fast*
qed

lemma *Partial-order-eq-Image1-Image1-iff*:
Partial-order r $\Longrightarrow$ $a \in$ *Field r* $\Longrightarrow$ $b \in$ *Field r* $\Longrightarrow$ r `` $\{a\} = r$ `` $\{b\} \longleftrightarrow a = b$
by (*auto simp*: *order-on-defs Refl-antisym-eq-Image1-Image1-iff*)

lemma *Total-Id-Field*:
assumes *Total r*
and *not-Id*: $\neg\ r \subseteq Id$
shows *Field r* $=$ *Field* $(r - Id)$
proof −
have *Field r* $\subseteq$ *Field* $(r - Id)$
proof (*rule subsetI*)
fix *a* **assume** $*$: $a \in$ *Field r*
from *not-Id* **have** $r \neq \{\}$ **by** *fast*
with *not-Id* **obtain** *b* **and** *c* **where** $b \neq c \wedge (b,c) \in r$ **by** *auto*
then have $b \neq c \wedge \{b, c\} \subseteq$ *Field r* **by** (*auto simp*: *Field-def*)
with $*$ **obtain** *d* **where** $d \in$ *Field r* $d \neq a$ **by** *auto*
with $*$ ‹*Total r*› **have** $(a, d) \in r \vee (d, a) \in r$ **by** (*simp add*: *total-on-def*)
with ‹$d \neq a$› **show** $a \in$ *Field* $(r - Id)$ **unfolding** *Field-def* **by** *blast*
qed
then show *?thesis*
using *mono-Field*[*of r* − *Id r*] *Diff-subset*[*of r Id*] **by** *auto*
qed

24.3 Relations given by a predicate and the field

definition *relation-of* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow ('a \times 'a)\ set$
where *relation-of P A* $\equiv \{ (a, b) \in A \times A.\ P\ a\ b \}$

lemma *refl-relation-ofD*: *refl* (*relation-of R S*) $\Longrightarrow$ *reflp-on S R*
by (*auto simp*: *relation-of-def intro*: *reflp-onI dest*: *reflD*)

lemma *irrefl-relation-ofD*: *irrefl* (*relation-of R S*) $\Longrightarrow$ *irreflp-on S R*
by (*auto simp*: *relation-of-def intro*: *irreflp-onI dest*: *irreflD*)

lemma *sym-relation-of*[*simp*]: *sym* (*relation-of R S*) $\longleftrightarrow$ *symp-on S R*
proof (*rule iffI*)
show *sym* (*relation-of R S*) $\Longrightarrow$ *symp-on S R*
by (*auto simp*: *relation-of-def intro*: *symp-onI dest*: *symD*)
next
show *symp-on S R* $\Longrightarrow$ *sym* (*relation-of R S*)

by (*auto simp*: *relation-of-def intro*: *symI dest*: *symp-onD*)
qed

lemma *asym-relation-of*[*simp*]: *asym* (*relation-of R S*) ⟷ *asymp-on S R*
proof (*rule iffI*)
show *asym* (*relation-of R S*) ⟹ *asymp-on S R*
by (*auto simp*: *relation-of-def intro*: *asymp-onI dest*: *asymD*)
next
show *asymp-on S R* ⟹ *asym* (*relation-of R S*)
by (*auto simp*: *relation-of-def intro*: *asymI dest*: *asymp-onD*)
qed

lemma *antisym-relation-of*[*simp*]: *antisym* (*relation-of R S*) ⟷ *antisymp-on S R*
proof (*rule iffI*)
show *antisym* (*relation-of R S*) ⟹ *antisymp-on S R*
by (*simp add*: *antisym-on-def antisymp-on-def relation-of-def*)
next
show *antisymp-on S R* ⟹ *antisym* (*relation-of R S*)
by (*simp add*: *antisym-on-def antisymp-on-def relation-of-def*)
qed

lemma *trans-relation-of*[*simp*]: *trans* (*relation-of R S*) ⟷ *transp-on S R*
proof (*rule iffI*)
show *trans* (*relation-of R S*) ⟹ *transp-on S R*
by (*auto simp*: *relation-of-def intro*: *transp-onI dest*: *transD*)
next
show *transp-on S R* ⟹ *trans* (*relation-of R S*)
by (*auto simp*: *relation-of-def intro*: *transI dest*: *transp-onD*)
qed

lemma *total-relation-ofD*: *total* (*relation-of R S*) ⟹ *totalp-on S R*
by (*auto simp*: *relation-of-def total-on-def intro*: *totalp-onI*)

lemma *Field-relation-of*:
assumes *relation-of P A* ⊆ *A* × *A* **and** *refl-on A* (*relation-of P A*)
shows *Field* (*relation-of P A*) = *A*
using *assms* **unfolding** *refl-on-def Field-def* **by** *auto*

lemma *partial-order-on-relation-ofI*:
assumes *refl*: ⋀*a*. *a* ∈ *A* ⟹ *P a a*
and *trans*: ⋀*a b c*. ⟦ *a* ∈ *A*; *b* ∈ *A*; *c* ∈ *A* ⟧ ⟹ *P a b* ⟹ *P b c* ⟹ *P a c*
and *antisym*: ⋀*a b*. ⟦ *a* ∈ *A*; *b* ∈ *A* ⟧ ⟹ *P a b* ⟹ *P b a* ⟹ *a* = *b*
shows *partial-order-on A* (*relation-of P A*)
proof −
have *relation-of P A* ⊆ *A* × *A*
unfolding *relation-of-def* **by** *auto*
moreover have *refl-on A* (*relation-of P A*)
using *refl* **unfolding** *refl-on-def relation-of-def* **by** *auto*

moreover have *trans* (*relation-of P A*) **and** *antisym* (*relation-of P A*)
unfolding *relation-of-def*
by (*auto intro*: *transI dest*: *trans*, *auto intro*: *antisymI dest*: *antisym*)
ultimately show *?thesis*
unfolding *partial-order-on-def preorder-on-def* **by** *simp*
qed

lemma *Partial-order-relation-ofI*:
assumes *partial-order-on A* (*relation-of P A*)
shows *Partial-order* (*relation-of P A*)
proof −
have ∗: *Field* (*relation-of P A*) = *A*
using *assms* **by** (*simp-all only*: *Field-relation-of partial-order-on-def preorder-on-def*)
show *?thesis*
unfolding ∗
using *assms* **.**
qed

24.4 Orders on a type

abbreviation *strict-linear-order* ≡ *strict-linear-order-on UNIV*

abbreviation *linear-order* ≡ *linear-order-on UNIV*

abbreviation *well-order* ≡ *well-order-on UNIV*

24.5 Order-like relations

In this subsection, we develop basic concepts and results pertaining to order-like relations, i.e., to reflexive and/or transitive and/or symmetric and/or total relations. We also further define upper and lower bounds operators.

24.5.1 Auxiliaries

corollary *well-order-on-domain*: *well-order-on A r* ⟹ (*a*, *b*) ∈ *r* ⟹ *a* ∈ *A* ∧ *b* ∈ *A*
by (*auto simp add*: *order-on-defs*)

lemma *well-order-on-Field*: *well-order-on A r* ⟹ *A* = *Field r*
by (*auto simp add*: *refl-on-def Field-def order-on-defs*)

lemma *well-order-on-Well-order*: *well-order-on A r* ⟹ *A* = *Field r* ∧ *Well-order r*
using *well-order-on-Field* [*of A*] **by** *auto*

lemma *Total-subset-Id*:
assumes *Total r*
and *r* ⊆ *Id*
shows *r* = {} ∨ (∃ *a*. *r* = {(*a*, *a*)})

proof −
have $\exists\, a.\ r = \{(a,\ a)\}$ **if** $r \neq \{\}$
proof −
from *that* **obtain** $a\ b$ **where** *ab*: $(a,\ b) \in r$ **by** *fast*
with ‹$r \subseteq Id$› **have** $a = b$ **by** *blast*
with *ab* **have** *aa*: $(a,\ a) \in r$ **by** *simp*
have $a = c \wedge a = d$ **if** $(c,\ d) \in r$ **for** $c\ d$
proof −
from *that* **have** $\{a,\ c,\ d\} \subseteq Field\ r$
using *ab* **unfolding** *Field-def* **by** *blast*
then have $((a,\ c) \in r \vee (c,\ a) \in r \vee a = c) \wedge ((a,\ d) \in r \vee (d,\ a) \in r \vee a = d)$
using ‹*Total r*› **unfolding** *total-on-def* **by** *blast*
with ‹$r \subseteq Id$› **show** *?thesis* **by** *blast*
qed
then have $r \subseteq \{(a,\ a)\}$ **by** *auto*
with *aa* **show** *?thesis* **by** *blast*
qed
then show *?thesis* **by** *blast*
qed

lemma *Linear-order-in-diff-Id*:
assumes *Linear-order r*
and $a \in Field\ r$
and $b \in Field\ r$
shows $(a,\ b) \in r \longleftrightarrow (b,\ a) \notin r - Id$
using *assms* **unfolding** *order-on-defs total-on-def antisym-def Id-def refl-on-def* **by** *force*

24.5.2 The upper and lower bounds operators

Here we define upper ("above") and lower ("below") bounds operators. We think of r as a *non-strict* relation. The suffix S at the names of some operators indicates that the bounds are strict – e.g., *underS a* is the set of all strict lower bounds of a (w.r.t. r). Capitalization of the first letter in the name reminds that the operator acts on sets, rather than on individual elements.

definition *under* :: *'a rel* ⇒ *'a* ⇒ *'a set*
where *under r a* ≡ $\{b.\ (b,\ a) \in r\}$

definition *underS* :: *'a rel* ⇒ *'a* ⇒ *'a set*
where *underS r a* ≡ $\{b.\ b \neq a \wedge (b,\ a) \in r\}$

definition *Under* :: *'a rel* ⇒ *'a set* ⇒ *'a set*
where *Under r A* ≡ $\{b \in Field\ r.\ \forall\, a \in A.\ (b,\ a) \in r\}$

definition *UnderS* :: *'a rel* ⇒ *'a set* ⇒ *'a set*
where *UnderS r A* ≡ $\{b \in Field\ r.\ \forall\, a \in A.\ b \neq a \wedge (b,\ a) \in r\}$

definition *above* :: *'a rel ⇒ 'a ⇒ 'a set*
where *above r a ≡ {b. (a, b) ∈ r}*

definition *aboveS* :: *'a rel ⇒ 'a ⇒ 'a set*
where *aboveS r a ≡ {b. b ≠ a ∧ (a, b) ∈ r}*

definition *Above* :: *'a rel ⇒ 'a set ⇒ 'a set*
where *Above r A ≡ {b ∈ Field r. ∀ a ∈ A. (a, b) ∈ r}*

definition *AboveS* :: *'a rel ⇒ 'a set ⇒ 'a set*
where *AboveS r A ≡ {b ∈ Field r. ∀ a ∈ A. b ≠ a ∧ (a, b) ∈ r}*

definition *ofilter* :: *'a rel ⇒ 'a set ⇒ bool*
where *ofilter r A ≡ A ⊆ Field r ∧ (∀ a ∈ A. under r a ⊆ A)*

Note: In the definitions of *Above*[*S*] and *Under*[*S*], we bounded comprehension by *Field r* in order to properly cover the case of *A* being empty.

lemma *underS-subset-under*: *underS r a ⊆ under r a*
by (*auto simp add*: *underS-def under-def*)

lemma *underS-notIn*: *a ∉ underS r a*
by (*simp add*: *underS-def*)

lemma *Refl-under-in*: *Refl r ⟹ a ∈ Field r ⟹ a ∈ under r a*
by (*simp add*: *refl-on-def under-def*)

lemma *AboveS-disjoint*: *A ∩ (AboveS r A) = {}*
by (*auto simp add*: *AboveS-def*)

lemma *in-AboveS-underS*: *a ∈ Field r ⟹ a ∈ AboveS r (underS r a)*
by (*auto simp add*: *AboveS-def underS-def*)

lemma *Refl-under-underS*: *Refl r ⟹ a ∈ Field r ⟹ under r a = underS r a ∪ {a}*
unfolding *under-def underS-def*
using *refl-on-def*[*of - r*] **by** *fastforce*

lemma *underS-empty*: *a ∉ Field r ⟹ underS r a = {}*
by (*auto simp*: *Field-def underS-def*)

lemma *under-Field*: *under r a ⊆ Field r*
by (*auto simp*: *under-def Field-def*)

lemma *underS-Field*: *underS r a ⊆ Field r*
by (*auto simp*: *underS-def Field-def*)

lemma *underS-Field2*: *a ∈ Field r ⟹ underS r a ⊂ Field r*
using *underS-notIn underS-Field* **by** *fast*

lemma *underS-Field3*: *Field r* ≠ {} ⟹ *underS r a* ⊂ *Field r*
by (*cases a* ∈ *Field r*) (*auto simp*: *underS-Field2 underS-empty*)

lemma *AboveS-Field*: *AboveS r A* ⊆ *Field r*
by (*auto simp*: *AboveS-def Field-def*)

lemma *under-incr*:
assumes *trans r*
and (*a*, *b*) ∈ *r*
shows *under r a* ⊆ *under r b*
unfolding *under-def*
proof *safe*
fix *x* **assume** (*x*, *a*) ∈ *r*
with *assms trans-def*[*of r*] **show** (*x*, *b*) ∈ *r* **by** *blast*
qed

lemma *underS-incr*:
assumes *trans r*
and *antisym r*
and *ab*: (*a*, *b*) ∈ *r*
shows *underS r a* ⊆ *underS r b*
unfolding *underS-def*
proof *safe*
assume ∗: *b* ≠ *a* **and** ∗∗: (*b*, *a*) ∈ *r*
with ‹*antisym r*› *antisym-def*[*of r*] *ab* **show** *False*
by *blast*
next
fix *x* **assume** *x* ≠ *a* (*x*, *a*) ∈ *r*
with *ab* ‹*trans r*› *trans-def*[*of r*] **show** (*x*, *b*) ∈ *r*
by *blast*
qed

lemma *underS-incl-iff*:
assumes *LO*: *Linear-order r*
and *INa*: *a* ∈ *Field r*
and *INb*: *b* ∈ *Field r*
shows *underS r a* ⊆ *underS r b* ⟷ (*a*, *b*) ∈ *r*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
with ‹*Linear-order r*› **show** *?lhs*
by (*simp add*: *order-on-defs underS-incr*)
next
assume ∗: *?lhs*
have (*a*, *b*) ∈ *r* **if** *a* = *b*
using *assms that* **by** (*simp add*: *order-on-defs refl-on-def*)
moreover have *False* **if** *a* ≠ *b* (*b*, *a*) ∈ *r*
proof −

```
    from that have b ∈ underS r a unfolding underS-def by blast
    with ∗ have b ∈ underS r b by blast
    then show ?thesis by (simp add: underS-notIn)
  qed
  ultimately show (a,b) ∈ r
    using assms order-on-defs[of Field r r] total-on-def[of Field r r] by blast
qed

lemma finite-Partial-order-induct[consumes 3, case-names step]:
  assumes Partial-order r
    and x ∈ Field r
    and finite r
    and step: ⋀x. x ∈ Field r ⟹ (⋀y. y ∈ aboveS r x ⟹ P y) ⟹ P x
  shows P x
  using assms(2)
proof (induct rule: wf-induct[of r⁻¹ − Id])
  case 1
  from assms(1,3) show wf (r⁻¹ − Id)
    using partial-order-on-well-order-on partial-order-on-converse by blast
next
  case prems: (2 x)
  show ?case
    by (rule step) (use prems in ‹auto simp: aboveS-def intro: FieldI2›)
qed

lemma finite-Linear-order-induct[consumes 3, case-names step]:
  assumes Linear-order r
    and x ∈ Field r
    and finite r
    and step: ⋀x. x ∈ Field r ⟹ (⋀y. y ∈ aboveS r x ⟹ P y) ⟹ P x
  shows P x
  using assms(2)
proof (induct rule: wf-induct[of r⁻¹ − Id])
  case 1
  from assms(1,3) show wf (r⁻¹ − Id)
    using linear-order-on-well-order-on linear-order-on-converse
    unfolding well-order-on-def by blast
next
  case prems: (2 x)
  show ?case
    by (rule step) (use prems in ‹auto simp: aboveS-def intro: FieldI2›)
qed
```

24.6 Variations on Well-Founded Relations

This subsection contains some variations of the results from *HOL.Wellfounded*:

- means for slightly more direct definitions by well-founded recursion;
- variations of well-founded induction;

- means for proving a linear order to be a well-order.

24.6.1 Characterizations of well-foundedness

A transitive relation is well-founded iff it is "locally" well-founded, i.e., iff its restriction to the lower bounds of of any element is well-founded.

lemma *trans-wf-iff*:
assumes *trans r*
shows *wf r* $\longleftrightarrow$ $(\forall\, a.\ wf\ (r \cap (r^{-1}\text{``}\{a\} \times r^{-1}\text{``}\{a\})))$
proof −
define *R* **where** $R\ a = r \cap (r^{-1}\text{``}\{a\} \times r^{-1}\text{``}\{a\})$ **for** *a*
have *wf* (*R a*) **if** *wf r* **for** *a*
using *that R-def wf-subset*[*of r R a*] **by** *auto*
moreover
have *wf r* **if** $*$: $\forall\, a.\ wf(R\ a)$
unfolding *wf-def*
proof *clarify*
fix *phi a*
assume $**$: $\forall\, a.\ (\forall\, b.\ (b,\ a) \in r \longrightarrow phi\ b) \longrightarrow phi\ a$
define *chi* **where** *chi b* $\longleftrightarrow$ $(b,\ a) \in r \longrightarrow phi\ b$ **for** *b*
with $*$ **have** *wf* (*R a*) **by** *auto*
then have $(\forall\, b.\ (\forall\, c.\ (c,\ b) \in R\ a \longrightarrow chi\ c) \longrightarrow chi\ b) \longrightarrow (\forall\, b.\ chi\ b)$
unfolding *wf-def* **by** *blast*
also have $\forall\, b.\ (\forall\, c.\ (c,\ b) \in R\ a \longrightarrow chi\ c) \longrightarrow chi\ b$
proof *safe*
fix *b*
assume $\forall\, c.\ (c,\ b) \in R\ a \longrightarrow chi\ c$
moreover have $(b,\ a) \in r \Longrightarrow \forall\, c.\ (c,\ b) \in r \wedge (c,\ a) \in r \longrightarrow phi\ c \Longrightarrow phi\ b$
proof −
assume $(b,\ a) \in r$ **and** $\forall\, c.\ (c,\ b) \in r \wedge (c,\ a) \in r \longrightarrow phi\ c$
then have $\forall\, c.\ (c,\ b) \in r \longrightarrow phi\ c$
using *assms trans-def*[*of r*] **by** *blast*
with $**$ **show** *phi b* **by** *blast*
qed
ultimately show *chi b*
by (*auto simp add*: *chi-def R-def*)
qed
finally have $\forall\, b.\ chi\ b$ **.**
with $**$ *chi-def* **show** *phi a* **by** *blast*
qed
ultimately show *?thesis* **unfolding** *R-def* **by** *blast*
qed

A transitive relation is well-founded if all initial segments are finite.

corollary *wf-finite-segments*:
assumes *irrefl r* **and** *trans r* **and** $\bigwedge x.\ finite\ \{y.\ (y,\ x) \in r\}$

shows *wf r*
proof −
have $\bigwedge a$. *acyclic* $(r \cap \{x.\ (x,\ a) \in r\} \times \{x.\ (x,\ a) \in r\})$
proof −
fix *a*
have *trans* $(r \cap (\{x.\ (x,\ a) \in r\} \times \{x.\ (x,\ a) \in r\}))$
using *assms* **unfolding** *trans-def Field-def* **by** *blast*
then show *acyclic* $(r \cap \{x.\ (x,\ a) \in r\} \times \{x.\ (x,\ a) \in r\})$
using *assms acyclic-def assms irrefl-def* **by** *fastforce*
qed
then show *?thesis*
by (*clarsimp simp*: *trans-wf-iff wf-iff-acyclic-if-finite converse-def assms*)
qed

The next lemma is a variation of *wf-eq-minimal* from Wellfounded, allowing one to assume the set included in the field.

lemma *wf-eq-minimal2*: *wf r* $\longleftrightarrow$ $(\forall A.\ A \subseteq Field\ r \wedge A \neq \{\} \longrightarrow (\exists a \in A.\ \forall a' \in A.\ (a',\ a) \notin r))$
proof−
let *?phi* $= \lambda A.\ A \neq \{\} \longrightarrow (\exists a \in A.\ \forall a' \in A.\ (a',a) \notin r)$
have *wf r* $\longleftrightarrow$ $(\forall A.$ *?phi A*$)$
proof
assume *wf r*
show $\forall A.$ *?phi A*
proof *clarify*
fix *A*:: *'a set*
assume $A \neq \{\}$
then obtain *x* **where** $x \in A$
by *auto*
show $\exists a{\in}A.\ \forall a'{\in}A.\ (a',\ a) \notin r$
apply (*rule wfE-min*[*of r x A*])
apply *fact*+
by *blast*
qed
next
assume $*$: $\forall A.$ *?phi A*
then show *wf r*
apply (*clarsimp simp*: *ex-in-conv* [*THEN sym*])
apply (*rule wfI-min*)
by *fast*
qed
also have $(\forall A.$ *?phi A*$) \longleftrightarrow (\forall B \subseteq Field\ r.$ *?phi B*$)$
proof
assume $\forall A.$ *?phi A*
then show $\forall B \subseteq Field\ r.$ *?phi B* **by** *simp*
next
assume $*$: $\forall B \subseteq Field\ r.$ *?phi B*
show $\forall A.$ *?phi A*
proof *clarify*

```
    fix A :: 'a set
    assume **: A ≠ {}
    define B where B = A ∩ Field r
    show ∃ a ∈ A. ∀ a' ∈ A. (a', a) ∉ r
    proof (cases B = {})
      case True
      with * obtain a where a: a ∈ A a ∉ Field r
        unfolding B-def by blast
      with a have ∀ a' ∈ A. (a',a) ∉ r
        unfolding Field-def by blast
      with a show ?thesis by blast
    next
      case False
      have B ⊆ Field r unfolding B-def by blast
      with False * obtain a where a: a ∈ B ∀ a' ∈ B. (a', a) ∉ r
        by blast
      have (a', a) ∉ r if a' ∈ A for a'
      proof
        assume a'a: (a', a) ∈ r
        with that have a' ∈ B unfolding B-def Field-def by blast
        with a a'a show False by blast
      qed
      with a show ?thesis unfolding B-def by blast
    qed
  qed
  finally show ?thesis by blast
qed
```

24.6.2 Characterizations of well-foundedness

The next lemma and its corollary enable one to prove that a linear order is a well-order in a way which is more standard than via well-foundedness of the strict version of the relation.

```
lemma Linear-order-wf-diff-Id:
  assumes Linear-order r
  shows wf (r − Id) ⟷ (∀ A ⊆ Field r. A ≠ {} ⟶ (∃ a ∈ A. ∀ a' ∈ A. (a, a')
∈ r))
proof (cases r ⊆ Id)
  case True
  then have *: r − Id = {} by blast
  have wf (r − Id) by (simp add: *)
  moreover have ∃ a ∈ A. ∀ a' ∈ A. (a, a') ∈ r
    if *: A ⊆ Field r and **: A ≠ {} for A
  proof −
    from ‹Linear-order r› True
    obtain a where a: r = {} ∨ r = {(a, a)}
      unfolding order-on-defs using Total-subset-Id [of r] by blast
    with * ** have A = {a} ∧ r = {(a, a)}
```

```
    unfolding Field-def by blast
   with a show ?thesis by blast
  qed
  ultimately show ?thesis by blast
next
  case False
  with ‹Linear-order r› have Field: Field r = Field (r − Id)
   unfolding order-on-defs using Total-Id-Field [of r] by blast
  show ?thesis
  proof
   assume ∗: wf (r − Id)
   show ∀ A ⊆ Field r. A ≠ {} ⟶ (∃ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r)
   proof clarify
    fix A
    assume ∗∗: A ⊆ Field r and ∗∗∗: A ≠ {}
    then have ∃ a ∈ A. ∀ a′ ∈ A. (a′,a) ∉ r − Id
     using Field ∗ unfolding wf-eq-minimal2 by simp
    moreover have ∀ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r ⟷ (a′, a) ∉ r − Id
     using Linear-order-in-diff-Id [OF ‹Linear-order r›] ∗∗ by blast
    ultimately show ∃ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r by blast
   qed
  next
   assume ∗: ∀ A ⊆ Field r. A ≠ {} ⟶ (∃ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r)
   show wf (r − Id)
    unfolding wf-eq-minimal2
   proof clarify
    fix A
    assume ∗∗: A ⊆ Field(r − Id) and ∗∗∗: A ≠ {}
    then have ∃ a ∈ A. ∀ a′ ∈ A. (a,a′) ∈ r
     using Field ∗ by simp
    moreover have ∀ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r ⟷ (a′, a) ∉ r − Id
      using Linear-order-in-diff-Id [OF ‹Linear-order r›] ∗∗ mono-Field[of r −
Id r] by blast
    ultimately show ∃ a ∈ A. ∀ a′ ∈ A. (a′,a) ∉ r − Id
     by blast
   qed
  qed
qed

corollary Linear-order-Well-order-iff:
  Linear-order r ⟹
   Well-order r ⟷ (∀ A ⊆ Field r. A ≠ {} ⟶ (∃ a ∈ A. ∀ a′ ∈ A. (a, a′) ∈ r))
  unfolding well-order-on-def using Linear-order-wf-diff-Id[of r] by blast

end
```

25 Hilbert's Epsilon-Operator and the Axiom of Choice

theory *Hilbert-Choice*
imports *Wellfounded*
keywords *specification* :: *thy-goal-defn*
begin

25.1 Hilbert's epsilon

axiomatization *Eps* :: $('a \Rightarrow bool) \Rightarrow 'a$
where *someI*: $P\ x \Longrightarrow P\ (Eps\ P)$

syntax (*epsilon*)
-Eps :: *pttrn* ⇒ *bool* ⇒ *'a* (‹(‹*indent=3 notation=‹binder ϵ ›› ϵ -./ -*)› [*0*, *10*] *10*)
syntax (*input*)
-Eps :: *pttrn* ⇒ *bool* ⇒ *'a* (‹(‹*indent=3 notation=‹binder @›› @ -./ -*)› [*0*, *10*] *10*)
syntax
-Eps :: *pttrn* ⇒ *bool* ⇒ *'a* (‹(‹*indent=3 notation=‹binder SOME›› SOME -./ -*)› [*0*, *10*] *10*)

syntax-consts *-Eps* ⇌ *Eps*

translations
SOME x. P ⇌ *CONST Eps* (*λx. P*)

print-translation ‹
[(***const-syntax*** ‹*Eps*›, *fn ctxt => fn* [*Abs abs*] =>
let val (*x*, *t*) = *Syntax-Trans.atomic-abs-tr' ctxt abs*
in Syntax.const ***syntax-const*** ‹*-Eps*› $ *x* $ *t end*)]
› — to avoid eta-contraction of body

definition *inv-into* :: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ (*'b* ⇒ *'a*) **where**
inv-into A f = (*λx. SOME y. y* ∈ *A* ∧ *f y* = *x*)

lemma *inv-into-def2*: *inv-into A f x* = (*SOME y. y* ∈ *A* ∧ *f y* = *x*)
by(*simp add*: *inv-into-def*)

abbreviation *inv* :: (*'a* ⇒ *'b*) ⇒ (*'b* ⇒ *'a*) **where**
inv ≡ *inv-into UNIV*

25.2 Hilbert's Epsilon-operator

lemma *Eps-cong*:
assumes $\bigwedge x.\ P\ x = Q\ x$
shows *Eps P* = *Eps Q*
using *ext*[*of P Q*, *OF assms*] **by** *simp*

Easier to use than *someI* if the witness comes from an existential formula.

lemma *someI-ex* [*elim?*]: $\exists\, x.\ P\ x \Longrightarrow P\ (SOME\ x.\ P\ x)$
by (*elim exE someI*)

lemma *some-eq-imp*:
assumes *Eps P* = *a P b* **shows** *P a*
using *assms someI-ex* **by** *force*

Easier to use than *someI* because the conclusion has only one occurrence of *P*.

lemma *someI2*: $P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ P\ x)$
by (*blast intro*: *someI*)

Easier to use than *someI2* if the witness comes from an existential formula.

lemma *someI2-ex*: $\exists\, a.\ P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ P\ x)$
by (*blast intro*: *someI2*)

lemma *someI2-bex*: $\exists\, a{\in}A.\ P\ a \Longrightarrow (\bigwedge x.\ x \in A \wedge P\ x \Longrightarrow Q\ x) \Longrightarrow Q\ (SOME\ x.\ x \in A \wedge P\ x)$
by (*blast intro*: *someI2*)

lemma *some-equality* [*intro*]: $P\ a \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow x = a) \Longrightarrow (SOME\ x.\ P\ x) = a$
by (*blast intro*: *someI2*)

lemma *some1-equality*: $\exists!x.\ P\ x \Longrightarrow P\ a \Longrightarrow (SOME\ x.\ P\ x) = a$
by *blast*

lemma *some-eq-ex*: $P\ (SOME\ x.\ P\ x) \longleftrightarrow (\exists\, x.\ P\ x)$
by (*blast intro*: *someI*)

lemma *some-in-eq*: $(SOME\ x.\ x \in A) \in A \longleftrightarrow A \neq \{\}$
unfolding *ex-in-conv*[*symmetric*] **by** (*rule some-eq-ex*)

lemma *some-eq-trivial* [*simp*]: $(SOME\ y.\ y = x) = x$
by (*rule some-equality*) (*rule refl*)

lemma *some-sym-eq-trivial* [*simp*]: $(SOME\ y.\ x = y) = x$
by (*iprover intro*: *some-equality*)

25.3 Axiom of Choice, Proved Using the Description Operator

lemma *choice*: $\forall\, x.\ \exists\, y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall\, x.\ Q\ x\ (f\ x)$
by (*fast elim*: *someI*)

lemma *bchoice*: $\forall\, x{\in}S.\ \exists\, y.\ Q\ x\ y \Longrightarrow \exists f.\ \forall\, x{\in}S.\ Q\ x\ (f\ x)$
by (*fast elim*: *someI*)

lemma *choice-iff*: $(\forall\, x.\ \exists\, y.\ Q\ x\ y) \longleftrightarrow (\exists f.\ \forall\, x.\ Q\ x\ (f\ x))$

by (*fast elim*: *someI*)

lemma *choice-iff'*: $(\forall x.\ P\ x \longrightarrow (\exists y.\ Q\ x\ y)) \longleftrightarrow (\exists f.\ \forall x.\ P\ x \longrightarrow Q\ x\ (f\ x))$
by (*fast elim*: *someI*)

lemma *bchoice-iff*: $(\forall x{\in}S.\ \exists y.\ Q\ x\ y) \longleftrightarrow (\exists f.\ \forall x{\in}S.\ Q\ x\ (f\ x))$
by (*fast elim*: *someI*)

lemma *bchoice-iff'*: $(\forall x{\in}S.\ P\ x \longrightarrow (\exists y.\ Q\ x\ y)) \longleftrightarrow (\exists f.\ \forall x{\in}S.\ P\ x \longrightarrow Q\ x\ (f\ x))$
by (*fast elim*: *someI*)

lemma *dependent-nat-choice*:
assumes *1*: $\exists x.\ P\ 0\ x$
and *2*: $\bigwedge x\ n.\ P\ n\ x \Longrightarrow \exists y.\ P\ (Suc\ n)\ y \wedge Q\ n\ x\ y$
shows $\exists f.\ \forall n.\ P\ n\ (f\ n) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$
proof (*intro exI allI conjI*)
fix *n*
define *f* **where** $f = rec\text{-}nat\ (SOME\ x.\ P\ 0\ x)\ (\lambda n\ x.\ SOME\ y.\ P\ (Suc\ n)\ y \wedge Q\ n\ x\ y)$
then have $P\ 0\ (f\ 0)\ \bigwedge n.\ P\ n\ (f\ n) \Longrightarrow P\ (Suc\ n)\ (f\ (Suc\ n)) \wedge Q\ n\ (f\ n)\ (f\ (Suc\ n))$
using *someI-ex*[*OF 1*] *someI-ex*[*OF 2*] **by** *simp-all*
then show $P\ n\ (f\ n)\ Q\ n\ (f\ n)\ (f\ (Suc\ n))$
by (*induct n*) *auto*
qed

lemma *finite-subset-Union*:
assumes *finite A* $A \subseteq \bigcup \mathcal{B}$
obtains $\mathcal{F}$ **where** *finite* $\mathcal{F}$ $\mathcal{F} \subseteq \mathcal{B}$ $A \subseteq \bigcup \mathcal{F}$
proof −
have $\forall x{\in}A.\ \exists B{\in}\mathcal{B}.\ x{\in}B$
using *assms* **by** *blast*
then obtain *f* **where** *f*: $\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathcal{B} \wedge x \in f\ x$
by (*auto simp add*: *bchoice-iff Bex-def*)
show *thesis*
proof
show *finite* (*f* ‘ *A*)
using *assms* **by** *auto*
qed (*use f* **in** *auto*)
qed

25.4 Getting an element of a nonempty set

definition *some-elem* :: *'a set* ⇒ *'a*
where *some-elem A* = (*SOME x*. *x* ∈ *A*)

lemma *some-elem-eq* [*simp*]: *some-elem* {*x*} = *x*
by (*simp add*: *some-elem-def*)

lemma *some-elem-nonempty*: $A \neq \{\} \Longrightarrow$ *some-elem* $A \in A$
unfolding *some-elem-def* **by** (*auto intro*: *someI*)

lemma *is-singleton-some-elem*: *is-singleton* $A \longleftrightarrow A = \{$*some-elem* $A\}$
by (*auto simp*: *is-singleton-def*)

lemma *some-elem-image-unique*:
assumes $A \neq \{\}$
and $*$: $\bigwedge y.\ y \in A \Longrightarrow f\ y = a$
shows *some-elem* $(f\ `\ A) = a$
unfolding *some-elem-def*
proof (*rule some1-equality*)
from ‹$A \neq \{\}$› **obtain** y **where** $y \in A$ **by** *auto*
with $*$ ‹$y \in A$› **have** $a \in f\ `\ A$ **by** *blast*
then show $a \in f\ `\ A$ **by** *auto*
with $*$ **show** $\exists!x.\ x \in f\ `\ A$
by *auto*
qed

25.5 Function Inverse

lemma *inv-def*: *inv* $f = (\lambda y.$ *SOME* $x.\ f\ x = y)$
by (*simp add*: *inv-into-def*)

lemma *inv-into-into*: $x \in f\ `\ A \Longrightarrow$ *inv-into* $A\ f\ x \in A$
by (*simp add*: *inv-into-def*) (*fast intro*: *someI2*)

lemma *inv-identity* [*simp*]: *inv* $(\lambda a.\ a) = (\lambda a.\ a)$
by (*simp add*: *inv-def*)

lemma *inv-id* [*simp*]: *inv id* = *id*
by (*simp add*: *id-def*)

lemma *inv-into-f-f* [*simp*]: *inj-on* $f\ A \Longrightarrow x \in A \Longrightarrow$ *inv-into* $A\ f\ (f\ x) = x$
by (*simp add*: *inv-into-def inj-on-def*) (*blast intro*: *someI2*)

lemma *inv-f-f*: *inj* $f \Longrightarrow$ *inv* $f\ (f\ x) = x$
by *simp*

lemma *f-inv-into-f*: $y \in f`A \Longrightarrow f$ (*inv-into* $A\ f\ y) = y$
by (*simp add*: *inv-into-def*) (*fast intro*: *someI2*)

lemma *inv-into-f-eq*: *inj-on* $f\ A \Longrightarrow x \in A \Longrightarrow f\ x = y \Longrightarrow$ *inv-into* $A\ f\ y = x$
by (*erule subst*) (*fast intro*: *inv-into-f-f*)

lemma *inv-f-eq*: *inj* $f \Longrightarrow f\ x = y \Longrightarrow$ *inv* $f\ y = x$
by (*simp add*:*inv-into-f-eq*)

lemma *inj-imp-inv-eq*: *inj f* $\Longrightarrow$ $\forall x.$ *f* (*g x*) = *x* $\Longrightarrow$ *inv f* = *g*
by (*blast intro*: *inv-into-f-eq*)

But is it useful?

lemma *inj-transfer*:
assumes *inj*: *inj f*
and *minor*: $\bigwedge y.$ *y* $\in$ *range f* $\Longrightarrow$ *P* (*inv f y*)
shows *P x*
proof −
have *f x* $\in$ *range f* **by** *auto*
then have *P*(*inv f* (*f x*)) **by** (*rule minor*)
then show *P x* **by** (*simp add*: *inv-into-f-f* [*OF inj*])
qed

lemma *inj-iff*: *inj f* $\longleftrightarrow$ *inv f* ∘ *f* = *id*
by (*simp add*: *o-def fun-eq-iff*) (*blast intro*: *inj-on-inverseI inv-into-f-f*)

lemma *inv-o-cancel*[*simp*]: *inj f* $\Longrightarrow$ *inv f* ∘ *f* = *id*
by (*simp add*: *inj-iff*)

lemma *o-inv-o-cancel*[*simp*]: *inj f* $\Longrightarrow$ *g* ∘ *inv f* ∘ *f* = *g*
by (*simp add*: *comp-assoc*)

lemma *inv-into-image-cancel*[*simp*]: *inj-on f A* $\Longrightarrow$ *S* $\subseteq$ *A* $\Longrightarrow$ *inv-into A f* ‘ *f* ‘ *S* = *S*
by (*fastforce simp*: *image-def*)

lemma *inj-imp-surj-inv*: *inj f* $\Longrightarrow$ *surj* (*inv f*)
by (*blast intro*!: *surjI inv-into-f-f*)

lemma *surj-f-inv-f*: *surj f* $\Longrightarrow$ *f* (*inv f y*) = *y*
by (*simp add*: *f-inv-into-f*)

lemma *bij-inv-eq-iff*: *bij p* $\Longrightarrow$ *x* = *inv p y* $\longleftrightarrow$ *p x* = *y*
using *surj-f-inv-f*[*of p*] **by** (*auto simp add*: *bij-def*)

lemma *inv-into-injective*:
assumes *eq*: *inv-into A f x* = *inv-into A f y*
and *x*: *x* $\in$ *f*‘*A*
and *y*: *y* $\in$ *f*‘*A*
shows *x* = *y*
proof −
from *eq* **have** *f* (*inv-into A f x*) = *f* (*inv-into A f y*)
by *simp*
with *x y* **show** *?thesis*
by (*simp add*: *f-inv-into-f*)
qed

lemma *inj-on-inv-into*: *B* $\subseteq$ *f*‘*A* $\Longrightarrow$ *inj-on* (*inv-into A f*) *B*

by (*blast intro*: *inj-onI dest*: *inv-into-injective injD*)

lemma *inj-imp-bij-betw-inv*: *inj f* $\Longrightarrow$ *bij-betw* (*inv f*) (*f* ' *M*) *M*
by (*simp add*: *bij-betw-def image-subsetI inj-on-inv-into*)

lemma *bij-betw-inv-into*: *bij-betw f A B* $\Longrightarrow$ *bij-betw* (*inv-into A f*) *B A*
by (*auto simp add*: *bij-betw-def inj-on-inv-into*)

lemma *surj-imp-inj-inv*: *surj f* $\Longrightarrow$ *inj* (*inv f*)
by (*simp add*: *inj-on-inv-into*)

lemma *surj-iff*: *surj f* $\longleftrightarrow$ *f* $\circ$ *inv f* = *id*
by (*auto intro*!: *surjI simp*: *surj-f-inv-f fun-eq-iff*[**where** *'b*=*'a*])

lemma *surj-iff-all*: *surj f* $\longleftrightarrow$ ($\forall$ *x*. *f* (*inv f x*) = *x*)
by (*simp add*: *o-def surj-iff fun-eq-iff*)

lemma *surj-imp-inv-eq*:
assumes *surj f* **and** *gf*: $\bigwedge$*x*. *g* (*f x*) = *x*
shows *inv f* = *g*
proof (*rule ext*)
fix *x*
have *g* (*f* (*inv f x*)) = *inv f x*
by (*rule gf*)
then show *inv f x* = *g x*
by (*simp add*: *surj-f-inv-f* ‹*surj f*›)
qed

lemma *bij-imp-bij-inv*: *bij f* $\Longrightarrow$ *bij* (*inv f*)
by (*simp add*: *bij-def inj-imp-surj-inv surj-imp-inj-inv*)

lemma *inv-equality*: ($\bigwedge$*x*. *g* (*f x*) = *x*) $\Longrightarrow$ ($\bigwedge$*y*. *f* (*g y*) = *y*) $\Longrightarrow$ *inv f* = *g*
by (*rule ext*) (*auto simp add*: *inv-into-def*)

lemma *inv-inv-eq*: *bij f* $\Longrightarrow$ *inv* (*inv f*) = *f*
by (*rule inv-equality*) (*auto simp add*: *bij-def surj-f-inv-f*)

bij (*inv f*) implies little about *f*. Consider *f* :: *bool* $\Rightarrow$ *bool* such that *f True* = *f False* = *True*. Then it ia consistent with axiom *someI* that *inv f* could be any function at all, including the identity function. If *inv f* = *id* then *inv f* is a bijection, but *inj f*, *surj f* and *inv* (*inv f*) = *f* all fail.

lemma *inv-into-comp*:
inj-on f (*g* ' *A*) $\Longrightarrow$ *inj-on g A* $\Longrightarrow$ *x* $\in$ *f* ' *g* ' *A* $\Longrightarrow$
inv-into A (*f* $\circ$ *g*) *x* = (*inv-into A g* $\circ$ *inv-into* (*g* ' *A*) *f*) *x*
by (*auto simp*: *f-inv-into-f inv-into-into intro*: *inv-into-f-eq comp-inj-on*)

lemma *o-inv-distrib*: *bij f* $\Longrightarrow$ *bij g* $\Longrightarrow$ *inv* (*f* $\circ$ *g*) = *inv g* $\circ$ *inv f*
by (*rule inv-equality*) (*auto simp add*: *bij-def surj-f-inv-f*)

lemma *image-f-inv-f*: *surj f* $\Longrightarrow$ *f ‘ (inv f ‘ A) = A*
by (*simp add*: *surj-f-inv-f image-comp comp-def*)

lemma *image-inv-f-f*: *inj f* $\Longrightarrow$ *inv f ‘ (f ‘ A) = A*
by *simp*

lemma *bij-image-Collect-eq*:
assumes *bij f*
shows *f ‘ Collect P = {y. P (inv f y)}*
proof
show *f ‘ Collect P ⊆ {y. P (inv f y)}*
using *assms* **by** (*force simp add*: *bij-is-inj*)
show *{y. P (inv f y)} ⊆ f ‘ Collect P*
using *assms* **by** (*blast intro*: *bij-is-surj* [*THEN surj-f-inv-f*, *symmetric*])
qed

lemma *bij-vimage-eq-inv-image*:
assumes *bij f*
shows *f −‘ A = inv f ‘ A*
proof
show *f −‘ A ⊆ inv f ‘ A*
using *assms* **by** (*blast intro*: *bij-is-inj* [*THEN inv-into-f-f*, *symmetric*])
show *inv f ‘ A ⊆ f −‘ A*
using *assms* **by** (*auto simp add*: *bij-is-surj* [*THEN surj-f-inv-f*])
qed

lemma *inv-fn-o-fn-is-id*:
fixes *f*::*'a* ⇒ *'a*
assumes *bij f*
shows *((inv f)^^n) o (f^^n) = (λx. x)*
proof −
have *((inv f)^^n)((f^^n) x) = x* **for** *x n*
proof (*induction n*)
case (*Suc n*)
have ∗: *(inv f) (f y) = y* **for** *y*
by (*simp add*: *assms bij-is-inj*)
have *(inv f ^^ Suc n) ((f ^^ Suc n) x) = (inv f^^n) (inv f (f ((f^^n) x)))*
by (*simp add*: *funpow-swap1*)
also have *... = (inv f^^n) ((f^^n) x)*
using ∗ **by** *auto*
also have *... = x* **using** *Suc.IH* **by** *auto*
finally show *?case* **by** *simp*
qed (*auto*)
then show *?thesis* **unfolding** *o-def* **by** *blast*
qed

lemma *fn-o-inv-fn-is-id*:
fixes *f*::*'a* ⇒ *'a*
assumes *bij f*

```
  shows (f^^n) o ((inv f)^^n) = (λx. x)
proof −
  have (f^^n) (((inv f)^^n) x) = x for x n
  proof (induction n)
    case (Suc n)
    have ∗: f(inv f y) = y for y
      using bij-inv-eq-iff[OF assms] by auto
    have (f ^^ Suc n) ((inv f ^^ Suc n) x) = (f^^n) (f (inv f ((inv f^^n) x)))
      by (simp add: funpow-swap1)
    also have ... = (f^^n) ((inv f^^n) x)
      using ∗ by auto
    also have ... = x using Suc.IH by auto
    finally show ?case by simp
  qed (auto)
  then show ?thesis unfolding o-def by blast
qed

lemma inv-fn:
  fixes f::'a ⇒ 'a
  assumes bij f
  shows inv (f^^n) = ((inv f)^^n)
proof −
  have inv (f^^n) x = ((inv f)^^n) x for x
  proof (rule inv-into-f-eq)
    show inj (f ^^ n)
      by (simp add: inj-fn[OF bij-is-inj [OF assms]])
    show (f ^^ n) ((inv f ^^ n) x) = x
      using fn-o-inv-fn-is-id[OF assms, THEN fun-cong] by force
  qed auto
  then show ?thesis by auto
qed

lemma funpow-inj-finite:
  assumes ‹inj p› ‹finite {y. ∃ n. y = (p ^^ n) x}›
  obtains n where ‹n > 0› ‹(p ^^ n) x = x›
proof −
  have ‹infinite (UNIV :: nat set)›
    by simp
  moreover have ‹{y. ∃ n. y = (p ^^ n) x} = (λn. (p ^^ n) x) ‘ UNIV›
    by auto
  with assms have ‹finite ...›
    by simp
  ultimately have ∃ n ∈ UNIV. ¬ finite {m ∈ UNIV. (p ^^ m) x = (p ^^ n) x}
    by (rule pigeonhole-infinite)
  then obtain n where infinite {m. (p ^^ m) x = (p ^^ n) x} by auto
  then have infinite ({m. (p ^^ m) x = (p ^^ n) x} − {n}) by auto
  then have ({m. (p ^^ m) x = (p ^^ n) x} − {n}) ≠ {}
    by (auto simp add: subset-singleton-iff)
  then obtain m where m: (p ^^ m) x = (p ^^ n) x m ≠ n by auto
```

```
  { fix m n assume (p ^^ n) x = (p ^^ m) x m < n
    have (p ^^ (n − m)) x = inv (p ^^ m) ((p ^^ m) ((p ^^ (n − m)) x))
      using ‹inj p› by (simp add: inv-f-f)
    also have ((p ^^ m) ((p ^^ (n − m)) x)) = (p ^^ n) x
      using ‹m < n› funpow-add [of m ‹n − m› p] by simp
    also have inv (p ^^ m) ... = x
      using ‹inj p›  by (simp add: ‹(p ^^ n) x = -›)
    finally have (p ^^ (n − m)) x = x 0 < n − m
      using ‹m < n› by auto }
  note general = this

  show thesis
  proof (cases m n rule: linorder-cases)
    case less
    then have ‹n − m > 0› ‹(p ^^ (n − m)) x = x›
      using general [of n m] m by simp-all
    then show thesis by (blast intro: that)
  next
    case equal
    then show thesis using m by simp
  next
    case greater
    then have ‹m − n > 0› ‹(p ^^ (m − n)) x = x›
      using general [of m n] m by simp-all
    then show thesis by (blast intro: that)
  qed
qed


lemma mono-inv:
  fixes f::'a::linorder ⇒ 'b::order
  assumes mono f bij f
  shows mono (inv f)
proof
  fix x y::'b assume x ≤ y
  from ‹bij f› obtain a b where x: x = f a and y: y = f b by(fastforce simp:
bij-def surj-def)
  show inv f x ≤ inv f y
  proof (rule le-cases)
    assume a ≤ b
    thus ?thesis using  ‹bij f› x y by(simp add: bij-def inv-f-f)
  next
    assume b ≤ a
    hence f b ≤ f a by(rule monoD[OF ‹mono f›])
    hence y ≤ x using x y by simp
    hence x = y using ‹x ≤ y› by auto
    thus ?thesis by simp
  qed
```

qed

lemma *strict-mono-inv-on-range*:
fixes $f :: 'a::linorder \Rightarrow 'b::preorder$
assumes *strict-mono-on UNIV f*
shows *strict-mono-on* (*range f*) (*inv f*)
proof (*clarsimp simp*: *strict-mono-on-def*)
fix *x y*
assume $f\ x < f\ y$
then show $inv\ f\ (f\ x) < inv\ f\ (f\ y)$
using *assms strict-mono-on-imp-inj-on strict-mono-less* **by** *fastforce*
qed

lemma *mono-bij-Inf*:
fixes $f :: 'a::complete\text{-}linorder \Rightarrow 'b::complete\text{-}linorder$
assumes *mono f bij f*
shows $f\ (Inf\ A) = Inf\ (f`A)$
proof −
have *surj f* **using** ‹*bij f*› **by** (*auto simp*: *bij-betw-def*)
have ∗: $(inv\ f)\ (Inf\ (f`A)) \leq Inf\ ((inv\ f)`(f`A))$
using *mono-Inf*[*OF mono-inv*[*OF assms*], *of f`A*] **by** *simp*
have $Inf\ (f`A) \leq f\ (Inf\ ((inv\ f)`(f`A)))$
using *monoD*[*OF* ‹*mono f*› ∗] **by**(*simp add*: *surj-f-inv-f*[*OF* ‹*surj f*›])
also have $... = f(Inf\ A)$
using *assms* **by** (*simp add*: *bij-is-inj*)
finally show *?thesis* **using** *mono-Inf*[*OF assms*(*1*), *of A*] **by** *auto*
qed

lemma *finite-fun-UNIVD1*:
assumes *fin*: *finite* (*UNIV* :: $('a \Rightarrow 'b)$ *set*)
and *card*: *card* (*UNIV* :: *'b set*) ≠ *Suc 0*
shows *finite* (*UNIV* :: *'a set*)
proof −
let *?UNIV-b* = *UNIV* :: *'b set*
from *fin* **have** *finite ?UNIV-b*
by (*rule finite-fun-UNIVD2*)
with *card* **have** *card ?UNIV-b* ≥ *Suc* (*Suc 0*)
by (*cases card ?UNIV-b*) (*auto simp*: *card-eq-0-iff*)
then have *card ?UNIV-b* = *Suc* (*Suc* (*card ?UNIV-b* − *Suc* (*Suc 0*)))
by *simp*
then obtain *b1 b2* :: *'b* **where** *b1b2*: *b1* ≠ *b2*
by (*auto simp*: *card-Suc-eq*)
from *fin* **have** *fin'*: *finite* (*range* $(\lambda f :: 'a \Rightarrow 'b.\ inv\ f\ b1)$)
by (*rule finite-imageI*)
have *UNIV* = *range* $(\lambda f :: 'a \Rightarrow 'b.\ inv\ f\ b1)$
proof (*rule UNIV-eq-I*)
fix *x* :: *'a*
from *b1b2* **have** $x = inv\ (\lambda y.\ if\ y = x\ then\ b1\ else\ b2)\ b1$
by (*simp add*: *inv-into-def*)

```
    then show x ∈ range (λf::'a ⇒ 'b. inv f b1)
      by blast
  qed
  with fin' show ?thesis
    by simp
qed
```

Every infinite set contains a countable subset. More precisely we show that a set S is infinite if and only if there exists an injective function from the naturals into S.

The "only if" direction is harder because it requires the construction of a sequence of pairwise different elements of an infinite set S. The idea is to construct a sequence of non-empty and infinite subsets of S obtained by successively removing elements of S.

```
lemma infinite-countable-subset:
  assumes inf: ¬ finite S
  shows ∃f::nat ⇒ 'a. inj f ∧ range f ⊆ S
  — Courtesy of Stephan Merz
proof −
  define Sseq where Sseq = rec-nat S (λn T. T − {SOME e. e ∈ T})
  define pick where pick n = (SOME e. e ∈ Sseq n) for n
  have ∗: Sseq n ⊆ S ¬ finite (Sseq n) for n
    by (induct n) (auto simp: Sseq-def inf)
  then have ∗∗: ⋀n. pick n ∈ Sseq n
    unfolding pick-def by (subst (asm) finite.simps) (auto simp add: ex-in-conv
intro: someI-ex)
  with ∗ have range pick ⊆ S by auto
  moreover have pick n ≠ pick (n + Suc m) for m n
  proof −
    have pick n ∉ Sseq (n + Suc m)
      by (induct m) (auto simp add: Sseq-def pick-def)
    with ∗∗ show ?thesis by auto
  qed
  then have inj pick
    by (intro linorder-injI) (auto simp add: less-iff-Suc-add)
  ultimately show ?thesis by blast
qed

lemma infinite-iff-countable-subset: ¬ finite S ⟷ (∃f::nat ⇒ 'a. inj f ∧ range f
⊆ S)
  — Courtesy of Stephan Merz
  using finite-imageD finite-subset infinite-UNIV-char-0 infinite-countable-subset
by auto

lemma image-inv-into-cancel:
  assumes surj: f‘A = A'
    and sub: B' ⊆ A'
  shows f ‘((inv-into A f)‘B') = B'
```

```
  using assms
proof (auto simp: f-inv-into-f)
  let ?f' = inv-into A f
  fix a'
  assume ∗: a' ∈ B'
  with sub have a' ∈ A' by auto
  with surj have a' = f (?f' a')
    by (auto simp: f-inv-into-f)
  with ∗ show a' ∈ f ‘ (?f' ‘ B') by blast
qed

lemma inv-into-inv-into-eq:
  assumes bij-betw f A A'
    and a: a ∈ A
  shows inv-into A' (inv-into A f) a = f a
proof −
  let ?f' = inv-into A f
  let ?f'' = inv-into A' ?f'
  from assms have ∗: bij-betw ?f' A' A
    by (auto simp: bij-betw-inv-into)
  with a obtain a' where a': a' ∈ A' ?f' a' = a
    unfolding bij-betw-def by force
  with a ∗ have ?f'' a = a'
    by (auto simp: f-inv-into-f bij-betw-def)
  moreover from assms a' have f a = a'
    by (auto simp: bij-betw-def)
  ultimately show ?f'' a = f a by simp
qed

lemma inj-on-iff-surj:
  assumes A ≠ {}
  shows (∃ f. inj-on f A ∧ f ‘ A ⊆ A') ⟷ (∃ g. g ‘ A' = A)
proof safe
  fix f
  assume inj: inj-on f A and incl: f ‘ A ⊆ A'
  let ?phi = λa' a. a ∈ A ∧ f a = a'
  let ?csi = λa. a ∈ A
  let ?g = λa'. if a' ∈ f ‘ A then (SOME a. ?phi a' a) else (SOME a. ?csi a)
  have ?g ‘ A' = A
  proof
    show ?g ‘ A' ⊆ A
    proof clarify
      fix a'
      assume ∗: a' ∈ A'
      show ?g a' ∈ A
      proof (cases a' ∈ f ‘ A)
        case True
        then obtain a where ?phi a' a by blast
        then have ?phi a' (SOME a. ?phi a' a)
```

```
        using someI[of ?phi a' a] by blast
      with True show ?thesis by auto
    next
      case False
      with assms have ?csi (SOME a. ?csi a)
        using someI-ex[of ?csi] by blast
      with False show ?thesis by auto
    qed
  qed
 next
  show A ⊆ ?g ` A'
  proof −
    have ?g (f a) = a ∧ f a ∈ A' if a: a ∈ A for a
    proof −
      let ?b = SOME aa. ?phi (f a) aa
      from a have ?phi (f a) a by auto
      then have ∗: ?phi (f a) ?b
        using someI[of ?phi(f a) a] by blast
      then have ?g (f a) = ?b using a by auto
      moreover from inj ∗ a have a = ?b
        by (auto simp add: inj-on-def)
      ultimately have ?g(f a) = a by simp
      with incl a show ?thesis by auto
    qed
    then show ?thesis by force
  qed
 qed
 then show ∃ g. g ` A' = A by blast
next
 fix g
 let ?f = inv-into A' g
 have inj-on ?f (g ` A')
  by (auto simp: inj-on-inv-into)
 moreover have ?f (g a') ∈ A' if a': a' ∈ A' for a'
 proof −
  let ?phi = λ b'. b' ∈ A' ∧ g b' = g a'
  from a' have ?phi a' by auto
  then have ?phi (SOME b'. ?phi b')
    using someI[of ?phi] by blast
  then show ?thesis by (auto simp: inv-into-def)
 qed
 ultimately show ∃f. inj-on f (g ` A') ∧ f ` g ` A' ⊆ A'
  by auto
qed

lemma Ex-inj-on-UNION-Sigma:
 ∃f. (inj-on f (⋃ i ∈ I. A i) ∧ f ` (⋃ i ∈ I. A i) ⊆ (SIGMA i : I. A i))
proof
 let ?phi = λa i. i ∈ I ∧ a ∈ A i
```

let *?sm* = $\lambda a.$ *SOME i. ?phi a i*
let *?f* = $\lambda a.$ (*?sm a*, *a*)
have *inj-on ?f* ($\bigcup i \in I.\ A\ i$)
by (*auto simp*: *inj-on-def*)
moreover
have *?sm a* $\in I \wedge a \in A$(*?sm a*) **if** $i \in I$ **and** $a \in A\ i$ **for** *i a*
using *that someI*[*of ?phi a i*] **by** *auto*
then have *?f* ‘ ($\bigcup i \in I.\ A\ i$) $\subseteq$ (*SIGMA i* : *I*. *A i*)
by *auto*
ultimately show *inj-on ?f* ($\bigcup i \in I.\ A\ i$) $\wedge$ *?f* ‘ ($\bigcup i \in I.\ A\ i$) $\subseteq$ (*SIGMA i* : *I*. *A i*)
by *auto*
qed

lemma *inv-unique-comp*:
assumes *fg*: $f \circ g = id$
and *gf*: $g \circ f = id$
shows *inv f* = *g*
using *fg gf inv-equality*[*of g f*] **by** (*auto simp add*: *fun-eq-iff*)

lemma *subset-image-inj*:
$S \subseteq f$ ‘ $T \longleftrightarrow (\exists U.\ U \subseteq T \wedge$ *inj-on f U* $\wedge\ S = f$ ‘ $U)$
proof *safe*
show $\exists U \subseteq T.$ *inj-on f U* $\wedge\ S = f$ ‘ *U*
if $S \subseteq f$ ‘ *T*
proof −
from *that* [*unfolded subset-image-iff subset-iff*]
obtain *g* **where** *g*: $\bigwedge x.\ x \in S \Longrightarrow g\ x \in T \wedge x = f\ (g\ x)$
by (*auto simp add*: *image-iff Bex-def choice-iff'*)
show *?thesis*
proof (*intro exI conjI*)
show *g* ‘ $S \subseteq T$
by (*simp add*: *g image-subsetI*)
show *inj-on f* (*g* ‘ *S*)
using *g* **by** (*auto simp*: *inj-on-def*)
show $S = f$ ‘ (*g* ‘ *S*)
using *g image-subset-iff* **by** *auto*
qed
qed
qed *blast*

25.6 Other Consequences of Hilbert's Epsilon

Hilbert's Epsilon and the *split* Operator

Looping simprule!

lemma *split-paired-Eps*: (*SOME x. P x*) = (*SOME* (*a*, *b*). *P* (*a*, *b*))
by *simp*

lemma *Eps-case-prod*: *Eps* (*case-prod P*) = (*SOME xy*. *P* (*fst xy*) (*snd xy*))
by (*simp add*: *split-def*)

lemma *Eps-case-prod-eq* [*simp*]: (*SOME* (*x'*, *y'*). *x* = *x'* ∧ *y* = *y'*) = (*x*, *y*)
by *blast*

A relation is wellfounded iff it has no infinite descending chain.

lemma *wf-iff-no-infinite-down-chain*: *wf r* ⟷ (∄*f*. ∀ *i*. (*f* (*Suc i*), *f i*) ∈ *r*)
(**is** - ⟷ ¬ *?ex*)
proof
assume *wf r*
show ¬ *?ex*
proof
assume *?ex*
then obtain *f* **where** *f*: (*f* (*Suc i*), *f i*) ∈ *r* **for** *i*
by *blast*
from ‹*wf r*› **have** *minimal*: *x* ∈ *Q* ⟹ ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *r* ⟶ *y* ∉ *Q* **for** *x* *Q*
by (*auto simp*: *wf-eq-minimal*)
let *?Q* = {*w*. ∃ *i*. *w* = *f i*}
fix *n*
have *f n* ∈ *?Q* **by** *blast*
from *minimal* [*OF this*] **obtain** *j* **where** (*y*, *f j*) ∈ *r* ⟹ *y* ∉ *?Q* **for** *y* **by** *blast*
with *this* [*OF* ‹(*f* (*Suc j*), *f j*) ∈ *r*›] **have** *f* (*Suc j*) ∉ *?Q* **by** *simp*
then show *False* **by** *blast*
qed
next
assume ¬ *?ex*
then show *wf r*
proof (*rule contrapos-np*)
assume ¬ *wf r*
then obtain *Q x* **where** *x*: *x* ∈ *Q* **and** *rec*: *z* ∈ *Q* ⟹ ∃ *y*. (*y*, *z*) ∈ *r* ∧ *y* ∈ *Q* **for** *z*
by (*auto simp add*: *wf-eq-minimal*)
obtain *descend* :: *nat* ⇒ *'a*
where *descend-0*: *descend 0* = *x*
and *descend-Suc*: *descend* (*Suc n*) = (*SOME y*. *y* ∈ *Q* ∧ (*y*, *descend n*) ∈ *r*) **for** *n*
by (*rule that* [*of rec-nat x* (λ- *rec*. (*SOME y*. *y* ∈ *Q* ∧ (*y*, *rec*) ∈ *r*))]) *simp-all*
have *descend-Q*: *descend n* ∈ *Q* **for** *n*
proof (*induct n*)
case *0*
with *x* **show** *?case* **by** (*simp only*: *descend-0*)
next
case *Suc*
then show *?case* **by** (*simp only*: *descend-Suc*) (*rule someI2-ex*; *use rec* **in** *blast*)
qed

```
    have (descend (Suc i), descend i) ∈ r for i
      by (simp only: descend-Suc) (rule someI2-ex; use descend-Q rec in blast)
    then show ∃f. ∀i. (f (Suc i), f i) ∈ r by blast
  qed
qed

lemma wf-no-infinite-down-chainE:
  assumes wf r
  obtains k where (f (Suc k), f k) ∉ r
  using assms wf-iff-no-infinite-down-chain[of r] by blast

corollary no-infinite-less-nat-down-chain:
  fixes f :: nat ⇒ nat
  shows (⋀i. f i > f (Suc i)) ⟹ False
using wf-iff-no-infinite-down-chain[of measure f] in-measure[of - - f] wf-measure[of
f] by blast
```

A dynamically-scoped fact for TFL

```
lemma tfl-some: ∀P x. P x ⟶ P (Eps P)
  by (blast intro: someI)
```

25.7 An aside: bounded accessible part

Finite monotone eventually stable sequences

```
lemma finite-mono-remains-stable-implies-strict-prefix:
  fixes f :: nat ⇒ 'a::order
  assumes S: finite (range f) mono f
    and eq: ∀n. f n = f (Suc n) ⟶ f (Suc n) = f (Suc (Suc n))
  shows ∃N. (∀n≤N. ∀m≤N. m < n ⟶ f m < f n) ∧ (∀n≥N. f N = f n)
  using assms
proof −
  have ∃n. f n = f (Suc n)
  proof (rule ccontr)
    assume ¬ ?thesis
    then have ⋀n. f n ≠ f (Suc n) by auto
    with ‹mono f› have ⋀n. f n < f (Suc n)
      by (auto simp: le-less mono-iff-le-Suc)
    with lift-Suc-mono-less-iff[of f] have ∗: ⋀n m. n < m ⟹ f n < f m
      by auto
    have inj f
    proof (intro injI)
      fix x y
      assume f x = f y
      then show x = y
        by (cases x y rule: linorder-cases) (auto dest: ∗)
    qed
    with ‹finite (range f)› have finite (UNIV::nat set)
      by (rule finite-imageD)
    then show False by simp
```

```
  qed
  then obtain n where n: f n = f (Suc n) ..
  define N where N = (LEAST n. f n = f (Suc n))
  have N: f N = f (Suc N)
    unfolding N-def using n by (rule LeastI)
  show ?thesis
  proof (intro exI[of - N] conjI allI impI)
    fix n
    assume N ≤ n
    then have ⋀m. N ≤ m ⟹ m ≤ n ⟹ f m = f N
    proof (induct rule: dec-induct)
      case base
      then show ?case by simp
    next
      case (step n)
      then show ?case
        using eq [rule-format, of n − 1] N
        by (cases n) (auto simp add: le-Suc-eq)
    qed
    from this[of n] ‹N ≤ n› show f N = f n by auto
  next
    fix n m :: nat
    assume m < n n ≤ N
    then show f m < f n
    proof (induct rule: less-Suc-induct)
      case (1 i)
      then have i < N by simp
      then have f i ≠ f (Suc i)
        unfolding N-def by (rule not-less-Least)
      with ‹mono f› show ?case by (simp add: mono-iff-le-Suc less-le)
    next
      case 2
      then show ?case by simp
    qed
  qed
qed

lemma finite-mono-strict-prefix-implies-finite-fixpoint:
  fixes f :: nat ⇒ 'a set
  assumes S: ⋀i. f i ⊆ S finite S
    and ex: ∃ N. (∀ n≤N. ∀ m≤N. m < n ⟶ f m ⊂ f n) ∧ (∀ n≥N. f N = f n)
  shows f (card S) = (⋃ n. f n)
proof −
  from ex obtain N where inj: ⋀n m. n ≤ N ⟹ m ≤ N ⟹ m < n ⟹ f m
⊂ f n
    and eq: ∀ n≥N. f N = f n
    by atomize auto
  have i ≤ N ⟹ i ≤ card (f i) for i
  proof (induct i)
```

```
    case 0
    then show ?case by simp
  next
    case (Suc i)
    with inj [of Suc i i] have (f i) ⊂ (f (Suc i)) by auto
    moreover have finite (f (Suc i)) using S by (rule finite-subset)
    ultimately have card (f i) < card (f (Suc i)) by (intro psubset-card-mono)
    with Suc inj show ?case by auto
  qed
  then have N ≤ card (f N) by simp
  also have ... ≤ card S using S by (intro card-mono)
  finally have §: f (card S) = f N using eq by auto
  moreover have ⋃ (range f) ⊆ f N
  proof clarify
    fix x n
    assume x ∈ f n
    with eq inj [of N] show x ∈ f N
      by (cases n < N) (auto simp: not-less)
  qed
  ultimately show ?thesis
    by auto
qed
```

25.8 More on injections, bijections, and inverses

```
locale bijection =
  fixes f :: 'a ⇒ 'a
  assumes bij: bij f
begin

lemma bij-inv: bij (inv f)
  using bij by (rule bij-imp-bij-inv)

lemma surj [simp]: surj f
  using bij by (rule bij-is-surj)

lemma inj: inj f
  using bij by (rule bij-is-inj)

lemma surj-inv [simp]: surj (inv f)
  using inj by (rule inj-imp-surj-inv)

lemma inj-inv: inj (inv f)
  using surj by (rule surj-imp-inj-inv)

lemma eqI: f a = f b ⟹ a = b
  using inj by (rule injD)

lemma eq-iff [simp]: f a = f b ⟷ a = b
```

by (*auto intro*: *eqI*)

lemma *eq-invI*: *inv f a* = *inv f b* ⟹ *a* = *b*
using *inj-inv* **by** (*rule injD*)

lemma *eq-inv-iff* [*simp*]: *inv f a* = *inv f b* ⟷ *a* = *b*
by (*auto intro*: *eq-invI*)

lemma *inv-left* [*simp*]: *inv f* (*f a*) = *a*
using *inj* **by** (*simp add*: *inv-f-eq*)

lemma *inv-comp-left* [*simp*]: *inv f* ∘ *f* = *id*
by (*simp add*: *fun-eq-iff*)

lemma *inv-right* [*simp*]: *f* (*inv f a*) = *a*
using *surj* **by** (*simp add*: *surj-f-inv-f*)

lemma *inv-comp-right* [*simp*]: *f* ∘ *inv f* = *id*
by (*simp add*: *fun-eq-iff*)

lemma *inv-left-eq-iff* [*simp*]: *inv f a* = *b* ⟷ *f b* = *a*
by *auto*

lemma *inv-right-eq-iff* [*simp*]: *b* = *inv f a* ⟷ *f b* = *a*
by *auto*

end

lemma *infinite-imp-bij-betw*:
assumes *infinite*: ¬ *finite A*
shows ∃ *h*. *bij-betw h A* (*A* − {*a*})
proof (*cases a* ∈ *A*)
case *False*
then have *A* − {*a*} = *A* **by** *blast*
then show *?thesis*
using *bij-betw-id*[*of A*] **by** *auto*
next
case *True*
with *infinite* **have** ¬ *finite* (*A* − {*a*}) **by** *auto*
with *infinite-iff-countable-subset*[*of A* − {*a*}]
obtain *f* :: *nat* ⇒ *'a* **where** *inj f* **and** *f*: *f* ' *UNIV* ⊆ *A* − {*a*} **by** *blast*
define *g* **where** *g n* = (*if n* = *0 then a else f* (*Suc n*)) **for** *n*
define *A'* **where** *A'* = *g* ' *UNIV*
have ∗: ∀ *y*. *f y* ≠ *a* **using** *f* **by** *blast*
have *3*: *inj-on g UNIV* ∧ *g* ' *UNIV* ⊆ *A* ∧ *a* ∈ *g* ' *UNIV*
using ‹*inj f*› *f* ∗ **unfolding** *inj-on-def g-def*
by (*auto simp add*: *True image-subset-iff*)
then have *4*: *bij-betw g UNIV A'* ∧ *a* ∈ *A'* ∧ *A'* ⊆ *A*
using *inj-on-imp-bij-betw*[*of g*] **by** (*auto simp*: *A'-def*)

then have *5*: *bij-betw* (*inv g*) *A′ UNIV*
by (*auto simp add*: *bij-betw-inv-into*)
from *3* **obtain** *n* **where** *n*: *g n* = *a* **by** *auto*
have *6*: *bij-betw g* (*UNIV* − {*n*}) (*A′* − {*a*})
by (*rule bij-betw-subset*) (*use 3 4 n* **in** ‹*auto simp*: *image-set-diff A′-def*›)
define *v* **where** *v m* = (*if m* < *n then m else Suc m*) **for** *m*
have *m* < *n* ∨ *m* = *n* **if** ⋀*k*. *k* < *n* ∨ *m* ≠ *Suc k* **for** *m*
using *that* [*of m−1*] **by** *auto*
then have *7*: *bij-betw v UNIV* (*UNIV* − {*n*})
unfolding *bij-betw-def inj-on-def v-def* **by** *auto*
define *h′* **where** *h′* = *g* ∘ *v* ∘ (*inv g*)
with *5 6 7* **have** *8*: *bij-betw h′ A′* (*A′* − {*a*})
by (*auto simp add*: *bij-betw-trans*)
define *h* **where** *h b* = (*if b* ∈ *A′ then h′ b else b*) **for** *b*
with *8* **have** *bij-betw h A′* (*A′* − {*a*})
using *bij-betw-cong*[*of A′ h*] **by** *auto*
moreover
have ∀ *b* ∈ *A* − *A′*. *h b* = *b* **by** (*auto simp*: *h-def*)
then have *bij-betw h* (*A* − *A′*) (*A* − *A′*)
using *bij-betw-cong*[*of A* − *A′ h id*] *bij-betw-id*[*of A* − *A′*] **by** *auto*
moreover
from *4* **have** (*A′* ∩ (*A* − *A′*) = {} ∧ *A′* ∪ (*A* − *A′*) = *A*) ∧
((*A′* − {*a*}) ∩ (*A* − *A′*) = {} ∧ (*A′* − {*a*}) ∪ (*A* − *A′*) = *A* − {*a*})
by *blast*
ultimately have *bij-betw h A* (*A* − {*a*})
using *bij-betw-combine*[*of h A′ A′* − {*a*} *A* − *A′ A* − *A′*] **by** *simp*
then show *?thesis* **by** *blast*
qed

lemma *infinite-imp-bij-betw2*:
assumes ¬ *finite A*
shows ∃ *h*. *bij-betw h A* (*A* ∪ {*a*})
proof (*cases a* ∈ *A*)
case *True*
then have *A* ∪ {*a*} = *A* **by** *blast*
then show *?thesis* **using** *bij-betw-id*[*of A*] **by** *auto*
next
case *False*
let *?A′* = *A* ∪ {*a*}
from *False* **have** *A* = *?A′* − {*a*} **by** *blast*
moreover from *assms* **have** ¬ *finite ?A′* **by** *auto*
ultimately obtain *f* **where** *bij-betw f ?A′ A*
using *infinite-imp-bij-betw*[*of ?A′ a*] **by** *auto*
then have *bij-betw* (*inv-into ?A′ f*) *A ?A′* **by** (*rule bij-betw-inv-into*)
then show *?thesis* **by** *auto*
qed

lemma *bij-betw-inv-into-left*: *bij-betw f A A′* ⟹ *a* ∈ *A* ⟹ *inv-into A f* (*f a*) = *a*
unfolding *bij-betw-def* **by** *clarify* (*rule inv-into-f-f*)

lemma *bij-betw-inv-into-right*: *bij-betw f A A′* $\Longrightarrow$ *a′* $\in$ *A′* $\Longrightarrow$ *f* (*inv-into A f a′*) = *a′*
unfolding *bij-betw-def* **using** *f-inv-into-f* **by** *force*

lemma *bij-betw-inv-into-subset*:
bij-betw f A A′ $\Longrightarrow$ *B* $\subseteq$ *A* $\Longrightarrow$ *f ‘ B* = *B′* $\Longrightarrow$ *bij-betw* (*inv-into A f*) *B′ B*
by (*auto simp*: *bij-betw-def intro*: *inj-on-inv-into*)

25.9 Specification package – Hilbertized version

lemma *exE-some*: *Ex P* $\Longrightarrow$ *c* $\equiv$ *Eps P* $\Longrightarrow$ *P c*
by (*simp only*: *someI-ex*)

ML-file ‹*Tools/choice-specification.ML*›

25.10 Complete Distributive Lattices – Properties depending on Hilbert Choice

context *complete-distrib-lattice*
begin

lemma *Sup-Inf*: $\bigsqcup$ (*Inf ‘ A*) = $\bigsqcap$ (*Sup ‘* {*f ‘ A* |*f*. $\forall$ *B*$\in$*A*. *f B* $\in$ *B*})
proof (*rule order.antisym*)
show $\bigsqcup$ (*Inf ‘ A*) $\leq$ $\bigsqcap$ (*Sup ‘* {*f ‘ A* |*f*. $\forall$ *B*$\in$*A*. *f B* $\in$ *B*})
using *Inf-lower2 Sup-upper*
by (*fastforce simp add*: *intro*: *Sup-least INF-greatest*)
next
show $\bigsqcap$ (*Sup ‘* {*f ‘ A* |*f*. $\forall$ *B*$\in$*A*. *f B* $\in$ *B*}) $\leq$ $\bigsqcup$ (*Inf ‘ A*)
proof (*simp add*: *Inf-Sup*, *rule SUP-least*, *simp*, *safe*)
fix *f*
assume $\forall$ *Y*. ($\exists$*f*. *Y* = *f ‘ A* $\wedge$ ($\forall$ *Y*$\in$*A*. *f Y* $\in$ *Y*)) $\longrightarrow$ *f Y* $\in$ *Y*
then have *B*: $\bigwedge$ *F* . ($\forall$ *Y* $\in$ *A* . *F Y* $\in$ *Y*) $\Longrightarrow$ $\exists$ *Z* $\in$ *A* . *f* (*F ‘ A*) = *F Z*
by *auto*
show $\bigsqcap$(*f ‘* {*f ‘ A* |*f*. $\forall$ *Y*$\in$*A*. *f Y* $\in$ *Y*}) $\leq$ $\bigsqcup$(*Inf ‘ A*)
proof (*cases* $\exists$ *Z* $\in$ *A* . $\bigsqcap$(*f ‘* {*f ‘ A* |*f*. $\forall$ *Y*$\in$*A*. *f Y* $\in$ *Y*}) $\leq$ *Inf Z*)
case *True*
from *this* **obtain** *Z* **where** [*simp*]: *Z* $\in$ *A* **and** *A*: $\bigsqcap$(*f ‘* {*f ‘ A* |*f*. $\forall$ *Y*$\in$*A*. *f Y* $\in$ *Y*}) $\leq$ *Inf Z*
by *blast*
have *B*: ... $\leq$ $\bigsqcup$(*Inf ‘ A*)
by (*simp add*: *SUP-upper*)
from *A* **and** *B* **show** *?thesis*
by *simp*
next
case *False*
then have *X*: $\bigwedge$ *Z* . *Z* $\in$ *A* $\Longrightarrow$ $\exists$ *x* . *x* $\in$ *Z* $\wedge$ $\neg$ $\bigsqcap$(*f ‘* {*f ‘ A* |*f*. $\forall$ *Y*$\in$*A*. *f Y* $\in$ *Y*}) $\leq$ *x*
using *Inf-greatest* **by** *blast*

define F **where** $F = (\lambda\ Z\ .\ SOME\ x\ .\ x \in Z \wedge \neg \bigsqcap (f\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\}) \leq x)$
have C: $\bigwedge Y.\ Y \in A \Longrightarrow F\ Y \in Y$
using X **by** (*simp add*: *F-def*, *rule someI2-ex*, *auto*)
have E: $\bigwedge Y.\ Y \in A \Longrightarrow \neg \bigsqcap (f\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\}) \leq F\ Y$
using X **by** (*simp add*: *F-def*, *rule someI2-ex*, *auto*)
from C **and** B **obtain** Z **where** D: $Z \in A$ **and** Y: $f\ (F\ `\ A) = F\ Z$
by *blast*
from E **and** D **have** W: $\neg \bigsqcap (f\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\}) \leq F\ Z$
by *simp*
have $\bigsqcap (f\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\}) \leq f\ (F\ `\ A)$
using C **by** (*blast intro*: *INF-lower*)
with $W\ Y$ **show** *?thesis*
by *simp*
qed
qed
qed

lemma *dual-complete-distrib-lattice*:
class.complete-distrib-lattice Sup Inf sup $(\geq)$ $(>)$ *inf* $\top$ $\bot$
by (*simp add*: *class.complete-distrib-lattice.intro* [*OF dual-complete-lattice*]
class.complete-distrib-lattice-axioms-def Sup-Inf)

lemma *sup-Inf*: $a \sqcup \bigsqcap B = \bigsqcap ((\sqcup)\ a\ `\ B)$
proof (*rule order.antisym*)
show $a \sqcup \bigsqcap B \leq \bigsqcap ((\sqcup)\ a\ `\ B)$
using *Inf-lower sup.mono* **by** (*fastforce intro*: *INF-greatest*)
next
have $\bigsqcap ((\sqcup)\ a\ `\ B) \leq \bigsqcap (Sup\ `\ \{\{f\ \{a\}, f\ B\}\ |f.\ f\ \{a\} = a \wedge f\ B \in B\})$
by (*rule INF-greatest*, *auto simp add*: *INF-lower*)
also have $... = \bigsqcup (Inf\ `\ \{\{a\},\ B\})$
by (*unfold Sup-Inf*, *simp*)
finally show $\bigsqcap ((\sqcup)\ a\ `\ B) \leq a \sqcup \bigsqcap B$
by *simp*
qed

lemma *inf-Sup*: $a \sqcap \bigsqcup B = \bigsqcup ((\sqcap)\ a\ `\ B)$
using *dual-complete-distrib-lattice*
by (*rule complete-distrib-lattice.sup-Inf*)

lemma *INF-SUP*: $(\bigsqcap y.\ \bigsqcup x.\ P\ x\ y) = (\bigsqcup f.\ \bigsqcap x.\ P\ (f\ x)\ x)$
proof (*rule order.antisym*)
show $(SUP\ x.\ INF\ y.\ P\ (x\ y)\ y) \leq (INF\ y.\ SUP\ x.\ P\ x\ y)$
by (*rule SUP-least*, *rule INF-greatest*, *rule SUP-upper2*, *simp-all*, *rule INF-lower2*, *simp*, *blast*)
next
have $(INF\ y.\ SUP\ x.\ ((P\ x\ y))) \leq Inf\ (Sup\ `\ \{\{P\ x\ y\ |\ x\ .\ True\}\ |\ y\ .\ True\ \})$ (**is** *?A* $\leq$ *?B*)
proof (*rule INF-greatest*, *clarsimp*)

```
    fix y
    have ?A ≤ (SUP x. P x y)
      by (rule INF-lower, simp)
    also have ... ≤ Sup {uu. ∃x. uu = P x y}
      by (simp add: full-SetCompr-eq)
    finally show ?A ≤ Sup {uu. ∃x. uu = P x y}
      by simp
  qed
  also have ... ≤  (SUP x. INF y. P (x y) y)
  proof (subst Inf-Sup, rule SUP-least, clarsimp)
    fix f
    assume A: ∀Y. (∃y. Y = {uu. ∃x. uu = P x y}) ⟶ f Y ∈ Y

    have ⨅(f ` {uu. ∃y. uu = {uu. ∃x. uu = P x y}}) ≤
      (⨅y. P (SOME x. f {P x y |x. True} = P x y) y)
    proof (rule INF-greatest, clarsimp)
      fix y
        have (INF x∈{uu. ∃y. uu = {uu. ∃x. uu = P x y}}. f x) ≤ f {uu. ∃x. uu
= P x y}
          by (rule INF-lower, blast)
        also have ... ≤ P (SOME x. f {uu . ∃x. uu = P x y} = P x y) y
          by (rule someI2-ex) (use A in auto)
        finally show ⨅(f ` {uu. ∃y. uu = {uu. ∃x. uu = P x y}}) ≤
          P (SOME x. f {uu. ∃x. uu = P x y} = P x y) y
          by simp
      qed
      also have ... ≤ (SUP x. INF y. P (x y) y)
        by (rule SUP-upper, simp)
      finally show ⨅(f ` {uu. ∃y. uu = {uu. ∃x. uu = P x y}}) ≤ (⨆x. ⨅y. P
(x y) y)
        by simp
    qed
  finally show (INF y. SUP x. P x y) ≤ (SUP x. INF y. P (x y) y)
    by simp
qed

lemma INF-SUP-set: (⨅B∈A. ⨆(g ` B)) = (⨆B∈{f ` A |f. ∀C∈A. f C ∈ C}.
⨅(g ` B))
                  (is - = (⨆B∈?F. -))
proof (rule order.antisym)
  have ⨅ ((g ∘ f) ` A) ≤ ⨆ (g ` B) if ⋀B. B ∈ A ⟹ f B ∈ B B ∈ A for f B
    using that by (auto intro: SUP-upper2 INF-lower2)
  then show (⨆x∈?F. ⨅a∈x. g a) ≤ (⨅x∈A. ⨆a∈x. g a)
    by (auto intro!: SUP-least INF-greatest simp add: image-comp)
next
  show (⨅x∈A. ⨆a∈x. g a) ≤ (⨆x∈?F. ⨅a∈x. g a)
  proof (cases {} ∈ A)
    case True
    then show ?thesis
```

```
    by (rule INF-lower2) simp-all
  next
    case False
    {fix x
     have (⨅x∈A. ⨆x∈x. g x) ≤ (⨆u. if x ∈ A then if u ∈ x then g u else ⊥ else
⊤)
      proof (cases x ∈ A)
        case True
        then show ?thesis
          by (intro INF-lower2 SUP-least SUP-upper2) auto
      qed auto
    }
    then have (⨅Y∈A. ⨆a∈Y. g a) ≤ (⨅Y. ⨆y. if Y ∈ A then if y ∈ Y then g
y else ⊥ else ⊤)
      by (rule INF-greatest)
    also have ... = (⨆x. ⨅Y. if Y ∈ A then if x Y ∈ Y then g (x Y) else ⊥ else
⊤)
      by (simp only: INF-SUP)
    also have ... ≤ (⨆x∈?F. ⨅a∈x. g a)
    proof (rule SUP-least)
      show (⨅B. if B ∈ A then if x B ∈ B then g (x B) else ⊥ else ⊤)
              ≤ (⨆x∈?F. ⨅x∈x. g x) for x
      proof −
        define G where G ≡ λY. if x Y ∈ Y then x Y else (SOME x. x ∈Y)
        have ∀ Y∈A. G Y ∈ Y
          using False some-in-eq G-def by auto
        then have A: G ‘ A ∈ ?F
          by blast
          show (⨅Y. if Y ∈ A then if x Y ∈ Y then g (x Y) else ⊥ else ⊤) ≤
(⨆x∈?F. ⨅x∈x. g x)
       by (fastforce simp: G-def intro: SUP-upper2 [OF A] INF-greatest INF-lower2)
      qed
    qed
    finally show ?thesis by simp
  qed
qed

lemma SUP-INF: (⨆y. ⨅x. P x y) = (⨅x. ⨆y. P (x y) y)
  using dual-complete-distrib-lattice
  by (rule complete-distrib-lattice.INF-SUP)

lemma SUP-INF-set: (⨆x∈A. ⨅ (g ‘ x)) = (⨅x∈{f ‘ A |f. ∀ Y∈A. f Y ∈ Y}.
⨆ (g ‘ x))
  using dual-complete-distrib-lattice
  by (rule complete-distrib-lattice.INF-SUP-set)

end
```

context *complete-distrib-lattice*
begin

lemma *sup-INF*: $a \sqcup (\bigsqcap b{\in}B.\ f\ b) = (\bigsqcap b{\in}B.\ a \sqcup f\ b)$
by (*simp add*: *sup-Inf image-comp*)

lemma *inf-SUP*: $a \sqcap (\bigsqcup b{\in}B.\ f\ b) = (\bigsqcup b{\in}B.\ a \sqcap f\ b)$
by (*simp add*: *inf-Sup image-comp*)

lemma *Inf-sup*: $\bigsqcap B \sqcup a = (\bigsqcap b{\in}B.\ b \sqcup a)$
by (*simp add*: *sup-Inf sup-commute*)

lemma *Sup-inf*: $\bigsqcup B \sqcap a = (\bigsqcup b{\in}B.\ b \sqcap a)$
by (*simp add*: *inf-Sup inf-commute*)

lemma *INF-sup*: $(\bigsqcap b{\in}B.\ f\ b) \sqcup a = (\bigsqcap b{\in}B.\ f\ b \sqcup a)$
by (*simp add*: *sup-INF sup-commute*)

lemma *SUP-inf*: $(\bigsqcup b{\in}B.\ f\ b) \sqcap a = (\bigsqcup b{\in}B.\ f\ b \sqcap a)$
by (*simp add*: *inf-SUP inf-commute*)

lemma *Inf-sup-eq-top-iff*: $(\bigsqcap B \sqcup a = \top) \longleftrightarrow (\forall\, b{\in}B.\ b \sqcup a = \top)$
by (*simp only*: *Inf-sup INF-top-conv*)

lemma *Sup-inf-eq-bot-iff*: $(\bigsqcup B \sqcap a = \bot) \longleftrightarrow (\forall\, b{\in}B.\ b \sqcap a = \bot)$
by (*simp only*: *Sup-inf SUP-bot-conv*)

lemma *INF-sup-distrib2*: $(\bigsqcap a{\in}A.\ f\ a) \sqcup (\bigsqcap b{\in}B.\ g\ b) = (\bigsqcap a{\in}A.\ \bigsqcap b{\in}B.\ f\ a \sqcup g\ b)$
by (*subst INF-commute*) (*simp add*: *sup-INF INF-sup*)

lemma *SUP-inf-distrib2*: $(\bigsqcup a{\in}A.\ f\ a) \sqcap (\bigsqcup b{\in}B.\ g\ b) = (\bigsqcup a{\in}A.\ \bigsqcup b{\in}B.\ f\ a \sqcap g\ b)$
by (*subst SUP-commute*) (*simp add*: *inf-SUP SUP-inf*)

end

instantiation *set* :: (*type*) *complete-distrib-lattice*
begin
instance proof (*standard*, *clarsimp*)
fix A :: (('a *set*) *set*) *set*
fix x::'a
assume A: $\forall\, \mathcal{S}{\in}A.\ \exists\, X{\in}\mathcal{S}.\ x \in X$
define F **where** $F \equiv \lambda Y.\ SOME\ X.\ Y \in A \wedge X \in Y \wedge x \in X$
have $(\forall\, S \in F\ `\ A.\ x \in S)$
using A **unfolding** *F-def* **by** (*fastforce intro*: *someI2-ex*)
moreover have $\forall\, Y{\in}A.\ F\ Y \in Y$
using A **unfolding** *F-def* **by** (*fastforce intro*: *someI2-ex*)
then have $\exists f.\ F\ `\ A\ = f\ `\ A \wedge (\forall\, Y{\in}A.\ f\ Y \in Y)$

```
    by blast
  ultimately show ∃X. (∃f. X = f ‘ A ∧ (∀Y∈A. f Y ∈ Y)) ∧ (∀S∈X. x ∈ S)
    by auto
qed
end

instance set :: (type) complete-boolean-algebra ..

instantiation fun :: (type, complete-distrib-lattice) complete-distrib-lattice
begin
instance by standard (simp add: le-fun-def INF-SUP-set image-comp)
end

instance fun :: (type, complete-boolean-algebra) complete-boolean-algebra ..

context complete-linorder
begin

subclass complete-distrib-lattice
proof (standard, rule ccontr)
  fix A :: 'a set set
  let ?F = {f ‘ A |f. ∀Y∈A. f Y ∈ Y}
  assume ¬ ⨅(Sup ‘ A) ≤ ⨆(Inf ‘ ?F)
  then have C: ⨅(Sup ‘ A) > ⨆(Inf ‘ ?F)
    by (simp add: not-le)
  show False
  proof (cases ∃ z . ⨅(Sup ‘ A) > z ∧ z > ⨆(Inf ‘ ?F))
    case True
    then obtain z where A: z < ⨅(Sup ‘ A) and X: z > ⨆(Inf ‘ ?F)
      by blast
    then have B: ⋀Y. Y ∈ A ⟹ ∃k∈Y . z < k
      using local.less-Sup-iff by(force dest: less-INF-D)

    define G where G ≡ λY. SOME k . k ∈ Y ∧ z < k
    have E: ⋀Y. Y ∈ A ⟹ G Y ∈ Y
      using B unfolding G-def by (fastforce intro: someI2-ex)
    have z ≤ Inf (G ‘ A)
    proof (rule INF-greatest)
      show ⋀Y. Y ∈ A ⟹ z ≤ G Y
        using B unfolding G-def by (fastforce intro: someI2-ex)
    qed
    also have ... ≤ ⨆(Inf ‘ ?F)
      by (rule SUP-upper) (use E in blast)
    finally have z ≤ ⨆(Inf ‘ ?F)
      by simp

    with X show ?thesis
      using local.not-less by blast
  next
```

case *False*
have *B*: $\bigwedge Y.\ Y \in A \Longrightarrow \exists\ k \in Y\ .\ \bigsqcup(Inf\ `\ ?F) < k$
using *C local.less-Sup-iff* **by**(*force dest*: *less-INF-D*)
define *G* **where** $G \equiv \lambda\ Y\ .\ SOME\ k\ .\ k \in\ Y \wedge \bigsqcup(Inf\ `\ ?F) < k$
have *E*: $\bigwedge Y.\ Y \in A \Longrightarrow G\ Y \in Y$
using *B* **unfolding** *G-def* **by** (*fastforce intro*: *someI2-ex*)
have $\bigwedge Y.\ Y \in A \Longrightarrow \bigsqcap(Sup\ `\ A) \leq G\ Y$
using *B False local.leI* **unfolding** *G-def* **by** (*fastforce intro*: *someI2-ex*)
then have $\bigsqcap(Sup\ `\ A) \leq Inf\ (G\ `\ A)$
by (*simp add*: *local.INF-greatest*)
also have $Inf\ (G\ `\ A) \leq \bigsqcup(Inf\ `\ ?F)$
by (*rule SUP-upper*) (*use E* **in** *blast*)
finally have $\bigsqcap(Sup\ `\ A) \leq \bigsqcup(Inf\ `\ ?F)$
by *simp*
with *C* **show** *?thesis*
using *not-less* **by** *blast*
qed
qed
end

end

26 Zorn's Lemma and the Well-ordering Theorem

theory *Zorn*
imports *Order-Relation Hilbert-Choice*
begin

26.1 Zorn's Lemma for the Subset Relation

26.1.1 Results that do not require an order

Let P be a binary predicate on the set A.

locale *pred-on* =
fixes $A :: 'a\ set$
and $P :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹⊏› *50*)
begin

abbreviation $Peq :: 'a \Rightarrow 'a \Rightarrow bool$ (**infix** ‹⊑› *50*)
where $x \sqsubseteq y \equiv P^{==}\ x\ y$

A chain is a totally ordered subset of A.

definition $chain :: 'a\ set \Rightarrow bool$
where $chain\ C \longleftrightarrow C \subseteq A \wedge (\forall x{\in}C.\ \forall y{\in}C.\ x \sqsubseteq y \vee y \sqsubseteq x)$

We call a chain that is a proper superset of some set X, but not necessarily a chain itself, a superchain of X.

abbreviation $superchain :: 'a\ set \Rightarrow 'a\ set \Rightarrow bool$ (**infix** ‹<c› *50*)

where $X <c C \equiv chain\ C \wedge X \subset C$

A maximal chain is a chain that does not have a superchain.

definition *maxchain* :: *'a set* ⇒ *bool*
where $maxchain\ C \longleftrightarrow chain\ C \wedge (\nexists S.\ C <c S)$

We define the successor of a set to be an arbitrary superchain, if such exists, or the set itself, otherwise.

definition *suc* :: *'a set* ⇒ *'a set*
where $suc\ C = (if\ \neg\ chain\ C \vee maxchain\ C\ then\ C\ else\ (SOME\ D.\ C <c D))$

lemma *chainI* [*Pure.intro?*]: $C \subseteq A \Longrightarrow (\bigwedge x\ y.\ x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x) \Longrightarrow chain\ C$
unfolding *chain-def* **by** *blast*

lemma *chain-total*: $chain\ C \Longrightarrow x \in C \Longrightarrow y \in C \Longrightarrow x \sqsubseteq y \vee y \sqsubseteq x$
by (*simp add*: *chain-def*)

lemma *not-chain-suc* [*simp*]: $\neg\ chain\ X \Longrightarrow suc\ X = X$
by (*simp add*: *suc-def*)

lemma *maxchain-suc* [*simp*]: $maxchain\ X \Longrightarrow suc\ X = X$
by (*simp add*: *suc-def*)

lemma *suc-subset*: $X \subseteq suc\ X$
by (*auto simp*: *suc-def maxchain-def intro*: *someI2*)

lemma *chain-empty* [*simp*]: *chain* {}
by (*auto simp*: *chain-def*)

lemma *not-maxchain-Some*: $chain\ C \Longrightarrow \neg\ maxchain\ C \Longrightarrow C <c (SOME\ D.\ C <c D)$
by (*rule someI-ex*) (*auto simp*: *maxchain-def*)

lemma *suc-not-equals*: $chain\ C \Longrightarrow \neg\ maxchain\ C \Longrightarrow suc\ C \neq C$
using *not-maxchain-Some* **by** (*auto simp*: *suc-def*)

lemma *subset-suc*:
assumes $X \subseteq Y$
shows $X \subseteq suc\ Y$
using *assms* **by** (*rule subset-trans*) (*rule suc-subset*)

We build a set $\mathcal{C}$ that is closed under applications of *suc* and contains the union of all its subsets.

inductive-set *suc-Union-closed* (‹$\mathcal{C}$›)
where
suc: $X \in \mathcal{C} \Longrightarrow suc\ X \in \mathcal{C}$
| *Union* [*unfolded Pow-iff*]: $X \in Pow\ \mathcal{C} \Longrightarrow \bigcup X \in \mathcal{C}$

Since the empty set as well as the set itself is a subset of every set, $\mathcal{C}$ contains at least $\{\} \in \mathcal{C}$ and $\bigcup\ \mathcal{C} \in \mathcal{C}$.

lemma *suc-Union-closed-empty*: $\{\} \in \mathcal{C}$
  and *suc-Union-closed-Union*: $\bigcup\mathcal{C} \in \mathcal{C}$
  using *Union* [*of* $\{\}$] **and** *Union* [*of* $\mathcal{C}$] **by** *simp-all*

Thus closure under *suc* will hit a maximal chain eventually, as is shown below.

lemma *suc-Union-closed-induct* [*consumes 1*, *case-names suc Union*, *induct pred*: *suc-Union-closed*]:
  assumes $X \in \mathcal{C}$
    and $\bigwedge X.\ X \in \mathcal{C} \Longrightarrow Q\ X \Longrightarrow Q\ (\mathit{suc}\ X)$
    and $\bigwedge X.\ X \subseteq \mathcal{C} \Longrightarrow \forall x{\in}X.\ Q\ x \Longrightarrow Q\ (\bigcup X)$
  shows $Q\ X$
  using *assms* **by** *induct blast+*

lemma *suc-Union-closed-cases* [*consumes 1*, *case-names suc Union*, *cases pred*: *suc-Union-closed*]:
  assumes $X \in \mathcal{C}$
    and $\bigwedge Y.\ X = \mathit{suc}\ Y \Longrightarrow Y \in \mathcal{C} \Longrightarrow Q$
    and $\bigwedge Y.\ X = \bigcup Y \Longrightarrow Y \subseteq \mathcal{C} \Longrightarrow Q$
  shows Q
  using *assms* **by** *cases simp-all*

On chains, *suc* yields a chain.

lemma *chain-suc*:
  assumes *chain X*
  shows *chain* (*suc X*)
  using *assms*
 by (*cases* $\neg$ *chain X* $\vee$ *maxchain X*) (*force simp*: *suc-def dest*: *not-maxchain-Some*)+

lemma *chain-sucD*:
  assumes *chain X*
  shows *suc X* $\subseteq$ *A* $\wedge$ *chain* (*suc X*)
proof −
  from ‹*chain X*› **have** ∗: *chain* (*suc X*)
    by (*rule chain-suc*)
  then have *suc X* $\subseteq$ *A*
    unfolding *chain-def* **by** *blast*
  with ∗ **show** *?thesis* **by** *blast*
qed

lemma *suc-Union-closed-total′*:
  assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
    and ∗: $\bigwedge Z.\ Z \in \mathcal{C} \Longrightarrow Z \subseteq Y \Longrightarrow Z = Y \vee \mathit{suc}\ Z \subseteq Y$
  shows $X \subseteq Y \vee \mathit{suc}\ Y \subseteq X$
  using ‹$X \in \mathcal{C}$›
proof *induct*

```
  case (suc X)
  with ∗ show ?case by (blast del: subsetI intro: subset-suc)
next
  case Union
  then show ?case by blast
qed

lemma suc-Union-closed-subsetD:
  assumes Y ⊆ X and X ∈ 𝒞 and Y ∈ 𝒞
  shows X = Y ∨ suc Y ⊆ X
  using assms(2,3,1)
proof (induct arbitrary: Y)
  case (suc X)
  note ∗ = ‹⋀Y. Y ∈ 𝒞 ⟹ Y ⊆ X ⟹ X = Y ∨ suc Y ⊆ X›
  with suc-Union-closed-total' [OF ‹Y ∈ 𝒞› ‹X ∈ 𝒞›]
  have Y ⊆ X ∨ suc X ⊆ Y by blast
  then show ?case
  proof
    assume Y ⊆ X
    with ∗ and ‹Y ∈ 𝒞› subset-suc show ?thesis
      by fastforce
  next
    assume suc X ⊆ Y
    with ‹Y ⊆ suc X› show ?thesis by blast
  qed
next
  case (Union X)
  show ?case
  proof (rule ccontr)
    assume ¬ ?thesis
    with ‹Y ⊆ ⋃X› obtain x y z
      where ¬ suc Y ⊆ ⋃X
        and x ∈ X and y ∈ x and y ∉ Y
        and z ∈ suc Y and ∀x∈X. z ∉ x by blast
    with ‹X ⊆ 𝒞› have x ∈ 𝒞 by blast
    from Union and ‹x ∈ X› have ∗: ⋀y. y ∈ 𝒞 ⟹ y ⊆ x ⟹ x = y ∨ suc y ⊆
x
      by blast
    with suc-Union-closed-total' [OF ‹Y ∈ 𝒞› ‹x ∈ 𝒞›] have Y ⊆ x ∨ suc x ⊆ Y
      by blast
    then show False
    proof
      assume Y ⊆ x
      with ∗ [OF ‹Y ∈ 𝒞›] ‹y ∈ x› ‹y ∉ Y› ‹x ∈ X› ‹¬ suc Y ⊆ ⋃X› show False
        by blast
    next
      assume suc x ⊆ Y
      with ‹y ∉ Y› suc-subset ‹y ∈ x› show False by blast
    qed
```

qed
qed

The elements of $\mathcal{C}$ are totally ordered by the subset relation.

lemma *suc-Union-closed-total*:
assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$
shows $X \subseteq Y \vee Y \subseteq X$
proof (*cases* $\forall Z{\in}\mathcal{C}.\ Z \subseteq Y \longrightarrow Z = Y \vee \mathit{suc}\ Z \subseteq Y$)
case *True*
with *suc-Union-closed-total'* [*OF assms*]
have $X \subseteq Y \vee \mathit{suc}\ Y \subseteq X$ **by** *blast*
with *suc-subset* [*of* Y] **show** *?thesis* **by** *blast*
next
case *False*
then obtain Z **where** $Z \in \mathcal{C}$ **and** $Z \subseteq Y$ **and** $Z \neq Y$ **and** $\neg\ \mathit{suc}\ Z \subseteq Y$
by *blast*
with *suc-Union-closed-subsetD* **and** ‹$Y \in \mathcal{C}$› **show** *?thesis*
by *blast*
qed

Once we hit a fixed point w.r.t. *suc*, all other elements of $\mathcal{C}$ are subsets of this fixed point.

lemma *suc-Union-closed-suc*:
assumes $X \in \mathcal{C}$ **and** $Y \in \mathcal{C}$ **and** $\mathit{suc}\ Y = Y$
shows $X \subseteq Y$
using ‹$X \in \mathcal{C}$›
proof *induct*
case (*suc* X)
with ‹$Y \in \mathcal{C}$› **and** *suc-Union-closed-subsetD* **have** $X = Y \vee \mathit{suc}\ X \subseteq Y$
by *blast*
then show *?case*
by (*auto simp*: ‹$\mathit{suc}\ Y = Y$›)
next
case *Union*
then show *?case* **by** *blast*
qed

lemma *eq-suc-Union*:
assumes $X \in \mathcal{C}$
shows $\mathit{suc}\ X = X \longleftrightarrow X = \bigcup \mathcal{C}$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have $\bigcup \mathcal{C} \subseteq X$
by (*rule suc-Union-closed-suc* [*OF suc-Union-closed-Union* ‹$X \in \mathcal{C}$›])
with ‹$X \in \mathcal{C}$› **show** *?rhs*
by *blast*
next
from ‹$X \in \mathcal{C}$› **have** $\mathit{suc}\ X \in \mathcal{C}$ **by** (*rule suc*)

then have *suc X* $\subseteq \bigcup \mathcal{C}$ **by** *blast*
moreover assume *?rhs*
ultimately have *suc X* $\subseteq X$ **by** *simp*
moreover have $X \subseteq$ *suc X* **by** (*rule suc-subset*)
ultimately show *?lhs* **..**
qed

lemma *suc-in-carrier*:
assumes $X \subseteq A$
shows *suc X* $\subseteq A$
using *assms*
by (*cases* $\neg$ *chain X* $\vee$ *maxchain X*) (*auto dest*: *chain-sucD*)

lemma *suc-Union-closed-in-carrier*:
assumes $X \in \mathcal{C}$
shows $X \subseteq A$
using *assms*
by *induct* (*auto dest*: *suc-in-carrier*)

All elements of $\mathcal{C}$ are chains.

lemma *suc-Union-closed-chain*:
assumes $X \in \mathcal{C}$
shows *chain X*
using *assms*
proof *induct*
case (*suc X*)
then show *?case*
using *not-maxchain-Some* **by** (*simp add*: *suc-def*)
next
case (*Union X*)
then have $\bigcup X \subseteq A$
by (*auto dest*: *suc-Union-closed-in-carrier*)
moreover have $\forall x{\in}\bigcup X.\ \forall y{\in}\bigcup X.\ x \sqsubseteq y \vee y \sqsubseteq x$
proof (*intro ballI*)
fix *x y*
assume $x \in \bigcup X$ **and** $y \in \bigcup X$
then obtain *u v* **where** $x \in u$ **and** $u \in X$ **and** $y \in v$ **and** $v \in X$
by *blast*
with *Union* **have** $u \in \mathcal{C}$ **and** $v \in \mathcal{C}$ **and** *chain u* **and** *chain v*
by *blast+*
with *suc-Union-closed-total* **have** $u \subseteq v \vee v \subseteq u$
by *blast*
then show $x \sqsubseteq y \vee y \sqsubseteq x$
proof
assume $u \subseteq v$
from ‹*chain v*› **show** *?thesis*
proof (*rule chain-total*)
show $y \in v$ **by** *fact*
show $x \in v$ **using** ‹$u \subseteq v$› **and** ‹$x \in u$› **by** *blast*

```
    qed
  next
    assume v ⊆ u
    from ‹chain u› show ?thesis
    proof (rule chain-total)
      show x ∈ u by fact
      show y ∈ u using ‹v ⊆ u› and ‹y ∈ v› by blast
    qed
  qed
 qed
 ultimately show ?case unfolding chain-def ..
qed
```

26.1.2 Hausdorff's Maximum Principle

There exists a maximal totally ordered subset of A. (Note that we do not require A to be partially ordered.)

```
theorem Hausdorff: ∃ C. maxchain C
proof −
 let ?M = ⋃𝒞
 have maxchain ?M
 proof (rule ccontr)
  assume ¬ ?thesis
  then have suc ?M ≠ ?M
   using suc-not-equals and suc-Union-closed-chain [OF suc-Union-closed-Union]
by simp
  moreover have suc ?M = ?M
   using eq-suc-Union [OF suc-Union-closed-Union] by simp
  ultimately show False by contradiction
 qed
 then show ?thesis by blast
qed
```

Make notation $\mathcal{C}$ available again.

```
no-notation suc-Union-closed  (‹𝒞›)
```

```
lemma chain-extend: chain C ⟹ z ∈ A ⟹ ∀ x∈C. x ⊑ z ⟹ chain ({z} ∪ C)
 unfolding chain-def by blast
```

```
lemma maxchain-imp-chain: maxchain C ⟹ chain C
 by (simp add: maxchain-def)
```

```
end
```

Hide constant *pred-on.suc-Union-closed*, which was just needed for the proof of Hausforff's maximum principle.

```
hide-const pred-on.suc-Union-closed
```

lemma *chain-mono*:
assumes $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
and *pred-on.chain A P C*
shows *pred-on.chain A Q C*
using *assms* **unfolding** *pred-on.chain-def* **by** *blast*

26.1.3 Results for the proper subset relation

interpretation *subset*: *pred-on A* $(\subset)$ **for** A **.**

lemma *subset-maxchain-max*:
assumes *subset.maxchain A C*
and $X \in A$
and $\bigcup C \subseteq X$
shows $\bigcup C = X$
proof (*rule ccontr*)
let *?C* $= \{X\} \cup C$
from ‹*subset.maxchain A C*› **have** *subset.chain A C*
and $*$: $\bigwedge S.$ *subset.chain A S* $\Longrightarrow \neg\ C \subset S$
by (*auto simp*: *subset.maxchain-def*)
moreover have $\forall\, x{\in}C.\ x \subseteq X$ **using** ‹$\bigcup C \subseteq X$› **by** *auto*
ultimately have *subset.chain A ?C*
using *subset.chain-extend* [*of A C X*] **and** ‹$X \in A$› **by** *auto*
moreover assume $**$: $\bigcup C \neq X$
moreover from $**$ **have** $C \subset$ *?C* **using** ‹$\bigcup C \subseteq X$› **by** *auto*
ultimately show *False* **using** $*$ **by** *blast*
qed

lemma *subset-chain-def*: $\bigwedge \mathcal{A}.$ *subset.chain* $\mathcal{A}\ \mathcal{C} = (\mathcal{C} \subseteq \mathcal{A} \wedge (\forall\, X{\in}\mathcal{C}.\ \forall\, Y{\in}\mathcal{C}.\ X \subseteq Y \vee Y \subseteq X))$
by (*auto simp*: *subset.chain-def*)

lemma *subset-chain-insert*:
subset.chain $\mathcal{A}$ (*insert B* $\mathcal{B}$) $\longleftrightarrow B \in \mathcal{A} \wedge (\forall\, X{\in}\mathcal{B}.\ X \subseteq B \vee B \subseteq X) \wedge$ *subset.chain* $\mathcal{A}\ \mathcal{B}$
by (*fastforce simp add*: *subset-chain-def*)

26.1.4 Zorn's lemma

If every chain has an upper bound, then there is a maximal set.

theorem *subset-Zorn*:
assumes $\bigwedge C.$ *subset.chain A C* $\Longrightarrow \exists\, U{\in}A.\ \forall\, X{\in}C.\ X \subseteq U$
shows $\exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
proof −
from *subset.Hausdorff* [*of A*] **obtain** *M* **where** *subset.maxchain A M* **..**
then have *subset.chain A M*
by (*rule subset.maxchain-imp-chain*)
with *assms* **obtain** *Y* **where** $Y \in A$ **and** $\forall\, X{\in}M.\ X \subseteq Y$
by *blast*

```
  moreover have ∀X∈A. Y ⊆ X ⟶ Y = X
  proof (intro ballI impI)
    fix X
    assume X ∈ A and Y ⊆ X
    show Y = X
    proof (rule ccontr)
      assume ¬ ?thesis
      with ‹Y ⊆ X› have ¬ X ⊆ Y by blast
     from subset.chain-extend [OF ‹subset.chain A M› ‹X ∈ A›] and ‹∀X∈M. X
⊆ Y›
      have subset.chain A ({X} ∪ M)
        using ‹Y ⊆ X› by auto
      moreover have M ⊂ {X} ∪ M
        using ‹∀X∈M. X ⊆ Y› and ‹¬ X ⊆ Y› by auto
      ultimately show False
        using ‹subset.maxchain A M› by (auto simp: subset.maxchain-def)
    qed
  qed
  ultimately show ?thesis by blast
qed
```

Alternative version of Zorn's lemma for the subset relation.

```
lemma subset-Zorn':
  assumes ⋀C. subset.chain A C ⟹ ⋃C ∈ A
  shows ∃M∈A. ∀X∈A. M ⊆ X ⟶ X = M
proof −
  from subset.Hausdorff [of A] obtain M where subset.maxchain A M ..
  then have subset.chain A M
    by (rule subset.maxchain-imp-chain)
  with assms have ⋃M ∈ A .
  moreover have ∀Z∈A. ⋃M ⊆ Z ⟶ ⋃M = Z
  proof (intro ballI impI)
    fix Z
    assume Z ∈ A and ⋃M ⊆ Z
    with subset-maxchain-max [OF ‹subset.maxchain A M›]
      show ⋃M = Z .
  qed
  ultimately show ?thesis by blast
qed
```

26.2 Zorn's Lemma for Partial Orders

Relate old to new definitions.

```
definition chain-subset :: 'a set set ⇒ bool  (‹chain⊆›)
  where chain⊆ C ⟷ (∀A∈C. ∀B∈C. A ⊆ B ∨ B ⊆ A)

definition chains :: 'a set set ⇒ 'a set set set
  where chains A = {C. C ⊆ A ∧ chain⊆ C}
```

definition *Chains* :: $('a \times 'a)$ *set* $\Rightarrow$ $'a$ *set set*
where *Chains* $r = \{C.\ \forall a{\in}C.\ \forall b{\in}C.\ (a,\ b) \in r \vee (b,\ a) \in r\}$

lemma *chains-extend*: $c \in$ *chains* $S \Longrightarrow z \in S \Longrightarrow \forall x \in c.\ x \subseteq z \Longrightarrow \{z\} \cup c \in$ *chains* S
for z :: $'a$ *set*
unfolding *chains-def chain-subset-def* **by** *blast*

lemma *mono-Chains*: $r \subseteq s \Longrightarrow$ *Chains* $r \subseteq$ *Chains* s
unfolding *Chains-def* **by** *blast*

lemma *chain-subset-alt-def*: $chain_{\subseteq}\ C =$ *subset.chain UNIV* C
unfolding *chain-subset-def subset.chain-def* **by** *fast*

lemma *chains-alt-def*: *chains* $A = \{C.$ *subset.chain* $A\ C\}$
by (*simp add*: *chains-def chain-subset-alt-def subset.chain-def*)

lemma *Chains-subset*: *Chains* $r \subseteq \{C.$ *pred-on.chain UNIV* $(\lambda x\ y.\ (x,\ y) \in r)\ C\}$
by (*force simp add*: *Chains-def pred-on.chain-def*)

lemma *Chains-subset'*:
assumes *refl* r
shows $\{C.$ *pred-on.chain UNIV* $(\lambda x\ y.\ (x,\ y) \in r)\ C\} \subseteq$ *Chains* r
using *assms*
by (*auto simp add*: *Chains-def pred-on.chain-def refl-on-def*)

lemma *Chains-alt-def*:
assumes *refl* r
shows *Chains* $r = \{C.$ *pred-on.chain UNIV* $(\lambda x\ y.\ (x,\ y) \in r)\ C\}$
using *assms Chains-subset Chains-subset'* **by** *blast*

lemma *Chains-relation-of*:
assumes $C \in$ *Chains* (*relation-of* $P\ A$) **shows** $C \subseteq A$
using *assms* **unfolding** *Chains-def relation-of-def* **by** *auto*

lemma *pairwise-chain-Union*:
assumes P: $\bigwedge S.\ S \in \mathcal{C} \Longrightarrow$ *pairwise* $R\ S$ **and** $chain_{\subseteq}\ \mathcal{C}$
shows *pairwise* R $(\bigcup \mathcal{C})$
using ‹$chain_{\subseteq}\ \mathcal{C}$› **unfolding** *pairwise-def chain-subset-def*
by (*blast intro*: P [*unfolded pairwise-def*, *rule-format*])

lemma *Zorn-Lemma*: $\forall\, C{\in}$*chains* $A.\ \bigcup C \in A \Longrightarrow \exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
using *subset-Zorn'* [*of* A] **by** (*force simp*: *chains-alt-def*)

lemma *Zorn-Lemma2*: $\forall\, C{\in}$*chains* $A.\ \exists\, U{\in}A.\ \forall\, X{\in}C.\ X \subseteq U \Longrightarrow \exists\, M{\in}A.\ \forall\, X{\in}A.\ M \subseteq X \longrightarrow X = M$
using *subset-Zorn* [*of* A] **by** (*auto simp*: *chains-alt-def*)

26.3 Other variants of Zorn's Lemma

lemma *chainsD*: $c \in chains\ S \Longrightarrow x \in c \Longrightarrow y \in c \Longrightarrow x \subseteq y \vee y \subseteq x$
unfolding *chains-def chain-subset-def* **by** *blast*

lemma *chainsD2*: $c \in chains\ S \Longrightarrow c \subseteq S$
unfolding *chains-def* **by** *blast*

lemma *Zorns-po-lemma*:
assumes *po*: *Partial-order r*
and *u*: $\bigwedge C.\ C \in Chains\ r \Longrightarrow \exists\, u{\in}Field\ r.\ \forall\, a{\in}C.\ (a,\ u) \in r$
shows $\exists\, m{\in}Field\ r.\ \forall\, a{\in}Field\ r.\ (m,\ a) \in r \longrightarrow a = m$
proof −
have *Preorder r*
using *po* **by** (*simp add*: *partial-order-on-def*)

Mirror r in the set of subsets below (wrt r) elements of A.

let *?B* $= \lambda x.\ r^{-1}$ `` $\{x\}$
let *?S* $=$ *?B* ` *Field r*
have $\exists\, u{\in}Field\ r.\ \forall\, A{\in}C.\ A \subseteq r^{-1}$ `` $\{u\}$ (**is** $\exists\, u{\in}Field\ r.$ *?P u*)
if *1*: $C \subseteq$ *?S* **and** *2*: $\forall\, A{\in}C.\ \forall\, B{\in}C.\ A \subseteq B \vee B \subseteq A$ **for** *C*
proof −
let *?A* $= \{x{\in}Field\ r.\ \exists\, M{\in}C.\ M =$ *?B x*$\}$
from *1* **have** *C* $=$ *?B* ` *?A* **by** (*auto simp*: *image-def*)
have *?A* $\in$ *Chains r*
proof (*simp add*: *Chains-def*, *intro allI impI*, *elim conjE*)
fix *a b*
assume *a* $\in$ *Field r* **and** *?B a* $\in$ *C* **and** *b* $\in$ *Field r* **and** *?B b* $\in$ *C*
with *2* **have** *?B a* $\subseteq$ *?B b* $\vee$ *?B b* $\subseteq$ *?B a* **by** *auto*
then show $(a,\ b) \in r \vee (b,\ a) \in r$
using ‹*Preorder r*› **and** ‹*a* $\in$ *Field r*› **and** ‹*b* $\in$ *Field r*›
by (*simp add*:*subset-Image1-Image1-iff*)
qed
then obtain *u* **where** *uA*: $u \in Field\ r\ \forall\, a{\in}?A.\ (a,\ u) \in r$
by (*auto simp*: *dest*: *u*)
have *?P u*
proof *auto*
fix *a B* **assume** *aB*: $B \in C\ a \in B$
with *1* **obtain** *x* **where** $x \in Field\ r$ **and** $B = r^{-1}$ `` $\{x\}$ **by** *auto*
then show $(a,\ u) \in r$
using *uA* **and** *aB* **and** ‹*Preorder r*›
unfolding *preorder-on-def refl-on-def* **by** *simp* (*fast dest*: *transD*)
qed
then show *?thesis*
using ‹*u* $\in$ *Field r*› **by** *blast*
qed
then have $\forall\, C{\in}chains\ ?S.\ \exists\, U{\in}?S.\ \forall\, A{\in}C.\ A \subseteq U$
by (*auto simp*: *chains-def chain-subset-def*)
from *Zorn-Lemma2* [*OF this*] **obtain** *m B*
where *m* $\in$ *Field r*

and $B = r^{-1}$ `` ``{m}
and $\forall x{\in}$*Field r*. $B \subseteq r^{-1}$ `` ``{x} $\longrightarrow$ r^{-1} `` ``{x} = B
by *auto*
then have $\forall a{\in}$*Field r*. (*m*, *a*) $\in$ *r* $\longrightarrow$ *a* = *m*
using *po* **and** ‹*Preorder r*› **and** ‹*m* $\in$ *Field r*›
by (*auto simp*: *subset-Image1-Image1-iff Partial-order-eq-Image1-Image1-iff*)
then show *?thesis*
using ‹*m* $\in$ *Field r*› **by** *blast*
qed

lemma *predicate-Zorn*:
assumes *po*: *partial-order-on A* (*relation-of P A*)
and *ch*: $\bigwedge C$. *C* $\in$ *Chains* (*relation-of P A*) $\Longrightarrow$ $\exists u \in A. \forall a \in C.$ *P a u*
shows $\exists m \in A. \forall a \in A.$ *P m a* $\longrightarrow$ *a* = *m*
proof −
have *a* $\in$ *A* **if** *C* $\in$ *Chains* (*relation-of P A*) **and** *a* $\in$ *C* **for** *C a*
using *that* **unfolding** *Chains-def relation-of-def* **by** *auto*
moreover have (*a*, *u*) $\in$ *relation-of P A* **if** *a* $\in$ *A* **and** *u* $\in$ *A* **and** *P a u* **for** *a u*
unfolding *relation-of-def* **using** *that* **by** *auto*
ultimately have $\exists m{\in}A. \forall a{\in}A.$ (*m*, *a*) $\in$ *relation-of P A* $\longrightarrow$ *a* = *m*
using *Zorns-po-lemma*[*OF Partial-order-relation-ofI*[*OF po*], *rule-format*] *ch*
unfolding *Field-relation-of*[*OF partial-order-onD*(*4*)[*OF po*] *partial-order-onD*(*1*)[*OF po*]] **by** *blast*
then show *?thesis*
by (*auto simp*: *relation-of-def*)
qed

lemma *Union-in-chain*: ⟦*finite* $\mathcal{B}$; $\mathcal{B} \neq \{\}$; *subset.chain* $\mathcal{A}$ $\mathcal{B}$⟧ $\Longrightarrow$ $\bigcup \mathcal{B} \in \mathcal{B}$
proof (*induction* $\mathcal{B}$ *rule*: *finite-induct*)
case (*insert B* $\mathcal{B}$)
show *?case*
proof (*cases* $\mathcal{B} = \{\}$)
case *False*
then show *?thesis*
using *insert sup.absorb2* **by** (*auto simp*: *subset-chain-insert dest*!: *bspec* [**where** *x*=$\bigcup \mathcal{B}$])
qed *auto*
qed *simp*

lemma *Inter-in-chain*: ⟦*finite* $\mathcal{B}$; $\mathcal{B} \neq \{\}$; *subset.chain* $\mathcal{A}$ $\mathcal{B}$⟧ $\Longrightarrow$ $\bigcap \mathcal{B} \in \mathcal{B}$
proof (*induction* $\mathcal{B}$ *rule*: *finite-induct*)
case (*insert B* $\mathcal{B}$)
show *?case*
proof (*cases* $\mathcal{B} = \{\}$)
case *False*
then show *?thesis*
using *insert inf.absorb2* **by** (*auto simp*: *subset-chain-insert dest*!: *bspec* [**where** *x*=$\bigcap \mathcal{B}$])
qed *auto*

qed *simp*

lemma *finite-subset-Union-chain*:
assumes *finite A* $A \subseteq \bigcup \mathcal{B}$ $\mathcal{B} \neq \{\}$ **and** *sub*: *subset.chain* $\mathcal{A}$ $\mathcal{B}$
obtains B **where** $B \in \mathcal{B}$ $A \subseteq B$
proof −
obtain $\mathcal{F}$ **where** $\mathcal{F}$: *finite* $\mathcal{F}$ $\mathcal{F} \subseteq \mathcal{B}$ $A \subseteq \bigcup \mathcal{F}$
using *assms* **by** (*auto intro*: *finite-subset-Union*)
show *thesis*
proof (*cases* $\mathcal{F} = \{\}$)
case *True*
then show *?thesis*
using ‹$A \subseteq \bigcup \mathcal{F}$› ‹$\mathcal{B} \neq \{\}$› *that* **by** *fastforce*
next
case *False*
show *?thesis*
proof
show $\bigcup \mathcal{F} \in \mathcal{B}$
using *sub* ‹$\mathcal{F} \subseteq \mathcal{B}$› ‹*finite* $\mathcal{F}$›
by (*simp add*: *Union-in-chain False subset.chain-def subset-iff*)
show $A \subseteq \bigcup \mathcal{F}$
using ‹$A \subseteq \bigcup \mathcal{F}$› **by** *blast*
qed
qed
qed

lemma *subset-Zorn-nonempty*:
assumes $\mathcal{A} \neq \{\}$ **and** *ch*: $\bigwedge \mathcal{C}$. $[\![\mathcal{C} \neq \{\}$; *subset.chain* $\mathcal{A}$ $\mathcal{C}]\!] \Longrightarrow \bigcup \mathcal{C} \in \mathcal{A}$
shows $\exists M \in \mathcal{A}.\ \forall X \in \mathcal{A}.\ M \subseteq X \longrightarrow X = M$
proof (*rule subset-Zorn*)
show $\exists U \in \mathcal{A}.\ \forall X \in \mathcal{C}.\ X \subseteq U$ **if** *subset.chain* $\mathcal{A}$ $\mathcal{C}$ **for** $\mathcal{C}$
proof (*cases* $\mathcal{C} = \{\}$)
case *True*
then show *?thesis*
using ‹$\mathcal{A} \neq \{\}$› **by** *blast*
next
case *False*
show *?thesis*
by (*blast intro*!: *ch False that Union-upper*)
qed
qed

26.4 The Well Ordering Theorem

definition *init-seg-of* :: $(('a \times 'a)\ set \times ('a \times 'a)\ set)\ set$
where *init-seg-of* = $\{(r, s).\ r \subseteq s \wedge (\forall a\ b\ c.\ (a, b) \in s \wedge (b, c) \in r \longrightarrow (a, b) \in r)\}$

abbreviation *initial-segment-of-syntax* :: $('a \times 'a)\ set \Rightarrow ('a \times 'a)\ set \Rightarrow bool$

(**infix** ‹*initial'-segment'-of*› *55*)
where *r initial-segment-of s* ≡ (*r*, *s*) ∈ *init-seg-of*

lemma *refl-on-init-seg-of* [*simp*]: *r initial-segment-of r*
by (*simp add*: *init-seg-of-def*)

lemma *trans-init-seg-of*:
r initial-segment-of s ⟹ *s initial-segment-of t* ⟹ *r initial-segment-of t*
by (*simp* (*no-asm-use*) *add*: *init-seg-of-def*) *blast*

lemma *antisym-init-seg-of*: *r initial-segment-of s* ⟹ *s initial-segment-of r* ⟹ *r* = *s*
unfolding *init-seg-of-def* **by** *safe*

lemma *Chains-init-seg-of-Union*: *R* ∈ *Chains init-seg-of* ⟹ *r*∈*R* ⟹ *r initial-segment-of* ⋃ *R*
by (*auto simp*: *init-seg-of-def Ball-def Chains-def*) *blast*

lemma *chain-subset-trans-Union*:
assumes *chain*⊆ *R* ∀ *r*∈*R*. *trans r*
shows *trans* (⋃ *R*)
proof (*intro transI*, *elim UnionE*)
fix *S1 S2* :: *'a rel* **and** *x y z* :: *'a*
assume *S1* ∈ *R S2* ∈ *R*
with *assms*(*1*) **have** *S1* ⊆ *S2* ∨ *S2* ⊆ *S1*
unfolding *chain-subset-def* **by** *blast*
moreover assume (*x*, *y*) ∈ *S1* (*y*, *z*) ∈ *S2*
ultimately have ((*x*, *y*) ∈ *S1* ∧ (*y*, *z*) ∈ *S1*) ∨ ((*x*, *y*) ∈ *S2* ∧ (*y*, *z*) ∈ *S2*)
by *blast*
with ‹*S1* ∈ *R*› ‹*S2* ∈ *R*› *assms*(*2*) **show** (*x*, *z*) ∈ ⋃ *R*
by (*auto elim*: *transE*)
qed

lemma *chain-subset-antisym-Union*:
assumes *chain*⊆ *R* ∀ *r*∈*R*. *antisym r*
shows *antisym* (⋃ *R*)
proof (*intro antisymI*, *elim UnionE*)
fix *S1 S2* :: *'a rel* **and** *x y* :: *'a*
assume *S1* ∈ *R S2* ∈ *R*
with *assms*(*1*) **have** *S1* ⊆ *S2* ∨ *S2* ⊆ *S1*
unfolding *chain-subset-def* **by** *blast*
moreover assume (*x*, *y*) ∈ *S1* (*y*, *x*) ∈ *S2*
ultimately have ((*x*, *y*) ∈ *S1* ∧ (*y*, *x*) ∈ *S1*) ∨ ((*x*, *y*) ∈ *S2* ∧ (*y*, *x*) ∈ *S2*)
by *blast*
with ‹*S1* ∈ *R*› ‹*S2* ∈ *R*› *assms*(*2*) **show** *x* = *y*
unfolding *antisym-def* **by** *auto*
qed

lemma *chain-subset-Total-Union*:

assumes *chain*$_{\subseteq}$ *R* **and** $\forall r{\in}R.$ *Total r*
shows *Total* $(\bigcup R)$
proof (*simp add*: *total-on-def Ball-def*, *auto del*: *disjCI*)
fix *r s a b*
assume *A*: $r \in R$ $s \in R$ $a \in Field\ r$ $b \in Field\ s$ $a \neq b$
from ‹*chain*$_{\subseteq}$ *R*› **and** ‹$r \in R$› **and** ‹$s \in R$› **have** $r \subseteq s \vee s \subseteq r$
by (*auto simp add*: *chain-subset-def*)
then show $(\exists r{\in}R.\ (a, b) \in r) \vee (\exists r{\in}R.\ (b, a) \in r)$
proof
assume $r \subseteq s$
then have $(a, b) \in s \vee (b, a) \in s$
using *assms*(*2*) *A mono-Field*[*of r s*]
by (*auto simp add*: *total-on-def*)
then show *?thesis*
using ‹$s \in R$› **by** *blast*
next
assume $s \subseteq r$
then have $(a, b) \in r \vee (b, a) \in r$
using *assms*(*2*) *A mono-Field*[*of s r*]
by (*fastforce simp add*: *total-on-def*)
then show *?thesis*
using ‹$r \in R$› **by** *blast*
qed
qed

lemma *wf-Union-wf-init-segs*:
assumes $R \in Chains\ init\text{-}seg\text{-}of$
and $\forall r{\in}R.\ wf\ r$
shows *wf* $(\bigcup R)$
proof (*simp add*: *wf-iff-no-infinite-down-chain*, *rule ccontr*, *auto*)
fix *f*
assume *1*: $\forall i.\ \exists r{\in}R.\ (f\ (Suc\ i), f\ i) \in r$
then obtain *r* **where** $r \in R$ **and** $(f\ (Suc\ 0), f\ 0) \in r$ **by** *auto*
have $(f\ (Suc\ i), f\ i) \in r$ **for** *i*
proof (*induct i*)
case *0*
show *?case* **by** *fact*
next
case (*Suc i*)
then obtain *s* **where** *s*: $s \in R$ $(f\ (Suc\ (Suc\ i)), f(Suc\ i)) \in s$
using *1* **by** *auto*
then have *s initial-segment-of r* $\vee$ *r initial-segment-of s*
using *assms*(*1*) ‹$r \in R$› **by** (*simp add*: *Chains-def*)
with *Suc s* **show** *?case* **by** (*simp add*: *init-seg-of-def*) *blast*
qed
then show *False*
using *assms*(*2*) **and** ‹$r \in R$›
by (*simp add*: *wf-iff-no-infinite-down-chain*) *blast*
qed

lemma *initial-segment-of-Diff*: *p initial-segment-of q* $\Longrightarrow$ *p* − *s initial-segment-of q* − *s*
unfolding *init-seg-of-def* **by** *blast*

lemma *Chains-inits-DiffI*: *R* ∈ *Chains init-seg-of* $\Longrightarrow$ {*r* − *s* |*r*. *r* ∈ *R*} ∈ *Chains init-seg-of*
unfolding *Chains-def* **by** (*blast intro*: *initial-segment-of-Diff*)

theorem *well-ordering*: ∃ *r*::'*a rel*. *Well-order r* ∧ *Field r* = *UNIV*
proof −
— The initial segment relation on well-orders:
let *?WO* = {*r*::'*a rel*. *Well-order r*}
define *I* **where** *I* = *init-seg-of* ∩ *?WO* × *?WO*
then have *I-init*: *I* ⊆ *init-seg-of* **by** *simp*
then have *subch*: ⋀*R*. *R* ∈ *Chains I* $\Longrightarrow$ *chain*$_{\subseteq}$ *R*
unfolding *init-seg-of-def chain-subset-def Chains-def* **by** *blast*
have *Chains-wo*: ⋀*R r*. *R* ∈ *Chains I* $\Longrightarrow$ *r* ∈ *R* $\Longrightarrow$ *Well-order r*
by (*simp add*: *Chains-def I-def*) *blast*
have *FI*: *Field I* = *?WO*
by (*auto simp add*: *I-def init-seg-of-def Field-def*)
then have *0*: *Partial-order I*
by (*auto simp*: *partial-order-on-def preorder-on-def antisym-def antisym-init-seg-of refl-on-def*
trans-def I-def elim!: *trans-init-seg-of*)
— *I*-chains have upper bounds in *?WO* wrt *I*: their Union
have ⋃*R* ∈ *?WO* ∧ (∀ *r*∈*R*. (*r*, ⋃*R*) ∈ *I*) **if** *R* ∈ *Chains I* **for** *R*
proof −
from *that* **have** *Ris*: *R* ∈ *Chains init-seg-of*
using *mono-Chains* [*OF I-init*] **by** *blast*
have *subch*: *chain*$_{\subseteq}$ *R*
using ‹*R* ∈ *Chains I*› *I-init* **by** (*auto simp*: *init-seg-of-def chain-subset-def Chains-def*)
have ∀ *r*∈*R*. *Refl r* **and** ∀ *r*∈*R*. *trans r* **and** ∀ *r*∈*R*. *antisym r*
and ∀ *r*∈*R*. *Total r* **and** ∀ *r*∈*R*. *wf* (*r* − *Id*)
using *Chains-wo* [*OF* ‹*R* ∈ *Chains I*›] **by** (*simp-all add*: *order-on-defs*)
have (⋃ *R*) ⊆ *Field* (⋃ *R*) × *Field* (⋃ *R*)
unfolding *Field-def* **by** *auto*
moreover have *Refl* (⋃*R*)
using ‹∀ *r*∈*R*. *Refl r*› **unfolding** *refl-on-def* **by** *fastforce*
moreover have *trans* (⋃*R*)
by (*rule chain-subset-trans-Union* [*OF subch* ‹∀ *r*∈*R*. *trans r*›])
moreover have *antisym* (⋃*R*)
by (*rule chain-subset-antisym-Union* [*OF subch* ‹∀ *r*∈*R*. *antisym r*›])
moreover have *Total* (⋃*R*)
by (*rule chain-subset-Total-Union* [*OF subch* ‹∀ *r*∈*R*. *Total r*›])
moreover have *wf* ((⋃*R*) − *Id*)
proof −
have (⋃*R*) − *Id* = ⋃{*r* − *Id* | *r*. *r* ∈ *R*} **by** *blast*

with ‹∀ *r*∈*R*. *wf* (*r* − *Id*)› **and** *wf-Union-wf-init-segs* [*OF Chains-inits-DiffI* [*OF Ris*]]
show *?thesis* **by** *fastforce*
qed
ultimately have *Well-order* (⋃ *R*)
by (*simp add*:*order-on-defs*)
moreover have ∀ *r* ∈ *R*. *r initial-segment-of* ⋃ *R*
using *Ris* **by** (*simp add*: *Chains-init-seg-of-Union*)
ultimately show *?thesis*
using *mono-Chains* [*OF I-init*] *Chains-wo*[*of R*] **and** ‹*R* ∈ *Chains I*›
unfolding *I-def* **by** *blast*
qed
then have *1*: ∃ *u*∈*Field I*. ∀ *r*∈*R*. (*r*, *u*) ∈ *I* **if** *R* ∈ *Chains I* **for** *R*
using *that* **by** (*subst FI*) *blast*
— Zorn's Lemma yields a maximal well-order *m*:
then obtain *m* :: *'a rel*
where *Well-order m*
and *max*: ∀ *r*. *Well-order r* ∧ (*m*, *r*) ∈ *I* ⟶ *r* = *m*
using *Zorns-po-lemma*[*OF 0 1*] **unfolding** *FI* **by** *fastforce*
— Now show by contradiction that *m* covers the whole type:
have *False* **if** *x* ∉ *Field m* **for** *x* :: *'a*
proof −
— Assuming that *x* is not covered and extend *m* at the top with *x*
have *m* ≠ {}
proof
assume *m* = {}
moreover have *Well-order* {(*x*, *x*)}
by (*simp add*: *order-on-defs refl-on-def trans-def antisym-def total-on-def Field-def*)
ultimately show *False* **using** *max*
by (*auto simp*: *I-def init-seg-of-def simp del*: *Field-insert*)
qed
then have *Field m* ≠ {} **by** (*auto simp*: *Field-def*)
moreover have *wf* (*m* − *Id*)
using ‹*Well-order m*› **by** (*simp add*: *well-order-on-def*)
— The extension of *m* by *x*:
let *?s* = {(*a*, *x*) | *a*. *a* ∈ *Field m*}
let *?m* = *insert* (*x*, *x*) *m* ∪ *?s*
have *Fm*: *Field ?m* = *insert x* (*Field m*)
by (*auto simp*: *Field-def*)
have *Refl m* **and** *trans m* **and** *antisym m* **and** *Total m* **and** *wf* (*m* − *Id*) **and**
m ⊆ *Field m* × *Field m*
using ‹*Well-order m*› **by** (*simp-all add*: *order-on-defs*)
— We show that the extension is a well-order
have *?m* ⊆ *Field ?m* × *Field ?m*
using ‹*m* ⊆ *Field m* × *Field m*› **by** *auto*
moreover have *Refl ?m*
using ‹*Refl m*› *Fm* **unfolding** *refl-on-def* **by** *blast*
moreover have *trans ?m* **using** ‹*trans m*› **and** ‹*x* ∉ *Field m*›

```
    unfolding trans-def Field-def by blast
  moreover have antisym ?m
    using ‹antisym m› and ‹x ∉ Field m› unfolding antisym-def Field-def by
blast
  moreover have Total ?m
    using ‹Total m› and Fm by (auto simp: total-on-def)
  moreover have wf (?m − Id)
  proof −
    have wf ?s
      using ‹x ∉ Field m› by (auto simp: wf-eq-minimal Field-def Bex-def)
    then show ?thesis
      using ‹wf (m − Id)› and ‹x ∉ Field m› wf-subset [OF ‹wf ?s› Diff-subset]
      by (auto simp: Un-Diff Field-def intro: wf-Un)
  qed
  ultimately have Well-order ?m
    by (simp add: order-on-defs)
— We show that the extension is above m
  moreover have (m, ?m) ∈ I
    using ‹Well-order ?m› and ‹Well-order m› and ‹x ∉ Field m›
    by (fastforce simp: I-def init-seg-of-def Field-def)
  ultimately
— This contradicts maximality of m:
  show False
    using max and ‹x ∉ Field m› unfolding Field-def by blast
 qed
 then have Field m = UNIV by auto
 with ‹Well-order m› show ?thesis by blast
qed

corollary well-order-on: ∃ r::'a rel. well-order-on A r
proof −
 obtain r :: 'a rel where wo: Well-order r and univ: Field r = UNIV
   using well-ordering [where 'a = 'a] by blast
 let ?r = {(x, y). x ∈ A ∧ y ∈ A ∧ (x, y) ∈ r}
 have 1: Field ?r = A
   using wo univ by (fastforce simp: Field-def order-on-defs refl-on-def)
 from ‹Well-order r› have Refl r trans r antisym r Total r wf (r − Id) and
   r ⊆ Field r × Field r
   by (simp-all add: order-on-defs)
 have ?r ⊆ Field ?r × Field ?r
    using ‹r ⊆ Field r × Field r› by (auto simp: 1)
 moreover from ‹Refl r› have Refl ?r
   by (auto simp: refl-on-def 1 univ)
 moreover from ‹trans r› have trans ?r
   unfolding trans-def by blast
 moreover from ‹antisym r› have antisym ?r
   unfolding antisym-def by blast
 moreover from ‹Total r› have Total ?r
   by (simp add:total-on-def 1 univ)
```

moreover have *wf* (*?r* − *Id*)
by (*rule wf-subset* [*OF* ‹*wf* (*r* − *Id*)›]) *blast*
ultimately have *Well-order ?r*
by (*simp add*: *order-on-defs*)
with *1* **show** *?thesis* **by** *auto*
qed

lemma *dependent-wf-choice*:
fixes $P :: ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$
assumes *wf R*
and *adm*: $\bigwedge f\ g\ x\ r.\ (\bigwedge z.\ (z, x) \in R \Longrightarrow f\ z = g\ z) \Longrightarrow P\ f\ x\ r = P\ g\ x\ r$
and *P*: $\bigwedge x\ f.\ (\bigwedge y.\ (y, x) \in R \Longrightarrow P\ f\ y\ (f\ y)) \Longrightarrow \exists\, r.\ P\ f\ x\ r$
shows $\exists f.\ \forall x.\ P\ f\ x\ (f\ x)$
proof (*intro exI allI*)
fix *x*
define *f* **where** $f \equiv wfrec\ R\ (\lambda f\ x.\ SOME\ r.\ P\ f\ x\ r)$
from ‹*wf R*› **show** *P f x* (*f x*)
proof (*induct x*)
case (*less x*)
show *P f x* (*f x*)
proof (*subst* (*2*) *wfrec-def-adm*[*OF f-def* ‹*wf R*›])
show *adm-wf R* ($\lambda f\ x.$ *SOME r. P f x r*)
by (*auto simp*: *adm-wf-def intro*!: *arg-cong*[**where** *f*=*Eps*] *adm*)
show *P f x* (*Eps* (*P f x*))
using *P* **by** (*rule someI-ex*) *fact*
qed
qed
qed

lemma (**in** *wellorder*) *dependent-wellorder-choice*:
assumes $\bigwedge r\ f\ g\ x.\ (\bigwedge y.\ y < x \Longrightarrow f\ y = g\ y) \Longrightarrow P\ f\ x\ r = P\ g\ x\ r$
and *P*: $\bigwedge x\ f.\ (\bigwedge y.\ y < x \Longrightarrow P\ f\ y\ (f\ y)) \Longrightarrow \exists\, r.\ P\ f\ x\ r$
shows $\exists f.\ \forall x.\ P\ f\ x\ (f\ x)$
using *wf* **by** (*rule dependent-wf-choice*) (*auto intro*!: *assms*)

end

27 Well-Order Relations as Needed by Bounded Natural Functors

theory *BNF-Wellorder-Relation*
imports *Order-Relation*
begin

In this section, we develop basic concepts and results pertaining to well-order relations. Note that we consider well-order relations as *non-strict relations*, i.e., as containing the diagonals of their fields.

locale *wo-rel* =

fixes $r :: 'a\ rel$
assumes *WELL*: *Well-order r*
begin

The following context encompasses all this section. In other words, for the whole section, we consider a fixed well-order relation r.

abbreviation *under* **where** *under* $\equiv$ *Order-Relation.under r*
abbreviation *underS* **where** *underS* $\equiv$ *Order-Relation.underS r*
abbreviation *Under* **where** *Under* $\equiv$ *Order-Relation.Under r*
abbreviation *UnderS* **where** *UnderS* $\equiv$ *Order-Relation.UnderS r*
abbreviation *above* **where** *above* $\equiv$ *Order-Relation.above r*
abbreviation *aboveS* **where** *aboveS* $\equiv$ *Order-Relation.aboveS r*
abbreviation *Above* **where** *Above* $\equiv$ *Order-Relation.Above r*
abbreviation *AboveS* **where** *AboveS* $\equiv$ *Order-Relation.AboveS r*
abbreviation *ofilter* **where** *ofilter* $\equiv$ *Order-Relation.ofilter r*
lemmas *ofilter-def* = *Order-Relation.ofilter-def*[*of r*]

27.1 Auxiliaries

lemma *REFL*: *Refl r*
using *WELL order-on-defs*[*of - r*] **by** *auto*

lemma *TRANS*: *trans r*
using *WELL order-on-defs*[*of - r*] **by** *auto*

lemma *ANTISYM*: *antisym r*
using *WELL order-on-defs*[*of - r*] **by** *auto*

lemma *TOTAL*: *Total r*
using *WELL order-on-defs*[*of - r*] **by** *auto*

lemma *TOTALS*: $\forall a \in Field\ r.\ \forall b \in Field\ r.\ (a,b) \in r \vee (b,a) \in r$
using *REFL TOTAL refl-on-def*[*of - r*] *total-on-def*[*of - r*] **by** *force*

lemma *LIN*: *Linear-order r*
using *WELL well-order-on-def*[*of - r*] **by** *auto*

lemma *WF*: *wf* $(r - Id)$
using *WELL well-order-on-def*[*of - r*] **by** *auto*

lemma *cases-Total*:
$\bigwedge$ *phi a b*. $[\![\{a,b\} <= Field\ r;\ ((a,b) \in r \Longrightarrow phi\ a\ b);\ ((b,a) \in r \Longrightarrow phi\ a\ b)]\!]$
$\Longrightarrow$ *phi a b*
using *TOTALS* **by** *auto*

lemma *cases-Total3*:
$\bigwedge$ *phi a b*. $[\![\{a,b\} \leq Field\ r;\ ((a,b) \in r - Id \vee (b,a) \in r - Id \Longrightarrow phi\ a\ b);$
$(a = b \Longrightarrow phi\ a\ b)]\!] \Longrightarrow phi\ a\ b$
using *TOTALS* **by** *auto*

27.2 Well-founded induction and recursion adapted to non-strict well-order relations

Here we provide induction and recursion principles specific to *non-strict* well-order relations. Although minor variations of those for well-founded relations, they will be useful for doing away with the tediousness of having to take out the diagonal each time in order to switch to a well-founded relation.

lemma *well-order-induct*:
assumes *IND*: $\bigwedge x.\ \forall y.\ y \neq x \wedge (y, x) \in r \longrightarrow P\ y \Longrightarrow P\ x$
shows $P\ a$
proof−
have $\bigwedge x.\ \forall y.\ (y, x) \in r - Id \longrightarrow P\ y \Longrightarrow P\ x$
using *IND* **by** *blast*
thus $P\ a$ **using** *WF wf-induct*[*of* $r - Id\ P\ a$] **by** *blast*
qed

definition
worec :: $(('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
$worec\ F \equiv wfrec\ (r - Id)\ F$

definition
adm-wo :: $(('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b) \Rightarrow bool$
where
$adm\text{-}wo\ H \equiv \forall f\ g\ x.\ (\forall y \in underS\ x.\ f\ y = g\ y) \longrightarrow H\ f\ x = H\ g\ x$

lemma *worec-fixpoint*:
assumes *ADM*: *adm-wo H*
shows $worec\ H = H\ (worec\ H)$
proof−
let *?rS* $= r - Id$
have *adm-wf* $(r - Id)\ H$
unfolding *adm-wf-def*
using *ADM adm-wo-def*[*of H*] *underS-def*[*of r*] **by** *auto*
hence *wfrec ?rS H* $= H$ (*wfrec ?rS H*)
using *WF wfrec-fixpoint*[*of ?rS H*] **by** *simp*
thus *?thesis* **unfolding** *worec-def* **.**
qed

27.3 The notions of maximum, minimum, supremum, successor and order filter

We define the successor *of a set*, and not of an element (the latter is of course a particular case). Also, we define the maximum *of two elements*, *max2*, and the minimum *of a set*, *minim* – we chose these variants since we consider them the most useful for well-orders. The minimum is defined in terms of the auxiliary relational operator *isMinim*. Then, supremum and

successor are defined in terms of minimum as expected. The minimum is only meaningful for non-empty sets, and the successor is only meaningful for sets for which strict upper bounds exist. Order filters for well-orders are also known as "initial segments".

definition *max2* :: *'a* ⇒ *'a* ⇒ *'a*
  where *max2 a b* ≡ *if* (*a*,*b*) ∈ *r then b else a*

definition *isMinim* :: *'a set* ⇒ *'a* ⇒ *bool*
  where *isMinim A b* ≡ *b* ∈ *A* ∧ (∀ *a* ∈ *A*. (*b*,*a*) ∈ *r*)

definition *minim* :: *'a set* ⇒ *'a*
  where *minim A* ≡ *THE b. isMinim A b*

definition *supr* :: *'a set* ⇒ *'a*
  where *supr A* ≡ *minim* (*Above A*)

definition *suc* :: *'a set* ⇒ *'a*
  where *suc A* ≡ *minim* (*AboveS A*)

27.3.1 Properties of max2

lemma *max2-greater-among*:
  assumes *a* ∈ *Field r* **and** *b* ∈ *Field r*
  shows (*a*, *max2 a b*) ∈ *r* ∧ (*b*, *max2 a b*) ∈ *r* ∧ *max2 a b* ∈ {*a*,*b*}
proof−
  {assume (*a*,*b*) ∈ *r*
    hence *?thesis* **using** *max2-def assms REFL refl-on-def*
      by (*auto simp add*: *refl-on-def*)
  }
  moreover
  {assume *a* = *b*
    hence (*a*,*b*) ∈ *r* **using** *REFL assms*
      by (*auto simp add*: *refl-on-def*)
  }
  moreover
  {assume ∗: *a* ≠ *b* ∧ (*b*,*a*) ∈ *r*
    hence (*a*,*b*) ∉ *r* **using** *ANTISYM*
      by (*auto simp add*: *antisym-def*)
    hence *?thesis* **using** ∗ *max2-def assms REFL refl-on-def*
      by (*auto simp add*: *refl-on-def*)
  }
  ultimately show *?thesis* **using** *assms TOTAL*
      total-on-def[*of Field r r*] **by** *blast*
qed

lemma *max2-greater*:
  assumes *a* ∈ *Field r* **and** *b* ∈ *Field r*
  shows (*a*, *max2 a b*) ∈ *r* ∧ (*b*, *max2 a b*) ∈ *r*
  using *assms* **by** (*auto simp add*: *max2-greater-among*)

lemma *max2-among*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $max2\ a\ b \in \{a, b\}$
using *assms max2-greater-among*[*of a b*] **by** *simp*

lemma *max2-equals1*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $(max2\ a\ b = a) = ((b,a) \in r)$
using *assms ANTISYM* **unfolding** *antisym-def* **using** *TOTALS*
by(*auto simp add*: *max2-def max2-among*)

lemma *max2-equals2*:
assumes $a \in Field\ r$ **and** $b \in Field\ r$
shows $(max2\ a\ b = b) = ((a,b) \in r)$
using *assms ANTISYM* **unfolding** *antisym-def* **using** *TOTALS*
unfolding *max2-def* **by** *auto*

lemma *in-notinI*:
assumes $(j,i) \notin r \vee j = i$ **and** $i \in Field\ r$ **and** $j \in Field\ r$
shows $(i,j) \in r$ **using** *assms max2-def max2-greater-among* **by** *fastforce*

27.3.2 Existence and uniqueness for isMinim and well-definedness of minim

lemma *isMinim-unique*:
assumes *isMinim B a isMinim B a'*
shows $a = a'$
using *assms ANTISYM antisym-def*[*of r*] **by** (*auto simp*: *isMinim-def*)

lemma *Well-order-isMinim-exists*:
assumes *SUB*: $B \leq Field\ r$ **and** *NE*: $B \neq \{\}$
shows $\exists\, b.\ isMinim\ B\ b$
proof−
from *spec*[*OF WF*[*unfolded wf-eq-minimal*[*of r − Id*]], *of B*] *NE* **obtain** b **where**
$*$: $b \in B \wedge (\forall\, b'.\ b' \neq b \wedge (b',b) \in r \longrightarrow b' \notin B)$ **by** *auto*
have $\forall\, b'.\ b' \in B \longrightarrow (b,\ b') \in r$
proof
fix b'
show $b' \in B \longrightarrow (b,\ b') \in r$
proof
assume *As*: $b' \in B$
hence $**$: $b \in Field\ r \wedge b' \in Field\ r$ **using** *As SUB* $*$ **by** *auto*
from *As* $*$ **have** $b' = b \vee (b',b) \notin r$ **by** *auto*
moreover have $b' = b \Longrightarrow (b,\ b') \in r$
using $**$ *REFL* **by** (*auto simp add*: *refl-on-def*)
moreover have $b' \neq b \wedge (b',b) \notin r \Longrightarrow (b,b') \in r$
using $**$ *TOTAL* **by** (*auto simp add*: *total-on-def*)
ultimately show $(b,b') \in r$ **by** *blast*

qed
qed
then show *?thesis*
unfolding *isMinim-def* using ∗ by *auto*
qed

lemma *minim-isMinim*:
assumes *SUB*: $B \leq Field\ r$ and *NE*: $B \neq \{\}$
shows *isMinim B* (*minim B*)
proof−
let *?phi* = (λ *b*. *isMinim B b*)
from *assms Well-order-isMinim-exists*
obtain *b* where ∗: *?phi b* by *blast*
moreover
have $\bigwedge b'$. *?phi* $b' \Longrightarrow b' = b$
using *isMinim-unique* ∗ by *auto*
ultimately show *?thesis*
unfolding *minim-def* using *theI*[*of ?phi b*] by *blast*
qed

27.3.3 Properties of minim

lemma *minim-in*:
assumes $B \leq Field\ r$ and $B \neq \{\}$
shows *minim B* ∈ *B*
using *assms minim-isMinim*[*of B*] by (*auto simp*: *isMinim-def*)

lemma *minim-inField*:
assumes $B \leq Field\ r$ and $B \neq \{\}$
shows *minim B* ∈ *Field r*
proof−
have *minim B* ∈ *B* using *assms* by (*simp add*: *minim-in*)
thus *?thesis* using *assms* by *blast*
qed

lemma *minim-least*:
assumes *SUB*: $B \leq Field\ r$ and *IN*: $b \in B$
shows (*minim B*, *b*) ∈ *r*
proof−
from *minim-isMinim*[*of B*] *assms*
have *isMinim B* (*minim B*) by *auto*
thus *?thesis* by (*auto simp add*: *isMinim-def IN*)
qed

lemma *equals-minim*:
assumes *SUB*: $B \leq Field\ r$ and *IN*: $a \in B$ and
LEAST: $\bigwedge b.\ b \in B \Longrightarrow (a,b) \in r$
shows $a = minim\ B$
proof−

from *minim-isMinim*[*of B*] *assms*
have *isMinim B* (*minim B*) **by** *auto*
moreover have *isMinim B a* **using** *IN LEAST isMinim-def* **by** *auto*
ultimately show *?thesis*
using *isMinim-unique* **by** *auto*
qed

27.3.4 Properties of successor

lemma *suc-AboveS*:
assumes *SUB*: $B \leq$ *Field r* **and** *ABOVES*: *AboveS B* $\neq \{\}$
shows *suc B* $\in$ *AboveS B*
proof(*unfold suc-def*)
have *AboveS B* $\leq$ *Field r*
using *AboveS-Field*[*of r*] **by** *auto*
thus *minim* (*AboveS B*) $\in$ *AboveS B*
using *assms* **by** (*simp add*: *minim-in*)
qed

lemma *suc-greater*:
assumes *SUB*: $B \leq$ *Field r* **and** *ABOVES*: *AboveS B* $\neq \{\}$ **and** *IN*: $b \in B$
shows *suc B* $\neq b \wedge$ (*b*,*suc B*) $\in r$
using *IN AboveS-def*[*of r*] *assms suc-AboveS* **by** *auto*

lemma *suc-least-AboveS*:
assumes *ABOVES*: $a \in$ *AboveS B*
shows (*suc B*,*a*) $\in r$
using *assms minim-least AboveS-Field*[*of r*] **by** (*auto simp*: *suc-def*)

lemma *suc-inField*:
assumes $B \leq$ *Field r* **and** *AboveS B* $\neq \{\}$
shows *suc B* $\in$ *Field r*
using *suc-AboveS assms AboveS-Field*[*of r*] **by** *auto*

lemma *equals-suc-AboveS*:
assumes $B \leq$ *Field r* **and** $a \in$ *AboveS B* **and** $\bigwedge a'.\ a' \in$ *AboveS B* $\Longrightarrow$ (*a*,*a'*) $\in r$
shows $a =$ *suc B*
using *assms equals-minim AboveS-Field*[*of r B*] **by** (*auto simp*: *suc-def*)

lemma *suc-underS*:
assumes *IN*: $a \in$ *Field r*
shows $a =$ *suc* (*underS a*)
proof−
have *underS a* $\leq$ *Field r*
using *underS-Field*[*of r*] **by** *auto*
moreover
have $a \in$ *AboveS* (*underS a*)
using *in-AboveS-underS IN* **by** *fast*

```
  moreover
  have ∀ a' ∈ AboveS (underS a). (a,a') ∈ r
  proof(clarify)
    fix a'
    assume ∗: a' ∈ AboveS (underS a)
    hence ∗∗: a' ∈ Field r
      using AboveS-Field by fast
    {assume (a,a') ∉ r
      hence a' = a ∨ (a',a) ∈ r
        using TOTAL IN ∗∗ by (auto simp add: total-on-def)
      moreover
      {assume a' = a
        hence (a,a') ∈ r
          using REFL IN ∗∗ by (auto simp add: refl-on-def)
      }
      moreover
      {assume a' ≠ a ∧ (a',a) ∈ r
        hence a' ∈ underS a
          unfolding underS-def by simp
        hence a' ∉ AboveS (underS a)
          using AboveS-disjoint by fast
        with ∗ have False by simp
      }
      ultimately have (a,a') ∈ r by blast
    }
    thus  (a, a') ∈ r by blast
  qed
  ultimately show ?thesis
    using equals-suc-AboveS by auto
qed
```

27.3.5 Properties of order filters

```
lemma under-ofilter: ofilter (under a)
  using TRANS by (auto simp: ofilter-def under-def Field-iff trans-def)

lemma underS-ofilter: ofilter (underS a)
  unfolding ofilter-def underS-def under-def
proof safe
  fix b assume (a, b) ∈ r (b, a) ∈ r and DIFF: b ≠ a
  thus False
    using ANTISYM antisym-def[of r] by blast
next
  fix b x
  assume (b,a) ∈ r b ≠ a (x,b) ∈ r
  thus (x,a) ∈ r
    using TRANS trans-def[of r] by blast
next
  fix x
```

```
  assume x ≠ a and (x, a) ∈ r
  then show x ∈ Field r
    unfolding Field-def
    by auto
qed

lemma Field-ofilter:
  ofilter (Field r)
  by(unfold ofilter-def under-def, auto simp add: Field-def)

lemma ofilter-underS-Field:
  ofilter A = ((∃ a ∈ Field r. A = underS a) ∨ (A = Field r))
proof
  assume (∃ a∈Field r. A = underS a) ∨ A = Field r
  thus ofilter A
    by (auto simp: underS-ofilter Field-ofilter)
next
  assume ∗: ofilter A
  let ?One = (∃ a∈Field r. A = underS a)
  let ?Two = (A = Field r)
  show ?One ∨ ?Two
  proof(cases ?Two)
    let ?B = (Field r) − A
    let ?a = minim ?B
    assume A ≠ Field r
    moreover have A ≤ Field r using ∗ ofilter-def by simp
    ultimately have 1: ?B ≠ {} by blast
    hence 2: ?a ∈ Field r using minim-inField[of ?B] by blast
    have 3: ?a ∈ ?B using minim-in[of ?B] 1 by blast
    hence 4: ?a ∉ A by blast
    have 5: A ≤ Field r using ∗ ofilter-def by auto

    moreover
    have A = underS ?a
    proof
      show A ≤ underS ?a
      proof
        fix x assume ∗∗: x ∈ A
        hence 11: x ∈ Field r using 5 by auto
        have 12: x ≠ ?a using 4 ∗∗ by auto
        have 13: under x ≤ A using ∗ ofilter-def ∗∗ by auto
        {assume (x,?a) ∉ r
          hence (?a,x) ∈ r
            using TOTAL total-on-def[of Field r r]
              2 4 11 12 by auto
          hence ?a ∈ under x using under-def[of r] by auto
          hence ?a ∈ A using ∗∗ 13 by blast
          with 4 have False by simp
        }
```

```
      then have (x,?a) ∈ r by blast
      thus x ∈ underS ?a
        unfolding underS-def by (auto simp add: 12)
    qed
  next
    show underS ?a ≤ A
    proof
      fix x
      assume **: x ∈ underS ?a
      hence 11: x ∈ Field r
        using Field-def unfolding underS-def by fastforce
      {assume x ∉ A
        hence x ∈ ?B using 11 by auto
        hence (?a,x) ∈ r using 3 minim-least[of ?B x] by blast
        hence False
          using ANTISYM antisym-def[of r] ** unfolding underS-def by auto
      }
      thus x ∈ A by blast
    qed
  qed
  ultimately have ?One using 2 by blast
  thus ?thesis by simp
 next
  assume A = Field r
  then show ?thesis
    by simp
 qed
qed

lemma ofilter-UNION:
 (⋀ i. i ∈ I ⟹ ofilter(A i)) ⟹ ofilter (⋃ i ∈ I. A i)
 unfolding ofilter-def by blast

lemma ofilter-under-UNION:
 assumes ofilter A
 shows A = (⋃ a ∈ A. under a)
proof
 have ∀ a ∈ A. under a ≤ A
   using assms ofilter-def by auto
 thus (⋃ a ∈ A. under a) ≤ A by blast
next
 have ∀ a ∈ A. a ∈ under a
   using REFL Refl-under-in[of r] assms ofilter-def[of A] by blast
 thus A ≤ (⋃ a ∈ A. under a) by blast
qed
```

27.3.6 Other properties

lemma *ofilter-linord*:

assumes *OF1*: *ofilter A* **and** *OF2*: *ofilter B*
shows $A \leq B \vee B \leq A$
proof(*cases* $A = Field\ r$)
assume *Case1*: $A = Field\ r$
hence $B \leq A$ **using** *OF2 ofilter-def* **by** *auto*
thus *?thesis* **by** *simp*
next
assume *Case2*: $A \neq Field\ r$
with *ofilter-underS-Field OF1* **obtain** a **where**
1: $a \in Field\ r \wedge A = underS\ a$ **by** *auto*
show *?thesis*
proof(*cases* $B = Field\ r$)
assume *Case21*: $B = Field\ r$
hence $A \leq B$ **using** *OF1 ofilter-def* **by** *auto*
thus *?thesis* **by** *simp*
next
assume *Case22*: $B \neq Field\ r$
with *ofilter-underS-Field OF2* **obtain** b **where**
2: $b \in Field\ r \wedge B = underS\ b$ **by** *auto*
have $a = b \vee (a,b) \in r \vee (b,a) \in r$
using *1 2 TOTAL total-on-def*[*of* - r] **by** *auto*
moreover
{**assume** $a = b$ **with** *1 2* **have** *?thesis* **by** *auto*
}
moreover
{**assume** $(a,b) \in r$
with *underS-incr*[*of* r] *TRANS ANTISYM 1 2*
have $A \leq B$ **by** *auto*
hence *?thesis* **by** *auto*
}
moreover
{**assume** $(b,a) \in r$
with *underS-incr*[*of* r] *TRANS ANTISYM 1 2*
have $B \leq A$ **by** *auto*
hence *?thesis* **by** *auto*
}
ultimately show *?thesis* **by** *blast*
qed
qed

lemma *ofilter-AboveS-Field*:
assumes *ofilter A*
shows $A \cup (AboveS\ A) = Field\ r$
proof
show $A \cup (AboveS\ A) \leq Field\ r$
using *assms ofilter-def AboveS-Field*[*of* r] **by** *auto*
next
{**fix** x **assume** $*$: $x \in Field\ r$ **and** $**$: $x \notin A$
{**fix** y **assume** $***$: $y \in A$

```
    with ∗∗ have 1: y ≠ x by auto
    {assume (y,x) ∉ r
     moreover
     have y ∈ Field r using assms ofilter-def ∗∗∗ by auto
     ultimately have (x,y) ∈ r
      using 1 ∗ TOTAL total-on-def[of - r] by auto
     with ∗∗∗ assms ofilter-def under-def[of r] have x ∈ A by auto
     with ∗∗ have False by contradiction
    }
    hence (y,x) ∈ r by blast
    with 1 have y ≠ x ∧ (y,x) ∈ r by auto
   }
   with ∗ have x ∈ AboveS A unfolding AboveS-def by auto
  }
  thus Field r ≤ A ∪ (AboveS A) by blast
qed

lemma suc-ofilter-in:
  assumes OF: ofilter A and ABOVE-NE: AboveS A ≠ {} and
    REL: (b,suc A) ∈ r and DIFF: b ≠ suc A
  shows b ∈ A
proof−
  have ∗: suc A ∈ Field r ∧ b ∈ Field r
    using WELL REL well-order-on-domain[of Field r] by auto
  {assume ∗∗: b ∉ A
    hence b ∈ AboveS A
      using OF ∗ ofilter-AboveS-Field by auto
    hence (suc A, b) ∈ r
      using suc-least-AboveS by auto
    hence False using REL DIFF ANTISYM ∗
      by (auto simp add: antisym-def)
  }
  thus ?thesis by blast
qed

end

end
```

28 Well-Order Embeddings as Needed by Bounded Natural Functors

```
theory BNF-Wellorder-Embedding
  imports Hilbert-Choice BNF-Wellorder-Relation
begin
```

In this section, we introduce well-order *embeddings* and *isomorphisms* and prove their basic properties. The notion of embedding is considered from the

point of view of the theory of ordinals, and therefore requires the source to be injected as an *initial segment* (i.e., *order filter*) of the target. A main result of this section is the existence of embeddings (in one direction or another) between any two well-orders, having as a consequence the fact that, given any two sets on any two types, one is smaller than (i.e., can be injected into) the other.

28.1 Auxiliaries

lemma *UNION-inj-on-ofilter*:
assumes *WELL*: *Well-order r* **and**
OF: $\bigwedge i.\ i \in I \Longrightarrow$ *wo-rel.ofilter r* (*A i*) **and**
INJ: $\bigwedge i.\ i \in I \Longrightarrow$ *inj-on f* (*A i*)
shows *inj-on f* ($\bigcup i \in I.\ A\ i$)
proof−
have *wo-rel r* **using** *WELL* **by** (*simp add*: *wo-rel-def*)
hence $\bigwedge i\ j.\ [\![i \in I; j \in I]\!] \Longrightarrow A\ i <= A\ j \vee A\ j <= A\ i$
using *wo-rel.ofilter-linord*[*of r*] *OF* **by** *blast*
with *WELL INJ* **show** *?thesis*
by (*auto simp add*: *inj-on-UNION-chain*)
qed

lemma *under-underS-bij-betw*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
IN: *a* ∈ *Field r* **and** *IN'*: *f a* ∈ *Field r'* **and**
BIJ: *bij-betw f* (*underS r a*) (*underS r'* (*f a*))
shows *bij-betw f* (*under r a*) (*under r'* (*f a*))
proof−
have *a* ∉ *underS r a* ∧ *f a* ∉ *underS r'* (*f a*)
unfolding *underS-def* **by** *auto*
moreover
{have *Refl r* ∧ *Refl r'* **using** *WELL WELL'*
by (*auto simp add*: *order-on-defs*)
hence *under r a* = *underS r a* ∪ {*a*} ∧
under r' (*f a*) = *underS r'* (*f a*) ∪ {*f a*}
using *IN IN'* **by**(*auto simp add*: *Refl-under-underS*)
}
ultimately show *?thesis*
using *BIJ notIn-Un-bij-betw*[*of a underS r a f underS r'* (*f a*)] **by** *auto*
qed

28.2 (Well-order) embeddings, strict embeddings, isomorphisms and order-compatible functions

Standardly, a function is an embedding of a well-order in another if it injectively and order-compatibly maps the former into an order filter of the latter. Here we opt for a more succinct definition (operator *embed*), asking that, for any element in the source, the function should be a bijection

between the set of strict lower bounds of that element and the set of strict lower bounds of its image. (Later we prove equivalence with the standard definition – lemma *embed-iff-compat-inj-on-ofilter*.) A *strict embedding* (operator *embedS*) is a non-bijective embedding and an isomorphism (operator *iso*) is a bijective embedding.

definition *embed* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
$embed\ r\ r'\ f \equiv \forall\, a \in Field\ r.\ bij\text{-}betw\ f\ (under\ r\ a)\ (under\ r'\ (f\ a))$

lemmas *embed-defs* = *embed-def embed-def*[*abs-def*]

Strict embeddings:

definition *embedS* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
$embedS\ r\ r'\ f \equiv embed\ r\ r'\ f \wedge \neg\ bij\text{-}betw\ f\ (Field\ r)\ (Field\ r')$

lemmas *embedS-defs* = *embedS-def embedS-def*[*abs-def*]

definition *iso* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
$iso\ r\ r'\ f \equiv embed\ r\ r'\ f \wedge bij\text{-}betw\ f\ (Field\ r)\ (Field\ r')$

lemmas *iso-defs* = *iso-def iso-def*[*abs-def*]

definition *compat* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow ('a \Rightarrow 'a') \Rightarrow bool$
where
$compat\ r\ r'\ f \equiv \forall\, a\ b.\ (a,b) \in r \longrightarrow (f\ a,\ f\ b) \in r'$

lemma *compat-wf*:
assumes *CMP*: *compat r r' f* **and** *WF*: *wf r'*
shows *wf r*
proof−
have $r \leq inv\text{-}image\ r'\ f$
unfolding *inv-image-def* **using** *CMP*
by (*auto simp add*: *compat-def*)
with *WF* **show** *?thesis*
using *wf-inv-image*[*of r' f*] *wf-subset*[*of inv-image r' f*] **by** *auto*
qed

lemma *id-embed*: *embed r r id*
by(*auto simp add*: *id-def embed-def bij-betw-def*)

lemma *id-iso*: *iso r r id*
by(*auto simp add*: *id-def embed-def iso-def bij-betw-def*)

lemma *embed-compat*:
assumes *EMB*: *embed r r' f*
shows *compat r r' f*

unfolding *compat-def*
proof *clarify*
fix *a b*
assume ∗: $(a,b) \in r$
hence *1*: $b \in Field\ r$ **using** *Field-def*[*of r*] **by** *blast*
have $a \in under\ r\ b$
using ∗ *under-def*[*of r*] **by** *simp*
hence $f\ a \in under\ r'\ (f\ b)$
using *EMB embed-def*[*of r r' f*]
bij-betw-def[*of f under r b under r' (f b)*]
image-def[*of f under r b*] *1* **by** *auto*
thus $(f\ a,\ f\ b) \in r'$
by (*auto simp add*: *under-def*)
qed

lemma *embed-in-Field*:
assumes *EMB*: *embed r r' f* **and** *IN*: $a \in Field\ r$
shows $f\ a \in Field\ r'$
proof −
have $a \in Domain\ r \vee a \in Range\ r$
using *IN* **unfolding** *Field-def* **by** *blast*
then show *?thesis*
using *embed-compat* [*OF EMB*]
unfolding *Field-def compat-def* **by** *force*
qed

lemma *comp-embed*:
assumes *EMB*: *embed r r' f* **and** *EMB'*: *embed r' r'' f'*
shows *embed r r''* $(f' \circ f)$
proof(*unfold embed-def*, *auto*)
fix *a* **assume** ∗: $a \in Field\ r$
hence *bij-betw f* (*under r a*) (*under r'* (*f a*))
using *embed-def*[*of r*] *EMB* **by** *auto*
moreover
{have $f\ a \in Field\ r'$
using *EMB* ∗ **by** (*auto simp add*: *embed-in-Field*)
hence *bij-betw f'* (*under r'* (*f a*)) (*under r''* (*f'* (*f a*)))
using *embed-def*[*of r'*] *EMB'* **by** *auto*
}
ultimately
show *bij-betw* $(f' \circ f)$ (*under r a*) (*under r''* (*f'*(*f a*)))
by(*auto simp add*: *bij-betw-trans*)
qed

lemma *comp-iso*:
assumes *EMB*: *iso r r' f* **and** *EMB'*: *iso r' r'' f'*
shows *iso r r''* $(f' \circ f)$
using *assms* **unfolding** *iso-def*
by (*auto simp add*: *comp-embed bij-betw-trans*)

That *embedS* is also preserved by function composition shall be proved only later.

lemma *embed-Field*: *embed r r′ f* $\Longrightarrow$ *f*‘(*Field r*) $\leq$ *Field r′*
by (*auto simp add*: *embed-in-Field*)

lemma *embed-preserves-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′* **and**
EMB: *embed r r′ f* **and** *OF*: *wo-rel.ofilter r A*
shows *wo-rel.ofilter r′* (*f*‘*A*)
proof−

from *WELL* **have** *Well*: *wo-rel r* **unfolding** *wo-rel-def* **.**
from *WELL′* **have** *Well′*: *wo-rel r′* **unfolding** *wo-rel-def* **.**
from *OF* **have** *0*: *A* $\leq$ *Field r* **by**(*auto simp add*: *Well wo-rel.ofilter-def*)

show *?thesis* **using** *Well′ WELL EMB 0 embed-Field*[*of r r′ f*]
proof(*unfold wo-rel.ofilter-def*, *auto simp add*: *image-def*)
fix *a b′*
assume ∗: *a* $\in$ *A* **and** ∗∗: *b′* $\in$ *under r′* (*f a*)
hence *a* $\in$ *Field r* **using** *0* **by** *auto*
hence *bij-betw f* (*under r a*) (*under r′* (*f a*))
using ∗ *EMB* **by** (*auto simp add*: *embed-def*)
hence *f*‘(*under r a*) = *under r′* (*f a*)
by (*simp add*: *bij-betw-def*)
with ∗∗ *image-def*[*of f under r a*] **obtain** *b* **where**
1: *b* $\in$ *under r a* $\wedge$ *b′* = *f b* **by** *blast*
hence *b* $\in$ *A* **using** *Well* ∗ *OF*
by (*auto simp add*: *wo-rel.ofilter-def*)
with *1* **show** $\exists$ *b* $\in$ *A*. *b′* = *f b* **by** *blast*
qed
qed

lemma *embed-Field-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′* **and**
EMB: *embed r r′ f*
shows *wo-rel.ofilter r′* (*f*‘(*Field r*))
proof−
have *wo-rel.ofilter r* (*Field r*)
using *WELL* **by** (*auto simp add*: *wo-rel-def wo-rel.Field-ofilter*)
with *WELL WELL′ EMB*
show *?thesis* **by** (*auto simp add*: *embed-preserves-ofilter*)
qed

lemma *embed-inj-on*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embed r r′ f*
shows *inj-on f* (*Field r*)
proof(*unfold inj-on-def*, *clarify*)

from *WELL* **have** *Well*: *wo-rel r* **unfolding** *wo-rel-def* **.**

```
  with wo-rel.TOTAL[of r]
  have Total: Total r by simp
  from Well wo-rel.REFL[of r]
  have Refl: Refl r by simp

  fix a b
  assume ∗: a ∈ Field r and ∗∗: b ∈ Field r and
    ∗∗∗: f a = f b
  hence 1: a ∈ Field r ∧ b ∈ Field r
    unfolding Field-def by auto
  {assume (a,b) ∈ r
    hence a ∈ under r b ∧ b ∈ under r b
      using Refl by (auto simp add: under-def refl-on-def Field-def)
    hence a = b
      using EMB 1 ∗∗∗
      by (auto simp add: embed-def bij-betw-def inj-on-def)
  }
  moreover
  {assume (b,a) ∈ r
    hence a ∈ under r a ∧ b ∈ under r a
      using Refl by (auto simp add: under-def refl-on-def Field-def)
    hence a = b
      using EMB 1 ∗∗∗
      by (auto simp add: embed-def bij-betw-def inj-on-def)
  }
  ultimately
  show a = b using Total 1
    by (auto simp add: total-on-def)
qed

lemma embed-underS:
  assumes WELL: Well-order r and
    EMB: embed r r' f and IN: a ∈ Field r
  shows bij-betw f (underS r a) (underS r' (f a))
proof−
  have f a ∈ Field r' using assms embed-Field[of r r' f] by auto
  then have 0: under r a = underS r a ∪ {a}
    by (simp add: IN Refl-under-underS WELL wo-rel.REFL wo-rel.intro)
  moreover have 1: bij-betw f (under r a) (under r' (f a))
    using assms by (auto simp add: embed-def)
  moreover have under r' (f a) = underS r' (f a) ∪ {f a}
  proof
    show under r' (f a) ⊆ underS r' (f a) ∪ {f a}
      using underS-def under-def by fastforce
    show underS r' (f a) ∪ {f a} ⊆ under r' (f a)
      using bij-betwE 0 1 underS-subset-under by fastforce
  qed
  moreover have a ∉ underS r a ∧ f a ∉ underS r' (f a)
    unfolding underS-def by blast
```

ultimately show *?thesis*
by (*auto simp add*: *notIn-Un-bij-betw3*)
qed

lemma *embed-iff-compat-inj-on-ofilter*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows *embed r r' f* = (*compat r r' f* ∧ *inj-on f* (*Field r*) ∧ *wo-rel.ofilter r'* (*f*`(*Field r*)))
using *assms*
proof(*auto simp add*: *embed-compat embed-inj-on embed-Field-ofilter*,
unfold embed-def, *auto*)
fix *a*
assume ∗: *inj-on f* (*Field r*) **and**
∗∗: *compat r r' f* **and**
∗∗∗: *wo-rel.ofilter r'* (*f*`(*Field r*)) **and**
∗∗∗∗: *a* ∈ *Field r*

have *Well*: *wo-rel r*
using *WELL wo-rel-def*[*of r*] **by** *simp*
hence *Refl*: *Refl r*
using *wo-rel.REFL*[*of r*] **by** *simp*
have *Total*: *Total r*
using *Well wo-rel.TOTAL*[*of r*] **by** *simp*
have *Well'*: *wo-rel r'*
using *WELL' wo-rel-def*[*of r'*] **by** *simp*
hence *Antisym'*: *antisym r'*
using *wo-rel.ANTISYM*[*of r'*] **by** *simp*
have (*a*,*a*) ∈ *r*
using ∗∗∗∗ *Well wo-rel.REFL*[*of r*]
refl-on-def[*of* - *r*] **by** *auto*
hence (*f a*, *f a*) ∈ *r'*
using ∗∗ **by**(*auto simp add*: *compat-def*)
hence *0*: *f a* ∈ *Field r'*
unfolding *Field-def* **by** *auto*
have *f a* ∈ *f*`(*Field r*)
using ∗∗∗∗ **by** *auto*
hence *2*: *under r'* (*f a*) ≤ *f*`(*Field r*)
using *Well'* ∗∗∗ *wo-rel.ofilter-def*[*of r' f*`(*Field r*)] **by** *fastforce*

show *bij-betw f* (*under r a*) (*under r'* (*f a*))
proof(*unfold bij-betw-def*, *auto*)
show *inj-on f* (*under r a*) **by** (*rule inj-on-subset*[*OF* ∗ *under-Field*])
next
fix *b* **assume** *b* ∈ *under r a*
thus *f b* ∈ *under r'* (*f a*)
unfolding *under-def* **using** ∗∗
by (*auto simp add*: *compat-def*)
next
fix *b'* **assume** ∗∗∗∗∗: *b'* ∈ *under r'* (*f a*)

```
    hence b' ∈ f`(Field r)
      using 2 by auto
    with Field-def[of r] obtain b where
      3: b ∈ Field r and 4: b' = f b by auto
    have (b,a) ∈ r
    proof−
      {assume (a,b) ∈ r
        with ** 4 have (f a, b') ∈ r'
          by (auto simp add: compat-def)
        with ***** Antisym' have f a = b'
          by(auto simp add: under-def antisym-def)
        with 3 **** 4 * have a = b
          by(auto simp add: inj-on-def)
      }
      moreover
      {assume a = b
        hence (b,a) ∈ r using Refl **** 3
          by (auto simp add: refl-on-def)
      }
      ultimately
      show ?thesis using Total **** 3 by (fastforce simp add: total-on-def)
    qed
    with 4 show  b' ∈ f`(under r a)
      unfolding under-def by auto
  qed
qed

lemma inv-into-ofilter-embed:
  assumes WELL: Well-order r and OF: wo-rel.ofilter r A and
    BIJ: ∀ b ∈ A. bij-betw f (under r b) (under r' (f b)) and
    IMAGE: f ` A = Field r'
  shows embed r' r (inv-into A f)
proof−

  have Well: wo-rel r
    using WELL wo-rel-def[of r] by simp
  have Refl: Refl r
    using Well wo-rel.REFL[of r] by simp
  have Total: Total r
    using Well wo-rel.TOTAL[of r] by simp

  have 1: bij-betw f A (Field r')
  proof(unfold bij-betw-def inj-on-def, auto simp add: IMAGE)
    fix b1 b2
    assume *: b1 ∈ A and **: b2 ∈ A and
      ***: f b1 = f b2
    have 11: b1 ∈ Field r ∧ b2 ∈ Field r
      using * ** Well OF by (auto simp add: wo-rel.ofilter-def)
    moreover
```

```
  {assume (b1,b2) ∈ r
   hence b1 ∈ under r b2 ∧ b2 ∈ under r b2
     unfolding under-def using 11 Refl
     by (auto simp add: refl-on-def)
   hence b1 = b2 using BIJ * ** ***
     by (simp add: bij-betw-def inj-on-def)
  }
  moreover
  {assume (b2,b1) ∈ r
   hence b1 ∈ under r b1 ∧ b2 ∈ under r b1
     unfolding under-def using 11 Refl
     by (auto simp add: refl-on-def)
   hence b1 = b2 using BIJ * ** ***
     by (simp add: bij-betw-def inj-on-def)
  }
  ultimately
  show b1 = b2
    using Total by (auto simp add: total-on-def)
qed

let ?f' = (inv-into A f)

have 2: ∀ b ∈ A. bij-betw ?f' (under r' (f b)) (under r b)
proof(clarify)
  fix b assume *: b ∈ A
  hence under r b ≤ A
    using Well OF by(auto simp add: wo-rel.ofilter-def)
  moreover
  have f ‘ (under r b) = under r' (f b)
    using * BIJ by (auto simp add: bij-betw-def)
  ultimately
  show bij-betw ?f' (under r' (f b)) (under r b)
    using 1 by (auto simp add: bij-betw-inv-into-subset)
qed

have 3: ∀ b' ∈ Field r'. bij-betw ?f' (under r' b') (under r (?f' b'))
proof(clarify)
  fix b' assume *: b' ∈ Field r'
  have b' = f (?f' b') using * 1
    by (auto simp add: bij-betw-inv-into-right)
  moreover
  {obtain b where 31: b ∈ A and f b = b' using IMAGE * by force
   hence ?f' b' = b using 1 by (auto simp add: bij-betw-inv-into-left)
   with 31 have ?f' b' ∈ A by auto
  }
  ultimately
  show  bij-betw ?f' (under r' b') (under r (?f' b'))
    using 2 by auto
qed
```

thus *?thesis* **unfolding** *embed-def* **.**
qed

lemma *inv-into-underS-embed*:
assumes *WELL*: *Well-order r* **and**
BIJ: $\forall\, b \in$ *underS r a*. *bij-betw f* (*under r b*) (*under r′* (*f b*)) **and**
IN: *a* $\in$ *Field r* **and**
IMAGE: *f* ‘ (*underS r a*) = *Field r′*
shows *embed r′ r* (*inv-into* (*underS r a*) *f*)
using *assms*
by(*auto simp add*: *wo-rel-def wo-rel.underS-ofilter inv-into-ofilter-embed*)

lemma *inv-into-Field-embed*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embed r r′ f* **and**
IMAGE: *Field r′* $\leq$ *f* ‘ (*Field r*)
shows *embed r′ r* (*inv-into* (*Field r*) *f*)
proof−
have ($\forall\, b \in$ *Field r*. *bij-betw f* (*under r b*) (*under r′* (*f b*)))
using *EMB* **by** (*auto simp add*: *embed-def*)
moreover
have *f* ‘ (*Field r*) $\leq$ *Field r′*
using *EMB WELL* **by** (*auto simp add*: *embed-Field*)
ultimately
show *?thesis* **using** *assms*
by(*auto simp add*: *wo-rel-def wo-rel.Field-ofilter inv-into-ofilter-embed*)
qed

lemma *inv-into-Field-embed-bij-betw*:
assumes *EMB*: *embed r r′ f* **and** *BIJ*: *bij-betw f* (*Field r*) (*Field r′*)
shows *embed r′ r* (*inv-into* (*Field r*) *f*)
proof−
have *Field r′* $\leq$ *f* ‘ (*Field r*)
using *BIJ* **by** (*auto simp add*: *bij-betw-def*)
then have *iso*: *iso r r′ f*
by (*simp add*: *BIJ EMB iso-def*)
have $*$: $\forall\, a.\ a \in$ *Field r* $\longrightarrow$ *bij-betw f* (*under r a*) (*under r′* (*f a*))
using *EMB embed-def* **by** *fastforce*
show *?thesis*
proof (*clarsimp simp add*: *embed-def*)
fix *a*
assume *a*: *a* $\in$ *Field r′*
then have *ar*: *a* $\in$ *f* ‘ *Field r*
using *BIJ bij-betw-imp-surj-on* **by** *blast*
have [*simp*]: *f* (*inv-into* (*Field r*) *f a*) = *a*
by (*simp add*: *ar f-inv-into-f*)
show *bij-betw* (*inv-into* (*Field r*) *f*) (*under r′ a*) (*under r* (*inv-into* (*Field r*) *f* *a*))
proof (*rule bij-betw-inv-into-subset* [*OF BIJ*])

```
    show under r (inv-into (Field r) f a) ⊆ Field r
      by (simp add: under-Field)
    have inv-into (Field r) f a ∈ Field r
      by (simp add: ar inv-into-into)
    then show f ‘ under r (inv-into (Field r) f a) = under r′ a
      using bij-betw-imp-surj-on ∗ by fastforce
  qed
 qed
qed
```

28.3 Given any two well-orders, one can be embedded in the other

Here is an overview of the proof of of this fact, stated in theorem *wellorders-totally-ordered*:

Fix the well-orders $r::'a\ rel$ and $r'::'a'\ rel$. Attempt to define an embedding $f::'a \Rightarrow 'a'$ from r to r' in the natural way by well-order recursion ("hoping" that *Field* r turns out to be smaller than *Field* r'), but also record, at the recursive step, in a function $g::'a \Rightarrow bool$, the extra information of whether *Field* r' gets exhausted or not.

If *Field* r' does not get exhausted, then *Field* r is indeed smaller and f is the desired embedding from r to r' (lemma *wellorders-totally-ordered-aux*).

Otherwise, it means that *Field* r' is the smaller one, and the inverse of (the "good" segment of) f is the desired embedding from r' to r (lemma *wellorders-totally-ordered-aux2*).

```
lemma wellorders-totally-ordered-aux:
 fixes r ::'a rel  and r'::'a' rel and
   f :: 'a ⇒ 'a' and a::'a
 assumes WELL: Well-order r and WELL': Well-order r' and IN: a ∈ Field r
and
   IH: ∀ b ∈ underS r a. bij-betw f (under r b) (under r' (f b)) and
   NOT: f ‘ (underS r a) ≠ Field r' and SUC: f a = wo-rel.suc r' (f‘(underS r
a))
 shows bij-betw f (under r a) (under r' (f a))
proof−

 have Well: wo-rel r using WELL unfolding wo-rel-def .
 hence Refl: Refl r using wo-rel.REFL[of r] by auto
 have Trans: trans r using Well wo-rel.TRANS[of r] by auto
 have Well': wo-rel r' using WELL' unfolding wo-rel-def .
 have OF: wo-rel.ofilter r (underS r a)
   by (auto simp add: Well wo-rel.underS-ofilter)
 hence UN: underS r a = (⋃ b ∈ underS r a. under r b)
   using Well wo-rel.ofilter-under-UNION[of r underS r a] by blast

 {fix b assume ∗: b ∈ underS r a
   hence t0: (b,a) ∈ r ∧ b ≠ a unfolding underS-def by auto
   have t1: b ∈ Field r
```

```
    using ∗ underS-Field[of r a] by auto
  have t2: f‘(under r b) = under r′ (f b)
    using IH ∗ by (auto simp add: bij-betw-def)
  hence t3: wo-rel.ofilter r′ (f‘(under r b))
    using Well′ by (auto simp add: wo-rel.under-ofilter)
  have f‘(under r b) ≤ Field r′
    using t2 by (auto simp add: under-Field)
  moreover
  have b ∈ under r b
    using t1 by(auto simp add: Refl Refl-under-in)
  ultimately
  have t4: f b ∈ Field r′ by auto
  have f‘(under r b) = under r′ (f b) ∧
       wo-rel.ofilter r′ (f‘(under r b)) ∧
       f b ∈ Field r′
    using t2 t3 t4 by auto
}
hence bFact:
  ∀ b ∈ underS r a. f‘(under r b) = under r′ (f b) ∧
                    wo-rel.ofilter r′ (f‘(under r b)) ∧
                    f b ∈ Field r′ by blast

have subField: f‘(underS r a) ≤ Field r′
  using bFact by blast

have OF′: wo-rel.ofilter r′ (f‘(underS r a))
proof−
  have f‘(underS r a) = f‘(⋃ b ∈ underS r a. under r b)
    using UN by auto
  also have ... = (⋃ b ∈ underS r a. f‘(under r b)) by blast
  also have ... = (⋃ b ∈ underS r a. (under r′ (f b)))
    using bFact by auto
  finally
  have f‘(underS r a) = (⋃ b ∈ underS r a. (under r′ (f b))) .
  thus ?thesis
    using Well′ bFact
      wo-rel.ofilter-UNION[of r′ underS r a λ b. under r′ (f b)] by fastforce
qed

have f‘(underS r a) ∪ AboveS r′ (f‘(underS r a)) = Field r′
  using Well′ OF′ by (auto simp add: wo-rel.ofilter-AboveS-Field)
hence NE: AboveS r′ (f‘(underS r a)) ≠ {}
  using subField NOT by blast

have INCL1: f‘(underS r a) ≤ underS r′ (f a)
proof(auto)
  fix b assume ∗: b ∈ underS r a
  have f b ≠ f a ∧ (f b, f a) ∈ r′
    using subField Well′ SUC NE ∗
```

```
      wo-rel.suc-greater[of r' f`(underS r a) f b] by force
    thus f b ∈ underS r' (f a)
      unfolding underS-def by simp
  qed

  have INCL2: underS r' (f a) ≤ f`(underS r a)
  proof
    fix b' assume b' ∈ underS r' (f a)
    hence b' ≠ f a ∧ (b', f a) ∈ r'
      unfolding underS-def by simp
    thus b' ∈ f`(underS r a)
      using Well' SUC NE OF'
        wo-rel.suc-ofilter-in[of r' f ` underS r a b'] by auto
  qed

  have INJ: inj-on f (underS r a)
  proof−
    have ∀ b ∈ underS r a. inj-on f (under r b)
      using IH by (auto simp add: bij-betw-def)
    moreover
    have ∀ b. wo-rel.ofilter r (under r b)
      using Well by (auto simp add: wo-rel.under-ofilter)
    ultimately show ?thesis
      using WELL bFact UN
        UNION-inj-on-ofilter[of r underS r a λb. under r b f]
      by auto
  qed

  have BIJ: bij-betw f (underS r a) (underS r' (f a))
    unfolding bij-betw-def
    using INJ INCL1 INCL2 by auto

  have f a ∈ Field r'
    using Well' subField NE SUC
    by (auto simp add: wo-rel.suc-inField)
  thus ?thesis
    using WELL WELL' IN BIJ under-underS-bij-betw[of r r' a f] by auto
qed

lemma wellorders-totally-ordered-aux2:
  fixes r ::'a rel and r'::'a' rel and
    f :: 'a ⇒ 'a' and g :: 'a ⇒ bool and a::'a
  assumes WELL: Well-order r and WELL': Well-order r' and
    MAIN1:
    ⋀ a. (False ∉ g`(underS r a) ∧ f`(underS r a) ≠ Field r'
          ⟶ f a = wo-rel.suc r' (f`(underS r a)) ∧ g a = True)
        ∧
        (¬(False ∉ (g`(underS r a)) ∧ f`(underS r a) ≠ Field r')
         ⟶ g a = False) and
```

```
    MAIN2: ⋀ a. a ∈ Field r ∧ False ∉ g‘(under r a) ⟶
           bij-betw f (under r a) (under r' (f a)) and
    Case: a ∈ Field r ∧ False ∈ g‘(under r a)
  shows ∃f'. embed r' r f'
proof−
  have Well: wo-rel r using WELL unfolding wo-rel-def .
  hence Refl: Refl r using wo-rel.REFL[of r] by auto
  have Trans: trans r using Well wo-rel.TRANS[of r] by auto
  have Antisym: antisym r using Well wo-rel.ANTISYM[of r] by auto
  have Well': wo-rel r' using WELL' unfolding wo-rel-def .

  have 0: under r a = underS r a ∪ {a}
    using Refl Case by(auto simp add: Refl-under-underS)

  have 1: g a = False
  proof−
    {assume g a ≠ False
     with 0 Case have False ∈ g‘(underS r a) by blast
     with MAIN1 have g a = False by blast}
    thus ?thesis by blast
  qed
  let ?A = {a ∈ Field r. g a = False}
  let ?a = (wo-rel.minim r ?A)

  have 2: ?A ≠ {} ∧ ?A ≤ Field r using Case 1 by blast

  have 3: False ∉ g‘(underS r ?a)
  proof
    assume False ∈ g‘(underS r ?a)
    then obtain b where b ∈ underS r ?a and 31: g b = False by auto
    hence 32: (b,?a) ∈ r ∧ b ≠ ?a
      by (auto simp add: underS-def)
    hence b ∈ Field r unfolding Field-def by auto
    with 31 have b ∈ ?A by auto
    hence (?a,b) ∈ r using wo-rel.minim-least 2 Well by fastforce

    with 32 Antisym show False
      by (auto simp add: antisym-def)
  qed
  have temp: ?a ∈ ?A
    using Well 2 wo-rel.minim-in[of r ?A] by auto
  hence 4: ?a ∈ Field r by auto

  have 5: g ?a = False using temp by blast

  have 6: f‘(underS r ?a) = Field r'
    using MAIN1[of ?a] 3 5 by blast

  have 7: ∀ b ∈ underS r ?a. bij-betw f (under r b) (under r' (f b))
```

proof
fix *b* **assume** *as*: $b \in underS\ r\ ?a$
moreover
have *wo-rel.ofilter r* (*underS r ?a*)
using *Well* **by** (*auto simp add*: *wo-rel.underS-ofilter*)
ultimately
have $False \notin g`(under\ r\ b)$ **using** *3 Well* **by** (*subst* (*asm*) *wo-rel.ofilter-def*) *fast+*
moreover have $b \in Field\ r$
unfolding *Field-def* **using** *as* **by** (*auto simp add*: *underS-def*)
ultimately
show *bij-betw f* (*under r b*) (*under r′* (*f b*))
using *MAIN2* **by** *auto*
qed

have *embed r′ r* (*inv-into* (*underS r ?a*) *f*)
using *WELL WELL′ 7 4 6 inv-into-underS-embed*[*of r ?a f r′*] **by** *auto*
thus *?thesis*
unfolding *embed-def* **by** *blast*
qed

theorem *wellorders-totally-ordered*:
fixes $r :: 'a\ rel$ **and** $r' :: 'a'\ rel$
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′*
shows $(\exists f.\ embed\ r\ r'\ f) \vee (\exists f'.\ embed\ r'\ r\ f')$
proof−

have *Well*: *wo-rel r* **using** *WELL* **unfolding** *wo-rel-def* **.**
hence *Refl*: *Refl r* **using** *wo-rel.REFL*[*of r*] **by** *auto*
have *Trans*: *trans r* **using** *Well wo-rel.TRANS*[*of r*] **by** *auto*
have *Well′*: *wo-rel r′* **using** *WELL′* **unfolding** *wo-rel-def* **.**

obtain *H* **where** *H-def*: $H =$
$(\lambda h\ a.\ if\ False \notin (snd \circ h)`(underS\ r\ a) \wedge (fst \circ h)`(underS\ r\ a) \neq Field\ r'$
$\quad then\ (wo\text{-}rel.suc\ r'\ ((fst \circ h)`(underS\ r\ a)),\ True)$
$\quad else\ (undefined,\ False))$ **by** *blast*
have *Adm*: *wo-rel.adm-wo r H*
using *Well*
proof(*unfold wo-rel.adm-wo-def*, *clarify*)
fix $h1 :: 'a \Rightarrow 'a' * bool$ **and** $h2 :: 'a \Rightarrow 'a' * bool$ **and** *x*
assume $\forall y \in underS\ r\ x.\ h1\ y = h2\ y$
hence $\forall y \in underS\ r\ x.\ (fst \circ h1)\ y = (fst \circ h2)\ y \wedge$
$(snd \circ h1)\ y = (snd \circ h2)\ y$ **by** *auto*
hence $(fst \circ h1)`(underS\ r\ x) = (fst \circ h2)`(underS\ r\ x) \wedge$
$(snd \circ h1)`(underS\ r\ x) = (snd \circ h2)`(underS\ r\ x)$
by (*auto simp add*: *image-def*)
thus *H h1 x* = *H h2 x* **by** (*simp add*: *H-def del*: *not-False-in-image-Ball*)
qed

obtain $h{::}'a \Rightarrow 'a' * bool$ **and** $f{::}'a \Rightarrow 'a'$ **and** $g{::}'a \Rightarrow bool$
  where *h-def*: $h = wo\text{-}rel.worec\ r\ H$ **and**
    f-def: $f = fst \circ h$ **and** *g-def*: $g = snd \circ h$ **by** *blast*
obtain *test* **where** *test-def*:
  $test = (\lambda\ a.\ False \notin (g`(underS\ r\ a)) \wedge f`(underS\ r\ a) \neq Field\ r')$ **by** *blast*

have $*$: $\bigwedge\ a.\ h\ a\ = H\ h\ a$
  using *Adm Well wo-rel.worec-fixpoint*[*of r H*] **by** (*simp add*: *h-def*)
have *Main1*:
  $\bigwedge\ a.\ (test\ a \longrightarrow f\ a = wo\text{-}rel.suc\ r'\ (f`(underS\ r\ a)) \wedge g\ a = True) \wedge$
      $(\neg(test\ a) \longrightarrow g\ a = False)$
proof−
  fix *a* **show** $(test\ a \longrightarrow f\ a = wo\text{-}rel.suc\ r'\ (f`(underS\ r\ a)) \wedge g\ a = True) \wedge$
          $(\neg(test\ a) \longrightarrow g\ a = False)$
    using $*$[*of a*] *test-def f-def g-def H-def* **by** *auto*
qed

let *?phi* $= \lambda\ a.\ a \in Field\ r \wedge False \notin g`(under\ r\ a) \longrightarrow$
              bij-betw f $(under\ r\ a)\ (under\ r'\ (f\ a))$
have *Main2*: $\bigwedge\ a.$ *?phi a*
proof−
  fix *a* **show** *?phi a*
  proof(*rule wo-rel.well-order-induct*[*of r ?phi*],
      simp only: *Well*, *clarify*)
    fix *a*
    assume *IH*: $\forall\ b.\ b \neq a \wedge (b,a) \in r \longrightarrow$ *?phi b* **and**
      $*$: $a \in Field\ r$ **and**
      $**$: $False \notin g`(under\ r\ a)$
    have *1*: $\forall\ b \in underS\ r\ a.$ *bij-betw f* $(under\ r\ b)\ (under\ r'\ (f\ b))$
    proof(*clarify*)
      fix *b* **assume** $***$: $b \in underS\ r\ a$
      hence *0*: $(b,a) \in r \wedge b \neq a$ **unfolding** *underS-def* **by** *auto*
      moreover have $b \in Field\ r$
        using $***$ *underS-Field*[*of r a*] **by** *auto*
      moreover have $False \notin g`(under\ r\ b)$
        using *0* $**$ *Trans under-incr*[*of r b a*] **by** *auto*
      ultimately show *bij-betw f* $(under\ r\ b)\ (under\ r'\ (f\ b))$
        using *IH* **by** *auto*
    qed

    have *21*: $False \notin g`(underS\ r\ a)$
      using $**$ *underS-subset-under*[*of r a*] **by** *auto*
    have *22*: $g`(under\ r\ a) \leq \{True\}$ **using** $**$ **by** *auto*
    moreover have *23*: $a \in under\ r\ a$
      using *Refl* $*$ **by** (*auto simp add*: *Refl-under-in*)
    ultimately have *24*: $g\ a = True$ **by** *blast*
    have *2*: $f`(underS\ r\ a) \neq Field\ r'$
    proof
      assume $f`(underS\ r\ a) = Field\ r'$

```
      hence g a = False using Main1 test-def by blast
      with 24 show False using ** by blast
    qed

    have 3: f a = wo-rel.suc r' (f`(underS r a))
      using 21 2 Main1 test-def by blast

    show bij-betw f (under r a) (under r' (f a))
      using WELL  WELL' 1 2 3 *
        wellorders-totally-ordered-aux[of r r' a f] by auto
  qed
 qed

 let ?chi = (λ a. a ∈ Field r ∧ False ∈ g`(under r a))
 show ?thesis
 proof(cases ∃ a. ?chi a)
  assume ¬ (∃ a. ?chi a)
  hence ∀ a ∈ Field r.  bij-betw f (under r a) (under r' (f a))
   using Main2 by blast
  thus ?thesis unfolding embed-def by blast
 next
  assume ∃ a. ?chi a
  then obtain a where ?chi a by blast
  hence ∃ f'. embed r r' f'
   using wellorders-totally-ordered-aux2[of r r' g f a]
    WELL WELL' Main1 Main2 test-def by fast
  thus ?thesis by blast
 qed
qed
```

28.4 Uniqueness of embeddings

Here we show a fact complementary to the one from the previous subsection – namely, that between any two well-orders there is *at most* one embedding, and is the one definable by the expected well-order recursive equation. As a consequence, any two embeddings of opposite directions are mutually inverse.

```
lemma embed-determined:
 assumes WELL: Well-order r and WELL': Well-order r' and
  EMB: embed r r' f and IN: a ∈ Field r
 shows f a = wo-rel.suc r' (f`(underS r a))
proof−
 have bij-betw f (underS r a) (underS r' (f a))
  using assms by (auto simp add: embed-underS)
 hence f`(underS r a) = underS r' (f a)
  by (auto simp add: bij-betw-def)
 moreover
 {have f a ∈ Field r' using IN
```

```
    using EMB WELL embed-Field[of r r' f] by auto
  hence f a = wo-rel.suc r' (underS r' (f a))
    using WELL' by (auto simp add: wo-rel-def wo-rel.suc-underS)
 }
 ultimately show ?thesis by simp
qed

lemma embed-unique:
 assumes WELL: Well-order r and WELL': Well-order r' and
   EMBf: embed r r' f and EMBg: embed r r' g
 shows a ∈ Field r ⟶ f a = g a
proof(rule wo-rel.well-order-induct[of r], auto simp add: WELL wo-rel-def)
 fix a
 assume IH: ∀ b. b ≠ a ∧ (b,a) ∈ r ⟶ b ∈ Field r ⟶ f b = g b and
   ∗: a ∈ Field r
 hence ∀ b ∈ underS r a. f b = g b
   unfolding underS-def by (auto simp add: Field-def)
 hence f`(underS r a) = g`(underS r a) by force
 thus f a = g a
   using assms ∗ embed-determined[of r r' f a] embed-determined[of r r' g a] by
auto
qed

lemma embed-bothWays-inverse:
 assumes WELL: Well-order r and WELL': Well-order r' and
   EMB: embed r r' f and EMB': embed r' r f'
 shows (∀ a ∈ Field r. f'(f a) = a) ∧ (∀ a' ∈ Field r'. f(f' a') = a')
proof−
 have embed r r (f' ∘ f) using assms
   by(auto simp add: comp-embed)
 moreover have embed r r id using assms
   by (auto simp add: id-embed)
 ultimately have ∀ a ∈ Field r. f'(f a) = a
   using assms embed-unique[of r r f' ∘ f id] id-def by auto
 moreover
 {have embed r' r' (f ∘ f') using assms
    by(auto simp add: comp-embed)
  moreover have embed r' r' id using assms
    by (auto simp add: id-embed)
  ultimately have ∀ a' ∈ Field r'. f(f' a') = a'
    using assms embed-unique[of r' r' f ∘ f' id] id-def by auto
 }
 ultimately show ?thesis by blast
qed

lemma embed-bothWays-bij-betw:
 assumes WELL: Well-order r and WELL': Well-order r' and
   EMB: embed r r' f and EMB': embed r' r g
 shows bij-betw f (Field r) (Field r')
```

proof−
let *?A* = *Field r* **let** *?A′* = *Field r′*
have *embed r r* (*g* ∘ *f*) ∧ *embed r′ r′* (*f* ∘ *g*)
using *assms* **by** (*auto simp add*: *comp-embed*)
hence *1*: (∀ *a* ∈ *?A*. *g*(*f a*) = *a*) ∧ (∀ *a′* ∈ *?A′*. *f*(*g a′*) = *a′*)
using *WELL id-embed*[*of r*] *embed-unique*[*of r r g* ∘ *f id*]
WELL′ id-embed[*of r′*] *embed-unique*[*of r′ r′ f* ∘ *g id*]
id-def **by** *auto*
have *2*: (∀ *a* ∈ *?A*. *f a* ∈ *?A′*) ∧ (∀ *a′* ∈ *?A′*. *g a′* ∈ *?A*)
using *assms embed-Field*[*of r r′ f*] *embed-Field*[*of r′ r g*] **by** *blast*

show *?thesis*
proof(*unfold bij-betw-def inj-on-def*, *auto simp add*: *2*)
fix *a b* **assume** ∗: *a* ∈ *?A b* ∈ *?A* **and** ∗∗: *f a* = *f b*
have *a* = *g*(*f a*) ∧ *b* = *g*(*f b*) **using** ∗ *1* **by** *auto*
with ∗∗ **show** *a* = *b* **by** *auto*
next
fix *a′* **assume** ∗: *a′* ∈ *?A′*
hence *g a′* ∈ *?A* ∧ *f*(*g a′*) = *a′* **using** *1 2* **by** *auto*
thus *a′* ∈ *f* ‘ *?A* **by** *force*
qed
qed

lemma *embed-bothWays-iso*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′* **and**
EMB: *embed r r′ f* **and** *EMB′*: *embed r′ r g*
shows *iso r r′ f*
unfolding *iso-def* **using** *assms* **by** (*auto simp add*: *embed-bothWays-bij-betw*)

28.5 More properties of embeddings, strict embeddings and isomorphisms

lemma *embed-bothWays-Field-bij-betw*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′* **and**
EMB: *embed r r′ f* **and** *EMB′*: *embed r′ r f′*
shows *bij-betw f* (*Field r*) (*Field r′*)
proof−
have (∀ *a* ∈ *Field r*. *f′*(*f a*) = *a*) ∧ (∀ *a′* ∈ *Field r′*. *f*(*f′ a′*) = *a′*)
using *assms* **by** (*auto simp add*: *embed-bothWays-inverse*)
moreover
have *f*‘(*Field r*) ≤ *Field r′* ∧ *f′* ‘ (*Field r′*) ≤ *Field r*
using *assms* **by** (*auto simp add*: *embed-Field*)
ultimately
show *?thesis* **using** *bij-betw-byWitness*[*of Field r f′ f Field r′*] **by** *auto*
qed

lemma *embedS-comp-embed*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′*
and *EMB*: *embedS r r′ f* **and** *EMB′*: *embed r′ r″ f′*

shows *embedS r r'' (f' ∘ f)*
proof−
let *?g = (f' ∘ f)* **let** *?h = inv-into (Field r) ?g*
have *1*: *embed r r' f ∧ ¬ (bij-betw f (Field r) (Field r'))*
using *EMB* **by** (*auto simp add*: *embedS-def*)
hence *2*: *embed r r'' ?g*
using *EMB' comp-embed*[*of r r' f r'' f'*] **by** *auto*
moreover
{assume *bij-betw ?g (Field r) (Field r'')*
hence *embed r'' r ?h* **using** *2*
by (*auto simp add*: *inv-into-Field-embed-bij-betw*)
hence *embed r' r (?h ∘ f')* **using** *EMB'*
by (*auto simp add*: *comp-embed*)
hence *bij-betw f (Field r) (Field r')* **using** *WELL WELL' 1*
by (*auto simp add*: *embed-bothWays-Field-bij-betw*)
with *1* **have** *False* **by** *blast*
}
ultimately show *?thesis* **unfolding** *embedS-def* **by** *auto*
qed

lemma *embed-comp-embedS*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
and *EMB*: *embed r r' f* **and** *EMB'*: *embedS r' r'' f'*
shows *embedS r r'' (f' ∘ f)*
proof−
let *?g = (f' ∘ f)* **let** *?h = inv-into (Field r) ?g*
have *1*: *embed r' r'' f' ∧ ¬ (bij-betw f' (Field r') (Field r''))*
using *EMB'* **by** (*auto simp add*: *embedS-def*)
hence *2*: *embed r r'' ?g*
using *WELL EMB comp-embed*[*of r r' f r'' f'*] **by** *auto*
moreover have §: *f' ‘ Field r' ⊆ Field r''*
by (*simp add*: *1 embed-Field*)
{assume §: *bij-betw ?g (Field r) (Field r'')*
hence *embed r'' r ?h* **using** *2 WELL*
by (*auto simp add*: *inv-into-Field-embed-bij-betw*)
hence *embed r' r (inv-into (Field r) ?g ∘ f')*
using *1 BNF-Wellorder-Embedding.comp-embed WELL'* **by** *blast*
then have *bij-betw f' (Field r') (Field r'')*
using *EMB WELL WELL'* § *bij-betw-comp-iff* **by** (*blast intro*: *embed-bothWays-Field-bij-betw*)
with *1* **have** *False* **by** *blast*
}
ultimately show *?thesis* **unfolding** *embedS-def* **by** *auto*
qed

lemma *embed-comp-iso*:
assumes *EMB*: *embed r r' f* **and** *EMB'*: *iso r' r'' f'*
shows *embed r r'' (f' ∘ f)* **using** *assms* **unfolding** *iso-def*
by (*auto simp add*: *comp-embed*)

lemma *iso-comp-embed*:
assumes *EMB*: *iso r r′ f* **and** *EMB′*: *embed r′ r″ f′*
shows *embed r r″* (*f′* ∘ *f*)
using *assms* **unfolding** *iso-def* **by** (*auto simp add*: *comp-embed*)

lemma *embedS-comp-iso*:
assumes *EMB*: *embedS r r′ f* **and** *EMB′*: *iso r′ r″ f′*
shows *embedS r r″* (*f′* ∘ *f*)
proof −
have §: *embed r r′ f* ∧ ¬ *bij-betw f* (*Field r*) (*Field r′*)
using *EMB embedS-def* **by** *blast*
then have *embed r r″* (*f′* ∘ *f*)
using *embed-comp-iso EMB′* **by** *blast*
then have *f* ‘ *Field r* ⊆ *Field r′*
by (*simp add*: *embed-Field* §)
then have ¬ *bij-betw* (*f′* ∘ *f*) (*Field r*) (*Field r″*)
using § *EMB′* **by** (*auto simp*: *bij-betw-comp-iff2 iso-def*)
then show *?thesis*
by (*simp add*: ‹*embed r r″* (*f′* ∘ *f*)› *embedS-def*)
qed

lemma *iso-comp-embedS*:
assumes *WELL*: *Well-order r* **and** *WELL′*: *Well-order r′*
and *EMB*: *iso r r′ f* **and** *EMB′*: *embedS r′ r″ f′*
shows *embedS r r″* (*f′* ∘ *f*)
using *assms* **unfolding** *iso-def* **by** (*auto simp add*: *embed-comp-embedS*)

lemma *embedS-Field*:
assumes *WELL*: *Well-order r* **and** *EMB*: *embedS r r′ f*
shows *f* ‘ (*Field r*) < *Field r′*
proof−
have *f*‘(*Field r*) ≤ *Field r′* **using** *assms*
by (*auto simp add*: *embed-Field embedS-def*)
moreover
{have *inj-on f* (*Field r*) **using** *assms*
by (*auto simp add*: *embedS-def embed-inj-on*)
hence *f*‘(*Field r*) ≠ *Field r′* **using** *EMB*
by (*auto simp add*: *embedS-def bij-betw-def*)
}
ultimately show *?thesis* **by** *blast*
qed

lemma *embedS-iff*:
assumes *WELL*: *Well-order r* **and** *ISO*: *embed r r′ f*
shows *embedS r r′ f* = (*f* ‘ (*Field r*) < *Field r′*)
proof
assume *embedS r r′ f*
thus *f* ‘ *Field r* ⊂ *Field r′*
using *WELL* **by** (*auto simp add*: *embedS-Field*)

next
assume f ‘ $Field\ r \subset Field\ r'$
hence $\neg$ *bij-betw* f (*Field* r) (*Field* r')
unfolding *bij-betw-def* **by** *blast*
thus *embedS* $r\ r'\ f$ **unfolding** *embedS-def*
using *ISO* **by** *auto*
qed

lemma *iso-Field*: *iso* $r\ r'\ f \Longrightarrow f$ ‘ (*Field* r) = *Field* r'
by (*auto simp add*: *iso-def bij-betw-def*)

lemma *iso-iff*:
assumes *Well-order* r
shows *iso* $r\ r'\ f$ = (*embed* $r\ r'\ f \wedge f$ ‘ (*Field* r) = *Field* r')
proof
assume *iso* $r\ r'\ f$
thus *embed* $r\ r'\ f \wedge f$ ‘ (*Field* r) = *Field* r'
by (*auto simp add*: *iso-Field iso-def*)
next
assume $*$: *embed* $r\ r'\ f \wedge f$ ‘ *Field* r = *Field* r'
hence *inj-on* f (*Field* r) **using** *assms* **by** (*auto simp add*: *embed-inj-on*)
with $*$ **have** *bij-betw* f (*Field* r) (*Field* r')
unfolding *bij-betw-def* **by** *simp*
with $*$ **show** *iso* $r\ r'\ f$ **unfolding** *iso-def* **by** *auto*
qed

lemma *iso-iff2*: *iso* $r\ r'\ f \longleftrightarrow$
bij-betw f (*Field* r) (*Field* r') $\wedge$
$(\forall\ a \in Field\ r.\ \forall\ b \in Field\ r.\ (a,\ b) \in r \longleftrightarrow (f\ a,\ f\ b) \in r')$
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
then have *bij-betw* f (*Field* r) (*Field* r') **and** *emb*: *embed* $r\ r'\ f$
by (*auto simp*: *bij-betw-def iso-def*)
then obtain g **where** g: $\bigwedge x.\ x \in Field\ r \Longrightarrow g\ (f\ x) = x$
by (*auto simp*: *bij-betw-iff-bijections*)
moreover
have $(a,\ b) \in r$ **if** $a \in Field\ r\ b \in Field\ r\ (f\ a,\ f\ b) \in r'$ **for** $a\ b$
using *that emb g g* [*OF FieldI1*] — yes it's weird
by (*force simp add*: *embed-def under-def bij-betw-iff-bijections*)
ultimately show *?rhs*
using *L* **by** (*auto simp*: *compat-def iso-def dest*: *embed-compat*)
next
assume *R*: *?rhs*
then show *?lhs*
apply (*clarsimp simp add*: *iso-def embed-def under-def bij-betw-iff-bijections*)
apply (*rule-tac* $x{=}g$ **in** *exI*)
apply (*fastforce simp add*: *intro*: *FieldI1*)+
done

qed

lemma *iso-iff3*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows *iso r r' f* = (*bij-betw f* (*Field r*) (*Field r'*) ∧ *compat r r' f*)
proof
assume *iso r r' f*
thus *bij-betw f* (*Field r*) (*Field r'*) ∧ *compat r r' f*
unfolding *compat-def* **using** *WELL* **by** (*auto simp add*: *iso-iff2 Field-def*)
next
have *Well*: *wo-rel r* ∧ *wo-rel r'* **using** *WELL WELL'*
by (*auto simp add*: *wo-rel-def*)
assume ∗: *bij-betw f* (*Field r*) (*Field r'*) ∧ *compat r r' f*
thus *iso r r' f*
unfolding *compat-def* **using** *assms*
proof(*auto simp add*: *iso-iff2*)
fix *a b* **assume** ∗∗: *a* ∈ *Field r b* ∈ *Field r* **and**
∗∗∗: (*f a*, *f b*) ∈ *r'*
{**assume** (*b*,*a*) ∈ *r* ∨ *b* = *a*
hence (*b*,*a*) ∈ *r***using** *Well* ∗∗ *wo-rel.REFL*[*of r*] *refl-on-def*[*of* - *r*] **by** *blast*
hence (*f b*, *f a*) ∈ *r'* **using** ∗ **unfolding** *compat-def* **by** *auto*
hence *f a* = *f b*
using *Well* ∗∗∗ *wo-rel.ANTISYM*[*of r'*] *antisym-def*[*of r'*] **by** *blast*
hence *a* = *b* **using** ∗ ∗∗ **unfolding** *bij-betw-def inj-on-def* **by** *auto*
hence (*a*,*b*) ∈ *r* **using** *Well* ∗∗ *wo-rel.REFL*[*of r*] *refl-on-def*[*of* - *r*] **by** *blast*
}
thus (*a*,*b*) ∈ *r*
using *Well* ∗∗ *wo-rel.TOTAL*[*of r*] *total-on-def*[*of* - *r*] **by** *blast*
qed
qed

lemma *iso-imp-inj-on*:
assumes *iso r r' f* **shows** *inj-on f* (*Field r*)
using *assms* **unfolding** *iso-iff2 bij-betw-def* **by** *blast*

lemma *iso-backward-Field*:
assumes *x* ∈ *Field r' iso r r' f*
shows *inv-into* (*Field r*) *f x* ∈ *Field r*
using *assms iso-Field* **by** (*blast intro*!: *inv-into-into*)

lemma *iso-backward*:
assumes (*x*,*y*) ∈ *r'* **and** *iso*: *iso r r' f*
shows (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f y*) ∈ *r*
proof −
have §: ⋀*x*. (∃ *xa*∈*Field r*. *x* = *f xa*) = (*x* ∈ *Field r'*)
using *assms iso-Field* [*OF iso*] **by** (*force simp add*:)
have *x* ∈ *Field r' y* ∈ *Field r'*
using *assms* **by** (*auto simp*: *Field-def*)
with § [*of x*] § [*of y*] *assms* **show** *?thesis*

by (*auto simp add*: *iso-iff2 bij-betw-inv-into-left*)
qed

lemma *iso-forward*:
assumes (*x*,*y*) ∈ *r iso r r′ f* **shows** (*f x*,*f y*) ∈ *r′*
using *assms* **by** (*auto simp*: *Field-def iso-iff2*)

lemma *iso-trans*:
assumes *trans r* **and** *iso*: *iso r r′ f* **shows** *trans r′*
unfolding *trans-def*
proof *clarify*
fix *x y z*
assume *xyz*: (*x*, *y*) ∈ *r′* (*y*, *z*) ∈ *r′*
then have ∗: (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f y*) ∈ *r*
(*inv-into* (*Field r*) *f y*, *inv-into* (*Field r*) *f z*) ∈ *r*
using *iso-backward* [*OF - iso*] **by** *blast*+
then have *inv-into* (*Field r*) *f x* ∈ *Field r inv-into* (*Field r*) *f y* ∈ *Field r*
by (*auto simp*: *Field-def*)
with ∗ **have** (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f z*) ∈ *r*
using *assms*(*1*) **by** (*blast intro*: *transD*)
then have (*f* (*inv-into* (*Field r*) *f x*), *f* (*inv-into* (*Field r*) *f z*)) ∈ *r′*
by (*blast intro*: *iso iso-forward*)
moreover have *x* ∈ *f* ‘ *Field r z* ∈ *f* ‘ *Field r*
using *xyz iso iso-Field* **by** (*blast intro*: *FieldI1 FieldI2*)+
ultimately show (*x*, *z*) ∈ *r′*
by (*simp add*: *f-inv-into-f*)
qed

lemma *iso-Total*:
assumes *Total r* **and** *iso*: *iso r r′ f* **shows** *Total r′*
unfolding *total-on-def*
proof *clarify*
fix *x y*
assume *xy*: *x* ∈ *Field r′ y* ∈ *Field r′ x* ≠ *y* (*y*,*x*) ∉ *r′*
then have §: *inv-into* (*Field r*) *f x* ∈ *Field r inv-into* (*Field r*) *f y* ∈ *Field r*
using *iso-backward-Field* [*OF - iso*] **by** *auto*
moreover have *x* ∈ *f* ‘ *Field r y* ∈ *f* ‘ *Field r*
using *xy iso iso-Field* **by** (*blast intro*: *FieldI1 FieldI2*)+
ultimately have *False* **if** *inv-into* (*Field r*) *f x* = *inv-into* (*Field r*) *f y*
using *inv-into-injective* [*OF that*] ‹*x* ≠ *y*› **by** *simp*
then have (*inv-into* (*Field r*) *f x*, *inv-into* (*Field r*) *f y*) ∈ *r* ∨ (*inv-into* (*Field r*) *f y*, *inv-into* (*Field r*) *f x*) ∈ *r*
using *assms* § **by** (*auto simp*: *total-on-def*)
then show (*x*, *y*) ∈ *r′*
using *assms xy* **by** (*auto simp*: *iso-Field f-inv-into-f dest*!: *iso-forward* [*OF - iso*])
qed

lemma *iso-wf*:

assumes *wf r* **and** *iso*: *iso r r′ f* **shows** *wf r′*
proof −
have *bij-betw f* (*Field r*) (*Field r′*)
and *iff*: (∀ *a* ∈ *Field r*. ∀ *b* ∈ *Field r*. (*a*, *b*) ∈ *r* ⟷ (*f a*, *f b*) ∈ *r′*)
using *assms* **by** (*auto simp*: *iso-iff2*)
show *?thesis*
proof (*rule wfI-min*)
fix *x*::′*b* **and** *Q*
assume *x* ∈ *Q*
let *?g* = *inv-into* (*Field r*) *f*
obtain *z0* **where** *z0* ∈ *?g* ‘ *Q*
using ‹*x* ∈ *Q*› **by** *blast*
then obtain *z* **where** *z*: *z* ∈ *?g* ‘ *Q* **and** ⋀*x y*. ⟦(*y*, *z*) ∈ *r*; *x* ∈ *Q*⟧ ⟹ *y* ≠ *?g x*
by (*rule wfE-min* [*OF* ‹*wf r*›]) *auto*
then have ∀ *y*. (*y*, *inv-into Q ?g z*) ∈ *r′* ⟶ *y* ∉ *Q*
by (*clarsimp simp*: *f-inv-into-f*[*OF z*] *z dest*!: *iso-backward* [*OF - iso*]) *blast*
moreover have *inv-into Q ?g z* ∈ *Q*
by (*simp add*: *inv-into-into z*)
ultimately show ∃ *z*∈*Q*. ∀ *y*. (*y*, *z*) ∈ *r′* ⟶ *y* ∉ *Q* **..**
qed
qed

end

29 Constructions on Wellorders as Needed by Bounded Natural Functors

theory *BNF-Wellorder-Constructions*
imports *BNF-Wellorder-Embedding*
begin

In this section, we study basic constructions on well-orders, such as restriction to a set/order filter, copy via direct images, ordinal-like sum of disjoint well-orders, and bounded square. We also define between well-orders the relations *ordLeq*, of being embedded (abbreviated ≤*o*), *ordLess*, of being strictly embedded (abbreviated <*o*), and *ordIso*, of being isomorphic (abbreviated =*o*). We study the connections between these relations, order filters, and the aforementioned constructions. A main result of this section is that <*o* is well-founded.

29.1 Restriction to a set

abbreviation *Restr* :: ′*a rel* ⇒ ′*a set* ⇒ ′*a rel*
where *Restr r A* ≡ *r Int* (*A* × *A*)

lemma *Restr-subset*:

$A \leq B \Longrightarrow Restr\ (Restr\ r\ B)\ A = Restr\ r\ A$
by *blast*

lemma *Restr-Field*: $Restr\ r\ (Field\ r) = r$
unfolding *Field-def* **by** *auto*

lemma *Refl-Restr*: $Refl\ r \Longrightarrow Refl(Restr\ r\ A)$
unfolding *refl-on-def Field-def* **by** *auto*

lemma *linear-order-on-Restr*:
$linear\text{-}order\text{-}on\ A\ r \Longrightarrow linear\text{-}order\text{-}on\ (A \cap above\ r\ x)\ (Restr\ r\ (above\ r\ x))$
by(*simp add*: *order-on-defs refl-on-def trans-def antisym-def total-on-def*)(*safe*; *blast*)

lemma *antisym-Restr*:
$antisym\ r \Longrightarrow antisym(Restr\ r\ A)$
unfolding *antisym-def Field-def* **by** *auto*

lemma *Total-Restr*:
$Total\ r \Longrightarrow Total(Restr\ r\ A)$
unfolding *total-on-def Field-def* **by** *auto*

lemma *total-on-imp-Total-Restr*: $total\text{-}on\ A\ r \Longrightarrow Total\ (Restr\ r\ A)$
by (*auto simp*: *Field-def total-on-def*)

lemma *trans-Restr*:
$trans\ r \Longrightarrow trans(Restr\ r\ A)$
unfolding *trans-def Field-def* **by** *blast*

lemma *Preorder-Restr*:
assumes *Preorder r*
shows *Preorder*$(Restr\ r\ A)$
unfolding *preorder-on-def*
proof (*intro conjI*)
have $r \subseteq Field\ r \times Field\ r$ **and** *Refl r* **and** *trans r*
using ‹*Preorder r*›
by (*simp-all only*: *preorder-on-def*)

show $Restr\ r\ A \subseteq Field\ (Restr\ r\ A) \times Field\ (Restr\ r\ A)$
using ‹$r \subseteq Field\ r \times Field\ r$›
by (*auto simp add*: *Field-def*)

show *Refl* $(Restr\ r\ A)$
using *Refl-Restr*[*OF* ‹*Refl r*›] **.**

show *trans* $(Restr\ r\ A)$
using *trans-Restr*[*OF* ‹*trans r*›] **.**
qed

lemma *Partial-order-Restr*:
Partial-order r $\Longrightarrow$ *Partial-order*(*Restr r A*)
unfolding *partial-order-on-def* **by** (*simp add*: *Preorder-Restr antisym-Restr*)

lemma *Linear-order-Restr*:
Linear-order r $\Longrightarrow$ *Linear-order*(*Restr r A*)
unfolding *linear-order-on-def* **by** (*simp add*: *Partial-order-Restr Total-Restr*)

lemma *Well-order-Restr*:
assumes *Well-order r*
shows *Well-order*(*Restr r A*)
using *assms*
by (*auto simp*: *well-order-on-def Linear-order-Restr elim*: *wf-subset*)

lemma *Field-Restr-subset*: *Field*(*Restr r A*) $\leq$ *A*
by (*auto simp add*: *Field-def*)

lemma *Refl-Field-Restr*:
Refl r $\Longrightarrow$ *Field*(*Restr r A*) = (*Field r*) *Int A*
unfolding *refl-on-def Field-def* **by** *blast*

lemma *Refl-Field-Restr2*:
$[\![$*Refl r*; *A* $\leq$ *Field r*$]\!] \Longrightarrow$ *Field*(*Restr r A*) = *A*
by (*auto simp add*: *Refl-Field-Restr*)

lemma *well-order-on-Restr*:
assumes *WELL*: *Well-order r* **and** *SUB*: *A* $\leq$ *Field r*
shows *well-order-on A* (*Restr r A*)
using *assms*
using *Well-order-Restr*[*of r A*] *Refl-Field-Restr2*[*of r A*]
order-on-defs[*of Field r r*] **by** *auto*

29.2 Order filters versus restrictions and embeddings

lemma *Field-Restr-ofilter*:
$[\![$*Well-order r*; *wo-rel.ofilter r A*$]\!] \Longrightarrow$ *Field*(*Restr r A*) = *A*
by (*auto simp add*: *wo-rel-def wo-rel.ofilter-def wo-rel.REFL Refl-Field-Restr2*)

lemma *ofilter-Restr-under*:
assumes *WELL*: *Well-order r* **and** *OF*: *wo-rel.ofilter r A* **and** *IN*: *a* $\in$ *A*
shows *under* (*Restr r A*) *a* = *under r a*
unfolding *wo-rel.ofilter-def under-def*
proof
show {*b*. (*b*, *a*) $\in$ *Restr r A*} $\subseteq$ {*b*. (*b*, *a*) $\in$ *r*}
by *auto*
next
have *under r a* $\subseteq$ *A*
proof
fix *x*

```
    assume ∗: x ∈ under r a
    then have a ∈ Field r
      unfolding under-def using Field-def by fastforce
    then show x ∈ A using IN assms ∗
      by (auto simp add: wo-rel-def wo-rel.ofilter-def)
  qed
  then show {b. (b, a) ∈ r} ⊆ {b. (b, a) ∈ Restr r A}
    unfolding under-def using assms by auto
qed

lemma ofilter-embed:
  assumes Well-order r
  shows wo-rel.ofilter r A = (A ≤ Field r ∧ embed (Restr r A) r id)
proof
  assume ∗: wo-rel.ofilter r A
  show A ≤ Field r ∧ embed (Restr r A) r id
    unfolding embed-def
  proof safe
    fix a assume a ∈ A thus a ∈ Field r using assms ∗
      by (auto simp add: wo-rel-def wo-rel.ofilter-def)
  next
    fix a assume a ∈ Field (Restr r A)
    thus bij-betw id (under (Restr r A) a) (under r (id a)) using assms ∗
      by (simp add: ofilter-Restr-under Field-Restr-ofilter)
  qed
next
  assume ∗: A ≤ Field r ∧ embed (Restr r A) r id
  hence Field(Restr r A) ≤ Field r
    using assms  embed-Field[of Restr r A r id] id-def
      Well-order-Restr[of r] by auto
  {fix a assume a ∈ A
    hence a ∈ Field(Restr r A) using ∗ assms
      by (simp add: order-on-defs Refl-Field-Restr2)
    hence bij-betw id (under (Restr r A) a) (under r a)
      using ∗ unfolding embed-def by auto
    hence under r a ≤ under (Restr r A) a
      unfolding bij-betw-def by auto
    also have ... ≤ Field(Restr r A) by (simp add: under-Field)
    also have ... ≤ A by (simp add: Field-Restr-subset)
    finally have under r a ≤ A .
  }
  thus wo-rel.ofilter r A using assms ∗ by (simp add: wo-rel-def wo-rel.ofilter-def)
qed

lemma ofilter-Restr-Int:
  assumes WELL: Well-order r and OFA: wo-rel.ofilter r A
  shows wo-rel.ofilter (Restr r B) (A Int B)
proof−
  let ?rB = Restr r B
```

```
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field: Field ?rB = Field r Int B
    using Refl-Field-Restr by blast
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
    by (simp add: Well-order-Restr wo-rel-def)

  show ?thesis using WellB assms
    unfolding wo-rel.ofilter-def under-def ofilter-def
  proof safe
    fix a assume a ∈ A and ∗: a ∈ B
    hence a ∈ Field r using OFA Well by (auto simp add: wo-rel.ofilter-def)
    with ∗ show a ∈ Field ?rB using Field by auto
  next
    fix a b assume a ∈ A and (b,a) ∈ r
    thus b ∈ A using Well OFA by (auto simp add: wo-rel.ofilter-def under-def)
  qed
qed

lemma ofilter-Restr-subset:
  assumes WELL: Well-order r and OFA: wo-rel.ofilter r A and SUB: A ≤ B
  shows wo-rel.ofilter (Restr r B) A
proof−
  have A Int B = A using SUB by blast
  thus ?thesis using assms ofilter-Restr-Int[of r A B] by auto
qed

lemma ofilter-subset-embed:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows (A ≤ B) = (embed (Restr r A) (Restr r B) id)
proof−
  let ?rA = Restr r A  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence FieldA: Field ?rA = Field r Int A
    using Refl-Field-Restr by blast
  have FieldB: Field ?rB = Field r Int B
    using Refl Refl-Field-Restr by blast
  have WellA: wo-rel ?rA ∧ Well-order ?rA using WELL
    by (simp add: Well-order-Restr wo-rel-def)
  have WellB: wo-rel ?rB ∧ Well-order ?rB using WELL
    by (simp add: Well-order-Restr wo-rel-def)

  show ?thesis
  proof
    assume ∗: A ≤ B
    hence wo-rel.ofilter (Restr r B) A using assms
      by (simp add: ofilter-Restr-subset)
```

```
    hence embed (Restr ?rB A) (Restr r B) id
      using WellB ofilter-embed[of ?rB A] by auto
    thus embed (Restr r A) (Restr r B) id
      using ∗ by (simp add: Restr-subset)
  next
    assume ∗: embed (Restr r A) (Restr r B) id
    {fix a assume ∗∗: a ∈ A
      hence a ∈ Field r using Well OFA by (auto simp add: wo-rel.ofilter-def)
      with ∗∗ FieldA have a ∈ Field ?rA by auto
      hence a ∈ Field ?rB using ∗ WellA embed-Field[of ?rA ?rB id] by auto
      hence a ∈ B using FieldB by auto
    }
    thus A ≤ B by blast
  qed
qed

lemma ofilter-subset-embedS-iso:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows ((A < B) = (embedS (Restr r A) (Restr r B) id)) ∧
      ((A = B) = (iso (Restr r A) (Restr r B) id))
proof−
  let ?rA = Restr r A  let ?rB = Restr r B
  have Well: wo-rel r unfolding wo-rel-def using WELL .
  hence Refl: Refl r by (simp add: wo-rel.REFL)
  hence Field ?rA = Field r Int A
    using Refl-Field-Restr by blast
  hence FieldA: Field ?rA = A using OFA Well
    by (auto simp add: wo-rel.ofilter-def)
  have Field ?rB = Field r Int B
    using Refl Refl-Field-Restr by blast
  hence FieldB: Field ?rB = B using OFB Well
    by (auto simp add: wo-rel.ofilter-def)

  show ?thesis unfolding embedS-def iso-def
    using assms ofilter-subset-embed[of r A B]
      FieldA FieldB bij-betw-id-iff[of A B] by auto
qed

lemma ofilter-subset-embedS:
  assumes WELL: Well-order r and
    OFA: wo-rel.ofilter r A and OFB: wo-rel.ofilter r B
  shows (A < B) = embedS (Restr r A) (Restr r B) id
  using assms by (simp add: ofilter-subset-embedS-iso)

lemma embed-implies-iso-Restr:
  assumes WELL: Well-order r and WELL′: Well-order r′ and
    EMB: embed r′ r f
  shows iso r′ (Restr r (f ‘ (Field r′))) f
```

proof−
let *?A′* = *Field r′*
let *?r″* = *Restr r* (*f* ‘ *?A′*)
have *0*: *Well-order ?r″* **using** *WELL Well-order-Restr* **by** *blast*
have *1*: *wo-rel.ofilter r* (*f* ‘ *?A′*) **using** *assms embed-Field-ofilter* **by** *blast*
hence *Field ?r″* = *f* ‘ (*Field r′*) **using** *WELL Field-Restr-ofilter* **by** *blast*
hence *bij-betw f ?A′* (*Field ?r″*)
using *EMB embed-inj-on WELL′* **unfolding** *bij-betw-def* **by** *blast*
moreover
{have ∀ *a b*. (*a*,*b*) ∈ *r′* ⟶ *a* ∈ *Field r′* ∧ *b* ∈ *Field r′*
unfolding *Field-def* **by** *auto*
hence *compat r′ ?r″ f*
using *assms embed-iff-compat-inj-on-ofilter*
unfolding *compat-def* **by** *blast*
}
ultimately show *?thesis* **using** *WELL′ 0 iso-iff3* **by** *blast*
qed

29.3 The strict inclusion on proper ofilters is well-founded

definition *ofilterIncl* :: *′a rel* ⇒ *′a set rel*
where
ofilterIncl r ≡ {(*A*,*B*). *wo-rel.ofilter r A* ∧ *A* ≠ *Field r* ∧
wo-rel.ofilter r B ∧ *B* ≠ *Field r* ∧ *A* < *B*}

lemma *wf-ofilterIncl*:
assumes *WELL*: *Well-order r*
shows *wf*(*ofilterIncl r*)
proof−
have *Well*: *wo-rel r* **using** *WELL* **by** (*simp add*: *wo-rel-def*)
hence *Lo*: *Linear-order r* **by** (*simp add*: *wo-rel.LIN*)
let *?h* = (λ *A*. *wo-rel.suc r A*)
let *?rS* = *r* − *Id*
have *wf ?rS* **using** *WELL* **by** (*simp add*: *order-on-defs*)
moreover
have *compat* (*ofilterIncl r*) *?rS ?h*
proof(*unfold compat-def ofilterIncl-def*,
intro allI impI, *simp*, *elim conjE*)
fix *A B*
assume ∗: *wo-rel.ofilter r A A* ≠ *Field r* **and**
∗∗: *wo-rel.ofilter r B B* ≠ *Field r* **and** ∗∗∗: *A* < *B*
then obtain *a* **and** *b* **where** *0*: *a* ∈ *Field r* ∧ *b* ∈ *Field r* **and**
1: *A* = *underS r a* ∧ *B* = *underS r b*
using *Well* **by** (*auto simp add*: *wo-rel.ofilter-underS-Field*)
hence *a* ≠ *b* **using** ∗∗∗ **by** *auto*
moreover
have (*a*,*b*) ∈ *r* **using** *0 1 Lo* ∗∗∗
by (*auto simp add*: *underS-incl-iff*)
moreover

have $a = wo\text{-}rel.suc\ r\ A \wedge b = wo\text{-}rel.suc\ r\ B$
using *Well 0 1* **by** (*simp add*: *wo-rel.suc-underS*)
ultimately
show $(wo\text{-}rel.suc\ r\ A,\ wo\text{-}rel.suc\ r\ B) \in r \wedge wo\text{-}rel.suc\ r\ A \neq wo\text{-}rel.suc\ r\ B$
by *simp*
qed
ultimately show *wf* (*ofilterIncl r*) **by** (*simp add*: *compat-wf*)
qed

29.4 Ordering the well-orders by existence of embeddings

We define three relations between well-orders:

- *ordLeq*, of being embedded (abbreviated $\leq o$);
- *ordLess*, of being strictly embedded (abbreviated $<o$);
- *ordIso*, of being isomorphic (abbreviated $=o$).

The prefix "ord" and the index "o" in these names stand for "ordinal-like". These relations shall be proved to be inter-connected in a similar fashion as the trio $\leq$, $<$, $=$ associated to a total order on a set.

definition *ordLeq* :: $('a\ rel * 'a'\ rel)\ set$
where
$ordLeq = \{(r,r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ embed\ r\ r'\ f)\}$

abbreviation *ordLeq2* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** ‹<=o› *50*)
where $r <=o\ r' \equiv (r,r') \in ordLeq$

abbreviation *ordLeq3* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** ‹≤o› *50*)
where $r \leq o\ r' \equiv r <=o\ r'$

definition *ordLess* :: $('a\ rel * 'a'\ rel)\ set$
where
$ordLess = \{(r,r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ embedS\ r\ r'\ f)\}$

abbreviation *ordLess2* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** ‹<o› *50*)
where $r <o\ r' \equiv (r,r') \in ordLess$

definition *ordIso* :: $('a\ rel * 'a'\ rel)\ set$
where
$ordIso = \{(r,r').\ Well\text{-}order\ r \wedge Well\text{-}order\ r' \wedge (\exists f.\ iso\ r\ r'\ f)\}$

abbreviation *ordIso2* :: $'a\ rel \Rightarrow 'a'\ rel \Rightarrow bool$ (**infix** ‹=o› *50*)
where $r =o\ r' \equiv (r,r') \in ordIso$

lemmas *ordRels-def* = *ordLeq-def ordLess-def ordIso-def*

lemma *ordLeq-Well-order-simp*:

assumes $r \leq o\ r'$
shows *Well-order* $r \wedge$ *Well-order* r'
using *assms* **unfolding** *ordLeq-def* **by** *simp*

Notice that the relations $\leq o$, $< o$, $= o$ connect well-orders on potentially *distinct* types. However, some of the lemmas below, including the next one, restrict implicitly the type of these relations to $((' a\ rel) * (' a\ rel))\ set$, i.e., to $' a\ rel\ rel$.

lemma *ordLeq-reflexive*:
Well-order $r \Longrightarrow r \leq o\ r$
unfolding *ordLeq-def* **using** *id-embed*[*of* r] **by** *blast*

lemma *ordLeq-transitive*[*trans*]:
assumes $r \leq o\ r'$ **and** $r' \leq o\ r''$
shows $r \leq o\ r''$
using *assms* **by** (*auto simp*: *ordLeq-def intro*: *comp-embed*)

lemma *ordLeq-total*:
$[\![$*Well-order* r; *Well-order* $r'$$]\!] \Longrightarrow r \leq o\ r' \vee r' \leq o\ r$
unfolding *ordLeq-def* **using** *wellorders-totally-ordered* **by** *blast*

lemma *ordIso-reflexive*:
Well-order $r \Longrightarrow r =o\ r$
unfolding *ordIso-def* **using** *id-iso*[*of* r] **by** *blast*

lemma *ordIso-transitive*[*trans*]:
assumes $*$: $r =o\ r'$ **and** $**$: $r' =o\ r''$
shows $r =o\ r''$
using *assms* **by** (*auto simp*: *ordIso-def intro*: *comp-iso*)

lemma *ordIso-symmetric*:
assumes $*$: $r =o\ r'$
shows $r' =o\ r$
proof−
obtain f **where** *1*: *Well-order* $r \wedge$ *Well-order* r' **and**
2: *embed* $r\ r'\ f \wedge$ *bij-betw* f (*Field* r) (*Field* r')
using $*$ **by** (*auto simp add*: *ordIso-def iso-def*)
let *?f'* = *inv-into* (*Field* r) f
have *embed* $r'\ r$ *?f'* $\wedge$ *bij-betw ?f'* (*Field* r') (*Field* r)
using *1 2* **by** (*simp add*: *bij-betw-inv-into inv-into-Field-embed-bij-betw*)
thus $r' =o\ r$ **unfolding** *ordIso-def* **using** *1* **by** (*auto simp add*: *iso-def*)
qed

lemma *ordLeq-ordLess-trans*[*trans*]:
assumes $r \leq o\ r'$ **and** $r' < o\ r''$
shows $r < o\ r''$
proof−
have *Well-order* $r \wedge$ *Well-order* r''
using *assms* **unfolding** *ordLeq-def ordLess-def* **by** *auto*

thus *?thesis* **using** *assms* **unfolding** *ordLeq-def ordLess-def*
using *embed-comp-embedS* **by** *blast*
qed

lemma *ordLess-ordLeq-trans*[*trans*]:
assumes $r <o r'$ **and** $r' \leq o\ r''$
shows $r <o r''$
using *embedS-comp-embed assms* **by** (*force simp*: *ordLeq-def ordLess-def*)

lemma *ordLeq-ordIso-trans*[*trans*]:
assumes $r \leq o\ r'$ **and** $r' =o r''$
shows $r \leq o\ r''$
using *embed-comp-iso assms* **by** (*force simp*: *ordLeq-def ordIso-def*)

lemma *ordIso-ordLeq-trans*[*trans*]:
assumes $r =o r'$ **and** $r' \leq o\ r''$
shows $r \leq o\ r''$
using *iso-comp-embed assms* **by** (*force simp*: *ordLeq-def ordIso-def*)

lemma *ordLess-ordIso-trans*[*trans*]:
assumes $r <o r'$ **and** $r' =o r''$
shows $r <o r''$
using *embedS-comp-iso assms* **by** (*force simp*: *ordLess-def ordIso-def*)

lemma *ordIso-ordLess-trans*[*trans*]:
assumes $r =o r'$ **and** $r' <o r''$
shows $r <o r''$
using *iso-comp-embedS assms* **by** (*force simp*: *ordLess-def ordIso-def*)

lemma *ordLess-not-embed*:
assumes $r <o r'$
shows $\neg(\exists f'.\ embed\ r'\ r\ f')$
proof−
obtain f **where** *1*: *Well-order r* $\wedge$ *Well-order r'* **and** *2*: *embed r r' f* **and**
3: $\neg$ *bij-betw f* (*Field r*) (*Field r'*)
using *assms* **unfolding** *ordLess-def* **by** (*auto simp add*: *embedS-def*)
{fix f' **assume** $*$: *embed r' r f'*
hence *bij-betw f* (*Field r*) (*Field r'*) **using** *1 2*
by (*simp add*: *embed-bothWays-Field-bij-betw*)
with *3* **have** *False* **by** *contradiction*
}
thus *?thesis* **by** *blast*
qed

lemma *ordLess-Field*:
assumes *OL*: *r1* $<o$ *r2* **and** *EMB*: *embed r1 r2 f*
shows $\neg$ (*f*‘(*Field r1*) = *Field r2*)
proof−
let *?A1* = *Field r1* **let** *?A2* = *Field r2*

obtain *g* **where**
0: *Well-order r1* ∧ *Well-order r2* **and**
1: *embed r1 r2 g* ∧ ¬(*bij-betw g ?A1 ?A2*)
using *OL* **unfolding** *ordLess-def* **by** (*auto simp add*: *embedS-def*)
hence ∀ *a* ∈ *?A1*. *f a* = *g a*
using *0 EMB embed-unique*[*of r1*] **by** *auto*
hence ¬(*bij-betw f ?A1 ?A2*)
using *1 bij-betw-cong*[*of ?A1*] **by** *blast*
moreover
have *inj-on f ?A1* **using** *EMB 0* **by** (*simp add*: *embed-inj-on*)
ultimately show *?thesis* **by** (*simp add*: *bij-betw-def*)
qed

lemma *ordLess-iff*:
r <*o r'* = (*Well-order r* ∧ *Well-order r'* ∧ ¬(∃*f'*. *embed r' r f'*))
proof
assume ∗: *r* <*o r'*
hence ¬(∃*f'*. *embed r' r f'*) **using** *ordLess-not-embed*[*of r r'*] **by** *simp*
with ∗ **show** *Well-order r* ∧ *Well-order r'* ∧ ¬ (∃*f'*. *embed r' r f'*)
unfolding *ordLess-def* **by** *auto*
next
assume ∗: *Well-order r* ∧ *Well-order r'* ∧ ¬ (∃*f'*. *embed r' r f'*)
then obtain *f* **where** *1*: *embed r r' f*
using *wellorders-totally-ordered*[*of r r'*] **by** *blast*
moreover
{assume *bij-betw f* (*Field r*) (*Field r'*)
with ∗ *1* **have** *embed r' r* (*inv-into* (*Field r*) *f*)
using *inv-into-Field-embed-bij-betw*[*of r r' f*] **by** *auto*
with ∗ **have** *False* **by** *blast*
}
ultimately show (*r*,*r'*) ∈ *ordLess*
unfolding *ordLess-def* **using** ∗ **by** (*fastforce simp add*: *embedS-def*)
qed

lemma *ordLess-irreflexive*: ¬ *r* <*o r*
using *id-embed*[*of r*] **by** (*auto simp*: *ordLess-iff*)

lemma *ordLeq-iff-ordLess-or-ordIso*:
r ≤*o r'* = (*r* <*o r'* ∨ *r* =*o r'*)
unfolding *ordRels-def embedS-defs iso-defs* **by** *blast*

lemma *ordIso-iff-ordLeq*:
(*r* =*o r'*) = (*r* ≤*o r'* ∧ *r'* ≤*o r*)
proof
assume *r* =*o r'*
then obtain *f* **where** *1*: *Well-order r* ∧ *Well-order r'* ∧
embed r r' f ∧ *bij-betw f* (*Field r*) (*Field r'*)
unfolding *ordIso-def iso-defs* **by** *auto*
hence *embed r r' f* ∧ *embed r' r* (*inv-into* (*Field r*) *f*)

by (*simp add*: *inv-into-Field-embed-bij-betw*)
thus $r \leq o\ r' \wedge r' \leq o\ r$
unfolding *ordLeq-def* **using** *1* **by** *auto*
next
assume $r \leq o\ r' \wedge r' \leq o\ r$
then obtain f **and** g **where** *1*: *Well-order* $r \wedge$ *Well-order* $r' \wedge$
embed $r\ r'\ f \wedge$ *embed* $r'\ r\ g$
unfolding *ordLeq-def* **by** *auto*
hence *iso* $r\ r'\ f$ **by** (*auto simp add*: *embed-bothWays-iso*)
thus $r =o\ r'$ **unfolding** *ordIso-def* **using** *1* **by** *auto*
qed

lemma *not-ordLess-ordLeq*:
$r <o\ r' \Longrightarrow \neg\ r' \leq o\ r$
using *ordLess-ordLeq-trans ordLess-irreflexive* **by** *blast*

lemma *not-ordLeq-ordLess*:
$r \leq o\ r' \Longrightarrow \neg\ r' <o\ r$
using *not-ordLess-ordLeq* **by** *blast*

lemma *ordLess-or-ordLeq*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $r <o\ r' \vee r' \leq o\ r$
proof−
have $r \leq o\ r' \vee r' \leq o\ r$
using *assms* **by** (*simp add*: *ordLeq-total*)
moreover
{assume $\neg\ r <o\ r' \wedge r \leq o\ r'$
hence $r =o\ r'$ **using** *ordLeq-iff-ordLess-or-ordIso* **by** *blast*
hence $r' \leq o\ r$ **using** *ordIso-symmetric ordIso-iff-ordLeq* **by** *blast*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *not-ordLess-ordIso*:
$r <o\ r' \Longrightarrow \neg\ r =o\ r'$
using *ordLess-ordIso-trans ordIso-symmetric ordLess-irreflexive* **by** *blast*

lemma *not-ordLeq-iff-ordLess*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(\neg\ r' \leq o\ r) = (r <o\ r')$
using *assms not-ordLess-ordLeq ordLess-or-ordLeq* **by** *blast*

lemma *not-ordLess-iff-ordLeq*:
assumes *WELL*: *Well-order* r **and** *WELL'*: *Well-order* r'
shows $(\neg\ r' <o\ r) = (r \leq o\ r')$
using *assms not-ordLess-ordLeq ordLess-or-ordLeq* **by** *blast*

lemma *ordLess-transitive*[*trans*]:

$[\![r <o r'; r' <o r'']\!] \Longrightarrow r <o r''$
using *ordLess-ordLeq-trans ordLeq-iff-ordLess-or-ordIso* **by** *blast*

corollary *ordLess-trans*: *trans ordLess*
unfolding *trans-def* **using** *ordLess-transitive* **by** *blast*

lemmas *ordIso-equivalence* = *ordIso-transitive ordIso-reflexive ordIso-symmetric*

lemma *ordIso-imp-ordLeq*:
$r =o r' \Longrightarrow r \leq o\ r'$
using *ordIso-iff-ordLeq* **by** *blast*

lemma *ordLess-imp-ordLeq*:
$r <o r' \Longrightarrow r \leq o\ r'$
using *ordLeq-iff-ordLess-or-ordIso* **by** *blast*

lemma *ofilter-subset-ordLeq*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A \leq B) = (Restr\ r\ A \leq o\ Restr\ r\ B)$
proof
assume $A \leq B$
thus *Restr r A* $\leq o$ *Restr r B*
unfolding *ordLeq-def* **using** *assms*
Well-order-Restr Well-order-Restr ofilter-subset-embed **by** *blast*
next
assume $*$: *Restr r A* $\leq o$ *Restr r B*
then obtain *f* **where** *embed* (*Restr r A*) (*Restr r B*) *f*
unfolding *ordLeq-def* **by** *blast*
{assume $B < A$
hence *Restr r B* $<o$ *Restr r A*
unfolding *ordLess-def* **using** *assms*
Well-order-Restr Well-order-Restr ofilter-subset-embedS **by** *blast*
hence *False* **using** $*$ *not-ordLess-ordLeq* **by** *blast*
}
thus $A \leq B$ **using** *OFA OFB WELL*
wo-rel-def[*of r*] *wo-rel.ofilter-linord*[*of r A B*] **by** *blast*
qed

lemma *ofilter-subset-ordLess*:
assumes *WELL*: *Well-order r* **and**
OFA: *wo-rel.ofilter r A* **and** *OFB*: *wo-rel.ofilter r B*
shows $(A < B) = (Restr\ r\ A <o\ Restr\ r\ B)$
proof−
let *?rA* = *Restr r A* **let** *?rB* = *Restr r B*
have *1*: *Well-order ?rA* $\wedge$ *Well-order ?rB*
using *WELL Well-order-Restr* **by** *blast*
have $(A < B) = (\neg\ B \leq A)$ **using** *assms*
wo-rel-def wo-rel.ofilter-linord[*of r A B*] **by** *blast*

also have ... = (¬ *Restr r B* ≤*o Restr r A*)
using *assms ofilter-subset-ordLeq* **by** *blast*
also have ... = (*Restr r A* <*o Restr r B*)
using *1 not-ordLeq-iff-ordLess* **by** *blast*
finally show *?thesis* **.**
qed

lemma *ofilter-ordLess*:
⟦*Well-order r*; *wo-rel.ofilter r A*⟧ ⟹ (*A* < *Field r*) = (*Restr r A* <*o r*)
by (*simp add*: *ofilter-subset-ordLess wo-rel.Field-ofilter*
wo-rel-def Restr-Field)

corollary *underS-Restr-ordLess*:
assumes *Well-order r* **and** *Field r* ≠ {}
shows *Restr r* (*underS r a*) <*o r*
proof−
have *underS r a* < *Field r* **using** *assms*
by (*simp add*: *underS-Field3*)
thus *?thesis* **using** *assms*
by (*simp add*: *ofilter-ordLess wo-rel.underS-ofilter wo-rel-def*)
qed

lemma *embed-ordLess-ofilterIncl*:
assumes
OL12: *r1* <*o r2* **and** *OL23*: *r2* <*o r3* **and**
EMB13: *embed r1 r3 f13* **and** *EMB23*: *embed r2 r3 f23*
shows (*f13*‘(*Field r1*), *f23*‘(*Field r2*)) ∈ (*ofilterIncl r3*)
proof−
have *OL13*: *r1* <*o r3*
using *OL12 OL23* **using** *ordLess-transitive* **by** *auto*
let *?A1* = *Field r1* **let** *?A2* =*Field r2* **let** *?A3* =*Field r3*
obtain *f12 g23* **where**
0: *Well-order r1* ∧ *Well-order r2* ∧ *Well-order r3* **and**
1: *embed r1 r2 f12* ∧ ¬(*bij-betw f12 ?A1 ?A2*) **and**
2: *embed r2 r3 g23* ∧ ¬(*bij-betw g23 ?A2 ?A3*)
using *OL12 OL23* **by** (*auto simp add*: *ordLess-def embedS-def*)
hence ∀ *a* ∈ *?A2*. *f23 a* = *g23 a*
using *EMB23 embed-unique*[*of r2 r3*] **by** *blast*
hence *3*: ¬(*bij-betw f23 ?A2 ?A3*)
using *2 bij-betw-cong*[*of ?A2 f23 g23*] **by** *blast*

have *4*: *wo-rel.ofilter r2* (*f12* ‘ *?A1*) ∧ *f12* ‘ *?A1* ≠ *?A2*
using *0 1 OL12* **by** (*simp add*: *embed-Field-ofilter ordLess-Field*)
have *5*: *wo-rel.ofilter r3* (*f23* ‘ *?A2*) ∧ *f23* ‘ *?A2* ≠ *?A3*
using *0 EMB23 OL23* **by** (*simp add*: *embed-Field-ofilter ordLess-Field*)
have *6*: *wo-rel.ofilter r3* (*f13* ‘ *?A1*) ∧ *f13* ‘ *?A1* ≠ *?A3*
using *0 EMB13 OL13* **by** (*simp add*: *embed-Field-ofilter ordLess-Field*)

have *f12* ‘ *?A1* < *?A2*

```
    using 0 4 by (auto simp add: wo-rel-def wo-rel.ofilter-def)
  moreover have inj-on f23 ?A2
    using EMB23 0 by (simp add: wo-rel-def embed-inj-on)
  ultimately
  have f23 ‘ (f12 ‘ ?A1) < f23 ‘ ?A2 by (simp add: image-strict-mono)
  moreover
  {have embed r1 r3 (f23 ∘ f12)
     using 1 EMB23 0 by (auto simp add: comp-embed)
   hence ∀ a ∈ ?A1. f23(f12 a) = f13 a
     using EMB13 0 embed-unique[of r1 r3 f23 ∘ f12 f13] by auto
   hence f23 ‘ (f12 ‘ ?A1) = f13 ‘ ?A1 by force
  }
  ultimately
  have f13 ‘ ?A1 < f23 ‘ ?A2 by simp

  with 5 6 show ?thesis
    unfolding ofilterIncl-def by auto
qed

lemma ordLess-iff-ordIso-Restr:
  assumes WELL: Well-order r and WELL': Well-order r'
  shows (r' <o r) = (∃ a ∈ Field r. r' =o Restr r (underS r a))
proof safe
  fix a assume *: a ∈ Field r and **: r' =o Restr r (underS r a)
  hence Restr r (underS r a) <o r using WELL underS-Restr-ordLess[of r] by
blast
  thus r' <o r using ** ordIso-ordLess-trans by blast
next
  assume r' <o r
  then obtain f where 1: Well-order r ∧ Well-order r' and
    2: embed r' r f ∧ f ‘ (Field r') ≠ Field r
    unfolding ordLess-def embedS-def[abs-def] bij-betw-def using embed-inj-on by
blast
  hence wo-rel.ofilter r (f ‘ (Field r')) using embed-Field-ofilter by blast
  then obtain a where 3: a ∈ Field r and 4: underS r a = f ‘ (Field r')
    using 1 2 by (auto simp add: wo-rel.ofilter-underS-Field wo-rel-def)
  have iso r' (Restr r (f ‘ (Field r'))) f
    using embed-implies-iso-Restr 2 assms by blast
  moreover have Well-order (Restr r (f ‘ (Field r')))
    using WELL Well-order-Restr by blast
  ultimately have r' =o Restr r (f ‘ (Field r'))
    using WELL' unfolding ordIso-def by auto
  hence r' =o Restr r (underS r a) using 4 by auto
  thus ∃ a ∈ Field r. r' =o Restr r (underS r a) using 3 by auto
qed

lemma internalize-ordLess:
  (r' <o r) = (∃ p. Field p < Field r ∧ r' =o p ∧ p <o r)
proof
```

assume $*$: $r' <o\ r$
hence *0*: *Well-order r* $\wedge$ *Well-order r'* **unfolding** *ordLess-def* **by** *auto*
with $*$ **obtain** *a* **where** *1*: $a \in$ *Field r* **and** *2*: $r' =o$ *Restr r* (*underS r a*)
using *ordLess-iff-ordIso-Restr* **by** *blast*
let *?p* = *Restr r* (*underS r a*)
have *wo-rel.ofilter r* (*underS r a*) **using** *0*
by (*simp add*: *wo-rel-def wo-rel.underS-ofilter*)
hence *Field ?p* = *underS r a* **using** *0 Field-Restr-ofilter* **by** *blast*
hence *Field ?p* $<$ *Field r* **using** *underS-Field2 1* **by** *fast*
moreover have *?p* $<o\ r$ **using** *underS-Restr-ordLess*[*of r a*] *0 1* **by** *blast*
ultimately show $\exists p.$ *Field p* $<$ *Field r* $\wedge\ r' =o\ p \wedge p <o\ r$ **using** *2* **by** *blast*
next
assume $\exists p.$ *Field p* $<$ *Field r* $\wedge\ r' =o\ p \wedge p <o\ r$
thus $r' <o\ r$ **using** *ordIso-ordLess-trans* **by** *blast*
qed

lemma *internalize-ordLeq*:
$(r' \leq o\ r) = (\exists p.$ *Field p* $\leq$ *Field r* $\wedge\ r' =o\ p \wedge p \leq o\ r)$
proof
assume $*$: $r' \leq o\ r$
moreover
have $r' <o\ r \Longrightarrow \exists p.$ *Field p* $\leq$ *Field r* $\wedge\ r' =o\ p \wedge p \leq o\ r$
using *ordLeq-iff-ordLess-or-ordIso internalize-ordLess*[*of r' r*] **by** *blast*
moreover
have $r \leq o\ r$ **using** $*$ *ordLeq-def ordLeq-reflexive* **by** *blast*
ultimately show $\exists p.$ *Field p* $\leq$ *Field r* $\wedge\ r' =o\ p \wedge p \leq o\ r$
using *ordLeq-iff-ordLess-or-ordIso* **by** *blast*
next
assume $\exists p.$ *Field p* $\leq$ *Field r* $\wedge\ r' =o\ p \wedge p \leq o\ r$
thus $r' \leq o\ r$ **using** *ordIso-ordLeq-trans* **by** *blast*
qed

lemma *ordLeq-iff-ordLess-Restr*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'*
shows $(r \leq o\ r') = (\forall a \in$ *Field r*. *Restr r* (*underS r a*) $<o\ r')$
proof *safe*
assume $*$: $r \leq o\ r'$
fix *a* **assume** $a \in$ *Field r*
hence *Restr r* (*underS r a*) $<o\ r$
using *WELL underS-Restr-ordLess*[*of r*] **by** *blast*
thus *Restr r* (*underS r a*) $<o\ r'$
using $*$ *ordLess-ordLeq-trans* **by** *blast*
next
assume $*$: $\forall a \in$ *Field r*. *Restr r* (*underS r a*) $<o\ r'$
{assume $r' <o\ r$
then obtain *a* **where** $a \in$ *Field r* $\wedge\ r' =o$ *Restr r* (*underS r a*)
using *assms ordLess-iff-ordIso-Restr* **by** *blast*
hence *False* **using** $*$ *not-ordLess-ordIso ordIso-symmetric* **by** *blast*
}

thus $r \leq o\ r'$ **using** *ordLess-or-ordLeq assms* **by** *blast*
qed

lemma *finite-ordLess-infinite*:
assumes *WELL*: *Well-order r* **and** *WELL'*: *Well-order r'* **and**
FIN: *finite*(*Field r*) **and** *INF*: ¬*finite*(*Field r'*)
shows $r <o\ r'$
proof−
{**assume** $r' \leq o\ r$
then obtain *h* **where** *inj-on h* (*Field r'*) ∧ *h* ‘ (*Field r'*) ≤ *Field r*
unfolding *ordLeq-def* **using** *assms embed-inj-on embed-Field* **by** *blast*
hence *False* **using** *finite-imageD finite-subset FIN INF* **by** *blast*
}
thus *?thesis* **using** *WELL WELL' ordLess-or-ordLeq* **by** *blast*
qed

lemma *finite-well-order-on-ordIso*:
assumes *FIN*: *finite A* **and**
WELL: *well-order-on A r* **and** *WELL'*: *well-order-on A r'*
shows $r =o\ r'$
proof−
have *0*: *Well-order r* ∧ *Well-order r'* ∧ *Field r* = *A* ∧ *Field r'* = *A*
using *assms well-order-on-Well-order* **by** *blast*
moreover
have ∀ *r r'*. *well-order-on A r* ∧ *well-order-on A r'* ∧ $r \leq o\ r'$
⟶ $r =o\ r'$
proof(*clarify*)
fix *r r'* **assume** ∗: *well-order-on A r* **and** ∗∗: *well-order-on A r'*
have *2*: *Well-order r* ∧ *Well-order r'* ∧ *Field r* = *A* ∧ *Field r'* = *A*
using ∗ ∗∗ *well-order-on-Well-order* **by** *blast*
assume $r \leq o\ r'$
then obtain *f* **where** *1*: *embed r r' f* **and**
inj-on f A ∧ *f* ‘ *A* ≤ *A*
unfolding *ordLeq-def* **using** *2 embed-inj-on embed-Field* **by** *blast*
hence *bij-betw f A A* **unfolding** *bij-betw-def* **using** *FIN endo-inj-surj* **by** *blast*
thus $r =o\ r'$ **unfolding** *ordIso-def iso-def*[*abs-def*] **using** *1 2* **by** *auto*
qed
ultimately show *?thesis* **using** *assms ordLeq-total ordIso-symmetric* **by** *blast*
qed

29.5 $<o$ is well-founded

Of course, it only makes sense to state that the $<o$ is well-founded on the restricted type *'a rel rel*. We prove this by first showing that, for any set of well-orders all embedded in a fixed well-order, the function mapping each well-order in the set to an order filter of the fixed well-order is compatible w.r.t. to $<o$ versus *strict inclusion*; and we already know that strict inclusion of order filters is well-founded.

definition *ord-to-filter* :: *'a rel ⇒ 'a rel ⇒ 'a set*
where *ord-to-filter r0 r ≡ (SOME f. embed r r0 f) ‘ (Field r)*

lemma *ord-to-filter-compat*:
compat (ordLess Int (ordLess⁻¹ ‘‘{r0} × ordLess⁻¹ ‘‘{r0}))
(ofilterIncl r0)
(ord-to-filter r0)
proof(*unfold compat-def ord-to-filter-def*, *clarify*)
fix *r1*::*'a rel* **and** *r2*::*'a rel*
let *?A1 = Field r1* **let** *?A2 =Field r2* **let** *?A0 =Field r0*
let *?phi10 = λ f10. embed r1 r0 f10* **let** *?f10 = SOME f. ?phi10 f*
let *?phi20 = λ f20. embed r2 r0 f20* **let** *?f20 = SOME f. ?phi20 f*
assume ∗: *r1 <o r0 r2 <o r0* **and** ∗∗: *r1 <o r2*
hence *(∃f. ?phi10 f) ∧ (∃f. ?phi20 f)*
by (*auto simp add*: *ordLess-def embedS-def*)
hence *?phi10 ?f10 ∧ ?phi20 ?f20* **by** (*auto simp add*: *someI-ex*)
thus (*?f10 ‘ ?A1*, *?f20 ‘ ?A2*) ∈ *ofilterIncl r0*
using ∗ ∗∗ **by** (*simp add*: *embed-ordLess-ofilterIncl*)
qed

theorem *wf-ordLess*: *wf ordLess*
proof−
{fix *r0* :: (*'a* × *'a*) *set*

let *?ordLess = ordLess*::(*'d rel* ∗ *'d rel*) *set*
let *?R = ?ordLess Int (?ordLess⁻¹ ‘‘{r0} × ?ordLess⁻¹ ‘‘{r0})*
{assume *Case1*: *Well-order r0*
hence *wf ?R*
using *wf-ofilterIncl*[*of r0*]
compat-wf[*of ?R ofilterIncl r0 ord-to-filter r0*]
ord-to-filter-compat[*of r0*] **by** *auto*
}
moreover
{assume *Case2*: ¬ *Well-order r0*
hence *?R = {}* **unfolding** *ordLess-def* **by** *auto*
hence *wf ?R* **by** *simp*
}
ultimately have *wf ?R* **by** *blast*
}
thus *?thesis* **by** (*simp add*: *trans-wf-iff ordLess-trans*)
qed

corollary *exists-minim-Well-order*:
assumes *NE*: *R ≠ {}* **and** *WELL*: *∀ r ∈ R. Well-order r*
shows *∃ r ∈ R. ∀ r' ∈ R. r ≤o r'*
proof−
obtain *r* **where** *r ∈ R ∧ (∀ r' ∈ R. ¬ r' <o r)*
using *NE spec*[*OF spec*[*OF subst*[*OF wf-eq-minimal*, *of %x. x*, *OF wf-ordLess*]],
of - R]

equals0I[*of R*] **by** *blast*
with *not-ordLeq-iff-ordLess WELL* **show** *?thesis* **by** *blast*
qed

29.6 Copy via direct images

The direct image operator is the dual of the inverse image operator *inv-image* from *Relation.thy*. It is useful for transporting a well-order between different types.

definition *dir-image* :: *'a rel* ⇒ (*'a* ⇒ *'a'*) ⇒ *'a' rel*
where
dir-image r f = {(*f a*, *f b*)| *a b*. (*a*,*b*) ∈ *r*}

lemma *dir-image-Field*:
Field(*dir-image r f*) = *f* ‘ (*Field r*)
unfolding *dir-image-def Field-def Range-def Domain-def* **by** *fast*

lemma *dir-image-minus-Id*:
inj-on f (*Field r*) ⟹ (*dir-image r f*) − *Id* = *dir-image* (*r* − *Id*) *f*
unfolding *inj-on-def Field-def dir-image-def* **by** *auto*

lemma *dir-image-subset*:
assumes *r* ⊆ *A* × *B*
shows *dir-image r f* ⊆ *f* ‘ *A* × *f* ‘ *B*
proof (*rule subsetI*)
fix *x*
assume *x* ∈ *dir-image r f*
then obtain *a b* **where** *x* = (*f a*, *f b*) **and** (*a*, *b*) ∈ *r*
unfolding *dir-image-def* **by** *blast*
thus *x* ∈ *f* ‘ *A* × *f* ‘ *B*
using ‹*r* ⊆ *A* × *B*› **by** *auto*
qed

lemma *Refl-dir-image*:
assumes *Refl r*
shows *Refl*(*dir-image r f*)
proof−
{fix *a' b'*
assume (*a'*,*b'*) ∈ *dir-image r f*
then obtain *a b* **where** *1*: *a'* = *f a* ∧ *b'* = *f b* ∧ (*a*,*b*) ∈ *r*
unfolding *dir-image-def* **by** *blast*
hence *a* ∈ *Field r* ∧ *b* ∈ *Field r* **using** *Field-def* **by** *fastforce*
hence (*a*,*a*) ∈ *r* ∧ (*b*,*b*) ∈ *r* **using** *assms* **by** (*simp add*: *refl-on-def*)
with *1* **have** (*a'*,*a'*) ∈ *dir-image r f* ∧ (*b'*,*b'*) ∈ *dir-image r f*
unfolding *dir-image-def* **by** *auto*
}
thus *?thesis*
by(*unfold refl-on-def Field-def Domain-def Range-def*, *auto*)
qed

lemma *trans-dir-image*:
assumes *TRANS*: *trans r* **and** *INJ*: *inj-on f* (*Field r*)
shows *trans*(*dir-image r f*)
unfolding *trans-def*
proof *safe*
fix *a' b' c'*
assume (*a'*,*b'*) ∈ *dir-image r f* (*b'*,*c'*) ∈ *dir-image r f*
then obtain *a b1 b2 c* **where** *1*: *a'* = *f a* ∧ *b'* = *f b1* ∧ *b'* = *f b2* ∧ *c'* = *f c*
and
2: (*a*,*b1*) ∈ *r* ∧ (*b2*,*c*) ∈ *r*
unfolding *dir-image-def* **by** *blast*
hence *b1* ∈ *Field r* ∧ *b2* ∈ *Field r*
unfolding *Field-def* **by** *auto*
hence *b1* = *b2* **using** *1 INJ* **unfolding** *inj-on-def* **by** *auto*
hence (*a*,*c*) ∈ *r* **using** *2 TRANS* **unfolding** *trans-def* **by** *blast*
thus (*a'*,*c'*) ∈ *dir-image r f*
unfolding *dir-image-def* **using** *1* **by** *auto*
qed

lemma *Preorder-dir-image*:
assumes *Preorder r* **and** *inj*: *inj-on f* (*Field r*)
shows *Preorder* (*dir-image r f*)
unfolding *preorder-on-def*
proof (*intro conjI*)
have *r* ⊆ *Field r* × *Field r* **and** *Refl r* **and** *trans r*
using ‹*Preorder r*› **by** (*simp-all only*: *preorder-on-def*)

show *dir-image r f* ⊆ *Field* (*dir-image r f*) × *Field* (*dir-image r f*)
using *dir-image-subset*[*OF* ‹*r* ⊆ *Field r* × *Field r*›]
unfolding *dir-image-Field* **.**

show *Refl* (*dir-image r f*)
using *Refl-dir-image*[*OF* ‹*Refl r*›] **.**

show *trans* (*dir-image r f*)
using *trans-dir-image*[*OF* ‹*trans r*› *inj*] **.**
qed

lemma *antisym-dir-image*:
assumes *AN*: *antisym r* **and** *INJ*: *inj-on f* (*Field r*)
shows *antisym*(*dir-image r f*)
unfolding *antisym-def*
proof *safe*
fix *a' b'*
assume (*a'*,*b'*) ∈ *dir-image r f* (*b'*,*a'*) ∈ *dir-image r f*
then obtain *a1 b1 a2 b2* **where** *1*: *a'* = *f a1* ∧ *a'* = *f a2* ∧ *b'* = *f b1* ∧ *b'* = *f b2* **and**
2: (*a1*,*b1*) ∈ *r* ∧ (*b2*,*a2*) ∈ *r* **and**

3: $\{a1,a2,b1,b2\} \leq$ *Field r*
unfolding *dir-image-def Field-def* **by** *blast*
hence *a1* $=$ *a2* $\wedge$ *b1* $=$ *b2* **using** *INJ* **unfolding** *inj-on-def* **by** *auto*
hence *a1* $=$ *b2* **using** *2 AN* **unfolding** *antisym-def* **by** *auto*
thus $a' = b'$ **using** *1* **by** *auto*
qed

lemma *Partial-order-dir-image*:
$[\![$*Partial-order r*; *inj-on f* (*Field r*)$]\!] \Longrightarrow$ *Partial-order* (*dir-image r f*)
by (*simp add*: *partial-order-on-def Preorder-dir-image antisym-dir-image*)

lemma *Total-dir-image*:
assumes *TOT*: *Total r* **and** *INJ*: *inj-on f* (*Field r*)
shows *Total*(*dir-image r f*)
proof(*unfold total-on-def*, *intro ballI impI*)
fix a' b'
assume $a' \in$ *Field* (*dir-image r f*) $b' \in$ *Field* (*dir-image r f*)
then obtain *a* **and** *b* **where** *1*: $a \in$ *Field r* $\wedge$ $b \in$ *Field r* $\wedge$ $f\ a = a' \wedge f\ b = b'$
unfolding *dir-image-Field*[*of r f*] **by** *blast*
moreover assume $a' \neq b'$
ultimately have $a \neq b$ **using** *INJ* **unfolding** *inj-on-def* **by** *auto*
hence $(a,b) \in r \vee (b,a) \in r$ **using** *1 TOT* **unfolding** *total-on-def* **by** *auto*
thus $(a',b') \in$ *dir-image r f* $\vee$ $(b',a') \in$ *dir-image r f*
using *1* **unfolding** *dir-image-def* **by** *auto*
qed

lemma *Linear-order-dir-image*:
$[\![$*Linear-order r*; *inj-on f* (*Field r*)$]\!] \Longrightarrow$ *Linear-order* (*dir-image r f*)
by (*simp add*: *linear-order-on-def Partial-order-dir-image Total-dir-image*)

lemma *wf-dir-image*:
assumes *WF*: *wf r* **and** *INJ*: *inj-on f* (*Field r*)
shows *wf*(*dir-image r f*)
proof(*unfold wf-eq-minimal2*, *intro allI impI*, *elim conjE*)
fix A'::'*b set*
assume *SUB*: $A' \leq$ *Field*(*dir-image r f*) **and** *NE*: $A' \neq \{\}$
obtain *A* **where** *A-def*: $A = \{a \in$ *Field r*. $f\ a \in A'\}$ **by** *blast*
have $A \neq \{\} \wedge A \leq$ *Field r* **using** *A-def SUB NE* **by** (*auto simp*: *dir-image-Field*)
then obtain *a* **where** *1*: $a \in A \wedge (\forall\, b \in A.\ (b,a) \notin r)$
using *spec*[*OF WF*[*unfolded wf-eq-minimal2*], *of A*] **by** *blast*
have $\forall\, b' \in A'.\ (b',f\ a) \notin$ *dir-image r f*
proof(*clarify*)
fix b' **assume** $*$: $b' \in A'$ **and** $**$: $(b',f\ a) \in$ *dir-image r f*
obtain *b1 a1* **where** *2*: $b' = f\ b1 \wedge f\ a = f\ a1$ **and**
3: $(b1,a1) \in r \wedge \{a1,b1\} \leq$ *Field r*
using $**$ **unfolding** *dir-image-def Field-def* **by** *blast*
hence $a = a1$ **using** *1 A-def INJ* **unfolding** *inj-on-def* **by** *auto*
hence *b1* $\in A \wedge (b1,a) \in r$ **using** *2 3 A-def* $*$ **by** *auto*
with *1* **show** *False* **by** *auto*

qed
thus $\exists\, a'{\in}A'.\ \forall\, b'{\in}A'.\ (b',\ a') \notin \mathit{dir\text{-}image}\ r\ f$
using *A-def 1* **by** *blast*
qed

lemma *Well-order-dir-image*:
$[\![\mathit{Well\text{-}order}\ r;\ \mathit{inj\text{-}on}\ f\ (\mathit{Field}\ r)]\!] \Longrightarrow \mathit{Well\text{-}order}\ (\mathit{dir\text{-}image}\ r\ f)$
unfolding *well-order-on-def*
using *Linear-order-dir-image*[*of r f*] *wf-dir-image*[*of r − Id f*]
dir-image-minus-Id[*of f r*]
inj-on-subset[*of f Field r Field*(*r − Id*)]
mono-Field[*of r − Id r*] **by** *auto*

lemma *dir-image-bij-betw*:
$[\![\mathit{inj\text{-}on}\ f\ (\mathit{Field}\ r)]\!] \Longrightarrow \mathit{bij\text{-}betw}\ f\ (\mathit{Field}\ r)\ (\mathit{Field}\ (\mathit{dir\text{-}image}\ r\ f))$
unfolding *bij-betw-def* **by** (*simp add*: *dir-image-Field order-on-defs*)

lemma *dir-image-compat*:
compat r (*dir-image r f*) *f*
unfolding *compat-def dir-image-def* **by** *auto*

lemma *dir-image-iso*:
$[\![\mathit{Well\text{-}order}\ r;\ \mathit{inj\text{-}on}\ f\ (\mathit{Field}\ r)]\!] \Longrightarrow \mathit{iso}\ r\ (\mathit{dir\text{-}image}\ r\ f)\ f$
using *iso-iff3 dir-image-compat dir-image-bij-betw Well-order-dir-image* **by** *blast*

lemma *dir-image-ordIso*:
$[\![\mathit{Well\text{-}order}\ r;\ \mathit{inj\text{-}on}\ f\ (\mathit{Field}\ r)]\!] \Longrightarrow r =o\ \mathit{dir\text{-}image}\ r\ f$
unfolding *ordIso-def* **using** *dir-image-iso Well-order-dir-image* **by** *blast*

lemma *Well-order-iso-copy*:
assumes *WELL*: *well-order-on A r* **and** *BIJ*: *bij-betw f A A′*
shows $\exists\, r'.\ \mathit{well\text{-}order\text{-}on}\ A'\ r' \wedge r =o\ r'$
proof−
let *?r′* = *dir-image r f*
have *1*: $A = \mathit{Field}\ r \wedge \mathit{Well\text{-}order}\ r$
using *WELL well-order-on-Well-order* **by** *blast*
hence *2*: *iso r ?r′ f*
using *dir-image-iso* **using** *BIJ* **unfolding** *bij-betw-def* **by** *auto*
hence *f* ‘ (*Field r*) = *Field ?r′* **using** *1 iso-iff*[*of r ?r′*] **by** *blast*
hence *Field ?r′* = *A′*
using *1 BIJ* **unfolding** *bij-betw-def* **by** *auto*
moreover have *Well-order ?r′*
using *1 Well-order-dir-image BIJ* **unfolding** *bij-betw-def* **by** *blast*
ultimately show *?thesis* **unfolding** *ordIso-def* **using** *1 2* **by** *blast*
qed

29.7 Bounded square

This construction essentially defines, for an order relation r, a lexicographic order *bsqr r* on (*Field r*) × (*Field r*), applying the following criteria (in this order):

- compare the maximums;
- compare the first components;
- compare the second components.

The only application of this construction that we are aware of is at proving that the square of an infinite set has the same cardinal as that set. The essential property required there (and which is ensured by this construction) is that any proper order filter of the product order is included in a rectangle, i.e., in a product of proper filters on the original relation (assumed to be a well-order).

definition *bsqr* :: *'a rel => ('a * 'a)rel*
where
bsqr r = {((a1,a2),(b1,b2)).
{a1,a2,b1,b2} ≤ Field r ∧
(a1 = b1 ∧ a2 = b2 ∨
(wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r − Id ∨
wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id ∨
wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) ∈ r − Id
)}

lemma *Field-bsqr*:
Field (bsqr r) = Field r × Field r
proof
show *Field (bsqr r) ≤ Field r × Field r*
proof−
{fix *a1 a2* **assume** *(a1,a2) ∈ Field (bsqr r)*
moreover
have ⋀ *b1 b2. ((a1,a2),(b1,b2)) ∈ bsqr r ∨ ((b1,b2),(a1,a2)) ∈ bsqr r ⟹ a1 ∈ Field r ∧ a2 ∈ Field r* **unfolding** *bsqr-def* **by** *auto*
ultimately have *a1 ∈ Field r ∧ a2 ∈ Field r* **unfolding** *Field-def* **by** *auto*
}
thus *?thesis* **unfolding** *Field-def* **by** *force*
qed
next
show *Field r × Field r ≤ Field (bsqr r)*
proof *safe*
fix *a1 a2* **assume** *a1 ∈ Field r* **and** *a2 ∈ Field r*
hence *((a1,a2),(a1,a2)) ∈ bsqr r* **unfolding** *bsqr-def* **by** *blast*
thus *(a1,a2) ∈ Field (bsqr r)* **unfolding** *Field-def* **by** *auto*
qed

qed

lemma *bsqr-subset*:
assumes $r \subseteq Field\ r \times Field\ r$
shows $bsqr\ r \subseteq Field\ (bsqr\ r) \times Field\ (bsqr\ r)$
using ‹$r \subseteq Field\ r \times Field\ r$›
unfolding *Field-bsqr*
by (*auto simp add*: *bsqr-def*)

lemma *bsqr-Refl*: *Refl*(*bsqr r*)
by(*unfold refl-on-def Field-bsqr*, *auto simp add*: *bsqr-def*)

lemma *bsqr-Trans*:
assumes *Well-order r*
shows *trans* (*bsqr r*)
unfolding *trans-def*
proof *safe*

have *Well*: *wo-rel r* **using** *assms wo-rel-def* **by** *auto*
hence *Trans*: *trans r* **using** *wo-rel.TRANS* **by** *auto*
have *Anti*: *antisym r* **using** *wo-rel.ANTISYM Well* **by** *auto*
hence *TransS*: *trans*($r - Id$) **using** *Trans* **by** (*simp add*: *trans-diff-Id*)

fix *a1 a2 b1 b2 c1 c2*
assume ∗: ((*a1*,*a2*),(*b1*,*b2*)) ∈ *bsqr r* **and** ∗∗: ((*b1*,*b2*),(*c1*,*c2*)) ∈ *bsqr r*
hence *0*: {*a1*,*a2*,*b1*,*b2*,*c1*,*c2*} ≤ *Field r* **unfolding** *bsqr-def* **by** *auto*
have *1*: *a1* = *b1* ∧ *a2* = *b2* ∨ (*wo-rel.max2 r a1 a2*, *wo-rel.max2 r b1 b2*) ∈ *r* − *Id* ∨
wo-rel.max2 r a1 a2 = *wo-rel.max2 r b1 b2* ∧ (*a1*,*b1*) ∈ *r* − *Id* ∨
wo-rel.max2 r a1 a2 = *wo-rel.max2 r b1 b2* ∧ *a1* = *b1* ∧ (*a2*,*b2*) ∈ *r* − *Id*
using ∗ **unfolding** *bsqr-def* **by** *auto*
have *2*: *b1* = *c1* ∧ *b2* = *c2* ∨ (*wo-rel.max2 r b1 b2*, *wo-rel.max2 r c1 c2*) ∈ *r* − *Id* ∨
wo-rel.max2 r b1 b2 = *wo-rel.max2 r c1 c2* ∧ (*b1*,*c1*) ∈ *r* − *Id* ∨
wo-rel.max2 r b1 b2 = *wo-rel.max2 r c1 c2* ∧ *b1* = *c1* ∧ (*b2*,*c2*) ∈ *r* − *Id*
using ∗∗ **unfolding** *bsqr-def* **by** *auto*
show ((*a1*,*a2*),(*c1*,*c2*)) ∈ *bsqr r*
proof−
{assume *Case1*: *a1* = *b1* ∧ *a2* = *b2*
hence *?thesis* **using** ∗∗ **by** *simp*
}
moreover
{assume *Case2*: (*wo-rel.max2 r a1 a2*, *wo-rel.max2 r b1 b2*) ∈ *r* − *Id*
{assume *Case21*: *b1* = *c1* ∧ *b2* = *c2*
hence *?thesis* **using** ∗ **by** *simp*
}
moreover

{assume *Case22*: (*wo-rel.max2 r b1 b2*, *wo-rel.max2 r c1 c2*) ∈ *r* − *Id*
hence (*wo-rel.max2 r a1 a2*, *wo-rel.max2 r c1 c2*) ∈ *r* − *Id*
using *Case2 TransS trans-def*[*of r* − *Id*] **by** *blast*
hence *?thesis* **using** *0* **unfolding** *bsqr-def* **by** *auto*
}
moreover
{assume *Case23-4*: *wo-rel.max2 r b1 b2* = *wo-rel.max2 r c1 c2*
hence *?thesis* **using** *Case2 0* **unfolding** *bsqr-def* **by** *auto*
}
ultimately have *?thesis* **using** *0 2* **by** *auto*
}
moreover
{assume *Case3*: *wo-rel.max2 r a1 a2* = *wo-rel.max2 r b1 b2* ∧ (*a1*,*b1*) ∈ *r* − *Id*
{assume *Case31*: *b1* = *c1* ∧ *b2* = *c2*
hence *?thesis* **using** ∗ **by** *simp*
}
moreover
{assume *Case32*: (*wo-rel.max2 r b1 b2*, *wo-rel.max2 r c1 c2*) ∈ *r* − *Id*
hence *?thesis* **using** *Case3 0* **unfolding** *bsqr-def* **by** *auto*
}
moreover
{assume *Case33*: *wo-rel.max2 r b1 b2* = *wo-rel.max2 r c1 c2* ∧ (*b1*,*c1*) ∈ *r* − *Id*
hence (*a1*,*c1*) ∈ *r* − *Id*
using *Case3 TransS trans-def*[*of r* − *Id*] **by** *blast*
hence *?thesis* **using** *Case3 Case33 0* **unfolding** *bsqr-def* **by** *auto*
}
moreover
{assume *Case33*: *wo-rel.max2 r b1 b2* = *wo-rel.max2 r c1 c2* ∧ *b1* = *c1*
hence *?thesis* **using** *Case3 0* **unfolding** *bsqr-def* **by** *auto*
}
ultimately have *?thesis* **using** *0 2* **by** *auto*
}
moreover
{assume *Case4*: *wo-rel.max2 r a1 a2* = *wo-rel.max2 r b1 b2* ∧ *a1* = *b1* ∧ (*a2*,*b2*) ∈ *r* − *Id*
{assume *Case41*: *b1* = *c1* ∧ *b2* = *c2*
hence *?thesis* **using** ∗ **by** *simp*
}
moreover
{assume *Case42*: (*wo-rel.max2 r b1 b2*, *wo-rel.max2 r c1 c2*) ∈ *r* − *Id*
hence *?thesis* **using** *Case4 0* **unfolding** *bsqr-def* **by** *force*
}
moreover
{assume *Case43*: *wo-rel.max2 r b1 b2* = *wo-rel.max2 r c1 c2* ∧ (*b1*,*c1*) ∈ *r* − *Id*
hence *?thesis* **using** *Case4 0* **unfolding** *bsqr-def* **by** *auto*
}

```
    moreover
    {assume Case44: wo-rel.max2 r b1 b2 = wo-rel.max2 r c1 c2 ∧ b1 = c1 ∧
(b2,c2) ∈ r − Id
     hence (a2,c2) ∈ r − Id
      using Case4 TransS trans-def[of r − Id] by blast
     hence ?thesis using Case4 Case44 0 unfolding bsqr-def by auto
    }
    ultimately have ?thesis using 0 2 by auto
   }
   ultimately show ?thesis using 0 1 by auto
  qed
qed

lemma bsqr-antisym:
  assumes Well-order r
  shows antisym (bsqr r)
proof(unfold antisym-def, clarify)

  have Well: wo-rel r using assms wo-rel-def by auto
  hence Trans: trans r using wo-rel.TRANS by auto
  have Anti: antisym r using wo-rel.ANTISYM Well by auto
  hence TransS: trans(r − Id) using Trans by (simp add: trans-diff-Id)
  hence IrrS: ∀ a b. ¬((a,b) ∈ r − Id ∧ (b,a) ∈ r − Id)
    using Anti trans-def[of r − Id] antisym-def[of r − Id] by blast

  fix a1 a2 b1 b2
  assume ∗: ((a1,a2),(b1,b2)) ∈ bsqr r and ∗∗: ((b1,b2),(a1,a2)) ∈ bsqr r
  hence {a1,a2,b1,b2} ≤ Field r unfolding bsqr-def by auto
  moreover
  have a1 = b1 ∧ a2 = b2 ∨ (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r −
Id ∨
         wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ (a1,b1) ∈ r − Id ∨
         wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2 ∧ a1 = b1 ∧ (a2,b2) ∈ r −
Id
    using ∗ unfolding bsqr-def by auto
  moreover
  have b1 = a1 ∧ b2 = a2 ∨ (wo-rel.max2 r b1 b2, wo-rel.max2 r a1 a2) ∈ r −
Id ∨
         wo-rel.max2 r b1 b2 = wo-rel.max2 r a1 a2 ∧ (b1,a1) ∈ r − Id ∨
         wo-rel.max2 r b1 b2 = wo-rel.max2 r a1 a2 ∧ b1 = a1 ∧ (b2,a2) ∈ r −
Id
    using ∗∗ unfolding bsqr-def by auto
  ultimately show a1 = b1 ∧ a2 = b2
    using IrrS by auto
qed

lemma bsqr-Total:
  assumes Well-order r
  shows Total(bsqr r)
```

proof−

have *Well*: *wo-rel r* **using** *assms wo-rel-def* **by** *auto*
hence *Total*: $\forall\, a \in \mathit{Field}\ r.\ \forall\, b \in \mathit{Field}\ r.\ (a,b) \in r \vee (b,a) \in r$
using *wo-rel.TOTALS* **by** *auto*

{fix *a1 a2 b1 b2* **assume** $\{(a1,a2), (b1,b2)\} \leq \mathit{Field}(\mathit{bsqr}\ r)$
hence *0*: $a1 \in \mathit{Field}\ r \wedge a2 \in \mathit{Field}\ r \wedge b1 \in \mathit{Field}\ r \wedge b2 \in \mathit{Field}\ r$
using *Field-bsqr* **by** *blast*
have $((a1,a2) = (b1,b2) \vee ((a1,a2),(b1,b2)) \in \mathit{bsqr}\ r \vee ((b1,b2),(a1,a2)) \in \mathit{bsqr}\ r)$
proof(*rule wo-rel.cases-Total*[*of r a1 a2*], *clarsimp simp add*: *Well*, *simp add*: *0*)

assume *Case1*: $(a1,a2) \in r$
hence *1*: *wo-rel.max2 r a1 a2* = *a2*
using *Well 0* **by** (*simp add*: *wo-rel.max2-equals2*)
show *?thesis*
proof(*rule wo-rel.cases-Total*[*of r b1 b2*], *clarsimp simp add*: *Well*, *simp add*: *0*)
assume *Case11*: $(b1,b2) \in r$
hence *2*: *wo-rel.max2 r b1 b2* = *b2*
using *Well 0* **by** (*simp add*: *wo-rel.max2-equals2*)
show *?thesis*
proof(*rule wo-rel.cases-Total3*[*of r a2 b2*], *clarsimp simp add*: *Well*, *simp add*: *0*)
assume *Case111*: $(a2,b2) \in r - \mathit{Id} \vee (b2,a2) \in r - \mathit{Id}$
thus *?thesis* **using** *0 1 2* **unfolding** *bsqr-def* **by** *auto*
next
assume *Case112*: *a2* = *b2*
show *?thesis*
proof(*rule wo-rel.cases-Total3*[*of r a1 b1*], *clarsimp simp add*: *Well*, *simp add*: *0*)
assume *Case1121*: $(a1,b1) \in r - \mathit{Id} \vee (b1,a1) \in r - \mathit{Id}$
thus *?thesis* **using** *0 1 2 Case112* **unfolding** *bsqr-def* **by** *auto*
next
assume *Case1122*: *a1* = *b1*
thus *?thesis* **using** *Case112* **by** *auto*
qed
qed
next
assume *Case12*: $(b2,b1) \in r$
hence *3*: *wo-rel.max2 r b1 b2* = *b1* **using** *Well 0* **by** (*simp add*: *wo-rel.max2-equals1*)
show *?thesis*
proof(*rule wo-rel.cases-Total3*[*of r a2 b1*], *clarsimp simp add*: *Well*, *simp add*: *0*)
assume *Case121*: $(a2,b1) \in r - \mathit{Id} \vee (b1,a2) \in r - \mathit{Id}$
thus *?thesis* **using** *0 1 3* **unfolding** *bsqr-def* **by** *auto*
next

```
        assume Case122: a2 = b1
        show ?thesis
        proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
          assume Case1221: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
          thus ?thesis using 0 1 3 Case122 unfolding bsqr-def by auto
        next
          assume Case1222: a1 = b1
          show ?thesis
         proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
            assume Case12221: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id
             thus ?thesis using 0 1 3 Case122 Case1222 unfolding bsqr-def by
auto
          next
            assume Case12222: a2 = b2
            thus ?thesis using Case122 Case1222 by auto
          qed
        qed
      qed
    qed
  next
    assume Case2: (a2,a1) ∈ r
   hence 1: wo-rel.max2 r a1 a2 = a1 using Well 0 by (simp add: wo-rel.max2-equals1)
    show ?thesis
    proof(rule wo-rel.cases-Total[of r b1 b2], clarsimp simp add: Well, simp add:
0)
      assume Case21: (b1,b2) ∈ r
    hence 2: wo-rel.max2 r b1 b2 = b2 using Well 0 by (simp add: wo-rel.max2-equals2)
      show ?thesis
       proof(rule wo-rel.cases-Total3[of r a1 b2], clarsimp simp add: Well, simp
add: 0)
        assume Case211: (a1,b2) ∈ r - Id ∨ (b2,a1) ∈ r - Id
        thus ?thesis using 0 1 2 unfolding bsqr-def by auto
       next
        assume Case212: a1 = b2
        show ?thesis
        proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
          assume Case2121: (a1,b1) ∈ r - Id ∨ (b1,a1) ∈ r - Id
          thus ?thesis using 0 1 2 Case212 unfolding bsqr-def by auto
        next
          assume Case2122: a1 = b1
          show ?thesis
         proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
            assume Case21221: (a2,b2) ∈ r - Id ∨ (b2,a2) ∈ r - Id
             thus ?thesis using 0 1 2 Case212 Case2122 unfolding bsqr-def by
auto
```

```
          next
            assume Case21222: a2 = b2
            thus ?thesis using Case2122 Case212 by auto
          qed
        qed
      next
        assume Case22: (b2,b1) ∈ r
           hence 3: wo-rel.max2 r b1 b2 = b1  using Well 0 by (simp add:
wo-rel.max2-equals1)
        show ?thesis
         proof(rule wo-rel.cases-Total3[of r a1 b1], clarsimp simp add: Well, simp
add: 0)
          assume Case221: (a1,b1) ∈ r − Id ∨ (b1,a1) ∈ r − Id
          thus ?thesis using 0 1 3 unfolding bsqr-def by auto
        next
          assume Case222: a1 = b1
          show ?thesis
          proof(rule wo-rel.cases-Total3[of r a2 b2], clarsimp simp add: Well, simp
add: 0)
            assume Case2221: (a2,b2) ∈ r − Id ∨ (b2,a2) ∈ r − Id
            thus ?thesis using 0 1 3 Case222 unfolding bsqr-def by auto
          next
            assume Case2222: a2 = b2
            thus ?thesis using Case222 by auto
          qed
        qed
      qed
    qed
  }
  thus ?thesis unfolding total-on-def by fast
qed

lemma bsqr-Linear-order:
  assumes Well-order r
  shows Linear-order(bsqr r)
proof −
  have r ⊆ Field r × Field r
    using ‹Well-order r›
    by (simp only: order-on-defs)

  thus ?thesis
    unfolding order-on-defs
    using assms bsqr-Refl bsqr-Trans bsqr-antisym bsqr-Total bsqr-subset[OF ‹r ⊆
Field r × Field r›]
    by auto
qed

lemma bsqr-Well-order:
```

assumes *Well-order r*
shows *Well-order*(*bsqr r*)
using *assms*
proof(*simp add*: *bsqr-Linear-order Linear-order-Well-order-iff*, *intro allI impI*)
have *0*: $\forall\, A \leq \mathit{Field}\ r.\ A \neq \{\} \longrightarrow (\exists\, a \in A.\ \forall\, a' \in A.\ (a,a') \in r)$
using *assms well-order-on-def Linear-order-Well-order-iff* **by** *blast*
fix *D* **assume** $*$: $D \leq \mathit{Field}\ (\mathit{bsqr}\ r)$ **and** $**$: $D \neq \{\}$
hence *1*: $D \leq \mathit{Field}\ r \times \mathit{Field}\ r$ **unfolding** *Field-bsqr* **by** *simp*

obtain *M* **where** *M-def*: $M = \{\mathit{wo\text{-}rel.max2}\ r\ a1\ a2 |\ a1\ a2.\ (a1,a2) \in D\}$ **by** *blast*
have $M \neq \{\}$ **using** *1 M-def* $**$ **by** *auto*
moreover
have $M \leq \mathit{Field}\ r$ **unfolding** *M-def*
using *1 assms wo-rel-def*[*of r*] *wo-rel.max2-among*[*of r*] **by** *fastforce*
ultimately obtain *m* **where** *m-min*: $m \in M \wedge (\forall\, a \in M.\ (m,a) \in r)$
using *0* **by** *blast*

obtain *A1* **where** *A1-def*: $A1 = \{a1.\ \exists\, a2.\ (a1,a2) \in D \wedge \mathit{wo\text{-}rel.max2}\ r\ a1\ a2 = m\}$ **by** *blast*
have $A1 \leq \mathit{Field}\ r$ **unfolding** *A1-def* **using** *1* **by** *auto*
moreover have $A1 \neq \{\}$ **unfolding** *A1-def* **using** *m-min* **unfolding** *M-def* **by** *blast*
ultimately obtain *a1* **where** *a1-min*: $a1 \in A1 \wedge (\forall\, a \in A1.\ (a1,a) \in r)$
using *0* **by** *blast*

obtain *A2* **where** *A2-def*: $A2 = \{a2.\ (a1,a2) \in D \wedge \mathit{wo\text{-}rel.max2}\ r\ a1\ a2 = m\}$ **by** *blast*
have $A2 \leq \mathit{Field}\ r$ **unfolding** *A2-def* **using** *1* **by** *auto*
moreover have $A2 \neq \{\}$ **unfolding** *A2-def*
using *m-min a1-min* **unfolding** *A1-def M-def* **by** *blast*
ultimately obtain *a2* **where** *a2-min*: $a2 \in A2 \wedge (\forall\, a \in A2.\ (a2,a) \in r)$
using *0* **by** *blast*

have *2*: $\mathit{wo\text{-}rel.max2}\ r\ a1\ a2 = m$
using *a1-min a2-min* **unfolding** *A1-def A2-def* **by** *auto*
have *3*: $(a1,a2) \in D$ **using** *a2-min* **unfolding** *A2-def* **by** *auto*

moreover
{fix *b1 b2* **assume** $***$: $(b1,b2) \in D$
hence *4*: $\{a1,a2,b1,b2\} \leq \mathit{Field}\ r$ **using** *1 3* **by** *blast*
have *5*: $(\mathit{wo\text{-}rel.max2}\ r\ a1\ a2,\ \mathit{wo\text{-}rel.max2}\ r\ b1\ b2) \in r$
using $***$ *a1-min a2-min m-min* **unfolding** *A1-def A2-def M-def* **by** *auto*
have $((a1,a2),(b1,b2)) \in \mathit{bsqr}\ r$
proof(*cases wo-rel.max2 r a1 a2 = wo-rel.max2 r b1 b2*)
assume *Case1*: $\mathit{wo\text{-}rel.max2}\ r\ a1\ a2 \neq \mathit{wo\text{-}rel.max2}\ r\ b1\ b2$
thus *?thesis* **unfolding** *bsqr-def* **using** *4 5* **by** *auto*
next
assume *Case2*: $\mathit{wo\text{-}rel.max2}\ r\ a1\ a2 = \mathit{wo\text{-}rel.max2}\ r\ b1\ b2$

```
    hence b1 ∈ A1 unfolding A1-def using 2 *** by auto
    hence 6: (a1,b1) ∈ r using a1-min by auto
    show ?thesis
    proof(cases a1 = b1)
     assume Case21: a1 ≠ b1
     thus ?thesis unfolding bsqr-def using 4 Case2 6 by auto
    next
     assume Case22: a1 = b1
     hence b2 ∈ A2 unfolding A2-def using 2 *** Case2 by auto
     hence 7: (a2,b2) ∈ r using a2-min by auto
     thus ?thesis unfolding bsqr-def using 4 7 Case2 Case22 by auto
    qed
   qed
  }

  ultimately show ∃ d ∈ D. ∀ d' ∈ D. (d,d') ∈ bsqr r by fastforce
qed

lemma bsqr-max2:
  assumes WELL: Well-order r and LEQ: ((a1,a2),(b1,b2)) ∈ bsqr r
  shows (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r
proof−
  have {(a1,a2),(b1,b2)} ≤ Field(bsqr r)
    using LEQ unfolding Field-def by auto
  hence {a1,a2,b1,b2} ≤ Field r unfolding Field-bsqr by auto
  hence {wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2} ≤ Field r
    using WELL wo-rel-def[of r] wo-rel.max2-among[of r] by fastforce
  moreover have (wo-rel.max2 r a1 a2, wo-rel.max2 r b1 b2) ∈ r ∨ wo-rel.max2
r a1 a2 = wo-rel.max2 r b1 b2
    using LEQ unfolding bsqr-def by auto
  ultimately show ?thesis using WELL unfolding order-on-defs refl-on-def by
auto
qed

lemma bsqr-ofilter:
  assumes WELL: Well-order r and
    OF: wo-rel.ofilter (bsqr r) D and SUB: D < Field r × Field r and
    NE: ¬ (∃ a. Field r = under r a)
  shows ∃ A. wo-rel.ofilter r A ∧ A < Field r ∧ D ≤ A × A
proof−
  let ?r' = bsqr r
  have Well: wo-rel r using WELL wo-rel-def by blast
  hence Trans: trans r using wo-rel.TRANS by blast
  have Well': Well-order ?r' ∧ wo-rel ?r'
    using WELL bsqr-Well-order wo-rel-def by blast

  have D < Field ?r' unfolding Field-bsqr using SUB .
  with OF obtain a1 and a2 where
    (a1,a2) ∈ Field ?r' and 1: D = underS ?r' (a1,a2)
```

```
    using Well' wo-rel.ofilter-underS-Field[of ?r' D] by auto
  hence 2: {a1,a2} ≤ Field r unfolding Field-bsqr by auto
  let ?m = wo-rel.max2 r a1 a2
  have D ≤ (under r ?m) × (under r ?m)
  proof(unfold 1)
    {fix b1 b2
     let ?n = wo-rel.max2 r b1 b2
     assume (b1,b2) ∈ underS ?r' (a1,a2)
     hence 3: ((b1,b2),(a1,a2)) ∈ ?r'
       unfolding underS-def by blast
     hence (?n,?m) ∈ r using WELL by (simp add: bsqr-max2)
     moreover
     {have (b1,b2) ∈ Field ?r' using 3 unfolding Field-def by auto
      hence {b1,b2} ≤ Field r unfolding Field-bsqr by auto
      hence (b1,?n) ∈ r ∧ (b2,?n) ∈ r
        using Well by (simp add: wo-rel.max2-greater)
     }
     ultimately have (b1,?m) ∈ r ∧ (b2,?m) ∈ r
      using Trans trans-def[of r] by blast
     hence (b1,b2) ∈ (under r ?m) × (under r ?m) unfolding under-def by
simp}
    thus underS ?r' (a1,a2) ≤ (under r ?m) × (under r ?m) by auto
  qed
  moreover have wo-rel.ofilter r (under r ?m)
    using Well by (simp add: wo-rel.under-ofilter)
  moreover have under r ?m < Field r
    using NE under-Field[of r ?m] by blast
  ultimately show ?thesis by blast
qed

definition Func where
  Func A B = {f . (∀ a ∈ A. f a ∈ B) ∧ (∀ a. a ∉ A ⟶ f a = undefined)}

lemma Func-empty:
  Func {} B = {λx. undefined}
  unfolding Func-def by auto

lemma Func-elim:
  assumes g ∈ Func A B and a ∈ A
  shows ∃ b. b ∈ B ∧ g a = b
  using assms unfolding Func-def by (cases g a = undefined) auto

definition curr where
  curr A f ≡ λ a. if a ∈ A then λb. f (a,b) else undefined

lemma curr-in:
  assumes f: f ∈ Func (A × B) C
  shows curr A f ∈ Func A (Func B C)
  using assms unfolding curr-def Func-def by auto
```

lemma *curr-inj*:
assumes *f1* ∈ *Func* (*A* × *B*) *C* **and** *f2* ∈ *Func* (*A* × *B*) *C*
shows *curr A f1* = *curr A f2* ⟷ *f1* = *f2*
proof *safe*
assume *c*: *curr A f1* = *curr A f2*
show *f1* = *f2*
proof (*rule ext*, *clarify*)
fix *a b* **show** *f1* (*a*, *b*) = *f2* (*a*, *b*)
proof (*cases* (*a*,*b*) ∈ *A* × *B*)
case *False*
thus *?thesis* **using** *assms* **unfolding** *Func-def* **by** *auto*
next
case *True* **hence** *a*: *a* ∈ *A* **and** *b*: *b* ∈ *B* **by** *auto*
thus *?thesis*
using *c* **unfolding** *curr-def fun-eq-iff* **by**(*elim allE*[*of* - *a*]) *simp*
qed
qed
qed

lemma *curr-surj*:
assumes *g* ∈ *Func A* (*Func B C*)
shows ∃ *f* ∈ *Func* (*A* × *B*) *C*. *curr A f* = *g*
proof
let *?f* = λ *ab*. *if fst ab* ∈ *A* ∧ *snd ab* ∈ *B then g* (*fst ab*) (*snd ab*) *else undefined*
show *curr A ?f* = *g*
proof (*rule ext*)
fix *a* **show** *curr A ?f a* = *g a*
proof (*cases a* ∈ *A*)
case *False*
hence *g a* = *undefined* **using** *assms* **unfolding** *Func-def* **by** *auto*
thus *?thesis* **unfolding** *curr-def* **using** *False* **by** *simp*
next
case *True*
obtain *g1* **where** *g1* ∈ *Func B C* **and** *g a* = *g1*
using *assms* **using** *Func-elim*[*OF assms True*] **by** *blast*
thus *?thesis* **using** *True* **unfolding** *Func-def curr-def* **by** *auto*
qed
qed
show *?f* ∈ *Func* (*A* × *B*) *C* **using** *assms* **unfolding** *Func-def mem-Collect-eq*
by *auto*
qed

lemma *bij-betw-curr*:
bij-betw (*curr A*) (*Func* (*A* × *B*) *C*) (*Func A* (*Func B C*))
unfolding *bij-betw-def inj-on-def*
using *curr-surj curr-in curr-inj* **by** *blast*

definition *Func-map* **where**

Func-map B2 f1 f2 g b2 ≡ *if b2* ∈ *B2 then f1* (*g* (*f2 b2*)) *else undefined*

lemma *Func-map*:
assumes *g*: *g* ∈ *Func A2 A1* **and** *f1*: *f1* ‘ *A1* ⊆ *B1* **and** *f2*: *f2* ‘ *B2* ⊆ *A2*
shows *Func-map B2 f1 f2 g* ∈ *Func B2 B1*
using *assms* **unfolding** *Func-def Func-map-def mem-Collect-eq* **by** *auto*

lemma *Func-non-emp*:
assumes *B* ≠ {}
shows *Func A B* ≠ {}
proof−
obtain *b* **where** *b*: *b* ∈ *B* **using** *assms* **by** *auto*
hence (λ *a*. *if a* ∈ *A then b else undefined*) ∈ *Func A B* **unfolding** *Func-def* **by** *auto*
thus *?thesis* **by** *blast*
qed

lemma *Func-is-emp*:
Func A B = {} ⟷ *A* ≠ {} ∧ *B* = {} (**is** *?L* ⟷ *?R*)
proof
assume *?L*
then show *?R*
using *Func-empty Func-non-emp*[*of B A*] **by** *blast*
next
assume *?R*
then show *?L*
using *Func-empty Func-non-emp*[*of B A*] **by** (*auto simp*: *Func-def*)
qed

lemma *Func-map-surj*:
assumes *B1*: *f1* ‘ *A1* = *B1* **and** *A2*: *inj-on f2 B2 f2* ‘ *B2* ⊆ *A2*
and *B2A2*: *B2* = {} ⟹ *A2* = {}
shows *Func B2 B1* = *Func-map B2 f1 f2* ‘ *Func A2 A1*
proof(*cases B2* = {})
case *True*
thus *?thesis* **using** *B2A2* **by** (*auto simp*: *Func-empty Func-map-def*)
next
case *False* **note** *B2* = *False*
show *?thesis*
proof *safe*
fix *h* **assume** *h*: *h* ∈ *Func B2 B1*
define *j1* **where** *j1* = *inv-into A1 f1*
have ∀ *a2* ∈ *f2* ‘ *B2*. ∃ *b2*. *b2* ∈ *B2* ∧ *f2 b2* = *a2* **by** *blast*
then obtain *k* **where** *k*: ∀ *a2* ∈ *f2* ‘ *B2*. *k a2* ∈ *B2* ∧ *f2* (*k a2*) = *a2*
by *atomize-elim* (*rule bchoice*)
{**fix** *b2* **assume** *b2*: *b2* ∈ *B2*
hence *f2* (*k* (*f2 b2*)) = *f2 b2* **using** *k A2*(*2*) **by** *auto*
moreover have *k* (*f2 b2*) ∈ *B2* **using** *b2 A2*(*2*) *k* **by** *auto*
ultimately have *k* (*f2 b2*) = *b2* **using** *b2 A2*(*1*) **unfolding** *inj-on-def* **by**

```
blast
  } note kk = this
  obtain b22 where b22: b22 ∈ B2 using B2 by auto
  define j2 where [abs-def]: j2 a2 = (if a2 ∈ f2 ‘ B2 then k a2 else b22) for a2
  have j2A2: j2 ‘ A2 ⊆ B2 unfolding j2-def using k b22 by auto
  have j2: ⋀ b2. b2 ∈ B2 ⟹ j2 (f2 b2) = b2
    using kk unfolding j2-def by auto
  define g where g = Func-map A2 j1 j2 h
  have Func-map B2 f1 f2 g = h
  proof (rule ext)
    fix b2 show Func-map B2 f1 f2 g b2 = h b2
    proof(cases b2 ∈ B2)
      case True
      show ?thesis
      proof (cases h b2 = undefined)
        case True
        hence b1: h b2 ∈ f1 ‘ A1 using h ‹b2 ∈ B2› unfolding B1 Func-def by
auto
        show ?thesis using A2 f-inv-into-f[OF b1]
          unfolding True g-def Func-map-def j1-def j2[OF ‹b2 ∈ B2›] by auto
          qed(insert A2 True j2[OF True] h B1, unfold j1-def g-def Func-def
Func-map-def,
            auto intro: f-inv-into-f)
    qed(insert h, unfold Func-def Func-map-def, auto)
  qed
  moreover have g ∈ Func A2 A1 unfolding g-def apply(rule Func-map[OF
h])
    using j2A2 B1 A2 unfolding j1-def by (fast intro: inv-into-into)+
  ultimately show h ∈ Func-map B2 f1 f2 ‘ Func A2 A1
    unfolding Func-map-def[abs-def] by auto
qed(use B1 Func-map[OF - - A2(2)] in auto)
qed

end
```

30 Cardinal-Order Relations as Needed by Bounded Natural Functors

```
theory BNF-Cardinal-Order-Relation
  imports Zorn BNF-Wellorder-Constructions
begin
```

In this section, we define cardinal-order relations to be minim well-orders on their field. Then we define the cardinal of a set to be *some* cardinal-order relation on that set, which will be unique up to order isomorphism. Then we study the connection between cardinals and:

- standard set-theoretic constructions: products, sums, unions, lists, pow-

ersets, set-of finite sets operator;

- finiteness and infiniteness (in particular, with the numeric cardinal operator for finite sets, *card*, from the theory *Finite-Sets.thy*).

On the way, we define the canonical ω cardinal and finite cardinals. We also define (again, up to order isomorphism) the successor of a cardinal, and show that any cardinal admits a successor.

Main results of this section are the existence of cardinal relations and the facts that, in the presence of infiniteness, most of the standard set-theoretic constructions (except for the powerset) *do not increase cardinality.* In particular, e.g., the set of words/lists over any infinite set has the same cardinality (hence, is in bijection) with that set.

30.1 Cardinal orders

A cardinal order in our setting shall be a well-order *minim* w.r.t. the order-embedding relation, $\leq o$ (which is the same as being *minimal* w.r.t. the strict order-embedding relation, $< o$), among all the well-orders on its field.

definition *card-order-on* :: *'a set* ⇒ *'a rel* ⇒ *bool*
  where
    card-order-on A r ≡ *well-order-on A r* ∧ (∀ *r'*. *well-order-on A r'* ⟶ *r* ≤*o r'*)

abbreviation *Card-order r* ≡ *card-order-on* (*Field r*) *r*
abbreviation *card-order r* ≡ *card-order-on UNIV r*

lemma *card-order-on-well-order-on*:
  assumes *card-order-on A r*
  shows *well-order-on A r*
  using *assms* **unfolding** *card-order-on-def* **by** *simp*

lemma *card-order-on-Card-order*:
  card-order-on A r ⟹ *A* = *Field r* ∧ *Card-order r*
  unfolding *card-order-on-def* **using** *well-order-on-Field* **by** *blast*

The existence of a cardinal relation on any given set (which will mean that any set has a cardinal) follows from two facts:

- Zermelo's theorem (proved in *Zorn.thy* as theorem *well-order-on*), which states that on any given set there exists a well-order;
- The well-founded-ness of $< o$, ensuring that then there exists a minimal such well-order, i.e., a cardinal order.

theorem *card-order-on*: ∃ *r*. *card-order-on A r*
proof −
  define *R* **where** *R* ≡ {*r*. *well-order-on A r*}

have $R \neq \{\} \wedge (\forall\, r \in R.\ \textit{Well-order}\ r)$
using *well-order-on*[*of A*] *R-def well-order-on-Well-order* **by** *blast*
with *exists-minim-Well-order*[*of R*] **show** *?thesis*
by (*auto simp*: *R-def card-order-on-def*)
qed

lemma *card-order-on-ordIso*:
assumes *CO*: *card-order-on A r* **and** *CO'*: *card-order-on A r'*
shows $r =o\ r'$
using *assms* **unfolding** *card-order-on-def*
using *ordIso-iff-ordLeq* **by** *blast*

lemma *Card-order-ordIso*:
assumes *CO*: *Card-order r* **and** *ISO*: $r' =o\ r$
shows *Card-order r'*
using *ISO* **unfolding** *ordIso-def*
proof(*unfold card-order-on-def*, *auto*)
fix p' **assume** *well-order-on* (*Field r'*) p'
hence *0*: *Well-order p'* $\wedge$ *Field p'* = *Field r'*
using *well-order-on-Well-order* **by** *blast*
obtain *f* **where** *1*: *iso r' r f* **and** *2*: *Well-order r* $\wedge$ *Well-order r'*
using *ISO* **unfolding** *ordIso-def* **by** *auto*
hence *3*: *inj-on f* (*Field r'*) $\wedge$ *f* ' (*Field r'*) = *Field r*
by (*auto simp add*: *iso-iff embed-inj-on*)
let *?p* = *dir-image p' f*
have *4*: $p' =o$ *?p* $\wedge$ *Well-order ?p*
using *0 2 3* **by** (*auto simp add*: *dir-image-ordIso Well-order-dir-image*)
moreover have *Field ?p* = *Field r*
using *0 3* **by** (*auto simp add*: *dir-image-Field*)
ultimately have *well-order-on* (*Field r*) *?p* **by** *auto*
hence $r \leq o$ *?p* **using** *CO* **unfolding** *card-order-on-def* **by** *auto*
thus $r' \leq o\ p'$
using *ISO 4 ordLeq-ordIso-trans ordIso-ordLeq-trans ordIso-symmetric* **by** *blast*
qed

lemma *Card-order-ordIso2*:
assumes *CO*: *Card-order r* **and** *ISO*: $r =o\ r'$
shows *Card-order r'*
using *assms Card-order-ordIso ordIso-symmetric* **by** *blast*

30.2 Cardinal of a set

We define the cardinal of set to be *some* cardinal order on that set. We shall prove that this notion is unique up to order isomorphism, meaning that order isomorphism shall be the true identity of cardinals.

definition *card-of* :: *'a set* $\Rightarrow$ *'a rel* (‹(‹*open-block notation=‹mixfix card-of›*›|-|)›)
where *card-of A* = (*SOME r*. *card-order-on A r*)

lemma *card-of-card-order-on*: *card-order-on A* |*A*|

unfolding *card-of-def* **by** (*auto simp add*: *card-order-on someI-ex*)

lemma *card-of-well-order-on*: *well-order-on A |A|*
using *card-of-card-order-on card-order-on-def* **by** *blast*

lemma *Field-card-of*: *Field |A| = A*
using *card-of-card-order-on*[*of A*] **unfolding** *card-order-on-def*
using *well-order-on-Field* **by** *blast*

lemma *card-of-Card-order*: *Card-order |A|*
by (*simp only*: *card-of-card-order-on Field-card-of*)

corollary *ordIso-card-of-imp-Card-order*:
r =o |A| ⟹ Card-order r
using *card-of-Card-order Card-order-ordIso* **by** *blast*

lemma *card-of-Well-order*: *Well-order |A|*
using *card-of-Card-order* **unfolding** *card-order-on-def* **by** *auto*

lemma *card-of-refl*: *|A| =o |A|*
using *card-of-Well-order ordIso-reflexive* **by** *blast*

lemma *card-of-least*: *well-order-on A r ⟹ |A| ≤o r*
using *card-of-card-order-on* **unfolding** *card-order-on-def* **by** *blast*

lemma *card-of-ordIso*:
(∃f. bij-betw f A B) = (|A| =o |B|)
proof(*auto*)
fix *f* **assume** ∗: *bij-betw f A B*
then obtain *r* **where** *well-order-on B r ∧ |A| =o r*
using *Well-order-iso-copy card-of-well-order-on* **by** *blast*
hence *|B| ≤o |A|* **using** *card-of-least*
ordLeq-ordIso-trans ordIso-symmetric **by** *blast*
moreover
{let *?g = inv-into A f*
have *bij-betw ?g B A* **using** ∗ *bij-betw-inv-into* **by** *blast*
then obtain *r* **where** *well-order-on A r ∧ |B| =o r*
using *Well-order-iso-copy card-of-well-order-on* **by** *blast*
hence *|A| ≤o |B|*
using *card-of-least ordLeq-ordIso-trans ordIso-symmetric* **by** *blast*
}
ultimately show *|A| =o |B|* **using** *ordIso-iff-ordLeq* **by** *blast*
next
assume *|A| =o |B|*
then obtain *f* **where** *iso (|A|) (|B|) f*
unfolding *ordIso-def* **by** *auto*
hence *bij-betw f A B* **unfolding** *iso-def Field-card-of* **by** *simp*
thus *∃f. bij-betw f A B* **by** *auto*
qed

lemma *card-of-ordLeq*:
(∃*f*. *inj-on f A* ∧ *f* ‘ *A* ≤ *B*) = (|*A*| ≤*o* |*B*|)
proof(*auto*)
fix *f* **assume** ∗: *inj-on f A* **and** ∗∗: *f* ‘ *A* ≤ *B*
{assume |*B*| <*o* |*A*|
hence |*B*| ≤*o* |*A*| **using** *ordLeq-iff-ordLess-or-ordIso* **by** *blast*
then obtain *g* **where** *embed* (|*B*|) (|*A*|) *g*
unfolding *ordLeq-def* **by** *auto*
hence *1*: *inj-on g B* ∧ *g* ‘ *B* ≤ *A* **using** *embed-inj-on*[*of* |*B*| |*A*| *g*]
card-of-Well-order[*of B*] *Field-card-of*[*of B*] *Field-card-of*[*of A*]
embed-Field[*of* |*B*| |*A*| *g*] **by** *auto*
obtain *h* **where** *bij-betw h A B*
using ∗ ∗∗ *1 Schroeder-Bernstein*[*of f*] **by** *fastforce*
hence |*A*| ≤*o* |*B*| **using** *card-of-ordIso ordIso-iff-ordLeq* **by** *auto*
}
thus |*A*| ≤*o* |*B*| **using** *ordLess-or-ordLeq*[*of* |*B*| |*A*|]
by (*auto simp*: *card-of-Well-order*)
next
assume ∗: |*A*| ≤*o* |*B*|
obtain *f* **where** *embed* |*A*| |*B*| *f*
using ∗ **unfolding** *ordLeq-def* **by** *auto*
hence *inj-on f A* ∧ *f* ‘ *A* ≤ *B*
using *embed-inj-on*[*of* |*A*| |*B*|] *card-of-Well-order embed-Field*[*of* |*A*| |*B*|]
by (*auto simp*: *Field-card-of*)
thus ∃*f*. *inj-on f A* ∧ *f* ‘ *A* ≤ *B* **by** *auto*
qed

lemma *card-of-ordLeq2*:
A ≠ {} ⟹ (∃*g*. *g* ‘ *B* = *A*) = (|*A*| ≤*o* |*B*|)
using *card-of-ordLeq*[*of A B*] *inj-on-iff-surj*[*of A B*] **by** *auto*

lemma *card-of-ordLess*:
(¬(∃*f*. *inj-on f A* ∧ *f* ‘ *A* ≤ *B*)) = (|*B*| <*o* |*A*|)
proof −
have (¬(∃*f*. *inj-on f A* ∧ *f* ‘ *A* ≤ *B*)) = (¬ |*A*| ≤*o* |*B*|)
using *card-of-ordLeq* **by** *blast*
also have . . . = (|*B*| <*o* |*A*|)
using *not-ordLeq-iff-ordLess* **by** (*auto intro*: *card-of-Well-order*)
finally show *?thesis* **.**
qed

lemma *card-of-ordLess2*:
B ≠ {} ⟹ (¬(∃*f*. *f* ‘ *A* = *B*)) = (|*A*| <*o* |*B*|)
using *card-of-ordLess*[*of B A*] *inj-on-iff-surj*[*of B A*] **by** *auto*

lemma *card-of-ordIsoI*:
assumes *bij-betw f A B*
shows |*A*| =*o* |*B*|

using *assms* **unfolding** *card-of-ordIso*[*symmetric*] **by** *auto*

lemma *card-of-ordLeqI*:
assumes *inj-on f A* **and** $\bigwedge$ *a*. $a \in A \Longrightarrow f\ a \in B$
shows $|A| \leq o\ |B|$
using *assms* **unfolding** *card-of-ordLeq*[*symmetric*] **by** *auto*

lemma *card-of-unique*:
card-order-on A r $\Longrightarrow$ $r =o\ |A|$
by (*simp only*: *card-order-on-ordIso card-of-card-order-on*)

lemma *card-of-mono1*:
$A \leq B \Longrightarrow |A| \leq o\ |B|$
using *inj-on-id*[*of A*] *card-of-ordLeq*[*of A B*] **by** *fastforce*

lemma *card-of-mono2*:
assumes $r \leq o\ r'$
shows $|Field\ r| \leq o\ |Field\ r'|$
proof −
obtain *f* **where**
1: *well-order-on* (*Field r*) *r* ∧ *well-order-on* (*Field r*) *r* ∧ *embed r r' f*
using *assms* **unfolding** *ordLeq-def*
by (*auto simp add*: *well-order-on-Well-order*)
hence *inj-on f* (*Field r*) ∧ *f* ' (*Field r*) ≤ *Field r'*
by (*auto simp add*: *embed-inj-on embed-Field*)
thus $|Field\ r| \leq o\ |Field\ r'|$ **using** *card-of-ordLeq* **by** *blast*
qed

lemma *card-of-cong*: $r =o\ r' \Longrightarrow |Field\ r| =o\ |Field\ r'|$
by (*simp add*: *ordIso-iff-ordLeq card-of-mono2*)

lemma *card-of-Field-ordIso*:
assumes *Card-order r*
shows $|Field\ r| =o\ r$
proof −
have *card-order-on* (*Field r*) *r*
using *assms card-order-on-Card-order* **by** *blast*
moreover have *card-order-on* (*Field r*) $|Field\ r|$
using *card-of-card-order-on* **by** *blast*
ultimately show *?thesis* **using** *card-order-on-ordIso* **by** *blast*
qed

lemma *Card-order-iff-ordIso-card-of*:
Card-order r = (*r* =*o* $|Field\ r|$)
using *ordIso-card-of-imp-Card-order card-of-Field-ordIso ordIso-symmetric* **by** *blast*

lemma *Card-order-iff-ordLeq-card-of*:
Card-order r = (*r* ≤*o* $|Field\ r|$)

proof −
have *Card-order r* = (*r* =*o* |*Field r*|)
unfolding *Card-order-iff-ordIso-card-of* **by** *simp*
also have ... = (*r* ≤*o* |*Field r*| ∧ |*Field r*| ≤*o r*)
unfolding *ordIso-iff-ordLeq* **by** *simp*
also have ... = (*r* ≤*o* |*Field r*|)
using *card-of-least*
by (*auto simp*: *card-of-least ordLeq-Well-order-simp*)
finally show *?thesis* **.**
qed

lemma *Card-order-iff-Restr-underS*:
assumes *Well-order r*
shows *Card-order r* = (∀ *a* ∈ *Field r*. *Restr r* (*underS r a*) <*o* |*Field r*|)
using *assms ordLeq-iff-ordLess-Restr card-of-Well-order*
unfolding *Card-order-iff-ordLeq-card-of* **by** *blast*

lemma *card-of-underS*:
assumes *r*: *Card-order r* **and** *a*: *a* ∈ *Field r*
shows |*underS r a*| <*o r*
proof −
let *?A* = *underS r a* **let** *?r'* = *Restr r ?A*
have *1*: *Well-order r*
using *r* **unfolding** *card-order-on-def* **by** *simp*
have *Well-order ?r'* **using** *1 Well-order-Restr* **by** *auto*
with *card-of-card-order-on* **have** |*Field ?r'*| ≤*o ?r'*
unfolding *card-order-on-def* **by** *auto*
moreover have *Field ?r'* = *?A*
using *1 wo-rel.underS-ofilter Field-Restr-ofilter*
unfolding *wo-rel-def* **by** *fastforce*
ultimately have |*?A*| ≤*o ?r'* **by** *simp*
also have *?r'* <*o* |*Field r*|
using *1 a r Card-order-iff-Restr-underS* **by** *blast*
also have |*Field r*| =*o r*
using *r ordIso-symmetric* **unfolding** *Card-order-iff-ordIso-card-of* **by** *auto*
finally show *?thesis* **.**
qed

lemma *ordLess-Field*:
assumes *r* <*o r'*
shows |*Field r*| <*o r'*
proof −
have *well-order-on* (*Field r*) *r* **using** *assms* **unfolding** *ordLess-def*
by (*auto simp add*: *well-order-on-Well-order*)
hence |*Field r*| ≤*o r* **using** *card-of-least* **by** *blast*
thus *?thesis* **using** *assms ordLeq-ordLess-trans* **by** *blast*
qed

lemma *internalize-card-of-ordLeq*:

$(\ |A| \leq o\ r) = (\exists\, B \leq \mathit{Field}\ r.\ |A| =o\ |B| \wedge |B| \leq o\ r)$
proof
assume $|A| \leq o\ r$
then obtain p **where** *1*: $\mathit{Field}\ p \leq \mathit{Field}\ r \wedge |A| =o\ p \wedge p \leq o\ r$
using *internalize-ordLeq*[*of* $|A|$ r] **by** *blast*
hence *Card-order p* **using** *card-of-Card-order Card-order-ordIso2* **by** *blast*
hence $|\mathit{Field}\ p| =o\ p$ **using** *card-of-Field-ordIso* **by** *blast*
hence $|A| =o\ |\mathit{Field}\ p| \wedge |\mathit{Field}\ p| \leq o\ r$
using *1 ordIso-equivalence ordIso-ordLeq-trans* **by** *blast*
thus $\exists\, B \leq \mathit{Field}\ r.\ |A| =o\ |B| \wedge |B| \leq o\ r$ **using** *1* **by** *blast*
next
assume $\exists\, B \leq \mathit{Field}\ r.\ |A| =o\ |B| \wedge |B| \leq o\ r$
thus $|A| \leq o\ r$ **using** *ordIso-ordLeq-trans* **by** *blast*
qed

lemma *internalize-card-of-ordLeq2*:
$(\ |A| \leq o\ |C|\ ) = (\exists\, B \leq C.\ |A| =o\ |B| \wedge |B| \leq o\ |C|\ )$
using *internalize-card-of-ordLeq*[*of* A $|C|$] *Field-card-of*[*of* C] **by** *auto*

30.3 Cardinals versus set operations on arbitrary sets

Here we embark in a long journey of simple results showing that the standard set-theoretic operations are well-behaved w.r.t. the notion of cardinal – essentially, this means that they preserve the "cardinal identity" $=o$ and are monotonic w.r.t. $\leq o$.

lemma *card-of-empty*: $|\{\}| \leq o\ |A|$
using *card-of-ordLeq inj-on-id* **by** *blast*

lemma *card-of-empty1*:
assumes *Well-order r* $\vee$ *Card-order r*
shows $|\{\}| \leq o\ r$
proof −
have *Well-order r* **using** *assms* **unfolding** *card-order-on-def* **by** *auto*
hence $|\mathit{Field}\ r| \leq o\ r$
using *assms card-of-least* **by** *blast*
moreover have $|\{\}| \leq o\ |\mathit{Field}\ r|$ **by** (*simp add*: *card-of-empty*)
ultimately show *?thesis* **using** *ordLeq-transitive* **by** *blast*
qed

corollary *Card-order-empty*:
Card-order r $\Longrightarrow |\{\}| \leq o\ r$ **by** (*simp add*: *card-of-empty1*)

lemma *card-of-empty2*:
assumes $|A| =o\ |\{\}|$
shows $A = \{\}$
using *assms card-of-ordIso*[*of A*] *bij-betw-empty2* **by** *blast*

lemma *card-of-empty3*:

assumes $|A| \leq o\ |\{\}|$
shows $A = \{\}$
using *assms*
by (*simp add*: *ordIso-iff-ordLeq card-of-empty1 card-of-empty2*
ordLeq-Well-order-simp)

lemma *card-of-empty-ordIso*:
$|\{\}::'a\ set| =o\ |\{\}::'b\ set|$
using *card-of-ordIso* **unfolding** *bij-betw-def inj-on-def* **by** *blast*

lemma *card-of-image*:
$|f\ `\ A| \leq o\ |A|$
proof(*cases* $A = \{\}$)
case *False*
hence $f\ `\ A \neq \{\}$ **by** *auto*
thus *?thesis*
using *card-of-ordLeq2*[*of f ` A A*] **by** *auto*
qed (*simp add*: *card-of-empty*)

lemma *surj-imp-ordLeq*:
assumes $B \subseteq f\ `\ A$
shows $|B| \leq o\ |A|$
proof −
have $|B| \leq o\ |f\ `\ A|$ **using** *assms card-of-mono1* **by** *auto*
thus *?thesis* **using** *card-of-image ordLeq-transitive* **by** *blast*
qed

lemma *card-of-singl-ordLeq*:
assumes $A \neq \{\}$
shows $|\{b\}| \leq o\ |A|$
proof −
obtain *a* **where** $*$: $a \in A$ **using** *assms* **by** *auto*
let *?h* $= \lambda\ b'::'b$. *if* $b' = b$ *then a else undefined*
have *inj-on ?h* $\{b\} \wedge$ *?h* $`\ \{b\} \leq A$
using $*$ **unfolding** *inj-on-def* **by** *auto*
thus *?thesis* **unfolding** *card-of-ordLeq*[*symmetric*] **by** (*intro exI*)
qed

corollary *Card-order-singl-ordLeq*:
$[\![$*Card-order r*; *Field r* $\neq \{\}]\!] \Longrightarrow |\{b\}| \leq o\ r$
using *card-of-singl-ordLeq*[*of Field r b*]
card-of-Field-ordIso[*of r*] *ordLeq-ordIso-trans* **by** *blast*

lemma *card-of-Pow*: $|A| <o\ |Pow\ A|$
using *card-of-ordLess2*[*of Pow A A*] *Cantors-theorem*[*of A*]
Pow-not-empty[*of A*] **by** *auto*

corollary *Card-order-Pow*:
Card-order r $\Longrightarrow r <o\ |Pow(Field\ r)|$

using *card-of-Pow card-of-Field-ordIso ordIso-ordLess-trans ordIso-symmetric* **by** *blast*

lemma *card-of-Plus1*: $|A| \leq o\ |A <+> B|$ **and** *card-of-Plus2*: $|B| \leq o\ |A <+> B|$
using *card-of-ordLeq* **by** *force+*

corollary *Card-order-Plus1*:
$Card\text{-}order\ r \Longrightarrow r \leq o\ |(Field\ r) <+> B|$
using *card-of-Plus1 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric* **by** *blast*

corollary *Card-order-Plus2*:
$Card\text{-}order\ r \Longrightarrow r \leq o\ |A <+> (Field\ r)|$
using *card-of-Plus2 card-of-Field-ordIso ordIso-ordLeq-trans ordIso-symmetric* **by** *blast*

lemma *card-of-Plus-empty1*: $|A| =o\ |A <+> \{\}|$
proof −
have *bij-betw Inl A* $(A <+> \{\})$ **unfolding** *bij-betw-def inj-on-def* **by** *auto*
thus *?thesis* **using** *card-of-ordIso* **by** *auto*
qed

lemma *card-of-Plus-empty2*: $|A| =o\ |\{\} <+> A|$
proof −
have *bij-betw Inr A* $(\{\} <+> A)$ **unfolding** *bij-betw-def inj-on-def* **by** *auto*
thus *?thesis* **using** *card-of-ordIso* **by** *auto*
qed

lemma *card-of-Plus-commute*: $|A <+> B| =o\ |B <+> A|$
proof −
let *?f* $= \lambda c.$ *case c of Inl a* $\Rightarrow$ *Inr a* | *Inr b* $\Rightarrow$ *Inl b*
have *bij-betw ?f* $(A <+> B)\ (B <+> A)$
unfolding *bij-betw-def inj-on-def* **by** *force*
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

lemma *card-of-Plus-assoc*:
fixes $A :: 'a\ set$ **and** $B :: 'b\ set$ **and** $C :: 'c\ set$
shows $|(A <+> B) <+> C| =o\ |A <+> B <+> C|$
proof −
define $f :: ('a + 'b) + 'c \Rightarrow 'a + 'b + 'c$
where [*abs-def*]: $f\ k =$
(*case k of*
Inl ab $\Rightarrow$
(*case ab of*
Inl a $\Rightarrow$ *Inl a*
| *Inr b* $\Rightarrow$ *Inr* (*Inl b*))
| *Inr c* $\Rightarrow$ *Inr* (*Inr c*))
for *k*

```
  have A <+> B <+> C ⊆ f ‘ ((A <+> B) <+> C)
  proof
    fix x assume x: x ∈ A <+> B <+> C
    show x ∈ f ‘ ((A <+> B) <+> C)
    proof(cases x)
      case (Inl a)
      hence a ∈ A x = f (Inl (Inl a))
        using x unfolding f-def by auto
      thus ?thesis by auto
    next
      case (Inr bc) with x show ?thesis
        by (cases bc) (force simp: f-def)+
    qed
  qed
  hence bij-betw f ((A <+> B) <+> C) (A <+> B <+> C)
    unfolding bij-betw-def inj-on-def f-def by fastforce
  thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Plus-mono1:
  assumes |A| ≤o |B|
  shows |A <+> C| ≤o |B <+> C|
proof −
  obtain f where f: inj-on f A ∧ f ‘ A ≤ B
    using assms card-of-ordLeq[of A] by fastforce
  define g where g ≡ λd. case d of Inl a ⇒ Inl(f a) | Inr (c::'c) ⇒ Inr c
  have inj-on g (A <+> C) ∧ g ‘ (A <+> C) ≤ (B <+> C)
    using f unfolding inj-on-def g-def by force
  thus ?thesis using card-of-ordLeq by blast
qed

corollary ordLeq-Plus-mono1:
  assumes r ≤o r′
  shows |(Field r) <+> C| ≤o |(Field r′) <+> C|
  using assms card-of-mono2 card-of-Plus-mono1 by blast

lemma card-of-Plus-mono2:
  assumes |A| ≤o |B|
  shows |C <+> A| ≤o |C <+> B|
  using card-of-Plus-mono1[OF assms]
  by (blast intro: card-of-Plus-commute ordIso-ordLeq-trans ordLeq-ordIso-trans)

corollary ordLeq-Plus-mono2:
  assumes r ≤o r′
  shows |A <+> (Field r)| ≤o |A <+> (Field r′)|
  using assms card-of-mono2 card-of-Plus-mono2 by blast

lemma card-of-Plus-mono:
  assumes |A| ≤o |B| and |C| ≤o |D|
```

shows $|A <+> C| \leq o\ |B <+> D|$
using *assms card-of-Plus-mono1*[*of A B C*] *card-of-Plus-mono2*[*of C D B*]
ordLeq-transitive **by** *blast*

corollary *ordLeq-Plus-mono*:
assumes $r \leq o\ r'$ **and** $p \leq o\ p'$
shows $|(Field\ r) <+> (Field\ p)| \leq o\ |(Field\ r') <+> (Field\ p')|$
using *assms card-of-mono2*[*of r r'*] *card-of-mono2*[*of p p'*] *card-of-Plus-mono* **by** *blast*

lemma *card-of-Plus-cong1*:
assumes $|A| =o\ |B|$
shows $|A <+> C| =o\ |B <+> C|$
using *assms* **by** (*simp add*: *ordIso-iff-ordLeq card-of-Plus-mono1*)

corollary *ordIso-Plus-cong1*:
assumes $r =o\ r'$
shows $|(Field\ r) <+> C| =o\ |(Field\ r') <+> C|$
using *assms card-of-cong card-of-Plus-cong1* **by** *blast*

lemma *card-of-Plus-cong2*:
assumes $|A| =o\ |B|$
shows $|C <+> A| =o\ |C <+> B|$
using *assms* **by** (*simp add*: *ordIso-iff-ordLeq card-of-Plus-mono2*)

corollary *ordIso-Plus-cong2*:
assumes $r =o\ r'$
shows $|A <+> (Field\ r)| =o\ |A <+> (Field\ r')|$
using *assms card-of-cong card-of-Plus-cong2* **by** *blast*

lemma *card-of-Plus-cong*:
assumes $|A| =o\ |B|$ **and** $|C| =o\ |D|$
shows $|A <+> C| =o\ |B <+> D|$
using *assms* **by** (*simp add*: *ordIso-iff-ordLeq card-of-Plus-mono*)

corollary *ordIso-Plus-cong*:
assumes $r =o\ r'$ **and** $p =o\ p'$
shows $|(Field\ r) <+> (Field\ p)| =o\ |(Field\ r') <+> (Field\ p')|$
using *assms card-of-cong*[*of r r'*] *card-of-cong*[*of p p'*] *card-of-Plus-cong* **by** *blast*

lemma *card-of-Un-Plus-ordLeq*:
$|A \cup B| \leq o\ |A <+> B|$
proof −
let *?f* $= \lambda\ c.$ *if* $c \in A$ *then Inl c else Inr c*
have *inj-on ?f* $(A \cup B) \wedge$ *?f* ‘ $(A \cup B) \leq A <+> B$
unfolding *inj-on-def* **by** *auto*
thus *?thesis* **using** *card-of-ordLeq* **by** *blast*
qed

lemma *card-of-Times1*:
assumes $A \neq \{\}$
shows $|B| \leq o\ |B \times A|$
proof(*cases* $B = \{\}$)
case *False*
have *fst* ‘$(B \times A) = B$ **using** *assms* **by** *auto*
thus *?thesis* **using** *inj-on-iff-surj*[*of B* $B \times A$]
card-of-ordLeq False **by** *blast*
qed (*simp add*: *card-of-empty*)

lemma *card-of-Times-commute*: $|A \times B| =o\ |B \times A|$
proof −
have *bij-betw* $(\lambda(a,b).\ (b,a))\ (A \times B)\ (B \times A)$
unfolding *bij-betw-def inj-on-def* **by** *auto*
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

lemma *card-of-Times2*:
assumes $A \neq \{\}$ **shows** $|B| \leq o\ |A \times B|$
using *assms card-of-Times1*[*of A B*] *card-of-Times-commute*[*of B A*]
ordLeq-ordIso-trans **by** *blast*

corollary *Card-order-Times1*:
$[\![$*Card-order r*; $B \neq \{\}]\!] \Longrightarrow r \leq o\ |(Field\ r) \times B|$
using *card-of-Times1*[*of B*] *card-of-Field-ordIso*
ordIso-ordLeq-trans ordIso-symmetric **by** *blast*

corollary *Card-order-Times2*:
$[\![$*Card-order r*; $A \neq \{\}]\!] \Longrightarrow r \leq o\ |A \times (Field\ r)|$
using *card-of-Times2*[*of A*] *card-of-Field-ordIso*
ordIso-ordLeq-trans ordIso-symmetric **by** *blast*

lemma *card-of-Times3*: $|A| \leq o\ |A \times A|$
using *card-of-Times1*[*of A*]
by(*cases* $A = \{\}$, *simp add*: *card-of-empty*)

lemma *card-of-Plus-Times-bool*: $|A <+> A| =o\ |A \times (UNIV::bool\ set)|$
proof −
let *?f* $= \lambda c::'a + 'a.$ *case c of Inl a* $\Rightarrow (a, True)$
$|$*Inr a* $\Rightarrow (a, False)$
have *bij-betw ?f* $(A <+> A)\ (A \times (UNIV::bool\ set))$
proof −
have $\bigwedge$*c1 c2*. *?f c1* $=$ *?f c2* $\Longrightarrow$ *c1* $=$ *c2*
by (*force split*: *sum.split-asm*)
moreover
have $\bigwedge c.\ c \in A <+> A \Longrightarrow$ *?f* $c \in A \times (UNIV::bool\ set)$
by (*force split*: *sum.split-asm*)
moreover
{fix *a bl* **assume** $(a,bl) \in A \times (UNIV::bool\ set)$

```
    hence (a,bl) ∈ ?f ‘ ( A <+> A)
    by (cases bl) (force split: sum.split-asm)+
   }
   ultimately show ?thesis unfolding bij-betw-def inj-on-def by auto
 qed
 thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-Times-mono1:
 assumes |A| ≤o |B|
 shows |A × C| ≤o |B × C|
proof −
 obtain f where f: inj-on f A ∧ f ‘ A ≤ B
  using assms card-of-ordLeq[of A] by fastforce
 define g where g ≡ (λ(a,c::'c). (f a,c))
 have inj-on g (A × C) ∧ g ‘ (A × C) ≤ (B × C)
  using f unfolding inj-on-def using g-def by auto
 thus ?thesis using card-of-ordLeq by blast
qed

corollary ordLeq-Times-mono1:
 assumes r ≤o r'
 shows |(Field r) × C| ≤o |(Field r') × C|
 using assms card-of-mono2 card-of-Times-mono1 by blast

lemma card-of-Times-mono2:
 assumes |A| ≤o |B|
 shows |C × A| ≤o |C × B|
 using assms card-of-Times-mono1[of A B C]
 by (blast intro: card-of-Times-commute ordIso-ordLeq-trans ordLeq-ordIso-trans)

corollary ordLeq-Times-mono2:
 assumes r ≤o r'
 shows |A × (Field r)| ≤o |A × (Field r')|
 using assms card-of-mono2 card-of-Times-mono2 by blast

lemma card-of-Sigma-mono1:
 assumes ∀ i ∈ I. |A i| ≤o |B i|
 shows |SIGMA i : I. A i| ≤o |SIGMA i : I. B i|
proof −
 have ∀ i. i ∈ I ⟶ (∃ f. inj-on f (A i) ∧ f ‘ (A i) ≤ B i)
  using assms by (auto simp add: card-of-ordLeq)
 with choice[of λ i f. i ∈ I ⟶ inj-on f (A i) ∧ f ‘ (A i) ≤ B i]
 obtain F where F: ∀ i ∈ I. inj-on (F i) (A i) ∧ (F i) ‘ (A i) ≤ B i
  by atomize-elim (auto intro: bchoice)
 define g where g ≡ (λ(i,a::'b). (i,F i a))
 have inj-on g (Sigma I A) ∧ g ‘ (Sigma I A) ≤ (Sigma I B)
  using F unfolding inj-on-def using g-def by force
 thus ?thesis using card-of-ordLeq by blast
```

qed

lemma *card-of-UNION-Sigma*:
$|\bigcup i \in I.\ A\ i| \leq o\ |SIGMA\ i : I.\ A\ i|$
using *Ex-inj-on-UNION-Sigma* [*of A I*] *card-of-ordLeq* **by** *blast*

lemma *card-of-bool*:
assumes *a1* $\neq$ *a2*
shows $|UNIV{::}bool\ set| =o\ |\{a1,a2\}|$
proof −
let *?f* = λ *bl. if bl then a1 else a2*
have *bij-betw ?f UNIV* $\{a1,a2\}$
proof −
have $\bigwedge bl1\ bl2.\ ?f\ bl1 = ?f\ bl2 \Longrightarrow bl1 = bl2$
using *assms* **by** (*force split*: *if-split-asm*)
moreover
have $\bigwedge bl.\ ?f\ bl \in \{a1,a2\}$
using *assms* **by** (*force split*: *if-split-asm*)
ultimately show *?thesis* **unfolding** *bij-betw-def inj-on-def* **by** *force*
qed
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

lemma *card-of-Plus-Times-aux*:
assumes *A2*: *a1* $\neq$ *a2* $\wedge$ $\{a1,a2\} \leq A$ **and**
LEQ: $|A| \leq o\ |B|$
shows $|A <+> B| \leq o\ |A \times B|$
proof −
have *1*: $|UNIV{::}bool\ set| \leq o\ |A|$
using *A2 card-of-mono1*[*of* $\{a1,a2\}$] *card-of-bool*[*of a1 a2*]
by (*blast intro*: *ordIso-ordLeq-trans*)
have $|A <+> B| \leq o\ |B <+> B|$
using *LEQ card-of-Plus-mono1* **by** *blast*
moreover have $|B <+> B| =o\ |B \times (UNIV{::}bool\ set)|$
using *card-of-Plus-Times-bool* **by** *blast*
moreover have $|B \times (UNIV{::}bool\ set)| \leq o\ |B \times A|$
using *1* **by** (*simp add*: *card-of-Times-mono2*)
moreover have $|B \times A| =o\ |A \times B|$
using *card-of-Times-commute* **by** *blast*
ultimately show $|A <+> B| \leq o\ |A \times B|$
by (*blast intro*: *ordLeq-transitive dest*: *ordLeq-ordIso-trans*)
qed

lemma *card-of-Plus-Times*:
assumes *A2*: *a1* $\neq$ *a2* $\wedge$ $\{a1,a2\} \leq A$ **and** *B2*: *b1* $\neq$ *b2* $\wedge$ $\{b1,b2\} \leq B$
shows $|A <+> B| \leq o\ |A \times B|$
proof −
{assume $|A| \leq o\ |B|$
hence *?thesis* **using** *assms* **by** (*auto simp add*: *card-of-Plus-Times-aux*)

```
  }
  moreover
  {assume |B| ≤o |A|
    hence |B <+> A| ≤o |B × A|
      using assms by (auto simp add: card-of-Plus-Times-aux)
    hence ?thesis
      using card-of-Plus-commute card-of-Times-commute
        ordIso-ordLeq-trans ordLeq-ordIso-trans by blast
  }
  ultimately show ?thesis
    using card-of-Well-order[of A] card-of-Well-order[of B]
      ordLeq-total[of |A|] by blast
qed

lemma card-of-Times-Plus-distrib:
  |A × (B <+> C)| =o |A × B <+> A × C| (is |?RHS| =o |?LHS|)
proof −
  let ?f = λ(a, bc). case bc of Inl b ⇒ Inl (a, b) | Inr c ⇒ Inr (a, c)
  have bij-betw ?f ?RHS ?LHS unfolding bij-betw-def inj-on-def by force
  thus ?thesis using card-of-ordIso by blast
qed

lemma card-of-ordLeq-finite:
  assumes |A| ≤o |B| and finite B
  shows finite A
  using assms unfolding ordLeq-def
  using embed-inj-on[of |A| |B|]  embed-Field[of |A| |B|]
    Field-card-of[of A] Field-card-of[of B] inj-on-finite[of - A B] by fastforce

lemma card-of-ordLeq-infinite:
  assumes |A| ≤o |B| and ¬ finite A
  shows ¬ finite B
  using assms card-of-ordLeq-finite by auto

lemma card-of-ordIso-finite:
  assumes |A| =o |B|
  shows finite A = finite B
  using assms unfolding ordIso-def iso-def[abs-def]
  by (auto simp: bij-betw-finite Field-card-of)

lemma card-of-ordIso-finite-Field:
  assumes Card-order r and r =o |A|
  shows finite(Field r) = finite A
  using assms card-of-Field-ordIso card-of-ordIso-finite ordIso-equivalence by blast
```

30.4 Cardinals versus set operations involving infinite sets

Here we show that, for infinite sets, most set-theoretic constructions do not increase the cardinality. The cornerstone for this is theorem *Card-order-Times-same-infinite*,

which states that self-product does not increase cardinality – the proof of this fact adapts a standard set-theoretic argument, as presented, e.g., in the proof of theorem 1.5.11 at page 47 in [4]. Then everything else follows fairly easily.

lemma *infinite-iff-card-of-nat*:
 ¬ *finite A* ⟷ (|*UNIV*::*nat set*| ≤*o* |*A*|)
 unfolding *infinite-iff-countable-subset card-of-ordLeq* **..**

The next two results correspond to the ZF fact that all infinite cardinals are limit ordinals:

lemma *Card-order-infinite-not-under*:
 assumes *CARD*: *Card-order r* **and** *INF*: ¬*finite* (*Field r*)
 shows ¬ (∃ *a*. *Field r* = *under r a*)
proof(*auto*)
 have *0*: *Well-order r* ∧ *wo-rel r* ∧ *Refl r*
  using *CARD* **unfolding** *wo-rel-def card-order-on-def order-on-defs* **by** *auto*
 fix *a* **assume** ∗: *Field r* = *under r a*
 show *False*
 proof(*cases a* ∈ *Field r*)
  assume *Case1*: *a* ∉ *Field r*
  hence *under r a* = {} **unfolding** *Field-def under-def* **by** *auto*
  thus *False* **using** *INF* ∗ **by** *auto*
 next
  let *?r′* = *Restr r* (*underS r a*)
  assume *Case2*: *a* ∈ *Field r*
  hence *1*: *under r a* = *underS r a* ∪ {*a*} ∧ *a* ∉ *underS r a*
   using *0 Refl-under-underS*[*of r a*] *underS-notIn*[*of a r*] **by** *blast*
  have *2*: *wo-rel.ofilter r* (*underS r a*) ∧ *underS r a* < *Field r*
   using *0 wo-rel.underS-ofilter* ∗ *1 Case2* **by** *fast*
  hence *?r′* <*o r* **using** *0* **using** *ofilter-ordLess* **by** *blast*
  moreover
  have *Field ?r′* = *underS r a* ∧ *Well-order ?r′*
   using *2 0 Field-Restr-ofilter*[*of r*] *Well-order-Restr*[*of r*] **by** *blast*
  ultimately have |*underS r a*| <*o r* **using** *ordLess-Field*[*of ?r′*] **by** *auto*
  moreover have |*under r a*| =*o r* **using** ∗ *CARD card-of-Field-ordIso*[*of r*] **by** *auto*
  ultimately have |*underS r a*| <*o* |*under r a*|
   using *ordIso-symmetric ordLess-ordIso-trans* **by** *blast*
  moreover
  {have ∃*f*. *bij-betw f* (*under r a*) (*underS r a*)
     using *infinite-imp-bij-betw*[*of Field r a*] *INF* ∗ *1* **by** *auto*
   hence |*under r a*| =*o* |*underS r a*| **using** *card-of-ordIso* **by** *blast*
  }
  ultimately show *False* **using** *not-ordLess-ordIso ordIso-symmetric* **by** *blast*
 qed
qed

lemma *infinite-Card-order-limit*:

assumes *r*: *Card-order r* **and** ¬*finite* (*Field r*)
and *a*: *a* ∈ *Field r*
shows ∃ *b* ∈ *Field r*. *a* ≠ *b* ∧ (*a*,*b*) ∈ *r*
proof −
have *Field r* ≠ *under r a*
using *assms Card-order-infinite-not-under* **by** *blast*
moreover have *under r a* ≤ *Field r*
using *under-Field* **.**
ultimately obtain *b* **where** *b*: *b* ∈ *Field r* ∧ ¬ (*b*,*a*) ∈ *r*
unfolding *under-def* **by** *blast*
moreover have *ba*: *b* ≠ *a*
using *b r* **unfolding** *card-order-on-def well-order-on-def*
linear-order-on-def partial-order-on-def preorder-on-def refl-on-def **by** *auto*
ultimately have (*a*,*b*) ∈ *r*
using *a r* **unfolding** *card-order-on-def well-order-on-def linear-order-on-def*
total-on-def **by** *blast*
thus *?thesis* **using** *b ba* **by** *auto*
qed

theorem *Card-order-Times-same-infinite*:
assumes *CO*: *Card-order r* **and** *INF*: ¬*finite*(*Field r*)
shows |*Field r* × *Field r*| ≤*o r*
proof −
define *phi* **where**
phi ≡ λ*r*::′*a rel*. *Card-order r* ∧ ¬*finite*(*Field r*) ∧ ¬ |*Field r* × *Field r*| ≤*o r*
have *temp1*: ∀ *r*. *phi r* ⟶ *Well-order r*
unfolding *phi-def card-order-on-def* **by** *auto*
have *Ft*: ¬(∃ *r*. *phi r*)
proof
assume ∃ *r*. *phi r*
hence {*r*. *phi r*} ≠ {} ∧ {*r*. *phi r*} ≤ {*r*. *Well-order r*}
using *temp1* **by** *auto*
then obtain *r* **where** *1*: *phi r* **and** *2*: ∀ *r*′. *phi r*′ ⟶ *r* ≤*o r*′ **and**
3: *Card-order r* ∧ *Well-order r*
using *exists-minim-Well-order*[*of* {*r*. *phi r*}] *temp1 phi-def* **by** *blast*
let *?A* = *Field r* **let** *?r*′ = *bsqr r*
have *4*: *Well-order ?r*′ ∧ *Field ?r*′ = *?A* × *?A* ∧ |*?A*| =*o r*
using *3 bsqr-Well-order Field-bsqr card-of-Field-ordIso* **by** *blast*
have *5*: *Card-order* |*?A* × *?A*| ∧ *Well-order* |*?A* × *?A*|
using *card-of-Card-order card-of-Well-order* **by** *blast*

have *r* <*o* |*?A* × *?A*|
using *1 3 5 ordLess-or-ordLeq* **unfolding** *phi-def* **by** *blast*
moreover have |*?A* × *?A*| ≤*o ?r*′
using *card-of-least*[*of ?A* × *?A*] *4* **by** *auto*
ultimately have *r* <*o ?r*′ **using** *ordLess-ordLeq-trans* **by** *auto*
then obtain *f* **where** *6*: *embed r ?r*′ *f* **and** *7*: ¬ *bij-betw f ?A* (*?A* × *?A*)
unfolding *ordLess-def embedS-def*[*abs-def*]
by (*auto simp add*: *Field-bsqr*)

let *?B* = *f* ‘ *?A*
have |*?A*| =*o* |*?B*|
using *3 6 embed-inj-on inj-on-imp-bij-betw card-of-ordIso* **by** *blast*
hence *8*: *r* =*o* |*?B*| **using** *4 ordIso-transitive ordIso-symmetric* **by** *blast*

have *wo-rel.ofilter ?r′ ?B*
using *6 embed-Field-ofilter 3 4* **by** *blast*
hence *wo-rel.ofilter ?r′ ?B* ∧ *?B* ≠ *?A* × *?A* ∧ *?B* ≠ *Field ?r′*
using *7* **unfolding** *bij-betw-def* **using** *6 3 embed-inj-on 4* **by** *auto*
hence *temp2*: *wo-rel.ofilter ?r′ ?B* ∧ *?B* < *?A* × *?A*
using *4 wo-rel-def*[*of ?r′*] *wo-rel.ofilter-def*[*of ?r′ ?B*] **by** *blast*
have ¬ (∃ *a*. *Field r* = *under r a*)
using *1* **unfolding** *phi-def* **using** *Card-order-infinite-not-under*[*of r*] **by** *auto*
then obtain *A1* **where** *temp3*: *wo-rel.ofilter r A1* ∧ *A1* < *?A* **and** *9*: *?B* ≤ *A1* × *A1*
using *temp2 3 bsqr-ofilter*[*of r ?B*] **by** *blast*
hence |*?B*| ≤*o* |*A1* × *A1*| **using** *card-of-mono1* **by** *blast*
hence *10*: *r* ≤*o* |*A1* × *A1*| **using** *8 ordIso-ordLeq-trans* **by** *blast*
let *?r1* = *Restr r A1*
have *?r1* <*o r* **using** *temp3 ofilter-ordLess 3* **by** *blast*
moreover
{have *well-order-on A1 ?r1* **using** *3 temp3 well-order-on-Restr* **by** *blast*
hence |*A1*| ≤*o ?r1* **using** *3 Well-order-Restr card-of-least* **by** *blast*
}
ultimately have *11*: |*A1*| <*o r* **using** *ordLeq-ordLess-trans* **by** *blast*

have ¬ *finite* (*Field r*) **using** *1* **unfolding** *phi-def* **by** *simp*
hence ¬ *finite ?B* **using** *8 3 card-of-ordIso-finite-Field*[*of r ?B*] **by** *blast*
hence ¬ *finite A1* **using** *9 finite-cartesian-product finite-subset* **by** *blast*
moreover have *temp4*: *Field* |*A1*| = *A1* ∧ *Well-order* |*A1*| ∧ *Card-order* |*A1*|
using *card-of-Card-order*[*of A1*] *card-of-Well-order*[*of A1*]
by (*simp add*: *Field-card-of*)
moreover have ¬ *r* ≤*o* | *A1* |
using *temp4 11 3* **using** *not-ordLeq-iff-ordLess* **by** *blast*
ultimately have ¬ *finite*(*Field* |*A1*|) ∧ *Card-order* |*A1*| ∧ ¬ *r* ≤*o* | *A1* |
by (*simp add*: *card-of-card-order-on*)
hence |*Field* |*A1*| × *Field* |*A1*| | ≤*o* |*A1*|
using *2* **unfolding** *phi-def* **by** *blast*
hence |*A1* × *A1* | ≤*o* |*A1*| **using** *temp4* **by** *auto*
hence *r* ≤*o* |*A1*| **using** *10 ordLeq-transitive* **by** *blast*
thus *False* **using** *11 not-ordLess-ordLeq* **by** *auto*
qed
thus *?thesis* **using** *assms* **unfolding** *phi-def* **by** *blast*
qed

corollary *card-of-Times-same-infinite*:
assumes ¬*finite A*
shows |*A* × *A*| =*o* |*A*|
proof −

let *?r* = |*A*|
have *Field ?r* = *A* ∧ *Card-order ?r*
using *Field-card-of card-of-Card-order*[*of A*] **by** *fastforce*
hence |*A* × *A*| ≤*o* |*A*|
using *Card-order-Times-same-infinite*[*of ?r*] *assms* **by** *auto*
thus *?thesis* **using** *card-of-Times3 ordIso-iff-ordLeq* **by** *blast*
qed

lemma *card-of-Times-infinite*:
assumes *INF*: ¬*finite A* **and** *NE*: *B* ≠ {} **and** *LEQ*: |*B*| ≤*o* |*A*|
shows |*A* × *B*| =*o* |*A*| ∧ |*B* × *A*| =*o* |*A*|
proof −
have |*A*| ≤*o* |*A* × *B*| ∧ |*A*| ≤*o* |*B* × *A*|
using *assms* **by** (*simp add*: *card-of-Times1 card-of-Times2*)
moreover
{have |*A* × *B*| ≤*o* |*A* × *A*| ∧ |*B* × *A*| ≤*o* |*A* × *A*|
using *LEQ card-of-Times-mono1 card-of-Times-mono2* **by** *blast*
moreover have |*A* × *A*| =*o* |*A*| **using** *INF card-of-Times-same-infinite* **by** *blast*
ultimately have |*A* × *B*| ≤*o* |*A*| ∧ |*B* × *A*| ≤*o* |*A*|
using *ordLeq-ordIso-trans*[*of* |*A* × *B*|] *ordLeq-ordIso-trans*[*of* |*B* × *A*|] **by** *auto*
}
ultimately show *?thesis* **by** (*simp add*: *ordIso-iff-ordLeq*)
qed

corollary *Card-order-Times-infinite*:
assumes *INF*: ¬*finite*(*Field r*) **and** *CARD*: *Card-order r* **and**
NE: *Field p* ≠ {} **and** *LEQ*: *p* ≤*o r*
shows | (*Field r*) × (*Field p*) | =*o r* ∧ | (*Field p*) × (*Field r*) | =*o r*
proof −
have |*Field r* × *Field p*| =*o* |*Field r*| ∧ |*Field p* × *Field r*| =*o* |*Field r*|
using *assms* **by** (*simp add*: *card-of-Times-infinite card-of-mono2*)
thus *?thesis*
using *assms card-of-Field-ordIso* **by** (*blast intro*: *ordIso-transitive*)
qed

lemma *card-of-Sigma-ordLeq-infinite*:
assumes *INF*: ¬*finite B* **and**
LEQ-I: |*I*| ≤*o* |*B*| **and** *LEQ*: ∀ *i* ∈ *I*. |*A i*| ≤*o* |*B*|
shows |*SIGMA i* : *I*. *A i*| ≤*o* |*B*|
proof(*cases I* = {})
case *False*
have |*SIGMA i* : *I*. *A i*| ≤*o* |*I* × *B*|
using *card-of-Sigma-mono1*[*OF LEQ*] **by** *blast*
moreover have |*I* × *B*| =*o* |*B*|
using *INF False LEQ-I* **by** (*auto simp add*: *card-of-Times-infinite*)
ultimately show *?thesis* **using** *ordLeq-ordIso-trans* **by** *blast*
qed (*simp add*: *card-of-empty*)

lemma *card-of-Sigma-ordLeq-infinite-Field*:
assumes *INF*: ¬*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LEQ-I: $|I| \leq o\ r$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ r$
shows $|SIGMA\ i : I.\ A\ i| \leq o\ r$
proof −
let *?B* = *Field r*
have *1*: $r =o\ |?B| \wedge |?B| =o\ r$
using *r card-of-Field-ordIso ordIso-symmetric* **by** *blast*
hence $|I| \leq o\ |?B|\ \ \forall\, i \in I.\ |A\ i| \leq o\ |?B|$
using *LEQ-I LEQ ordLeq-ordIso-trans* **by** *blast*+
hence $|SIGMA\ i : I.\ A\ i| \leq o\ |?B|$ **using** *INF LEQ*
card-of-Sigma-ordLeq-infinite **by** *blast*
thus *?thesis* **using** *1 ordLeq-ordIso-trans* **by** *blast*
qed

lemma *card-of-Times-ordLeq-infinite-Field*:
$[\![\neg$*finite* (*Field r*); $|A| \leq o\ r$; $|B| \leq o\ r$; *Card-order r*$]\!] \Longrightarrow |A \times B| \leq o\ r$
by(*simp add*: *card-of-Sigma-ordLeq-infinite-Field*)

lemma *card-of-Times-infinite-simps*:
$[\![\neg$*finite A*; $B \neq \{\}$; $|B| \leq o\ |A|]\!] \Longrightarrow |A \times B| =o\ |A|$
$[\![\neg$*finite A*; $B \neq \{\}$; $|B| \leq o\ |A|]\!] \Longrightarrow |A| =o\ |A \times B|$
$[\![\neg$*finite A*; $B \neq \{\}$; $|B| \leq o\ |A|]\!] \Longrightarrow |B \times A| =o\ |A|$
$[\![\neg$*finite A*; $B \neq \{\}$; $|B| \leq o\ |A|]\!] \Longrightarrow |A| =o\ |B \times A|$
by (*auto simp add*: *card-of-Times-infinite ordIso-symmetric*)

lemma *card-of-UNION-ordLeq-infinite*:
assumes *INF*: ¬*finite B* **and** *LEQ-I*: $|I| \leq o\ |B|$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ |B|$
shows $|\bigcup i \in I.\ A\ i| \leq o\ |B|$
proof(*cases* $I = \{\}$)
case *False*
have $|\bigcup i \in I.\ A\ i| \leq o\ |SIGMA\ i : I.\ A\ i|$
using *card-of-UNION-Sigma* **by** *blast*
moreover have $|SIGMA\ i : I.\ A\ i| \leq o\ |B|$
using *assms card-of-Sigma-ordLeq-infinite* **by** *blast*
ultimately show *?thesis* **using** *ordLeq-transitive* **by** *blast*
qed (*simp add*: *card-of-empty*)

corollary *card-of-UNION-ordLeq-infinite-Field*:
assumes *INF*: ¬*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LEQ-I: $|I| \leq o\ r$ **and** *LEQ*: $\forall\, i \in I.\ |A\ i| \leq o\ r$
shows $|\bigcup i \in I.\ A\ i| \leq o\ r$
proof −
let *?B* = *Field r*
have *1*: $r =o\ |?B| \wedge |?B| =o\ r$
using *r card-of-Field-ordIso ordIso-symmetric* **by** *blast*
hence $|I| \leq o\ |?B|\ \ \forall\, i \in I.\ |A\ i| \leq o\ |?B|$
using *LEQ-I LEQ ordLeq-ordIso-trans* **by** *blast*+

hence $|\bigcup i \in I.\ A\ i| \leq o\ |?B|$ **using** *INF LEQ*
card-of-UNION-ordLeq-infinite **by** *blast*
thus *?thesis* **using** *1 ordLeq-ordIso-trans* **by** *blast*
qed

lemma *card-of-Plus-infinite1*:
assumes *INF*: ¬*finite A* **and** *LEQ*: $|B| \leq o\ |A|$
shows $|A <+> B| =o\ |A|$
proof(*cases* $B = \{\}$)
case *True*
then show *?thesis*
by (*simp add*: *card-of-Plus-empty1 card-of-Plus-empty2 ordIso-symmetric*)
next
case *False*
let *?Inl* = *Inl*::$'a \Rightarrow 'a + 'b$ **let** *?Inr* = *Inr*::$'b \Rightarrow 'a + 'b$
assume ∗: $B \neq \{\}$
then obtain *b1* **where** *1*: $b1 \in B$ **by** *blast*
show *?thesis*
proof(*cases* $B = \{b1\}$)
case *True*
have *2*: *bij-betw ?Inl A* ((*?Inl* ‘ *A*))
unfolding *bij-betw-def inj-on-def* **by** *auto*
hence *3*: ¬*finite* (*?Inl* ‘ *A*)
using *INF bij-betw-finite*[*of ?Inl A*] **by** *blast*
let $?A' = ?Inl$ ‘ $A \cup \{?Inr\ b1\}$
obtain *g* **where** *bij-betw g* (*?Inl* ‘ *A*) *?A′*
using *3 infinite-imp-bij-betw2*[*of ?Inl* ‘ *A*] **by** *auto*
moreover have $?A' = A <+> B$ **using** *True* **by** *blast*
ultimately have *bij-betw g* (*?Inl* ‘ *A*) $(A <+> B)$ **by** *simp*
hence *bij-betw* $(g \circ ?Inl)\ A\ (A <+> B)$
using *2* **by** (*auto simp add*: *bij-betw-trans*)
thus *?thesis* **using** *card-of-ordIso ordIso-symmetric* **by** *blast*
next
case *False*
with ∗ *1* **obtain** *b2* **where** *3*: $b1 \neq b2 \wedge \{b1,b2\} \leq B$ **by** *fastforce*
obtain *f* **where** *inj-on f B* $\wedge$ *f* ‘ $B \leq A$
using *LEQ card-of-ordLeq*[*of B*] **by** *fastforce*
with *3* **have** $f\ b1 \neq f\ b2 \wedge \{f\ b1, f\ b2\} \leq A$
unfolding *inj-on-def* **by** *auto*
with *3* **have** $|A <+> B| \leq o\ |A \times B|$
by (*auto simp add*: *card-of-Plus-Times*)
moreover have $|A \times B| =o\ |A|$
using *assms* ∗ **by** (*simp add*: *card-of-Times-infinite-simps*)
ultimately have $|A <+> B| \leq o\ |A|$ **using** *ordLeq-ordIso-trans* **by** *blast*
thus *?thesis* **using** *card-of-Plus1 ordIso-iff-ordLeq* **by** *blast*
qed
qed

lemma *card-of-Plus-infinite2*:

assumes *INF*: ¬*finite A* **and** *LEQ*: $|B| \leq o\ |A|$
shows $|B <+> A| =o\ |A|$
using *assms card-of-Plus-commute card-of-Plus-infinite1*
ordIso-equivalence **by** *blast*

lemma *card-of-Plus-infinite*:
assumes *INF*: ¬*finite A* **and** *LEQ*: $|B| \leq o\ |A|$
shows $|A <+> B| =o\ |A| \wedge |B <+> A| =o\ |A|$
using *assms* **by** (*auto simp*: *card-of-Plus-infinite1 card-of-Plus-infinite2*)

corollary *Card-order-Plus-infinite*:
assumes *INF*: ¬*finite*(*Field r*) **and** *CARD*: *Card-order r* **and**
LEQ: $p \leq o\ r$
shows $|\ (Field\ r) <+> (Field\ p)\ | =o\ r \wedge |\ (Field\ p) <+> (Field\ r)\ | =o\ r$
proof −
have $|\ Field\ r <+> Field\ p\ | =o\ |\ Field\ r\ | \wedge$
$|\ Field\ p <+> Field\ r\ | =o\ |\ Field\ r\ |$
using *assms* **by** (*simp add*: *card-of-Plus-infinite card-of-mono2*)
thus *?thesis*
using *assms card-of-Field-ordIso* **by** (*blast intro*: *ordIso-transitive*)

qed

30.5 The cardinal ω and the finite cardinals

The cardinal ω, of natural numbers, shall be the standard non-strict order relation on *nat*, that we abbreviate by *natLeq*. The finite cardinals shall be the restrictions of these relations to the numbers smaller than fixed numbers n, that we abbreviate by *natLeq-on n*.

definition (*natLeq*::(*nat* ∗ *nat*) *set*) ≡ $\{(x,y).\ x \leq y\}$
definition (*natLess*::(*nat* ∗ *nat*) *set*) ≡ $\{(x,y).\ x < y\}$

abbreviation *natLeq-on* :: *nat* ⇒ (*nat* ∗ *nat*) *set*
where *natLeq-on n* ≡ $\{(x,y).\ x < n \wedge y < n \wedge x \leq y\}$

lemma *infinite-cartesian-product*:
assumes ¬*finite A* ¬*finite B*
shows ¬*finite* (*A* × *B*)
using *assms finite-cartesian-productD2* **by** *auto*

30.5.1 First as well-orders

lemma *Field-natLeq*: *Field natLeq* = (*UNIV*::*nat set*)
by(*unfold Field-def natLeq-def*, *auto*)

lemma *natLeq-Refl*: *Refl natLeq*
unfolding *refl-on-def Field-def natLeq-def* **by** *auto*

lemma *natLeq-trans*: *trans natLeq*

unfolding *trans-def natLeq-def* **by** *auto*

lemma *natLeq-Preorder*: *Preorder natLeq*
unfolding *preorder-on-def*
proof (*intro conjI*)
show *natLeq* ⊆ *Field natLeq* × *Field natLeq*
unfolding *natLeq-def Field-def* **by** *blast*
next
show *Refl natLeq*
using *natLeq-Refl* **.**
next
show *trans natLeq*
using *natLeq-trans* **.**
qed

lemma *natLeq-antisym*: *antisym natLeq*
unfolding *antisym-def natLeq-def* **by** *auto*

lemma *natLeq-Partial-order*: *Partial-order natLeq*
unfolding *partial-order-on-def*
by (*auto simp add*: *natLeq-Preorder natLeq-antisym*)

lemma *natLeq-Total*: *Total natLeq*
unfolding *total-on-def natLeq-def* **by** *auto*

lemma *natLeq-Linear-order*: *Linear-order natLeq*
unfolding *linear-order-on-def*
by (*auto simp add*: *natLeq-Partial-order natLeq-Total*)

lemma *natLeq-natLess-Id*: *natLess* = *natLeq* − *Id*
unfolding *natLeq-def natLess-def* **by** *auto*

lemma *natLeq-Well-order*: *Well-order natLeq*
unfolding *well-order-on-def*
using *natLeq-Linear-order wf-less natLeq-natLess-Id natLeq-def natLess-def* **by** *auto*

lemma *Field-natLeq-on*: *Field* (*natLeq-on n*) = {*x*. *x* < *n*}
unfolding *Field-def* **by** *auto*

lemma *natLeq-underS-less*: *underS natLeq n* = {*x*. *x* < *n*}
unfolding *underS-def natLeq-def* **by** *auto*

lemma *Restr-natLeq*: *Restr natLeq* {*x*. *x* < *n*} = *natLeq-on n*
unfolding *natLeq-def* **by** *force*

lemma *Restr-natLeq2*:
Restr natLeq (*underS natLeq n*) = *natLeq-on n*
by (*auto simp add*: *Restr-natLeq natLeq-underS-less*)

lemma *natLeq-on-Well-order*: *Well-order*(*natLeq-on n*)
using *Restr-natLeq*[*of n*] *natLeq-Well-order*
Well-order-Restr[*of natLeq* {*x*. *x* < *n*}] **by** *auto*

corollary *natLeq-on-well-order-on*: *well-order-on* {*x*. *x* < *n*} (*natLeq-on n*)
using *natLeq-on-Well-order Field-natLeq-on* **by** *auto*

lemma *natLeq-on-wo-rel*: *wo-rel*(*natLeq-on n*)
unfolding *wo-rel-def* **using** *natLeq-on-Well-order* **.**

30.5.2 Then as cardinals

lemma *natLeq-Card-order*: *Card-order natLeq*
proof −
have *natLeq-on n* <*o* |*UNIV*::*nat set*| **for** *n*
proof −
have *finite*(*Field* (*natLeq-on n*)) **by** (*auto simp*: *Field-def*)
moreover have ¬*finite*(*UNIV*::*nat set*) **by** *auto*
ultimately show *?thesis*
using *finite-ordLess-infinite*[*of natLeq-on n* |*UNIV*::*nat set*|]
card-of-Well-order[*of UNIV*::*nat set*] *natLeq-on-Well-order*
by (*force simp add*: *Field-card-of*)
qed
then show *?thesis*
apply (*simp add*: *natLeq-Well-order Card-order-iff-Restr-underS Restr-natLeq2*)
apply (*force simp add*: *Field-natLeq*)
done
qed

corollary *card-of-Field-natLeq*:
|*Field natLeq*| =*o natLeq*
using *Field-natLeq natLeq-Card-order Card-order-iff-ordIso-card-of*[*of natLeq*]
ordIso-symmetric[*of natLeq*] **by** *blast*

corollary *card-of-nat*:
|*UNIV*::*nat set*| =*o natLeq*
using *Field-natLeq card-of-Field-natLeq* **by** *auto*

corollary *infinite-iff-natLeq-ordLeq*:
¬*finite A* = (*natLeq* ≤*o* |*A*|)
using *infinite-iff-card-of-nat*[*of A*] *card-of-nat*
ordIso-ordLeq-trans ordLeq-ordIso-trans ordIso-symmetric **by** *blast*

corollary *finite-iff-ordLess-natLeq*:
finite A = (|*A*| <*o natLeq*)
using *infinite-iff-natLeq-ordLeq not-ordLeq-iff-ordLess*
card-of-Well-order natLeq-Well-order **by** *blast*

30.6 The successor of a cardinal

First we define *isCardSuc r r'*, the notion of r' being a successor cardinal of r. Although the definition does not require r to be a cardinal, only this case will be meaningful.

definition *isCardSuc* :: *'a rel ⇒ 'a set rel ⇒ bool*
  where
    isCardSuc r r' ≡
        Card-order r' ∧ r <o r' ∧
        (∀(r''::'a set rel). Card-order r'' ∧ r <o r'' ⟶ r' ≤o r'')

Now we introduce the cardinal-successor operator *cardSuc*, by picking *some* cardinal-order relation fulfilling *isCardSuc*. Again, the picked item shall be proved unique up to order-isomorphism.

definition *cardSuc* :: *'a rel ⇒ 'a set rel*
  where *cardSuc r ≡ SOME r'. isCardSuc r r'*

lemma *exists-minim-Card-order*:
  ⟦R ≠ {}; ∀r ∈ R. Card-order r⟧ ⟹ ∃r ∈ R. ∀r' ∈ R. r ≤o r'
  unfolding *card-order-on-def* **using** *exists-minim-Well-order* **by** *blast*

lemma *exists-isCardSuc*:
  assumes *Card-order r*
  shows *∃r'. isCardSuc r r'*
proof −
  let *?R = {(r'::'a set rel). Card-order r' ∧ r <o r'}*
  have *|Pow(Field r)| ∈ ?R ∧ (∀r ∈ ?R. Card-order r)* **using** *assms*
    by (*simp add*: *card-of-Card-order Card-order-Pow*)
  then obtain *r* **where** *r ∈ ?R ∧ (∀r' ∈ ?R. r ≤o r')*
    using *exists-minim-Card-order*[*of ?R*] **by** *blast*
  thus *?thesis* **unfolding** *isCardSuc-def* **by** *auto*
qed

lemma *cardSuc-isCardSuc*:
  assumes *Card-order r*
  shows *isCardSuc r* (*cardSuc r*)
  unfolding *cardSuc-def* **using** *assms*
  by (*simp add*: *exists-isCardSuc someI-ex*)

lemma *cardSuc-Card-order*:
  Card-order r ⟹ Card-order(*cardSuc r*)
  using *cardSuc-isCardSuc* **unfolding** *isCardSuc-def* **by** *blast*

lemma *cardSuc-greater*:
  Card-order r ⟹ r <o cardSuc r
  using *cardSuc-isCardSuc* **unfolding** *isCardSuc-def* **by** *blast*

lemma *cardSuc-ordLeq*:

Card-order r $\Longrightarrow$ *r* $\leq o$ *cardSuc r*
using *cardSuc-greater ordLeq-iff-ordLess-or-ordIso* **by** *blast*

The minimality property of *cardSuc* originally present in its definition is local to the type *'a set rel*, i.e., that of *cardSuc r*:

lemma *cardSuc-least-aux*:
⟦*Card-order* (*r*::*'a rel*); *Card-order* (*r'*::*'a set rel*); *r* $<o$ *r'*⟧ $\Longrightarrow$ *cardSuc r* $\leq o$ *r'*
using *cardSuc-isCardSuc* **unfolding** *isCardSuc-def* **by** *blast*

But from this we can infer general minimality:

lemma *cardSuc-least*:
assumes *CARD*: *Card-order r* **and** *CARD'*: *Card-order r'* **and** *LESS*: *r* $<o$ *r'*
shows *cardSuc r* $\leq o$ *r'*
proof −
let *?p* = *cardSuc r*
have *0*: *Well-order ?p* ∧ *Well-order r'*
using *assms cardSuc-Card-order* **unfolding** *card-order-on-def* **by** *blast*
{ assume *r'* $<o$ *?p*
then obtain *r''* **where** *1*: *Field r''* < *Field ?p* **and** *2*: *r'* $=o$ *r''* ∧ *r''* $<o$ *?p*
using *internalize-ordLess*[*of r' ?p*] **by** *blast*

have *Card-order r''* **using** *CARD' Card-order-ordIso2 2* **by** *blast*
moreover have *r* $<o$ *r''* **using** *LESS 2 ordLess-ordIso-trans* **by** *blast*
ultimately have *?p* $\leq o$ *r''* **using** *cardSuc-least-aux CARD* **by** *blast*
hence *False* **using** *2 not-ordLess-ordLeq* **by** *blast*
}
thus *?thesis* **using** *0 ordLess-or-ordLeq* **by** *blast*
qed

lemma *cardSuc-ordLess-ordLeq*:
assumes *CARD*: *Card-order r* **and** *CARD'*: *Card-order r'*
shows (*r* $<o$ *r'*) = (*cardSuc r* $\leq o$ *r'*)
proof
show *cardSuc r* $\leq o$ *r'* $\Longrightarrow$ *r* $<o$ *r'*
using *assms cardSuc-greater ordLess-ordLeq-trans* **by** *blast*
qed (*auto simp add*: *assms cardSuc-least*)

lemma *cardSuc-ordLeq-ordLess*:
assumes *CARD*: *Card-order r* **and** *CARD'*: *Card-order r'*
shows (*r'* $<o$ *cardSuc r*) = (*r'* $\leq o$ *r*)
proof −
have *Well-order r* ∧ *Well-order r'*
using *assms* **unfolding** *card-order-on-def* **by** *auto*
moreover have *Well-order*(*cardSuc r*)
using *assms cardSuc-Card-order card-order-on-def* **by** *blast*
ultimately show *?thesis*
using *assms cardSuc-ordLess-ordLeq* **by** (*blast dest*: *not-ordLeq-iff-ordLess*)
qed

lemma *cardSuc-mono-ordLeq*:
assumes *CARD*: *Card-order r* **and** *CARD'*: *Card-order r'*
shows (*cardSuc r* $\leq o$ *cardSuc r'*) = (*r* $\leq o$ *r'*)
using *assms cardSuc-ordLeq-ordLess cardSuc-ordLess-ordLeq cardSuc-Card-order*
by *blast*

lemma *cardSuc-invar-ordIso*:
assumes *CARD*: *Card-order r* **and** *CARD'*: *Card-order r'*
shows (*cardSuc r* $=o$ *cardSuc r'*) = (*r* $=o$ *r'*)
proof −
have *0*: *Well-order r* ∧ *Well-order r'* ∧ *Well-order*(*cardSuc r*) ∧ *Well-order*(*cardSuc r'*)
using *assms* **by** (*simp add*: *card-order-on-well-order-on cardSuc-Card-order*)
thus *?thesis*
using *ordIso-iff-ordLeq*[*of r r'*] *ordIso-iff-ordLeq*
using *cardSuc-mono-ordLeq*[*of r r'*] *cardSuc-mono-ordLeq*[*of r' r*] *assms* **by** *blast*
qed

lemma *card-of-cardSuc-finite*:
finite(*Field*(*cardSuc* |*A*|)) = *finite A*
proof
assume ∗: *finite* (*Field* (*cardSuc* |*A*|))
have *0*: |*Field*(*cardSuc* |*A*|)| $=o$ *cardSuc* |*A*|
using *card-of-Card-order cardSuc-Card-order card-of-Field-ordIso* **by** *blast*
hence |*A*| $\leq o$ |*Field*(*cardSuc* |*A*|)|
using *card-of-Card-order*[*of A*] *cardSuc-ordLeq*[*of* |*A*|] *ordIso-symmetric ordLeq-ordIso-trans* **by** *blast*
thus *finite A* **using** ∗ *card-of-ordLeq-finite* **by** *blast*
next
assume *finite A*
then have *finite* (*Field* |*Pow A*|) **unfolding** *Field-card-of* **by** *simp*
moreover
have *cardSuc* |*A*| $\leq o$ |*Pow A*|
by (*rule iffD1*[*OF cardSuc-ordLess-ordLeq card-of-Pow*]) (*simp-all add*: *card-of-Card-order*)
ultimately show *finite* (*Field* (*cardSuc* |*A*|))
by (*blast intro*: *card-of-ordLeq-finite card-of-mono2*)
qed

lemma *cardSuc-finite*:
assumes *Card-order r*
shows *finite* (*Field* (*cardSuc r*)) = *finite* (*Field r*)
proof −
let *?A* = *Field r*
have |*?A*| $=o$ *r* **using** *assms* **by** (*simp add*: *card-of-Field-ordIso*)
hence *cardSuc* |*?A*| $=o$ *cardSuc r* **using** *assms*
by (*simp add*: *card-of-Card-order cardSuc-invar-ordIso*)
moreover have |*Field* (*cardSuc* |*?A*|) | $=o$ *cardSuc* |*?A*|
by (*simp add*: *card-of-card-order-on Field-card-of card-of-Field-ordIso card-*

Suc-Card-order)
moreover
{ have |*Field* (*cardSuc r*)| =*o cardSuc r*
using *assms* **by** (*simp add*: *card-of-Field-ordIso cardSuc-Card-order*)
hence *cardSuc r* =*o* |*Field* (*cardSuc r*)|
using *ordIso-symmetric* **by** *blast*
}
ultimately have |*Field* (*cardSuc* |*?A*|)| =*o* |*Field* (*cardSuc r*)|
using *ordIso-transitive* **by** *blast*
hence *finite* (*Field* (*cardSuc* |*?A*|)) = *finite* (*Field* (*cardSuc r*))
using *card-of-ordIso-finite* **by** *blast*
thus *?thesis* **by** (*simp only*: *card-of-cardSuc-finite*)
qed

lemma *Field-cardSuc-not-empty*:
assumes *Card-order r*
shows *Field* (*cardSuc r*) ≠ {}
proof
assume *Field*(*cardSuc r*) = {}
then have |*Field*(*cardSuc r*)| ≤*o r* **using** *assms Card-order-empty*[*of r*] **by** *auto*
then have *cardSuc r* ≤*o r* **using** *assms card-of-Field-ordIso*
cardSuc-Card-order ordIso-symmetric ordIso-ordLeq-trans **by** *blast*
then show *False* **using** *cardSuc-greater not-ordLess-ordLeq assms* **by** *blast*
qed

typedef *'a suc* = *Field* (*cardSuc* |*UNIV* :: *'a set*|)
using *Field-cardSuc-not-empty card-of-Card-order* **by** *blast*

definition *card-suc* :: *'a rel* ⇒ *'a suc rel* **where**
card-suc ≡ λ-. *map-prod Abs-suc Abs-suc* ‘ *cardSuc* |*UNIV* :: *'a set*|

lemma *Field-card-suc*: *Field* (*card-suc r*) = *UNIV*
proof −
let *?r* = *cardSuc* |*UNIV*|
let *?ar* = λ*x*. *Abs-suc* (*Rep-suc x*)
have *1*: ⋀*P*. (∀ *x*∈*Field ?r*. *P x*) = (∀ *x*. *P* (*Rep-suc x*)) **using** *Rep-suc-induct Rep-suc* **by** *blast*
have *2*: ⋀*P*. (∃ *x*∈*Field ?r*. *P x*) = (∃ *x*. *P* (*Rep-suc x*)) **using** *Rep-suc-cases Rep-suc* **by** *blast*
have *3*: ⋀*A a b*. (*a*, *b*) ∈ *A* ⟹ (*Abs-suc a*, *Abs-suc b*) ∈ *map-prod Abs-suc Abs-suc* ‘ *A* **unfolding** *map-prod-def* **by** *auto*
have ∀ *x*∈*Field ?r*. (∃ *b*∈*Field ?r*. (*x*, *b*) ∈ *?r*) ∨ (∃ *a*∈*Field ?r*. (*a*, *x*) ∈ *?r*)
unfolding *Field-def Range.simps Domain.simps Un-iff* **by** *blast*
then have ∀ *x*. (∃ *b*. (*Rep-suc x*, *Rep-suc b*) ∈ *?r*) ∨ (∃ *a*. (*Rep-suc a*, *Rep-suc x*) ∈ *?r*) **unfolding** *1 2* **.**
then have ∀ *x*. (∃ *b*. (*?ar x*, *?ar b*) ∈ *map-prod Abs-suc Abs-suc* ‘ *?r*) ∨ (∃ *a*. (*?ar a*, *?ar x*) ∈ *map-prod Abs-suc Abs-suc* ‘ *?r*) **using** *3* **by** *fast*
then have ∀ *x*. (∃ *b*. (*x*, *b*) ∈ *card-suc r*) ∨ (∃ *a*. (*a*, *x*) ∈ *card-suc r*) **unfolding** *card-suc-def Rep-suc-inverse* **.**

then show *?thesis* **unfolding** *Field-def Domain.simps Range.simps set-eq-iff Un-iff eqTrueI*[*OF UNIV-I*] *ex-simps simp-thms* **.**
qed

lemma *card-suc-alt*: *card-suc r* = *dir-image* (*cardSuc* |*UNIV* :: *'a set*|) *Abs-suc*
unfolding *card-suc-def dir-image-def* **by** *auto*

lemma *cardSuc-Well-order*: *Card-order r* ⟹ *Well-order*(*cardSuc r*)
using *cardSuc-Card-order* **unfolding** *card-order-on-def* **by** *blast*

lemma *cardSuc-ordIso-card-suc*:
assumes *card-order r*
shows *cardSuc r* =*o card-suc* (*r* :: *'a rel*)
proof −
have *cardSuc* (*r* :: *'a rel*) =*o cardSuc* |*UNIV* :: *'a set*|
using *cardSuc-invar-ordIso*[*THEN iffD2*, *OF - card-of-Card-order card-of-unique*[*OF assms*]] *assms*
by (*simp add*: *card-order-on-Card-order*)
also have *cardSuc* |*UNIV* :: *'a set*| =*o card-suc* (*r* :: *'a rel*)
unfolding *card-suc-alt*
by (*rule dir-image-ordIso*) (*simp-all add*: *inj-on-def Abs-suc-inject cardSuc-Well-order card-of-Card-order*)
finally show *?thesis* **.**
qed

lemma *Card-order-card-suc*: *card-order r* ⟹ *Card-order* (*card-suc r*)
using *cardSuc-Card-order*[*THEN Card-order-ordIso2*[*OF - cardSuc-ordIso-card-suc*]] *card-order-on-Card-order* **by** *blast*

lemma *card-order-card-suc*: *card-order r* ⟹ *card-order* (*card-suc r*)
using *Card-order-card-suc arg-cong2*[*OF Field-card-suc refl*, *of card-order-on*] **by** *blast*

lemma *card-suc-greater*: *card-order r* ⟹ *r* <*o card-suc r*
using *ordLess-ordIso-trans*[*OF cardSuc-greater cardSuc-ordIso-card-suc*] *card-order-on-Card-order* **by** *blast*

lemma *card-of-Plus-ordLess-infinite*:
assumes *INF*: ¬*finite C* **and** *LESS1*: |*A*| <*o* |*C*| **and** *LESS2*: |*B*| <*o* |*C*|
shows |*A* <+> *B*| <*o* |*C*|
proof(*cases A* = {} ∨ *B* = {})
case *True*
hence |*A*| =*o* |*A* <+> *B*| ∨ |*B*| =*o* |*A* <+> *B*|
using *card-of-Plus-empty1 card-of-Plus-empty2* **by** *blast*
hence |*A* <+> *B*| =*o* |*A*| ∨ |*A* <+> *B*| =*o* |*B*|
using *ordIso-symmetric*[*of* |*A*|] *ordIso-symmetric*[*of* |*B*|] **by** *blast*
thus *?thesis* **using** *LESS1 LESS2*
ordIso-ordLess-trans[*of* |*A* <+> *B*| |*A*|]
ordIso-ordLess-trans[*of* |*A* <+> *B*| |*B*|] **by** *blast*

next
case *False*
have *False* **if** $|C| \leq o\ |A <+> B|$
proof −
have §: $\neg$*finite* $A \vee \neg$*finite* B
using *that INF card-of-ordLeq-finite finite-Plus* **by** *blast*
consider $|A| \leq o\ |B| \mid |B| \leq o\ |A|$
using *ordLeq-total* [*OF card-of-Well-order card-of-Well-order*] **by** *blast*
then show *False*
proof *cases*
case *1*
hence $\neg$*finite* B **using** § *card-of-ordLeq-finite* **by** *blast*
hence $|A <+> B| =o\ |B|$ **using** *False 1*
by (*auto simp add*: *card-of-Plus-infinite*)
thus *False* **using** *LESS2 not-ordLess-ordLeq that ordLeq-ordIso-trans* **by** *blast*
next
case *2*
hence $\neg$*finite* A **using** § *card-of-ordLeq-finite* **by** *blast*
hence $|A <+> B| =o\ |A|$ **using** *False 2*
by (*auto simp add*: *card-of-Plus-infinite*)
thus *False* **using** *LESS1 not-ordLess-ordLeq that ordLeq-ordIso-trans* **by** *blast*
qed
qed
thus *?thesis*
using *ordLess-or-ordLeq*[*of* $|A <+> B|$ $|C|$]
card-of-Well-order[*of* $A <+> B$] *card-of-Well-order*[*of* C] **by** *auto*
qed

lemma *card-of-Plus-ordLess-infinite-Field*:
assumes *INF*: $\neg$*finite* (*Field* r) **and** r: *Card-order* r **and**
LESS1: $|A| <o\ r$ **and** *LESS2*: $|B| <o\ r$
shows $|A <+> B| <o\ r$
proof −
let *?C* = *Field* r
have *1*: $r =o\ |?C| \wedge |?C| =o\ r$
using r *card-of-Field-ordIso ordIso-symmetric* **by** *blast*
hence $|A| <o\ |?C|$ $|B| <o\ |?C|$
using *LESS1 LESS2 ordLess-ordIso-trans* **by** *blast*+
hence $|A <+> B| <o\ |?C|$ **using** *INF*
card-of-Plus-ordLess-infinite **by** *blast*
thus *?thesis* **using** *1 ordLess-ordIso-trans* **by** *blast*
qed

lemma *card-of-Plus-ordLeq-infinite-Field*:
assumes r: $\neg$*finite* (*Field* r) **and** A: $|A| \leq o\ r$ **and** B: $|B| \leq o\ r$
and c: *Card-order* r
shows $|A <+> B| \leq o\ r$
proof −
let *?r'* = *cardSuc* r

have *Card-order ?r′* ∧ ¬*finite* (*Field ?r′*) **using** *assms*
by (*simp add*: *cardSuc-Card-order cardSuc-finite*)
moreover have |*A*| <*o* *?r′* **and** |*B*| <*o* *?r′* **using** *A B c*
by (*auto simp*: *card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess*)
ultimately have |*A* <+> *B*| <*o* *?r′*
using *card-of-Plus-ordLess-infinite-Field* **by** *blast*
thus *?thesis* **using** *c r*
by (*simp add*: *card-of-card-order-on Field-card-of cardSuc-ordLeq-ordLess*)
qed

lemma *card-of-Un-ordLeq-infinite-Field*:
assumes *C*: ¬*finite* (*Field r*) **and** *A*: |*A*| ≤*o* *r* **and** *B*: |*B*| ≤*o* *r*
and *Card-order r*
shows |*A Un B*| ≤*o* *r*
using *assms card-of-Plus-ordLeq-infinite-Field card-of-Un-Plus-ordLeq*
ordLeq-transitive **by** *fast*

lemma *card-of-Un-ordLess-infinite*:
assumes *INF*: ¬*finite C* **and**
LESS1: |*A*| <*o* |*C*| **and** *LESS2*: |*B*| <*o* |*C*|
shows |*A* ∪ *B*| <*o* |*C*|
using *assms card-of-Plus-ordLess-infinite card-of-Un-Plus-ordLeq*
ordLeq-ordLess-trans **by** *blast*

lemma *card-of-Un-ordLess-infinite-Field*:
assumes *INF*: ¬*finite* (*Field r*) **and** *r*: *Card-order r* **and**
LESS1: |*A*| <*o* *r* **and** *LESS2*: |*B*| <*o* *r*
shows |*A Un B*| <*o* *r*
proof −
let *?C* = *Field r*
have *1*: *r* =*o* |*?C*| ∧ |*?C*| =*o* *r* **using** *r card-of-Field-ordIso*
ordIso-symmetric **by** *blast*
hence |*A*| <*o* |*?C*| |*B*| <*o* |*?C*|
using *LESS1 LESS2 ordLess-ordIso-trans* **by** *blast*+
hence |*A Un B*| <*o* |*?C*| **using** *INF*
card-of-Un-ordLess-infinite **by** *blast*
thus *?thesis* **using** *1 ordLess-ordIso-trans* **by** *blast*
qed

30.7 Regular cardinals

definition *cofinal* **where**
cofinal A r ≡ ∀ *a* ∈ *Field r*. ∃ *b* ∈ *A*. *a* ≠ *b* ∧ (*a*,*b*) ∈ *r*

definition *regularCard* **where**
regularCard r ≡ ∀ *K*. *K* ≤ *Field r* ∧ *cofinal K r* ⟶ |*K*| =*o* *r*

definition *relChain* **where**
relChain r As ≡ ∀ *i j*. (*i*,*j*) ∈ *r* ⟶ *As i* ≤ *As j*

lemma *regularCard-UNION*:
assumes *r*: *Card-order r* *regularCard r*
and *As*: *relChain r As*
and *Bsub*: $B \leq (\bigcup i \in Field\ r.\ As\ i)$
and *cardB*: $|B| <o\ r$
shows $\exists i \in Field\ r.\ B \leq As\ i$
proof −
let *?phi* = $\lambda b\ j.\ j \in Field\ r \wedge b \in As\ j$
have $\forall b \in B.\ \exists j.$ *?phi b j* **using** *Bsub* **by** *blast*
then obtain *f* **where** *f*: $\bigwedge b.\ b \in B \Longrightarrow$ *?phi b* (*f b*)
using *bchoice*[*of B ?phi*] **by** *blast*
let *?K* = *f* ‘ *B*
{assume *1*: $\bigwedge i.\ i \in Field\ r \Longrightarrow \neg\ B \leq As\ i$
have *2*: *cofinal ?K r*
unfolding *cofinal-def*
proof (*intro strip*)
fix *i* **assume** *i*: $i \in Field\ r$
with *1* **obtain** *b* **where** *b*: $b \in B \wedge b \notin As\ i$ **by** *blast*
hence $i \neq f\ b \wedge \neg\ (f\ b,i) \in r$
using *As f* **unfolding** *relChain-def* **by** *auto*
hence $i \neq f\ b \wedge (i, f\ b) \in r$ **using** *r*
unfolding *card-order-on-def well-order-on-def linear-order-on-def*
total-on-def **using** *i f b* **by** *auto*
with *b* **show** $\exists b \in f`B.\ i \neq b \wedge (i,b) \in r$ **by** *blast*
qed
moreover have *?K* $\leq$ *Field r* **using** *f* **by** *blast*
ultimately have $|?K| =o\ r$ **using** *2 r* **unfolding** *regularCard-def* **by** *blast*
moreover
have $|?K| <o\ r$ **using** *cardB ordLeq-ordLess-trans card-of-image* **by** *blast*
ultimately have *False* **using** *not-ordLess-ordIso* **by** *blast*
}
thus *?thesis* **by** *blast*
qed

lemma *infinite-cardSuc-regularCard*:
assumes *r-inf*: ¬*finite* (*Field r*) **and** *r-card*: *Card-order r*
shows *regularCard* (*cardSuc r*)
proof −
let *?r′* = *cardSuc r*
have *r′*: *Card-order ?r′* $\bigwedge p.$ *Card-order p* $\longrightarrow$ $(p \leq o\ r) = (p <o\ ?r')$
using *r-card* **by** (*auto simp*: *cardSuc-Card-order cardSuc-ordLeq-ordLess*)
show *?thesis*
unfolding *regularCard-def* **proof** *auto*
fix *K* **assume** *1*: $K \leq$ *Field ?r′* **and** *2*: *cofinal K ?r′*
hence $|K| \leq o\ |Field\ ?r'|$ **by** (*simp only*: *card-of-mono1*)
also have *22*: $|Field\ ?r'| =o\ ?r'$
using *r′* **by** (*simp add*: *card-of-Field-ordIso*[*of ?r′*])
finally have $|K| \leq o\ ?r'$ **.**

```
    moreover
    { let ?L = ⋃ j ∈ K. underS ?r' j
      let ?J = Field r
      have rJ: r =o |?J|
        using r-card card-of-Field-ordIso ordIso-symmetric by blast
      assume |K| <o ?r'
      hence |K| ≤o r using r' card-of-Card-order[of K] by blast
      hence |K| ≤o |?J| using rJ ordLeq-ordIso-trans by blast
      moreover
      {have ∀j∈K. |underS ?r' j| <o ?r'
         using r' 1 by (auto simp: card-of-underS)
        hence ∀j∈K. |underS ?r' j| ≤o r
          using r' card-of-Card-order by blast
        hence ∀j∈K. |underS ?r' j| ≤o |?J|
          using rJ ordLeq-ordIso-trans by blast
      }
      ultimately have |?L| ≤o |?J|
        using r-inf card-of-UNION-ordLeq-infinite by blast
      hence |?L| ≤o r using rJ ordIso-symmetric ordLeq-ordIso-trans by blast
      hence |?L| <o ?r' using r' card-of-Card-order by blast
      moreover
      {
        have Field ?r' ≤ ?L
          using 2 unfolding underS-def cofinal-def by auto
        hence |Field ?r'| ≤o |?L| by (simp add: card-of-mono1)
        hence ?r' ≤o |?L|
          using 22 ordIso-ordLeq-trans ordIso-symmetric by blast
      }
      ultimately have |?L| <o |?L| using ordLess-ordLeq-trans by blast
      hence False using ordLess-irreflexive by blast
    }
    ultimately show |K| =o ?r'
      unfolding ordLeq-iff-ordLess-or-ordIso by blast
  qed
qed

lemma cardSuc-UNION:
  assumes r: Card-order r and ¬finite (Field r)
    and As: relChain (cardSuc r) As
    and Bsub: B ≤ (⋃ i ∈ Field (cardSuc r). As i)
    and cardB: |B| ≤o r
  shows ∃ i ∈ Field (cardSuc r). B ≤ As i
proof −
  let ?r' = cardSuc r
  have Card-order ?r' ∧ |B| <o ?r'
    using r cardB cardSuc-ordLeq-ordLess cardSuc-Card-order
      card-of-Card-order by blast
  moreover have regularCard ?r'
    using assms by(simp add: infinite-cardSuc-regularCard)
```

ultimately show *?thesis*
using *As Bsub cardB regularCard-UNION* **by** *blast*
qed

30.8 Others

lemma *card-of-Func-Times*:
$|Func\ (A \times B)\ C| =o |Func\ A\ (Func\ B\ C)|$
unfolding *card-of-ordIso*[*symmetric*]
using *bij-betw-curr* **by** *blast*

lemma *card-of-Pow-Func*:
$|Pow\ A| =o |Func\ A\ (UNIV::bool\ set)|$
proof −
define *F* **where** [*abs-def*]: $F\ A'\ a \equiv$
$(if\ a \in A\ then\ (if\ a \in A'\ then\ True\ else\ False)\ else\ undefined)$ **for** $A'\ a$
have *bij-betw F* (*Pow A*) (*Func A* (*UNIV*::*bool set*))
unfolding *bij-betw-def inj-on-def* **proof** (*intro ballI impI conjI*)
fix *A1 A2* **assume** $A1 \in Pow\ A\ A2 \in Pow\ A\ F\ A1 = F\ A2$
thus $A1 = A2$ **unfolding** *F-def Pow-def fun-eq-iff* **by** (*auto split*: *if-split-asm*)
next
show $F\ `\ Pow\ A = Func\ A\ UNIV$
proof *safe*
fix *f* **assume** *f*: $f \in Func\ A\ (UNIV::bool\ set)$
show $f \in F\ `\ Pow\ A$
unfolding *image-iff*
proof
show $f = F\ \{a \in A.\ f\ a = True\}$
using *f* **unfolding** *Func-def F-def* **by** *force*
qed *auto*
qed(*unfold Func-def F-def*, *auto*)
qed
thus *?thesis* **unfolding** *card-of-ordIso*[*symmetric*] **by** *blast*
qed

lemma *card-of-Func-UNIV*:
$|Func\ (UNIV::'a\ set)\ (B::'b\ set)| =o |\{f::'a \Rightarrow 'b.\ range\ f \subseteq B\}|$
proof −
let *?F* = $\lambda\ f\ (a::'a).\ ((f\ a)::'b)$
have *bij-betw ?F* $\{f.\ range\ f \subseteq B\}$ (*Func UNIV B*)
unfolding *bij-betw-def inj-on-def*
proof *safe*
fix $h :: 'a \Rightarrow 'b$ **assume** *h*: $h \in Func\ UNIV\ B$
then obtain *f* **where** *f*: $\forall\ a.\ h\ a = f\ a$ **by** *blast*
hence $range\ f \subseteq B$ **using** *h* **unfolding** *Func-def* **by** *auto*
thus $h \in (\lambda f\ a.\ f\ a)\ `\ \{f.\ range\ f \subseteq B\}$ **using** *f* **by** *auto*
qed(*unfold Func-def fun-eq-iff*, *auto*)
then show *?thesis*
by (*blast intro*: *ordIso-symmetric card-of-ordIsoI*)

qed

lemma *Func-Times-Range*:
|Func A (B × C)| =o |Func A B × Func A C| (**is** *|?LHS| =o |?RHS|*)
proof −
let *?F = λfg. (λx. if x ∈ A then fst (fg x) else undefined,*
λx. if x ∈ A then snd (fg x) else undefined)
let *?G = λ(f, g) x. if x ∈ A then (f x, g x) else undefined*
have *bij-betw ?F ?LHS ?RHS* **unfolding** *bij-betw-def inj-on-def*
proof (*intro conjI impI ballI equalityI subsetI*)
fix *f g* **assume** ∗: *f ∈ Func A (B × C) g ∈ Func A (B × C) ?F f = ?F g*
show *f = g*
proof
fix *x* **from** ∗ **have** *fst (f x) = fst (g x) ∧ snd (f x) = snd (g x)*
by (*cases x ∈ A*) (*auto simp*: *Func-def fun-eq-iff split*: *if-splits*)
then show *f x = g x* **by** (*subst* (*1 2*) *surjective-pairing*) *simp*
qed
next
fix *fg* **assume** *fg ∈ Func A B × Func A C*
thus *fg ∈ ?F ‘ Func A (B × C)*
by (*intro image-eqI*[*of - - ?G fg*]) (*auto simp*: *Func-def*)
qed (*auto simp*: *Func-def fun-eq-iff*)
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

30.9 Regular vs. stable cardinals

definition *stable* :: *'a rel ⇒ bool*
where
stable r ≡ ∀(A::'a set) (F :: 'a ⇒ 'a set).
|A| <o r ∧ (∀a ∈ A. |F a| <o r)
⟶ |SIGMA a : A. F a| <o r

lemma *regularCard-stable*:
assumes *cr*: *Card-order r* **and** *ir*: *¬finite (Field r)* **and** *reg*: *regularCard r*
shows *stable r*
unfolding *stable-def* **proof** *safe*
fix *A :: 'a set* **and** *F :: 'a ⇒ 'a set* **assume** *A*: *|A| <o r* **and** *F*: *∀a∈A. |F a|*
<o r
{assume *r ≤o |Sigma A F|*
hence *|Field r| ≤o |Sigma A F|* **using** *card-of-Field-ordIso*[*OF cr*] *ordIso-ordLeq-trans*
by *blast*
moreover have *Fi*: *Field r ≠ {}* **using** *ir* **by** *auto*
ultimately have *∃f. f ‘ Sigma A F = Field r* **using** *card-of-ordLeq2*[*OF Fi*]
by *blast*
then obtain *f* **where** *f*: *f ‘ Sigma A F = Field r* **by** *blast*
have *r*: *wo-rel r* **using** *cr* **unfolding** *card-order-on-def wo-rel-def* **by** *auto*
{fix *a* **assume** *a*: *a ∈ A*
define *L* **where** *L = {(a,u) | u. u ∈ F a}*

have *fL*: f ‘ $L \subseteq$ *Field* r **using** $f\ a$ **unfolding** *L-def* **by** *auto*
have *bij-betw snd* $\{(a,\ u)\ |u.\ u \in F\ a\}$ $(F\ a)$
unfolding *bij-betw-def inj-on-def* **by** (*auto simp*: *image-def*)
then have $|L| =o |F\ a|$ **unfolding** *L-def card-of-ordIso*[*symmetric*] **by** *blast*
hence $|L| <o\ r$ **using** *F a ordIso-ordLess-trans*[*of* $|L|$ $|F\ a|$] **unfolding** *L-def* **by** *auto*
hence $|f$ ‘ $L| <o\ r$ **using** *ordLeq-ordLess-trans*[*OF card-of-image*, *of L*] **unfolding** *L-def* **by** *auto*
hence ¬ *cofinal* $(f$ ‘ $L)$ r **using** *reg fL* **unfolding** *regularCard-def*
by (*force simp add*: *dest*: *not-ordLess-ordIso*)
then obtain k **where** k: $k \in$ *Field* r **and** $\forall\ l \in L.\ \neg\ (f\ l \neq k \wedge (k,\ f\ l) \in r)$
unfolding *cofinal-def image-def* **by** *auto*
hence $\exists\ k \in$ *Field* $r.\ \forall\ l \in L.\ (f\ l,\ k) \in r$
using *wo-rel.in-notinI*[*OF r - - ‹k ∈ Field r›*] *fL* **unfolding** *image-subset-iff* **by** *fast*
hence $\exists\ k \in$ *Field* $r.\ \forall\ u \in F\ a.\ (f\ (a,u),\ k) \in r$ **unfolding** *L-def* **by** *auto*
}
then have x: $\bigwedge a.\ a \in A \Longrightarrow \exists\, k.\ k \in$ *Field* $r \wedge (\forall\, u \in F\ a.\ (f\ (a,\ u),\ k) \in r)$ **by** *blast*
obtain *gg* **where** $\bigwedge a.\ a \in A \Longrightarrow gg\ a = (SOME\ k.\ k \in$ *Field* $r \wedge (\forall\, u \in F\ a.\ (f\ (a,\ u),\ k) \in r))$ **by** *simp*
then have *gg*: $\forall\, a \in A.\ \forall\, u \in F\ a.\ (f\ (a,\ u),\ gg\ a) \in r$ **using** *someI-ex*[*OF x*] **by** *auto*
obtain *j0* **where** *j0*: *j0* $\in$ *Field* r **using** *Fi* **by** *auto*
define g **where** [*abs-def*]: $g\ a = (if\ F\ a \neq \{\}\ then\ gg\ a\ else\ j0)$ **for** a
have g: $\forall\ a \in A.\ \forall\ u \in F\ a.\ (f\ (a,u),g\ a) \in r$ **using** *gg* **unfolding** *g-def* **by** *auto*
hence *1*: *Field* $r \subseteq (\bigcup a \in A.\ under\ r\ (g\ a))$
using *f*[*symmetric*] **unfolding** *under-def image-def* **by** *auto*
have *gA*: g ‘ $A \subseteq$ *Field* r **using** *gg j0* **unfolding** *Field-def g-def* **by** *auto*
moreover have *cofinal* $(g$ ‘ $A)$ r **unfolding** *cofinal-def*
proof *safe*
fix i **assume** $i \in$ *Field* r
then obtain j **where** *ij*: $(i,j) \in r$ $i \neq j$ **using** *cr ir infinite-Card-order-limit* **by** *fast*
hence $j \in$ *Field* r **using** *card-order-on-def cr well-order-on-domain* **by** *fast*
then obtain a **where** a: $a \in A$ **and** j: $(j,\ g\ a) \in r$
using *1* **unfolding** *under-def* **by** *auto*
hence $(i,\ g\ a) \in r$ **using** *ij wo-rel.TRANS*[*OF r*] **unfolding** *trans-def* **by** *blast*
moreover have $i \neq g\ a$
using *ij j r* **unfolding** *wo-rel-def* **unfolding** *well-order-on-def linear-order-on-def partial-order-on-def antisym-def* **by** *auto*
ultimately show $\exists j \in g$ ‘ $A.\ i \neq j \wedge (i,\ j) \in r$ **using** a **by** *auto*
qed
ultimately have $|g$ ‘ $A| =o\ r$ **using** *reg* **unfolding** *regularCard-def* **by** *auto*
moreover have $|g$ ‘ $A| \leq o\ |A|$ **using** *card-of-image* **by** *blast*
ultimately have *False* **using** *A* **using** *not-ordLess-ordIso ordLeq-ordLess-trans* **by** *blast*

}
thus $|Sigma\ A\ F| <o\ r$
using *cr not-ordLess-iff-ordLeq* **using** *card-of-Well-order card-order-on-well-order-on*
by *blast*
qed

lemma *stable-regularCard*:
assumes *cr*: *Card-order r* **and** *ir*: ¬*finite* (*Field r*) **and** *st*: *stable r*
shows *regularCard r*
unfolding *regularCard-def* **proof** *safe*
fix *K* **assume** *K*: $K \subseteq Field\ r$ **and** *cof*: *cofinal K r*
have $|K| \leq o\ r$ **using** *K card-of-Field-ordIso card-of-mono1 cr ordLeq-ordIso-trans*
by *blast*
moreover
{assume *Kr*: $|K| <o\ r$
have *x*: $\bigwedge a.\ a \in Field\ r \Longrightarrow \exists\, b.\ b \in K \wedge a \neq b \wedge (a,\ b) \in r$ **using** *cof*
unfolding *cofinal-def* **by** *blast*
then obtain *f* **where** $\bigwedge a.\ a \in Field\ r \Longrightarrow f\ a = (SOME\ b.\ b \in K \wedge a \neq b \wedge (a,\ b) \in r)$ **by** *simp*
then have $\forall\, a{\in}Field\ r.\ f\ a \in K \wedge a \neq f\ a \wedge (a,\ f\ a) \in r$ **using** *someI-ex*[*OF x*] **by** *simp*
hence $Field\ r \subseteq (\bigcup a \in K.\ underS\ r\ a)$ **unfolding** *underS-def* **by** *auto*
hence $r \leq o\ |\bigcup a \in K.\ underS\ r\ a|$
using *cr Card-order-iff-ordLeq-card-of card-of-mono1 ordLeq-transitive* **by** *blast*
also have $|\bigcup a \in K.\ underS\ r\ a| \leq o\ |SIGMA\ a{:}\ K.\ underS\ r\ a|$ **by** (*rule card-of-UNION-Sigma*)
also
{have $\forall\ a \in K.\ |underS\ r\ a| <o\ r$ **using** *K card-of-underS*[*OF cr*] *subsetD* **by** *auto*
hence $|SIGMA\ a{:}\ K.\ underS\ r\ a| <o\ r$ **using** *st Kr* **unfolding** *stable-def* **by** *auto*
}
finally have $r <o\ r$ **.**
hence *False* **using** *ordLess-irreflexive* **by** *blast*
}
ultimately show $|K| =o\ r$ **using** *ordLeq-iff-ordLess-or-ordIso* **by** *blast*
qed

lemma *internalize-card-of-ordLess*:
$(\ |A| <o\ r) = (\exists B < Field\ r.\ |A| =o\ |B| \wedge |B| <o\ r)$
proof
assume $|A| <o\ r$
then obtain *p* **where** *1*: $Field\ p < Field\ r \wedge |A| =o\ p \wedge p <o\ r$
using *internalize-ordLess*[*of* $|A|$ *r*] **by** *blast*
hence *Card-order p* **using** *card-of-Card-order Card-order-ordIso2* **by** *blast*
hence $|Field\ p| =o\ p$ **using** *card-of-Field-ordIso* **by** *blast*
hence $|A| =o\ |Field\ p| \wedge |Field\ p| <o\ r$
using *1 ordIso-equivalence ordIso-ordLess-trans* **by** *blast*

thus $\exists B < Field\ r.\ |A| =o |B| \wedge |B| <o r$ **using** *1* **by** *blast*
next
assume $\exists B < Field\ r.\ |A| =o |B| \wedge |B| <o r$
thus $|A| <o r$ **using** *ordIso-ordLess-trans* **by** *blast*
qed

lemma *card-of-Sigma-cong1*:
assumes $\forall i \in I.\ |A\ i| =o |B\ i|$
shows $|SIGMA\ i : I.\ A\ i| =o |SIGMA\ i : I.\ B\ i|$
using *assms* **by** (*auto simp add*: *card-of-Sigma-mono1 ordIso-iff-ordLeq*)

lemma *card-of-Sigma-cong2*:
assumes *bij-betw f* (I::$'i$ *set*) (J::$'j$ *set*)
shows $|SIGMA\ i : I.\ (A::'j \Rightarrow 'a\ set)\ (f\ i)| =o |SIGMA\ j : J.\ A\ j|$
proof −
let *?LEFT* = $SIGMA\ i : I.\ A\ (f\ i)$
let *?RIGHT* = $SIGMA\ j : J.\ A\ j$
define *u* **where** $u \equiv \lambda(i::'i,a::'a).\ (f\ i,a)$
have *bij-betw u ?LEFT ?RIGHT*
using *assms* **unfolding** *u-def bij-betw-def inj-on-def* **by** *auto*
thus *?thesis* **using** *card-of-ordIso* **by** *blast*
qed

lemma *card-of-Sigma-cong*:
assumes *BIJ*: *bij-betw f I J* **and** *ISO*: $\forall j \in J.\ |A\ j| =o |B\ j|$
shows $|SIGMA\ i : I.\ A\ (f\ i)| =o |SIGMA\ j : J.\ B\ j|$
proof −
have $\forall i \in I.\ |A(f\ i)| =o |B(f\ i)|$
using *ISO BIJ* **unfolding** *bij-betw-def* **by** *blast*
hence $|SIGMA\ i : I.\ A\ (f\ i)| =o |SIGMA\ i : I.\ B\ (f\ i)|$ **by** (*rule card-of-Sigma-cong1*)
moreover have $|SIGMA\ i : I.\ B\ (f\ i)| =o |SIGMA\ j : J.\ B\ j|$
using *BIJ card-of-Sigma-cong2* **by** *blast*
ultimately show *?thesis* **using** *ordIso-transitive* **by** *blast*
qed

lemma *stable-elim*:
assumes *ST*: *stable r* **and** *A-LESS*: $|A| <o r$ **and**
F-LESS: $\bigwedge a.\ a \in A \Longrightarrow |F\ a| <o r$
shows $|SIGMA\ a : A.\ F\ a| <o r$
proof −
obtain A' **where** *1*: $A' < Field\ r \wedge |A'| <o r$ **and** *2*: $|A| =o |A'|$
using *internalize-card-of-ordLess*[*of A r*] *A-LESS* **by** *blast*
then obtain *G* **where** *3*: *bij-betw G A′ A*
using *card-of-ordIso ordIso-symmetric* **by** *blast*

{**fix** *a* **assume** $a \in A$
hence $\exists B'.\ B' \leq Field\ r \wedge |F\ a| =o |B'| \wedge |B'| <o r$
using *internalize-card-of-ordLess*[*of F a r*] *F-LESS* **by** *blast*

```
  }
  then obtain F' where
    temp: ∀ a ∈ A. F' a ≤ Field r ∧ |F a| =o |F' a| ∧ |F' a| <o r
    using bchoice[of A λ a B'. B' ≤ Field r ∧ |F a| =o |B'| ∧ |B'| <o r] by blast
  hence 4: ∀ a ∈ A. F' a ≤ Field r ∧ |F' a| <o r by auto
  have 5: ∀ a ∈ A. |F' a| =o |F a| using temp ordIso-symmetric by auto

  have ∀ a' ∈ A'. F'(G a') ≤ Field r ∧ |F'(G a')| <o r
    using 3 4 bij-betw-ball[of G A' A] by auto
  hence |SIGMA a' : A'. F'(G a')| <o r
    using ST 1 unfolding stable-def by auto
  moreover have |SIGMA a' : A'. F'(G a')| =o |SIGMA a : A. F a|
    using card-of-Sigma-cong[of G A' A F' F] 5 3 by blast
  ultimately show ?thesis using ordIso-symmetric ordIso-ordLess-trans by blast
qed

lemma stable-natLeq: stable natLeq
proof(unfold stable-def, safe)
  fix A :: 'a set and F :: 'a ⇒ 'a set
  assume |A| <o natLeq and ∀ a∈A. |F a| <o natLeq
  hence finite A ∧ (∀ a ∈ A. finite(F a))
    by (auto simp add: finite-iff-ordLess-natLeq)
  hence finite(Sigma A F) by (simp only: finite-SigmaI[of A F])
  thus |Sigma A F | <o natLeq
    by (auto simp add: finite-iff-ordLess-natLeq)
qed

corollary regularCard-natLeq: regularCard natLeq
  using stable-regularCard[OF natLeq-Card-order - stable-natLeq] Field-natLeq by
simp

lemma stable-ordIso1:
  assumes ST: stable r and ISO: r' =o r
  shows stable r'
proof(unfold stable-def, auto)
  fix A::'b set and F::'b ⇒ 'b set
  assume |A| <o r' and ∀ a ∈ A. |F a| <o r'
  hence ( |A| <o r) ∧ (∀ a ∈ A. |F a| <o r)
    using ISO ordLess-ordIso-trans by blast
  hence |SIGMA a : A. F a| <o r using assms stable-elim by blast
  thus |SIGMA a : A. F a| <o r'
    using ISO ordIso-symmetric ordLess-ordIso-trans by blast
qed

lemma stable-UNION:
  assumes stable r and |A| <o r and ⋀ a. a ∈ A ⟹ |F a| <o r
  shows |⋃ a ∈ A. F a| <o r
  using assms card-of-UNION-Sigma stable-elim ordLeq-ordLess-trans by blast
```

corollary *card-of-UNION-ordLess-infinite*:
assumes *stable* $|B|$ **and** $|I| <o |B|$ **and** $\forall i \in I.\ |A\ i| <o |B|$
shows $|\bigcup i \in I.\ A\ i| <o |B|$
using *assms stable-UNION* **by** *blast*

corollary *card-of-UNION-ordLess-infinite-Field*:
assumes *ST*: *stable r* **and** *r*: *Card-order r* **and**
LEQ-I: $|I| <o r$ **and** *LEQ*: $\forall i \in I.\ |A\ i| <o r$
shows $|\bigcup i \in I.\ A\ i| <o r$
proof −
let *?B* = *Field r*
have *1*: $r =o |?B| \wedge |?B| =o r$ **using** *r card-of-Field-ordIso*
ordIso-symmetric **by** *blast*
hence $|I| <o |?B|\ \ \forall i \in I.\ |A\ i| <o |?B|$
using *LEQ-I LEQ ordLess-ordIso-trans* **by** *blast+*
moreover have *stable* $|?B|$ **using** *stable-ordIso1 ST 1* **by** *blast*
ultimately have $|\bigcup i \in I.\ A\ i| <o |?B|$ **using** *LEQ*
card-of-UNION-ordLess-infinite **by** *blast*
thus *?thesis* **using** *1 ordLess-ordIso-trans* **by** *blast*
qed

end

31 Cardinal Arithmetic as Needed by Bounded Natural Functors

theory *BNF-Cardinal-Arithmetic*
imports *BNF-Cardinal-Order-Relation*
begin

lemma *dir-image*: $[\![\bigwedge x\ y.\ (f\ x = f\ y) = (x = y);$ *Card-order r*$]\!] \Longrightarrow r =o$ *dir-image r f*
by (*rule dir-image-ordIso*) (*auto simp add*: *inj-on-def card-order-on-def*)

lemma *card-order-dir-image*:
assumes *bij*: *bij f* **and** *co*: *card-order r*
shows *card-order* (*dir-image r f*)
proof −
have *Field* (*dir-image r f*) = *UNIV*
using *assms card-order-on-Card-order*[*of UNIV r*]
unfolding *bij-def dir-image-Field* **by** *auto*
moreover from *bij* **have** $\bigwedge x\ y.\ (f\ x = f\ y) = (x = y)$
unfolding *bij-def inj-on-def* **by** *auto*
with *co* **have** *Card-order* (*dir-image r f*)
using *card-order-on-Card-order Card-order-ordIso2*[*OF - dir-image*] **by** *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *ordIso-refl*: *Card-order r* $\Longrightarrow$ *r* $=o$ *r*
by (*rule card-order-on-ordIso*)

lemma *ordLeq-refl*: *Card-order r* $\Longrightarrow$ *r* $\leq o$ *r*
by (*rule ordIso-imp-ordLeq*, *rule card-order-on-ordIso*)

lemma *card-of-ordIso-subst*: *A* = *B* $\Longrightarrow$ |*A*| $=o$ |*B*|
by (*simp only*: *ordIso-refl card-of-Card-order*)

lemma *Field-card-order*: *card-order r* $\Longrightarrow$ *Field r* = *UNIV*
using *card-order-on-Card-order*[*of UNIV r*] **by** *simp*

31.1 Zero

definition *czero* **where**
czero = *card-of* {}

lemma *czero-ordIso*: *czero* $=o$ *czero*
using *card-of-empty-ordIso* **by** (*simp add*: *czero-def*)

lemma *card-of-ordIso-czero-iff-empty*:
|*A*| $=o$ (*czero* :: *'b rel*) $\longleftrightarrow$ *A* = ({} :: *'a set*)
unfolding *czero-def* **by** (*rule iffI*[*OF card-of-empty2*]) (*auto simp*: *card-of-refl card-of-empty-ordIso*)

abbreviation *Cnotzero* **where**
Cnotzero (*r* :: *'a rel*) $\equiv \neg$(*r* $=o$ (*czero* :: *'a rel*)) $\wedge$ *Card-order r*

lemma *Cnotzero-imp-not-empty*: *Cnotzero r* $\Longrightarrow$ *Field r* $\neq$ {}
unfolding *Card-order-iff-ordIso-card-of czero-def* **by** *force*

lemma *czeroI*:
⟦*Card-order r*; *Field r* = {}⟧ $\Longrightarrow$ *r* $=o$ *czero*
using *Cnotzero-imp-not-empty ordIso-transitive*[*OF - czero-ordIso*] **by** *blast*

lemma *czeroE*:
r $=o$ *czero* $\Longrightarrow$ *Field r* = {}
unfolding *czero-def*
by (*drule card-of-cong*) (*simp only*: *Field-card-of card-of-empty2*)

lemma *Cnotzero-mono*:
⟦*Cnotzero r*; *Card-order q*; *r* $\leq o$ *q*⟧ $\Longrightarrow$ *Cnotzero q*
by (*force intro*: *czeroI dest*: *card-of-mono2 card-of-empty3 czeroE*)

31.2 (In)finite cardinals

definition *cinfinite* **where**
cinfinite r $\equiv$ ($\neg$ *finite* (*Field r*))

abbreviation *Cinfinite* **where**
Cinfinite r ≡ *cinfinite r* ∧ *Card-order r*

definition *cfinite* **where**
cfinite r = *finite* (*Field r*)

abbreviation *Cfinite* **where**
Cfinite r ≡ *cfinite r* ∧ *Card-order r*

lemma *Cfinite-ordLess-Cinfinite*: ⟦*Cfinite r*; *Cinfinite s*⟧ ⟹ *r* <*o s*
unfolding *cfinite-def cinfinite-def*
by (*blast intro*: *finite-ordLess-infinite card-order-on-well-order-on*)

lemmas *natLeq-card-order* = *natLeq-Card-order*[*unfolded Field-natLeq*]

lemma *natLeq-cinfinite*: *cinfinite natLeq*
unfolding *cinfinite-def Field-natLeq* **by** (*rule infinite-UNIV-nat*)

lemma *natLeq-Cinfinite*: *Cinfinite natLeq*
using *natLeq-cinfinite natLeq-Card-order* **by** *simp*

lemma *natLeq-ordLeq-cinfinite*:
assumes *inf*: *Cinfinite r*
shows *natLeq* ≤*o r*
proof −
from *inf* **have** *natLeq* ≤*o* |*Field r*| **unfolding** *cinfinite-def*
using *infinite-iff-natLeq-ordLeq* **by** *blast*
also from *inf* **have** |*Field r*| =*o r* **by** (*simp add*: *card-of-unique ordIso-symmetric*)
finally show *?thesis* **.**
qed

lemma *cinfinite-not-czero*: *cinfinite r* ⟹ ¬ (*r* =*o* (*czero* :: *'a rel*))
unfolding *cinfinite-def* **by** (*cases Field r* = {}) (*auto dest*: *czeroE*)

lemma *Cinfinite-Cnotzero*: *Cinfinite r* ⟹ *Cnotzero r*
using *cinfinite-not-czero* **by** *auto*

lemma *Cinfinite-cong*: ⟦*r1* =*o r2*; *Cinfinite r1*⟧ ⟹ *Cinfinite r2*
using *Card-order-ordIso2*[*of r1 r2*] **unfolding** *cinfinite-def ordIso-iff-ordLeq*
by (*auto dest*: *card-of-ordLeq-infinite*[*OF card-of-mono2*])

lemma *cinfinite-mono*: ⟦*r1* ≤*o r2*; *cinfinite r1*⟧ ⟹ *cinfinite r2*
unfolding *cinfinite-def* **by** (*auto dest*: *card-of-ordLeq-infinite*[*OF card-of-mono2*])

lemma *regularCard-ordIso*:
assumes *k* =*o k'* **and** *Cinfinite k* **and** *regularCard k*
shows *regularCard k'*
proof−

have *stable k* **using** *assms cinfinite-def regularCard-stable* **by** *blast*
hence *stable k'* **using** *assms stable-ordIso1 ordIso-symmetric* **by** *blast*
thus *?thesis* **using** *assms cinfinite-def stable-regularCard Cinfinite-cong* **by** *blast*
qed

corollary *card-of-UNION-ordLess-infinite-Field-regularCard*:
assumes *regularCard r* **and** *Cinfinite r* **and** $|I| <o r$ **and** $\forall i \in I.\ |A\ i| <o r$
shows $|\bigcup i \in I.\ A\ i| <o r$
using *card-of-UNION-ordLess-infinite-Field regularCard-stable assms cinfinite-def*
by *blast*

31.3 Binary sum

definition *csum* (**infixr** ‹+c› *65*)
where *r1 +c r2* ≡ |*Field r1* <+> *Field r2*|

lemma *Field-csum*: *Field* (*r +c s*) = *Inl* ‘ *Field r* ∪ *Inr* ‘ *Field s*
unfolding *csum-def Field-card-of* **by** *auto*

lemma *Card-order-csum*: *Card-order* (*r1 +c r2*)
unfolding *csum-def* **by** (*simp add*: *card-of-Card-order*)

lemma *csum-Cnotzero1*: *Cnotzero r1* ⟹ *Cnotzero* (*r1 +c r2*)
using *Cnotzero-imp-not-empty*
by (*auto intro*: *card-of-Card-order simp*: *csum-def card-of-ordIso-czero-iff-empty*)

lemma *card-order-csum*:
assumes *card-order r1 card-order r2*
shows *card-order* (*r1 +c r2*)
proof −
have *Field r1 = UNIV Field r2 = UNIV* **using** *assms card-order-on-Card-order*
by *auto*
thus *?thesis* **unfolding** *csum-def* **by** (*auto simp*: *card-of-card-order-on*)
qed

lemma *cinfinite-csum*:
cinfinite r1 ∨ *cinfinite r2* ⟹ *cinfinite* (*r1 +c r2*)
unfolding *cinfinite-def csum-def* **by** (*auto simp*: *Field-card-of*)

lemma *Cinfinite-csum*:
Cinfinite r1 ∨ *Cinfinite r2* ⟹ *Cinfinite* (*r1 +c r2*)
using *card-of-Card-order*
by (*auto simp*: *cinfinite-def csum-def Field-card-of*)

lemma *Cinfinite-csum1*: *Cinfinite r1* ⟹ *Cinfinite* (*r1 +c r2*)
by (*blast intro*!: *Cinfinite-csum elim*:)

lemma *Cinfinite-csum-weak*:
⟦*Cinfinite r1*; *Cinfinite r2*⟧ ⟹ *Cinfinite* (*r1 +c r2*)

by (*erule Cinfinite-csum1*)

lemma *csum-cong*: ⟦*p1* =*o r1*; *p2* =*o r2*⟧ ⟹ *p1* +*c p2* =*o r1* +*c r2*
by (*simp only*: *csum-def ordIso-Plus-cong*)

lemma *csum-cong1*: *p1* =*o r1* ⟹ *p1* +*c q* =*o r1* +*c q*
by (*simp only*: *csum-def ordIso-Plus-cong1*)

lemma *csum-cong2*: *p2* =*o r2* ⟹ *q* +*c p2* =*o q* +*c r2*
by (*simp only*: *csum-def ordIso-Plus-cong2*)

lemma *csum-mono*: ⟦*p1* ≤*o r1*; *p2* ≤*o r2*⟧ ⟹ *p1* +*c p2* ≤*o r1* +*c r2*
by (*simp only*: *csum-def ordLeq-Plus-mono*)

lemma *csum-mono1*: *p1* ≤*o r1* ⟹ *p1* +*c q* ≤*o r1* +*c q*
by (*simp only*: *csum-def ordLeq-Plus-mono1*)

lemma *csum-mono2*: *p2* ≤*o r2* ⟹ *q* +*c p2* ≤*o q* +*c r2*
by (*simp only*: *csum-def ordLeq-Plus-mono2*)

lemma *ordLeq-csum1*: *Card-order p1* ⟹ *p1* ≤*o p1* +*c p2*
by (*simp only*: *csum-def Card-order-Plus1*)

lemma *ordLeq-csum2*: *Card-order p2* ⟹ *p2* ≤*o p1* +*c p2*
by (*simp only*: *csum-def Card-order-Plus2*)

lemma *csum-com*: *p1* +*c p2* =*o p2* +*c p1*
by (*simp only*: *csum-def card-of-Plus-commute*)

lemma *csum-assoc*: (*p1* +*c p2*) +*c p3* =*o p1* +*c p2* +*c p3*
by (*simp only*: *csum-def Field-card-of card-of-Plus-assoc*)

lemma *Cfinite-csum*: ⟦*Cfinite r*; *Cfinite s*⟧ ⟹ *Cfinite* (*r* +*c s*)
unfolding *cfinite-def csum-def Field-card-of* **using** *card-of-card-order-on* **by** *simp*

lemma *csum-csum*: (*r1* +*c r2*) +*c* (*r3* +*c r4*) =*o* (*r1* +*c r3*) +*c* (*r2* +*c r4*)
proof −
have (*r1* +*c r2*) +*c* (*r3* +*c r4*) =*o r1* +*c r2* +*c* (*r3* +*c r4*)
by (*rule csum-assoc*)
also have *r1* +*c r2* +*c* (*r3* +*c r4*) =*o r1* +*c* (*r2* +*c r3*) +*c r4*
by (*intro csum-assoc csum-cong2 ordIso-symmetric*)
also have *r1* +*c* (*r2* +*c r3*) +*c r4* =*o r1* +*c* (*r3* +*c r2*) +*c r4*
by (*intro csum-com csum-cong1 csum-cong2*)
also have *r1* +*c* (*r3* +*c r2*) +*c r4* =*o r1* +*c r3* +*c r2* +*c r4*
by (*intro csum-assoc csum-cong2 ordIso-symmetric*)
also have *r1* +*c r3* +*c r2* +*c r4* =*o* (*r1* +*c r3*) +*c* (*r2* +*c r4*)
by (*intro csum-assoc ordIso-symmetric*)
finally show *?thesis* **.**

qed

lemma *Plus-csum*: $|A <+> B| =o |A| +c |B|$
by (*simp only*: *csum-def Field-card-of card-of-refl*)

lemma *Un-csum*: $|A \cup B| \leq o |A| +c |B|$
using *ordLeq-ordIso-trans*[*OF card-of-Un-Plus-ordLeq Plus-csum*] **by** *blast*

31.4 One

definition *cone* **where**
$cone = card\text{-}of\ \{()\}$

lemma *Card-order-cone*: *Card-order cone*
unfolding *cone-def* **by** (*rule card-of-Card-order*)

lemma *Cfinite-cone*: *Cfinite cone*
unfolding *cfinite-def* **by** (*simp add*: *Card-order-cone*)

lemma *cone-not-czero*: $\neg\ (cone =o czero)$
unfolding *czero-def cone-def ordIso-iff-ordLeq*
using *card-of-empty3 empty-not-insert* **by** *blast*

lemma *cone-ordLeq-Cnotzero*: $Cnotzero\ r \Longrightarrow cone \leq o\ r$
unfolding *cone-def* **by** (*rule Card-order-singl-ordLeq*) (*auto intro*: *czeroI*)

31.5 Two

definition *ctwo* **where**
$ctwo = |UNIV :: bool\ set|$

lemma *Card-order-ctwo*: *Card-order ctwo*
unfolding *ctwo-def* **by** (*rule card-of-Card-order*)

lemma *ctwo-not-czero*: $\neg\ (ctwo =o czero)$
using *card-of-empty3*[*of UNIV :: bool set*] *ordIso-iff-ordLeq*
unfolding *czero-def ctwo-def* **using** *UNIV-not-empty* **by** *auto*

lemma *ctwo-Cnotzero*: *Cnotzero ctwo*
by (*simp add*: *ctwo-not-czero Card-order-ctwo*)

31.6 Family sum

definition *Csum* **where**
$Csum\ r\ rs \equiv |SIGMA\ i : Field\ r.\ Field\ (rs\ i)|$

syntax -*Csum* ::
pttrn => ('*a* * '*a*) *set* => '*b* * '*b set* => (('*a* * '*b*) * ('*a* * '*b*)) *set*
(‹(‹*indent=3 notation=‹binder CSUM›› CSUM* -:-. -)› [*0*, *51*, *10*] *10*)

syntax-consts
-Csum == *Csum*

translations
CSUM i:r. rs == *CONST Csum r* (%*i. rs*)

lemma *SIGMA-CSUM*: |*SIGMA i* : *I. As i*| = (*CSUM i* : |*I*|. |*As i*|)
by (*auto simp*: *Csum-def Field-card-of*)

31.7 Product

definition *cprod* (**infixr** ‹∗*c*› *80*) **where**
r1 ∗*c r2* = |*Field r1* × *Field r2*|

lemma *card-order-cprod*:
assumes *card-order r1 card-order r2*
shows *card-order* (*r1* ∗*c r2*)
proof −
have *Field r1* = *UNIV Field r2* = *UNIV*
using *assms card-order-on-Card-order* **by** *auto*
thus *?thesis* **by** (*auto simp*: *cprod-def card-of-card-order-on*)
qed

lemma *Card-order-cprod*: *Card-order* (*r1* ∗*c r2*)
by (*simp only*: *cprod-def Field-card-of card-of-card-order-on*)

lemma *cprod-mono1*: *p1* ≤*o r1* ⟹ *p1* ∗*c q* ≤*o r1* ∗*c q*
by (*simp only*: *cprod-def ordLeq-Times-mono1*)

lemma *cprod-mono2*: *p2* ≤*o r2* ⟹ *q* ∗*c p2* ≤*o q* ∗*c r2*
by (*simp only*: *cprod-def ordLeq-Times-mono2*)

lemma *cprod-mono*: ⟦*p1* ≤*o r1*; *p2* ≤*o r2*⟧ ⟹ *p1* ∗*c p2* ≤*o r1* ∗*c r2*
by (*rule ordLeq-transitive*[*OF cprod-mono1 cprod-mono2*])

lemma *ordLeq-cprod2*: ⟦*Cnotzero p1*; *Card-order p2*⟧ ⟹ *p2* ≤*o p1* ∗*c p2*
unfolding *cprod-def* **by** (*rule Card-order-Times2*) (*auto intro*: *czeroI*)

lemma *cinfinite-cprod*: ⟦*cinfinite r1*; *cinfinite r2*⟧ ⟹ *cinfinite* (*r1* ∗*c r2*)
by (*simp add*: *cinfinite-def cprod-def Field-card-of infinite-cartesian-product*)

lemma *cinfinite-cprod2*: ⟦*Cnotzero r1*; *Cinfinite r2*⟧ ⟹ *cinfinite* (*r1* ∗*c r2*)
by (*rule cinfinite-mono*) (*auto intro*: *ordLeq-cprod2*)

lemma *Cinfinite-cprod2*: ⟦*Cnotzero r1*; *Cinfinite r2*⟧ ⟹ *Cinfinite* (*r1* ∗*c r2*)
by (*blast intro*: *cinfinite-cprod2 Card-order-cprod*)

lemma *cprod-cong*: ⟦*p1* =*o r1*; *p2* =*o r2*⟧ ⟹ *p1* ∗*c p2* =*o r1* ∗*c r2*

unfolding *ordIso-iff-ordLeq* **by** (*blast intro*: *cprod-mono*)

lemma *cprod-cong1*: ⟦*p1* =*o* *r1*⟧ ⟹ *p1* ∗*c* *p2* =*o* *r1* ∗*c* *p2*
unfolding *ordIso-iff-ordLeq* **by** (*blast intro*: *cprod-mono1*)

lemma *cprod-cong2*: *p2* =*o* *r2* ⟹ *q* ∗*c* *p2* =*o* *q* ∗*c* *r2*
unfolding *ordIso-iff-ordLeq* **by** (*blast intro*: *cprod-mono2*)

lemma *cprod-com*: *p1* ∗*c* *p2* =*o* *p2* ∗*c* *p1*
by (*simp only*: *cprod-def card-of-Times-commute*)

lemma *card-of-Csum-Times*:
∀ *i* ∈ *I*. |*A* *i*| ≤*o* |*B*| ⟹ (*CSUM* *i* : |*I*|. |*A* *i*|) ≤*o* |*I*| ∗*c* |*B*|
by (*simp only*: *Csum-def cprod-def Field-card-of card-of-Sigma-mono1*)

lemma *card-of-Csum-Times'*:
assumes *Card-order* *r* ∀ *i* ∈ *I*. |*A* *i*| ≤*o* *r*
shows (*CSUM* *i* : |*I*|. |*A* *i*|) ≤*o* |*I*| ∗*c* *r*
proof −
from *assms*(*1*) **have** ∗: *r* =*o* |*Field* *r*| **by** (*simp add*: *card-of-unique*)
with *assms*(*2*) **have** ∀ *i* ∈ *I*. |*A* *i*| ≤*o* |*Field* *r*| **by** (*blast intro*: *ordLeq-ordIso-trans*)
hence (*CSUM* *i* : |*I*|. |*A* *i*|) ≤*o* |*I*| ∗*c* |*Field* *r*| **by** (*simp only*: *card-of-Csum-Times*)
also from ∗ **have** |*I*| ∗*c* |*Field* *r*| ≤*o* |*I*| ∗*c* *r*
by (*simp only*: *Field-card-of card-of-refl cprod-def ordIso-imp-ordLeq*)
finally show *?thesis* **.**
qed

lemma *cprod-csum-distrib1*: *r1* ∗*c* *r2* +*c* *r1* ∗*c* *r3* =*o* *r1* ∗*c* (*r2* +*c* *r3*)
unfolding *csum-def cprod-def* **by** (*simp add*: *Field-card-of card-of-Times-Plus-distrib ordIso-symmetric*)

lemma *csum-absorb2'*: ⟦*Card-order* *r2*; *r1* ≤*o* *r2*; *cinfinite* *r1* ∨ *cinfinite* *r2*⟧ ⟹ *r1* +*c* *r2* =*o* *r2*
unfolding *csum-def*
using *Card-order-Plus-infinite*
by (*fastforce simp*: *cinfinite-def dest*: *cinfinite-mono*)

lemma *csum-absorb1'*:
assumes *card*: *Card-order* *r2*
and *r12*: *r1* ≤*o* *r2* **and** *cr12*: *cinfinite* *r1* ∨ *cinfinite* *r2*
shows *r2* +*c* *r1* =*o* *r2*
proof −
have *r1* +*c* *r2* =*o* *r2*
by (*simp add*: *csum-absorb2'* *assms*)
then show *?thesis*
by (*blast intro*: *ordIso-transitive csum-com*)
qed

lemma *csum-absorb1*: ⟦*Cinfinite* *r2*; *r1* ≤*o* *r2*⟧ ⟹ *r2* +*c* *r1* =*o* *r2*

by (*rule csum-absorb1'*) *auto*

lemma *csum-absorb2*: ⟦*Cinfinite r2* ; *r1* ≤*o r2*⟧ ⟹ *r1* +*c r2* =*o r2*
using *ordIso-transitive csum-com csum-absorb1* **by** *blast*

lemma *regularCard-csum*:
assumes *Cinfinite r Cinfinite s regularCard r regularCard s*
shows *regularCard* (*r* +*c s*)
proof (*cases r* ≤*o s*)
case *True*
then show *?thesis* **using** *regularCard-ordIso*[*of s*] *csum-absorb2'*[*THEN ordIso-symmetric*] *assms* **by** *auto*
next
case *False*
have *Well-order s Well-order r* **using** *assms card-order-on-well-order-on* **by** *auto*
then have *s* ≤*o r* **using** *not-ordLeq-iff-ordLess False ordLess-imp-ordLeq* **by** *auto*
then show *?thesis* **using** *regularCard-ordIso*[*of r*] *csum-absorb1'*[*THEN ordIso-symmetric*] *assms* **by** *auto*
qed

lemma *csum-mono-strict*:
assumes *Card-order*: *Card-order r Card-order q*
and *Cinfinite*: *Cinfinite r' Cinfinite q'*
and *less*: *r* <*o r' q* <*o q'*
shows *r* +*c q* <*o r'* +*c q'*
proof −
have *Well-order*: *Well-order r Well-order q Well-order r' Well-order q'*
using *card-order-on-well-order-on Card-order Cinfinite* **by** *auto*
show *?thesis*
proof (*cases Cinfinite r*)
case *outer*: *True*
then show *?thesis*
proof (*cases Cinfinite q*)
case *inner*: *True*
then show *?thesis*
proof (*cases r* ≤*o q*)
case *True*
then have *r* +*c q* =*o q* **using** *csum-absorb2 inner* **by** *blast*
then show *?thesis*
using *ordIso-ordLess-trans ordLess-ordLeq-trans less Cinfinite ordLeq-csum2*
by *blast*
next
case *False*
then have *q* ≤*o r* **using** *not-ordLeq-iff-ordLess Well-order ordLess-imp-ordLeq*
by *blast*
then have *r* +*c q* =*o r* **using** *csum-absorb1 outer* **by** *blast*
then show *?thesis*
using *ordIso-ordLess-trans ordLess-ordLeq-trans less Cinfinite ordLeq-csum1*

by *blast*
qed
next
case *False*
then have *Cfinite q* **using** *Card-order cinfinite-def cfinite-def* **by** *blast*
then have *q* $\leq o$ *r* **using** *finite-ordLess-infinite cfinite-def cinfinite-def outer Well-order ordLess-imp-ordLeq* **by** *blast*
then have *r* $+c$ *q* $=o$ *r* **by** (*rule csum-absorb1*[*OF outer*])
then show *?thesis* **using** *ordIso-ordLess-trans ordLess-ordLeq-trans less ordLeq-csum1 Cinfinite* **by** *blast*
qed
next
case *False*
then have *outer*: *Cfinite r* **using** *Card-order cinfinite-def cfinite-def* **by** *blast*
then show *?thesis*
proof (*cases Cinfinite q*)
case *True*
then have *r* $\leq o$ *q* **using** *finite-ordLess-infinite cinfinite-def cfinite-def outer Well-order ordLess-imp-ordLeq* **by** *blast*
then have *r* $+c$ *q* $=o$ *q* **by** (*rule csum-absorb2*[*OF True*])
then show *?thesis* **using** *ordIso-ordLess-trans ordLess-ordLeq-trans less ordLeq-csum2 Cinfinite* **by** *blast*
next
case *False*
then have *Cfinite q* **using** *Card-order cinfinite-def cfinite-def* **by** *blast*
then have *Cfinite* (*r* $+c$ *q*) **using** *Cfinite-csum outer* **by** *blast*
moreover have *Cinfinite* (*r*′ $+c$ *q*′) **using** *Cinfinite-csum1 Cinfinite* **by** *blast*
ultimately show *?thesis* **using** *Cfinite-ordLess-Cinfinite* **by** *blast*
qed
qed
qed

31.8 Exponentiation

definition *cexp* (**infixr** ‹$\hat{}c$› *90*) **where**
r1 $\hat{}c$ *r2* $\equiv$ |*Func* (*Field r2*) (*Field r1*)|

lemma *Card-order-cexp*: *Card-order* (*r1* $\hat{}c$ *r2*)
unfolding *cexp-def* **by** (*rule card-of-Card-order*)

lemma *cexp-mono*′:
assumes *1*: *p1* $\leq o$ *r1* **and** *2*: *p2* $\leq o$ *r2*
and *n*: *Field p2* $= \{\} \Longrightarrow$ *Field r2* $= \{\}$
shows *p1* $\hat{}c$ *p2* $\leq o$ *r1* $\hat{}c$ *r2*
proof(*cases Field p1* $= \{\}$)
case *True*
hence *Field p2* $\neq \{\} \Longrightarrow$ *Func* (*Field p2*) $\{\} = \{\}$ **unfolding** *Func-is-emp* **by** *simp*

```
  with True have |Field |Func (Field p2) (Field p1)|| ≤o cone
    unfolding cone-def Field-card-of
    by (cases Field p2 = {}, auto intro: surj-imp-ordLeq simp: Func-empty)
  hence |Func (Field p2) (Field p1)| ≤o cone by (simp add: Field-card-of cexp-def)
  hence p1 ^c p2 ≤o cone unfolding cexp-def .
  thus ?thesis
  proof (cases Field p2 = {})
    case True
    with n have Field r2 = {} .
    hence cone ≤o r1 ^c r2 unfolding cone-def cexp-def Func-def
      by (auto intro: card-of-ordLeqI[where f=λ- -. undefined])
    thus ?thesis using ‹p1 ^c p2 ≤o cone› ordLeq-transitive by auto
  next
    case False with True have |Field (p1 ^c p2)| =o czero
      unfolding card-of-ordIso-czero-iff-empty cexp-def Field-card-of Func-def by
auto
    thus ?thesis unfolding cexp-def card-of-ordIso-czero-iff-empty Field-card-of
      by (simp add: card-of-empty)
  qed
next
  case False
  have 1: |Field p1| ≤o |Field r1| and 2: |Field p2| ≤o |Field r2|
    using 1 2 by (auto simp: card-of-mono2)
  obtain f1 where f1: f1 ‘ Field r1 = Field p1
    using 1 unfolding card-of-ordLeq2[OF False, symmetric] by auto
  obtain f2 where f2: inj-on f2 (Field p2) f2 ‘ Field p2 ⊆ Field r2
    using 2 unfolding card-of-ordLeq[symmetric] by blast
  have 0: Func-map (Field p2) f1 f2 ‘ (Field (r1 ^c r2)) = Field (p1 ^c p2)
    unfolding cexp-def Field-card-of using Func-map-surj[OF f1 f2 n, symmetric]
.
  have 00: Field (p1 ^c p2) ≠ {} unfolding cexp-def Field-card-of Func-is-emp
    using False by simp
  show ?thesis
    using 0 card-of-ordLeq2[OF 00] unfolding cexp-def Field-card-of by blast
qed

lemma cexp-mono:
  assumes 1: p1 ≤o r1 and 2: p2 ≤o r2
    and n: p2 =o czero ⟹ r2 =o czero and card: Card-order p2
  shows p1 ^c p2 ≤o r1 ^c r2
  by (rule cexp-mono'[OF 1 2 czeroE[OF n[OF czeroI[OF card]]]])

lemma cexp-mono1:
  assumes 1: p1 ≤o r1 and q: Card-order q
  shows p1 ^c q ≤o r1 ^c q
  using ordLeq-refl[OF q] by (rule cexp-mono[OF 1]) (auto simp: q)

lemma cexp-mono2':
  assumes 2: p2 ≤o r2 and q: Card-order q
```

and *n*: *Field p2* = {} ⟹ *Field r2* = {}
shows *q* ^*c p2* ≤*o q* ^*c r2*
using *ordLeq-refl*[*OF q*] **by** (*rule cexp-mono'*[*OF - 2 n*]) *auto*

lemma *cexp-mono2*:
assumes *2*: *p2* ≤*o r2* **and** *q*: *Card-order q*
and *n*: *p2* =*o czero* ⟹ *r2* =*o czero* **and** *card*: *Card-order p2*
shows *q* ^*c p2* ≤*o q* ^*c r2*
using *ordLeq-refl*[*OF q*] **by** (*rule cexp-mono*[*OF - 2 n card*]) *auto*

lemma *cexp-mono2-Cnotzero*:
assumes *p2* ≤*o r2 Card-order q Cnotzero p2*
shows *q* ^*c p2* ≤*o q* ^*c r2*
using *assms*(*3*) *czeroI* **by** (*blast intro*: *cexp-mono2'*[*OF assms*(*1*,*2*)])

lemma *cexp-cong*:
assumes *1*: *p1* =*o r1* **and** *2*: *p2* =*o r2*
and *Cr*: *Card-order r2*
and *Cp*: *Card-order p2*
shows *p1* ^*c p2* =*o r1* ^*c r2*
proof −
obtain *f* **where** *bij-betw f* (*Field p2*) (*Field r2*)
using *2 card-of-ordIso*[*of Field p2 Field r2*] *card-of-cong* **by** *auto*
hence *0*: *Field p2* = {} ⟷ *Field r2* = {} **unfolding** *bij-betw-def* **by** *auto*
have *r*: *p2* =*o czero* ⟹ *r2* =*o czero*
and *p*: *r2* =*o czero* ⟹ *p2* =*o czero*
using *0 Cr Cp czeroE czeroI* **by** *auto*
show *?thesis* **using** *0 1 2* **unfolding** *ordIso-iff-ordLeq*
using *r p cexp-mono*[*OF - - - Cp*] *cexp-mono*[*OF - - - Cr*] **by** *blast*
qed

lemma *cexp-cong1*:
assumes *1*: *p1* =*o r1* **and** *q*: *Card-order q*
shows *p1* ^*c q* =*o r1* ^*c q*
by (*rule cexp-cong*[*OF 1 - q q*]) (*rule ordIso-refl*[*OF q*])

lemma *cexp-cong2*:
assumes *2*: *p2* =*o r2* **and** *q*: *Card-order q* **and** *p*: *Card-order p2*
shows *q* ^*c p2* =*o q* ^*c r2*
by (*rule cexp-cong*[*OF - 2*]) (*auto simp only*: *ordIso-refl Card-order-ordIso2*[*OF p 2*] *q p*)

lemma *cexp-cone*:
assumes *Card-order r*
shows *r* ^*c cone* =*o r*
proof −
have *r* ^*c cone* =*o* |*Field r*|
unfolding *cexp-def cone-def Field-card-of Func-empty*
card-of-ordIso[*symmetric*] *bij-betw-def Func-def inj-on-def image-def*

by (*rule exI*[*of* - $\lambda f.\ f\ ()$]) *auto*
also have $|Field\ r| =o\ r$ **by** (*rule card-of-Field-ordIso*[*OF assms*])
finally show *?thesis* **.**
qed

lemma *cexp-cprod*:
assumes *r1*: *Card-order r1*
shows $(r1\ ^c\ r2)\ ^c\ r3 =o\ r1\ ^c\ (r2 *c\ r3)$ (**is** $?L =o\ ?R$)
proof −
have $?L =o\ r1\ ^c\ (r3 *c\ r2)$
unfolding *cprod-def cexp-def Field-card-of*
using *card-of-Func-Times* **by**(*rule ordIso-symmetric*)
also have $r1\ ^c\ (r3 *c\ r2) =o\ ?R$
using *cprod-com r1* **by** (*intro cexp-cong2*, *auto simp*: *Card-order-cprod*)
finally show *?thesis* **.**
qed

lemma *cprod-infinite1'*: $[\![Cinfinite\ r;\ Cnotzero\ p;\ p \leq o\ r]\!] \Longrightarrow r *c\ p =o\ r$
unfolding *cinfinite-def cprod-def*
by (*rule Card-order-Times-infinite*[*THEN conjunct1*]) (*blast intro*: *czeroI*)+

lemma *cprod-infinite*: $Cinfinite\ r \Longrightarrow r *c\ r =o\ r$
using *cprod-infinite1' Cinfinite-Cnotzero ordLeq-refl* **by** *blast*

lemma *cexp-cprod-ordLeq*:
assumes *r1*: *Card-order r1* **and** *r2*: *Cinfinite r2*
and *r3*: *Cnotzero r3* $r3 \leq o\ r2$
shows $(r1\ ^c\ r2)\ ^c\ r3 =o\ r1\ ^c\ r2$ (**is** $?L =o\ ?R$)
proof−
have $?L =o\ r1\ ^c\ (r2 *c\ r3)$ **using** *cexp-cprod*[*OF r1*] **.**
also have $r1\ ^c\ (r2 *c\ r3) =o\ ?R$
using *assms* **by** (*fastforce simp*: *Card-order-cprod intro*: *cprod-infinite1' cexp-cong2*)
finally show *?thesis* **.**
qed

lemma *Cnotzero-UNIV*: *Cnotzero* $|UNIV|$
by (*auto simp*: *card-of-Card-order card-of-ordIso-czero-iff-empty*)

lemma *ordLess-ctwo-cexp*:
assumes *Card-order r*
shows $r <o\ ctwo\ ^c\ r$
proof −
have $r <o\ |Pow\ (Field\ r)|$ **using** *assms* **by** (*rule Card-order-Pow*)
also have $|Pow\ (Field\ r)| =o\ ctwo\ ^c\ r$
unfolding *ctwo-def cexp-def Field-card-of* **by** (*rule card-of-Pow-Func*)
finally show *?thesis* **.**
qed

lemma *ordLeq-cexp1*:

assumes *Cnotzero r Card-order q*
shows *q ≤o q ^c r*
proof (*cases q =o* (*czero* :: *'a rel*))
case *True* **thus** *?thesis* **by** (*simp only*: *card-of-empty cexp-def czero-def ordIso-ordLeq-trans*)
next
case *False*
have *q =o q ^c cone*
by (*blast intro*: *assms ordIso-symmetric cexp-cone*)
also have *q ^c cone ≤o q ^c r*
using *assms*
by (*intro cexp-mono2*) (*simp-all add*: *cone-ordLeq-Cnotzero cone-not-czero Card-order-cone*)
finally show *?thesis* **.**
qed

lemma *ordLeq-cexp2*:
assumes *ctwo ≤o q Card-order r*
shows *r ≤o q ^c r*
proof (*cases r =o* (*czero* :: *'a rel*))
case *True* **thus** *?thesis* **by** (*simp only*: *card-of-empty cexp-def czero-def ordIso-ordLeq-trans*)
next
case *False*
have *r <o ctwo ^c r*
by (*blast intro*: *assms ordLess-ctwo-cexp*)
also have *ctwo ^c r ≤o q ^c r*
by (*blast intro*: *assms cexp-mono1*)
finally show *?thesis* **by** (*rule ordLess-imp-ordLeq*)
qed

lemma *cinfinite-cexp*: ⟦*ctwo ≤o q*; *Cinfinite r*⟧ ⟹ *cinfinite* (*q ^c r*)
by (*rule cinfinite-mono*[*OF ordLeq-cexp2*]) *simp-all*

lemma *Cinfinite-cexp*:
⟦*ctwo ≤o q*; *Cinfinite r*⟧ ⟹ *Cinfinite* (*q ^c r*)
by (*simp add*: *cinfinite-cexp Card-order-cexp*)

lemma *card-order-cexp*:
assumes *card-order r1 card-order r2*
shows *card-order* (*r1 ^c r2*)
proof −
have *Field r1 = UNIV Field r2 = UNIV* **using** *assms card-order-on-Card-order* **by** *auto*
thus *?thesis* **unfolding** *cexp-def Func-def* **using** *card-of-card-order-on* **by** *simp*
qed

lemma *ctwo-ordLess-natLeq*: *ctwo <o natLeq*
unfolding *ctwo-def* **using** *finite-UNIV natLeq-cinfinite natLeq-Card-order*

by (*intro Cfinite-ordLess-Cinfinite*) (*auto simp*: *cfinite-def card-of-Card-order*)

lemma *ctwo-ordLess-Cinfinite*: *Cinfinite r* $\Longrightarrow$ *ctwo* $<o$ *r*
by (*rule ordLess-ordLeq-trans*[*OF ctwo-ordLess-natLeq natLeq-ordLeq-cinfinite*])

lemma *ctwo-ordLeq-Cinfinite*:
assumes *Cinfinite r*
shows *ctwo* $\leq o$ *r*
by (*rule ordLess-imp-ordLeq*[*OF ctwo-ordLess-Cinfinite*[*OF assms*]])

lemma *Un-Cinfinite-bound*: $[\![|A| \leq o\ r;\ |B| \leq o\ r;\ \textit{Cinfinite}\ r]\!] \Longrightarrow |A \cup B| \leq o\ r$
by (*auto simp add*: *cinfinite-def card-of-Un-ordLeq-infinite-Field*)

lemma *Un-Cinfinite-bound-strict*: $[\![|A| <o\ r;\ |B| <o\ r;\ \textit{Cinfinite}\ r]\!] \Longrightarrow |A \cup B| <o\ r$
by (*auto simp add*: *cinfinite-def card-of-Un-ordLess-infinite-Field*)

lemma *UNION-Cinfinite-bound*: $[\![|I| \leq o\ r;\ \forall i \in I.\ |A\ i| \leq o\ r;\ \textit{Cinfinite}\ r]\!] \Longrightarrow |\bigcup i \in I.\ A\ i| \leq o\ r$
by (*auto simp add*: *card-of-UNION-ordLeq-infinite-Field cinfinite-def*)

lemma *csum-cinfinite-bound*:
assumes $p \leq o\ r\ q \leq o\ r$ *Card-order p Card-order q Cinfinite r*
shows $p +c\ q \leq o\ r$
proof −
have $|\textit{Field}\ p| \leq o\ r\ |\textit{Field}\ q| \leq o\ r$
using *assms card-of-least ordLeq-transitive* **unfolding** *card-order-on-def* **by** *blast+*
with *assms* **show** *?thesis* **unfolding** *cinfinite-def csum-def*
by (*blast intro*: *card-of-Plus-ordLeq-infinite-Field*)
qed

lemma *cprod-cinfinite-bound*:
assumes $p \leq o\ r\ q \leq o\ r$ *Card-order p Card-order q Cinfinite r*
shows $p *c\ q \leq o\ r$
proof −
from *assms*(*1*−*4*) **have** $|\textit{Field}\ p| \leq o\ r\ |\textit{Field}\ q| \leq o\ r$
unfolding *card-order-on-def* **using** *card-of-least ordLeq-transitive* **by** *blast+*
with *assms* **show** *?thesis* **unfolding** *cinfinite-def cprod-def*
by (*blast intro*: *card-of-Times-ordLeq-infinite-Field*)
qed

lemma *cprod-infinite2'*: $[\![\textit{Cnotzero}\ r1;\ \textit{Cinfinite}\ r2;\ r1 \leq o\ r2]\!] \Longrightarrow r1 *c\ r2 =o\ r2$
unfolding *ordIso-iff-ordLeq*
by (*intro conjI cprod-cinfinite-bound ordLeq-cprod2 ordLeq-refl*)
(*auto dest*!: *ordIso-imp-ordLeq not-ordLeq-ordLess simp*: *czero-def Card-order-empty*)

lemma *regularCard-cprod*:

assumes *Cinfinite r Cinfinite s regularCard r regularCard s*
shows *regularCard* (*r* *c *s*)
proof (*cases r* ≤*o s*)
case *True*
with *assms Cinfinite-Cnotzero* **show** *?thesis*
by (*force intro*: *regularCard-ordIso ordIso-symmetric*[*OF cprod-infinite2* ′])
next
case *False*
have *Well-order r Well-order s* **using** *assms card-order-on-well-order-on* **by** *auto*
then have *s* ≤*o r* **using** *not-ordLeq-iff-ordLess ordLess-imp-ordLeq False* **by** *blast*
with *assms Cinfinite-Cnotzero* **show** *?thesis*
by (*force intro*: *regularCard-ordIso ordIso-symmetric*[*OF cprod-infinite1* ′])
qed

lemma *cprod-csum-cexp*:
r1 *c *r2* ≤*o* (*r1* +*c r2*) ^*c ctwo*
unfolding *cprod-def csum-def cexp-def ctwo-def Field-card-of*
proof −
let *?f* = λ(*a*, *b*). %*x*. *if x then Inl a else Inr b*
have *inj-on ?f* (*Field r1* × *Field r2*) (**is** *inj-on - ?LHS*)
by (*auto simp*: *inj-on-def fun-eq-iff split*: *bool.split*)
moreover
have *?f* ‘ *?LHS* ⊆ *Func* (*UNIV* :: *bool set*) (*Field r1* <+> *Field r2*) (**is** - ⊆ *?RHS*)
by (*auto simp*: *Func-def*)
ultimately show |*?LHS*| ≤*o* |*?RHS*| **using** *card-of-ordLeq* **by** *blast*
qed

lemma *Cfinite-cprod-Cinfinite*: ⟦*Cfinite r*; *Cinfinite s*⟧ ⟹ *r* *c *s* ≤*o s*
by (*intro cprod-cinfinite-bound*)
(*auto intro*: *ordLeq-refl ordLess-imp-ordLeq*[*OF Cfinite-ordLess-Cinfinite*])

lemma *cprod-cexp*: (*r* *c *s*) ^*c t* =*o r* ^*c t* *c *s* ^*c t*
unfolding *cprod-def cexp-def Field-card-of* **by** (*rule Func-Times-Range*)

lemma *cprod-cexp-csum-cexp-Cinfinite*:
assumes *t*: *Cinfinite t*
shows (*r* *c *s*) ^*c t* ≤*o* (*r* +*c s*) ^*c t*
proof −
have (*r* *c *s*) ^*c t* ≤*o* ((*r* +*c s*) ^*c ctwo*) ^*c t*
by (*rule cexp-mono1*[*OF cprod-csum-cexp conjunct2*[*OF t*]])
also have ((*r* +*c s*) ^*c ctwo*) ^*c t* =*o* (*r* +*c s*) ^*c* (*ctwo* *c *t*)
by (*rule cexp-cprod*[*OF Card-order-csum*])
also have (*r* +*c s*) ^*c* (*ctwo* *c *t*) =*o* (*r* +*c s*) ^*c* (*t* *c *ctwo*)
by (*rule cexp-cong2*[*OF cprod-com Card-order-csum Card-order-cprod*])
also have (*r* +*c s*) ^*c* (*t* *c *ctwo*) =*o* ((*r* +*c s*) ^*c t*) ^*c ctwo*
by (*rule ordIso-symmetric*[*OF cexp-cprod*[*OF Card-order-csum*]])
also have ((*r* +*c s*) ^*c t*) ^*c ctwo* =*o* (*r* +*c s*) ^*c t*

by (*rule cexp-cprod-ordLeq*[*OF Card-order-csum t ctwo-Cnotzero ctwo-ordLeq-Cinfinite*[*OF t*]])
finally show *?thesis* **.**
qed

lemma *Cfinite-cexp-Cinfinite*:
assumes *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows *s* ^*c t* ≤*o ctwo* ^*c t*
proof (*cases s* ≤*o ctwo*)
case *True* **thus** *?thesis* **using** *t* **by** (*blast intro*: *cexp-mono1*)
next
case *False*
hence *ctwo* ≤*o s* **using** *ordLeq-total*[*of s ctwo*] *Card-order-ctwo s*
by (*auto intro*: *card-order-on-well-order-on*)
hence *Cnotzero s* **using** *Cnotzero-mono*[*OF ctwo-Cnotzero*] *s* **by** *blast*
hence *st*: *Cnotzero* (*s* ∗*c t*) **by** (*intro Cinfinite-Cnotzero*[*OF Cinfinite-cprod2*]) (*auto simp*: *t*)
have *s* ^*c t* ≤*o* (*ctwo* ^*c s*) ^*c t*
using *assms* **by** (*blast intro*: *cexp-mono1 ordLess-imp-ordLeq*[*OF ordLess-ctwo-cexp*])
also have (*ctwo* ^*c s*) ^*c t* =*o ctwo* ^*c* (*s* ∗*c t*)
by (*blast intro*: *Card-order-ctwo cexp-cprod*)
also have *ctwo* ^*c* (*s* ∗*c t*) ≤*o ctwo* ^*c t*
using *assms st* **by** (*intro cexp-mono2-Cnotzero Cfinite-cprod-Cinfinite Card-order-ctwo*)
finally show *?thesis* **.**
qed

lemma *csum-Cfinite-cexp-Cinfinite*:
assumes *r*: *Card-order r* **and** *s*: *Cfinite s* **and** *t*: *Cinfinite t*
shows (*r* +*c s*) ^*c t* ≤*o* (*r* +*c ctwo*) ^*c t*
proof (*cases Cinfinite r*)
case *True*
hence *r* +*c s* =*o r* **by** (*intro csum-absorb1 ordLess-imp-ordLeq*[*OF Cfinite-ordLess-Cinfinite*] *s*)
hence (*r* +*c s*) ^*c t* =*o r* ^*c t* **using** *t* **by** (*blast intro*: *cexp-cong1*)
also have *r* ^*c t* ≤*o* (*r* +*c ctwo*) ^*c t* **using** *t* **by** (*blast intro*: *cexp-mono1 ordLeq-csum1 r*)
finally show *?thesis* **.**
next
case *False*
with *r* **have** *Cfinite r* **unfolding** *cinfinite-def cfinite-def* **by** *auto*
hence *Cfinite* (*r* +*c s*) **by** (*intro Cfinite-csum s*)
hence (*r* +*c s*) ^*c t* ≤*o ctwo* ^*c t* **by** (*intro Cfinite-cexp-Cinfinite t*)
also have *ctwo* ^*c t* ≤*o* (*r* +*c ctwo*) ^*c t* **using** *t*
by (*blast intro*: *cexp-mono1 ordLeq-csum2 Card-order-ctwo*)
finally show *?thesis* **.**
qed

lemma *Cinfinite-cardSuc*: *Cinfinite r* $\Longrightarrow$ *Cinfinite* (*cardSuc r*)
by (*simp add*: *cinfinite-def cardSuc-Card-order cardSuc-finite*)

lemma *cardSuc-UNION-Cinfinite*:
assumes *Cinfinite r relChain* (*cardSuc r*) *As B* $\leq$ ($\bigcup$ *i* $\in$ *Field* (*cardSuc r*). *As i*) |*B*| <=*o r*
shows $\exists$ *i* $\in$ *Field* (*cardSuc r*). *B* $\leq$ *As i*
using *cardSuc-UNION assms* **unfolding** *cinfinite-def* **by** *blast*

lemma *Cinfinite-card-suc*: ⟦ *Cinfinite r* ; *card-order r* ⟧ $\Longrightarrow$ *Cinfinite* (*card-suc r*)
using *Cinfinite-cong*[*OF cardSuc-ordIso-card-suc Cinfinite-cardSuc*] **.**

lemma *card-suc-least*: ⟦*card-order r*; *Card-order s*; *r* <*o s*⟧ $\Longrightarrow$ *card-suc r* $\leq$*o s*
by (*rule ordIso-ordLeq-trans*[*OF ordIso-symmetric*[*OF cardSuc-ordIso-card-suc*]])
(*auto intro*!: *cardSuc-least simp*: *card-order-on-Card-order*)

lemma *regularCard-cardSuc*: *Cinfinite k* $\Longrightarrow$ *regularCard* (*cardSuc k*)
by (*rule infinite-cardSuc-regularCard*) (*auto simp*: *cinfinite-def*)

lemma *regularCard-card-suc*: *card-order r* $\Longrightarrow$ *Cinfinite r* $\Longrightarrow$ *regularCard* (*card-suc r*)
using *cardSuc-ordIso-card-suc Cinfinite-cardSuc regularCard-cardSuc regularCard-ordIso*
by *blast*

end

32 Function Definition Base

theory *Fun-Def-Base*
imports *Ctr-Sugar Set Wellfounded*
begin

ML-file ‹*Tools/Function/function-lib.ML*›
named-theorems *termination-simp simplification rules for termination proofs*
ML-file ‹*Tools/Function/function-common.ML*›
ML-file ‹*Tools/Function/function-context-tree.ML*›

attribute-setup *fundef-cong* =
‹*Attrib.add-del Function-Context-Tree.cong-add Function-Context-Tree.cong-del*›
declaration of congruence rule for function definitions

ML-file ‹*Tools/Function/sum-tree.ML*›

end

33 Definition of Bounded Natural Functors

theory *BNF-Def*

imports *BNF-Cardinal-Arithmetic Fun-Def-Base*
keywords
print-bnfs :: *diag* **and**
bnf :: *thy-goal-defn*
begin

lemma *Collect-case-prodD*: $x \in Collect\ (case\text{-}prod\ A) \Longrightarrow A\ (fst\ x)\ (snd\ x)$
by *auto*

inductive
rel-sum :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow 'a + 'b \Rightarrow 'c + 'd \Rightarrow bool$
for *R1 R2*
where
R1 a c $\Longrightarrow$ *rel-sum R1 R2* (*Inl a*) (*Inl c*)
| *R2 b d* $\Longrightarrow$ *rel-sum R1 R2* (*Inr b*) (*Inr d*)

definition
rel-fun :: $('a \Rightarrow 'c \Rightarrow bool) \Rightarrow ('b \Rightarrow 'd \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('c \Rightarrow 'd) \Rightarrow bool$
where
rel-fun A B = $(\lambda f\ g.\ \forall x\ y.\ A\ x\ y \longrightarrow B\ (f\ x)\ (g\ y))$

lemma *rel-funI* [*intro*]:
assumes $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ (f\ x)\ (g\ y)$
shows *rel-fun A B f g*
using *assms* **by** (*simp add*: *rel-fun-def*)

lemma *rel-funD*:
assumes *rel-fun A B f g* **and** *A x y*
shows *B* (*f x*) (*g y*)
using *assms* **by** (*simp add*: *rel-fun-def*)

lemma *rel-fun-mono*:
⟦ *rel-fun X A f g*; $\bigwedge x\ y.\ Y\ x\ y \longrightarrow X\ x\ y$; $\bigwedge x\ y.\ A\ x\ y \Longrightarrow B\ x\ y$ ⟧ $\Longrightarrow$ *rel-fun Y B f g*
by(*simp add*: *rel-fun-def*)

lemma *rel-fun-mono'* [*mono*]:
⟦ $\bigwedge x\ y.\ Y\ x\ y \longrightarrow X\ x\ y$; $\bigwedge x\ y.\ A\ x\ y \longrightarrow B\ x\ y$ ⟧ $\Longrightarrow$ *rel-fun X A f g* $\longrightarrow$ *rel-fun Y B f g*
by(*simp add*: *rel-fun-def*)

definition *rel-set* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow bool$
where *rel-set R* = $(\lambda A\ B.\ (\forall x{\in}A.\ \exists y{\in}B.\ R\ x\ y) \wedge (\forall y{\in}B.\ \exists x{\in}A.\ R\ x\ y))$

lemma *rel-setI*:
assumes $\bigwedge x.\ x \in A \Longrightarrow \exists y{\in}B.\ R\ x\ y$
assumes $\bigwedge y.\ y \in B \Longrightarrow \exists x{\in}A.\ R\ x\ y$
shows *rel-set R A B*
using *assms* **unfolding** *rel-set-def* **by** *simp*

lemma *predicate2-transferD*:
  ⟦*rel-fun R1* (*rel-fun R2* (=)) *P Q*; *a* ∈ *A*; *b* ∈ *B*; *A* ⊆ {(*x*, *y*). *R1 x y*}; *B* ⊆ {(*x*, *y*). *R2 x y*}⟧ ⟹
  P (*fst a*) (*fst b*) ⟷ *Q* (*snd a*) (*snd b*)
  unfolding *rel-fun-def* **by** (*blast dest*!: *Collect-case-prodD*)

definition *collect* **where**
  collect F x = (⋃*f* ∈ *F*. *f x*)

lemma *fstI*: *x* = (*y*, *z*) ⟹ *fst x* = *y*
  by *simp*

lemma *sndI*: *x* = (*y*, *z*) ⟹ *snd x* = *z*
  by *simp*

lemma *bijI'*: ⟦⋀*x y*. (*f x* = *f y*) = (*x* = *y*); ⋀*y*. ∃ *x*. *y* = *f x*⟧ ⟹ *bij f*
  unfolding *bij-def inj-on-def* **by** *auto blast*

definition *Gr A f* = {(*a*, *f a*) | *a*. *a* ∈ *A*}

definition *Grp A f* = (λ*a b*. *b* = *f a* ∧ *a* ∈ *A*)

definition *vimage2p* **where**
  vimage2p f g R = (λ*x y*. *R* (*f x*) (*g y*))

lemma *collect-comp*: *collect F* ∘ *g* = *collect* ((λ*f*. *f* ∘ *g*) ‘ *F*)
  by (*rule ext*) (*simp add*: *collect-def*)

definition *convol* (‹(‹*indent=1 notation=‹mixfix convol›*›⟨-,/ -⟩)›) **where**
  ⟨*f*, *g*⟩ ≡ λ*a*. (*f a*, *g a*)

lemma *fst-convol*: *fst* ∘ ⟨*f*, *g*⟩ = *f*
  apply(*rule ext*)
  unfolding *convol-def* **by** *simp*

lemma *snd-convol*: *snd* ∘ ⟨*f*, *g*⟩ = *g*
  apply(*rule ext*)
  unfolding *convol-def* **by** *simp*

lemma *convol-mem-GrpI*:
  x ∈ *A* ⟹ ⟨*id*, *g*⟩ *x* ∈ (*Collect* (*case-prod* (*Grp A g*)))
  unfolding *convol-def Grp-def* **by** *auto*

definition *csquare* **where**
  csquare A f1 f2 p1 p2 ⟷ (∀ *a* ∈ *A*. *f1* (*p1 a*) = *f2* (*p2 a*))

lemma *eq-alt*: (=) = *Grp UNIV id*

unfolding *Grp-def* **by** *auto*

lemma *leq-conversepI*: $R = (=) \Longrightarrow R \leq R^{-1-1}$
by *auto*

lemma *leq-OOI*: $R = (=) \Longrightarrow R \leq R\ OO\ R$
by *auto*

lemma *OO-Grp-alt*: $(Grp\ A\ f)^{-1-1}\ OO\ Grp\ A\ g = (\lambda x\ y.\ \exists z.\ z \in A \wedge f\ z = x \wedge g\ z = y)$
unfolding *Grp-def* **by** *auto*

lemma *Grp-UNIV-id*: $f = id \Longrightarrow (Grp\ UNIV\ f)^{-1-1}\ OO\ Grp\ UNIV\ f = Grp\ UNIV\ f$
unfolding *Grp-def* **by** *auto*

lemma *Grp-UNIV-idI*: $x = y \Longrightarrow Grp\ UNIV\ id\ x\ y$
unfolding *Grp-def* **by** *auto*

lemma *Grp-mono*: $A \leq B \Longrightarrow Grp\ A\ f \leq Grp\ B\ f$
unfolding *Grp-def* **by** *auto*

lemma *GrpI*: $[\![f\ x = y;\ x \in A]\!] \Longrightarrow Grp\ A\ f\ x\ y$
unfolding *Grp-def* **by** *auto*

lemma *GrpE*: $Grp\ A\ f\ x\ y \Longrightarrow ([\![f\ x = y;\ x \in A]\!] \Longrightarrow R) \Longrightarrow R$
unfolding *Grp-def* **by** *auto*

lemma *Collect-case-prod-Grp-eqD*: $z \in Collect\ (case\text{-}prod\ (Grp\ A\ f)) \Longrightarrow (f \circ fst)\ z = snd\ z$
unfolding *Grp-def comp-def* **by** *auto*

lemma *Collect-case-prod-Grp-in*: $z \in Collect\ (case\text{-}prod\ (Grp\ A\ f)) \Longrightarrow fst\ z \in A$
unfolding *Grp-def comp-def* **by** *auto*

definition *pick-middlep* $P\ Q\ a\ c = (SOME\ b.\ P\ a\ b \wedge Q\ b\ c)$

lemma *pick-middlep*:
$(P\ OO\ Q)\ a\ c \Longrightarrow P\ a\ (pick\text{-}middlep\ P\ Q\ a\ c) \wedge Q\ (pick\text{-}middlep\ P\ Q\ a\ c)\ c$
unfolding *pick-middlep-def* **by** (*rule someI-ex*) *auto*

definition *fstOp* **where**
$fstOp\ P\ Q\ ac = (fst\ ac,\ pick\text{-}middlep\ P\ Q\ (fst\ ac)\ (snd\ ac))$

definition *sndOp* **where**
$sndOp\ P\ Q\ ac = (pick\text{-}middlep\ P\ Q\ (fst\ ac)\ (snd\ ac),\ (snd\ ac))$

lemma *fstOp-in*: $ac \in Collect\ (case\text{-}prod\ (P\ OO\ Q)) \Longrightarrow fstOp\ P\ Q\ ac \in Collect\ (case\text{-}prod\ P)$

unfolding *fstOp-def mem-Collect-eq*
by (*subst* (*asm*) *surjective-pairing*, *unfold prod.case*) (*erule pick-middlep*[*THEN conjunct1*])

lemma *fst-fstOp*: *fst bc* = (*fst* ∘ *fstOp P Q*) *bc*
unfolding *comp-def fstOp-def* **by** *simp*

lemma *snd-sndOp*: *snd bc* = (*snd* ∘ *sndOp P Q*) *bc*
unfolding *comp-def sndOp-def* **by** *simp*

lemma *sndOp-in*: *ac* ∈ *Collect* (*case-prod* (*P OO Q*)) ⟹ *sndOp P Q ac* ∈ *Collect* (*case-prod Q*)
unfolding *sndOp-def mem-Collect-eq*
by (*subst* (*asm*) *surjective-pairing*, *unfold prod.case*) (*erule pick-middlep*[*THEN conjunct2*])

lemma *csquare-fstOp-sndOp*:
csquare (*Collect* (*f* (*P OO Q*))) *snd fst* (*fstOp P Q*) (*sndOp P Q*)
unfolding *csquare-def fstOp-def sndOp-def* **using** *pick-middlep* **by** *simp*

lemma *snd-fst-flip*: *snd xy* = (*fst* ∘ (%(*x*, *y*). (*y*, *x*))) *xy*
by (*simp split*: *prod.split*)

lemma *fst-snd-flip*: *fst xy* = (*snd* ∘ (%(*x*, *y*). (*y*, *x*))) *xy*
by (*simp split*: *prod.split*)

lemma *flip-pred*: *A* ⊆ *Collect* (*case-prod* (*R* $^{-1-1}$)) ⟹ (%(*x*, *y*). (*y*, *x*)) ‘ *A* ⊆ *Collect* (*case-prod R*)
by *auto*

lemma *predicate2-eqD*: *A* = *B* ⟹ *A a b* ⟷ *B a b*
by *simp*

lemma *case-sum-o-inj*: *case-sum f g* ∘ *Inl* = *f case-sum f g* ∘ *Inr* = *g*
by *auto*

lemma *map-sum-o-inj*: *map-sum f g* ∘ *Inl* = *Inl* ∘ *f map-sum f g* ∘ *Inr* = *Inr* ∘ *g*
by *auto*

lemma *card-order-csum-cone-cexp-def*:
card-order r ⟹ (|*A1*| +*c cone*) ^*c r* = |*Func UNIV* (*Inl* ‘ *A1* ∪ {*Inr* ()})|
unfolding *cexp-def cone-def Field-csum Field-card-of* **by** (*auto dest*: *Field-card-order*)

lemma *If-the-inv-into-in-Func*:
⟦*inj-on g C*; *C* ⊆ *B* ∪ {*x*}⟧ ⟹
(λ*i*. *if i* ∈ *g* ‘ *C then the-inv-into C g i else x*) ∈ *Func UNIV* (*B* ∪ {*x*})
unfolding *Func-def* **by** (*auto dest*: *the-inv-into-into*)

lemma *If-the-inv-into-f-f*:

$[\![i \in C; \mathit{inj\text{-}on}\ g\ C]\!] \Longrightarrow ((\lambda i.\ \mathit{if}\ i \in g\ `\ C\ \mathit{then}\ \mathit{the\text{-}inv\text{-}into}\ C\ g\ i\ \mathit{else}\ x) \circ g)\ i = \mathit{id}\ i$
unfolding *Func-def* **by** (*auto elim*: *the-inv-into-f-f*)

lemma *the-inv-f-o-f-id*: $\mathit{inj}\ f \Longrightarrow (\mathit{the\text{-}inv}\ f \circ f)\ z = \mathit{id}\ z$
by (*simp add*: *the-inv-f-f*)

lemma *vimage2pI*: $R\ (f\ x)\ (g\ y) \Longrightarrow \mathit{vimage2p}\ f\ g\ R\ x\ y$
unfolding *vimage2p-def* **.**

lemma *rel-fun-iff-leq-vimage2p*: $(\mathit{rel\text{-}fun}\ R\ S)\ f\ g = (R \leq \mathit{vimage2p}\ f\ g\ S)$
unfolding *rel-fun-def vimage2p-def* **by** *auto*

lemma *convol-image-vimage2p*: $\langle f \circ \mathit{fst}, g \circ \mathit{snd}\rangle\ `\ \mathit{Collect}\ (\mathit{case\text{-}prod}\ (\mathit{vimage2p}\ f\ g\ R)) \subseteq \mathit{Collect}\ (\mathit{case\text{-}prod}\ R)$
unfolding *vimage2p-def convol-def* **by** *auto*

lemma *vimage2p-Grp*: $\mathit{vimage2p}\ f\ g\ P = \mathit{Grp}\ \mathit{UNIV}\ f\ OO\ P\ OO\ (\mathit{Grp}\ \mathit{UNIV}\ g)^{-1-1}$
unfolding *vimage2p-def Grp-def* **by** *auto*

lemma *subst-Pair*: $P\ x\ y \Longrightarrow a = (x,\ y) \Longrightarrow P\ (\mathit{fst}\ a)\ (\mathit{snd}\ a)$
by *simp*

lemma *comp-apply-eq*: $f\ (g\ x) = h\ (k\ x) \Longrightarrow (f \circ g)\ x = (h \circ k)\ x$
unfolding *comp-apply* **by** *assumption*

lemma *refl-ge-eq*: $(\bigwedge x.\ R\ x\ x) \Longrightarrow (=) \leq R$
by *auto*

lemma *ge-eq-refl*: $(=) \leq R \Longrightarrow R\ x\ x$
by *auto*

lemma *reflp-eq*: $\mathit{reflp}\ R = ((=) \leq R)$
by (*auto simp*: *reflp-def fun-eq-iff*)

lemma *transp-relcompp*: $\mathit{transp}\ r \longleftrightarrow r\ OO\ r \leq r$
by (*auto simp*: *transp-def*)

lemma *symp-conversep*: $\mathit{symp}\ R = (R^{-1-1} \leq R)$
by (*auto simp*: *symp-def fun-eq-iff*)

lemma *diag-imp-eq-le*: $(\bigwedge x.\ x \in A \Longrightarrow R\ x\ x) \Longrightarrow \forall x\ y.\ x \in A \longrightarrow y \in A \longrightarrow x = y \longrightarrow R\ x\ y$
by *blast*

definition *eq-onp* :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a \Rightarrow 'a \Rightarrow \mathit{bool}$
where $\mathit{eq\text{-}onp}\ R = (\lambda x\ y.\ R\ x \wedge x = y)$

lemma *eq-onp-Grp*: *eq-onp P = BNF-Def.Grp (Collect P) id*
unfolding *eq-onp-def Grp-def* **by** *auto*

lemma *eq-onp-to-eq*: *eq-onp P x y ⟹ x = y*
by (*simp add*: *eq-onp-def*)

lemma *eq-onp-top-eq-eq*: *eq-onp top = (=)*
by (*simp add*: *eq-onp-def*)

lemma *eq-onp-same-args*: *eq-onp P x x = P x*
by (*auto simp add*: *eq-onp-def*)

lemma *eq-onp-eqD*: *eq-onp P = Q ⟹ P x = Q x x*
unfolding *eq-onp-def* **by** *blast*

lemma *Ball-Collect*: *Ball A P = (A ⊆ (Collect P))*
by *auto*

lemma *eq-onp-mono0*: *∀ x∈A. P x ⟶ Q x ⟹ ∀ x∈A. ∀ y∈A. eq-onp P x y ⟶ eq-onp Q x y*
unfolding *eq-onp-def* **by** *auto*

lemma *eq-onp-True*: *eq-onp (λ-. True) = (=)*
unfolding *eq-onp-def* **by** *simp*

lemma *Ball-image-comp*: *Ball (f ‘ A) g = Ball A (g ∘ f)*
by *auto*

lemma *rel-fun-Collect-case-prodD*:
rel-fun A B f g ⟹ X ⊆ Collect (case-prod A) ⟹ x ∈ X ⟹ B ((f ∘ fst) x) ((g ∘ snd) x)
unfolding *rel-fun-def* **by** *auto*

lemma *eq-onp-mono-iff*: *eq-onp P ≤ eq-onp Q ⟷ P ≤ Q*
unfolding *eq-onp-def* **by** *auto*

ML-file ‹*Tools/BNF/bnf-util.ML*›
ML-file ‹*Tools/BNF/bnf-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-def-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-def.ML*›

end

34 Composition of Bounded Natural Functors

theory *BNF-Composition*
imports *BNF-Def*
begin

lemma *ssubst-mem*: $[\![t = s;\ s \in X]\!] \Longrightarrow t \in X$
by *simp*

lemma *empty-natural*: $(\lambda\text{-}.\ \{\}) \circ f = image\ g \circ (\lambda\text{-}.\ \{\})$
by (*rule ext*) *simp*

lemma *Cinfinite-gt-empty*: $Cinfinite\ r \Longrightarrow |\{\}| <o\ r$
by (*simp add*: *cinfinite-def finite-ordLess-infinite card-of-ordIso-finite Field-card-of card-of-well-order-on emptyI card-order-on-well-order-on*)

lemma *Union-natural*: $Union \circ image\ (image\ f) = image\ f \circ Union$
by (*rule ext*) (*auto simp only*: *comp-apply*)

lemma *in-Union-o-assoc*: $x \in (Union \circ gset \circ gmap)\ A \Longrightarrow x \in (Union \circ (gset \circ gmap))\ A$
by (*unfold comp-assoc*)

lemma *regularCard-UNION-bound*:
assumes *Cinfinite r regularCard r* **and** $|I| <o\ r$ $\bigwedge i.\ i \in I \Longrightarrow |A\ i| <o\ r$
shows $|\bigcup i{\in}I.\ A\ i| <o\ r$
using *assms cinfinite-def regularCard-stable stable-UNION* **by** *blast*

lemma *comp-single-set-bd-strict*:
assumes *fbd*: *Cinfinite fbd regularCard fbd* **and**
gbd: *Cinfinite gbd regularCard gbd* **and**
fset-bd: $\bigwedge x.\ |fset\ x| <o\ fbd$ **and**
gset-bd: $\bigwedge x.\ |gset\ x| <o\ gbd$
shows $|\bigcup (fset\ `\ gset\ x)| <o\ gbd *c\ fbd$
proof (*cases fbd <o gbd*)
case *True*
then have $|fset\ x| <o\ gbd$ **for** *x* **using** *fset-bd ordLess-transitive* **by** *blast*
then have $|\bigcup (fset\ `\ gset\ x)| <o\ gbd$ **using** *regularCard-UNION-bound*[*OF gbd gset-bd*] **by** *blast*
then have $|\bigcup\ (fset\ `\ gset\ x)| <o\ fbd *c\ gbd$
using *ordLess-ordLeq-trans ordLeq-cprod2 gbd*(*1*) *fbd*(*1*) *cinfinite-not-czero* **by** *blast*
then show *?thesis* **using** *ordLess-ordIso-trans cprod-com* **by** *blast*
next
case *False*
have *Well-order fbd Well-order gbd* **using** *fbd*(*1*) *gbd*(*1*) *card-order-on-well-order-on* **by** *auto*
then have $gbd \leq o\ fbd$ **using** *not-ordLess-iff-ordLeq False* **by** *blast*
then have $|gset\ x| <o\ fbd$ **for** *x* **using** *gset-bd ordLess-ordLeq-trans* **by** *blast*
then have $|\bigcup (fset\ `\ gset\ x)| <o\ fbd$ **using** *regularCard-UNION-bound*[*OF fbd*] *fset-bd* **by** *blast*
then show *?thesis* **using** *ordLess-ordLeq-trans ordLeq-cprod2 gbd*(*1*) *fbd*(*1*) *cinfinite-not-czero* **by** *blast*
qed

lemma *comp-single-set-bd*:
assumes *fbd-Card-order*: *Card-order fbd* **and**
fset-bd: $\bigwedge x.\ |fset\ x| \leq o\ fbd$ **and**
gset-bd: $\bigwedge x.\ |gset\ x| \leq o\ gbd$
shows $|\bigcup (fset\ `\ gset\ x)| \leq o\ gbd *c\ fbd$
apply *simp*
apply (*rule ordLeq-transitive*)
apply (*rule card-of-UNION-Sigma*)
apply (*subst SIGMA-CSUM*)
apply (*rule ordLeq-transitive*)
apply (*rule card-of-Csum-Times'*)
apply (*rule fbd-Card-order*)
apply (*rule ballI*)
apply (*rule fset-bd*)
apply (*rule ordLeq-transitive*)
apply (*rule cprod-mono1*)
apply (*rule gset-bd*)
apply (*rule ordIso-imp-ordLeq*)
apply (*rule ordIso-refl*)
apply (*rule Card-order-cprod*)
done

lemma *csum-dup*: *cinfinite r* $\Longrightarrow$ *Card-order r* $\Longrightarrow$ $p +c\ p' =o\ r +c\ r \Longrightarrow p +c\ p' =o\ r$
apply (*erule ordIso-transitive*)
apply (*frule csum-absorb2'*)
apply (*erule ordLeq-refl*)
by *simp*

lemma *cprod-dup*: *cinfinite r* $\Longrightarrow$ *Card-order r* $\Longrightarrow$ $p *c\ p' =o\ r *c\ r \Longrightarrow p *c\ p' =o\ r$
apply (*erule ordIso-transitive*)
apply (*rule cprod-infinite*)
by *simp*

lemma *Union-image-insert*: $\bigcup (f\ `\ insert\ a\ B) = f\ a \cup \bigcup (f\ `\ B)$
by *simp*

lemma *Union-image-empty*: $A \cup \bigcup (f\ `\ \{\}) = A$
by *simp*

lemma *image-o-collect*: *collect* $((\lambda f.\ image\ g \circ f)\ `\ F) = image\ g \circ collect\ F$
by (*rule ext*) (*auto simp add*: *collect-def*)

lemma *conj-subset-def*: $A \subseteq \{x.\ P\ x \wedge Q\ x\} = (A \subseteq \{x.\ P\ x\} \wedge A \subseteq \{x.\ Q\ x\})$
by *blast*

lemma *UN-image-subset*: $\bigcup (f\ `\ g\ x) \subseteq X = (g\ x \subseteq \{x.\ f\ x \subseteq X\})$
by *blast*

lemma *comp-set-bd-Union-o-collect*: $|\bigcup(\bigcup((\lambda f.\ f\ x)\ `\ X))| \leq o\ hbd \Longrightarrow |(Union \circ collect\ X)\ x| \leq o\ hbd$
by (*unfold comp-apply collect-def*) *simp*

lemma *comp-set-bd-Union-o-collect-strict*: $|\bigcup(\bigcup((\lambda f.\ f\ x)\ `\ X))| <o\ hbd \Longrightarrow |(Union \circ collect\ X)\ x| <o\ hbd$
by (*unfold comp-apply collect-def*) *simp*

lemma *Collect-inj*: *Collect P* = *Collect Q* $\Longrightarrow$ *P* = *Q*
by *blast*

lemma *Grp-fst-snd*: $(Grp\ (Collect\ (case\text{-}prod\ R))\ fst)^{-1-1}\ OO\ Grp\ (Collect\ (case\text{-}prod\ R))\ snd = R$
unfolding *Grp-def fun-eq-iff relcompp.simps* **by** *auto*

lemma *OO-Grp-cong*: *A* = *B* $\Longrightarrow$ $(Grp\ A\ f)^{-1-1}\ OO\ Grp\ A\ g = (Grp\ B\ f)^{-1-1}\ OO\ Grp\ B\ g$
by (*rule arg-cong*)

lemma *vimage2p-relcompp-mono*: *R OO S* $\leq$ *T* $\Longrightarrow$
vimage2p f g R OO vimage2p g h S $\leq$ *vimage2p f h T*
unfolding *vimage2p-def* **by** *auto*

lemma *type-copy-map-cong0*: *M* (*g x*) = *N* (*h x*) $\Longrightarrow$ $(f \circ M \circ g)\ x = (f \circ N \circ h)\ x$
by *auto*

lemma *type-copy-set-bd*: $(\bigwedge y.\ |S\ y| <o\ bd) \Longrightarrow |(S \circ Rep)\ x| <o\ bd$
by *auto*

lemma *vimage2p-cong*: *R* = *S* $\Longrightarrow$ *vimage2p f g R* = *vimage2p f g S*
by *simp*

lemma *Ball-comp-iff*: $(\lambda x.\ Ball\ (A\ x)\ f) \circ g = (\lambda x.\ Ball\ ((A \circ g)\ x)\ f)$
unfolding *o-def* **by** *auto*

lemma *conj-comp-iff*: $(\lambda x.\ P\ x \wedge Q\ x) \circ g = (\lambda x.\ (P \circ g)\ x \wedge (Q \circ g)\ x)$
unfolding *o-def* **by** *auto*

context
fixes *Rep Abs*
assumes *type-copy*: *type-definition Rep Abs UNIV*
begin

lemma *type-copy-map-id0*: *M* = *id* $\Longrightarrow$ *Abs* $\circ$ *M* $\circ$ *Rep* = *id*
using *type-definition.Rep-inverse*[*OF type-copy*] **by** *auto*

lemma *type-copy-map-comp0*: *M* = *M1* $\circ$ *M2* $\Longrightarrow$ $f \circ M \circ g = (f \circ M1 \circ Rep)$

$\circ$ (*Abs* $\circ$ *M2* $\circ$ *g*)
using *type-definition.Abs-inverse*[*OF type-copy UNIV-I*] **by** *auto*

lemma *type-copy-set-map0*: *S* $\circ$ *M* = *image f* $\circ$ *S'* $\Longrightarrow$ (*S* $\circ$ *Rep*) $\circ$ (*Abs* $\circ$ *M* $\circ$ *g*) = *image f* $\circ$ (*S'* $\circ$ *g*)
using *type-definition.Abs-inverse*[*OF type-copy UNIV-I*] **by** (*auto simp*: *o-def fun-eq-iff*)

lemma *type-copy-wit*: *x* $\in$ (*S* $\circ$ *Rep*) (*Abs y*) $\Longrightarrow$ *x* $\in$ *S y*
using *type-definition.Abs-inverse*[*OF type-copy UNIV-I*] **by** *auto*

lemma *type-copy-vimage2p-Grp-Rep*: *vimage2p f Rep* (*Grp* (*Collect P*) *h*) =
Grp (*Collect* (λ*x*. *P* (*f x*))) (*Abs* $\circ$ *h* $\circ$ *f*)
unfolding *vimage2p-def Grp-def fun-eq-iff*
by (*auto simp*: *type-definition.Abs-inverse*[*OF type-copy UNIV-I*]
type-definition.Rep-inverse[*OF type-copy*] *dest*: *sym*)

lemma *type-copy-vimage2p-Grp-Abs*:
$\bigwedge$*h*. *vimage2p g Abs* (*Grp* (*Collect P*) *h*) = *Grp* (*Collect* (λ*x*. *P* (*g x*))) (*Rep* $\circ$ *h* $\circ$ *g*)
unfolding *vimage2p-def Grp-def fun-eq-iff*
by (*auto simp*: *type-definition.Abs-inverse*[*OF type-copy UNIV-I*]
type-definition.Rep-inverse[*OF type-copy*] *dest*: *sym*)

lemma *type-copy-ex-RepI*: ($\exists$ *b*. *F b*) = ($\exists$ *b*. *F* (*Rep b*))
proof *safe*
fix *b* **assume** *F b*
show $\exists$ *b'*. *F* (*Rep b'*)
proof (*rule exI*)
from ‹*F b*› **show** *F* (*Rep* (*Abs b*)) **using** *type-definition.Abs-inverse*[*OF type-copy*]
by *auto*
qed
qed *blast*

lemma *vimage2p-relcompp-converse*:
vimage2p f g (*R*$^{-1-1}$ *OO S*) = (*vimage2p Rep f R*)$^{-1-1}$ *OO vimage2p Rep g S*
unfolding *vimage2p-def relcompp.simps conversep.simps fun-eq-iff image-def*
by (*auto simp*: *type-copy-ex-RepI*)

end

bnf *DEADID*: *'a*
map: *id* :: *'a* $\Rightarrow$ *'a*
bd: *natLeq*
rel: (=) :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
by (*auto simp add*: *natLeq-card-order natLeq-cinfinite regularCard-natLeq*)

definition *id-bnf* :: *'a* $\Rightarrow$ *'a* **where**
id-bnf $\equiv$ (λ*x*. *x*)

lemma *id-bnf-apply*: *id-bnf x = x*
unfolding *id-bnf-def* **by** *simp*

bnf *ID*: *'a*
map: *id-bnf* :: *('a ⇒ 'b) ⇒ 'a ⇒ 'b*
sets: *λx. {x}*
bd: *natLeq*
rel: *id-bnf* :: *('a ⇒ 'b ⇒ bool) ⇒ 'a ⇒ 'b ⇒ bool*
pred: *id-bnf* :: *('a ⇒ bool) ⇒ 'a ⇒ bool*
unfolding *id-bnf-def*
apply (*auto simp*: *Grp-def fun-eq-iff relcompp.simps natLeq-card-order natLeq-cinfinite regularCard-natLeq*)
apply (*rule finite-ordLess-infinite*[*OF - natLeq-Well-order*])
apply (*auto simp add*: *Field-card-of Field-natLeq card-of-well-order-on*)[*3*]
done

lemma *type-definition-id-bnf-UNIV*: *type-definition id-bnf id-bnf UNIV*
unfolding *id-bnf-def* **by** *unfold-locales auto*

ML-file ‹*Tools/BNF/bnf-comp-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-comp.ML*›

hide-fact
DEADID.inj-map DEADID.inj-map-strong DEADID.map-comp DEADID.map-cong DEADID.map-cong0
DEADID.map-cong-simp DEADID.map-id DEADID.map-id0 DEADID.map-ident DEADID.map-transfer
DEADID.rel-Grp DEADID.rel-compp DEADID.rel-compp-Grp DEADID.rel-conversep DEADID.rel-eq
DEADID.rel-flip DEADID.rel-map DEADID.rel-mono DEADID.rel-transfer
ID.inj-map ID.inj-map-strong ID.map-comp ID.map-cong ID.map-cong0 ID.map-cong-simp ID.map-id
ID.map-id0 ID.map-ident ID.map-transfer ID.rel-Grp ID.rel-compp ID.rel-compp-Grp ID.rel-conversep
ID.rel-eq ID.rel-flip ID.rel-map ID.rel-mono ID.rel-transfer ID.set-map ID.set-transfer

end

35 Registration of Basic Types as Bounded Natural Functors

theory *Basic-BNFs*
imports *BNF-Def*
begin

inductive-set *setl* :: *'a + 'b ⇒ 'a set* **for** *s* :: *'a + 'b* **where**
s = Inl x ⟹ x ∈ setl s

inductive-set *setr* :: *'a + 'b ⇒ 'b set* **for** *s* :: *'a + 'b* **where**
s = Inr x ⟹ x ∈ setr s

lemma *sum-set-defs* [*code*]:
setl = (λx. case x of Inl z ⇒ {z} | - ⇒ {})
setr = (λx. case x of Inr z ⇒ {z} | - ⇒ {})
by (*auto simp*: *fun-eq-iff intro*: *setl.intros setr.intros elim*: *setl.cases setr.cases split*: *sum.splits*)

lemma *rel-sum-simps* [*code*, *simp*]:
rel-sum R1 R2 (Inl a1) (Inl b1) = R1 a1 b1
rel-sum R1 R2 (Inl a1) (Inr b2) = False
rel-sum R1 R2 (Inr a2) (Inl b1) = False
rel-sum R1 R2 (Inr a2) (Inr b2) = R2 a2 b2
by (*auto intro*: *rel-sum.intros elim*: *rel-sum.cases*)

inductive
pred-sum :: *('a ⇒ bool) ⇒ ('b ⇒ bool) ⇒ 'a + 'b ⇒ bool* **for** *P1 P2*
where
P1 a ⟹ pred-sum P1 P2 (Inl a)
| *P2 b ⟹ pred-sum P1 P2 (Inr b)*

lemma *pred-sum-inject* [*code*, *simp*]:
pred-sum P1 P2 (Inl a) ⟷ P1 a
pred-sum P1 P2 (Inr b) ⟷ P2 b
by (*simp add*: *pred-sum.simps*)+

bnf *'a + 'b*
map: *map-sum*
sets: *setl setr*
bd: *natLeq*
wits: *Inl Inr*
rel: *rel-sum*
pred: *pred-sum*
proof −
show *map-sum id id = id* **by** (*rule map-sum.id*)
next
fix *f1* :: *'o ⇒ 's* **and** *f2* :: *'p ⇒ 't* **and** *g1* :: *'s ⇒ 'q* **and** *g2* :: *'t ⇒ 'r*
show *map-sum (g1 ∘ f1) (g2 ∘ f2) = map-sum g1 g2 ∘ map-sum f1 f2*
by (*rule map-sum.comp*[*symmetric*])
next
fix *x* **and** *f1* :: *'o ⇒ 'q* **and** *f2* :: *'p ⇒ 'r* **and** *g1 g2*
assume *a1*: *⋀z. z ∈ setl x ⟹ f1 z = g1 z* **and**
a2: *⋀z. z ∈ setr x ⟹ f2 z = g2 z*
thus *map-sum f1 f2 x = map-sum g1 g2 x*
proof (*cases x*)
case *Inl* **thus** *?thesis* **using** *a1* **by** (*clarsimp simp*: *sum-set-defs*(*1*))
next
case *Inr* **thus** *?thesis* **using** *a2* **by** (*clarsimp simp*: *sum-set-defs*(*2*))

qed
next
fix *f1* :: *'o* ⇒ *'q* **and** *f2* :: *'p* ⇒ *'r*
show *setl* ∘ *map-sum f1 f2* = *image f1* ∘ *setl*
by (*rule ext*, *unfold o-apply*) (*simp add*: *sum-set-defs*(*1*) *split*: *sum.split*)
next
fix *f1* :: *'o* ⇒ *'q* **and** *f2* :: *'p* ⇒ *'r*
show *setr* ∘ *map-sum f1 f2* = *image f2* ∘ *setr*
by (*rule ext*, *unfold o-apply*) (*simp add*: *sum-set-defs*(*2*) *split*: *sum.split*)
next
show *card-order natLeq* **by** (*rule natLeq-card-order*)
next
show *cinfinite natLeq* **by** (*rule natLeq-cinfinite*)
next
show *regularCard natLeq* **by** (*rule regularCard-natLeq*)
next
fix *x* :: *'o* + *'p*
show |*setl x*| <*o natLeq*
apply (*rule finite-iff-ordLess-natLeq*[*THEN iffD1*])
by (*simp add*: *sum-set-defs*(*1*) *split*: *sum.split*)
next
fix *x* :: *'o* + *'p*
show |*setr x*| <*o natLeq*
apply (*rule finite-iff-ordLess-natLeq*[*THEN iffD1*])
by (*simp add*: *sum-set-defs*(*2*) *split*: *sum.split*)
next
fix *R1 R2 S1 S2*
show *rel-sum R1 R2 OO rel-sum S1 S2* ≤ *rel-sum* (*R1 OO S1*) (*R2 OO S2*)
by (*force elim*: *rel-sum.cases*)
next
fix *R S*
show *rel-sum R S* = (λ*x y*.
∃*z*. (*setl z* ⊆ {(*x*, *y*). *R x y*} ∧ *setr z* ⊆ {(*x*, *y*). *S x y*}) ∧
map-sum fst fst z = *x* ∧ *map-sum snd snd z* = *y*)
unfolding *sum-set-defs relcompp.simps conversep.simps fun-eq-iff*
by (*fastforce elim*: *rel-sum.cases split*: *sum.splits*)
qed (*auto simp*: *sum-set-defs fun-eq-iff pred-sum.simps split*: *sum.splits*)

inductive-set *fsts* :: *'a* × *'b* ⇒ *'a set* **for** *p* :: *'a* × *'b* **where**
fst p ∈ *fsts p*
inductive-set *snds* :: *'a* × *'b* ⇒ *'b set* **for** *p* :: *'a* × *'b* **where**
snd p ∈ *snds p*

lemma *prod-set-defs*[*code*]: *fsts* = (λ*p*. {*fst p*}) *snds* = (λ*p*. {*snd p*})
by (*auto intro*: *fsts.intros snds.intros elim*: *fsts.cases snds.cases*)

inductive
rel-prod :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'c* ⇒ *'d* ⇒ *bool*) ⇒ *'a* × *'c* ⇒ *'b* × *'d* ⇒ *bool*
for *R1 R2*

where
  $[\![R1\ a\ b;\ R2\ c\ d]\!] \Longrightarrow$ *rel-prod R1 R2* $(a,\ c)\ (b,\ d)$

inductive
  pred-prod :: $('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'a \times 'b \Rightarrow bool$ **for** *P1 P2*
where
  $[\![P1\ a;\ P2\ b]\!] \Longrightarrow$ *pred-prod P1 P2* $(a,\ b)$

lemma *rel-prod-inject* [*code*, *simp*]:
  rel-prod R1 R2 $(a,\ b)\ (c,\ d) \longleftrightarrow$ *R1 a c* $\wedge$ *R2 b d*
  by (*auto intro*: *rel-prod.intros elim*: *rel-prod.cases*)

lemma *pred-prod-inject* [*code*, *simp*]:
  pred-prod P1 P2 $(a,\ b) \longleftrightarrow$ *P1 a* $\wedge$ *P2 b*
  by (*auto intro*: *pred-prod.intros elim*: *pred-prod.cases*)

lemma *rel-prod-conv*:
  rel-prod R1 R2 $= (\lambda(a,\ b)\ (c,\ d).$ *R1 a c* $\wedge$ *R2 b d*$)$
  by *force*

definition
  pred-fun :: $('a \Rightarrow bool) \Rightarrow ('b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where
  pred-fun A B $= (\lambda f.\ \forall x.\ A\ x \longrightarrow B\ (f\ x))$

lemma *pred-funI*: $(\bigwedge x.\ A\ x \Longrightarrow B\ (f\ x)) \Longrightarrow$ *pred-fun A B f*
  unfolding *pred-fun-def* **by** *simp*

bnf $'a \times 'b$
  map: *map-prod*
  sets: *fsts snds*
  bd: *natLeq*
  rel: *rel-prod*
  pred: *pred-prod*
proof (*unfold prod-set-defs*)
  show *map-prod id id = id* **by** (*rule map-prod.id*)
next
  fix *f1 f2 g1 g2*
  show *map-prod* $(g1 \circ f1)\ (g2 \circ f2) =$ *map-prod g1 g2* $\circ$ *map-prod f1 f2*
    by (*rule map-prod.comp*[*symmetric*])
next
  fix *x f1 f2 g1 g2*
  assume $\bigwedge z.\ z \in \{fst\ x\} \Longrightarrow f1\ z = g1\ z\ \bigwedge z.\ z \in \{snd\ x\} \Longrightarrow f2\ z = g2\ z$
  thus *map-prod f1 f2 x = map-prod g1 g2 x* **by** (*cases x*) *simp*
next
  fix *f1 f2*
  show $(\lambda x.\ \{fst\ x\}) \circ$ *map-prod f1 f2 = image f1* $\circ\ (\lambda x.\ \{fst\ x\})$
    by (*rule ext*, *unfold o-apply*) *simp*
next

```
  fix f1 f2
  show (λx. {snd x}) ∘ map-prod f1 f2 = image f2 ∘ (λx. {snd x})
    by (rule ext, unfold o-apply) simp
next
  show card-order natLeq by (rule natLeq-card-order)
next
  show cinfinite natLeq by (rule natLeq-cinfinite)
next
  show regularCard natLeq by (rule regularCard-natLeq)
next
  fix x
  show |{fst x}| <o natLeq
    by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
  fix x
  show |{snd x}| <o natLeq
    by (simp add: finite-iff-ordLess-natLeq[symmetric])
next
  fix R1 R2 S1 S2
  show rel-prod R1 R2 OO rel-prod S1 S2 ≤ rel-prod (R1 OO S1) (R2 OO S2)
by auto
next
  fix R S
  show rel-prod R S = (λx y.
    ∃z. ({fst z} ⊆ {(x, y). R x y} ∧ {snd z} ⊆ {(x, y). S x y}) ∧
      map-prod fst fst z = x ∧ map-prod snd snd z = y)
  unfolding prod-set-defs rel-prod-inject relcompp.simps conversep.simps fun-eq-iff
  by auto
qed auto

lemma card-order-bd-fun: card-order (natLeq +c card-suc ( |UNIV| ))
 by (auto simp: card-order-csum natLeq-card-order card-order-card-suc card-of-card-order-on)

lemma Cinfinite-bd-fun: Cinfinite (natLeq +c card-suc ( |UNIV| ))
  by (auto simp: Cinfinite-csum natLeq-Cinfinite)

lemma regularCard-bd-fun: regularCard (natLeq +c card-suc ( |UNIV| ))
  (is regularCard (- +c card-suc ?U))
proof (cases Cinfinite ?U)
  case True
  then show ?thesis
    by (intro regularCard-csum natLeq-Cinfinite Cinfinite-card-suc
            card-of-card-order-on regularCard-natLeq regularCard-card-suc)
next
  case False
  then have card-suc ?U ≤o natLeq
    unfolding cinfinite-def Field-card-of
    by (intro card-suc-least;
      simp add: natLeq-Card-order card-of-card-order-on flip: finite-iff-ordLess-natLeq)
```

```
  then have natLeq =o natLeq +c card-suc ?U
    using natLeq-Cinfinite csum-absorb1 ordIso-symmetric by blast
  then show ?thesis
    by (intro regularCard-ordIso[OF - natLeq-Cinfinite regularCard-natLeq])
qed

lemma ordLess-bd-fun: |UNIV::'a set| <o natLeq +c card-suc ( |UNIV::'a set| )
  (is - <o (- +c card-suc (?U :: 'a rel)))
proof (cases Cinfinite ?U)
  case True
 have ?U <o card-suc ?U using card-of-card-order-on natLeq-card-order card-suc-greater
by blast
  also have card-suc ?U =o natLeq +c card-suc ?U by (rule csum-absorb2[THEN
ordIso-symmetric])
   (auto simp: True card-of-card-order-on intro!: Cinfinite-card-suc natLeq-ordLeq-cinfinite)
  finally show ?thesis .
next
  case False
  then have ?U <o natLeq
   by (auto simp: cinfinite-def Field-card-of card-of-card-order-on finite-iff-ordLess-natLeq[symmetric])
  then show ?thesis
    by (rule ordLess-ordLeq-trans[OF - ordLeq-csum1[OF natLeq-Card-order]])
qed

bnf 'a ⇒ 'b
  map: (∘)
  sets: range
  bd: natLeq +c card-suc ( |UNIV::'a set| )
  rel: rel-fun (=)
  pred: pred-fun (λ-. True)
proof
  fix f show id ∘ f = id f by simp
next
  fix f g show (∘) (g ∘ f) = (∘) g ∘ (∘) f
  unfolding comp-def[abs-def] ..
next
  fix x f g
  assume ⋀z. z ∈ range x ⟹ f z = g z
  thus f ∘ x = g ∘ x by auto
next
  fix f show range ∘ (∘) f = (`) f ∘ range
    by (auto simp add: fun-eq-iff)
next
  show card-order (natLeq +c card-suc ( |UNIV| ))
    by (rule card-order-bd-fun)
next
  show cinfinite (natLeq +c card-suc ( |UNIV| ))
    by (rule Cinfinite-bd-fun[THEN conjunct1])
next
```

show *regularCard* (*natLeq* +*c* *card-suc* (|*UNIV*|))
by (*rule regularCard-bd-fun*)
next
fix *f* :: *'d* ⇒ *'a*
show |*range f*| <*o* *natLeq* +*c* *card-suc* |*UNIV* :: *'d set*|
by (*rule ordLeq-ordLess-trans*[*OF card-of-image ordLess-bd-fun*])
next
fix *R S*
show *rel-fun* (=) *R OO rel-fun* (=) *S* ≤ *rel-fun* (=) (*R OO S*) **by** (*auto simp*: *rel-fun-def*)
next
fix *R*
show *rel-fun* (=) *R* = (λ*x y*.
∃ *z*. *range z* ⊆ {(*x*, *y*). *R x y*} ∧ *fst* ∘ *z* = *x* ∧ *snd* ∘ *z* = *y*)
unfolding *rel-fun-def subset-iff* **by** (*force simp*: *fun-eq-iff*[*symmetric*])
qed (*auto simp*: *pred-fun-def*)

end

36 Shared Fixpoint Operations on Bounded Natural Functors

theory *BNF-Fixpoint-Base*
imports *BNF-Composition Basic-BNFs*
begin

lemma *conj-imp-eq-imp-imp*: (*P* ∧ *Q* ⟹ *PROP R*) ≡ (*P* ⟹ *Q* ⟹ *PROP R*)
by *standard simp-all*

lemma *predicate2D-conj*: *P* ≤ *Q* ∧ *R* ⟹ *R* ∧ (*P x y* ⟶ *Q x y*)
by *blast*

lemma *eq-sym-Unity-conv*: (*x* = ((() = ()))) = *x*
by *blast*

lemma *case-unit-Unity*: (*case u of* () ⇒ *f*) = *f*
by (*cases u*) (*hypsubst*, *rule unit.case*)

lemma *case-prod-Pair-iden*: (*case p of* (*x*, *y*) ⇒ (*x*, *y*)) = *p*
by *simp*

lemma *unit-all-impI*: (*P* () ⟹ *Q* ()) ⟹ ∀ *x*. *P x* ⟶ *Q x*
by *simp*

lemma *pointfree-idE*: *f* ∘ *g* = *id* ⟹ *f* (*g x*) = *x*
unfolding *comp-def fun-eq-iff* **by** *simp*

lemma *o-bij*:

assumes *gf*: $g \circ f = id$ **and** *fg*: $f \circ g = id$
shows *bij f*
unfolding *bij-def inj-on-def surj-def* **proof** *safe*
fix *a1 a2* **assume** *f a1* = *f a2*
hence *g* (*f a1*) = *g* (*f a2*) **by** *simp*
thus *a1* = *a2* **using** *gf* **unfolding** *fun-eq-iff* **by** *simp*
next
fix *b*
have *b* = *f* (*g b*)
using *fg* **unfolding** *fun-eq-iff* **by** *simp*
thus $\exists$ *a*. *b* = *f a* **by** *blast*
qed

lemma *case-sum-step*:
case-sum (*case-sum f′ g′*) *g* (*Inl p*) = *case-sum f′ g′ p*
case-sum f (*case-sum f′ g′*) (*Inr p*) = *case-sum f′ g′ p*
by *auto*

lemma *obj-one-pointE*: $\forall$ *x*. *s* = *x* $\longrightarrow$ *P* $\Longrightarrow$ *P*
by *blast*

lemma *type-copy-obj-one-point-absE*:
assumes *type-definition Rep Abs UNIV* $\forall$ *x*. *s* = *Abs x* $\longrightarrow$ *P* **shows** *P*
using *type-definition.Rep-inverse*[*OF assms*(*1*)]
by (*intro mp*[*OF spec*[*OF assms*(*2*), *of Rep s*]]) *simp*

lemma *obj-sumE-f*:
assumes $\forall$ *x*. *s* = *f* (*Inl x*) $\longrightarrow$ *P* $\forall$ *x*. *s* = *f* (*Inr x*) $\longrightarrow$ *P*
shows $\forall$ *x*. *s* = *f x* $\longrightarrow$ *P*
proof
fix *x* **from** *assms* **show** *s* = *f x* $\longrightarrow$ *P* **by** (*cases x*) *auto*
qed

lemma *case-sum-if*:
case-sum f g (*if p then Inl x else Inr y*) = (*if p then f x else g y*)
by *simp*

lemma *prod-set-simps*[*simp*]:
fsts (*x*, *y*) = {*x*}
snds (*x*, *y*) = {*y*}
unfolding *prod-set-defs* **by** *simp*+

lemma *sum-set-simps*[*simp*]:
setl (*Inl x*) = {*x*}
setl (*Inr x*) = {}
setr (*Inl x*) = {}
setr (*Inr x*) = {*x*}
unfolding *sum-set-defs* **by** *simp*+

lemma *Inl-Inr-False*: $(Inl\ x = Inr\ y) = False$
by *simp*

lemma *Inr-Inl-False*: $(Inr\ x = Inl\ y) = False$
by *simp*

lemma *spec2*: $\forall\, x\ y.\ P\ x\ y \Longrightarrow P\ x\ y$
by *blast*

lemma *rewriteR-comp-comp*: $[\![g \circ h = r]\!] \Longrightarrow f \circ g \circ h = f \circ r$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteR-comp-comp2*: $[\![g \circ h = r1 \circ r2;\ f \circ r1 = l]\!] \Longrightarrow f \circ g \circ h = l \circ r2$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteL-comp-comp*: $[\![f \circ g = l]\!] \Longrightarrow f \circ (g \circ h) = l \circ h$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *rewriteL-comp-comp2*: $[\![f \circ g = l1 \circ l2;\ l2 \circ h = r]\!] \Longrightarrow f \circ (g \circ h) = l1 \circ r$
unfolding *comp-def fun-eq-iff* **by** *auto*

lemma *convol-o*: $\langle f,\ g\rangle \circ h = \langle f \circ h,\ g \circ h\rangle$
unfolding *convol-def* **by** *auto*

lemma *map-prod-o-convol*: $map\text{-}prod\ h1\ h2 \circ \langle f,\ g\rangle = \langle h1 \circ f,\ h2 \circ g\rangle$
unfolding *convol-def* **by** *auto*

lemma *map-prod-o-convol-id*: $(map\text{-}prod\ f\ id \circ \langle id,\ g\rangle)\ x = \langle id \circ f,\ g\rangle\ x$
unfolding *map-prod-o-convol id-comp comp-id* **..**

lemma *o-case-sum*: $h \circ case\text{-}sum\ f\ g = case\text{-}sum\ (h \circ f)\ (h \circ g)$
unfolding *comp-def* **by** (*auto split*: *sum.splits*)

lemma *case-sum-o-map-sum*: $case\text{-}sum\ f\ g \circ map\text{-}sum\ h1\ h2 = case\text{-}sum\ (f \circ h1)\ (g \circ h2)$
unfolding *comp-def* **by** (*auto split*: *sum.splits*)

lemma *case-sum-o-map-sum-id*: $(case\text{-}sum\ id\ g \circ map\text{-}sum\ f\ id)\ x = case\text{-}sum\ (f \circ id)\ g\ x$
unfolding *case-sum-o-map-sum id-comp comp-id* **..**

lemma *rel-fun-def-butlast*:
$rel\text{-}fun\ R\ (rel\text{-}fun\ S\ T)\ f\ g = (\forall\, x\ y.\ R\ x\ y \longrightarrow (rel\text{-}fun\ S\ T)\ (f\ x)\ (g\ y))$
unfolding *rel-fun-def* **..**

lemma *subst-eq-imp*: $(\forall\, a\ b.\ a = b \longrightarrow P\ a\ b) \equiv (\forall\, a.\ P\ a\ a)$
by *auto*

lemma *eq-subset*: $(=) \leq (\lambda a\ b.\ P\ a\ b \vee a = b)$
by *auto*

lemma *eq-le-Grp-id-iff*: $((=) \leq Grp\ (Collect\ R)\ id) = (All\ R)$
unfolding *Grp-def id-apply* **by** *blast*

lemma *Grp-id-mono-subst*: $(\bigwedge x\ y.\ Grp\ P\ id\ x\ y \Longrightarrow Grp\ Q\ id\ (f\ x)\ (f\ y)) \equiv (\bigwedge x.\ x \in P \Longrightarrow f\ x \in Q)$
unfolding *Grp-def* **by** *rule auto*

lemma *vimage2p-mono*: $vimage2p\ f\ g\ R\ x\ y \Longrightarrow R \leq S \Longrightarrow vimage2p\ f\ g\ S\ x\ y$
unfolding *vimage2p-def* **by** *blast*

lemma *vimage2p-refl*: $(\bigwedge x.\ R\ x\ x) \Longrightarrow vimage2p\ f\ f\ R\ x\ x$
unfolding *vimage2p-def* **by** *auto*

lemma
assumes *type-definition Rep Abs UNIV*
shows *type-copy-Rep-o-Abs*: $Rep \circ Abs = id$ **and** *type-copy-Abs-o-Rep*: $Abs \circ Rep = id$
unfolding *fun-eq-iff comp-apply id-apply*
type-definition.Abs-inverse[*OF assms UNIV-I*] *type-definition.Rep-inverse*[*OF assms*] **by** *simp-all*

lemma *type-copy-map-comp0-undo*:
assumes *type-definition Rep Abs UNIV*
type-definition Rep' Abs' UNIV
type-definition Rep'' Abs'' UNIV
shows $Abs' \circ M \circ Rep'' = (Abs' \circ M1 \circ Rep) \circ (Abs \circ M2 \circ Rep'') \Longrightarrow M1 \circ M2 = M$
by (*rule sym*) (*auto simp*: *fun-eq-iff type-definition.Abs-inject*[*OF assms*(2) *UNIV-I UNIV-I*]
type-definition.Abs-inverse[*OF assms*(1) *UNIV-I*]
type-definition.Abs-inverse[*OF assms*(3) *UNIV-I*] *dest*: *spec*[*of - Abs'' x* **for** *x*])

lemma *vimage2p-id*: $vimage2p\ id\ id\ R = R$
unfolding *vimage2p-def* **by** *auto*

lemma *vimage2p-comp*: $vimage2p\ (f1 \circ f2)\ (g1 \circ g2) = vimage2p\ f2\ g2 \circ vimage2p\ f1\ g1$
unfolding *fun-eq-iff vimage2p-def o-apply* **by** *simp*

lemma *vimage2p-rel-fun*: $rel\text{-}fun\ (vimage2p\ f\ g\ R)\ R\ f\ g$
unfolding *rel-fun-def vimage2p-def* **by** *auto*

lemma *fun-cong-unused-0*: $f = (\lambda x.\ g) \Longrightarrow f\ (\lambda x.\ 0) = g$
by (*erule arg-cong*)

lemma *inj-on-convol-ident*: *inj-on* (λ*x*. (*x*, *f x*)) *X*
unfolding *inj-on-def* **by** *simp*

lemma *map-sum-if-distrib-then*:
⋀*f g e x y*. *map-sum f g* (*if e then Inl x else y*) = (*if e then Inl* (*f x*) *else map-sum f g y*)
⋀*f g e x y*. *map-sum f g* (*if e then Inr x else y*) = (*if e then Inr* (*g x*) *else map-sum f g y*)
by *simp-all*

lemma *map-sum-if-distrib-else*:
⋀*f g e x y*. *map-sum f g* (*if e then x else Inl y*) = (*if e then map-sum f g x else Inl* (*f y*))
⋀*f g e x y*. *map-sum f g* (*if e then x else Inr y*) = (*if e then map-sum f g x else Inr* (*g y*))
by *simp-all*

lemma *case-prod-app*: *case-prod f x y* = *case-prod* (λ*l r*. *f l r y*) *x*
by (*cases x*) *simp*

lemma *case-sum-map-sum*: *case-sum l r* (*map-sum f g x*) = *case-sum* (*l* ∘ *f*) (*r* ∘ *g*) *x*
by (*cases x*) *simp-all*

lemma *case-sum-transfer*:
rel-fun (*rel-fun R T*) (*rel-fun* (*rel-fun S T*) (*rel-fun* (*rel-sum R S*) *T*)) *case-sum case-sum*
unfolding *rel-fun-def* **by** (*auto split*: *sum.splits*)

lemma *case-prod-map-prod*: *case-prod h* (*map-prod f g x*) = *case-prod* (λ*l r*. *h* (*f l*) (*g r*)) *x*
by (*cases x*) *simp-all*

lemma *case-prod-o-map-prod*: *case-prod f* ∘ *map-prod g1 g2* = *case-prod* (λ*l r*. *f* (*g1 l*) (*g2 r*))
unfolding *comp-def* **by** *auto*

lemma *case-prod-transfer*:
(*rel-fun* (*rel-fun A* (*rel-fun B C*)) (*rel-fun* (*rel-prod A B*) *C*)) *case-prod case-prod*
unfolding *rel-fun-def* **by** *simp*

lemma *eq-ifI*: (*P* ⟶ *t* = *u1*) ⟹ (¬ *P* ⟶ *t* = *u2*) ⟹ *t* = (*if P then u1 else u2*)
by *simp*

lemma *comp-transfer*:
rel-fun (*rel-fun B C*) (*rel-fun* (*rel-fun A B*) (*rel-fun A C*)) (∘) (∘)
unfolding *rel-fun-def* **by** *simp*

lemma *If-transfer*: *rel-fun* (=) (*rel-fun A* (*rel-fun A A*)) *If If*
unfolding *rel-fun-def* **by** *simp*

lemma *Abs-transfer*:
assumes *type-copy1*: *type-definition Rep1 Abs1 UNIV*
assumes *type-copy2*: *type-definition Rep2 Abs2 UNIV*
shows *rel-fun R* (*vimage2p Rep1 Rep2 R*) *Abs1 Abs2*
unfolding *vimage2p-def rel-fun-def*
type-definition.Abs-inverse[*OF type-copy1 UNIV-I*]
type-definition.Abs-inverse[*OF type-copy2 UNIV-I*] **by** *simp*

lemma *Inl-transfer*:
rel-fun S (*rel-sum S T*) *Inl Inl*
by *auto*

lemma *Inr-transfer*:
rel-fun T (*rel-sum S T*) *Inr Inr*
by *auto*

lemma *Pair-transfer*: *rel-fun A* (*rel-fun B* (*rel-prod A B*)) *Pair Pair*
unfolding *rel-fun-def* **by** *simp*

lemma *eq-onp-live-step*: $x = y \Longrightarrow$ *eq-onp P a a* $\wedge$ $x \longleftrightarrow$ *P a* $\wedge$ y
by (*simp only*: *eq-onp-same-args*)

lemma *top-conj*: *top x* $\wedge$ *P* $\longleftrightarrow$ *P P* $\wedge$ *top x* $\longleftrightarrow$ *P*
by *blast*+

lemma *fst-convol'*: *fst* ($\langle f,\ g\rangle$ *x*) = *f x*
using *fst-convol* **unfolding** *convol-def* **by** *simp*

lemma *snd-convol'*: *snd* ($\langle f,\ g\rangle$ *x*) = *g x*
using *snd-convol* **unfolding** *convol-def* **by** *simp*

lemma *convol-expand-snd*: *fst* $\circ$ *f* = *g* $\Longrightarrow$ $\langle g,\ snd \circ f\rangle$ = *f*
unfolding *convol-def* **by** *auto*

lemma *convol-expand-snd'*:
assumes (*fst* $\circ$ *f* = *g*)
shows *h* = *snd* $\circ$ *f* $\longleftrightarrow$ $\langle g,\ h\rangle$ = *f*
proof −
from *assms* **have** $*$: $\langle g,\ snd \circ f\rangle$ = *f* **by** (*rule convol-expand-snd*)
then have *h* = *snd* $\circ$ *f* $\longleftrightarrow$ *h* = *snd* $\circ$ $\langle g,\ snd \circ f\rangle$ **by** *simp*
moreover have ... $\longleftrightarrow$ *h* = *snd* $\circ$ *f* **by** (*simp add*: *snd-convol*)
moreover have ... $\longleftrightarrow$ $\langle g,\ h\rangle$ = *f* **by** (*subst* (*2*) $*$[*symmetric*]) (*auto simp*: *convol-def fun-eq-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma *case-sum-expand-Inr-pointfree*: $f \circ Inl = g \Longrightarrow case\text{-}sum\ g\ (f \circ Inr) = f$
by (*auto split*: *sum.splits*)

lemma *case-sum-expand-Inr'*: $f \circ Inl = g \Longrightarrow h = f \circ Inr \longleftrightarrow case\text{-}sum\ g\ h = f$
by (*rule iffI*) (*auto simp add*: *fun-eq-iff split*: *sum.splits*)

lemma *case-sum-expand-Inr*: $f \circ Inl = g \Longrightarrow f\ x = case\text{-}sum\ g\ (f \circ Inr)\ x$
by (*auto split*: *sum.splits*)

lemma *id-transfer*: *rel-fun A A id id*
unfolding *rel-fun-def* **by** *simp*

lemma *fst-transfer*: *rel-fun* (*rel-prod A B*) *A fst fst*
unfolding *rel-fun-def* **by** *simp*

lemma *snd-transfer*: *rel-fun* (*rel-prod A B*) *B snd snd*
unfolding *rel-fun-def* **by** *simp*

lemma *convol-transfer*:
rel-fun (*rel-fun R S*) (*rel-fun* (*rel-fun R T*) (*rel-fun R* (*rel-prod S T*))) *BNF-Def.convol BNF-Def.convol*
unfolding *rel-fun-def convol-def* **by** *auto*

lemma *Let-const*: *Let x* (λ-. *c*) = *c*
unfolding *Let-def* **..**

ML-file ‹*Tools/BNF/bnf-fp-util-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-fp-util.ML*›
ML-file ‹*Tools/BNF/bnf-fp-def-sugar-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-fp-def-sugar.ML*›
ML-file ‹*Tools/BNF/bnf-fp-n2m-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-fp-n2m.ML*›
ML-file ‹*Tools/BNF/bnf-fp-n2m-sugar.ML*›

end

37 Least Fixpoint (Datatype) Operation on Bounded Natural Functors

theory *BNF-Least-Fixpoint*
imports *BNF-Fixpoint-Base*
keywords
datatype :: *thy-defn* **and**
datatype-compat :: *thy-defn*
begin

lemma *subset-emptyI*: $(\bigwedge x.\ x \in A \Longrightarrow False) \Longrightarrow A \subseteq \{\}$
by *blast*

lemma *image-Collect-subsetI*: $(\bigwedge x.\ P\ x \Longrightarrow f\ x \in B) \Longrightarrow f\ `\ \{x.\ P\ x\} \subseteq B$
by *blast*

lemma *Collect-restrict*: $\{x.\ x \in X \wedge P\ x\} \subseteq X$
by *auto*

lemma *prop-restrict*: $[\![x \in Z;\ Z \subseteq \{x.\ x \in X \wedge P\ x\}]\!] \Longrightarrow P\ x$
by *auto*

lemma *underS-I*: $[\![i \neq j;\ (i, j) \in R]\!] \Longrightarrow i \in underS\ R\ j$
unfolding *underS-def* **by** *simp*

lemma *underS-E*: $i \in underS\ R\ j \Longrightarrow i \neq j \wedge (i, j) \in R$
unfolding *underS-def* **by** *simp*

lemma *underS-Field*: $i \in underS\ R\ j \Longrightarrow i \in Field\ R$
unfolding *underS-def Field-def* **by** *auto*

lemma *ex-bij-betw*: $|A| \leq o\ (r :: 'b\ rel) \Longrightarrow \exists f\ B :: 'b\ set.\ bij\text{-}betw\ f\ B\ A$
by (*subst* (*asm*) *internalize-card-of-ordLeq*) (*auto dest*!: *iffD2*[*OF card-of-ordIso ordIso-symmetric*])

lemma *bij-betwI'*:
$[\![\bigwedge x\ y.\ [\![x \in X;\ y \in X]\!] \Longrightarrow (f\ x = f\ y) = (x = y);$
$\bigwedge x.\ x \in X \Longrightarrow f\ x \in Y;$
$\bigwedge y.\ y \in Y \Longrightarrow \exists x \in X.\ y = f\ x]\!] \Longrightarrow bij\text{-}betw\ f\ X\ Y$
unfolding *bij-betw-def inj-on-def* **by** *blast*

lemma *surj-fun-eq*:
assumes *surj-on*: $f\ `\ X = UNIV$ **and** *eq-on*: $\forall x \in X.\ (g1 \circ f)\ x = (g2 \circ f)\ x$
shows $g1 = g2$
proof (*rule ext*)
fix y
from *surj-on* **obtain** x **where** $x \in X$ **and** $y = f\ x$ **by** *blast*
thus $g1\ y = g2\ y$ **using** *eq-on* **by** *simp*
qed

lemma *Card-order-wo-rel*: $Card\text{-}order\ r \Longrightarrow wo\text{-}rel\ r$
unfolding *wo-rel-def card-order-on-def* **by** *blast*

lemma *Cinfinite-limit*: $[\![x \in Field\ r;\ Cinfinite\ r]\!] \Longrightarrow \exists y \in Field\ r.\ x \neq y \wedge (x, y) \in r$
unfolding *cinfinite-def* **by** (*auto simp add*: *infinite-Card-order-limit*)

lemma *Card-order-trans*:
$[\![Card\text{-}order\ r;\ x \neq y;\ (x, y) \in r;\ y \neq z;\ (y, z) \in r]\!] \Longrightarrow x \neq z \wedge (x, z) \in r$
unfolding *card-order-on-def well-order-on-def linear-order-on-def*
partial-order-on-def preorder-on-def trans-def antisym-def **by** *blast*

lemma *Cinfinite-limit2*:
assumes *x1*: $x1 \in Field\ r$ **and** *x2*: $x2 \in Field\ r$ **and** *r*: *Cinfinite r*
shows $\exists y \in Field\ r.\ (x1 \neq y \wedge (x1, y) \in r) \wedge (x2 \neq y \wedge (x2, y) \in r)$
proof −
from *r* **have** *trans*: *trans r* **and** *total*: *Total r* **and** *antisym*: *antisym r*
unfolding *card-order-on-def well-order-on-def linear-order-on-def partial-order-on-def preorder-on-def* **by** *auto*
obtain *y1* **where** *y1*: $y1 \in Field\ r\ x1 \neq y1\ (x1, y1) \in r$
using *Cinfinite-limit*[*OF x1 r*] **by** *blast*
obtain *y2* **where** *y2*: $y2 \in Field\ r\ x2 \neq y2\ (x2, y2) \in r$
using *Cinfinite-limit*[*OF x2 r*] **by** *blast*
show *?thesis*
proof (*cases y1 = y2*)
case *True* **with** *y1 y2* **show** *?thesis* **by** *blast*
next
case *False*
with *y1*(*1*) *y2*(*1*) *total* **have** $(y1, y2) \in r \vee (y2, y1) \in r$
unfolding *total-on-def* **by** *auto*
thus *?thesis*
proof
assume ∗: $(y1, y2) \in r$
with *trans y1*(*3*) **have** $(x1, y2) \in r$ **unfolding** *trans-def* **by** *blast*
with *False y1 y2* ∗ *antisym* **show** *?thesis* **by** (*cases x1 = y2*) (*auto simp*: *antisym-def*)
next
assume ∗: $(y2, y1) \in r$
with *trans y2*(*3*) **have** $(x2, y1) \in r$ **unfolding** *trans-def* **by** *blast*
with *False y1 y2* ∗ *antisym* **show** *?thesis* **by** (*cases x2 = y1*) (*auto simp*: *antisym-def*)
qed
qed
qed

lemma *Cinfinite-limit-finite*:
⟦*finite X*; $X \subseteq Field\ r$; *Cinfinite r*⟧ ⟹ $\exists y \in Field\ r.\ \forall x \in X.\ (x \neq y \wedge (x, y) \in r)$
proof (*induct X rule*: *finite-induct*)
case *empty* **thus** *?case* **unfolding** *cinfinite-def* **using** *ex-in-conv*[*of Field r*] *finite.emptyI* **by** *auto*
next
case (*insert x X*)
then obtain *y* **where** *y*: $y \in Field\ r\ \forall x \in X.\ (x \neq y \wedge (x, y) \in r)$ **by** *blast*
then obtain *z* **where** *z*: $z \in Field\ r\ x \neq z \wedge (x, z) \in r\ y \neq z \wedge (y, z) \in r$
using *Cinfinite-limit2*[*OF - y*(*1*) *insert*(*5*), *of x*] *insert*(*4*) **by** *blast*
show *?case*
apply (*intro bexI ballI*)
apply (*erule insertE*)
apply *hypsubst*

apply (*rule z*(*2*))
using *Card-order-trans*[*OF insert*(*5*)[*THEN conjunct2*]] *y*(*2*) *z*(*3*)
apply *blast*
apply (*rule z*(*1*))
done
qed

lemma *insert-subsetI*: ⟦*x* ∈ *A*; *X* ⊆ *A*⟧ ⟹ *insert x X* ⊆ *A*
by *auto*

lemmas *well-order-induct-imp* = *wo-rel.well-order-induct*[*of r* λ*x*. *x* ∈ *Field r* ⟶ *P x* **for** *r P*]

lemma *meta-spec2*:
assumes (⋀*x y*. *PROP P x y*)
shows *PROP P x y*
by (*rule assms*)

lemma *nchotomy-relcomppE*:
assumes ⋀*y*. ∃ *x*. *y* = *f x* (*r OO s*) *a c* ⋀*b*. *r a* (*f b*) ⟹ *s* (*f b*) *c* ⟹ *P*
shows *P*
proof (*rule relcompp.cases*[*OF assms*(*2*)], *hypsubst*)
fix *b* **assume** *r a b s b c*
moreover from *assms*(*1*) **obtain** *b*′ **where** *b* = *f b*′ **by** *blast*
ultimately show *P* **by** (*blast intro*: *assms*(*3*))
qed

lemma *predicate2D-vimage2p*: ⟦*R* ≤ *vimage2p f g S*; *R x y*⟧ ⟹ *S* (*f x*) (*g y*)
unfolding *vimage2p-def* **by** *auto*

lemma *ssubst-Pair-rhs*: ⟦(*r*, *s*) ∈ *R*; *s*′ = *s*⟧ ⟹ (*r*, *s*′) ∈ *R*
by (*rule ssubst*)

lemma *all-mem-range1*:
(⋀*y*. *y* ∈ *range f* ⟹ *P y*) ≡ (⋀*x*. *P* (*f x*))
by (*rule equal-intr-rule*) *fast*+

lemma *all-mem-range2*:
(⋀*fa y*. *fa* ∈ *range f* ⟹ *y* ∈ *range fa* ⟹ *P y*) ≡ (⋀*x xa*. *P* (*f x xa*))
by (*rule equal-intr-rule*) *fast*+

lemma *all-mem-range3*:
(⋀*fa fb y*. *fa* ∈ *range f* ⟹ *fb* ∈ *range fa* ⟹ *y* ∈ *range fb* ⟹ *P y*) ≡ (⋀*x xa xb*. *P* (*f x xa xb*))
by (*rule equal-intr-rule*) *fast*+

lemma *all-mem-range4*:
(⋀*fa fb fc y*. *fa* ∈ *range f* ⟹ *fb* ∈ *range fa* ⟹ *fc* ∈ *range fb* ⟹ *y* ∈ *range fc* ⟹ *P y*) ≡

$(\bigwedge x\ xa\ xb\ xc.\ P\ (f\ x\ xa\ xb\ xc))$
by (*rule equal-intr-rule*) *fast+*

lemma *all-mem-range5*:
$(\bigwedge fa\ fb\ fc\ fd\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$y \in range\ fd \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd.\ P\ (f\ x\ xa\ xb\ xc\ xd))$
by (*rule equal-intr-rule*) *fast+*

lemma *all-mem-range6*:
$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow y \in range\ ff \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf))$
by (*rule equal-intr-rule*) (*fastforce*, *fast*)

lemma *all-mem-range7*:
$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow fg \in range\ ff \Longrightarrow y \in range\ fg \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg))$
by (*rule equal-intr-rule*) (*fastforce*, *fast*)

lemma *all-mem-range8*:
$(\bigwedge fa\ fb\ fc\ fd\ fe\ ff\ fg\ fh\ y.\ fa \in range\ f \Longrightarrow fb \in range\ fa \Longrightarrow fc \in range\ fb \Longrightarrow fd \in range\ fc \Longrightarrow$
$fe \in range\ fd \Longrightarrow ff \in range\ fe \Longrightarrow fg \in range\ ff \Longrightarrow fh \in range\ fg \Longrightarrow y \in range\ fh \Longrightarrow P\ y) \equiv$
$(\bigwedge x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh.\ P\ (f\ x\ xa\ xb\ xc\ xd\ xe\ xf\ xg\ xh))$
by (*rule equal-intr-rule*) (*fastforce*, *fast*)

lemmas *all-mem-range* = *all-mem-range1 all-mem-range2 all-mem-range3 all-mem-range4 all-mem-range5*
all-mem-range6 all-mem-range7 all-mem-range8

lemma *pred-fun-True-id*: *NO-MATCH id p* $\Longrightarrow$ *pred-fun* $(\lambda x.\ True)\ p\ f$ = *pred-fun* $(\lambda x.\ True)\ id\ (p \circ f)$
unfolding *fun.pred-map* **unfolding** *comp-def id-def* **..**

ML-file ‹*Tools/BNF/bnf-lfp-util.ML*›
ML-file ‹*Tools/BNF/bnf-lfp-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-lfp.ML*›
ML-file ‹*Tools/BNF/bnf-lfp-compat.ML*›
ML-file ‹*Tools/BNF/bnf-lfp-rec-sugar-more.ML*›
ML-file ‹*Tools/BNF/bnf-lfp-size.ML*›

ML-file ‹*Tools/datatype-simprocs.ML*›

simproc-setup *datatype-no-proper-subterm*
((*x* :: *'a* :: *size*) = *y*) = ‹*K Datatype-Simprocs.no-proper-subterm-proc*›

end

38 Equivalence Relations in Higher-Order Set Theory

theory *Equiv-Relations*
imports *BNF-Least-Fixpoint*
begin

38.1 Equivalence relations – set version

definition *equiv* :: *'a set* ⇒ (*'a* × *'a*) *set* ⇒ *bool*
where *equiv A r* ⟷ *r* ⊆ *A* × *A* ∧ *refl-on A r* ∧ *sym r* ∧ *trans r*

lemma *equivI*: *r* ⊆ *A* × *A* ⟹ *refl-on A r* ⟹ *sym r* ⟹ *trans r* ⟹ *equiv A r*
by (*simp add*: *equiv-def*)

lemma *equivE*:
assumes *equiv A r*
obtains *r* ⊆ *A* × *A* **and** *refl-on A r* **and** *sym r* **and** *trans r*
using *assms* **by** (*simp add*: *equiv-def*)

Suppes, Theorem 70: *r* is an equiv relation iff $r^{-1}\ O\ r = r$.

First half: *equiv A r* ⟹ $r^{-1}\ O\ r = r$.

lemma *sym-trans-comp-subset*:
assumes *r* ⊆ *A* × *A* **and** *sym-on A r* **and** *trans-on A r*
shows $r^{-1}\ O\ r \subseteq r$
proof (*rule subsetI*)
fix *p*
assume *p* ∈ $r^{-1}\ O\ r$
then obtain *x y z* **where** *p* = (*x*, *z*) **and** (*y*, *x*) ∈ *r* **and** (*y*, *z*) ∈ *r*
by *auto*
hence *x* ∈ *A* **and** *y* ∈ *A* **and** *z* ∈ *A*
using ‹*r* ⊆ *A* × *A*› **by** *auto*
have (*x*, *y*) ∈ *r*
using ‹(*y*, *x*) ∈ *r*› ‹*x* ∈ *A*› ‹*y* ∈ *A*› ‹*sym-on A r*› **by** (*simp add*: *sym-on-def*)
hence (*x*, *z*) ∈ *r*
using ‹*trans-on A r*›[*THEN trans-onD*, *OF* ‹*x* ∈ *A*› ‹*y* ∈ *A*› ‹*z* ∈ *A*›] ‹(*y*, *z*)
∈ *r*› **by** *blast*
thus *p* ∈ *r*
unfolding ‹*p* = (*x*, *z*)› **.**
qed

lemma *refl-on-comp-subset*: *r* ⊆ *A* × *A* ⟹ *refl-on A r* ⟹ *r* ⊆ $r^{-1}\ O\ r$
unfolding *refl-on-def* **by** *blast*

lemma *equiv-comp-eq*: *equiv A r* $\Longrightarrow$ r^{-1} *O r* = *r*
proof (*rule subset-antisym*)
show *equiv A r* $\Longrightarrow$ r^{-1} *O r* $\subseteq$ *r*
by (*rule sym-trans-comp-subset*[*of r A*]) (*auto elim*: *equivE intro*: *sym-on-subset trans-on-subset*)
next
show *equiv A r* $\Longrightarrow$ *r* $\subseteq$ r^{-1} *O r*
by (*rule refl-on-comp-subset*[*of r A*]) (*auto elim*: *equivE*)
qed

Second half.

lemma *comp-equivI*:
assumes r^{-1} *O r* = *r Domain r* = *A*
shows *equiv A r*
proof (*rule equivI*)
show *r* $\subseteq$ *A* $\times$ *A*
using *assms* **by** *auto*

have $*$: $\bigwedge x\ y.\ (x, y) \in r \Longrightarrow (y, x) \in r$
using *assms* **by** *blast*

thus *refl-on A r sym r trans r*
unfolding *refl-on-def sym-def trans-def*
using *assms* **by** *auto*
qed

38.2 Equivalence classes

lemma *equiv-class-subset*: *equiv A r* $\Longrightarrow$ (*a*, *b*) $\in$ *r* $\Longrightarrow$ *r*``{*a*} $\subseteq$ *r*``{*b*}
— lemma for the next result
unfolding *equiv-def trans-def sym-def* **by** *blast*

theorem *equiv-class-eq*:
assumes *equiv A r* **and** (*a*, *b*) $\in$ *r*
shows *r*``{*a*} = *r*``{*b*}
proof (*intro subset-antisym equiv-class-subset*[*OF* ‹*equiv A r*›])
show (*a*, *b*) $\in$ *r*
using ‹(*a*, *b*) $\in$ *r*› **.**
next
have *sym r*
using ‹*equiv A r*› **by** (*auto elim*: *equivE*)
thus (*b*, *a*) $\in$ *r*
using ‹(*a*, *b*) $\in$ *r*›
by (*auto dest*: *symD*)
qed

lemma *equiv-class-self*: *equiv A r* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *a* $\in$ *r*``{*a*}
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *subset-equiv-class*: *equiv A r* $\Longrightarrow$ $r``\{b\} \subseteq r``\{a\} \Longrightarrow b \in A \Longrightarrow (a, b) \in r$
— lemma for the next result
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *eq-equiv-class*: $r``\{a\} = r``\{b\} \Longrightarrow$ *equiv A r* $\Longrightarrow b \in A \Longrightarrow (a, b) \in r$
by (*iprover intro*: *equalityD2 subset-equiv-class*)

lemma *equiv-class-nondisjoint*: *equiv A r* $\Longrightarrow x \in (r``\{a\} \cap r``\{b\}) \Longrightarrow (a, b) \in r$
unfolding *equiv-def trans-def sym-def* **by** *blast*

lemma *equiv-type*: *equiv A r* $\Longrightarrow r \subseteq A \times A$
unfolding *equiv-def refl-on-def* **by** *blast*

lemma *equiv-class-eq-iff*: *equiv A r* $\Longrightarrow (x, y) \in r \longleftrightarrow r``\{x\} = r``\{y\} \wedge x \in A \wedge y \in A$
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

lemma *eq-equiv-class-iff*: *equiv A r* $\Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow r``\{x\} = r``\{y\} \longleftrightarrow (x, y) \in r$
by (*blast intro*!: *equiv-class-eq dest*: *eq-equiv-class equiv-type*)

lemma *disjnt-equiv-class*: *equiv A r* $\Longrightarrow$ *disjnt* $(r``\{a\})$ $(r``\{b\}) \longleftrightarrow (a, b) \notin r$
by (*auto dest*: *equiv-class-self simp*: *equiv-class-eq-iff disjnt-def*)

38.3 Quotients

definition *quotient* :: *'a set* $\Rightarrow$ *('a* $\times$ *'a) set* $\Rightarrow$ *'a set set* (**infixl** $\langle'/'/\rangle$ *90*)
where $A//r = (\bigcup x \in A.\ \{r``\{x\}\})$ — set of equiv classes

lemma *quotientI*: $x \in A \Longrightarrow r``\{x\} \in A//r$
unfolding *quotient-def* **by** *blast*

lemma *quotientE*: $X \in A//r \Longrightarrow (\bigwedge x.\ X = r``\{x\} \Longrightarrow x \in A \Longrightarrow P) \Longrightarrow P$
unfolding *quotient-def* **by** *blast*

lemma *Union-quotient*: *equiv A r* $\Longrightarrow \bigcup(A//r) = A$
unfolding *equiv-def refl-on-def quotient-def* **by** *blast*

lemma *quotient-disj-strong*:
assumes $r \subseteq A \times A$ **and** *sym-on A r* **and** *trans-on A r* **and** $X \in A//r$ **and** $Y \in A//r$
shows $X = Y \vee X \cap Y = \{\}$
proof −
obtain *x* **where** $x \in A$ **and** $X = \{x'.\ (x, x') \in r\}$
using $\langle X \in A//r\rangle$ **unfolding** *quotient-def UN-iff* **by** *blast*

moreover obtain *y* **where** $y \in A$ **and** $Y = \{y'.\ (y, y') \in r\}$
using $\langle Y \in A//r\rangle$ **unfolding** *quotient-def UN-iff* **by** *blast*

have *f8*: $\forall a\ aa.\ (aa,\ a) \in r \vee (a,\ aa) \notin r$
using ‹$r \subseteq A \times A$›[*unfolded subset-eq*] ‹*sym-on A r*›[*THEN sym-onD*] **by** *blast*
have *f9*: $\forall a\ aa\ ab.\ (aa,\ ab) \in r \vee (aa,\ a) \notin r \vee (a,\ ab) \notin r$
using ‹$r \subseteq A \times A$›[*unfolded subset-eq*] ‹*trans-on A r*›[*THEN trans-onD*] **by** *blast*
then have $\forall a\ aa.\ aa \in Y \vee (y,\ a) \notin r \vee (a,\ aa) \notin r$
using ‹$Y = \{y'.\ (y,\ y') \in r\}$› **by** *simp*
then show *?thesis*
using *f9 f8* ‹$X = \{x'.\ (x,\ x') \in r\}$› ‹$Y = \{y'.\ (y,\ y') \in r\}$›
Collect-cong disjoint-iff-not-equal mem-Collect-eq **by** *blast*
qed

lemma *quotient-disj*: *equiv A r* $\Longrightarrow$ $X \in A//r \Longrightarrow Y \in A//r \Longrightarrow X = Y \vee X \cap Y = \{\}$
by (*rule quotient-disj-strong*[*of r A X Y*])
(*auto elim*: *equivE intro*: *sym-on-subset trans-on-subset*)

lemma *quotient-eqI*:
assumes *equiv A r X* $\in A//r$ *Y* $\in A//r$ **and** *xy*: $x \in X\ y \in Y\ (x,\ y) \in r$
shows $X = Y$
proof −
obtain *a b* **where** $a \in A$ **and** *a*: $X = r\ ``\ \{a\}$ **and** $b \in A$ **and** *b*: $Y = r\ ``\ \{b\}$
using *assms* **by** (*auto elim*!: *quotientE*)
moreover have *sym r* **and** *trans r*
using ‹*equiv A r*›
by (*auto elim*: *equivE*)
ultimately have (*a*,*b*) ∈ *r*
using *xy* **unfolding** *sym-def trans-def* **by** *blast*
then show *?thesis*
unfolding *a b* **by** (*rule equiv-class-eq* [*OF* ‹*equiv A r*›])
qed

lemma *quotient-eq-iff*:
assumes *equiv A r X* $\in A//r$ *Y* $\in A//r$ **and** *xy*: $x \in X\ y \in Y$
shows $X = Y \longleftrightarrow (x,\ y) \in r$
proof
assume *L*: $X = Y$
with *assms* **show** $(x,\ y) \in r$
unfolding *equiv-def sym-def trans-def* **by** (*blast elim*!: *quotientE*)
next
assume §: $(x,\ y) \in r$ **show** $X = Y$
by (*rule quotient-eqI*) (*use* § *assms* **in** ‹*blast*+›)
qed

lemma *eq-equiv-class-iff2*: *equiv A r* $\Longrightarrow x \in A \Longrightarrow y \in A \Longrightarrow \{x\}//r = \{y\}//r \longleftrightarrow (x,\ y) \in r$
by (*simp add*: *quotient-def eq-equiv-class-iff*)

lemma *quotient-empty* [*simp*]: $\{\}//r = \{\}$
by (*simp add*: *quotient-def*)

lemma *quotient-is-empty* [*iff*]: $A//r = \{\} \longleftrightarrow A = \{\}$
by (*simp add*: *quotient-def*)

lemma *quotient-is-empty2* [*iff*]: $\{\} = A//r \longleftrightarrow A = \{\}$
by (*simp add*: *quotient-def*)

lemma *singleton-quotient*: $\{x\}//r = \{r \mathbin{``} \{x\}\}$
by (*simp add*: *quotient-def*)

lemma *quotient-diff1*: *inj-on* $(\lambda a.\ \{a\}//r)\ A \Longrightarrow a \in A \Longrightarrow (A - \{a\})//r = A//r - \{a\}//r$
unfolding *quotient-def inj-on-def* **by** *blast*

38.4 Refinement of one equivalence relation WRT another

lemma *refines-equiv-class-eq*: $R \subseteq S \Longrightarrow$ *equiv* $A\ R \Longrightarrow$ *equiv* $A\ S \Longrightarrow R``(S``\{a\}) = S``\{a\}$
by (*auto simp*: *equiv-class-eq-iff*)

lemma *refines-equiv-class-eq2*: $R \subseteq S \Longrightarrow$ *equiv* $A\ R \Longrightarrow$ *equiv* $A\ S \Longrightarrow S``(R``\{a\}) = S``\{a\}$
by (*auto simp*: *equiv-class-eq-iff*)

lemma *refines-equiv-image-eq*: $R \subseteq S \Longrightarrow$ *equiv* $A\ R \Longrightarrow$ *equiv* $A\ S \Longrightarrow (\lambda X.\ S``X) \mathbin{`} (A//R) = A//S$
by (*auto simp*: *quotient-def image-UN refines-equiv-class-eq2*)

lemma *finite-refines-finite*:
finite $(A//R) \Longrightarrow R \subseteq S \Longrightarrow$ *equiv* $A\ R \Longrightarrow$ *equiv* $A\ S \Longrightarrow$ *finite* $(A//S)$
by (*erule finite-surj* [**where** $f = \lambda X.\ S``X$]) (*simp add*: *refines-equiv-image-eq*)

lemma *finite-refines-card-le*:
finite $(A//R) \Longrightarrow R \subseteq S \Longrightarrow$ *equiv* $A\ R \Longrightarrow$ *equiv* $A\ S \Longrightarrow$ *card* $(A//S) \leq$ *card* $(A//R)$
by (*subst refines-equiv-image-eq* [*of R S A*, *symmetric*])
(*auto simp*: *card-image-le* [**where** $f = \lambda X.\ S``X$])

lemma (**in** *type-definition*) *card*: *card* $(UNIV :: 'b\ set) =$ *card* A
by (*simp add*: *card-image inj-on-def Abs-inject flip*: *Abs-image*)

38.5 Defining unary operations upon equivalence classes

A congruence-preserving function.

definition *congruent* :: $('a \times 'a)\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where *congruent* $r\ f \longleftrightarrow (\forall (y,\ z) \in r.\ f\ y = f\ z)$

lemma *congruentI*: $(\bigwedge y\ z.\ (y,\ z) \in r \Longrightarrow f\ y = f\ z) \Longrightarrow$ *congruent r f*
by (*auto simp add*: *congruent-def*)

lemma *congruentD*: *congruent r f* $\Longrightarrow (y,\ z) \in r \Longrightarrow f\ y = f\ z$
by (*auto simp add*: *congruent-def*)

abbreviation *RESPECTS* :: $('a \Rightarrow 'b) \Rightarrow ('a \times 'a)$ *set* $\Rightarrow$ *bool* (**infixr** ‹*respects*› *80*)
where *f respects r* $\equiv$ *congruent r f*

lemma *UN-constant-eq*: $a \in A \Longrightarrow \forall y \in A.\ f\ y = c \Longrightarrow (\bigcup y \in A.\ f\ y) = c$
— lemma required to prove *UN-equiv-class*
by *auto*

lemma *UN-equiv-class*:
assumes *equiv A r f respects r a* $\in A$
shows $(\bigcup x \in r``\{a\}.\ f\ x) = f\ a$
— Conversion rule
proof −
have §: $\forall x \in r\ ``\ \{a\}.\ f\ x = f\ a$
using *assms* **unfolding** *equiv-def congruent-def sym-def* **by** *blast*
show *?thesis*
by (*iprover intro*: *assms UN-constant-eq* [*OF equiv-class-self* §])
qed

lemma *UN-equiv-class-type*:
assumes *r*: *equiv A r f respects r* **and** *X*: $X \in A//r$ **and** *AB*: $\bigwedge x.\ x \in A \Longrightarrow f\ x \in B$
shows $(\bigcup x \in X.\ f\ x) \in B$
using *assms* **unfolding** *quotient-def*
by (*auto simp*: *UN-equiv-class* [*OF r*])

Sufficient conditions for injectiveness. Could weaken premises! major premise could be an inclusion; *bcong* could be $\bigwedge y.\ y \in A \Longrightarrow f\ y \in B$.

lemma *UN-equiv-class-inject*:
assumes *equiv A r f respects r*
and *eq*: $(\bigcup x \in X.\ f\ x) = (\bigcup y \in Y.\ f\ y)$
and *X*: $X \in A//r$ **and** *Y*: $Y \in A//r$
and *fr*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow f\ x = f\ y \Longrightarrow (x,\ y) \in r$
shows $X = Y$
proof −
obtain *a b* **where** $a \in A$ **and** *a*: $X = r\ ``\ \{a\}$ **and** $b \in A$ **and** *b*: $Y = r\ ``\ \{b\}$
using *assms* **by** (*auto elim*!: *quotientE*)
then have $\bigcup\ (f\ `\ r\ ``\ \{a\}) = f\ a \bigcup\ (f\ `\ r\ ``\ \{b\}) = f\ b$
by (*iprover intro*: *UN-equiv-class* [*OF* ‹*equiv A r*›] *assms*)+
then have $f\ a = f\ b$
using *eq* **unfolding** *a b* **by** (*iprover intro*: *trans sym*)
then have $(a,b) \in r$

using *fr* ‹$a \in A$› ‹$b \in A$› **by** *blast*
then show *?thesis*
unfolding *a b* **by** (*rule equiv-class-eq* [*OF* ‹*equiv A r*›])
qed

38.6 Defining binary operations upon equivalence classes

A congruence-preserving function of two arguments.

definition *congruent2* :: $('a \times 'a)$ *set* $\Rightarrow$ $('b \times 'b)$ *set* $\Rightarrow$ $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow$ *bool*
where *congruent2 r1 r2 f* $\longleftrightarrow$ $(\forall (y1, z1) \in r1.\ \forall (y2, z2) \in r2.\ f\ y1\ y2 = f\ z1\ z2)$

lemma *congruent2I'*:
assumes $\bigwedge y1\ z1\ y2\ z2.\ (y1, z1) \in r1 \Longrightarrow (y2, z2) \in r2 \Longrightarrow f\ y1\ y2 = f\ z1\ z2$
shows *congruent2 r1 r2 f*
using *assms* **by** (*auto simp add*: *congruent2-def*)

lemma *congruent2D*: *congruent2 r1 r2 f* $\Longrightarrow$ $(y1, z1) \in r1 \Longrightarrow (y2, z2) \in r2 \Longrightarrow$ *f y1 y2 = f z1 z2*
by (*auto simp add*: *congruent2-def*)

Abbreviation for the common case where the relations are identical.

abbreviation *RESPECTS2*:: $('a \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('a \times 'a)$ *set* $\Rightarrow$ *bool* (**infixr** ‹*respects2*› *80*)
where *f respects2 r* $\equiv$ *congruent2 r r f*

lemma *congruent2-implies-congruent*:
equiv A r1 $\Longrightarrow$ *congruent2 r1 r2 f* $\Longrightarrow$ *a* $\in$ *A* $\Longrightarrow$ *congruent r2* (*f a*)
unfolding *congruent-def congruent2-def equiv-def refl-on-def* **by** *blast*

lemma *congruent2-implies-congruent-UN*:
assumes *equiv A1 r1 equiv A2 r2 congruent2 r1 r2 f a* $\in$ *A2*
shows *congruent r1* ($\lambda x1.\ \bigcup x2 \in r2``\{a\}.\ f\ x1\ x2$)
unfolding *congruent-def*
proof *clarify*
fix *c d*
assume *cd*: $(c,d) \in r1$
then have *c* $\in$ *A1 d* $\in$ *A1*
using ‹*equiv A1 r1*› **by** (*auto elim*!: *equiv-type* [*THEN subsetD*, *THEN SigmaE2*])
moreover have *f c a = f d a*
using *assms cd* **unfolding** *congruent2-def equiv-def refl-on-def* **by** *blast*
ultimately show $\bigcup$ (*f c* ‘ *r2* ‘‘ $\{a\}$) = $\bigcup$ (*f d* ‘ *r2* ‘‘ $\{a\}$)
using *assms* **by** (*simp add*: *UN-equiv-class congruent2-implies-congruent*)
qed

lemma *UN-equiv-class2*:

equiv A1 r1 $\Longrightarrow$ *equiv A2 r2* $\Longrightarrow$ *congruent2 r1 r2 f* $\Longrightarrow$ *a1* $\in$ *A1* $\Longrightarrow$ *a2* $\in$ *A2*
$\Longrightarrow$
$(\bigcup x1 \in r1``\{a1\}.\ \bigcup x2 \in r2``\{a2\}.\ f\ x1\ x2) = f\ a1\ a2$
by (*simp add*: *UN-equiv-class congruent2-implies-congruent congruent2-implies-congruent-UN*)

lemma *UN-equiv-class-type2*:
equiv A1 r1 $\Longrightarrow$ *equiv A2 r2* $\Longrightarrow$ *congruent2 r1 r2 f*
$\Longrightarrow$ *X1* $\in$ *A1*//*r1* $\Longrightarrow$ *X2* $\in$ *A2*//*r2*
$\Longrightarrow$ $(\bigwedge x1\ x2.\ x1 \in A1 \Longrightarrow x2 \in A2 \Longrightarrow f\ x1\ x2 \in B)$
$\Longrightarrow$ $(\bigcup x1 \in X1.\ \bigcup x2 \in X2.\ f\ x1\ x2) \in B$
unfolding *quotient-def*
by (*blast intro*: *UN-equiv-class-type congruent2-implies-congruent-UN congruent2-implies-congruent quotientI*)

lemma *UN-UN-split-split-eq*:
$(\bigcup (x1,\ x2) \in X.\ \bigcup (y1,\ y2) \in Y.\ A\ x1\ x2\ y1\ y2) =$
$(\bigcup x \in X.\ \bigcup y \in Y.\ (\lambda(x1,\ x2).\ (\lambda(y1,\ y2).\ A\ x1\ x2\ y1\ y2)\ y)\ x)$
— Allows a natural expression of binary operators,
— without explicit calls to *split*
by *auto*

lemma *congruent2I*:
equiv A1 r1 $\Longrightarrow$ *equiv A2 r2*
$\Longrightarrow$ $(\bigwedge y\ z\ w.\ w \in A2 \Longrightarrow (y,z) \in r1 \Longrightarrow f\ y\ w = f\ z\ w)$
$\Longrightarrow$ $(\bigwedge y\ z\ w.\ w \in A1 \Longrightarrow (y,z) \in r2 \Longrightarrow f\ w\ y = f\ w\ z)$
$\Longrightarrow$ *congruent2 r1 r2 f*
— Suggested by John Harrison – the two subproofs may be
— *much* simpler than the direct proof.
unfolding *congruent2-def equiv-def refl-on-def*
by (*blast intro*: *trans*)

lemma *congruent2-commuteI*:
assumes *equivA*: *equiv A r*
and *commute*: $\bigwedge y\ z.\ y \in A \Longrightarrow z \in A \Longrightarrow f\ y\ z = f\ z\ y$
and *congt*: $\bigwedge y\ z\ w.\ w \in A \Longrightarrow (y,z) \in r \Longrightarrow f\ w\ y = f\ w\ z$
shows *f respects2 r*
proof (*rule congruent2I* [*OF equivA equivA*])
note *eqv* = *equivA* [*THEN equiv-type*, *THEN subsetD*, *THEN SigmaE2*]
show $\bigwedge y\ z\ w.\ [\![w \in A;\ (y,\ z) \in r]\!] \Longrightarrow f\ y\ w = f\ z\ w$
by (*iprover intro*: *commute* [*THEN trans*] *sym congt elim*: *eqv*)
show $\bigwedge y\ z\ w.\ [\![w \in A;\ (y,\ z) \in r]\!] \Longrightarrow f\ w\ y = f\ w\ z$
by (*iprover intro*: *congt elim*: *eqv*)
qed

38.7 Quotients and finiteness

Suggested by Florian Kammüller

lemma *finite-quotient*:

assumes *finite A r* $\subseteq A \times A$
shows *finite* $(A//r)$
— recall *equiv ?A ?r* $\Longrightarrow$ *?r* $\subseteq$ *?A* $\times$ *?A*
proof −
have $A//r \subseteq Pow\ A$
using *assms* **unfolding** *quotient-def* **by** *blast*
moreover have *finite* $(Pow\ A)$
using *assms* **by** *simp*
ultimately show *?thesis*
by (*iprover intro*: *finite-subset*)
qed

lemma *finite-equiv-class*: *finite A* $\Longrightarrow$ *r* $\subseteq A \times A \Longrightarrow X \in A//r \Longrightarrow$ *finite X*
unfolding *quotient-def*
by (*erule rev-finite-subset*) *blast*

lemma *equiv-imp-dvd-card*:
assumes *finite A equiv A r* $\bigwedge X.\ X \in A//r \Longrightarrow k$ *dvd card X*
shows *k dvd card A*
proof (*rule Union-quotient* [*THEN subst*])
show *k dvd card* $(\bigcup\ (A\ //\ r))$
apply (*rule dvd-partition*)
using *assms*
by (*auto simp*: *Union-quotient dest*: *quotient-disj*)
qed (*use assms* **in** *blast*)

38.8 Kernel of a Function

definition *kernel* :: $('a \Rightarrow 'b) \Rightarrow ('a * 'a)$ *set* **where**
kernel f = $\{(x,y).\ f\ x = f\ y\}$

lemma *equiv-kernel*: *equiv UNIV* (*kernel f*)
unfolding *kernel-def equiv-def refl-on-def sym-def trans-def* **by** *auto*

lemma *respects-kernel*: *f respects* (*kernel f*)
by (*simp add*: *congruent-def kernel-def*)

lemma *inj-on-vimage-image*: *inj-on* $(\lambda b.\ f\ -`\ \{b\})\ (f\ `\ A)$
using *inj-on-def* **by** *fastforce*

lemma *kernel-Image*: *kernel f* `` *A* = *f* $-`$ (*f* ` *A*)
unfolding *kernel-def* **by** (*auto simp add*: *rev-image-eqI*)

lemma *quotient-kernel-eq-image*: *A* // *kernel f* = $(\lambda b.\ f\ -`\ \{b\})$ ` *f* ` *A*
by(*auto simp*: *quotient-def kernel-Image*)

lemma *bij-betw-image-quotient-kernel*: *bij-betw* $(\lambda b.\ f\ -`\ \{b\})\ (f\ `\ A)$ (*A* // *kernel f*)
by (*simp add*: *bij-betw-def inj-on-vimage-image quotient-kernel-eq-image*)

38.9 Projection

definition *proj* :: $('b \times 'a)$ *set* $\Rightarrow$ $'b \Rightarrow 'a$ *set*
  where *proj r x* = *r* `` $\{x\}$

lemma *proj-preserves*: $x \in A \Longrightarrow$ *proj r x* $\in A//r$
  unfolding *proj-def* **by** (*rule quotientI*)

lemma *proj-in-iff*:
  assumes *equiv A r*
  shows *proj r x* $\in A//r \longleftrightarrow x \in A$
    (**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
  assume *?rhs*
  then show *?lhs* **by** (*simp add*: *proj-preserves*)
next
  assume *?lhs*
  then show *?rhs*
    unfolding *proj-def quotient-def*
  proof *safe*
    fix *y*
    assume *y*: $y \in A$ **and** *r* `` $\{x\} = r$ `` $\{y\}$
    moreover have $y \in r$ `` $\{y\}$
      using *assms y* **unfolding** *equiv-def refl-on-def* **by** *blast*
    ultimately have $(x, y) \in r$ **by** *blast*
    then show $x \in A$
      using *assms* **unfolding** *equiv-def refl-on-def* **by** *blast*
  qed
qed

lemma *proj-iff*: *equiv A r* $\Longrightarrow \{x, y\} \subseteq A \Longrightarrow$ *proj r x* = *proj r y* $\longleftrightarrow (x, y) \in r$
  by (*simp add*: *proj-def eq-equiv-class-iff*)

lemma *proj-image*: *proj r* ` *A* = *A*//*r*
  unfolding *proj-def*[*abs-def*] *quotient-def* **by** *blast*

lemma *in-quotient-imp-non-empty*: *equiv A r* $\Longrightarrow X \in A//r \Longrightarrow X \neq \{\}$
  unfolding *quotient-def* **using** *equiv-class-self* **by** *fast*

lemma *in-quotient-imp-in-rel*: *equiv A r* $\Longrightarrow X \in A//r \Longrightarrow \{x, y\} \subseteq X \Longrightarrow (x, y) \in r$
  using *quotient-eq-iff*[*THEN iffD1*] **by** *fastforce*

lemma *in-quotient-imp-closed*: *equiv A r* $\Longrightarrow X \in A//r \Longrightarrow x \in X \Longrightarrow (x, y) \in r \Longrightarrow y \in X$
  unfolding *quotient-def equiv-def trans-def* **by** *blast*

lemma *in-quotient-imp-subset*: *equiv A r* $\Longrightarrow X \in A//r \Longrightarrow X \subseteq A$

using *in-quotient-imp-in-rel equiv-type* **by** *fastforce*

38.10 Equivalence relations – predicate version

Partial equivalences.

definition *part-equivp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where *part-equivp* $R \longleftrightarrow (\exists x.\ R\ x\ x) \wedge (\forall x\ y.\ R\ x\ y \longleftrightarrow R\ x\ x \wedge R\ y\ y \wedge R\ x = R\ y)$
— John-Harrison-style characterization

lemma *part-equivpI*: $\exists x.\ R\ x\ x \Longrightarrow$ *symp* $R \Longrightarrow$ *transp* $R \Longrightarrow$ *part-equivp* R
by (*auto simp add*: *part-equivp-def*) (*auto elim*: *sympE transpE*)

lemma *part-equivpE*:
assumes *part-equivp* R
obtains x **where** $R\ x\ x$ **and** *symp* R **and** *transp* R
proof −
from *assms* **have** *1*: $\exists x.\ R\ x\ x$
and *2*: $\bigwedge x\ y.\ R\ x\ y \longleftrightarrow R\ x\ x \wedge R\ y\ y \wedge R\ x = R\ y$
unfolding *part-equivp-def* **by** *blast+*
from *1* **obtain** x **where** $R\ x\ x$ **..**
moreover have *symp* R
proof (*rule sympI*)
fix $x\ y$
assume $R\ x\ y$
with *2* [*of* $x\ y$] **show** $R\ y\ x$ **by** *auto*
qed
moreover have *transp* R
proof (*rule transpI*)
fix $x\ y\ z$
assume $R\ x\ y$ **and** $R\ y\ z$
with *2* [*of* $x\ y$] *2* [*of* $y\ z$] **show** $R\ x\ z$ **by** *auto*
qed
ultimately show *thesis* **by** (*rule that*)
qed

lemma *part-equivp-refl-symp-transp*: *part-equivp* $R \longleftrightarrow (\exists x.\ R\ x\ x) \wedge$ *symp* $R \wedge$ *transp* R
by (*auto intro*: *part-equivpI elim*: *part-equivpE*)

lemma *part-equivp-symp*: *part-equivp* $R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ x$
by (*erule part-equivpE*, *erule sympE*)

lemma *part-equivp-transp*: *part-equivp* $R \Longrightarrow R\ x\ y \Longrightarrow R\ y\ z \Longrightarrow R\ x\ z$
by (*erule part-equivpE*, *erule transpE*)

lemma *part-equivp-typedef*: *part-equivp* $R \Longrightarrow \exists d.\ d \in \{c.\ \exists x.\ R\ x\ x \wedge c = Collect\ (R\ x)\}$
by (*auto elim*: *part-equivpE*)

Total equivalences.

definition *equivp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
  where *equivp R* $\longleftrightarrow$ $(\forall x\ y.\ R\ x\ y = (R\ x = R\ y))$ — John-Harrison-style characterization

lemma *equivpI*: *reflp R* $\Longrightarrow$ *symp R* $\Longrightarrow$ *transp R* $\Longrightarrow$ *equivp R*
  by (*auto elim*: *reflpE sympE transpE simp add*: *equivp-def*)

lemma *equivpE*:
  assumes *equivp R*
  obtains *reflp R* **and** *symp R* **and** *transp R*
  using *assms* **by** (*auto intro*!: *that reflpI sympI transpI simp add*: *equivp-def*)

lemma *equivp-implies-part-equivp*: *equivp R* $\Longrightarrow$ *part-equivp R*
  by (*auto intro*: *part-equivpI elim*: *equivpE reflpE*)

lemma *equivp-equiv*: *equiv UNIV A* $\longleftrightarrow$ *equivp* $(\lambda x\ y.\ (x, y) \in A)$
  by (*auto intro*!: *equivI equivpI* [*to-set*] *elim*!: *equivE equivpE* [*to-set*])

lemma *equivp-reflp-symp-transp*: *equivp R* $\longleftrightarrow$ *reflp R* $\wedge$ *symp R* $\wedge$ *transp R*
  by (*auto intro*: *equivpI elim*: *equivpE*)

lemma *identity-equivp*: *equivp* $(=)$
  by (*auto intro*: *equivpI reflpI sympI transpI*)

lemma *equivp-reflp*: *equivp R* $\Longrightarrow$ *R x x*
  by (*erule equivpE*, *erule reflpE*)

lemma *equivp-symp*: *equivp R* $\Longrightarrow$ *R x y* $\Longrightarrow$ *R y x*
  by (*erule equivpE*, *erule sympE*)

lemma *equivp-transp*: *equivp R* $\Longrightarrow$ *R x y* $\Longrightarrow$ *R y z* $\Longrightarrow$ *R x z*
  by (*erule equivpE*, *erule transpE*)

lemma *equivp-rtranclp*: *symp r* $\Longrightarrow$ *equivp* r^{**}
  by(*intro equivpI reflpI sympI transpI*)(*auto dest*: *sympD*[*OF symp-rtranclp*])

lemmas *equivp-rtranclp-symclp* [*simp*] = *equivp-rtranclp*[*OF symp-on-symclp*]

lemma *equivp-vimage2p*: *equivp R* $\Longrightarrow$ *equivp* (*vimage2p f f R*)
  by(*auto simp add*: *equivp-def vimage2p-def dest*: *fun-cong*)

lemma *equivp-imp-transp*: *equivp R* $\Longrightarrow$ *transp R*
  by(*simp add*: *equivp-reflp-symp-transp*)

38.11 Equivalence closure

definition *equivclp* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
  equivclp r = $(symclp\ r)^{**}$

lemma *transp-equivclp* [*simp*]: *transp* (*equivclp r*)
by(*simp add*: *equivclp-def*)

lemma *reflp-equivclp* [*simp*]: *reflp* (*equivclp r*)
by(*simp add*: *equivclp-def*)

lemma *symp-equivclp* [*simp*]: *symp* (*equivclp r*)
by(*simp add*: *equivclp-def*)

lemma *equivp-evquivclp* [*simp*]: *equivp* (*equivclp r*)
by(*simp add*: *equivpI*)

lemma *tranclp-equivclp* [*simp*]: $(equivclp\ r)^{++} = equivclp\ r$
by(*simp add*: *equivclp-def*)

lemma *rtranclp-equivclp* [*simp*]: $(equivclp\ r)^{**} = equivclp\ r$
by(*simp add*: *equivclp-def*)

lemma *symclp-equivclp* [*simp*]: *symclp* (*equivclp r*) = *equivclp r*
by(*simp add*: *equivclp-def symp-symclp-eq*)

lemma *equivclp-symclp* [*simp*]: *equivclp* (*symclp r*) = *equivclp r*
by(*simp add*: *equivclp-def*)

lemma *equivclp-conversep* [*simp*]: *equivclp* (*conversep r*) = *equivclp r*
by(*simp add*: *equivclp-def*)

lemma *equivclp-sym* [*sym*]: *equivclp r x y* $\Longrightarrow$ *equivclp r y x*
by(*rule sympD*[*OF symp-equivclp*])

lemma *equivclp-OO-equivclp-le-equivclp*: *equivclp r OO equivclp r* $\leq$ *equivclp r*
by(*rule transp-relcompp-less-eq transp-equivclp*)+

lemma *rtranlcp-le-equivclp*: $r^{**} \leq equivclp\ r$
unfolding *equivclp-def* **by**(*rule rtranclp-mono*)(*simp add*: *symclp-pointfree*)

lemma *rtranclp-conversep-le-equivclp*: $r^{-1-1**} \leq equivclp\ r$
unfolding *equivclp-def* **by**(*rule rtranclp-mono*)(*simp add*: *symclp-pointfree*)

lemma *symclp-rtranclp-le-equivclp*: *symclp* $r^{**} \leq equivclp\ r$
unfolding *symclp-pointfree*
by(*rule le-supI*)(*simp-all add*: *rtranclp-conversep*[*symmetric*] *rtranlcp-le-equivclp rtranclp-conversep-le-equivclp*)

lemma *r-OO-conversep-into-equivclp*:
r^{**} *OO* $r^{-1-1**} \leq equivclp\ r$
by(*blast intro*: *order-trans*[*OF - equivclp-OO-equivclp-le-equivclp*] *relcompp-mono rtranlcp-le-equivclp rtranclp-conversep-le-equivclp del*: *predicate2I*)

lemma *equivclp-induct* [*consumes 1*, *case-names base step*, *induct pred*: *equivclp*]:
assumes *a*: *equivclp r a b*
and *cases*: *P a* $\bigwedge$*y z. equivclp r a y* $\Longrightarrow$ *r y z* $\vee$ *r z y* $\Longrightarrow$ *P y* $\Longrightarrow$ *P z*
shows *P b*
using *a* **unfolding** *equivclp-def*
by(*induction rule*: *rtranclp-induct*; *fold equivclp-def*; *blast intro*: *cases elim*: *symclpE*)

lemma *converse-equivclp-induct* [*consumes 1*, *case-names base step*]:
assumes *major*: *equivclp r a b*
and *cases*: *P b* $\bigwedge$*y z. r y z* $\vee$ *r z y* $\Longrightarrow$ *equivclp r z b* $\Longrightarrow$ *P z* $\Longrightarrow$ *P y*
shows *P a*
using *major* **unfolding** *equivclp-def*
by(*induction rule*: *converse-rtranclp-induct*; *fold equivclp-def*; *blast intro*: *cases elim*: *symclpE*)

lemma *equivclp-refl* [*simp*]: *equivclp r x x*
by(*rule reflpD*[*OF reflp-equivclp*])

lemma *r-into-equivclp* [*intro*]: *r x y* $\Longrightarrow$ *equivclp r x y*
unfolding *equivclp-def* **by**(*blast intro*: *symclpI*)

lemma *converse-r-into-equivclp* [*intro*]: *r y x* $\Longrightarrow$ *equivclp r x y*
unfolding *equivclp-def* **by**(*blast intro*: *symclpI*)

lemma *rtranclp-into-equivclp*: r^{**} *x y* $\Longrightarrow$ *equivclp r x y*
using *rtranlcp-le-equivclp*[*of r*] **by** *blast*

lemma *converse-rtranclp-into-equivclp*: r^{**} *y x* $\Longrightarrow$ *equivclp r x y*
by(*blast intro*: *equivclp-sym rtranclp-into-equivclp*)

lemma *equivclp-into-equivclp*: ⟦ *equivclp r a b*; *r b c* $\vee$ *r c b* ⟧ $\Longrightarrow$ *equivclp r a c*
unfolding *equivclp-def* **by**(*erule rtranclp.rtrancl-into-rtrancl*)(*auto intro*: *symclpI*)

lemma *equivclp-trans* [*trans*]: ⟦ *equivclp r a b*; *equivclp r b c* ⟧ $\Longrightarrow$ *equivclp r a c*
using *equivclp-OO-equivclp-le-equivclp*[*of r*] **by** *blast*

hide-const (**open**) *proj*

end

theory *Basic-BNF-LFPs*
imports *BNF-Least-Fixpoint*
begin

definition *xtor* :: *'a* $\Rightarrow$ *'a* **where**

xtor x = x

lemma *xtor-map*: *f (xtor x) = xtor (f x)*
unfolding *xtor-def* **by** (*rule refl*)

lemma *xtor-map-unique*: *u ∘ xtor = xtor ∘ f ⟹ u = f*
unfolding *o-def xtor-def* **.**

lemma *xtor-set*: *f (xtor x) = f x*
unfolding *xtor-def* **by** (*rule refl*)

lemma *xtor-rel*: *R (xtor x) (xtor y) = R x y*
unfolding *xtor-def* **by** (*rule refl*)

lemma *xtor-induct*: *(⋀x. P (xtor x)) ⟹ P z*
unfolding *xtor-def* **by** *assumption*

lemma *xtor-xtor*: *xtor (xtor x) = x*
unfolding *xtor-def* **by** (*rule refl*)

lemmas *xtor-inject = xtor-rel*[*of* (=)]

lemma *xtor-rel-induct*: *(⋀x y. vimage2p id-bnf id-bnf R x y ⟹ IR (xtor x) (xtor y)) ⟹ R ≤ IR*
unfolding *xtor-def vimage2p-def id-bnf-def* **..**

lemma *xtor-iff-xtor*: *u = xtor w ⟷ xtor u = w*
unfolding *xtor-def* **..**

lemma *Inl-def-alt*: *Inl ≡ (λa. xtor (id-bnf (Inl a)))*
unfolding *xtor-def id-bnf-def* **by** (*rule reflexive*)

lemma *Inr-def-alt*: *Inr ≡ (λa. xtor (id-bnf (Inr a)))*
unfolding *xtor-def id-bnf-def* **by** (*rule reflexive*)

lemma *Pair-def-alt*: *Pair ≡ (λa b. xtor (id-bnf (a, b)))*
unfolding *xtor-def id-bnf-def* **by** (*rule reflexive*)

definition *ctor-rec* :: *'a ⇒ 'a* **where**
ctor-rec x = x

lemma *ctor-rec*: *g = id ⟹ ctor-rec f (xtor x) = f ((id-bnf ∘ g ∘ id-bnf) x)*
unfolding *ctor-rec-def id-bnf-def xtor-def comp-def id-def* **by** *hypsubst* (*rule refl*)

lemma *ctor-rec-unique*: *g = id ⟹ f ∘ xtor = s ∘ (id-bnf ∘ g ∘ id-bnf) ⟹ f = ctor-rec s*
unfolding *ctor-rec-def id-bnf-def xtor-def comp-def id-def* **by** *hypsubst* (*rule refl*)

lemma *ctor-rec-def-alt*: *f = ctor-rec (f ∘ id-bnf)*

unfolding *ctor-rec-def id-bnf-def comp-def* **by** (*rule refl*)

lemma *ctor-rec-o-map*: *ctor-rec f ∘ g = ctor-rec* (*f ∘* (*id-bnf ∘ g ∘ id-bnf*))
unfolding *ctor-rec-def id-bnf-def comp-def* **by** (*rule refl*)

lemma *ctor-rec-transfer*: *rel-fun* (*rel-fun* (*vimage2p id-bnf id-bnf R*) *S*) (*rel-fun R S*) *ctor-rec ctor-rec*
unfolding *rel-fun-def vimage2p-def id-bnf-def ctor-rec-def* **by** *simp*

lemma *eq-fst-iff*: *a = fst p* ⟷ (∃ *b*. *p* = (*a*, *b*))
by (*cases p*) *auto*

lemma *eq-snd-iff*: *b = snd p* ⟷ (∃ *a*. *p* = (*a*, *b*))
by (*cases p*) *auto*

lemma *ex-neg-all-pos*: ((∃ *x*. *P x*) ⟹ *Q*) ≡ (⋀*x*. *P x* ⟹ *Q*)
by *standard blast+*

lemma *hypsubst-in-prems*: (⋀*x*. *y = x* ⟹ *z = f x* ⟹ *P*) ≡ (*z = f y* ⟹ *P*)
by *standard blast+*

lemma *isl-map-sum*:
isl (*map-sum f g s*) = *isl s*
by (*cases s*) *simp-all*

lemma *map-sum-sel*:
isl s ⟹ *projl* (*map-sum f g s*) = *f* (*projl s*)
¬ *isl s* ⟹ *projr* (*map-sum f g s*) = *g* (*projr s*)
by (*cases s*; *simp*)+

lemma *set-sum-sel*:
isl s ⟹ *projl s* ∈ *setl s*
¬ *isl s* ⟹ *projr s* ∈ *setr s*
by (*cases s*; *auto intro*: *setl.intros setr.intros*)+

lemma *rel-sum-sel*: *rel-sum R1 R2 a b* = (*isl a = isl b* ∧
(*isl a* ⟶ *isl b* ⟶ *R1* (*projl a*) (*projl b*)) ∧
(¬ *isl a* ⟶ ¬ *isl b* ⟶ *R2* (*projr a*) (*projr b*)))
by (*cases a b rule*: *sum.exhaust*[*case-product sum.exhaust*]) *simp-all*

lemma *isl-transfer*: *rel-fun* (*rel-sum A B*) (=) *isl isl*
unfolding *rel-fun-def rel-sum-sel* **by** *simp*

lemma *rel-prod-sel*: *rel-prod R1 R2 p q* = (*R1* (*fst p*) (*fst q*) ∧ *R2* (*snd p*) (*snd q*))
by (*force simp*: *rel-prod.simps elim*: *rel-prod.cases*)

ML-file ‹*Tools/BNF/bnf-lfp-basic-sugar.ML*›

declare *prod.size* [*no-atp*]

hide-const (**open**) *xtor ctor-rec*

hide-fact (**open**)
xtor-def xtor-map xtor-set xtor-rel xtor-induct xtor-xtor xtor-inject ctor-rec-def ctor-rec
ctor-rec-def-alt ctor-rec-o-map xtor-rel-induct Inl-def-alt Inr-def-alt Pair-def-alt

end

39 MESON Proof Method

theory *Meson*
imports *Nat*
begin

39.1 Negation Normal Form

de Morgan laws

lemma *not-conjD*: $\neg(P \wedge Q) \Longrightarrow \neg P \vee \neg Q$
and *not-disjD*: $\neg(P \vee Q) \Longrightarrow \neg P \wedge \neg Q$
and *not-notD*: $\neg\neg P \Longrightarrow P$
and *not-allD*: $\bigwedge P.\ \neg(\forall x.\ P(x)) \Longrightarrow \exists x.\ \neg P(x)$
and *not-exD*: $\bigwedge P.\ \neg(\exists x.\ P(x)) \Longrightarrow \forall x.\ \neg P(x)$
by *fast+*

Removal of $\longrightarrow$ and $\longleftrightarrow$ (positive and negative occurrences)

lemma *imp-to-disjD*: $P \longrightarrow Q \Longrightarrow \neg P \vee Q$
and *not-impD*: $\neg(P \longrightarrow Q) \Longrightarrow P \wedge \neg Q$
and *iff-to-disjD*: $P = Q \Longrightarrow (\neg P \vee Q) \wedge (\neg Q \vee P)$
and *not-iffD*: $\neg(P = Q) \Longrightarrow (P \vee Q) \wedge (\neg P \vee \neg Q)$
— Much more efficient than $P \wedge \neg\ Q \vee Q \wedge \neg\ P$ for computing CNF
and *not-refl-disj-D*: $x \neq x \vee P \Longrightarrow P$
by *fast+*

39.2 Pulling out the existential quantifiers

Conjunction

lemma *conj-exD1*: $\bigwedge P\ Q.\ (\exists x.\ P(x)) \wedge Q \Longrightarrow \exists x.\ P(x) \wedge Q$
and *conj-exD2*: $\bigwedge P\ Q.\ P \wedge (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P \wedge Q(x)$
by *fast+*

Disjunction

lemma *disj-exD*: $\bigwedge P\ Q.\ (\exists x.\ P(x)) \vee (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P(x) \vee Q(x)$
— DO NOT USE with forall-Skolemization: makes fewer schematic variables!!
— With ex-Skolemization, makes fewer Skolem constants

and *disj-exD1*: $\bigwedge P\ Q.\ (\exists x.\ P(x)) \vee Q \Longrightarrow \exists x.\ P(x) \vee Q$
and *disj-exD2*: $\bigwedge P\ Q.\ P \vee (\exists x.\ Q(x)) \Longrightarrow \exists x.\ P \vee Q(x)$
by *fast+*

lemma *disj-assoc*: $(P \vee Q) \vee R \Longrightarrow P \vee (Q \vee R)$
and *disj-comm*: $P \vee Q \Longrightarrow Q \vee P$
and *disj-FalseD1*: $False \vee P \Longrightarrow P$
and *disj-FalseD2*: $P \vee False \Longrightarrow P$
by *fast+*

Generation of contrapositives

Inserts negated disjunct after removing the negation; P is a literal. Model elimination requires assuming the negation of every attempted subgoal, hence the negated disjuncts.

lemma *make-neg-rule*: $\neg P \vee Q \Longrightarrow ((\neg P \Longrightarrow P) \Longrightarrow Q)$
by *blast*

Version for Plaisted's "Postive refinement" of the Meson procedure

lemma *make-refined-neg-rule*: $\neg P \vee Q \Longrightarrow (P \Longrightarrow Q)$
by *blast*

P should be a literal

lemma *make-pos-rule*: $P \vee Q \Longrightarrow ((P \Longrightarrow \neg P) \Longrightarrow Q)$
by *blast*

Versions of *make-neg-rule* and *make-pos-rule* that don't insert new assumptions, for ordinary resolution.

lemmas *make-neg-rule'* = *make-refined-neg-rule*

lemma *make-pos-rule'*: $[\![P \vee Q;\ \neg P]\!] \Longrightarrow Q$
by *blast*

Generation of a goal clause – put away the final literal

lemma *make-neg-goal*: $\neg P \Longrightarrow ((\neg P \Longrightarrow P) \Longrightarrow False)$
by *blast*

lemma *make-pos-goal*: $P \Longrightarrow ((P \Longrightarrow \neg P) \Longrightarrow False)$
by *blast*

39.3 Lemmas for Forward Proof

There is a similarity to congruence rules. They are also useful in ordinary proofs.

lemma *conj-forward*: $[\![P' \wedge Q';\ \ P' \Longrightarrow P;\ \ Q' \Longrightarrow Q\ ]\!] \Longrightarrow P \wedge Q$
by *blast*

lemma *disj-forward*: $[\![P'\vee Q';\ \ P' \Longrightarrow P;\ \ Q' \Longrightarrow Q\ ]\!] \Longrightarrow P\vee Q$
by *blast*

lemma *imp-forward*: $[\![P' \longrightarrow Q';\ \ P \Longrightarrow P';\ \ Q' \Longrightarrow Q\ ]\!] \Longrightarrow P \longrightarrow Q$
by *blast*

lemma *imp-forward2*: $[\![P' \longrightarrow Q';\ \ P \Longrightarrow P';\ \ P' \Longrightarrow Q' \Longrightarrow Q\ ]\!] \Longrightarrow P \longrightarrow Q$
by *blast*

lemma *disj-forward2*: $[\![\ P'\vee Q';\ \ P' \Longrightarrow P;\ \ [\![Q';\ P\Longrightarrow False]\!] \Longrightarrow Q]\!] \Longrightarrow P\vee Q$
apply *blast*
done

lemma *all-forward*: $[|\ \forall x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \forall x.\ P(x)$
by *blast*

lemma *ex-forward*: $[|\ \exists x.\ P'(x);\ \ !!x.\ P'(x) ==> P(x)\ |] ==> \exists x.\ P(x)$
by *blast*

39.4 Clausification helper

lemma *TruepropI*: $P \equiv Q \Longrightarrow Trueprop\ P \equiv Trueprop\ Q$
by *simp*

lemma *ext-cong-neq*: $F\ g \neq F\ h \Longrightarrow F\ g \neq F\ h \wedge (\exists x.\ g\ x \neq h\ x)$
apply (*erule contrapos-np*)
apply *clarsimp*
apply (*rule cong*[**where** $f = F$])
by *auto*

Combinator translation helpers

definition *COMBI* :: $'a \Rightarrow 'a$ **where**
$COMBI\ P = P$

definition *COMBK* :: $'a \Rightarrow 'b \Rightarrow 'a$ **where**
$COMBK\ P\ Q = P$

definition *COMBB* :: $('b => 'c) \Rightarrow ('a => 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBB\ P\ Q\ R = P\ (Q\ R)$

definition *COMBC* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow 'b \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBC\ P\ Q\ R = P\ R\ Q$

definition *COMBS* :: $('a \Rightarrow 'b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'c$ **where**
$COMBS\ P\ Q\ R = P\ R\ (Q\ R)$

lemma *abs-S*: $\lambda x.\ (f\ x)\ (g\ x) \equiv COMBS\ f\ g$
apply (*rule eq-reflection*)

apply (*rule ext*)
apply (*simp add*: *COMBS-def*)
done

lemma *abs-I*: $\lambda x.\ x \equiv COMBI$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBI-def*)
done

lemma *abs-K*: $\lambda x.\ y \equiv COMBK\ y$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBK-def*)
done

lemma *abs-B*: $\lambda x.\ a\ (g\ x) \equiv COMBB\ a\ g$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBB-def*)
done

lemma *abs-C*: $\lambda x.\ (f\ x)\ b \equiv COMBC\ f\ b$
apply (*rule eq-reflection*)
apply (*rule ext*)
apply (*simp add*: *COMBC-def*)
done

39.5 Skolemization helpers

definition *skolem* :: $'a \Rightarrow 'a$ **where**
$skolem = (\lambda x.\ x)$

lemma *skolem-COMBK-iff*: $P \longleftrightarrow skolem\ (COMBK\ P\ (i{::}nat))$
unfolding *skolem-def COMBK-def* **by** (*rule refl*)

lemmas *skolem-COMBK-I* = *iffD1* [*OF skolem-COMBK-iff*]

39.6 Meson package

ML-file ‹*Tools/Meson/meson.ML*›
ML-file ‹*Tools/Meson/meson-clausify.ML*›
ML-file ‹*Tools/Meson/meson-tactic.ML*›

hide-const (**open**) *COMBI COMBK COMBB COMBC COMBS skolem*
hide-fact (**open**) *not-conjD not-disjD not-notD not-allD not-exD imp-to-disjD not-impD iff-to-disjD not-iffD not-refl-disj-D conj-exD1 conj-exD2 disj-exD disj-exD1 disj-exD2 disj-assoc disj-comm disj-FalseD1 disj-FalseD2 TruepropI ext-cong-neq COMBI-def COMBK-def COMBB-def COMBC-def COMBS-def abs-I abs-K*

abs-B abs-C abs-S skolem-def skolem-COMBK-iff skolem-COMBK-I
end

40 Automatic Theorem Provers (ATPs)

theory *ATP*
imports *Meson Hilbert-Choice*
begin

40.1 ATP problems and proofs

ML-file ‹*Tools/ATP/atp-util.ML*›
ML-file ‹*Tools/ATP/atp-problem.ML*›
ML-file ‹*Tools/ATP/atp-proof.ML*›
ML-file ‹*Tools/ATP/atp-proof-redirect.ML*›

40.2 Higher-order reasoning helpers

definition *fFalse* :: *bool* **where**
fFalse $\longleftrightarrow$ *False*

definition *fTrue* :: *bool* **where**
fTrue $\longleftrightarrow$ *True*

definition *fNot* :: *bool* $\Rightarrow$ *bool* **where**
fNot P $\longleftrightarrow$ $\neg$ *P*

definition *fComp* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
fComp P = ($\lambda x.$ $\neg$ *P x*)

definition *fconj* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where**
fconj P Q $\longleftrightarrow$ *P* $\wedge$ *Q*

definition *fdisj* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where**
fdisj P Q $\longleftrightarrow$ *P* $\vee$ *Q*

definition *fimplies* :: *bool* $\Rightarrow$ *bool* $\Rightarrow$ *bool* **where**
fimplies P Q $\longleftrightarrow$ (*P* $\longrightarrow$ *Q*)

definition *fAll* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
fAll P $\longleftrightarrow$ *All P*

definition *fEx* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ *bool* **where**
fEx P $\longleftrightarrow$ *Ex P*

definition *fequal* :: '*a* $\Rightarrow$ '*a* $\Rightarrow$ *bool* **where**
fequal x y $\longleftrightarrow$ (*x* = *y*)

definition *fChoice* :: ('*a* $\Rightarrow$ *bool*) $\Rightarrow$ '*a* **where**

fChoice ≡ *Hilbert-Choice.Eps*

lemma *fTrue-ne-fFalse*: *fFalse* ≠ *fTrue*
unfolding *fFalse-def fTrue-def* **by** *simp*

lemma *fNot-table*:
fNot fFalse = *fTrue*
fNot fTrue = *fFalse*
unfolding *fFalse-def fTrue-def fNot-def* **by** *auto*

lemma *fconj-table*:
fconj fFalse P = *fFalse*
fconj P fFalse = *fFalse*
fconj fTrue fTrue = *fTrue*
unfolding *fFalse-def fTrue-def fconj-def* **by** *auto*

lemma *fdisj-table*:
fdisj fTrue P = *fTrue*
fdisj P fTrue = *fTrue*
fdisj fFalse fFalse = *fFalse*
unfolding *fFalse-def fTrue-def fdisj-def* **by** *auto*

lemma *fimplies-table*:
fimplies P fTrue = *fTrue*
fimplies fFalse P = *fTrue*
fimplies fTrue fFalse = *fFalse*
unfolding *fFalse-def fTrue-def fimplies-def* **by** *auto*

lemma *fAll-table*:
Ex (*fComp P*) ∨ *fAll P* = *fTrue*
All P ∨ *fAll P* = *fFalse*
unfolding *fFalse-def fTrue-def fComp-def fAll-def* **by** *auto*

lemma *fEx-table*:
All (*fComp P*) ∨ *fEx P* = *fTrue*
Ex P ∨ *fEx P* = *fFalse*
unfolding *fFalse-def fTrue-def fComp-def fEx-def* **by** *auto*

lemma *fequal-table*:
fequal x x = *fTrue*
x = *y* ∨ *fequal x y* = *fFalse*
unfolding *fFalse-def fTrue-def fequal-def* **by** *auto*

lemma *fNot-law*:
fNot P ≠ *P*
unfolding *fNot-def* **by** *auto*

lemma *fComp-law*:
fComp P x ⟷ ¬ *P x*

unfolding *fComp-def* **..**

lemma *fconj-laws*:
fconj P P $\longleftrightarrow$ *P*
fconj P Q $\longleftrightarrow$ *fconj Q P*
fNot (*fconj P Q*) $\longleftrightarrow$ *fdisj* (*fNot P*) (*fNot Q*)
unfolding *fNot-def fconj-def fdisj-def* **by** *auto*

lemma *fdisj-laws*:
fdisj P P $\longleftrightarrow$ *P*
fdisj P Q $\longleftrightarrow$ *fdisj Q P*
fNot (*fdisj P Q*) $\longleftrightarrow$ *fconj* (*fNot P*) (*fNot Q*)
unfolding *fNot-def fconj-def fdisj-def* **by** *auto*

lemma *fimplies-laws*:
fimplies P Q $\longleftrightarrow$ *fdisj* ($\neg$ *P*) *Q*
fNot (*fimplies P Q*) $\longleftrightarrow$ *fconj P* (*fNot Q*)
unfolding *fNot-def fconj-def fdisj-def fimplies-def* **by** *auto*

lemma *fAll-law*:
fNot (*fAll R*) $\longleftrightarrow$ *fEx* (*fComp R*)
unfolding *fNot-def fComp-def fAll-def fEx-def* **by** *auto*

lemma *fEx-law*:
fNot (*fEx R*) $\longleftrightarrow$ *fAll* (*fComp R*)
unfolding *fNot-def fComp-def fAll-def fEx-def* **by** *auto*

lemma *fequal-laws*:
fequal x y = *fequal y x*
fequal x y = *fFalse* $\vee$ *fequal y z* = *fFalse* $\vee$ *fequal x z* = *fTrue*
fequal x y = *fFalse* $\vee$ *fequal* (*f x*) (*f y*) = *fTrue*
unfolding *fFalse-def fTrue-def fequal-def* **by** *auto*

lemma *fChoice-iff-Ex*: *P* (*fChoice P*) $\longleftrightarrow$ *HOL.Ex P*
unfolding *fChoice-def*
by (*fact some-eq-ex*)

We use the *Ex* constant on the right-hand side of *fChoice-iff-Ex* because we want to use the TPTP-native version if fChoice is introduced in a logic that supports FOOL. In logics that don't support it, it gets replaced by fEx during processing. Notice that we cannot use $\exists x.\ P\ x$, as existentials are not skolimized by the metis proof method but *Ex P* is eta-expanded if FOOL is supported.

40.3 Basic connection between ATPs and HOL

ML-file ‹*Tools/lambda-lifting.ML*›
ML-file ‹*Tools/monomorph.ML*›
ML-file ‹*Tools/ATP/atp-problem-generate.ML*›

ML-file ‹*Tools/ATP/atp-proof-reconstruct.ML*›

end

41 Metis Proof Method

theory *Metis*
imports *ATP*
begin

context notes [[*ML-check-strings* = *false*, *ML-catch-all*]]
begin
ML-file ‹~~/*src/Tools/Metis/metis.ML*›
end

41.1 Literal selection and lambda-lifting helpers

definition *select* :: $'a \Rightarrow 'a$ **where**
select = $(\lambda x.\ x)$

lemma *not-atomize*: $(\neg\ A \Longrightarrow \mathit{False}) \equiv \mathit{Trueprop}\ A$
by (*cut-tac atomize-not* [*of* $\neg\ A$]) *simp*

lemma *atomize-not-select*: $(A \Longrightarrow \mathit{select}\ \mathit{False}) \equiv \mathit{Trueprop}\ (\neg\ A)$
unfolding *select-def* **by** (*rule atomize-not*)

lemma *not-atomize-select*: $(\neg\ A \Longrightarrow \mathit{select}\ \mathit{False}) \equiv \mathit{Trueprop}\ A$
unfolding *select-def* **by** (*rule not-atomize*)

lemma *select-FalseI*: *False* $\Longrightarrow$ *select False*
by *simp*

definition *lambda* :: $'a \Rightarrow 'a$ **where**
lambda = $(\lambda x.\ x)$

lemma *eq-lambdaI*: $x \equiv y \Longrightarrow x \equiv \mathit{lambda}\ y$
unfolding *lambda-def* **by** *assumption*

41.2 Metis package

ML-file ‹*Tools/Metis/metis-generate.ML*›
ML-file ‹*Tools/Metis/metis-reconstruct.ML*›
ML-file ‹*Tools/Metis/metis-instantiations.ML*›
ML-file ‹*Tools/Metis/metis-tactic.ML*›

hide-const (**open**) *select fFalse fTrue fNot fComp fconj fdisj fimplies fAll fEx fequal lambda*
hide-fact (**open**) *select-def not-atomize atomize-not-select not-atomize-select select-FalseI*

fFalse-def fTrue-def fNot-def fconj-def fdisj-def fimplies-def fAll-def fEx-def fequal-def
fTrue-ne-fFalse fNot-table fconj-table fdisj-table fimplies-table fAll-table fEx-table
fequal-table fAll-table fEx-table fNot-law fComp-law fconj-laws fdisj-laws fimplies-laws
fequal-laws fAll-law fEx-law lambda-def eq-lambdaI

end

42 Generic theorem transfer using relations

theory *Transfer*
imports *Basic-BNF-LFPs Hilbert-Choice Metis*
begin

42.1 Relator for function space

bundle *lifting-syntax*
begin
notation *rel-fun* (**infixr** ‹===>› *55*)
notation *map-fun* (**infixr** ‹--->› *55*)
end

context includes *lifting-syntax*
begin

lemma *rel-funD2*:
assumes *rel-fun A B f g* **and** *A x x*
shows *B* (*f x*) (*g x*)
using *assms* **by** (*rule rel-funD*)

lemma *rel-funE*:
assumes *rel-fun A B f g* **and** *A x y*
obtains *B* (*f x*) (*g y*)
using *assms* **by** (*simp add*: *rel-fun-def*)

lemmas *rel-fun-eq* = *fun.rel-eq*

lemma *rel-fun-eq-rel*:
shows *rel-fun* (=) *R* = (λ*f g*. ∀ *x*. *R* (*f x*) (*g x*))
by (*simp add*: *rel-fun-def*)

42.2 Transfer method

Explicit tag for relation membership allows for backward proof methods.

definition *Rel* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a* ⇒ '*b* ⇒ *bool*
where *Rel r* ≡ *r*

Handling of equality relations

definition *is-equality* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$
where $is\text{-}equality\ R \longleftrightarrow R = (=)$

lemma *is-equality-eq*: *is-equality* $(=)$
unfolding *is-equality-def* **by** *simp*

Reverse implication for monotonicity rules

definition *rev-implies* **where**
$rev\text{-}implies\ x\ y \longleftrightarrow (y \longrightarrow x)$

Handling of meta-logic connectives

definition *transfer-forall* **where**
$transfer\text{-}forall \equiv All$

definition *transfer-implies* **where**
$transfer\text{-}implies \equiv (\longrightarrow)$

definition *transfer-bforall* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where $transfer\text{-}bforall \equiv (\lambda P\ Q.\ \forall x.\ P\ x \longrightarrow Q\ x)$

lemma *transfer-forall-eq*: $(\bigwedge x.\ P\ x) \equiv Trueprop\ (transfer\text{-}forall\ (\lambda x.\ P\ x))$
unfolding *atomize-all transfer-forall-def* **..**

lemma *transfer-implies-eq*: $(A \Longrightarrow B) \equiv Trueprop\ (transfer\text{-}implies\ A\ B)$
unfolding *atomize-imp transfer-implies-def* **..**

lemma *transfer-bforall-unfold*:
$Trueprop\ (transfer\text{-}bforall\ P\ (\lambda x.\ Q\ x)) \equiv (\bigwedge x.\ P\ x \Longrightarrow Q\ x)$
unfolding *transfer-bforall-def atomize-imp atomize-all* **..**

lemma *transfer-start*: $[\![P;\ Rel\ (=)\ P\ Q]\!] \Longrightarrow Q$
unfolding *Rel-def* **by** *simp*

lemma *transfer-start'*: $[\![P;\ Rel\ (\longrightarrow)\ P\ Q]\!] \Longrightarrow Q$
unfolding *Rel-def* **by** *simp*

lemma *transfer-prover-start*: $[\![x = x';\ Rel\ R\ x'\ y]\!] \Longrightarrow Rel\ R\ x\ y$
by *simp*

lemma *untransfer-start*: $[\![Q;\ Rel\ (=)\ P\ Q]\!] \Longrightarrow P$
unfolding *Rel-def* **by** *simp*

lemma *Rel-eq-refl*: $Rel\ (=)\ x\ x$
unfolding *Rel-def* **..**

lemma *Rel-app*:
assumes $Rel\ (A ===> B)\ f\ g$ **and** $Rel\ A\ x\ y$
shows $Rel\ B\ (f\ x)\ (g\ y)$
using *assms* **unfolding** *Rel-def rel-fun-def* **by** *fast*

lemma *Rel-abs*:
assumes $\bigwedge x\ y.\ Rel\ A\ x\ y \Longrightarrow Rel\ B\ (f\ x)\ (g\ y)$
shows $Rel\ (A ===> B)\ (\lambda x.\ f\ x)\ (\lambda y.\ g\ y)$
using *assms* **unfolding** *Rel-def rel-fun-def* **by** *fast*

42.3 Predicates on relations, i.e. "class constraints"

See also *left-unique* and *right-unique*.

definition *left-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where $left\text{-}total\ R \longleftrightarrow (\forall x.\ \exists y.\ R\ x\ y)$

definition *right-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where $right\text{-}total\ R \longleftrightarrow (\forall y.\ \exists x.\ R\ x\ y)$

definition *bi-total* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where $bi\text{-}total\ R \longleftrightarrow (\forall x.\ \exists y.\ R\ x\ y) \wedge (\forall y.\ \exists x.\ R\ x\ y)$

definition *bi-unique* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$
where $bi\text{-}unique\ R \longleftrightarrow$
$(\forall x\ y\ z.\ R\ x\ y \longrightarrow R\ x\ z \longrightarrow y = z) \wedge$
$(\forall x\ y\ z.\ R\ x\ z \longrightarrow R\ y\ z \longrightarrow x = y)$

lemma *left-totalI*:
$(\bigwedge x.\ \exists y.\ R\ x\ y) \Longrightarrow left\text{-}total\ R$
unfolding *left-total-def* **by** *blast*

lemma *left-totalE*:
assumes *left-total R*
obtains $(\bigwedge x.\ \exists y.\ R\ x\ y)$
using *assms* **unfolding** *left-total-def* **by** *blast*

lemma *bi-uniqueDr*: $[\![\ bi\text{-}unique\ A;\ A\ x\ y;\ A\ x\ z\ ]\!] \Longrightarrow y = z$
by(*simp add*: *bi-unique-def*)

lemma *bi-uniqueDl*: $[\![\ bi\text{-}unique\ A;\ A\ x\ y;\ A\ z\ y\ ]\!] \Longrightarrow x = z$
by(*simp add*: *bi-unique-def*)

lemma *bi-unique-iff*: $bi\text{-}unique\ R \longleftrightarrow (\forall z.\ \exists_{\leq 1} x.\ R\ x\ z) \wedge (\forall z.\ \exists_{\leq 1} x.\ R\ z\ x)$
unfolding *Uniq-def bi-unique-def* **by** *force*

lemma *right-totalI*: $(\bigwedge y.\ \exists x.\ A\ x\ y) \Longrightarrow right\text{-}total\ A$
by(*simp add*: *right-total-def*)

lemma *right-totalE*:
assumes *right-total A*
obtains *x* **where** *A x y*
using *assms* **by**(*auto simp add*: *right-total-def*)

lemma *right-total-alt-def2*:
right-total R ⟷ ((*R* ===> (⟶)) ===> (⟶)) *All All* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **then show** *?rhs*
unfolding *right-total-def rel-fun-def* **by** *blast*
next
assume §: *?rhs*
show *?lhs*
using § [*unfolded rel-fun-def*, *rule-format*, *of* λ*x*. *True* λ*y*. ∃ *x*. *R x y*]
unfolding *right-total-def* **by** *blast*
qed

lemma *right-unique-alt-def2*:
right-unique R ⟷ (*R* ===> *R* ===> (⟶)) (=) (=)
unfolding *right-unique-def rel-fun-def* **by** *auto*

lemma *bi-total-alt-def2*:
bi-total R ⟷ ((*R* ===> (=)) ===> (=)) *All All* (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **then show** *?rhs*
unfolding *bi-total-def rel-fun-def* **by** *blast*
next
assume §: *?rhs*
show *?lhs*
using § [*unfolded rel-fun-def*, *rule-format*, *of* λ*x*. ∃ *y*. *R x y* λ*y*. *True*]
using § [*unfolded rel-fun-def*, *rule-format*, *of* λ*x*. *True* λ*y*. ∃ *x*. *R x y*]
by (*auto simp*: *bi-total-def*)
qed

lemma *bi-unique-alt-def2*:
bi-unique R ⟷ (*R* ===> *R* ===> (=)) (=) (=)
unfolding *bi-unique-def rel-fun-def* **by** *auto*

lemma [*simp*]:
shows *left-total-conversep*: *left-total* A^{-1-1} ⟷ *right-total A*
and *right-total-conversep*: *right-total* A^{-1-1} ⟷ *left-total A*
by(*simp-all add*: *left-total-def right-total-def*)

lemma *bi-unique-conversep* [*simp*]: *bi-unique* R^{-1-1} = *bi-unique R*
by(*auto simp add*: *bi-unique-def*)

lemma *bi-total-conversep* [*simp*]: *bi-total* R^{-1-1} = *bi-total R*
by(*auto simp add*: *bi-total-def*)

lemma *right-unique-alt-def*: *right-unique R* = (*conversep R OO R* ≤ (=)) **unfolding** *right-unique-def* **by** *blast*
lemma *left-unique-alt-def*: *left-unique R* = (*R OO* (*conversep R*) ≤ (=)) **unfolding** *left-unique-def* **by** *blast*

lemma *right-total-alt-def*: *right-total R* = (*conversep R OO R* ≥ (=)) **unfolding** *right-total-def* **by** *blast*
lemma *left-total-alt-def*: *left-total R* = (*R OO conversep R* ≥ (=)) **unfolding** *left-total-def* **by** *blast*

lemma *bi-total-alt-def*: *bi-total A* = (*left-total A* ∧ *right-total A*)
unfolding *left-total-def right-total-def bi-total-def* **by** *blast*

lemma *bi-unique-alt-def*: *bi-unique A* = (*left-unique A* ∧ *right-unique A*)
unfolding *left-unique-def right-unique-def bi-unique-def* **by** *blast*

lemma *bi-totalI*: *left-total R* ⟹ *right-total R* ⟹ *bi-total R*
unfolding *bi-total-alt-def* **..**

lemma *bi-uniqueI*: *left-unique R* ⟹ *right-unique R* ⟹ *bi-unique R*
unfolding *bi-unique-alt-def* **..**

end

lemma *is-equality-lemma*: (⋀*R*. *is-equality R* ⟹ *PROP* (*P R*)) ≡ *PROP* (*P* (=))
unfolding *is-equality-def*
proof (*rule equal-intr-rule*)
show (⋀*R*. *R* = (=) ⟹ *PROP P R*) ⟹ *PROP P* (=)
apply (*drule meta-spec*)
apply (*erule meta-mp* [*OF - refl*])
done
qed *simp*

lemma *Domainp-lemma*: (⋀*R*. *Domainp T* = *R* ⟹ *PROP* (*P R*)) ≡ *PROP* (*P* (*Domainp T*))
proof (*rule equal-intr-rule*)
show (⋀*R*. *Domainp T* = *R* ⟹ *PROP P R*) ⟹ *PROP P* (*Domainp T*)
apply (*drule meta-spec*)
apply (*erule meta-mp* [*OF - refl*])
done
qed *simp*

ML-file ‹*Tools/Transfer/transfer.ML*›
declare *refl* [*transfer-rule*]

hide-const (**open**) *Rel*

context includes *lifting-syntax*
begin

Handling of domains

lemma *Domainp-iff*: *Domainp T x* ⟷ (∃ *y*. *T x y*)
by *auto*

lemma *Domainp-refl*[*transfer-domain-rule*]:
  Domainp T = Domainp T **..**

lemma *Domain-eq-top*[*transfer-domain-rule*]: *Domainp* (=) = *top* **by** *auto*

lemma *Domainp-pred-fun-eq*[*relator-domain*]:
  assumes *left-unique T*
  shows *Domainp* (*T* ===> *S*) = *pred-fun* (*Domainp T*) (*Domainp S*) (**is** *?lhs* = *?rhs*)
proof (*intro ext iffI*)
  fix *x*
  assume *?lhs x*
  then show *?rhs x*
    using *assms* **unfolding** *rel-fun-def pred-fun-def* **by** *blast*
next
  fix *x*
  assume *?rhs x*
  then have ∃ *g*. ∀ *y xa*. *T xa y* ⟶ *S* (*x xa*) (*g y*)
    using *assms* **unfolding** *Domainp-iff left-unique-def pred-fun-def*
    by (*intro choice*) *blast*
  then show *?lhs x*
    by *blast*
qed

Properties are preserved by relation composition.

lemma *OO-def*: *R OO S* = (λ*x z*. ∃ *y*. *R x y* ∧ *S y z*)
  by *auto*

lemma *bi-total-OO*: ⟦*bi-total A*; *bi-total B*⟧ ⟹ *bi-total* (*A OO B*)
  unfolding *bi-total-def OO-def* **by** *fast*

lemma *bi-unique-OO*: ⟦*bi-unique A*; *bi-unique B*⟧ ⟹ *bi-unique* (*A OO B*)
  unfolding *bi-unique-def OO-def* **by** *blast*

lemma *right-total-OO*:
  ⟦*right-total A*; *right-total B*⟧ ⟹ *right-total* (*A OO B*)
  unfolding *right-total-def OO-def* **by** *fast*

lemma *right-unique-OO*:
  ⟦*right-unique A*; *right-unique B*⟧ ⟹ *right-unique* (*A OO B*)
  unfolding *right-unique-def OO-def* **by** *fast*

lemma *left-total-OO*: *left-total R* ⟹ *left-total S* ⟹ *left-total* (*R OO S*)
  unfolding *left-total-def OO-def* **by** *fast*

lemma *left-unique-OO*: *left-unique R* ⟹ *left-unique S* ⟹ *left-unique* (*R OO S*)
  unfolding *left-unique-def OO-def* **by** *blast*

42.4 Properties of relators

lemma *left-total-eq*[*transfer-rule*]: *left-total* (=)
unfolding *left-total-def* **by** *blast*

lemma *left-unique-eq*[*transfer-rule*]: *left-unique* (=)
unfolding *left-unique-def* **by** *blast*

lemma *right-total-eq* [*transfer-rule*]: *right-total* (=)
unfolding *right-total-def* **by** *simp*

lemma *right-unique-eq* [*transfer-rule*]: *right-unique* (=)
unfolding *right-unique-def* **by** *simp*

lemma *bi-total-eq*[*transfer-rule*]: *bi-total* (=)
unfolding *bi-total-def* **by** *simp*

lemma *bi-unique-eq*[*transfer-rule*]: *bi-unique* (=)
unfolding *bi-unique-def* **by** *simp*

lemma *left-total-fun*[*transfer-rule*]:
assumes *left-unique A left-total B*
shows *left-total* (*A* ===> *B*)
unfolding *left-total-def*
proof
fix *f*
show *Ex* ((*A* ===> *B*) *f*)
unfolding *rel-fun-def*
proof (*intro exI strip*)
fix *x y*
assume *A*: *A x y*
have (*THE x*. *A x y*) = *x*
using *A assms* **by** (*simp add*: *left-unique-def the-equality*)
then show *B* (*f x*) (*SOME z*. *B* (*f* (*THE x*. *A x y*)) *z*)
using *assms* **by** (*force simp*: *left-total-def intro*: *someI-ex*)
qed
qed

lemma *left-unique-fun*[*transfer-rule*]:
⟦*left-total A*; *left-unique B*⟧ ⟹ *left-unique* (*A* ===> *B*)
unfolding *left-total-def left-unique-def rel-fun-def*
by (*clarify*, *rule ext*, *fast*)

lemma *right-total-fun* [*transfer-rule*]:
assumes *right-unique A right-total B*
shows *right-total* (*A* ===> *B*)
unfolding *right-total-def*
proof
fix *g*
show ∃ *x*. (*A* ===> *B*) *x g*

unfolding *rel-fun-def*
proof (*intro exI strip*)
fix *x y*
assume *A*: *A x y*
have (*THE y. A x y*) = *y*
using *A assms* **by** (*simp add*: *right-unique-def the-equality*)
then show *B* (*SOME z. B z* (*g* (*THE y. A x y*))) (*g y*)
using *assms* **by** (*force simp*: *right-total-def intro*: *someI-ex*)
qed
qed

lemma *right-unique-fun* [*transfer-rule*]:
⟦*right-total A*; *right-unique B*⟧ ⟹ *right-unique* (*A* ===> *B*)
unfolding *right-total-def right-unique-def rel-fun-def*
by (*clarify*, *rule ext*, *fast*)

lemma *bi-total-fun*[*transfer-rule*]:
⟦*bi-unique A*; *bi-total B*⟧ ⟹ *bi-total* (*A* ===> *B*)
unfolding *bi-unique-alt-def bi-total-alt-def*
by (*blast intro*: *right-total-fun left-total-fun*)

lemma *bi-unique-fun*[*transfer-rule*]:
⟦*bi-total A*; *bi-unique B*⟧ ⟹ *bi-unique* (*A* ===> *B*)
unfolding *bi-unique-alt-def bi-total-alt-def*
by (*blast intro*: *right-unique-fun left-unique-fun*)

end

lemma *if-conn*:
(*if P* ∧ *Q then t else e*) = (*if P then if Q then t else e else e*)
(*if P* ∨ *Q then t else e*) = (*if P then t else if Q then t else e*)
(*if P* ⟶ *Q then t else e*) = (*if P then if Q then t else e else t*)
(*if* ¬ *P then t else e*) = (*if P then e else t*)
by *auto*

ML-file ‹*Tools/Transfer/transfer-bnf.ML*›
ML-file ‹*Tools/BNF/bnf-fp-rec-sugar-transfer.ML*›

declare *pred-fun-def* [*simp*]
declare *rel-fun-eq* [*relator-eq*]

declare *fun.Domainp-rel*[*relator-domain del*]

42.5 Transfer rules

context includes *lifting-syntax*
begin

lemma *Domainp-forall-transfer* [*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> (=)) ===> (=))
(*transfer-bforall* (*Domainp A*)) *transfer-forall*
using *assms* **unfolding** *right-total-def*
unfolding *transfer-forall-def transfer-bforall-def rel-fun-def Domainp-iff*
by *fast*

Transfer rules using implication instead of equality on booleans.

lemma *transfer-forall-transfer* [*transfer-rule*]:
bi-total A ⟹ ((*A* ===> (=)) ===> (=)) *transfer-forall transfer-forall*
right-total A ⟹ ((*A* ===> (=)) ===> *implies*) *transfer-forall transfer-forall*
right-total A ⟹ ((*A* ===> *implies*) ===> *implies*) *transfer-forall transfer-forall*
bi-total A ⟹ ((*A* ===> (=)) ===> *rev-implies*) *transfer-forall transfer-forall*
bi-total A ⟹ ((*A* ===> *rev-implies*) ===> *rev-implies*) *transfer-forall transfer-forall*
unfolding *transfer-forall-def rev-implies-def rel-fun-def right-total-def bi-total-def*
by *fast+*

lemma *transfer-implies-transfer* [*transfer-rule*]:
((=) ===> (=) ===> (=)) *transfer-implies transfer-implies*
(*rev-implies* ===> *implies* ===> *implies*) *transfer-implies transfer-implies*
(*rev-implies* ===> (=) ===> *implies*) *transfer-implies transfer-implies*
((=) ===> *implies* ===> *implies*) *transfer-implies transfer-implies*
((=) ===> (=) ===> *implies*) *transfer-implies transfer-implies*
(*implies* ===> *rev-implies* ===> *rev-implies*) *transfer-implies transfer-implies*
(*implies* ===> (=) ===> *rev-implies*) *transfer-implies transfer-implies*
((=) ===> *rev-implies* ===> *rev-implies*) *transfer-implies transfer-implies*
((=) ===> (=) ===> *rev-implies*) *transfer-implies transfer-implies*
unfolding *transfer-implies-def rev-implies-def rel-fun-def* **by** *auto*

lemma *eq-imp-transfer* [*transfer-rule*]:
right-unique A ⟹ (*A* ===> *A* ===> (⟶)) (=) (=)
unfolding *right-unique-alt-def2* **.**

Transfer rules using equality.

lemma *left-unique-transfer* [*transfer-rule*]:
assumes *right-total A*
assumes *right-total B*
assumes *bi-unique A*
shows ((*A* ===> *B* ===> (=)) ===> *implies*) *left-unique left-unique*
using *assms* **unfolding** *left-unique-def right-total-def bi-unique-def rel-fun-def*
by *metis*

lemma *eq-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *A* ===> (=)) (=) (=)
using *assms* **unfolding** *bi-unique-def rel-fun-def* **by** *auto*

lemma *right-total-Ex-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> (=)) ===> (=)) (*Bex* (*Collect* (*Domainp A*))) *Ex*
using *assms* **unfolding** *right-total-def Bex-def rel-fun-def Domainp-iff*
by *fast*

lemma *right-total-All-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> (=)) ===> (=)) (*Ball* (*Collect* (*Domainp A*))) *All*
using *assms* **unfolding** *right-total-def Ball-def rel-fun-def Domainp-iff*
by *fast*

context
includes *lifting-syntax*
begin

lemma *right-total-fun-eq-transfer*:
assumes [*transfer-rule*]: *right-total A bi-unique B*
shows ((*A* ===> *B*) ===> (*A* ===> *B*) ===> (=)) (λ*f g*. ∀ *x*∈*Collect*(*Domainp A*). *f x* = *g x*) (=)
unfolding *fun-eq-iff*
by *transfer-prover*

end

lemma *All-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> (=)) ===> (=)) *All All*
using *assms* **unfolding** *bi-total-def rel-fun-def* **by** *fast*

lemma *Ex-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> (=)) ===> (=)) *Ex Ex*
using *assms* **unfolding** *bi-total-def rel-fun-def* **by** *fast*

lemma *Ex1-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A*
shows ((*A* ===> (=)) ===> (=)) *Ex1 Ex1*
unfolding *Ex1-def* **by** *transfer-prover*

declare *If-transfer* [*transfer-rule*]

lemma *Let-transfer* [*transfer-rule*]: (*A* ===> (*A* ===> *B*) ===> *B*) *Let Let*
unfolding *rel-fun-def* **by** *simp*

declare *id-transfer* [*transfer-rule*]

declare *comp-transfer* [*transfer-rule*]

lemma *curry-transfer* [*transfer-rule*]:
(*(rel-prod A B ===> C) ===> A ===> B ===> C) curry curry*
unfolding *curry-def* **by** *transfer-prover*

lemma *fun-upd-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*(A ===> B) ===> A ===> B ===> A ===> B) fun-upd fun-upd*
unfolding *fun-upd-def* **by** *transfer-prover*

lemma *case-nat-transfer* [*transfer-rule*]:
(*A ===> ((=) ===> A) ===> (=) ===> A) case-nat case-nat*
unfolding *rel-fun-def* **by** (*simp split*: *nat.split*)

lemma *rec-nat-transfer* [*transfer-rule*]:
(*A ===> ((=) ===> A ===> A) ===> (=) ===> A) rec-nat rec-nat*
unfolding *rel-fun-def*
apply *safe*
subgoal for - - - - - *n*
by (*induction n*) *simp-all*
done

lemma *funpow-transfer* [*transfer-rule*]:
(*(=) ===> (A ===> A) ===> (A ===> A)) compow compow*
unfolding *funpow-def* **by** *transfer-prover*

lemma *mono-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
assumes [*transfer-rule*]: (*A ===> A ===> (=)*) (≤) (≤)
assumes [*transfer-rule*]: (*B ===> B ===> (=)*) (≤) (≤)
shows (*(A ===> B) ===> (=)*) *mono mono*
unfolding *mono-def* **by** *transfer-prover*

lemma *right-total-relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*
shows (*(A ===> B ===> (=)) ===> (B ===> C ===> (=)) ===> A ===> C ===> (=)*)
(*λR S x z. ∃ y∈Collect (Domainp B). R x y ∧ S y z*) (*OO*)
unfolding *OO-def* **by** *transfer-prover*

lemma *relcompp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows (*(A ===> B ===> (=)) ===> (B ===> C ===> (=)) ===> A ===> C ===> (=)*) (*OO*) (*OO*)
unfolding *OO-def* **by** *transfer-prover*

lemma *right-total-Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *right-total B*

shows $((A ===> B ===> (=)) ===> A ===> (=))$ $(\lambda T\, x.\, \exists\, y{\in}Collect(Domainp\ B).\ T\ x\ y)$ *Domainp*
apply(*subst*(*2*) *Domainp-iff*[*abs-def*]) **by** *transfer-prover*

lemma *Domainp-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total B*
shows $((A ===> B ===> (=)) ===> A ===> (=))$ *Domainp Domainp*
unfolding *Domainp-iff* **by** *transfer-prover*

lemma *reflp-transfer*[*transfer-rule*]:
bi-total A $\Longrightarrow$ $((A ===> A ===> (=)) ===> (=))$ *reflp reflp*
right-total A $\Longrightarrow$ $((A ===> A ===> implies) ===> implies)$ *reflp reflp*
right-total A $\Longrightarrow$ $((A ===> A ===> (=)) ===> implies)$ *reflp reflp*
bi-total A $\Longrightarrow$ $((A ===> A ===> rev\text{-}implies) ===> rev\text{-}implies)$ *reflp reflp*
bi-total A $\Longrightarrow$ $((A ===> A ===> (=)) ===> rev\text{-}implies)$ *reflp reflp*
unfolding *reflp-def rev-implies-def bi-total-def right-total-def rel-fun-def*
by *fast+*

lemma *right-unique-transfer* [*transfer-rule*]:
$[\![$ *right-total A*; *right-total B*; *bi-unique B* $]\!]$
$\Longrightarrow$ $((A ===> B ===> (=)) ===> implies)$ *right-unique right-unique*
unfolding *right-unique-def right-total-def bi-unique-def rel-fun-def*
by *metis*

lemma *left-total-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-total B*
shows $((A ===> B ===> (=)) ===> (=))$ *left-total left-total*
unfolding *left-total-def* **by** *transfer-prover*

lemma *right-total-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-total B*
shows $((A ===> B ===> (=)) ===> (=))$ *right-total right-total*
unfolding *right-total-def* **by** *transfer-prover*

lemma *left-unique-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A bi-total B*
shows $((A ===> B ===> (=)) ===> (=))$ *left-unique left-unique*
unfolding *left-unique-def* **by** *transfer-prover*

lemma *prod-pred-parametric* [*transfer-rule*]:
$((A ===> (=)) ===> (B ===> (=)) ===> rel\text{-}prod\ A\ B ===> (=))$
pred-prod pred-prod
unfolding *prod.pred-set Basic-BNFs.fsts-def Basic-BNFs.snds-def fstsp.simps sndsp.simps*

by *simp transfer-prover*

lemma *apfst-parametric* [*transfer-rule*]:
$((A ===> B) ===> rel\text{-}prod\ A\ C ===> rel\text{-}prod\ B\ C)$ *apfst apfst*
unfolding *apfst-def* **by** *transfer-prover*

lemma *rel-fun-eq-eq-onp*: $((=) ===> \mathit{eq\text{-}onp}\ P) = \mathit{eq\text{-}onp}\ (\lambda f.\ \forall x.\ P(f\ x))$
unfolding *eq-onp-def rel-fun-def* **by** *auto*

lemma *rel-fun-eq-onp-rel*:
shows $((\mathit{eq\text{-}onp}\ R) ===> S) = (\lambda f\ g.\ \forall x.\ R\ x \longrightarrow S\ (f\ x)\ (g\ x))$
by (*auto simp add*: *eq-onp-def rel-fun-def*)

lemma *eq-onp-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows $((A ===> (=)) ===> A ===> A ===> (=))$ *eq-onp eq-onp*
unfolding *eq-onp-def* **by** *transfer-prover*

lemma *rtranclp-parametric* [*transfer-rule*]:
assumes *bi-unique A bi-total A*
shows $((A ===> A ===> (=)) ===> A ===> A ===> (=))$ *rtranclp rtranclp*
proof(*rule rel-funI iffI*)+
fix $R :: 'a \Rightarrow 'a \Rightarrow \mathit{bool}$ **and** $R'\ x\ y\ x'\ y'$
assume *R*: $(A ===> A ===> (=))\ R\ R'$ **and** $A\ x\ x'$
{
assume $R^{**}\ x\ y\ A\ y\ y'$
thus $R'^{**}\ x'\ y'$
proof(*induction arbitrary*: y')
case *base*
with ‹*bi-unique A*› ‹$A\ x\ x'$› **have** $x' = y'$ **by**(*rule bi-uniqueDr*)
thus *?case* **by** *simp*
next
case (*step y z z'*)
from ‹*bi-total A*› **obtain** y' **where** $A\ y\ y'$ **unfolding** *bi-total-def* **by** *blast*
hence $R'^{**}\ x'\ y'$ **by**(*rule step.IH*)
moreover from R ‹$A\ y\ y'$› ‹$A\ z\ z'$› ‹$R\ y\ z$›
have $R'\ y'\ z'$ **by**(*auto dest*: *rel-funD*)
ultimately show *?case* **..**
qed
next
assume $R'^{**}\ x'\ y'\ A\ y\ y'$
thus $R^{**}\ x\ y$
proof(*induction arbitrary*: *y*)
case *base*
with ‹*bi-unique A*› ‹$A\ x\ x'$› **have** $x = y$ **by**(*rule bi-uniqueDl*)
thus *?case* **by** *simp*
next
case (*step y' z' z*)
from ‹*bi-total A*› **obtain** *y* **where** $A\ y\ y'$ **unfolding** *bi-total-def* **by** *blast*
hence $R^{**}\ x\ y$ **by**(*rule step.IH*)
moreover from R ‹$A\ y\ y'$› ‹$A\ z\ z'$› ‹$R'\ y'\ z'$›
have $R\ y\ z$ **by**(*auto dest*: *rel-funD*)
ultimately show *?case* **..**

qed
}
qed

lemma *right-unique-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B bi-total B*
shows ((*A* ===> *B* ===> (=)) ===> (=)) *right-unique right-unique*
unfolding *right-unique-def* **by** *transfer-prover*

lemma *map-fun-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *C*) ===> *A* ===> *D*) *map-fun map-fun*
unfolding *map-fun-def* **by** *transfer-prover*

end

42.6 *of-bool* and *of-nat*

context
includes *lifting-syntax*
begin

lemma *transfer-rule-of-bool*:
‹((⟷) ===> (≅)) *of-bool of-bool*›
if [*transfer-rule*]: ‹*0* ≅ *0*› ‹*1* ≅ *1*›
for *R* :: ‹'*a*::*zero-neq-one* ⇒ '*b*::*zero-neq-one* ⇒ *bool*› (**infix** ‹≅› *50*)
unfolding *of-bool-def* **by** *transfer-prover*

lemma *transfer-rule-of-nat*:
((=) ===> (≅)) *of-nat of-nat*
if [*transfer-rule*]: ‹*0* ≅ *0*› ‹*1* ≅ *1*›
‹((≅) ===> (≅) ===> (≅)) (+) (+)›
for *R* :: ‹'*a*::*semiring-1* ⇒ '*b*::*semiring-1* ⇒ *bool*› (**infix** ‹≅› *50*)
unfolding *of-nat-def* **by** *transfer-prover*

end

end

43 Lifting package

theory *Lifting*
imports *Equiv-Relations Transfer*
keywords
parametric **and**
print-quot-maps print-quotients :: *diag* **and**
lift-definition :: *thy-goal-defn* **and**
setup-lifting lifting-forget lifting-update :: *thy-decl*
begin

43.1 Function map

context includes *lifting-syntax*
begin

lemma *map-fun-id*:
(*id* −−−> *id*) = *id*
by (*simp add*: *fun-eq-iff*)

43.2 Quotient Predicate

definition *Quotient* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*b*) ⇒ ('*b* ⇒ '*a*) ⇒ ('*a* ⇒ '*b* ⇒ *bool*) ⇒ *bool*
where
Quotient R Abs Rep T ⟷
(∀ *a*. *Abs* (*Rep a*) = *a*) ∧
(∀ *a*. *R* (*Rep a*) (*Rep a*)) ∧
(∀ *r s*. *R r s* ⟷ *R r r* ∧ *R s s* ∧ *Abs r* = *Abs s*) ∧
T = (λ*x y*. *R x x* ∧ *Abs x* = *y*)

lemma *QuotientI*:
assumes ⋀*a*. *Abs* (*Rep a*) = *a*
and ⋀*a*. *R* (*Rep a*) (*Rep a*)
and ⋀*r s*. *R r s* ⟷ *R r r* ∧ *R s s* ∧ *Abs r* = *Abs s*
and *T* = (λ*x y*. *R x x* ∧ *Abs x* = *y*)
shows *Quotient R Abs Rep T*
using *assms* **unfolding** *Quotient-def* **by** *blast*

context
fixes *R Abs Rep T*
assumes *a*: *Quotient R Abs Rep T*
begin

lemma *Quotient-abs-rep*: *Abs* (*Rep a*) = *a*
using *a* **unfolding** *Quotient-def*
by *simp*

lemma *Quotient-rep-reflp*: *R* (*Rep a*) (*Rep a*)
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rel*:
R r r ∧ *R s s* ∧ *Abs r* = *Abs s* ⟷ *R r s* — orientation does not loop on rewriting
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-cr-rel*: *T* = (λ*x y*. *R x x* ∧ *Abs x* = *y*)
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-refl1*: $R\ r\ s \Longrightarrow R\ r\ r$
using *a* **unfolding** *Quotient-def*
by *fast*

lemma *Quotient-refl2*: $R\ r\ s \Longrightarrow R\ s\ s$
using *a* **unfolding** *Quotient-def*
by *fast*

lemma *Quotient-rel-rep*: $R\ (Rep\ a)\ (Rep\ b) \longleftrightarrow a = b$
using *a* **unfolding** *Quotient-def*
by *metis*

lemma *Quotient-rep-abs*: $R\ r\ r \Longrightarrow R\ (Rep\ (Abs\ r))\ r$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rep-abs-eq*: $R\ t\ t \Longrightarrow R \leq (=) \Longrightarrow Rep\ (Abs\ t) = t$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rep-abs-fold-unmap*:
assumes $x' \equiv Abs\ x$ **and** $R\ x\ x$ **and** $Rep\ x' \equiv Rep'\ x'$
shows $R\ (Rep'\ x')\ x$
proof −
have $R\ (Rep\ x')\ x$ **using** *assms*(*1*−*2*) *Quotient-rep-abs* **by** *auto*
then show *?thesis* **using** *assms*(*3*) **by** *simp*
qed

lemma *Quotient-Rep-eq*:
assumes $x' \equiv Abs\ x$
shows $Rep\ x' \equiv Rep\ x'$
by *simp*

lemma *Quotient-rel-abs*: $R\ r\ s \Longrightarrow Abs\ r = Abs\ s$
using *a* **unfolding** *Quotient-def*
by *blast*

lemma *Quotient-rel-abs2*:
assumes $R\ (Rep\ x)\ y$
shows $x = Abs\ y$
proof −
from *assms* **have** $Abs\ (Rep\ x) = Abs\ y$ **by** (*auto intro*: *Quotient-rel-abs*)
then show *?thesis* **using** *assms*(*1*) **by** (*simp add*: *Quotient-abs-rep*)
qed

lemma *Quotient-symp*: *symp R*
using *a* **unfolding** *Quotient-def* **using** *sympI* **by** (*metis* (*full-types*))

lemma *Quotient-transp*: *transp R*

using *a* **unfolding** *Quotient-def* **using** *transpI* **by** (*metis* (*full-types*))

lemma *Quotient-part-equivp*: *part-equivp R*
by (*metis Quotient-rep-reflp Quotient-symp Quotient-transp part-equivpI*)

end

lemma *identity-quotient*: *Quotient* (=) *id id* (=)
unfolding *Quotient-def* **by** *simp*

TODO: Use one of these alternatives as the real definition.

lemma *Quotient-alt-def*:
Quotient R Abs Rep T ⟷
(∀ *a b*. *T a b* ⟶ *Abs a* = *b*) ∧
(∀ *b*. *T* (*Rep b*) *b*) ∧
(∀ *x y*. *R x y* ⟷ *T x* (*Abs x*) ∧ *T y* (*Abs y*) ∧ *Abs x* = *Abs y*)
apply *safe*
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*simp* (*no-asm-use*) *only*: *Quotient-def*, *fast*)
apply (*rule QuotientI*)
apply *simp*
apply *metis*
apply *simp*
apply (*rule ext*, *rule ext*, *metis*)
done

lemma *Quotient-alt-def2*:
Quotient R Abs Rep T ⟷
(∀ *a b*. *T a b* ⟶ *Abs a* = *b*) ∧
(∀ *b*. *T* (*Rep b*) *b*) ∧
(∀ *x y*. *R x y* ⟷ *T x* (*Abs y*) ∧ *T y* (*Abs x*))
unfolding *Quotient-alt-def* **by** (*safe*, *metis*+)

lemma *Quotient-alt-def3*:
Quotient R Abs Rep T ⟷
(∀ *a b*. *T a b* ⟶ *Abs a* = *b*) ∧ (∀ *b*. *T* (*Rep b*) *b*) ∧
(∀ *x y*. *R x y* ⟷ (∃ *z*. *T x z* ∧ *T y z*))
unfolding *Quotient-alt-def2* **by** (*safe*, *metis*+)

lemma *Quotient-alt-def4*:
Quotient R Abs Rep T ⟷
(∀ *a b*. *T a b* ⟶ *Abs a* = *b*) ∧ (∀ *b*. *T* (*Rep b*) *b*) ∧ *R* = *T OO conversep T*
unfolding *Quotient-alt-def3 fun-eq-iff* **by** *auto*

lemma *Quotient-alt-def5*:

Quotient R Abs Rep T $\longleftrightarrow$
T $\leq$ *BNF-Def.Grp UNIV Abs* $\wedge$ *BNF-Def.Grp UNIV Rep* $\leq$ T^{-1-1} $\wedge$ *R* = *T OO* T^{-1-1}
unfolding *Quotient-alt-def4 Grp-def* **by** *blast*

lemma *fun-quotient*:
assumes *1*: *Quotient R1 abs1 rep1 T1*
assumes *2*: *Quotient R2 abs2 rep2 T2*
shows *Quotient* (*R1* ===> *R2*) (*rep1* ---> *abs2*) (*abs1* ---> *rep2*) (*T1* ===> *T2*)
using *assms* **unfolding** *Quotient-alt-def2*
unfolding *rel-fun-def fun-eq-iff map-fun-apply*
by (*safe*, *metis+*)

lemma *apply-rsp*:
fixes *f g*::'*a* $\Rightarrow$ '*c*
assumes *q*: *Quotient R1 Abs1 Rep1 T1*
and *a*: (*R1* ===> *R2*) *f g R1 x y*
shows *R2* (*f x*) (*g y*)
using *a* **by** (*auto elim*: *rel-funE*)

lemma *apply-rsp'*:
assumes *a*: (*R1* ===> *R2*) *f g R1 x y*
shows *R2* (*f x*) (*g y*)
using *a* **by** (*auto elim*: *rel-funE*)

lemma *apply-rsp''*:
assumes *Quotient R Abs Rep T*
and (*R* ===> *S*) *f f*
shows *S* (*f* (*Rep x*)) (*f* (*Rep x*))
proof −
from *assms*(*1*) **have** *R* (*Rep x*) (*Rep x*) **by** (*rule Quotient-rep-reflp*)
then show *?thesis* **using** *assms*(*2*) **by** (*auto intro*: *apply-rsp'*)
qed

43.3 Quotient composition

lemma *Quotient-compose*:
assumes *1*: *Quotient R1 Abs1 Rep1 T1*
assumes *2*: *Quotient R2 Abs2 Rep2 T2*
shows *Quotient* (*T1 OO R2 OO conversep T1*) (*Abs2* $\circ$ *Abs1*) (*Rep1* $\circ$ *Rep2*) (*T1 OO T2*)
using *assms* **unfolding** *Quotient-alt-def4* **by** *fastforce*

lemma *equivp-reflp2*:
equivp R $\Longrightarrow$ *reflp R*
by (*erule equivpE*)

43.4 Respects predicate

definition *Respects* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set$
  where *Respects R* = $\{x.\ R\ x\ x\}$

lemma *in-respects*: $x \in Respects\ R \longleftrightarrow R\ x\ x$
  unfolding *Respects-def* **by** *simp*

lemma *UNIV-typedef-to-Quotient*:
  assumes *type-definition Rep Abs UNIV*
  and *T-def*: $T \equiv (\lambda x\ y.\ x = Rep\ y)$
  shows *Quotient* (=) *Abs Rep T*
proof −
  interpret *type-definition Rep Abs UNIV* **by** *fact*
  from *Abs-inject Rep-inverse Abs-inverse T-def* **show** *?thesis*
    by (*fastforce intro*!: *QuotientI fun-eq-iff*)
qed

lemma *UNIV-typedef-to-equivp*:
  fixes *Abs* :: $'a \Rightarrow 'b$
  and *Rep* :: $'b \Rightarrow 'a$
  assumes *type-definition Rep Abs* (*UNIV*::$'a\ set$)
  shows *equivp* ((=) ::$'a \Rightarrow 'a \Rightarrow bool$)
by (*rule identity-equivp*)

lemma *typedef-to-Quotient*:
  assumes *type-definition Rep Abs S*
  and *T-def*: $T \equiv (\lambda x\ y.\ x = Rep\ y)$
  shows *Quotient* (*eq-onp* $(\lambda x.\ x \in S)$) *Abs Rep T*
proof −
  interpret *type-definition Rep Abs S* **by** *fact*
  from *Rep Abs-inject Rep-inverse Abs-inverse T-def* **show** *?thesis*
    by (*auto intro*!: *QuotientI simp*: *eq-onp-def fun-eq-iff*)
qed

lemma *typedef-to-part-equivp*:
  assumes *type-definition Rep Abs S*
  shows *part-equivp* (*eq-onp* $(\lambda x.\ x \in S)$)
proof (*intro part-equivpI*)
  interpret *type-definition Rep Abs S* **by** *fact*
  show $\exists x.$ *eq-onp* $(\lambda x.\ x \in S)\ x\ x$ **using** *Rep* **by** (*auto simp*: *eq-onp-def*)
next
  show *symp* (*eq-onp* $(\lambda x.\ x \in S)$) **by** (*auto intro*: *sympI simp*: *eq-onp-def*)
next
  show *transp* (*eq-onp* $(\lambda x.\ x \in S)$) **by** (*auto intro*: *transpI simp*: *eq-onp-def*)
qed

lemma *open-typedef-to-Quotient*:
  assumes *type-definition Rep Abs* $\{x.\ P\ x\}$
  and *T-def*: $T \equiv (\lambda x\ y.\ x = Rep\ y)$

shows *Quotient* (*eq-onp P*) *Abs Rep T*
using *typedef-to-Quotient* [*OF assms*] **by** *simp*

lemma *open-typedef-to-part-equivp*:
assumes *type-definition Rep Abs* {*x*. *P x*}
shows *part-equivp* (*eq-onp P*)
using *typedef-to-part-equivp* [*OF assms*] **by** *simp*

lemma *type-definition-Quotient-not-empty*: *Quotient* (*eq-onp P*) *Abs Rep T* ⟹ ∃ *x*. *P x*
unfolding *eq-onp-def* **by** (*drule Quotient-rep-reflp*) *blast*

lemma *type-definition-Quotient-not-empty-witness*: *Quotient* (*eq-onp P*) *Abs Rep T* ⟹ *P* (*Rep undefined*)
unfolding *eq-onp-def* **by** (*drule Quotient-rep-reflp*) *blast*

Generating transfer rules for quotients.

context
fixes *R Abs Rep T*
assumes *1*: *Quotient R Abs Rep T*
begin

lemma *Quotient-right-unique*: *right-unique T*
using *1* **unfolding** *Quotient-alt-def right-unique-def* **by** *metis*

lemma *Quotient-right-total*: *right-total T*
using *1* **unfolding** *Quotient-alt-def right-total-def* **by** *metis*

lemma *Quotient-rel-eq-transfer*: (*T* ===> *T* ===> (=)) *R* (=)
using *1* **unfolding** *Quotient-alt-def rel-fun-def* **by** *simp*

lemma *Quotient-abs-induct*:
assumes ⋀*y*. *R y y* ⟹ *P* (*Abs y*) **shows** *P x*
using *1 assms* **unfolding** *Quotient-def* **by** *metis*

end

Generating transfer rules for total quotients.

context
fixes *R Abs Rep T*
assumes *1*: *Quotient R Abs Rep T* **and** *2*: *reflp R*
begin

lemma *Quotient-left-total*: *left-total T*
using *1 2* **unfolding** *Quotient-alt-def left-total-def reflp-def* **by** *auto*

lemma *Quotient-bi-total*: *bi-total T*
using *1 2* **unfolding** *Quotient-alt-def bi-total-def reflp-def* **by** *auto*

lemma *Quotient-id-abs-transfer*: $((=) ===> T)\ (\lambda x.\ x)\ Abs$
using *1 2* **unfolding** *Quotient-alt-def reflp-def rel-fun-def* **by** *simp*

lemma *Quotient-total-abs-induct*: $(\bigwedge y.\ P\ (Abs\ y)) \Longrightarrow P\ x$
using *1 2* **unfolding** *Quotient-alt-def reflp-def* **by** *metis*

lemma *Quotient-total-abs-eq-iff*: $Abs\ x = Abs\ y \longleftrightarrow R\ x\ y$
using *Quotient-rel* [*OF 1*] *2* **unfolding** *reflp-def* **by** *simp*

end

Generating transfer rules for a type defined with *typedef*.

context
fixes *Rep Abs A T*
assumes *type*: *type-definition Rep Abs A*
assumes *T-def*: $T \equiv (\lambda(x::'a)\ (y::'b).\ x = Rep\ y)$
begin

lemma *typedef-left-unique*: *left-unique T*
unfolding *left-unique-def T-def*
by (*simp add*: *type-definition.Rep-inject* [*OF type*])

lemma *typedef-bi-unique*: *bi-unique T*
unfolding *bi-unique-def T-def*
by (*simp add*: *type-definition.Rep-inject* [*OF type*])

lemma *typedef-right-unique*: *right-unique T*
using *T-def type Quotient-right-unique typedef-to-Quotient*
by *blast*

lemma *typedef-right-total*: *right-total T*
using *T-def type Quotient-right-total typedef-to-Quotient*
by *blast*

lemma *typedef-rep-transfer*: $(T ===> (=))\ (\lambda x.\ x)\ Rep$
unfolding *rel-fun-def T-def* **by** *simp*

end

Generating the correspondence rule for a constant defined with *lift-definition*.

lemma *Quotient-to-transfer*:
assumes *Quotient R Abs Rep T* **and** *R c c* **and** $c' \equiv Abs\ c$
shows *T c c'*
using *assms* **by** (*auto dest*: *Quotient-cr-rel*)

Proving reflexivity

lemma *Quotient-to-left-total*:

assumes *q*: *Quotient R Abs Rep T*
and *r-R*: *reflp R*
shows *left-total T*
using *r-R Quotient-cr-rel*[*OF q*] **unfolding** *left-total-def* **by** (*auto elim*: *reflpE*)

lemma *Quotient-composition-ge-eq*:
assumes *left-total T*
assumes $R \geq (=)$
shows $(T\ OO\ R\ OO\ T^{-1-1}) \geq (=)$
using *assms* **unfolding** *left-total-def* **by** *fast*

lemma *Quotient-composition-le-eq*:
assumes *left-unique T*
assumes $R \leq (=)$
shows $(T\ OO\ R\ OO\ T^{-1-1}) \leq (=)$
using *assms* **unfolding** *left-unique-def* **by** *blast*

lemma *eq-onp-le-eq*:
eq-onp P $\leq (=)$ **unfolding** *eq-onp-def* **by** *blast*

lemma *reflp-ge-eq*:
reflp R $\Longrightarrow R \geq (=)$ **unfolding** *reflp-def* **by** *blast*

Proving a parametrized correspondence relation

definition *POS* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
POS A B $\equiv A \leq B$

definition *NEG* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b \Rightarrow bool) \Rightarrow bool$ **where**
NEG A B $\equiv B \leq A$

lemma *pos-OO-eq*:
shows *POS* $(A\ OO\ (=))\ A$
unfolding *POS-def OO-def* **by** *blast*

lemma *pos-eq-OO*:
shows *POS* $((=)\ OO\ A)\ A$
unfolding *POS-def OO-def* **by** *blast*

lemma *neg-OO-eq*:
shows *NEG* $(A\ OO\ (=))\ A$
unfolding *NEG-def OO-def* **by** *auto*

lemma *neg-eq-OO*:
shows *NEG* $((=)\ OO\ A)\ A$
unfolding *NEG-def OO-def* **by** *blast*

lemma *POS-trans*:
assumes *POS A B*
assumes *POS B C*

shows *POS A C*
using *assms* **unfolding** *POS-def* **by** *auto*

lemma *NEG-trans*:
assumes *NEG A B*
assumes *NEG B C*
shows *NEG A C*
using *assms* **unfolding** *NEG-def* **by** *auto*

lemma *POS-NEG*:
POS A B ≡ *NEG B A*
unfolding *POS-def NEG-def* **by** *auto*

lemma *NEG-POS*:
NEG A B ≡ *POS B A*
unfolding *POS-def NEG-def* **by** *auto*

lemma *POS-pcr-rule*:
assumes *POS* (*A OO B*) *C*
shows *POS* (*A OO B OO X*) (*C OO X*)
using *assms* **unfolding** *POS-def OO-def* **by** *blast*

lemma *NEG-pcr-rule*:
assumes *NEG* (*A OO B*) *C*
shows *NEG* (*A OO B OO X*) (*C OO X*)
using *assms* **unfolding** *NEG-def OO-def* **by** *blast*

lemma *POS-apply*:
assumes *POS R R′*
assumes *R f g*
shows *R′ f g*
using *assms* **unfolding** *POS-def* **by** *auto*

Proving a parametrized correspondence relation

lemma *fun-mono*:
assumes *A* ≥ *C*
assumes *B* ≤ *D*
shows (*A* ===> *B*) ≤ (*C* ===> *D*)
using *assms* **unfolding** *rel-fun-def* **by** *blast*

lemma *pos-fun-distr*: ((*R* ===> *S*) *OO* (*R′* ===> *S′*)) ≤ ((*R OO R′*) ===> (*S OO S′*))
unfolding *OO-def rel-fun-def* **by** *blast*

lemma *functional-relation*: *right-unique R* ⟹ *left-total R* ⟹ ∀ *x*. ∃!*y*. *R x y*
unfolding *right-unique-def left-total-def* **by** *blast*

lemma *functional-converse-relation*: *left-unique R* ⟹ *right-total R* ⟹ ∀ *y*. ∃!*x*. *R x y*

unfolding *left-unique-def right-total-def* **by** *blast*

lemma *neg-fun-distr1*:
assumes *1*: *left-unique R right-total R*
assumes *2*: *right-unique R′ left-total R′*
shows $(R\ OO\ R' ===>\ S\ OO\ S') \leq ((R ===>\ S)\ OO\ (R' ===>\ S'))$
using *functional-relation*[*OF 2*] *functional-converse-relation*[*OF 1*]
unfolding *rel-fun-def OO-def*
apply *clarify*
apply (*subst all-comm*)
apply (*subst all-conj-distrib*[*symmetric*])
apply (*intro choice*)
by *metis*

lemma *neg-fun-distr2*:
assumes *1*: *right-unique R′ left-total R′*
assumes *2*: *left-unique S′ right-total S′*
shows $(R\ OO\ R' ===>\ S\ OO\ S') \leq ((R ===>\ S)\ OO\ (R' ===>\ S'))$
using *functional-converse-relation*[*OF 2*] *functional-relation*[*OF 1*]
unfolding *rel-fun-def OO-def*
apply *clarify*
apply (*subst all-comm*)
apply (*subst all-conj-distrib*[*symmetric*])
apply (*intro choice*)
by *metis*

43.5 Domains

lemma *composed-equiv-rel-eq-onp*:
assumes *left-unique R*
assumes $(R ===>\ (=))\ P\ P'$
assumes *Domainp R* = *P″*
shows $(R\ OO\ eq\text{-}onp\ P'\ OO\ R^{-1-1}) = eq\text{-}onp\ (inf\ P''\ P)$
using *assms* **unfolding** *OO-def conversep-iff Domainp-iff*[*abs-def*] *left-unique-def rel-fun-def eq-onp-def fun-eq-iff* **by** *blast*

lemma *composed-equiv-rel-eq-eq-onp*:
assumes *left-unique R*
assumes *Domainp R* = *P*
shows $(R\ OO\ (=)\ OO\ R^{-1-1}) = eq\text{-}onp\ P$
using *assms* **unfolding** *OO-def conversep-iff Domainp-iff*[*abs-def*] *left-unique-def eq-onp-def fun-eq-iff is-equality-def* **by** *metis*

lemma *pcr-Domainp-par-left-total*:
assumes *Domainp B* = *P*
assumes *left-total A*
assumes $(A ===>\ (=))\ P'\ P$

shows *Domainp* (*A OO B*) = *P'*
using *assms*
unfolding *Domainp-iff*[*abs-def*] *OO-def bi-unique-def left-total-def rel-fun-def*
by (*fast intro*: *fun-eq-iff*)

lemma *pcr-Domainp-par*:
assumes *Domainp B* = *P2*
assumes *Domainp A* = *P1*
assumes (*A* ===> (=)) *P2' P2*
shows *Domainp* (*A OO B*) = (*inf P1 P2'*)
using *assms* **unfolding** *rel-fun-def Domainp-iff*[*abs-def*] *OO-def*
by (*fast intro*: *fun-eq-iff*)

definition *rel-pred-comp* :: ('*a* => '*b* => *bool*) => ('*b* => *bool*) => '*a* => *bool*
where *rel-pred-comp R P* ≡ λ*x*. ∃ *y*. *R x y* ∧ *P y*

lemma *pcr-Domainp*:
assumes *Domainp B* = *P*
shows *Domainp* (*A OO B*) = (λ*x*. ∃ *y*. *A x y* ∧ *P y*)
using *assms* **by** *blast*

lemma *pcr-Domainp-total*:
assumes *left-total B*
assumes *Domainp A* = *P*
shows *Domainp* (*A OO B*) = *P*
using *assms* **unfolding** *left-total-def*
by *fast*

lemma *Quotient-to-Domainp*:
assumes *Quotient R Abs Rep T*
shows *Domainp T* = (λ*x*. *R x x*)
by (*simp add*: *Domainp-iff*[*abs-def*] *Quotient-cr-rel*[*OF assms*])

lemma *eq-onp-to-Domainp*:
assumes *Quotient* (*eq-onp P*) *Abs Rep T*
shows *Domainp T* = *P*
by (*simp add*: *eq-onp-def Domainp-iff*[*abs-def*] *Quotient-cr-rel*[*OF assms*])

end

lemma *right-total-UNIV-transfer*:
assumes *right-total A*
shows (*rel-set A*) (*Collect* (*Domainp A*)) *UNIV*
using *assms* **unfolding** *right-total-def rel-set-def Domainp-iff* **by** *blast*

43.6 ML setup

ML-file ‹*Tools/Lifting/lifting-util.ML*›

named-theorems *relator-eq-onp*
theorems that a relator of an eq-onp is an eq-onp of the corresponding predicate
ML-file ‹*Tools/Lifting/lifting-info.ML*›

declare *fun-quotient*[*quot-map*]
declare *fun-mono*[*relator-mono*]
lemmas [*relator-distr*] = *pos-fun-distr neg-fun-distr1 neg-fun-distr2*

ML-file ‹*Tools/Lifting/lifting-bnf.ML*›
ML-file ‹*Tools/Lifting/lifting-term.ML*›
ML-file ‹*Tools/Lifting/lifting-def.ML*›
ML-file ‹*Tools/Lifting/lifting-setup.ML*›
ML-file ‹*Tools/Lifting/lifting-def-code-dt.ML*›

lemma *pred-prod-beta*: *pred-prod P Q xy* ⟷ *P* (*fst xy*) ∧ *Q* (*snd xy*)
by(*cases xy*) *simp*

lemma *pred-prod-split*: *P* (*pred-prod Q R xy*) ⟷ (∀ *x y*. *xy* = (*x*, *y*) ⟶ *P* (*Q x* ∧ *R y*))
by(*cases xy*) *simp*

hide-const (**open**) *POS NEG*

end

44 Definition of Quotient Types

theory *Quotient*
imports *Lifting*
keywords
print-quotmapsQ3 print-quotientsQ3 print-quotconsts :: *diag* **and**
quotient-type :: *thy-goal-defn* **and** / **and**
quotient-definition :: *thy-goal-defn* **and**
copy-bnf :: *thy-defn* **and**
lift-bnf :: *thy-goal-defn*
begin

Basic definition for equivalence relations that are represented by predicates.

Composition of Relations

abbreviation
rel-conj :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*b* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*b* ⇒ *bool* (**infixr** ‹*OOO*› *75*)
where
r1 OOO r2 ≡ *r1 OO r2 OO r1*

lemma *eq-comp-r*:

shows $((=)\ OOO\ R) = R$
by (*auto simp add*: *fun-eq-iff*)

context includes *lifting-syntax*
begin

44.1 Quotient Predicate

definition
$Quotient3\ R\ Abs\ Rep \longleftrightarrow$
$(\forall a.\ Abs\ (Rep\ a) = a) \wedge (\forall a.\ R\ (Rep\ a)\ (Rep\ a)) \wedge$
$(\forall r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s)$

lemma *Quotient3I*:
assumes $\bigwedge a.\ Abs\ (Rep\ a) = a$
and $\bigwedge a.\ R\ (Rep\ a)\ (Rep\ a)$
and $\bigwedge r\ s.\ R\ r\ s \longleftrightarrow R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s$
shows *Quotient3 R Abs Rep*
using *assms* **unfolding** *Quotient3-def* **by** *blast*

context
fixes *R Abs Rep*
assumes *a*: *Quotient3 R Abs Rep*
begin

lemma *Quotient3-abs-rep*:
$Abs\ (Rep\ a) = a$
using *a*
unfolding *Quotient3-def*
by *simp*

lemma *Quotient3-rep-reflp*:
$R\ (Rep\ a)\ (Rep\ a)$
using *a*
unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-rel*:
$R\ r\ r \wedge R\ s\ s \wedge Abs\ r = Abs\ s \longleftrightarrow R\ r\ s$ — orientation does not loop on rewriting
using *a*
unfolding *Quotient3-def*
by *blast*

lemma *Quotient3-refl1*:
$R\ r\ s \Longrightarrow R\ r\ r$
using *a* **unfolding** *Quotient3-def*
by *fast*

lemma *Quotient3-refl2*:

R r s $\Longrightarrow$ *R s s*
using *a* **unfolding** *Quotient3-def*
by *fast*

lemma *Quotient3-rel-rep*:
R (*Rep a*) (*Rep b*) $\longleftrightarrow$ *a* = *b*
using *a*
unfolding *Quotient3-def*
by *metis*

lemma *Quotient3-rep-abs*:
R r r $\Longrightarrow$ *R* (*Rep* (*Abs r*)) *r*
using *a* **unfolding** *Quotient3-def*
by *blast*

lemma *Quotient3-rel-abs*:
R r s $\Longrightarrow$ *Abs r* = *Abs s*
using *a* **unfolding** *Quotient3-def*
by *blast*

lemma *Quotient3-symp*:
symp R
using *a* **unfolding** *Quotient3-def* **using** *sympI* **by** *metis*

lemma *Quotient3-transp*:
transp R
using *a* **unfolding** *Quotient3-def* **using** *transpI* **by** (*metis* (*full-types*))

lemma *Quotient3-part-equivp*:
part-equivp R
by (*metis Quotient3-rep-reflp Quotient3-symp Quotient3-transp part-equivpI*)

lemma *abs-o-rep*:
Abs $\circ$ *Rep* = *id*
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*)

lemma *equals-rsp*:
assumes *b*: *R xa xb R ya yb*
shows *R xa ya* = *R xb yb*
using *b Quotient3-symp Quotient3-transp*
by (*blast elim*: *sympE transpE*)

lemma *rep-abs-rsp*:
assumes *b*: *R x1 x2*
shows *R x1* (*Rep* (*Abs x2*))
using *b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp*
by *metis*

lemma *rep-abs-rsp-left*:
assumes *b*: *R x1 x2*
shows *R* (*Rep* (*Abs x1*)) *x2*
using *b Quotient3-rel Quotient3-abs-rep Quotient3-rep-reflp*
by *metis*

end

lemma *identity-quotient3*:
Quotient3 (=) *id id*
unfolding *Quotient3-def id-def*
by *blast*

lemma *fun-quotient3*:
assumes *q1*: *Quotient3 R1 abs1 rep1*
and *q2*: *Quotient3 R2 abs2 rep2*
shows *Quotient3* (*R1* ===> *R2*) (*rep1* ---> *abs2*) (*abs1* ---> *rep2*)
proof −
have (*rep1* ---> *abs2*) ((*abs1* ---> *rep2*) *a*) = *a* **for** *a*
using *q1 q2* **by** (*simp add*: *Quotient3-def fun-eq-iff*)
moreover
have (*R1* ===> *R2*) ((*abs1* ---> *rep2*) *a*) ((*abs1* ---> *rep2*) *a*) **for** *a*
by (*rule rel-funI*)
(*use q1 q2 Quotient3-rel-abs* [*of R1 abs1 rep1*] *Quotient3-rel-rep* [*of R2 abs2 rep2*]
in ‹*simp* (*no-asm*) *add*: *Quotient3-def*, *simp*›)
moreover
have (*R1* ===> *R2*) *r s* = ((*R1* ===> *R2*) *r r* ∧ (*R1* ===> *R2*) *s s* ∧
(*rep1* ---> *abs2*) *r* = (*rep1* ---> *abs2*) *s*) **for** *r s*
proof −
have (*R1* ===> *R2*) *r s* ⟹ (*R1* ===> *R2*) *r r* **unfolding** *rel-fun-def*
using *Quotient3-part-equivp*[*OF q1*] *Quotient3-part-equivp*[*OF q2*]
by (*metis* (*full-types*) *part-equivp-def*)
moreover have (*R1* ===> *R2*) *r s* ⟹ (*R1* ===> *R2*) *s s* **unfolding** *rel-fun-def*
using *Quotient3-part-equivp*[*OF q1*] *Quotient3-part-equivp*[*OF q2*]
by (*metis* (*full-types*) *part-equivp-def*)
moreover have (*R1* ===> *R2*) *r s* ⟹ (*rep1* ---> *abs2*) *r* = (*rep1* ---> *abs2*) *s*
by (*auto simp add*: *rel-fun-def fun-eq-iff*)
(*use q1 q2* **in** ‹*unfold Quotient3-def*, *metis*›)
moreover have ((*R1* ===> *R2*) *r r* ∧ (*R1* ===> *R2*) *s s* ∧
(*rep1* ---> *abs2*) *r* = (*rep1* ---> *abs2*) *s*) ⟹ (*R1* ===> *R2*) *r s*
by (*auto simp add*: *rel-fun-def fun-eq-iff*)
(*use q1 q2* **in** ‹*unfold Quotient3-def*, *metis map-fun-apply*›)
ultimately show *?thesis* **by** *blast*
qed
ultimately show *?thesis* **by** (*intro Quotient3I*) (*assumption*+)
qed

lemma *lambda-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *Abs2*) (λ*x*. *Rep2* (*f* (*Abs1 x*))) = (λ*x*. *f x*)
unfolding *fun-eq-iff*
using *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*]
by *simp*

lemma *lambda-prs1*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *Abs2*) (λ*x*. (*Abs1* −−−> *Rep2*) *f x*) = (λ*x*. *f x*)
unfolding *fun-eq-iff*
using *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*]
by *simp*

In the following theorem R1 can be instantiated with anything, but we know some of the types of the Rep and Abs functions; so by solving Quotient assumptions we can get a unique R1 that will be provable; which is why we need to use *apply-rsp* and not the primed version

lemma *apply-rspQ3*:
fixes *f g*::'*a* ⇒ '*c*
assumes *q*: *Quotient3 R1 Abs1 Rep1*
and *a*: (*R1* ===> *R2*) *f g R1 x y*
shows *R2* (*f x*) (*g y*)
using *a* **by** (*auto elim*: *rel-funE*)

lemma *apply-rspQ3''*:
assumes *Quotient3 R Abs Rep*
and (*R* ===> *S*) *f f*
shows *S* (*f* (*Rep x*)) (*f* (*Rep x*))
proof −
from *assms*(*1*) **have** *R* (*Rep x*) (*Rep x*) **by** (*rule Quotient3-rep-reflp*)
then show *?thesis* **using** *assms*(*2*) **by** (*auto intro*: *apply-rsp'*)
qed

44.2 lemmas for regularisation of ball and bex

lemma *ball-reg-eqv*:
fixes *P* :: '*a* ⇒ *bool*
assumes *a*: *equivp R*
shows *Ball* (*Respects R*) *P* = (*All P*)
using *a*
unfolding *equivp-def*
by (*auto simp add*: *in-respects*)

lemma *bex-reg-eqv*:
fixes *P* :: '*a* ⇒ *bool*

assumes *a*: *equivp R*
shows *Bex* (*Respects R*) *P* = (*Ex P*)
using *a*
unfolding *equivp-def*
by (*auto simp add*: *in-respects*)

lemma *ball-reg-right*:
assumes *a*: $\bigwedge x.\ x \in R \Longrightarrow P\ x \longrightarrow Q\ x$
shows *All P* $\longrightarrow$ *Ball R Q*
using *a* **by** *fast*

lemma *bex-reg-left*:
assumes *a*: $\bigwedge x.\ x \in R \Longrightarrow Q\ x \longrightarrow P\ x$
shows *Bex R Q* $\longrightarrow$ *Ex P*
using *a* **by** *fast*

lemma *ball-reg-left*:
assumes *a*: *equivp R*
shows $(\bigwedge x.\ (Q\ x \longrightarrow P\ x)) \Longrightarrow$ *Ball* (*Respects R*) *Q* $\longrightarrow$ *All P*
using *a* **by** (*metis equivp-reflp in-respects*)

lemma *bex-reg-right*:
assumes *a*: *equivp R*
shows $(\bigwedge x.\ (Q\ x \longrightarrow P\ x)) \Longrightarrow$ *Ex Q* $\longrightarrow$ *Bex* (*Respects R*) *P*
using *a* **by** (*metis equivp-reflp in-respects*)

lemma *ball-reg-eqv-range*:
fixes *P*::'*a* $\Rightarrow$ *bool*
and *x*::'*a*
assumes *a*: *equivp R2*
shows (*Ball* (*Respects* (*R1* ===> *R2*)) (λ*f*. *P* (*f x*)) = *All* (λ*f*. *P* (*f x*)))
proof (*intro allI iffI*)
fix *f*
assume ∀*f* ∈ *Respects* (*R1* ===> *R2*). *P* (*f x*)
moreover have (λ*y*. *f x*) ∈ *Respects* (*R1* ===> *R2*)
using *a equivp-reflp-symp-transp*[*of R2*]
by(*auto simp add*: *in-respects rel-fun-def elim*: *equivpE reflpE*)
ultimately show *P* (*f x*)
by *auto*
qed *auto*

lemma *bex-reg-eqv-range*:
assumes *a*: *equivp R2*
shows (*Bex* (*Respects* (*R1* ===> *R2*)) (λ*f*. *P* (*f x*)) = *Ex* (λ*f*. *P* (*f x*)))
proof −
have (λ*y*. *f x*) ∈ *Respects* (*R1* ===> *R2*) **for** *f*
using *a equivp-reflp-symp-transp*[*of R2*]
by (*auto simp add*: *Respects-def in-respects rel-fun-def elim*: *equivpE reflpE*)
then show *?thesis*

by *auto*
qed

lemma *all-reg*:
assumes *a*: $\forall x :: 'a.\ (P\ x \longrightarrow Q\ x)$
and *b*: *All P*
shows *All Q*
using *a b* **by** *fast*

lemma *ex-reg*:
assumes *a*: $\forall x :: 'a.\ (P\ x \longrightarrow Q\ x)$
and *b*: *Ex P*
shows *Ex Q*
using *a b* **by** *fast*

lemma *ball-reg*:
assumes *a*: $\forall x :: 'a.\ (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and *b*: *Ball R P*
shows *Ball R Q*
using *a b* **by** *fast*

lemma *bex-reg*:
assumes *a*: $\forall x :: 'a.\ (x \in R \longrightarrow P\ x \longrightarrow Q\ x)$
and *b*: *Bex R P*
shows *Bex R Q*
using *a b* **by** *fast*

lemma *ball-all-comm*:
assumes $\bigwedge y.\ (\forall x{\in}P.\ A\ x\ y) \longrightarrow (\forall x.\ B\ x\ y)$
shows $(\forall x{\in}P.\ \forall y.\ A\ x\ y) \longrightarrow (\forall x.\ \forall y.\ B\ x\ y)$
using *assms* **by** *auto*

lemma *bex-ex-comm*:
assumes $(\exists y.\ \exists x.\ A\ x\ y) \longrightarrow (\exists y.\ \exists x{\in}P.\ B\ x\ y)$
shows $(\exists x.\ \exists y.\ A\ x\ y) \longrightarrow (\exists x{\in}P.\ \exists y.\ B\ x\ y)$
using *assms* **by** *auto*

44.3 Bounded abstraction

definition
Babs :: $'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow 'a \Rightarrow 'b$
where
$x \in p \Longrightarrow Babs\ p\ m\ x = m\ x$

lemma *babs-rsp*:
assumes *q*: *Quotient3 R1 Abs1 Rep1*
and *a*: $(R1 \mathrel{===>} R2)\ f\ g$

```
    shows (R1 ===> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)
proof
  fix x y
  assume R1 x y
  then have x ∈ Respects R1 ∧ y ∈ Respects R1
    unfolding in-respects rel-fun-def using Quotient3-rel[OF q]by metis
  then show R2 (Babs (Respects R1) f x) (Babs (Respects R1) g y)
  using ‹R1 x y› a by (simp add: Babs-def rel-fun-def)
qed

lemma babs-prs:
  assumes q1: Quotient3 R1 Abs1 Rep1
  and     q2: Quotient3 R2 Abs2 Rep2
shows ((Rep1 ---> Abs2) (Babs (Respects R1) ((Abs1 ---> Rep2) f))) = f
proof -
  have Abs2 (Babs (Respects R1) ((Abs1 ---> Rep2) f) (Rep1 x)) = f x for x
  proof -
    have Rep1 x ∈ Respects R1
      by (simp add: in-respects Quotient3-rel-rep[OF q1])
    then show ?thesis
      by (simp add: Babs-def Quotient3-abs-rep[OF q1] Quotient3-abs-rep[OF q2])
  qed
  then show ?thesis
    by force
qed

lemma babs-simp:
  assumes q: Quotient3 R1 Abs Rep
  shows ((R1 ===> R2) (Babs (Respects R1) f) (Babs (Respects R1) g)) = ((R1
===> R2) f g)
        (is ?lhs = ?rhs)
proof
  assume ?lhs
  then show ?rhs
    unfolding rel-fun-def by (metis Babs-def in-respects  Quotient3-rel[OF q])
qed (simp add: babs-rsp[OF q])
```

If a user proves that a particular functional relation is an equivalence, this may be useful in regularising

```
lemma babs-reg-eqv:
  shows equivp R ⟹ Babs (Respects R) P = P
  by (simp add: fun-eq-iff Babs-def in-respects equivp-reflp)



lemma ball-rsp:
  assumes a: (R ===> (=)) f g
  shows Ball (Respects R) f = Ball (Respects R) g
  using a by (auto simp add: Ball-def in-respects elim: rel-funE)
```

lemma *bex-rsp*:
assumes *a*: $(R ===> (=))\ f\ g$
shows $(Bex\ (Respects\ R)\ f = Bex\ (Respects\ R)\ g)$
using *a* **by** (*auto simp add*: *Bex-def in-respects elim*: *rel-funE*)

lemma *bex1-rsp*:
assumes *a*: $(R ===> (=))\ f\ g$
shows $Ex1\ (\lambda x.\ x \in Respects\ R \wedge f\ x) = Ex1\ (\lambda x.\ x \in Respects\ R \wedge g\ x)$
using *a* **by** (*auto elim*: *rel-funE simp add*: *Ex1-def in-respects*)

Two lemmas needed for cleaning of quantifiers

lemma *all-prs*:
assumes *a*: *Quotient3 R absf repf*
shows $Ball\ (Respects\ R)\ ((absf\ ---> id)\ f) = All\ f$
using *a* **unfolding** *Quotient3-def Ball-def in-respects id-apply comp-def map-fun-def*
by *metis*

lemma *ex-prs*:
assumes *a*: *Quotient3 R absf repf*
shows $Bex\ (Respects\ R)\ ((absf\ ---> id)\ f) = Ex\ f$
using *a* **unfolding** *Quotient3-def Bex-def in-respects id-apply comp-def map-fun-def*
by *metis*

44.4 *Bex1-rel* quantifier

definition
$Bex1\text{-}rel :: ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow bool) \Rightarrow bool$
where
$Bex1\text{-}rel\ R\ P \longleftrightarrow (\exists x \in Respects\ R.\ P\ x) \wedge (\forall x \in Respects\ R.\ \forall y \in Respects\ R.\ ((P\ x \wedge P\ y) \longrightarrow (R\ x\ y)))$

lemma *bex1-rel-aux*:
$[\![\forall xa\ ya.\ R\ xa\ ya \longrightarrow x\ xa = y\ ya;\ Bex1\text{-}rel\ R\ x]\!] \Longrightarrow Bex1\text{-}rel\ R\ y$
unfolding *Bex1-rel-def*
by (*metis in-respects*)

lemma *bex1-rel-aux2*:
$[\![\forall xa\ ya.\ R\ xa\ ya \longrightarrow x\ xa = y\ ya;\ Bex1\text{-}rel\ R\ y]\!] \Longrightarrow Bex1\text{-}rel\ R\ x$
unfolding *Bex1-rel-def*
by (*metis in-respects*)

lemma *bex1-rel-rsp*:
assumes *a*: *Quotient3 R absf repf*
shows $((R ===> (=)) ===> (=))\ (Bex1\text{-}rel\ R)\ (Bex1\text{-}rel\ R)$
unfolding *rel-fun-def* **by** (*metis bex1-rel-aux bex1-rel-aux2*)

lemma *ex1-prs*:
assumes *Quotient3 R absf repf*

shows ((*absf* −−−> *id*) −−−> *id*) (*Bex1-rel R*) *f* = *Ex1 f*
(**is** *?lhs* = *?rhs*)
using *assms*
by (*auto simp add*: *Bex1-rel-def Respects-def*) (*metis* (*full-types*) *Quotient3-def*)+

lemma *bex1-bexeq-reg*:
shows (∃!*x*∈*Respects R*. *P x*) ⟶ (*Bex1-rel R* (λ*x*. *P x*))
by (*auto simp add*: *Ex1-def Bex1-rel-def Bex-def Ball-def in-respects*)

lemma *bex1-bexeq-reg-eqv*:
assumes *a*: *equivp R*
shows (∃!*x*. *P x*) ⟶ *Bex1-rel R P*
using *equivp-reflp*[*OF a*]
by (*metis* (*full-types*) *Bex1-rel-def in-respects*)

44.5 Various respects and preserve lemmas

lemma *quot-rel-rsp*:
assumes *a*: *Quotient3 R Abs Rep*
shows (*R* ===> *R* ===> (=)) *R R*
by (*rule rel-funI*)+ (*meson assms equals-rsp*)

lemma *o-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
and *q3*: *Quotient3 R3 Abs3 Rep3*
shows ((*Abs2* −−−> *Rep3*) −−−> (*Abs1* −−−> *Rep2*) −−−> (*Rep1* −−−> *Abs3*)) (∘) = (∘)
and (*id* −−−> (*Abs1* −−−> *id*) −−−> *Rep1* −−−> *id*) (∘) = (∘)
using *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*] *Quotient3-abs-rep*[*OF q3*]
by (*simp-all add*: *fun-eq-iff*)

lemma *o-rsp*:
((*R2* ===> *R3*) ===> (*R1* ===> *R2*) ===> (*R1* ===> *R3*)) (∘) (∘)
((=) ===> (*R1* ===> (=)) ===> *R1* ===> (=)) (∘) (∘)
by (*force elim*: *rel-funE*)+

lemma *cond-prs*:
assumes *a*: *Quotient3 R absf repf*
shows *absf* (*if a then repf b else repf c*) = (*if a then b else c*)
using *a* **unfolding** *Quotient3-def* **by** *auto*

lemma *if-prs*:
assumes *q*: *Quotient3 R Abs Rep*
shows (*id* −−−> *Rep* −−−> *Rep* −−−> *Abs*) *If* = *If*
using *Quotient3-abs-rep*[*OF q*]
by (*auto simp add*: *fun-eq-iff*)

lemma *if-rsp*:
assumes *q*: *Quotient3 R Abs Rep*
shows ((=) ===> *R* ===> *R* ===> *R*) *If If*
by *force*

lemma *let-prs*:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* ---> (*Abs2* ---> *Rep1*) ---> *Abs1*) *Let* = *Let*
using *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*]
by (*auto simp add*: *fun-eq-iff*)

lemma *let-rsp*:
shows (*R1* ===> (*R1* ===> *R2*) ===> *R2*) *Let Let*
by (*force elim*: *rel-funE*)

lemma *id-rsp*:
shows (*R* ===> *R*) *id id*
by *auto*

lemma *id-prs*:
assumes *a*: *Quotient3 R Abs Rep*
shows (*Rep* ---> *Abs*) *id* = *id*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep* [*OF a*])

end

locale *quot-type* =
fixes *R* :: *'a* ⇒ *'a* ⇒ *bool*
and *Abs* :: *'a set* ⇒ *'b*
and *Rep* :: *'b* ⇒ *'a set*
assumes *equivp*: *part-equivp R*
and *rep-prop*: ⋀*y*. ∃ *x*. *R x x* ∧ *Rep y* = *Collect* (*R x*)
and *rep-inverse*: ⋀*x*. *Abs* (*Rep x*) = *x*
and *abs-inverse*: ⋀*c*. (∃ *x*. ((*R x x*) ∧ (*c* = *Collect* (*R x*)))) ⟹ (*Rep* (*Abs c*)) = *c*
and *rep-inject*: ⋀*x y*. (*Rep x* = *Rep y*) = (*x* = *y*)
begin

definition
abs :: *'a* ⇒ *'b*
where
abs x = *Abs* (*Collect* (*R x*))

definition
rep :: *'b* ⇒ *'a*
where
rep a = (*SOME x*. *x* ∈ *Rep a*)

lemma *some-collect*:
assumes *R r r*
shows *R* (*SOME x*. *x* ∈ *Collect* (*R r*)) = *R r*
by *simp* (*metis assms exE-some equivp*[*simplified part-equivp-def*])

lemma *Quotient*: *Quotient3 R abs rep*
unfolding *Quotient3-def abs-def rep-def*
proof (*intro conjI allI*)
fix *a r s*
show *x*: *R* (*SOME x*. *x* ∈ *Rep a*) (*SOME x*. *x* ∈ *Rep a*) **proof** −
obtain *x* **where** *r*: *R x x* **and** *rep*: *Rep a* = *Collect* (*R x*) **using** *rep-prop*[*of a*] **by** *auto*
have *R* (*SOME x*. *x* ∈ *Rep a*) *x* **using** *r rep some-collect* **by** *metis*
then have *R x* (*SOME x*. *x* ∈ *Rep a*) **using** *part-equivp-symp*[*OF equivp*] **by** *fast*
then show *R* (*SOME x*. *x* ∈ *Rep a*) (*SOME x*. *x* ∈ *Rep a*)
using *part-equivp-transp*[*OF equivp*] **by** (*metis* ‹*R* (*SOME x*. *x* ∈ *Rep a*) *x*›)
qed
have *Collect* (*R* (*SOME x*. *x* ∈ *Rep a*)) = (*Rep a*) **by** (*metis some-collect rep-prop*)
then show *Abs* (*Collect* (*R* (*SOME x*. *x* ∈ *Rep a*))) = *a* **using** *rep-inverse* **by** *auto*
have *R r r* ⟹ *R s s* ⟹ *Abs* (*Collect* (*R r*)) = *Abs* (*Collect* (*R s*)) ⟷ *R r* = *R s*
proof −
assume *R r r* **and** *R s s*
then have *Abs* (*Collect* (*R r*)) = *Abs* (*Collect* (*R s*)) ⟷ *Collect* (*R r*) = *Collect* (*R s*)
by (*metis abs-inverse*)
also have *Collect* (*R r*) = *Collect* (*R s*) ⟷ (λ*A x*. *x* ∈ *A*) (*Collect* (*R r*)) = (λ*A x*. *x* ∈ *A*) (*Collect* (*R s*))
by (*rule iffI*) *simp-all*
finally show *Abs* (*Collect* (*R r*)) = *Abs* (*Collect* (*R s*)) ⟷ *R r* = *R s* **by** *simp*
qed
then show *R r s* ⟷ *R r r* ∧ *R s s* ∧ (*Abs* (*Collect* (*R r*)) = *Abs* (*Collect* (*R s*)))
using *equivp*[*simplified part-equivp-def*] **by** *metis*
qed

end

44.6 Quotient composition

lemma *OOO-quotient3*:
fixes *R1* :: *'a* ⇒ *'a* ⇒ *bool*
fixes *Abs1* :: *'a* ⇒ *'b* **and** *Rep1* :: *'b* ⇒ *'a*
fixes *Abs2* :: *'b* ⇒ *'c* **and** *Rep2* :: *'c* ⇒ *'b*
fixes *R2'* :: *'a* ⇒ *'a* ⇒ *bool*

fixes *R2* :: *'b* ⇒ *'b* ⇒ *bool*
assumes *R1*: *Quotient3 R1 Abs1 Rep1*
assumes *R2*: *Quotient3 R2 Abs2 Rep2*
assumes *Abs1*: ⋀*x y*. *R2' x y* ⟹ *R1 x x* ⟹ *R1 y y* ⟹ *R2* (*Abs1 x*) (*Abs1 y*)
assumes *Rep1*: ⋀*x y*. *R2 x y* ⟹ *R2'* (*Rep1 x*) (*Rep1 y*)
shows *Quotient3* (*R1 OO R2' OO R1*) (*Abs2* ∘ *Abs1*) (*Rep1* ∘ *Rep2*)
proof −
have ∗: (*R1 OOO R2'*) *r r* ∧ (*R1 OOO R2'*) *s s* ∧ (*Abs2* ∘ *Abs1*) *r* = (*Abs2* ∘ *Abs1*) *s*
⟷ (*R1 OOO R2'*) *r s* **for** *r s*
proof (*intro iffI conjI*; *clarify*)
show (*R1 OOO R2'*) *r s*
if *r*: *R1 r a R2' a b R1 b r* **and** *eq*: (*Abs2* ∘ *Abs1*) *r* = (*Abs2* ∘ *Abs1*) *s*
and *s*: *R1 s c R2' c d R1 d s* **for** *a b c d*
proof −
have *R1 r* (*Rep1* (*Abs1 r*))
using *r Quotient3-refl1 R1 rep-abs-rsp* **by** *fastforce*
moreover have *R2'* (*Rep1* (*Abs1 r*)) (*Rep1* (*Abs1 s*))
using *that*
by *simp* (*metis* (*full-types*) *Rep1 Abs1 Quotient3-rel R2 Quotient3-refl1* [*OF R1*]
Quotient3-refl2 [*OF R1*] *Quotient3-rel-abs* [*OF R1*])
moreover have *R1* (*Rep1* (*Abs1 s*)) *s*
by (*metis s Quotient3-rel R1 rep-abs-rsp-left*)
ultimately show *?thesis*
by (*metis relcomppI*)
qed
next
fix *x y*
assume *xy*: *R1 r x R2' x y R1 y s*
then have *R2* (*Abs1 x*) (*Abs1 y*)
by (*iprover dest*: *Abs1 elim*: *Quotient3-refl1* [*OF R1*] *Quotient3-refl2* [*OF R1*])
then have *R2'* (*Rep1* (*Abs1 x*)) (*Rep1* (*Abs1 x*)) *R2'* (*Rep1* (*Abs1 y*)) (*Rep1* (*Abs1 y*))
by (*simp-all add*: *Quotient3-refl1* [*OF R2*] *Quotient3-refl2* [*OF R2*] *Rep1*)
with ‹*R1 r x*› ‹*R1 y s*› **show** (*R1 OOO R2'*) *r r* (*R1 OOO R2'*) *s s*
by (*metis* (*full-types*) *Quotient3-def R1 relcompp.relcompI*)+
show (*Abs2* ∘ *Abs1*) *r* = (*Abs2* ∘ *Abs1*) *s*
using *xy* **by** *simp* (*metis* (*full-types*) *Abs1 Quotient3-rel R1 R2*)
qed
show *?thesis*
apply (*rule Quotient3I*)
using ∗ **apply** (*simp-all add*: *o-def Quotient3-abs-rep* [*OF R2*] *Quotient3-abs-rep* [*OF R1*])
apply (*metis Quotient3-rep-reflp R1 R2 Rep1 relcompp.relcompI*)
done
qed

lemma *OOO-eq-quotient3*:
fixes *R1* :: *'a* ⇒ *'a* ⇒ *bool*
fixes *Abs1* :: *'a* ⇒ *'b* **and** *Rep1* :: *'b* ⇒ *'a*
fixes *Abs2* :: *'b* ⇒ *'c* **and** *Rep2* :: *'c* ⇒ *'b*
assumes *R1*: *Quotient3 R1 Abs1 Rep1*
assumes *R2*: *Quotient3* (=) *Abs2 Rep2*
shows *Quotient3* (*R1 OOO* (=)) (*Abs2* ∘ *Abs1*) (*Rep1* ∘ *Rep2*)
using *assms*
by (*rule OOO-quotient3*) *auto*

44.7 Quotient3 to Quotient

lemma *Quotient3-to-Quotient*:
assumes *Quotient3 R Abs Rep*
and *T* ≡ λ*x y*. *R x x* ∧ *Abs x* = *y*
shows *Quotient R Abs Rep T*
using *assms* **unfolding** *Quotient3-def* **by** (*intro QuotientI*) *blast*+

lemma *Quotient3-to-Quotient-equivp*:
assumes *q*: *Quotient3 R Abs Rep*
and *T-def*: *T* ≡ λ*x y*. *Abs x* = *y*
and *eR*: *equivp R*
shows *Quotient R Abs Rep T*
proof (*intro QuotientI*)
show *Abs* (*Rep a*) = *a* **for** *a*
using *q* **by**(*rule Quotient3-abs-rep*)
show *R* (*Rep a*) (*Rep a*) **for** *a*
using *q* **by**(*rule Quotient3-rep-reflp*)
show *R r s* = (*R r r* ∧ *R s s* ∧ *Abs r* = *Abs s*) **for** *r s*
using *q* **by**(*rule Quotient3-rel*[*symmetric*])
show *T* = (λ*x y*. *R x x* ∧ *Abs x* = *y*)
using *T-def equivp-reflp*[*OF eR*] **by** *simp*
qed

44.8 ML setup

Auxiliary data for the quotient package

named-theorems *quot-equiv equivalence relation theorems*
and *quot-respect respectfulness theorems*
and *quot-preserve preservation theorems*
and *id-simps identity simp rules for maps*
and *quot-thm quotient theorems*
ML-file ‹*Tools/Quotient/quotient-info.ML*›

declare [[*mapQ3 fun* = (*rel-fun*, *fun-quotient3*)]]

lemmas [*quot-thm*] = *fun-quotient3*
lemmas [*quot-respect*] = *quot-rel-rsp if-rsp o-rsp let-rsp id-rsp*
lemmas [*quot-preserve*] = *if-prs o-prs let-prs id-prs*

lemmas [*quot-equiv*] = *identity-equivp*

Lemmas about simplifying id's.

lemmas [*id-simps*] =
id-def[*symmetric*]
map-fun-id
id-apply
id-o
o-id
eq-comp-r
vimage-id

Translation functions for the lifting process.

ML-file ‹*Tools/Quotient/quotient-term.ML*›

Definitions of the quotient types.

ML-file ‹*Tools/Quotient/quotient-type.ML*›

Definitions for quotient constants.

ML-file ‹*Tools/Quotient/quotient-def.ML*›

An auxiliary constant for recording some information about the lifted theorem in a tactic.

definition
Quot-True :: *'a* ⇒ *bool*
where
Quot-True x ⟷ *True*

lemma
shows *QT-all*: *Quot-True* (*All P*) ⟹ *Quot-True P*
and *QT-ex*: *Quot-True* (*Ex P*) ⟹ *Quot-True P*
and *QT-ex1*: *Quot-True* (*Ex1 P*) ⟹ *Quot-True P*
and *QT-lam*: *Quot-True* (λ*x*. *P x*) ⟹ (⋀*x*. *Quot-True* (*P x*))
and *QT-ext*: (⋀*x*. *Quot-True* (*a x*) ⟹ *f x* = *g x*) ⟹ (*Quot-True a* ⟹ *f* = *g*)
by (*simp-all add*: *Quot-True-def ext*)

lemma *QT-imp*: *Quot-True a* ≡ *Quot-True b*
by (*simp add*: *Quot-True-def*)

context includes *lifting-syntax*
begin

Tactics for proving the lifted theorems

ML-file ‹*Tools/Quotient/quotient-tacs.ML*›

end

44.9 Methods / Interface

method-setup *lifting* =
‹*Attrib.thms* >> (*fn thms* => *fn ctxt* =>
SIMPLE-METHOD' (*Quotient-Tacs.lift-tac ctxt* [] *thms*))›
‹*lift theorems to quotient types*›

method-setup *lifting-setup* =
‹*Attrib.thm* >> (*fn thm* => *fn ctxt* =>
SIMPLE-METHOD' (*Quotient-Tacs.lift-procedure-tac ctxt* [] *thm*))›
‹*set up the three goals for the quotient lifting procedure*›

method-setup *descending* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.descend-tac ctxt* []))›
‹*decend theorems to the raw level*›

method-setup *descending-setup* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.descend-procedure-tac ctxt* []))›
‹*set up the three goals for the decending theorems*›

method-setup *partiality-descending* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.partiality-descend-tac ctxt* []))›
‹*decend theorems to the raw level*›

method-setup *partiality-descending-setup* =
‹*Scan.succeed* (*fn ctxt* =>
SIMPLE-METHOD' (*Quotient-Tacs.partiality-descend-procedure-tac ctxt* []))›
‹*set up the three goals for the decending theorems*›

method-setup *regularize* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.regularize-tac ctxt*))›
‹*prove the regularization goals from the quotient lifting procedure*›

method-setup *injection* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.all-injection-tac ctxt*))›
‹*prove the rep/abs injection goals from the quotient lifting procedure*›

method-setup *cleaning* =
‹*Scan.succeed* (*fn ctxt* => *SIMPLE-METHOD'* (*Quotient-Tacs.clean-tac ctxt*))›
‹*prove the cleaning goals from the quotient lifting procedure*›

attribute-setup *quot-lifted* =
‹*Scan.succeed Quotient-Tacs.lifted-attrib*›
‹*lift theorems to quotient types*›

no-notation *rel-conj* (**infixr** ‹*OOO*› *75*)

45 Lifting of BNFs

lemma *sum-insert-Inl-unit*: $x \in A \Longrightarrow (\bigwedge y.\ x = Inr\ y \Longrightarrow Inr\ y \in B) \Longrightarrow x \in insert\ (Inl\ ())\ B$
by (*cases x*) (*simp-all*)

lemma *lift-sum-unit-vimage-commute*:
$insert\ (Inl\ ())\ (Inr\ `\ f\ -`\ A) = map\text{-}sum\ id\ f\ -`\ insert\ (Inl\ ())\ (Inr\ `\ A)$
by (*auto simp*: *map-sum-def split*: *sum.splits*)

lemma *insert-Inl-int-map-sum-unit*: $insert\ (Inl\ ())\ A \cap range\ (map\text{-}sum\ id\ f) \neq \{\}$
by (*auto simp*: *map-sum-def split*: *sum.splits*)

lemma *image-map-sum-unit-subset*:
$A \subseteq insert\ (Inl\ ())\ (Inr\ `\ B) \Longrightarrow map\text{-}sum\ id\ f\ `\ A \subseteq insert\ (Inl\ ())\ (Inr\ `\ f\ `\ B)$
by *auto*

lemma *subset-lift-sum-unitD*: $A \subseteq insert\ (Inl\ ())\ (Inr\ `\ B) \Longrightarrow Inr\ x \in A \Longrightarrow x \in B$
unfolding *insert-def* **by** *auto*

lemma *UNIV-sum-unit-conv*: $insert\ (Inl\ ())\ (range\ Inr) = UNIV$
unfolding *UNIV-sum UNIV-unit image-insert image-empty Un-insert-left sup-bot.left-neutral*..

lemma *subset-vimage-image-subset*: $A \subseteq f\ -`\ B \Longrightarrow f\ `\ A \subseteq B$
by *auto*

lemma *relcompp-mem-Grp-neq-bot*:
$A \cap range\ f \neq \{\} \Longrightarrow (\lambda x\ y.\ x \in A \wedge y \in A)\ OO\ (Grp\ UNIV\ f)^{-1-1} \neq bot$
unfolding *Grp-def relcompp-apply fun-eq-iff* **by** *blast*

lemma *comp-projr-Inr*: $projr \circ Inr = id$
by *auto*

lemma *in-rel-sum-in-image-projr*:
$B \subseteq \{(x,y).\ rel\text{-}sum\ ((=) :: unit \Rightarrow unit \Rightarrow bool)\ A\ x\ y\} \Longrightarrow$
$Inr\ `\ C = fst\ `\ B \Longrightarrow snd\ `\ B = Inr\ `\ D \Longrightarrow map\text{-}prod\ projr\ projr\ `\ B \subseteq \{(x,y).\ A\ x\ y\}$
by (*force simp*: *projr-def image-iff dest*!: *spec*[*of* - *Inl* ()] *split*: *sum.splits*)

lemma *subset-rel-sumI*: $B \subseteq \{(x,y).\ A\ x\ y\} \Longrightarrow rel\text{-}sum\ ((=) :: unit => unit => bool)\ A$
$(if\ x \in B\ then\ Inr\ (fst\ x)\ else\ Inl\ ())$
$(if\ x \in B\ then\ Inr\ (snd\ x)\ else\ Inl\ ())$
by *auto*

lemma *relcompp-eq-Grp-neq-bot*: $(=)\ OO\ (Grp\ UNIV\ f)^{-1-1} \neq bot$
unfolding *Grp-def relcompp-apply fun-eq-iff* **by** *blast*

lemma *rel-fun-rel-OO1*: $(rel\text{-}fun\ Q\ (rel\text{-}fun\ R\ (=)))\ A\ B \Longrightarrow conversep\ Q\ OO\ A\ OO\ R \leq B$
by (*auto simp*: *rel-fun-def*)

lemma *rel-fun-rel-OO2*: $(rel\text{-}fun\ Q\ (rel\text{-}fun\ R\ (=)))\ A\ B \Longrightarrow Q\ OO\ B\ OO\ conversep\ R \leq A$
by (*auto simp*: *rel-fun-def*)

lemma *rel-sum-eq2-nonempty*: $rel\text{-}sum\ (=)\ A\ OO\ rel\text{-}sum\ (=)\ B \neq bot$
by (*auto simp*: *fun-eq-iff relcompp-apply intro*!: *exI*[*of - Inl -*])

lemma *rel-sum-eq3-nonempty*: $rel\text{-}sum\ (=)\ A\ OO\ (rel\text{-}sum\ (=)\ B\ OO\ rel\text{-}sum\ (=)\ C) \neq bot$
by (*auto simp*: *fun-eq-iff relcompp-apply intro*!: *exI*[*of - Inl -*])

lemma *hypsubst*: $A = B \Longrightarrow x \in B \Longrightarrow (x \in A \Longrightarrow P) \Longrightarrow P$ **by** *simp*

lemma *Quotient-crel-quotient*: $Quotient\ R\ Abs\ Rep\ T \Longrightarrow equivp\ R \Longrightarrow T \equiv (\lambda x\ y.\ Abs\ x = y)$
by (*drule Quotient-cr-rel*) (*auto simp*: *fun-eq-iff equivp-reflp intro*!: *eq-reflection*)

lemma *Quotient-crel-typedef*: $Quotient\ (eq\text{-}onp\ P)\ Abs\ Rep\ T \Longrightarrow T \equiv (\lambda x\ y.\ x = Rep\ y)$
unfolding *Quotient-def*
by (*auto 0 4 simp*: *fun-eq-iff eq-onp-def intro*: *sym intro*!: *eq-reflection*)

lemma *Quotient-crel-typecopy*: $Quotient\ (=)\ Abs\ Rep\ T \Longrightarrow T \equiv (\lambda x\ y.\ x = Rep\ y)$
by (*subst* (*asm*) *eq-onp-True*[*symmetric*]) (*rule Quotient-crel-typedef*)

lemma *equivp-add-relconj*:
assumes *equiv*: *equivp R equivp R'* **and** *le*: $S\ OO\ T\ OO\ U \leq R\ OO\ STU\ OO\ R'$
shows $R\ OO\ S\ OO\ T\ OO\ U\ OO\ R' \leq R\ OO\ STU\ OO\ R'$
proof −
have *trans*: $R\ OO\ R \leq R\ R'\ OO\ R' \leq R'$
using *equiv* **unfolding** *equivp-reflp-symp-transp transp-relcompp* **by** *blast*+
have $R\ OO\ S\ OO\ T\ OO\ U\ OO\ R' = R\ OO\ (S\ OO\ T\ OO\ U)\ OO\ R'$
unfolding *relcompp-assoc* **..**
also have $\ldots \leq R\ OO\ (R\ OO\ STU\ OO\ R')\ OO\ R'$
by (*intro le relcompp-mono order-refl*)
also have $\ldots \leq (R\ OO\ R)\ OO\ STU\ OO\ (R'\ OO\ R')$
unfolding *relcompp-assoc* **..**
also have $\ldots \leq R\ OO\ STU\ OO\ R'$
by (*intro trans relcompp-mono order-refl*)
finally show *?thesis* **.**
qed

lemma *Grp-conversep-eq-onp*: $((BNF\text{-}Def.Grp\ UNIV\ f)^{-1-1}\ OO\ BNF\text{-}Def.Grp\ UNIV\ f) = eq\text{-}onp\ (\lambda x.\ x \in range\ f)$
by (*auto simp*: *fun-eq-iff Grp-def eq-onp-def image-iff*)

lemma *Grp-conversep-nonempty*: $(BNF\text{-}Def.Grp\ UNIV\ f)^{-1-1}\ OO\ BNF\text{-}Def.Grp\ UNIV\ f \neq bot$
by (*auto simp*: *fun-eq-iff Grp-def*)

lemma *relcomppI2*: $r\ a\ b \Longrightarrow s\ b\ c \Longrightarrow t\ c\ d \Longrightarrow (r\ OO\ s\ OO\ t)\ a\ d$
by (*auto*)

lemma *rel-conj-eq-onp*: $equivp\ R \Longrightarrow rel\text{-}conj\ R\ (eq\text{-}onp\ P) \leq R$
by (*auto simp*: *eq-onp-def transp-def equivp-def*)

lemma *Quotient-Quotient3*: $Quotient\ R\ Abs\ Rep\ T \Longrightarrow Quotient3\ R\ Abs\ Rep$
unfolding *Quotient-def Quotient3-def* **by** *blast*

lemma *Quotient-reflp-imp-equivp*: $Quotient\ R\ Abs\ Rep\ T \Longrightarrow reflp\ R \Longrightarrow equivp\ R$
using *Quotient-symp Quotient-transp equivpI* **by** *blast*

lemma *Quotient-eq-onp-typedef*:
$Quotient\ (eq\text{-}onp\ P)\ Abs\ Rep\ cr \Longrightarrow type\text{-}definition\ Rep\ Abs\ \{x.\ P\ x\}$
unfolding *Quotient-def eq-onp-def*
by *unfold-locales auto*

lemma *Quotient-eq-onp-type-copy*:
$Quotient\ (=)\ Abs\ Rep\ cr \Longrightarrow type\text{-}definition\ Rep\ Abs\ UNIV$
unfolding *Quotient-def eq-onp-def*
by *unfold-locales auto*

ML-file ‹*Tools/BNF/bnf-lift.ML*›

hide-fact
sum-insert-Inl-unit lift-sum-unit-vimage-commute insert-Inl-int-map-sum-unit
image-map-sum-unit-subset subset-lift-sum-unitD UNIV-sum-unit-conv subset-vimage-image-subset
relcompp-mem-Grp-neq-bot comp-projr-Inr in-rel-sum-in-image-projr subset-rel-sumI
relcompp-eq-Grp-neq-bot rel-fun-rel-OO1 rel-fun-rel-OO2 rel-sum-eq2-nonempty rel-sum-eq3-nonempty
hypsubst equivp-add-relconj Grp-conversep-eq-onp Grp-conversep-nonempty rel-comppI2 rel-conj-eq-onp
Quotient-reflp-imp-equivp Quotient-Quotient3

end

46 Binary Numerals

theory *Num*

imports *BNF-Least-Fixpoint Transfer*
begin

46.1 The *num* type

datatype *num* = *One* | *Bit0 num* | *Bit1 num*

Increment function for type *num*

primrec *inc* :: ‹*num* ⇒ *num*›
  where
    ‹*inc One* = *Bit0 One*›
  | ‹*inc* (*Bit0 x*) = *Bit1 x*›
  | ‹*inc* (*Bit1 x*) = *Bit0* (*inc x*)›

Converting between type *num* and type *nat*

primrec *nat-of-num* :: ‹*num* ⇒ *nat*›
  where
    ‹*nat-of-num One* = *Suc 0*›
  | ‹*nat-of-num* (*Bit0 x*) = *nat-of-num x* + *nat-of-num x*›
  | ‹*nat-of-num* (*Bit1 x*) = *Suc* (*nat-of-num x* + *nat-of-num x*)›

primrec *num-of-nat* :: ‹*nat* ⇒ *num*›
  where
    ‹*num-of-nat 0* = *One*›
  | ‹*num-of-nat* (*Suc n*) = (*if 0* < *n then inc* (*num-of-nat n*) *else One*)›

lemma *nat-of-num-pos*: ‹*0* < *nat-of-num x*›
  by (*induct x*) *simp-all*

lemma *nat-of-num-neq-0*: ‹ *nat-of-num x* ≠ *0*›
  by (*induct x*) *simp-all*

lemma *nat-of-num-inc*: ‹*nat-of-num* (*inc x*) = *Suc* (*nat-of-num x*)›
  by (*induct x*) *simp-all*

lemma *num-of-nat-double*: ‹*0* < *n* ⟹ *num-of-nat* (*n* + *n*) = *Bit0* (*num-of-nat n*)›
  by (*induct n*) *simp-all*

Type *num* is isomorphic to the strictly positive natural numbers.

lemma *nat-of-num-inverse*: ‹*num-of-nat* (*nat-of-num x*) = *x*›
  by (*induct x*) (*simp-all add*: *num-of-nat-double nat-of-num-pos*)

lemma *num-of-nat-inverse*: ‹*0* < *n* ⟹ *nat-of-num* (*num-of-nat n*) = *n*›
  by (*induct n*) (*simp-all add*: *nat-of-num-inc*)

lemma *num-eq-iff*: ‹*x* = *y* ⟷ *nat-of-num x* = *nat-of-num y*›
  apply *safe*
  apply (*drule arg-cong* [**where** *f*=*num-of-nat*])

apply (*simp add*: *nat-of-num-inverse*)
done

lemma *num-induct* [*case-names One inc*]:
fixes P :: ‹*num* ⇒ *bool*›
assumes *One*: ‹P *One*›
and *inc*: ‹$\bigwedge x.\ P\ x \Longrightarrow P\ (inc\ x)$›
shows ‹P x›
proof −
obtain n **where** n: ‹*Suc* n = *nat-of-num* x›
by (*cases* ‹*nat-of-num* x›) (*simp-all add*: *nat-of-num-neq-0*)
have ‹P (*num-of-nat* (*Suc* n))›
proof (*induct* n)
case *0*
from *One* **show** *?case* **by** *simp*
next
case (*Suc* n)
then have ‹P (*inc* (*num-of-nat* (*Suc* n)))› **by** (*rule inc*)
then show ‹P (*num-of-nat* (*Suc* (*Suc* n)))› **by** *simp*
qed
with n **show** ‹P x›
by (*simp add*: *nat-of-num-inverse*)
qed

From now on, there are two possible models for *num*: as positive naturals (rule *num-induct*) and as digit representation (rules *num.induct*, *num.cases*).

46.2 Numeral operations

instantiation *num* :: ‹{*plus*,*times*,*linorder*}›
begin

definition ‹$m + n$ = *num-of-nat* (*nat-of-num* m + *nat-of-num* n)›

definition ‹$m * n$ = *num-of-nat* (*nat-of-num* m * *nat-of-num* n)›

definition ‹$m \leq n \longleftrightarrow$ *nat-of-num* $m \leq$ *nat-of-num* n›

definition ‹$m < n \longleftrightarrow$ *nat-of-num* $m <$ *nat-of-num* n›

instance
by *standard* (*auto simp add*: *less-num-def less-eq-num-def num-eq-iff*)

end

lemma *nat-of-num-add*: ‹*nat-of-num* (x + y) = *nat-of-num* x + *nat-of-num* y›
unfolding *plus-num-def*
by (*intro num-of-nat-inverse add-pos-pos nat-of-num-pos*)

lemma *nat-of-num-mult*: ‹*nat-of-num* (*x* ∗ *y*) = *nat-of-num x* ∗ *nat-of-num y*›
unfolding *times-num-def*
by (*intro num-of-nat-inverse mult-pos-pos nat-of-num-pos*)

lemma *add-num-simps* [*simp*, *code*]:
‹*One* + *One* = *Bit0 One*›
‹*One* + *Bit0 n* = *Bit1 n*›
‹*One* + *Bit1 n* = *Bit0* (*n* + *One*)›
‹*Bit0 m* + *One* = *Bit1 m*›
‹*Bit0 m* + *Bit0 n* = *Bit0* (*m* + *n*)›
‹*Bit0 m* + *Bit1 n* = *Bit1* (*m* + *n*)›
‹*Bit1 m* + *One* = *Bit0* (*m* + *One*)›
‹*Bit1 m* + *Bit0 n* = *Bit1* (*m* + *n*)›
‹*Bit1 m* + *Bit1 n* = *Bit0* (*m* + *n* + *One*)›
by (*simp-all add*: *num-eq-iff nat-of-num-add*)

lemma *mult-num-simps* [*simp*, *code*]:
‹*m* ∗ *One* = *m*›
‹*One* ∗ *n* = *n*›
‹*Bit0 m* ∗ *Bit0 n* = *Bit0* (*Bit0* (*m* ∗ *n*))›
‹*Bit0 m* ∗ *Bit1 n* = *Bit0* (*m* ∗ *Bit1 n*)›
‹*Bit1 m* ∗ *Bit0 n* = *Bit0* (*Bit1 m* ∗ *n*)›
‹*Bit1 m* ∗ *Bit1 n* = *Bit1* (*m* + *n* + *Bit0* (*m* ∗ *n*))›
by (*simp-all add*: *num-eq-iff nat-of-num-add nat-of-num-mult distrib-right distrib-left*)

lemma *eq-num-simps*:
‹*One* = *One* ⟷ *True*›
‹*One* = *Bit0 n* ⟷ *False*›
‹*One* = *Bit1 n* ⟷ *False*›
‹*Bit0 m* = *One* ⟷ *False*›
‹*Bit1 m* = *One* ⟷ *False*›
‹*Bit0 m* = *Bit0 n* ⟷ *m* = *n*›
‹*Bit0 m* = *Bit1 n* ⟷ *False*›
‹*Bit1 m* = *Bit0 n* ⟷ *False*›
‹*Bit1 m* = *Bit1 n* ⟷ *m* = *n*›
by *simp-all*

lemma *le-num-simps* [*simp*, *code*]:
‹*One* ≤ *n* ⟷ *True*›
‹*Bit0 m* ≤ *One* ⟷ *False*›
‹*Bit1 m* ≤ *One* ⟷ *False*›
‹*Bit0 m* ≤ *Bit0 n* ⟷ *m* ≤ *n*›
‹*Bit0 m* ≤ *Bit1 n* ⟷ *m* ≤ *n*›
‹*Bit1 m* ≤ *Bit1 n* ⟷ *m* ≤ *n*›
‹*Bit1 m* ≤ *Bit0 n* ⟷ *m* < *n*›
using *nat-of-num-pos* [*of n*] *nat-of-num-pos* [*of m*]
by (*auto simp add*: *less-eq-num-def less-num-def*)

lemma *less-num-simps* [*simp*, *code*]:
‹*m* < *One* ⟷ *False*›
‹*One* < *Bit0 n* ⟷ *True*›
‹*One* < *Bit1 n* ⟷ *True*›
‹*Bit0 m* < *Bit0 n* ⟷ *m* < *n*›
‹*Bit0 m* < *Bit1 n* ⟷ *m* ≤ *n*›
‹*Bit1 m* < *Bit1 n* ⟷ *m* < *n*›
‹*Bit1 m* < *Bit0 n* ⟷ *m* < *n*›
using *nat-of-num-pos* [*of n*] *nat-of-num-pos* [*of m*]
by (*auto simp add*: *less-eq-num-def less-num-def*)

lemma *le-num-One-iff*: ‹*x* ≤ *One* ⟷ *x* = *One*›
by (*simp add*: *antisym-conv*)

Rules using *One* and *inc* as constructors.

lemma *add-One*: ‹*x* + *One* = *inc x*›
by (*simp add*: *num-eq-iff nat-of-num-add nat-of-num-inc*)

lemma *add-One-commute*: ‹*One* + *n* = *n* + *One*›
by (*induct n*) *simp-all*

lemma *add-inc*: ‹*x* + *inc y* = *inc* (*x* + *y*)›
by (*simp add*: *num-eq-iff nat-of-num-add nat-of-num-inc*)

lemma *mult-inc*: ‹*x* ∗ *inc y* = *x* ∗ *y* + *x*›
by (*simp add*: *num-eq-iff nat-of-num-mult nat-of-num-add nat-of-num-inc*)

The *num-of-nat* conversion.

lemma *num-of-nat-One*: ‹*n* ≤ *1* ⟹ *num-of-nat n* = *One*›
by (*cases n*) *simp-all*

lemma *num-of-nat-plus-distrib*:
‹*0* < *m* ⟹ *0* < *n* ⟹ *num-of-nat* (*m* + *n*) = *num-of-nat m* + *num-of-nat n*›
by (*induct n*) (*auto simp add*: *add-One add-One-commute add-inc*)

A double-and-decrement function.

primrec *BitM* :: ‹*num* ⇒ *num*›
where
‹*BitM One* = *One*›
| ‹*BitM* (*Bit0 n*) = *Bit1* (*BitM n*)›
| ‹*BitM* (*Bit1 n*) = *Bit1* (*Bit0 n*)›

lemma *BitM-plus-one*: ‹*BitM n* + *One* = *Bit0 n*›
by (*induct n*) *simp-all*

lemma *one-plus-BitM*: ‹*One* + *BitM n* = *Bit0 n*›
unfolding *add-One-commute BitM-plus-one* **..**

lemma *BitM-inc-eq*:

‹*BitM* (*inc n*) = *Bit1 n*›
by (*induction n*) *simp-all*

lemma *inc-BitM-eq*:
‹*inc* (*BitM n*) = *Bit0 n*›
by (*simp add*: *BitM-plus-one*[*symmetric*] *add-One*)

Squaring and exponentiation.

primrec *sqr* :: ‹*num* ⇒ *num*›
where
‹*sqr One* = *One*›
| ‹*sqr* (*Bit0 n*) = *Bit0* (*Bit0* (*sqr n*))›
| ‹*sqr* (*Bit1 n*) = *Bit1* (*Bit0* (*sqr n* + *n*))›

primrec *pow* :: ‹*num* ⇒ *num* ⇒ *num*›
where
‹*pow x One* = *x*›
| ‹*pow x* (*Bit0 y*) = *sqr* (*pow x y*)›
| ‹*pow x* (*Bit1 y*) = *sqr* (*pow x y*) ∗ *x*›

lemma *nat-of-num-sqr*: ‹*nat-of-num* (*sqr x*) = *nat-of-num x* ∗ *nat-of-num x*›
by (*induct x*) (*simp-all add*: *algebra-simps nat-of-num-add*)

lemma *sqr-conv-mult*: ‹*sqr x* = *x* ∗ *x*›
by (*simp add*: *num-eq-iff nat-of-num-sqr nat-of-num-mult*)

lemma *num-double* [*simp*]:
‹*Bit0 num.One* ∗ *n* = *Bit0 n*›
by (*simp add*: *num-eq-iff nat-of-num-mult*)

46.3 Binary numerals

We embed binary representations into a generic algebraic structure using *numeral*.

class *numeral* = *one* + *semigroup-add*
begin

primrec *numeral* :: ‹*num* ⇒ *'a*›
where
numeral-One: ‹*numeral One* = *1*›
| *numeral-Bit0*: ‹*numeral* (*Bit0 n*) = *numeral n* + *numeral n*›
| *numeral-Bit1*: ‹*numeral* (*Bit1 n*) = *numeral n* + *numeral n* + *1*›

lemma *numeral-code* [*code*]:
‹*numeral One* = *1*›
‹*numeral* (*Bit0 n*) = (*let m* = *numeral n in m* + *m*)›
‹*numeral* (*Bit1 n*) = (*let m* = *numeral n in m* + *m* + *1*)›
by (*simp-all add*: *Let-def*)

lemma *one-plus-numeral-commute*: ‹*1 + numeral x = numeral x + 1*›
proof (*induct x*)
case *One*
then show *?case* **by** *simp*
next
case *Bit0*
then show *?case* **by** (*simp add*: *add.assoc* [*symmetric*]) (*simp add*: *add.assoc*)
next
case *Bit1*
then show *?case* **by** (*simp add*: *add.assoc* [*symmetric*]) (*simp add*: *add.assoc*)
qed

lemma *numeral-inc*: ‹*numeral* (*inc x*) = *numeral x + 1*›
proof (*induct x*)
case *One*
then show *?case* **by** *simp*
next
case *Bit0*
then show *?case* **by** *simp*
next
case (*Bit1 x*)
have ‹*numeral x + (1 + numeral x) + 1 = numeral x + (numeral x + 1) + 1*›
by (*simp only*: *one-plus-numeral-commute*)
with *Bit1* **show** *?case*
by (*simp add*: *add.assoc*)
qed

declare *numeral.simps* [*simp del*]

abbreviation ‹*Numeral1* ≡ *numeral One*›

declare *numeral-One* [*code-post*]

end

Numeral syntax.

syntax
-Numeral :: ‹*num-const* ⇒ *'a*› (‹(‹*open-block notation=*‹*literal number*››-)›)

ML-file ‹*Tools/numeral.ML*›

parse-translation ‹
let
fun numeral-tr [(*c as Const* (***syntax-const*** ‹*-constrain*›, -)) $ *t* $ *u*] =
c $ *numeral-tr* [*t*] $ *u*
| *numeral-tr* [*Const* (*num*, -)] =
(*Numeral.mk-number-syntax o #value o Lexicon.read-num*) *num*
| *numeral-tr ts* = *raise TERM* (*numeral-tr*, *ts*);

```
  in [(syntax-const ‹-Numeral›, K numeral-tr)] end
›

typed-print-translation ‹
  let
    fun num-tr' ctxt T [n] =
      let
        val k = Numeral.dest-num-syntax n;
        val t' =
          Syntax.const syntax-const ‹-Numeral› $
            Syntax.free (string-of-int k);
      in
        (case T of
          Type (type-name ‹fun›, [-, T']) =>
            if Printer.type-emphasis ctxt T' then
              Syntax.const syntax-const ‹-constrain› $ t' $
                Syntax-Phases.term-of-typ ctxt T'
            else t'
        | - => if T = dummyT then t' else raise Match)
      end;
  in
   [(const-syntax ‹numeral›, num-tr')]
  end
›
```

46.4 Class-specific numeral rules

numeral is a morphism.

46.4.1 Structures with addition: class *numeral*

context *numeral*
begin

lemma *numeral-add*: ‹*numeral* (*m* + *n*) = *numeral m* + *numeral n*›
  by (*induct n rule*: *num-induct*)
    (*simp-all only*: *numeral-One add-One add-inc numeral-inc add.assoc*)

lemma *numeral-plus-numeral*: ‹*numeral m* + *numeral n* = *numeral* (*m* + *n*)›
  by (*rule numeral-add* [*symmetric*])

lemma *numeral-plus-one*: ‹*numeral n* + *1* = *numeral* (*n* + *One*)›
  using *numeral-add* [*of n One*] **by** (*simp add*: *numeral-One*)

lemma *one-plus-numeral*: ‹*1* + *numeral n* = *numeral* (*One* + *n*)›
  using *numeral-add* [*of One n*] **by** (*simp add*: *numeral-One*)

lemma *one-add-one*: ‹*1* + *1* = *2*›
  using *numeral-add* [*of One One*] **by** (*simp add*: *numeral-One*)

lemmas *add-numeral-special* =
numeral-plus-one one-plus-numeral one-add-one

end

46.4.2 Structures with negation: class *neg-numeral*

class *neg-numeral* = *numeral* + *group-add*
begin

lemma *uminus-numeral-One*: ‹− *Numeral1* = − *1*›
by (*simp add*: *numeral-One*)

Numerals form an abelian subgroup.

inductive *is-num* :: ‹'*a* ⇒ *bool*›
where
‹*is-num 1*›
| ‹*is-num x* ⟹ *is-num* (− *x*)›
| ‹*is-num x* ⟹ *is-num y* ⟹ *is-num* (*x* + *y*)›

lemma *is-num-numeral*: ‹*is-num* (*numeral k*)›
by (*induct k*) (*simp-all add*: *numeral.simps is-num.intros*)

lemma *is-num-add-commute*: ‹*is-num x* ⟹ *is-num y* ⟹ *x* + *y* = *y* + *x*›
proof(*induction x rule*: *is-num.induct*)
case *1*
then show *?case*
proof (*induction y rule*: *is-num.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 y*)
then have ‹*y* + (*1* + − *y*) + *y* = *y* + (− *y* + *1*) + *y*›
by (*simp add*: *add.assoc*)
then have ‹*y* + (*1* + − *y*) = *y* + (− *y* + *1*)›
by *simp*
then show *?case*
by (*rule add-left-imp-eq*[*of y*])
next
case (*3 x y*)
then have ‹*1* + (*x* + *y*) = *x* + *1* + *y*›
by (*simp add*: *add.assoc* [*symmetric*])
then show *?case* **using** *3*
by (*simp add*: *add.assoc*)
qed
next
case (*2 x*)
then have ‹*x* + (− *x* + *y*) + *x* = *x* + (*y* + − *x*) + *x*›

by (*simp add*: *add.assoc*)
then have ‹*x* + (− *x* + *y*) = *x* + (*y* + − *x*)›
by *simp*
then show *?case*
by (*rule add-left-imp-eq*[*of x*])
next
case (*3 x z*)
moreover have ‹*x* + (*y* + *z*) = (*x* + *y*) + *z*›
by (*simp add*: *add.assoc*[*symmetric*])
ultimately show *?case*
by (*simp add*: *add.assoc*)
qed

lemma *is-num-add-left-commute*: ‹*is-num x* ⟹ *is-num y* ⟹ *x* + (*y* + *z*) = *y* + (*x* + *z*)›
by (*simp only*: *add.assoc* [*symmetric*] *is-num-add-commute*)

lemmas *is-num-normalize* =
add.assoc is-num-add-commute is-num-add-left-commute
is-num.intros is-num-numeral
minus-add

definition *dbl* :: ‹'*a* ⇒ '*a*›
where ‹*dbl x* = *x* + *x*›

definition *dbl-inc* :: ‹'*a* ⇒ '*a*›
where ‹*dbl-inc x* = *x* + *x* + *1*›

definition *dbl-dec* :: ‹'*a* ⇒ '*a*›
where ‹*dbl-dec x* = *x* + *x* − *1*›

definition *sub* :: ‹*num* ⇒ *num* ⇒ '*a*›
where ‹*sub k l* = *numeral k* − *numeral l*›

lemma *numeral-BitM*: ‹*numeral* (*BitM n*) = *numeral* (*Bit0 n*) − *1*›
by (*simp only*: *BitM-plus-one* [*symmetric*] *numeral-add numeral-One eq-diff-eq*)

lemma *sub-inc-One-eq*:
‹*sub* (*inc n*) *num.One* = *numeral n*›
by (*simp-all add*: *sub-def diff-eq-eq numeral-inc numeral.numeral-One*)

lemma *dbl-simps* [*simp*]:
‹*dbl* (− *numeral k*) = − *dbl* (*numeral k*)›
‹*dbl 0* = *0*›
‹*dbl 1* = *2*›
‹*dbl* (− *1*) = − *2*›
‹*dbl* (*numeral k*) = *numeral* (*Bit0 k*)›
by (*simp-all add*: *dbl-def numeral.simps minus-add*)

lemma *dbl-inc-simps* [*simp*]:
‹*dbl-inc* (− *numeral k*) = − *dbl-dec* (*numeral k*)›
‹*dbl-inc 0* = *1*›
‹*dbl-inc 1* = *3*›
‹*dbl-inc* (− *1*) = − *1*›
‹*dbl-inc* (*numeral k*) = *numeral* (*Bit1 k*)›
by (*simp-all add*: *dbl-inc-def dbl-dec-def numeral.simps numeral-BitM is-num-normalize algebra-simps*
del: *add-uminus-conv-diff*)

lemma *dbl-dec-simps* [*simp*]:
‹*dbl-dec* (− *numeral k*) = − *dbl-inc* (*numeral k*)›
‹*dbl-dec 0* = − *1*›
‹*dbl-dec 1* = *1*›
‹*dbl-dec* (− *1*) = − *3*›
‹*dbl-dec* (*numeral k*) = *numeral* (*BitM k*)›
by (*simp-all add*: *dbl-dec-def dbl-inc-def numeral.simps numeral-BitM is-num-normalize*)

lemma *sub-num-simps* [*simp*]:
‹*sub One One* = *0*›
‹*sub One* (*Bit0 l*) = − *numeral* (*BitM l*)›
‹*sub One* (*Bit1 l*) = − *numeral* (*Bit0 l*)›
‹*sub* (*Bit0 k*) *One* = *numeral* (*BitM k*)›
‹*sub* (*Bit1 k*) *One* = *numeral* (*Bit0 k*)›
‹*sub* (*Bit0 k*) (*Bit0 l*) = *dbl* (*sub k l*)›
‹*sub* (*Bit0 k*) (*Bit1 l*) = *dbl-dec* (*sub k l*)›
‹*sub* (*Bit1 k*) (*Bit0 l*) = *dbl-inc* (*sub k l*)›
‹*sub* (*Bit1 k*) (*Bit1 l*) = *dbl* (*sub k l*)›
by (*simp-all add*: *dbl-def dbl-dec-def dbl-inc-def sub-def numeral.simps*
numeral-BitM is-num-normalize del: *add-uminus-conv-diff add*: *diff-conv-add-uminus*)

lemma *add-neg-numeral-simps*:
‹*numeral m* + − *numeral n* = *sub m n*›
‹− *numeral m* + *numeral n* = *sub n m*›
‹− *numeral m* + − *numeral n* = − (*numeral m* + *numeral n*)›
by (*simp-all add*: *sub-def numeral-add numeral.simps is-num-normalize*
del: *add-uminus-conv-diff add*: *diff-conv-add-uminus*)

lemma *add-neg-numeral-special*:
‹*1* + − *numeral m* = *sub One m*›
‹− *numeral m* + *1* = *sub One m*›
‹*numeral m* + − *1* = *sub m One*›
‹− *1* + *numeral n* = *sub n One*›
‹− *1* + − *numeral n* = − *numeral* (*inc n*)›
‹− *numeral m* + − *1* = − *numeral* (*inc m*)›
‹*1* + − *1* = *0*›
‹− *1* + *1* = *0*›
‹− *1* + − *1* = − *2*›
by (*simp-all add*: *sub-def numeral-add numeral.simps is-num-normalize right-minus*

numeral-inc
del: *add-uminus-conv-diff add*: *diff-conv-add-uminus*)

lemma *diff-numeral-simps*:
‹*numeral m − numeral n = sub m n*›
‹*numeral m − − numeral n = numeral (m + n)*›
‹*− numeral m − numeral n = − numeral (m + n)*›
‹*− numeral m − − numeral n = sub n m*›
by (*simp-all add*: *sub-def numeral-add numeral.simps is-num-normalize*
del: *add-uminus-conv-diff add*: *diff-conv-add-uminus*)

lemma *diff-numeral-special*:
‹*1 − numeral n = sub One n*›
‹*numeral m − 1 = sub m One*›
‹*1 − − numeral n = numeral (One + n)*›
‹*− numeral m − 1 = − numeral (m + One)*›
‹*− 1 − numeral n = − numeral (inc n)*›
‹*numeral m − − 1 = numeral (inc m)*›
‹*− 1 − − numeral n = sub n One*›
‹*− numeral m − − 1 = sub One m*›
‹*1 − 1 = 0*›
‹*− 1 − 1 = − 2*›
‹*1 − − 1 = 2*›
‹*− 1 − − 1 = 0*›
by (*simp-all add*: *sub-def numeral-add numeral.simps is-num-normalize numeral-inc*
del: *add-uminus-conv-diff add*: *diff-conv-add-uminus*)

end

46.4.3 Structures with multiplication: class *semiring-numeral*

class *semiring-numeral = semiring + monoid-mult*
begin

subclass *numeral* **..**

lemma *numeral-mult*: ‹*numeral (m * n) = numeral m * numeral n*›
by (*induct n rule*: *num-induct*)
(*simp-all add*: *numeral-One mult-inc numeral-inc numeral-add distrib-left*)

lemma *numeral-times-numeral*: ‹*numeral m * numeral n = numeral (m * n)*›
by (*rule numeral-mult* [*symmetric*])

lemma *mult-2*: ‹*2 * z = z + z*›
by (*simp add*: *one-add-one* [*symmetric*] *distrib-right*)

lemma *mult-2-right*: ‹*z * 2 = z + z*›
by (*simp add*: *one-add-one* [*symmetric*] *distrib-left*)

lemma *left-add-twice*:
‹$a + (a + b) = 2 * a + b$›
by (*simp add*: *mult-2 ac-simps*)

lemma *numeral-Bit0-eq-double*:
‹*numeral* (*Bit0 n*) = *2* * *numeral n*›
by (*simp add*: *mult-2*) (*simp add*: *numeral-Bit0*)

lemma *numeral-Bit1-eq-inc-double*:
‹*numeral* (*Bit1 n*) = *2* * *numeral n* + *1*›
by (*simp add*: *mult-2*) (*simp add*: *numeral-Bit1*)

end

46.4.4 Structures with a zero: class *semiring-1*

context *semiring-1*
begin

subclass *semiring-numeral* **..**

lemma *of-nat-numeral* [*simp*]: ‹*of-nat* (*numeral n*) = *numeral n*›
by (*induct n*) (*simp-all only*: *numeral.simps numeral-class.numeral.simps of-nat-add of-nat-1*)

end

lemma *nat-of-num-numeral* [*code-abbrev*]: ‹*nat-of-num* = *numeral*›
proof
fix *n*
have ‹*numeral n* = *nat-of-num n*›
by (*induct n*) (*simp-all add*: *numeral.simps*)
then show ‹*nat-of-num n* = *numeral n*›
by *simp*
qed

lemma *nat-of-num-code* [*code*]:
‹*nat-of-num One* = *1*›
‹*nat-of-num* (*Bit0 n*) = (*let m* = *nat-of-num n in m* + *m*)›
‹*nat-of-num* (*Bit1 n*) = (*let m* = *nat-of-num n in Suc* (*m* + *m*))›
by (*simp-all add*: *Let-def*)

46.4.5 Equality: class *semiring-char-0*

context *semiring-char-0*
begin

lemma *numeral-eq-iff*: ‹*numeral m* = *numeral n* ⟷ *m* = *n*›
by (*simp only*: *of-nat-numeral* [*symmetric*] *nat-of-num-numeral* [*symmetric*] *of-nat-eq-iff num-eq-iff*)

lemma *numeral-eq-one-iff*: ‹*numeral n = 1 ⟷ n = One*›
by (*rule numeral-eq-iff* [*of n One*, *unfolded numeral-One*])

lemma *one-eq-numeral-iff*: ‹*1 = numeral n ⟷ One = n*›
by (*rule numeral-eq-iff* [*of One n*, *unfolded numeral-One*])

lemma *numeral-neq-zero*: ‹*numeral n ≠ 0*›
by (*simp add*: *of-nat-numeral* [*symmetric*] *nat-of-num-numeral* [*symmetric*] *nat-of-num-pos*)

lemma *zero-neq-numeral*: ‹*0 ≠ numeral n*›
unfolding *eq-commute* [*of 0*] **by** (*rule numeral-neq-zero*)

lemmas *eq-numeral-simps* [*simp*] =
numeral-eq-iff
numeral-eq-one-iff
one-eq-numeral-iff
numeral-neq-zero
zero-neq-numeral

end

46.4.6 Comparisons: class *linordered-nonzero-semiring*

context *linordered-nonzero-semiring*
begin

lemma *numeral-le-iff*: ‹*numeral m ≤ numeral n ⟷ m ≤ n*›
proof −
have ‹*of-nat* (*numeral m*) ≤ *of-nat* (*numeral n*) ⟷ *m ≤ n*›
by (*simp only*: *less-eq-num-def nat-of-num-numeral of-nat-le-iff*)
then show *?thesis* **by** *simp*
qed

lemma *one-le-numeral*: ‹*1 ≤ numeral n*›
using *numeral-le-iff* [*of One n*] **by** (*simp add*: *numeral-One*)

lemma *numeral-le-one-iff*: ‹*numeral n ≤ 1 ⟷ n ≤ One*›
using *numeral-le-iff* [*of n One*] **by** (*simp add*: *numeral-One*)

lemma *numeral-less-iff*: ‹*numeral m < numeral n ⟷ m < n*›
proof −
have ‹*of-nat* (*numeral m*) < *of-nat* (*numeral n*) ⟷ *m < n*›
unfolding *less-num-def nat-of-num-numeral of-nat-less-iff* **..**
then show *?thesis* **by** *simp*
qed

lemma *not-numeral-less-one*: ‹¬ *numeral n < 1*›
using *numeral-less-iff* [*of n One*] **by** (*simp add*: *numeral-One*)

lemma *one-less-numeral-iff*: ‹*1 < numeral n ⟷ One < n*›
using *numeral-less-iff* [*of One n*] **by** (*simp add*: *numeral-One*)

lemma *zero-le-numeral*: ‹*0 ≤ numeral n*›
using *dual-order.trans one-le-numeral zero-le-one* **by** *blast*

lemma *zero-less-numeral*: ‹*0 < numeral n*›
using *less-linear not-numeral-less-one order.strict-trans zero-less-one* **by** *blast*

lemma *not-numeral-le-zero*: ‹¬ *numeral n ≤ 0*›
by (*simp add*: *not-le zero-less-numeral*)

lemma *not-numeral-less-zero*: ‹¬ *numeral n < 0*›
by (*simp add*: *not-less zero-le-numeral*)

lemma *one-of-nat-le-iff* [*simp*]: ‹*1 ≤ of-nat k ⟷ 1 ≤ k*›
using *of-nat-le-iff* [*of 1*] **by** *simp*

lemma *numeral-nat-le-iff* [*simp*]: ‹*numeral n ≤ of-nat k ⟷ numeral n ≤ k*›
using *of-nat-le-iff* [*of* ‹*numeral n*›] **by** *simp*

lemma *of-nat-le-1-iff* [*simp*]: ‹*of-nat k ≤ 1 ⟷ k ≤ 1*›
using *of-nat-le-iff* [*of - 1*] **by** *simp*

lemma *of-nat-le-numeral-iff* [*simp*]: ‹*of-nat k ≤ numeral n ⟷ k ≤ numeral n*›
using *of-nat-le-iff* [*of -* ‹*numeral n*›] **by** *simp*

lemma *one-of-nat-less-iff* [*simp*]: ‹*1 < of-nat k ⟷ 1 < k*›
using *of-nat-less-iff* [*of 1*] **by** *simp*

lemma *numeral-nat-less-iff* [*simp*]: ‹*numeral n < of-nat k ⟷ numeral n < k*›
using *of-nat-less-iff* [*of* ‹*numeral n*›] **by** *simp*

lemma *of-nat-less-1-iff* [*simp*]: ‹*of-nat k < 1 ⟷ k < 1*›
using *of-nat-less-iff* [*of - 1*] **by** *simp*

lemma *of-nat-less-numeral-iff* [*simp*]: ‹*of-nat k < numeral n ⟷ k < numeral n*›
using *of-nat-less-iff* [*of -* ‹*numeral n*›] **by** *simp*

lemma *of-nat-eq-numeral-iff* [*simp*]: ‹*of-nat k = numeral n ⟷ k = numeral n*›
using *of-nat-eq-iff* [*of -* ‹*numeral n*›] **by** *simp*

lemmas *le-numeral-extra* =
zero-le-one not-one-le-zero
order-refl [*of 0*] *order-refl* [*of 1*]

lemmas *less-numeral-extra* =
zero-less-one not-one-less-zero

less-irrefl [*of 0*] *less-irrefl* [*of 1*]

lemmas *le-numeral-simps* [*simp*] =
numeral-le-iff
one-le-numeral
numeral-le-one-iff
zero-le-numeral
not-numeral-le-zero

lemmas *less-numeral-simps* [*simp*] =
numeral-less-iff
one-less-numeral-iff
not-numeral-less-one
zero-less-numeral
not-numeral-less-zero

lemma *min-0-1* [*simp*]:
fixes *min'* :: ‹'*a* ⇒ '*a* ⇒ '*a*›
defines ‹*min'* ≡ *min*›
shows
‹*min' 0 1* = *0*›
‹*min' 1 0* = *0*›
‹*min' 0* (*numeral x*) = *0*›
‹*min'* (*numeral x*) *0* = *0*›
‹*min' 1* (*numeral x*) = *1*›
‹*min'* (*numeral x*) *1* = *1*›
by (*simp-all add*: *min'-def min-def le-num-One-iff*)

lemma *max-0-1* [*simp*]:
fixes *max'* :: ‹'*a* ⇒ '*a* ⇒ '*a*›
defines ‹*max'* ≡ *max*›
shows
‹*max' 0 1* = *1*›
‹*max' 1 0* = *1*›
‹*max' 0* (*numeral x*) = *numeral x*›
‹*max'* (*numeral x*) *0* = *numeral x*›
‹*max' 1* (*numeral x*) = *numeral x*›
‹*max'* (*numeral x*) *1* = *numeral x*›
by (*simp-all add*: *max'-def max-def le-num-One-iff*)

end

Unfold *min* and *max* on numerals.

lemmas *max-number-of* [*simp*] =
max-def [*of* ‹*numeral u*› ‹*numeral v*›]
max-def [*of* ‹*numeral u*› ‹− *numeral v*›]
max-def [*of* ‹− *numeral u*› ‹*numeral v*›]
max-def [*of* ‹− *numeral u*› ‹− *numeral v*›] **for** *u v*

lemmas *min-number-of* [*simp*] =
min-def [*of* ‹*numeral u*› ‹*numeral v*›]
min-def [*of* ‹*numeral u*› ‹− *numeral v*›]
min-def [*of* ‹− *numeral u*› ‹*numeral v*›]
min-def [*of* ‹− *numeral u*› ‹− *numeral v*›] **for** *u v*

46.4.7 Multiplication and negation: class *ring-1*

context *ring-1*
begin

subclass *neg-numeral* **..**

lemma *mult-neg-numeral-simps*:
‹− *numeral m* ∗ − *numeral n* = *numeral* (*m* ∗ *n*)›
‹− *numeral m* ∗ *numeral n* = − *numeral* (*m* ∗ *n*)›
‹*numeral m* ∗ − *numeral n* = − *numeral* (*m* ∗ *n*)›
by (*simp-all only*: *mult-minus-left mult-minus-right minus-minus numeral-mult*)

lemma *mult-minus1* [*simp*]: ‹− 1 ∗ *z* = − *z*›
by (*simp add*: *numeral.simps*)

lemma *mult-minus1-right* [*simp*]: ‹*z* ∗ − 1 = − *z*›
by (*simp add*: *numeral.simps*)

lemma *minus-sub-one-diff-one* [*simp*]:
‹− *sub m One* − 1 = − *numeral m*›
proof −
have ‹*sub m One* + 1 = *numeral m*›
by (*simp flip*: *eq-diff-eq add*: *diff-numeral-special*)
then have ‹− (*sub m One* + 1) = − *numeral m*›
by *simp*
then show *?thesis*
by *simp*
qed

end

46.4.8 Equality using *iszero* for rings with non-zero characteristic

context *ring-1*
begin

definition *iszero* :: ‹'*a* ⇒ *bool*›
where ‹*iszero z* ⟷ *z* = 0›

lemma *iszero-0* [*simp*]: ‹*iszero* 0›
by (*simp add*: *iszero-def*)

lemma *not-iszero-1* [*simp*]: ‹¬ *iszero* 1›

by (*simp add*: *iszero-def*)

lemma *not-iszero-Numeral1*: ‹¬ *iszero Numeral1*›
by (*simp add*: *numeral-One*)

lemma *not-iszero-neg-1* [*simp*]: ‹¬ *iszero* (− *1*)›
by (*simp add*: *iszero-def*)

lemma *not-iszero-neg-Numeral1*: ‹¬ *iszero* (− *Numeral1*)›
by (*simp add*: *numeral-One*)

lemma *iszero-neg-numeral* [*simp*]: ‹*iszero* (− *numeral w*) ⟷ *iszero* (*numeral w*)›
unfolding *iszero-def* **by** (*rule neg-equal-0-iff-equal*)

lemma *eq-iff-iszero-diff*: ‹*x* = *y* ⟷ *iszero* (*x* − *y*)›
unfolding *iszero-def* **by** (*rule eq-iff-diff-eq-0*)

The *eq-numeral-iff-iszero* lemmas are not declared [*simp*] by default, because for rings of characteristic zero, better simp rules are possible. For a type like integers mod n, type-instantiated versions of these rules should be added to the simplifier, along with a type-specific rule for deciding propositions of the form *iszero* (*numeral w*).

bh: Maybe it would not be so bad to just declare these as simp rules anyway? I should test whether these rules take precedence over the *ring-char-0* rules in the simplifier.

lemma *eq-numeral-iff-iszero*:
‹*numeral x* = *numeral y* ⟷ *iszero* (*sub x y*)›
‹*numeral x* = − *numeral y* ⟷ *iszero* (*numeral* (*x* + *y*))›
‹− *numeral x* = *numeral y* ⟷ *iszero* (*numeral* (*x* + *y*))›
‹− *numeral x* = − *numeral y* ⟷ *iszero* (*sub y x*)›
‹*numeral x* = *1* ⟷ *iszero* (*sub x One*)›
‹*1* = *numeral y* ⟷ *iszero* (*sub One y*)›
‹− *numeral x* = *1* ⟷ *iszero* (*numeral* (*x* + *One*))›
‹*1* = − *numeral y* ⟷ *iszero* (*numeral* (*One* + *y*))›
‹*numeral x* = *0* ⟷ *iszero* (*numeral x*)›
‹*0* = *numeral y* ⟷ *iszero* (*numeral y*)›
‹− *numeral x* = *0* ⟷ *iszero* (*numeral x*)›
‹*0* = − *numeral y* ⟷ *iszero* (*numeral y*)›
unfolding *eq-iff-iszero-diff diff-numeral-simps diff-numeral-special*
by *simp-all*

end

46.4.9 Equality and negation: class *ring-char-0*

context *ring-char-0*
begin

lemma *not-iszero-numeral* [*simp*]: ‹¬ *iszero* (*numeral w*)›
by (*simp add*: *iszero-def*)

lemma *neg-numeral-eq-iff*: ‹− *numeral m* = − *numeral n* ⟷ *m* = *n*›
by *simp*

lemma *numeral-neq-neg-numeral*: ‹*numeral m* ≠ − *numeral n*›
by (*simp add*: *eq-neg-iff-add-eq-0 numeral-plus-numeral*)

lemma *neg-numeral-neq-numeral*: ‹− *numeral m* ≠ *numeral n*›
by (*rule numeral-neq-neg-numeral* [*symmetric*])

lemma *zero-neq-neg-numeral*: ‹*0* ≠ − *numeral n*›
by *simp*

lemma *neg-numeral-neq-zero*: ‹− *numeral n* ≠ *0*›
by *simp*

lemma *one-neq-neg-numeral*: ‹*1* ≠ − *numeral n*›
using *numeral-neq-neg-numeral* [*of One n*] **by** (*simp add*: *numeral-One*)

lemma *neg-numeral-neq-one*: ‹− *numeral n* ≠ *1*›
using *neg-numeral-neq-numeral* [*of n One*] **by** (*simp add*: *numeral-One*)

lemma *neg-one-neq-numeral*: ‹− *1* ≠ *numeral n*›
using *neg-numeral-neq-numeral* [*of One n*] **by** (*simp add*: *numeral-One*)

lemma *numeral-neq-neg-one*: ‹*numeral n* ≠ − *1*›
using *numeral-neq-neg-numeral* [*of n One*] **by** (*simp add*: *numeral-One*)

lemma *neg-one-eq-numeral-iff*: ‹− *1* = − *numeral n* ⟷ *n* = *One*›
using *neg-numeral-eq-iff* [*of One n*] **by** (*auto simp add*: *numeral-One*)

lemma *numeral-eq-neg-one-iff*: ‹− *numeral n* = − *1* ⟷ *n* = *One*›
using *neg-numeral-eq-iff* [*of n One*] **by** (*auto simp add*: *numeral-One*)

lemma *neg-one-neq-zero*: ‹− *1* ≠ *0*›
by *simp*

lemma *zero-neq-neg-one*: ‹*0* ≠ − *1*›
by *simp*

lemma *neg-one-neq-one*: ‹− *1* ≠ *1*›
using *neg-numeral-neq-numeral* [*of One One*] **by** (*simp only*: *numeral-One not-False-eq-True*)

lemma *one-neq-neg-one*: ‹*1* ≠ − *1*›
using *numeral-neq-neg-numeral* [*of One One*] **by** (*simp only*: *numeral-One not-False-eq-True*)

lemmas *eq-neg-numeral-simps* [*simp*] =

neg-numeral-eq-iff
numeral-neq-neg-numeral neg-numeral-neq-numeral
one-neq-neg-numeral neg-numeral-neq-one
zero-neq-neg-numeral neg-numeral-neq-zero
neg-one-neq-numeral numeral-neq-neg-one
neg-one-eq-numeral-iff numeral-eq-neg-one-iff
neg-one-neq-zero zero-neq-neg-one
neg-one-neq-one one-neq-neg-one

end

46.4.10 Structures with negation and order: class *linordered-idom*

context *linordered-idom*
begin

subclass *ring-char-0* **..**

lemma *neg-numeral-le-iff*: ‹− *numeral m* ≤ − *numeral n* ⟷ *n* ≤ *m*›
by (*simp only*: *neg-le-iff-le numeral-le-iff*)

lemma *neg-numeral-less-iff*: ‹− *numeral m* < − *numeral n* ⟷ *n* < *m*›
by (*simp only*: *neg-less-iff-less numeral-less-iff*)

lemma *neg-numeral-less-zero*: ‹− *numeral n* < *0*›
by (*simp only*: *neg-less-0-iff-less zero-less-numeral*)

lemma *neg-numeral-le-zero*: ‹− *numeral n* ≤ *0*›
by (*simp only*: *neg-le-0-iff-le zero-le-numeral*)

lemma *not-zero-less-neg-numeral*: ‹¬ *0* < − *numeral n*›
by (*simp only*: *not-less neg-numeral-le-zero*)

lemma *not-zero-le-neg-numeral*: ‹¬ *0* ≤ − *numeral n*›
by (*simp only*: *not-le neg-numeral-less-zero*)

lemma *neg-numeral-less-numeral*: ‹− *numeral m* < *numeral n*›
using *neg-numeral-less-zero zero-less-numeral* **by** (*rule less-trans*)

lemma *neg-numeral-le-numeral*: ‹− *numeral m* ≤ *numeral n*›
by (*simp only*: *less-imp-le neg-numeral-less-numeral*)

lemma *not-numeral-less-neg-numeral*: ‹¬ *numeral m* < − *numeral n*›
by (*simp only*: *not-less neg-numeral-le-numeral*)

lemma *not-numeral-le-neg-numeral*: ‹¬ *numeral m* ≤ − *numeral n*›
by (*simp only*: *not-le neg-numeral-less-numeral*)

lemma *neg-numeral-less-one*: ‹− *numeral m* < *1*›

by (*rule neg-numeral-less-numeral* [*of m One*, *unfolded numeral-One*])

lemma *neg-numeral-le-one*: ‹*− numeral m ≤ 1*›
by (*rule neg-numeral-le-numeral* [*of m One*, *unfolded numeral-One*])

lemma *not-one-less-neg-numeral*: ‹*¬ 1 < − numeral m*›
by (*simp only*: *not-less neg-numeral-le-one*)

lemma *not-one-le-neg-numeral*: ‹*¬ 1 ≤ − numeral m*›
by (*simp only*: *not-le neg-numeral-less-one*)

lemma *not-numeral-less-neg-one*: ‹*¬ numeral m < − 1*›
using *not-numeral-less-neg-numeral* [*of m One*] **by** (*simp add*: *numeral-One*)

lemma *not-numeral-le-neg-one*: ‹*¬ numeral m ≤ − 1*›
using *not-numeral-le-neg-numeral* [*of m One*] **by** (*simp add*: *numeral-One*)

lemma *neg-one-less-numeral*: ‹*− 1 < numeral m*›
using *neg-numeral-less-numeral* [*of One m*] **by** (*simp add*: *numeral-One*)

lemma *neg-one-le-numeral*: ‹*− 1 ≤ numeral m*›
using *neg-numeral-le-numeral* [*of One m*] **by** (*simp add*: *numeral-One*)

lemma *neg-numeral-less-neg-one-iff*: ‹*− numeral m < − 1 ⟷ m ≠ One*›
by (*cases m*) *simp-all*

lemma *neg-numeral-le-neg-one*: ‹*− numeral m ≤ − 1*›
by *simp*

lemma *not-neg-one-less-neg-numeral*: ‹*¬ − 1 < − numeral m*›
by *simp*

lemma *not-neg-one-le-neg-numeral-iff*: ‹*¬ − 1 ≤ − numeral m ⟷ m ≠ One*›
by (*cases m*) *simp-all*

lemma *sub-non-negative*: ‹*sub n m ≥ 0 ⟷ n ≥ m*›
by (*simp only*: *sub-def le-diff-eq*) *simp*

lemma *sub-positive*: ‹*sub n m > 0 ⟷ n > m*›
by (*simp only*: *sub-def less-diff-eq*) *simp*

lemma *sub-non-positive*: ‹*sub n m ≤ 0 ⟷ n ≤ m*›
by (*simp only*: *sub-def diff-le-eq*) *simp*

lemma *sub-negative*: ‹*sub n m < 0 ⟷ n < m*›
by (*simp only*: *sub-def diff-less-eq*) *simp*

lemmas *le-neg-numeral-simps* [*simp*] =
neg-numeral-le-iff

neg-numeral-le-numeral not-numeral-le-neg-numeral
neg-numeral-le-zero not-zero-le-neg-numeral
neg-numeral-le-one not-one-le-neg-numeral
neg-one-le-numeral not-numeral-le-neg-one
neg-numeral-le-neg-one not-neg-one-le-neg-numeral-iff

lemma *le-minus-one-simps* [*simp*]:
‹− *1* ≤ *0*›
‹− *1* ≤ *1*›
‹¬ *0* ≤ − *1*›
‹¬ *1* ≤ − *1*›
by *simp-all*

lemmas *less-neg-numeral-simps* [*simp*] =
neg-numeral-less-iff
neg-numeral-less-numeral not-numeral-less-neg-numeral
neg-numeral-less-zero not-zero-less-neg-numeral
neg-numeral-less-one not-one-less-neg-numeral
neg-one-less-numeral not-numeral-less-neg-one
neg-numeral-less-neg-one-iff not-neg-one-less-neg-numeral

lemma *less-minus-one-simps* [*simp*]:
‹− *1* < *0*›
‹− *1* < *1*›
‹¬ *0* < − *1*›
‹¬ *1* < − *1*›
by (*simp-all add*: *less-le*)

lemma *abs-numeral* [*simp*]: ‹|*numeral n*| = *numeral n*›
by *simp*

lemma *abs-neg-numeral* [*simp*]: ‹|− *numeral n*| = *numeral n*›
by (*simp only*: *abs-minus-cancel abs-numeral*)

lemma *abs-neg-one* [*simp*]: ‹|− *1*| = *1*›
by *simp*

end

46.4.11 Natural numbers

lemma *numeral-num-of-nat*:
‹*numeral* (*num-of-nat n*) = *n*› **if** ‹*n* > *0*›
using *that nat-of-num-numeral num-of-nat-inverse* **by** *simp*

lemma *Suc-1* [*simp*]: ‹*Suc 1* = *2*›
unfolding *Suc-eq-plus1* **by** (*rule one-add-one*)

lemma *Suc-numeral* [*simp*]: ‹*Suc* (*numeral n*) = *numeral* (*n* + *One*)›

unfolding *Suc-eq-plus1* **by** (*rule numeral-plus-one*)

definition *pred-numeral* :: ‹*num* ⇒ *nat*›
where ‹*pred-numeral k* = *numeral k* − *1*›

declare [[*code drop*: *pred-numeral*]]

lemma *numeral-eq-Suc*: ‹*numeral k* = *Suc* (*pred-numeral k*)›
by (*simp add*: *pred-numeral-def*)

lemma *eval-nat-numeral*:
‹*numeral One* = *Suc 0*›
‹*numeral* (*Bit0 n*) = *Suc* (*numeral* (*BitM n*))›
‹*numeral* (*Bit1 n*) = *Suc* (*numeral* (*Bit0 n*))›
by (*simp-all add*: *numeral.simps BitM-plus-one*)

lemma *pred-numeral-simps* [*simp*]:
‹*pred-numeral One* = *0*›
‹*pred-numeral* (*Bit0 k*) = *numeral* (*BitM k*)›
‹*pred-numeral* (*Bit1 k*) = *numeral* (*Bit0 k*)›
by (*simp-all only*: *pred-numeral-def eval-nat-numeral diff-Suc-Suc diff-0*)

lemma *pred-numeral-inc* [*simp*]:
‹*pred-numeral* (*inc k*) = *numeral k*›
by (*simp only*: *pred-numeral-def numeral-inc diff-add-inverse2*)

lemma *numeral-2-eq-2*: ‹*2* = *Suc* (*Suc 0*)›
by (*simp add*: *eval-nat-numeral*)

lemma *numeral-3-eq-3*: ‹*3* = *Suc* (*Suc* (*Suc 0*))›
by (*simp add*: *eval-nat-numeral*)

lemma *numeral-1-eq-Suc-0*: ‹*Numeral1* = *Suc 0*›
by (*simp only*: *numeral-One One-nat-def*)

lemma *Suc-nat-number-of-add*: ‹*Suc* (*numeral v* + *n*) = *numeral* (*v* + *One*) + *n*›
by *simp*

lemma *numerals*: ‹*Numeral1* = (*1*::*nat*)› ‹*2* = *Suc* (*Suc 0*)›
by (*rule numeral-One*) (*rule numeral-2-eq-2*)

lemmas *numeral-nat* = *eval-nat-numeral BitM.simps One-nat-def*

Comparisons involving *Suc*.

lemma *eq-numeral-Suc* [*simp*]: ‹*numeral k* = *Suc n* ⟷ *pred-numeral k* = *n*›
by (*simp add*: *numeral-eq-Suc*)

lemma *Suc-eq-numeral* [*simp*]: ‹*Suc n* = *numeral k* ⟷ *n* = *pred-numeral k*›
by (*simp add*: *numeral-eq-Suc*)

lemma *less-numeral-Suc* [*simp*]: ‹*numeral k < Suc n ⟷ pred-numeral k < n*›
by (*simp add*: *numeral-eq-Suc*)

lemma *less-Suc-numeral* [*simp*]: ‹*Suc n < numeral k ⟷ n < pred-numeral k*›
by (*simp add*: *numeral-eq-Suc*)

lemma *le-numeral-Suc* [*simp*]: ‹*numeral k ≤ Suc n ⟷ pred-numeral k ≤ n*›
by (*simp add*: *numeral-eq-Suc*)

lemma *le-Suc-numeral* [*simp*]: ‹*Suc n ≤ numeral k ⟷ n ≤ pred-numeral k*›
by (*simp add*: *numeral-eq-Suc*)

lemma *diff-Suc-numeral* [*simp*]: ‹*Suc n − numeral k = n − pred-numeral k*›
by (*simp add*: *numeral-eq-Suc*)

lemma *diff-numeral-Suc* [*simp*]: ‹*numeral k − Suc n = pred-numeral k − n*›
by (*simp add*: *numeral-eq-Suc*)

lemma *max-Suc-numeral* [*simp*]: ‹*max* (*Suc n*) (*numeral k*) = *Suc* (*max n* (*pred-numeral k*))›
by (*simp add*: *numeral-eq-Suc*)

lemma *max-numeral-Suc* [*simp*]: ‹*max* (*numeral k*) (*Suc n*) = *Suc* (*max* (*pred-numeral k*) *n*)›
by (*simp add*: *numeral-eq-Suc*)

lemma *min-Suc-numeral* [*simp*]: ‹*min* (*Suc n*) (*numeral k*) = *Suc* (*min n* (*pred-numeral k*))›
by (*simp add*: *numeral-eq-Suc*)

lemma *min-numeral-Suc* [*simp*]: ‹*min* (*numeral k*) (*Suc n*) = *Suc* (*min* (*pred-numeral k*) *n*)›
by (*simp add*: *numeral-eq-Suc*)

For *case-nat* and *rec-nat*.

lemma *case-nat-numeral* [*simp*]: ‹*case-nat a f* (*numeral v*) = (*let pv* = *pred-numeral v in f pv*)›
by (*simp add*: *numeral-eq-Suc*)

lemma *case-nat-add-eq-if* [*simp*]:
‹*case-nat a f* ((*numeral v*) + *n*) = (*let pv* = *pred-numeral v in f* (*pv* + *n*))›
by (*simp add*: *numeral-eq-Suc*)

lemma *rec-nat-numeral* [*simp*]:
‹*rec-nat a f* (*numeral v*) = (*let pv* = *pred-numeral v in f pv* (*rec-nat a f pv*))›
by (*simp add*: *numeral-eq-Suc Let-def*)

lemma *rec-nat-add-eq-if* [*simp*]:

‹*rec-nat a f* (*numeral v* + *n*) = (*let pv* = *pred-numeral v in f* (*pv* + *n*) (*rec-nat a f* (*pv* + *n*)))›
by (*simp add*: *numeral-eq-Suc Let-def*)

Case analysis on $n < (2::'a)$.

lemma *less-2-cases*: ‹*n* < *2* ⟹ *n* = *0* ∨ *n* = *Suc 0*›
by (*auto simp add*: *numeral-2-eq-2*)

lemma *less-2-cases-iff*: ‹*n* < *2* ⟷ *n* = *0* ∨ *n* = *Suc 0*›
by (*auto simp add*: *numeral-2-eq-2*)

Removal of Small Numerals: 0, 1 and (in additive positions) 2.

bh: Are these rules really a good idea? LCP: well, it already happens for 0 and 1!

lemma *add-2-eq-Suc* [*simp*]: ‹*2* + *n* = *Suc* (*Suc n*)›
by *simp*

lemma *add-2-eq-Suc'* [*simp*]: ‹*n* + *2* = *Suc* (*Suc n*)›
by *simp*

Can be used to eliminate long strings of Sucs, but not by default.

lemma *Suc3-eq-add-3*: ‹*Suc* (*Suc* (*Suc n*)) = *3* + *n*›
by *simp*

lemmas *nat-1-add-1* = *one-add-one* [**where** *'a*=*nat*]

context *semiring-numeral*
begin

lemma *numeral-add-unfold-funpow*:
‹*numeral k* + *a* = ((+) *1* ^^ *numeral k*) *a*›
proof (*rule sym*, *induction k arbitrary*: *a*)
case *One*
then show *?case*
by (*simp add*: *Num.numeral-One numeral-One*)
next
case (*Bit0 k*)
then show *?case*
by (*simp add*: *Num.numeral-Bit0 numeral-Bit0 ac-simps funpow-add*)
next
case (*Bit1 k*)
then show *?case*
by (*simp add*: *Num.numeral-Bit1 numeral-Bit1 ac-simps funpow-add*)
qed

end

context *semiring-1*

begin

lemma *numeral-unfold-funpow*:
‹*numeral k* = ((+) *1* ^^ *numeral k*) *0*›
using *numeral-add-unfold-funpow* [*of k 0*] **by** *simp*

end

context
includes *lifting-syntax*
begin

lemma *transfer-rule-numeral*:
‹((=) ===> *R*) *numeral numeral*›
if [*transfer-rule*]: ‹*R 0 0*› ‹*R 1 1*›
‹(*R* ===> *R* ===> *R*) (+) (+)›
for *R* :: ‹'*a*::{*semiring-numeral*,*monoid-add*} ⇒ '*b*::{*semiring-numeral*,*monoid-add*}
⇒ *bool*›
proof −
have ‹((=) ===> *R*) (λ*k*. ((+) *1* ^^ *numeral k*) *0*) (λ*k*. ((+) *1* ^^ *numeral k*)
0)›
by *transfer-prover*
moreover have ‹*numeral* = (λ*k*. ((+) (*1*::'*a*) ^^ *numeral k*) *0*)›
using *numeral-add-unfold-funpow* [**where** *?'a* = '*a*, *of* - *0*]
by (*simp add*: *fun-eq-iff*)
moreover have ‹*numeral* = (λ*k*. ((+) (*1*::'*b*) ^^ *numeral k*) *0*)›
using *numeral-add-unfold-funpow* [**where** *?'a* = '*b*, *of* - *0*]
by (*simp add*: *fun-eq-iff*)
ultimately show *?thesis*
by *simp*
qed

end

46.5 Particular lemmas concerning *2*::'*a*

context *linordered-field*
begin

subclass *field-char-0* **..**

lemma *half-gt-zero-iff*: ‹*0* < *a* / *2* ⟷ *0* < *a*›
by (*auto simp add*: *field-simps*)

lemma *half-gt-zero* [*simp*]: ‹*0* < *a* ⟹ *0* < *a* / *2*›
by (*simp add*: *half-gt-zero-iff*)

end

46.6 Numeral equations as default simplification rules

declare (**in** *numeral*) *numeral-One* [*simp*]
declare (**in** *numeral*) *numeral-plus-numeral* [*simp*]
declare (**in** *numeral*) *add-numeral-special* [*simp*]
declare (**in** *neg-numeral*) *add-neg-numeral-simps* [*simp*]
declare (**in** *neg-numeral*) *add-neg-numeral-special* [*simp*]
declare (**in** *neg-numeral*) *diff-numeral-simps* [*simp*]
declare (**in** *neg-numeral*) *diff-numeral-special* [*simp*]
declare (**in** *semiring-numeral*) *numeral-times-numeral* [*simp*]
declare (**in** *ring-1*) *mult-neg-numeral-simps* [*simp*]

46.6.1 Special Simplification for Constants

These distributive laws move literals inside sums and differences.

lemmas *distrib-right-numeral* [*simp*] = *distrib-right* [*of* - - ‹*numeral v*›] **for** *v*
lemmas *distrib-left-numeral* [*simp*] = *distrib-left* [*of* ‹*numeral v*›] **for** *v*
lemmas *left-diff-distrib-numeral* [*simp*] = *left-diff-distrib* [*of* - - ‹*numeral v*›] **for** *v*
lemmas *right-diff-distrib-numeral* [*simp*] = *right-diff-distrib* [*of* ‹*numeral v*›] **for** *v*

These are actually for fields, like real

lemmas *zero-less-divide-iff-numeral* [*simp*, *no-atp*] = *zero-less-divide-iff* [*of* ‹*numeral w*›] **for** *w*
lemmas *divide-less-0-iff-numeral* [*simp*, *no-atp*] = *divide-less-0-iff* [*of* ‹*numeral w*›] **for** *w*
lemmas *zero-le-divide-iff-numeral* [*simp*, *no-atp*] = *zero-le-divide-iff* [*of* ‹*numeral w*›] **for** *w*
lemmas *divide-le-0-iff-numeral* [*simp*, *no-atp*] = *divide-le-0-iff* [*of* ‹*numeral w*›] **for** *w*

Replaces *inverse #nn* by *1/#nn*. It looks strange, but then other simprocs simplify the quotient.

lemmas *inverse-eq-divide-numeral* [*simp*] =
inverse-eq-divide [*of* ‹*numeral w*›] **for** *w*

lemmas *inverse-eq-divide-neg-numeral* [*simp*] =
inverse-eq-divide [*of* ‹− *numeral w*›] **for** *w*

These laws simplify inequalities, moving unary minus from a term into the literal.

lemmas *equation-minus-iff-numeral* [*no-atp*] =
equation-minus-iff [*of* ‹*numeral v*›] **for** *v*

lemmas *minus-equation-iff-numeral* [*no-atp*] =
minus-equation-iff [*of* - ‹*numeral v*›] **for** *v*

lemmas *le-minus-iff-numeral* [*no-atp*] =

le-minus-iff [*of* ‹*numeral v*›] **for** *v*

lemmas *minus-le-iff-numeral* [*no-atp*] =
minus-le-iff [*of* - ‹*numeral v*›] **for** *v*

lemmas *less-minus-iff-numeral* [*no-atp*] =
less-minus-iff [*of* ‹*numeral v*›] **for** *v*

lemmas *minus-less-iff-numeral* [*no-atp*] =
minus-less-iff [*of* - ‹*numeral v*›] **for** *v*

Cancellation of constant factors in comparisons (< and ≤)

lemmas *mult-less-cancel-left-numeral* [*simp*, *no-atp*] = *mult-less-cancel-left* [*of* ‹*numeral v*›] **for** *v*
lemmas *mult-less-cancel-right-numeral* [*simp*, *no-atp*] = *mult-less-cancel-right* [*of* - ‹*numeral v*›] **for** *v*
lemmas *mult-le-cancel-left-numeral* [*simp*, *no-atp*] = *mult-le-cancel-left* [*of* ‹*numeral v*›] **for** *v*
lemmas *mult-le-cancel-right-numeral* [*simp*, *no-atp*] = *mult-le-cancel-right* [*of* - ‹*numeral v*›] **for** *v*

Multiplying out constant divisors in comparisons (<, ≤ and =)

named-theorems *divide-const-simps* ‹*simplification rules to simplify comparisons involving constant divisors*›

lemmas *le-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-le-divide-eq [*of* ‹*numeral w*›, *OF zero-less-numeral*]
neg-le-divide-eq [*of* ‹− *numeral w*›, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-le-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-divide-le-eq [*of* ‹*numeral w*›, *OF zero-less-numeral*]
neg-divide-le-eq [*of* ‹− *numeral w*›, *OF neg-numeral-less-zero*] **for** *w*

lemmas *less-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-less-divide-eq [*of* ‹*numeral w*›, *OF zero-less-numeral*]
neg-less-divide-eq [*of* ‹− *numeral w*›, *OF neg-numeral-less-zero*] **for** *w*

lemmas *divide-less-eq-numeral1* [*simp*,*divide-const-simps*] =
pos-divide-less-eq [*of* ‹*numeral w*›, *OF zero-less-numeral*]
neg-divide-less-eq [*of* ‹− *numeral w*›, *OF neg-numeral-less-zero*] **for** *w*

lemmas *eq-divide-eq-numeral1* [*simp*,*divide-const-simps*] =
eq-divide-eq [*of* - - ‹*numeral w*›]
eq-divide-eq [*of* - - ‹− *numeral w*›] **for** *w*

lemmas *divide-eq-eq-numeral1* [*simp*,*divide-const-simps*] =
divide-eq-eq [*of* - ‹*numeral w*›]
divide-eq-eq [*of* - ‹− *numeral w*›] **for** *w*

46.6.2 Optional Simplification Rules Involving Constants

Simplify quotients that are compared with a literal constant.

lemmas *le-divide-eq-numeral* [*divide-const-simps*] =
le-divide-eq [*of* ‹*numeral w*›]
le-divide-eq [*of* ‹− *numeral w*›] **for** *w*

lemmas *divide-le-eq-numeral* [*divide-const-simps*] =
divide-le-eq [*of* - - ‹*numeral w*›]
divide-le-eq [*of* - - ‹− *numeral w*›] **for** *w*

lemmas *less-divide-eq-numeral* [*divide-const-simps*] =
less-divide-eq [*of* ‹*numeral w*›]
less-divide-eq [*of* ‹− *numeral w*›] **for** *w*

lemmas *divide-less-eq-numeral* [*divide-const-simps*] =
divide-less-eq [*of* - - ‹*numeral w*›]
divide-less-eq [*of* - - ‹− *numeral w*›] **for** *w*

lemmas *eq-divide-eq-numeral* [*divide-const-simps*] =
eq-divide-eq [*of* ‹*numeral w*›]
eq-divide-eq [*of* ‹− *numeral w*›] **for** *w*

lemmas *divide-eq-eq-numeral* [*divide-const-simps*] =
divide-eq-eq [*of* - - ‹*numeral w*›]
divide-eq-eq [*of* - - ‹− *numeral w*›] **for** *w*

Not good as automatic simprules because they cause case splits.

lemmas [*divide-const-simps*] =
le-divide-eq-1 divide-le-eq-1 less-divide-eq-1 divide-less-eq-1

46.7 Setting up simprocs

lemma *mult-numeral-1*: ‹*Numeral1* ∗ *a* = *a*›
for *a* :: ‹'*a*::*semiring-numeral*›
by *simp*

lemma *mult-numeral-1-right*: ‹*a* ∗ *Numeral1* = *a*›
for *a* :: ‹'*a*::*semiring-numeral*›
by *simp*

lemma *divide-numeral-1*: ‹*a* / *Numeral1* = *a*›
for *a* :: ‹'*a*::*field*›
by *simp*

lemma *inverse-numeral-1*: ‹*inverse Numeral1* = (*Numeral1*::'*a*::*division-ring*)›
by *simp*

Theorem lists for the cancellation simprocs. The use of a binary numeral

for 1 reduces the number of special cases.

lemma *mult-1s-semiring-numeral*:
‹*Numeral1 * a = a*›
‹*a * Numeral1 = a*›
for *a* :: ‹*'a*::*semiring-numeral*›
by *simp-all*

lemma *mult-1s-ring-1*:
‹*- Numeral1 * b = - b*›
‹*b * - Numeral1 = - b*›
for *b* :: ‹*'a*::*ring-1*›
by *simp-all*

lemmas *mult-1s = mult-1s-semiring-numeral mult-1s-ring-1*

setup ‹
Reorient-Proc.add
(*fn Const* (***const-name*** ‹*numeral*›, -) $ - => *true*
| *Const* (***const-name*** ‹*uminus*›, -) $ (*Const* (***const-name*** ‹*numeral*›, -) $ -)
=> *true*
| - => *false*)
›

simproc-setup *reorient-numeral* (‹*numeral w = x*› | ‹*- numeral w = y*›) =
‹*K Reorient-Proc.proc*›

46.7.1 Simplification of arithmetic operations on integer constants

lemmas *arith-special* =
add-numeral-special add-neg-numeral-special
diff-numeral-special

lemmas *arith-extra-simps* =
numeral-plus-numeral add-neg-numeral-simps add-0-left add-0-right
minus-zero
diff-numeral-simps diff-0 diff-0-right
numeral-times-numeral mult-neg-numeral-simps
mult-zero-left mult-zero-right
abs-numeral abs-neg-numeral

For making a minimal simpset, one must include these default simprules. Also include *simp-thms*.

lemmas *arith-simps* =
add-num-simps mult-num-simps sub-num-simps
BitM.simps dbl-simps dbl-inc-simps dbl-dec-simps
abs-zero abs-one arith-extra-simps

lemmas *more-arith-simps* =

neg-le-iff-le
minus-zero left-minus right-minus
mult-1-left mult-1-right
mult-minus-left mult-minus-right
minus-add-distrib minus-minus mult.assoc

lemmas *of-nat-simps* =
of-nat-0 of-nat-1 of-nat-Suc of-nat-add of-nat-mult

Simplification of relational operations.

lemmas *eq-numeral-extra* =
zero-neq-one one-neq-zero

lemmas *rel-simps* =
le-num-simps less-num-simps eq-num-simps
le-numeral-simps le-neg-numeral-simps le-minus-one-simps le-numeral-extra
less-numeral-simps less-neg-numeral-simps less-minus-one-simps less-numeral-extra
eq-numeral-simps eq-neg-numeral-simps eq-numeral-extra

lemma *Let-numeral* [*simp*]: ‹*Let* (*numeral v*) *f* = *f* (*numeral v*)›
— Unfold all *let*s involving constants
unfolding *Let-def* **..**

lemma *Let-neg-numeral* [*simp*]: ‹*Let* (− *numeral v*) *f* = *f* (− *numeral v*)›
— Unfold all *let*s involving constants
unfolding *Let-def* **..**

declaration ‹
let
fun number-of ctxt T n =
if not (*Sign.of-sort* (*Proof-Context.theory-of ctxt*) (*T*, ***sort***‹*numeral*›))
then raise CTERM (*number-of*, [])
else Numeral.mk-cnumber (*Thm.ctyp-of ctxt T*) *n*;
in
K (
Lin-Arith.set-number-of number-of
#> *Lin-Arith.add-simps*
@{*thms arith-simps more-arith-simps rel-simps pred-numeral-simps*
arith-special numeral-One of-nat-simps uminus-numeral-One
Suc-numeral Let-numeral Let-neg-numeral Let-0 Let-1
le-Suc-numeral le-numeral-Suc less-Suc-numeral less-numeral-Suc
Suc-eq-numeral eq-numeral-Suc mult-Suc mult-Suc-right of-nat-numeral})
end
›

46.7.2 Simplification of arithmetic when nested to the right

lemma *add-numeral-left* [*simp*]: ‹*numeral v* + (*numeral w* + *z*) = (*numeral*(*v* + *w*) + *z*)›

by (*simp-all add*: *add.assoc* [*symmetric*])

lemma *add-neg-numeral-left* [*simp*]:
‹*numeral v* + (− *numeral w* + *y*) = (*sub v w* + *y*)›
‹− *numeral v* + (*numeral w* + *y*) = (*sub w v* + *y*)›
‹− *numeral v* + (− *numeral w* + *y*) = (− *numeral*(*v* + *w*) + *y*)›
by (*simp-all add*: *add.assoc* [*symmetric*])

lemma *mult-numeral-left-semiring-numeral*:
‹*numeral v* ∗ (*numeral w* ∗ *z*) = (*numeral*(*v* ∗ *w*) ∗ *z* :: *'a*::*semiring-numeral*)›
by (*simp add*: *mult.assoc* [*symmetric*])

lemma *mult-numeral-left-ring-1*:
‹− *numeral v* ∗ (*numeral w* ∗ *y*) = (− *numeral*(*v* ∗ *w*) ∗ *y* :: *'a*::*ring-1*)›
‹*numeral v* ∗ (− *numeral w* ∗ *y*) = (− *numeral*(*v* ∗ *w*) ∗ *y* :: *'a*::*ring-1*)›
‹− *numeral v* ∗ (− *numeral w* ∗ *y*) = (*numeral*(*v* ∗ *w*) ∗ *y* :: *'a*::*ring-1*)›
by (*simp-all add*: *mult.assoc* [*symmetric*])

lemmas *mult-numeral-left* [*simp*] =
mult-numeral-left-semiring-numeral
mult-numeral-left-ring-1

46.8 Code module namespace

code-identifier
code-module *Num* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

46.9 Printing of evaluated natural numbers as numerals

lemma [*code-post*]:
‹*Suc 0* = *1*›
‹*Suc 1* = *2*›
‹*Suc* (*numeral n*) = *numeral* (*inc n*)›
by (*simp-all add*: *numeral-inc*)

lemmas [*code-post*] = *inc.simps*

46.10 More on auxiliary conversion

context *semiring-1*
begin

lemma *num-of-nat-numeral-eq* [*simp*]:
‹*num-of-nat* (*numeral q*) = *q*›
by (*simp flip*: *nat-of-num-numeral add*: *nat-of-num-inverse*)

lemma *numeral-num-of-nat-unfold*:
‹*numeral* (*num-of-nat n*) = (*if n* = *0 then 1 else of-nat n*)›
apply (*simp only*: *of-nat-numeral* [*symmetric*, *of* ‹*num-of-nat n*›] *flip*: *nat-of-num-numeral*)
apply (*auto simp add*: *num-of-nat-inverse*)

done

end

hide-const (**open**) *One Bit0 Bit1 BitM inc pow sqr sub dbl dbl-inc dbl-dec*

end

47 Exponentiation

theory *Power*
imports *Num*
begin

47.1 Powers for Arbitrary Monoids

class *power* = *one* + *times*
begin

primrec *power* :: *'a* ⇒ *nat* ⇒ *'a* (**infixr** ‹^› *80*)
where
power-0: *a* ^ *0* = *1*
| *power-Suc*: *a* ^ *Suc n* = *a* ∗ *a* ^ *n*

notation (*latex* **output**)
power (‹(-⁻)› [*1000*] *1000*)

Special syntax for squares.

abbreviation *power2* :: *'a* ⇒ *'a* (‹(‹*notation*=‹*postfix 2*››-²)› [*1000*] *999*)
where $x^2 \equiv x \mathbin{\hat{}} 2$

end

context
includes *lifting-syntax*
begin

lemma *power-transfer* [*transfer-rule*]:
‹(*R* ===> (=) ===> *R*) (^) (^)›
if [*transfer-rule*]: ‹*R 1 1*›
‹(*R* ===> *R* ===> *R*) (∗) (∗)›
for *R* :: ‹*'a*::*power* ⇒ *'b*::*power* ⇒ *bool*›
by (*simp only*: *power-def* [*abs-def*]) *transfer-prover*

end

context *monoid-mult*
begin

subclass *power* **.**

lemma *power-one* [*simp*]: *1 ^ n = 1*
by (*induct n*) *simp-all*

lemma *power-one-right* [*simp*]: *a ^ 1 = a*
by *simp*

lemma *power-Suc0-right* [*simp*]: *a ^ Suc 0 = a*
by *simp*

lemma *power-commutes*: *a ^ n * a = a * a ^ n*
by (*induct n*) (*simp-all add*: *mult.assoc*)

lemma *power-Suc2*: *a ^ Suc n = a ^ n * a*
by (*simp add*: *power-commutes*)

lemma *power-add*: *a ^ (m + n) = a ^ m * a ^ n*
by (*induct m*) (*simp-all add*: *algebra-simps*)

lemma *power-mult*: *a ^ (m * n) = (a ^ m) ^ n*
by (*induct n*) (*simp-all add*: *power-add*)

lemma *power-even-eq*: *a ^ (2 * n) = (a ^ n)²*
by (*subst mult.commute*) (*simp add*: *power-mult*)

lemma *power-odd-eq*: *a ^ Suc (2*n) = a * (a ^ n)²*
by (*simp add*: *power-even-eq*)

lemma *power-numeral-even*: *z ^ numeral (Num.Bit0 w) = (let w = z ^ (numeral w) in w * w)*
by (*simp only*: *numeral-Bit0 power-add Let-def*)

lemma *power-numeral-odd*: *z ^ numeral (Num.Bit1 w) = (let w = z ^ (numeral w) in z * w * w)*
by (*simp only*: *numeral-Bit1 One-nat-def add-Suc-right add-0-right*
power-Suc power-add Let-def mult.assoc)

lemma *power2-eq-square*: *a² = a * a*
by (*simp add*: *numeral-2-eq-2*)

lemma *power3-eq-cube*: *a ^ 3 = a * a * a*
by (*simp add*: *numeral-3-eq-3 mult.assoc*)

lemma *power4-eq-xxxx*: *x^4 = x * x * x * x*
by (*simp add*: *mult.assoc power-numeral-even*)

lemma *power-numeral-reduce*: *x ^ numeral n = x * x ^ pred-numeral n*

by (*simp add*: *numeral-eq-Suc*)

lemma *funpow-times-power*: (*times x* ^^ *f x*) = *times* (*x* ^ *f x*)
proof (*induct f x arbitrary*: *f*)
case *0*
then show *?case* **by** (*simp add*: *fun-eq-iff*)
next
case (*Suc n*)
define *g* **where** *g x* = *f x* − *1* **for** *x*
with *Suc* **have** *n* = *g x* **by** *simp*
with *Suc* **have** *times x* ^^ *g x* = *times* (*x* ^ *g x*) **by** *simp*
moreover from *Suc g-def* **have** *f x* = *g x* + *1* **by** *simp*
ultimately show *?case*
by (*simp add*: *power-add funpow-add fun-eq-iff mult.assoc*)
qed

lemma *power-commuting-commutes*:
assumes *x* ∗ *y* = *y* ∗ *x*
shows *x* ^ *n* ∗ *y* = *y* ∗ *x* ^*n*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have *x* ^ *Suc n* ∗ *y* = *x* ^ *n* ∗ *y* ∗ *x*
by (*subst power-Suc2*) (*simp add*: *assms ac-simps*)
also have ... = *y* ∗ *x* ^ *Suc n*
by (*simp only*: *Suc power-Suc2*) (*simp add*: *ac-simps*)
finally show *?case* **.**
qed

lemma *power-minus-mult*: *0* < *n* ⟹ *a* ^ (*n* − *1*) ∗ *a* = *a* ^ *n*
by (*simp add*: *power-commutes split*: *nat-diff-split*)

lemma *left-right-inverse-power*:
assumes *x* ∗ *y* = *1*
shows *x* ^ *n* ∗ *y* ^ *n* = *1*
proof (*induct n*)
case (*Suc n*)
moreover have *x* ^ *Suc n* ∗ *y* ^ *Suc n* = *x*^*n* ∗ (*x* ∗ *y*) ∗ *y*^*n*
by (*simp add*: *power-Suc2*[*symmetric*] *mult.assoc*[*symmetric*])
ultimately show *?case* **by** (*simp add*: *assms*)
qed *simp*

end

context *comm-monoid-mult*
begin

lemma *power-mult-distrib* [*algebra-simps*, *algebra-split-simps*, *field-simps*, *field-split-simps*, *divide-simps*]:
  $(a * b) \hat{} n = (a \hat{} n) * (b \hat{} n)$
  by (*induction n*) (*simp-all add*: *ac-simps*)

end

Extract constant factors from powers.

declare *power-mult-distrib* [**where** *a* = *numeral w* **for** *w*, *simp*]
declare *power-mult-distrib* [**where** *b* = *numeral w* **for** *w*, *simp*]

lemma *power-add-numeral* [*simp*]: $a\hat{}numeral\ m * a\hat{}numeral\ n = a\hat{}numeral\ (m + n)$
  for *a* :: *'a*::*monoid-mult*
  by (*simp add*: *power-add* [*symmetric*])

lemma *power-add-numeral2* [*simp*]: $a\hat{}numeral\ m * (a\hat{}numeral\ n * b) = a\hat{}numeral\ (m + n) * b$
  for *a* :: *'a*::*monoid-mult*
  by (*simp add*: *mult.assoc* [*symmetric*])

lemma *power-mult-numeral* [*simp*]: $(a\hat{}numeral\ m)\hat{}numeral\ n = a\hat{}numeral\ (m * n)$
  for *a* :: *'a*::*monoid-mult*
  by (*simp only*: *numeral-mult power-mult*)

context *semiring-numeral*
begin

lemma *numeral-sqr*: *numeral* (*Num.sqr k*) = *numeral k* * *numeral k*
  by (*simp only*: *sqr-conv-mult numeral-mult*)

lemma *numeral-pow*: *numeral* (*Num.pow k l*) = *numeral k* $\hat{}$ *numeral l*
  by (*induct l*)
    (*simp-all only*: *numeral-class.numeral.simps pow.simps*
      numeral-sqr numeral-mult power-add power-one-right)

lemma *power-numeral* [*simp*]: *numeral k* $\hat{}$ *numeral l* = *numeral* (*Num.pow k l*)
  by (*rule numeral-pow* [*symmetric*])

end

context *semiring-1*
begin

lemma *of-nat-power* [*simp*]: *of-nat* (*m* $\hat{}$ *n*) = *of-nat m* $\hat{}$ *n*
  by (*induct n*) *simp-all*

lemma *zero-power*: $0 < n \Longrightarrow 0 \hat{} n = 0$

by (*cases n*) *simp-all*

lemma *power-zero-numeral* [*simp*]: *0 ^ numeral k = 0*
by (*simp add*: *numeral-eq-Suc*)

lemma *zero-power2*: $0^2 = 0$
by (*rule power-zero-numeral*)

lemma *one-power2*: $1^2 = 1$
by (*rule power-one*)

lemma *power-0-Suc* [*simp*]: *0 ^ Suc n = 0*
by *simp*

It looks plausible as a simprule, but its effect can be strange.

lemma *power-0-left*: *0 ^ n = (if n = 0 then 1 else 0)*
by (*cases n*) *simp-all*

end

context *semiring-char-0* **begin**

lemma *numeral-power-eq-of-nat-cancel-iff* [*simp*]:
numeral x ^ n = of-nat y ⟷ numeral x ^ n = y
using *of-nat-eq-iff* **by** *fastforce*

lemma *real-of-nat-eq-numeral-power-cancel-iff* [*simp*]:
of-nat y = numeral x ^ n ⟷ y = numeral x ^ n
using *numeral-power-eq-of-nat-cancel-iff* [*of x n y*] **by** (*metis* (*mono-tags*))

lemma *of-nat-eq-of-nat-power-cancel-iff*[*simp*]: (*of-nat b*) *^ w = of-nat x ⟷ b ^ w = x*
by (*metis of-nat-power of-nat-eq-iff*)

lemma *of-nat-power-eq-of-nat-cancel-iff*[*simp*]: *of-nat x* = (*of-nat b*) *^ w ⟷ x = b ^ w*
by (*metis of-nat-eq-of-nat-power-cancel-iff*)

end

context *comm-semiring-1*
begin

The divides relation.

lemma *le-imp-power-dvd*:
assumes $m \leq n$
shows *a ^ m dvd a ^ n*
proof
from *assms* **have** *a ^ n = a ^ (m + (n − m))* **by** *simp*

also have ... $= a \mathbin{\hat{}} m * a \mathbin{\hat{}} (n - m)$ **by** (*rule power-add*)
finally show $a \mathbin{\hat{}} n = a \mathbin{\hat{}} m * a \mathbin{\hat{}} (n - m)$ **.**
qed

lemma *power-le-dvd*: $a \mathbin{\hat{}} n$ *dvd* $b \Longrightarrow m \leq n \Longrightarrow a \mathbin{\hat{}} m$ *dvd* b
by (*rule dvd-trans* [*OF le-imp-power-dvd*])

lemma *dvd-power-same*: x *dvd* $y \Longrightarrow x \mathbin{\hat{}} n$ *dvd* $y \mathbin{\hat{}} n$
by (*induct n*) (*auto simp add*: *mult-dvd-mono*)

lemma *dvd-power-le*: x *dvd* $y \Longrightarrow m \geq n \Longrightarrow x \mathbin{\hat{}} n$ *dvd* $y \mathbin{\hat{}} m$
by (*rule power-le-dvd* [*OF dvd-power-same*])

lemma *dvd-power* [*simp*]:
fixes n :: *nat*
assumes $n > 0 \vee x = 1$
shows x *dvd* $(x \mathbin{\hat{}} n)$
using *assms*
proof
assume $0 < n$
then have $x \mathbin{\hat{}} n = x \mathbin{\hat{}} Suc\ (n - 1)$ **by** *simp*
then show x *dvd* $(x \mathbin{\hat{}} n)$ **by** *simp*
next
assume $x = 1$
then show x *dvd* $(x \mathbin{\hat{}} n)$ **by** *simp*
qed

end

context *semiring-1-no-zero-divisors*
begin

subclass *power* **.**

lemma *power-eq-0-iff* [*simp*]: $a \mathbin{\hat{}} n = 0 \longleftrightarrow a = 0 \wedge n > 0$
by (*induct n*) *auto*

lemma *power-not-zero*: $a \neq 0 \Longrightarrow a \mathbin{\hat{}} n \neq 0$
by (*induct n*) *auto*

lemma *zero-eq-power2* [*simp*]: $a^2 = 0 \longleftrightarrow a = 0$
unfolding *power2-eq-square* **by** *simp*

end

context *ring-1*
begin

lemma *power-minus*: $(-a) \mathbin{\hat{}} n = (-1) \mathbin{\hat{}} n * a \mathbin{\hat{}} n$

proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
by (*simp del*: *power-Suc add*: *power-Suc2 mult.assoc*)
qed

lemma *power-minus'*: *NO-MATCH 1 x* $\Longrightarrow (-x) \mathbin{\hat{}} n = (-1) \mathbin{\hat{}} n * x \mathbin{\hat{}} n$
by (*rule power-minus*)

lemma *power-minus-Bit0*: $(-x) \mathbin{\hat{}} \mathit{numeral}\ (\mathit{Num.Bit0}\ k) = x \mathbin{\hat{}} \mathit{numeral}\ (\mathit{Num.Bit0}\ k)$
by (*induct k*, *simp-all only*: *numeral-class.numeral.simps power-add power-one-right mult-minus-left mult-minus-right minus-minus*)

lemma *power-minus-Bit1*: $(-x) \mathbin{\hat{}} \mathit{numeral}\ (\mathit{Num.Bit1}\ k) = -(x \mathbin{\hat{}} \mathit{numeral}\ (\mathit{Num.Bit1}\ k))$
by (*simp only*: *eval-nat-numeral*(*3*) *power-Suc power-minus-Bit0 mult-minus-left*)

lemma *power2-minus* [*simp*]: $(-a)^2 = a^2$
by (*fact power-minus-Bit0*)

lemma *power-minus1-even* [*simp*]: $(-1) \mathbin{\hat{}} (2*n) = 1$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case* **by** (*simp add*: *power-add power2-eq-square*)
qed

lemma *power-minus1-odd*: $(-1) \mathbin{\hat{}} \mathit{Suc}\ (2*n) = -1$
by *simp*

lemma *power-minus-even* [*simp*]: $(-a) \mathbin{\hat{}} (2*n) = a \mathbin{\hat{}} (2*n)$
by (*simp add*: *power-minus* [*of a*])

end

context *ring-1-no-zero-divisors*
begin

lemma *power2-eq-1-iff*: $a^2 = 1 \longleftrightarrow a = 1 \vee a = -1$
using *square-eq-1-iff* [*of a*] **by** (*simp add*: *power2-eq-square*)

end

context *idom*
begin

lemma *power2-eq-iff*: $x^2 = y^2 \longleftrightarrow x = y \vee x = - y$
unfolding *power2-eq-square* **by** (*rule square-eq-iff*)

end

context *semidom-divide*
begin

lemma *power-diff*:
$a \mathbin{\hat{}} (m - n) = (a \mathbin{\hat{}} m)\ div\ (a \mathbin{\hat{}} n)$ **if** $a \neq 0$ **and** $n \leq m$
proof −
define q **where** $q = m - n$
with ‹$n \leq m$› **have** $m = q + n$ **by** *simp*
with ‹$a \neq 0$› *q-def* **show** *?thesis*
by (*simp add*: *power-add*)
qed

lemma *power-diff-if*:
$a \mathbin{\hat{}} (m - n) = (if\ n \leq m\ then\ (a \mathbin{\hat{}} m)\ div\ (a \mathbin{\hat{}} n)\ else\ 1)$ **if** $a \neq 0$
by (*simp add*: *power-diff that*)

end

context *algebraic-semidom*
begin

lemma *div-power*: *b dvd a* $\Longrightarrow$ $(a\ div\ b) \mathbin{\hat{}} n = a \mathbin{\hat{}} n\ div\ b \mathbin{\hat{}} n$
by (*induct n*) (*simp-all add*: *div-mult-div-if-dvd dvd-power-same*)

lemma *is-unit-power-iff*: *is-unit* $(a \mathbin{\hat{}} n) \longleftrightarrow$ *is-unit a* $\vee$ $n = 0$
by (*induct n*) (*auto simp add*: *is-unit-mult-iff*)

lemma *dvd-power-iff*:
assumes $x \neq 0$
shows $x \mathbin{\hat{}} m\ dvd\ x \mathbin{\hat{}} n \longleftrightarrow$ *is-unit x* $\vee$ $m \leq n$
proof
assume $*$: $x \mathbin{\hat{}} m\ dvd\ x \mathbin{\hat{}} n$
{
assume $m > n$
note $*$
also have $x \mathbin{\hat{}} n = x \mathbin{\hat{}} n * 1$ **by** *simp*
also from ‹$m > n$› **have** $m = n + (m - n)$ **by** *simp*
also have $x \mathbin{\hat{}} \ldots = x \mathbin{\hat{}} n * x \mathbin{\hat{}} (m - n)$ **by** (*rule power-add*)
finally have $x \mathbin{\hat{}} (m - n)\ dvd\ 1$
using *assms* **by** (*subst* (*asm*) *dvd-times-left-cancel-iff*) *simp-all*
with ‹$m > n$› **have** *is-unit x* **by** (*simp add*: *is-unit-power-iff*)

}
thus *is-unit x* ∨ *m* ≤ *n* **by** *force*
qed (*auto intro*: *unit-imp-dvd simp*: *is-unit-power-iff le-imp-power-dvd*)

end

context *normalization-semidom-multiplicative*
begin

lemma *normalize-power*: *normalize* (*a* ^ *n*) = *normalize a* ^ *n*
by (*induct n*) (*simp-all add*: *normalize-mult*)

lemma *unit-factor-power*: *unit-factor* (*a* ^ *n*) = *unit-factor a* ^ *n*
by (*induct n*) (*simp-all add*: *unit-factor-mult*)

end

context *division-ring*
begin

Perhaps these should be simprules.

lemma *power-inverse* [*field-simps*, *field-split-simps*, *divide-simps*]: *inverse a* ^ *n* = *inverse* (*a* ^ *n*)
proof (*cases a* = *0*)
case *True*
then show *?thesis* **by** (*simp add*: *power-0-left*)
next
case *False*
then have *inverse* (*a* ^ *n*) = *inverse a* ^ *n*
by (*induct n*) (*simp-all add*: *nonzero-inverse-mult-distrib power-commutes*)
then show *?thesis* **by** *simp*
qed

lemma *power-one-over* [*field-simps*, *field-split-simps*, *divide-simps*]: (*1* / *a*) ^ *n* = *1* / *a* ^ *n*
using *power-inverse* [*of a*] **by** (*simp add*: *divide-inverse*)

end

context *field*
begin

lemma *power-divide* [*field-simps*, *field-split-simps*, *divide-simps*]: (*a* / *b*) ^ *n* = *a* ^ *n* / *b* ^ *n*
by (*induct n*) *simp-all*

end

47.2 Exponentiation on ordered types

context *ordered-semiring-1*
begin

lemma *zero-le-power* [*simp*]: $0 \leq a \Longrightarrow 0 \leq a \mathbin{\hat{}} n$
by (*induct n*) *simp-all*

lemma *power-mono*: $a \leq b \Longrightarrow 0 \leq a \Longrightarrow a \mathbin{\hat{}} n \leq b \mathbin{\hat{}} n$
by (*induct n*) (*auto intro*: *mult-mono order-trans* [*of 0 a b*])

lemma *one-le-power* [*simp*]: $1 \leq a \Longrightarrow 1 \leq a \mathbin{\hat{}} n$
using *power-mono* [*of 1 a n*] **by** *simp*

lemma *power-le-one*: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \mathbin{\hat{}} n \leq 1$
using *power-mono* [*of a 1 n*] **by** *simp*

lemma *power-gt1-lemma*:
assumes *gt1*: $1 < a$
shows $1 < a * a \mathbin{\hat{}} n$
proof −
from *gt1* **have** $0 \leq a$
by (*fact order-trans* [*OF zero-le-one less-imp-le*])
from *gt1* **have** $1 * 1 < a * 1$ **by** *simp*
also from *gt1* **have** $\ldots \leq a * a \mathbin{\hat{}} n$
by (*simp only*: *mult-mono* ‹$0 \leq a$› *one-le-power order-less-imp-le zero-le-one order-refl*)
finally show *?thesis* **by** *simp*
qed

lemma *power-gt1*: $1 < a \Longrightarrow 1 < a \mathbin{\hat{}} Suc\ n$
by (*simp add*: *power-gt1-lemma*)

lemma *one-less-power* [*simp*]: $1 < a \Longrightarrow 0 < n \Longrightarrow 1 < a \mathbin{\hat{}} n$
by (*cases n*) (*simp-all add*: *power-gt1-lemma*)

Proof resembles that of *power-strict-decreasing*.

lemma *power-increasing*: $n \leq N \Longrightarrow 1 \leq a \Longrightarrow a \mathbin{\hat{}} n \leq a \mathbin{\hat{}} N$
proof (*induct N*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc N*)
then show *?case*
using *mult-mono*[*of 1 a a* $\mathbin{\hat{}}$ *n a* $\mathbin{\hat{}}$ *N*]
by (*auto simp add*: *le-Suc-eq order-trans* [*OF zero-le-one*])
qed

Proof resembles that of *power-strict-decreasing*.

lemma *power-decreasing*: $n \leq N \Longrightarrow 0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \mathbin{\hat{}} N \leq a \mathbin{\hat{}} n$

proof (*induction N*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc N*)
then show *?case*
using *mult-mono*[*of a 1 a^N a ^ n*]
by (*auto simp add*: *le-Suc-eq*)
qed

lemma *power-Suc-le-self*: $0 \leq a \Longrightarrow a \leq 1 \Longrightarrow a \mathbin{\hat{}} Suc\ n \leq a$
using *power-decreasing* [*of 1 Suc n a*] **by** *simp*

end

context *linordered-semidom*
begin

lemma *zero-less-power* [*simp*]: $0 < a \Longrightarrow 0 < a \mathbin{\hat{}} n$
by (*induct n*) *simp-all*

lemma *power-le-imp-le-exp*:
assumes *gt1*: $1 < a$
shows $a \mathbin{\hat{}} m \leq a \mathbin{\hat{}} n \Longrightarrow m \leq n$
proof (*induct m arbitrary*: *n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc m*)
show *?case*
proof (*cases n*)
case *0*
with *Suc* **have** $a * a \mathbin{\hat{}} m \leq 1$ **by** *simp*
with *gt1* **show** *?thesis*
by (*force simp only*: *power-gt1-lemma not-less* [*symmetric*])
next
case (*Suc n*)
with *Suc.prems Suc.hyps* **show** *?thesis*
by (*force dest*: *mult-left-le-imp-le simp add*: *less-trans* [*OF zero-less-one gt1*])
qed
qed

lemma *of-nat-zero-less-power-iff* [*simp*]: *of-nat x ^ n > 0* $\longleftrightarrow$ $x > 0 \vee n = 0$
by (*induct n*) *auto*

lemma *power-strict-mono*: $a < b \Longrightarrow 0 \leq a \Longrightarrow 0 < n \Longrightarrow a \mathbin{\hat{}} n < b \mathbin{\hat{}} n$
proof (*induct n*)
case *0*

then show *?case* **by** *simp*
next
case (*Suc n*)
then show *?case*
by (*cases n = 0*) (*auto simp*: *mult-strict-mono le-less-trans* [*of 0 a b*])
qed

lemma *power-mono-iff* [*simp*]:
shows ⟦*a ≥ 0*; *b ≥ 0*; *n>0*⟧ ⟹ *a ^ n ≤ b ^ n ⟷ a ≤ b*
using *power-mono* [*of a b*] *power-strict-mono* [*of b a*] *not-le* **by** *auto*

Lemma for *power-strict-decreasing*

lemma *power-Suc-less*: *0 < a ⟹ a < 1 ⟹ a * a ^ n < a ^ n*
by (*induct n*) (*auto simp*: *mult-strict-left-mono*)

lemma *power-strict-decreasing*: *n < N ⟹ 0 < a ⟹ a < 1 ⟹ a ^ N < a ^ n*
proof (*induction N*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc N*)
then show *?case*
using *mult-strict-mono*[*of a 1 a ^ N a ^ n*]
by (*auto simp add*: *power-Suc-less less-Suc-eq*)
qed

lemma *power-decreasing-iff* [*simp*]: ⟦*0 < b*; *b < 1*⟧ ⟹ *b ^ m ≤ b ^ n ⟷ n ≤ m*
using *power-strict-decreasing* [*of m n b*]
by (*auto intro*: *power-decreasing ccontr*)

lemma *power-strict-decreasing-iff* [*simp*]: ⟦*0 < b*; *b < 1*⟧ ⟹ *b ^ m < b ^ n ⟷ n < m*
using *power-decreasing-iff* [*of b m n*] **unfolding** *le-less*
by (*auto dest*: *power-strict-decreasing le-neq-implies-less*)

lemma *power-Suc-less-one*: *0 < a ⟹ a < 1 ⟹ a ^ Suc n < 1*
using *power-strict-decreasing* [*of 0 Suc n a*] **by** *simp*

Lemma for *power-strict-increasing*.

lemma *power-less-power-Suc*: *1 < a ⟹ a ^ n < a * a ^ n*
by (*induct n*) (*auto simp*: *mult-strict-left-mono less-trans* [*OF zero-less-one*])

lemma *power-strict-increasing*: *n < N ⟹ 1 < a ⟹ a ^ n < a ^ N*
proof (*induct N*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc N*)

then show *?case*
using *mult-strict-mono*[*of 1 a a^n a^N*]
by (*auto simp add*: *power-less-power-Suc less-Suc-eq less-trans* [*OF zero-less-one*] *less-imp-le*)
qed

Surely we can strengthen this? It holds for *0<a<1* too.

lemma *power-inject-exp* [*simp*]:
‹*a ^ m = a ^ n ⟷ m = n*› **if** ‹*1 < a*›
using *that* **by** (*force simp add*: *order-class.order.antisym power-le-imp-le-exp*)

lemma *power-inject-exp'*:
‹*a ^ m = a ^ n ⟷ m = n*› **if** ‹*a≠1*› ‹*a>0*›
using *that*
by (*metis linorder-neqE-nat not-less-iff-gr-or-eq power-inject-exp power-strict-decreasing*)

Can relax the first premise to $0 < a$ in the case of the natural numbers.

lemma *power-less-imp-less-exp*: $1 < a \Longrightarrow a \mathbin{\hat{}} m < a \mathbin{\hat{}} n \Longrightarrow m < n$
by (*simp add*: *order-less-le* [*of m n*] *less-le* [*of a^m a^n*] *power-le-imp-le-exp*)

lemma *power-increasing-iff* [*simp*]: $1 < b \Longrightarrow b \mathbin{\hat{}} x \leq b \mathbin{\hat{}} y \longleftrightarrow x \leq y$
by (*blast intro*: *power-le-imp-le-exp power-increasing less-imp-le*)

lemma *power-strict-increasing-iff* [*simp*]: $1 < b \Longrightarrow b \mathbin{\hat{}} x < b \mathbin{\hat{}} y \longleftrightarrow x < y$
by (*blast intro*: *power-less-imp-less-exp power-strict-increasing*)

lemma *power-le-imp-le-base*:
assumes *le*: $a \mathbin{\hat{}} Suc\ n \leq b \mathbin{\hat{}} Suc\ n$
and $0 \leq b$
shows $a \leq b$
proof (*rule ccontr*)
assume ¬ *?thesis*
then have $b < a$ **by** (*simp only*: *linorder-not-le*)
then have $b \mathbin{\hat{}} Suc\ n < a \mathbin{\hat{}} Suc\ n$
by (*simp only*: *assms*(*2*) *power-strict-mono*)
with *le* **show** *False*
by (*simp add*: *linorder-not-less* [*symmetric*])
qed

lemma *power-less-imp-less-base*:
assumes *less*: $a \mathbin{\hat{}} n < b \mathbin{\hat{}} n$
assumes *nonneg*: $0 \leq b$
shows $a < b$
proof (*rule contrapos-pp* [*OF less*])
assume ¬ *?thesis*
then have $b \leq a$ **by** (*simp only*: *linorder-not-less*)
from *this nonneg* **have** $b \mathbin{\hat{}} n \leq a \mathbin{\hat{}} n$ **by** (*rule power-mono*)
then show ¬ $a \mathbin{\hat{}} n < b \mathbin{\hat{}} n$ **by** (*simp only*: *linorder-not-less*)
qed

lemma *power-inject-base*: $a \mathbin{\hat{}} Suc\ n = b \mathbin{\hat{}} Suc\ n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a = b$
by (*blast intro*: *power-le-imp-le-base order.antisym eq-refl sym*)

lemma *power-eq-imp-eq-base*: $a \mathbin{\hat{}} n = b \mathbin{\hat{}} n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 < n \Longrightarrow a = b$
by (*cases n*) (*simp-all del*: *power-Suc*, *rule power-inject-base*)

lemma *power-eq-iff-eq-base*: $0 < n \Longrightarrow 0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a \mathbin{\hat{}} n = b \mathbin{\hat{}} n \longleftrightarrow a = b$
using *power-eq-imp-eq-base* [*of a n b*] **by** *auto*

lemma *power2-le-imp-le*: $x^2 \leq y^2 \Longrightarrow 0 \leq y \Longrightarrow x \leq y$
unfolding *numeral-2-eq-2* **by** (*rule power-le-imp-le-base*)

lemma *power2-less-imp-less*: $x^2 < y^2 \Longrightarrow 0 \leq y \Longrightarrow x < y$
by (*rule power-less-imp-less-base*)

lemma *power2-eq-imp-eq*: $x^2 = y^2 \Longrightarrow 0 \leq x \Longrightarrow 0 \leq y \Longrightarrow x = y$
unfolding *numeral-2-eq-2* **by** (*erule* (*2*) *power-eq-imp-eq-base*) *simp*

lemma *power2-eq-iff-nonneg* [*simp*]:
assumes $0 \leq x$ $0 \leq y$
shows $(x \mathbin{\hat{}} 2 = y \mathbin{\hat{}} 2) \longleftrightarrow x = y$
using *assms power2-eq-imp-eq* **by** *blast*

lemma *of-nat-less-numeral-power-cancel-iff* [*simp*]:
of-nat x $<$ *numeral i* $\mathbin{\hat{}}$ *n* $\longleftrightarrow$ *x* $<$ *numeral i* $\mathbin{\hat{}}$ *n*
using *of-nat-less-iff* [*of x numeral i* $\mathbin{\hat{}}$ *n*, *unfolded of-nat-numeral of-nat-power*] .

lemma *of-nat-le-numeral-power-cancel-iff* [*simp*]:
of-nat x $\leq$ *numeral i* $\mathbin{\hat{}}$ *n* $\longleftrightarrow$ *x* $\leq$ *numeral i* $\mathbin{\hat{}}$ *n*
using *of-nat-le-iff* [*of x numeral i* $\mathbin{\hat{}}$ *n*, *unfolded of-nat-numeral of-nat-power*] .

lemma *numeral-power-less-of-nat-cancel-iff* [*simp*]:
numeral i $\mathbin{\hat{}}$ *n* $<$ *of-nat x* $\longleftrightarrow$ *numeral i* $\mathbin{\hat{}}$ *n* $<$ *x*
using *of-nat-less-iff* [*of numeral i* $\mathbin{\hat{}}$ *n x*, *unfolded of-nat-numeral of-nat-power*] .

lemma *numeral-power-le-of-nat-cancel-iff* [*simp*]:
numeral i $\mathbin{\hat{}}$ *n* $\leq$ *of-nat x* $\longleftrightarrow$ *numeral i* $\mathbin{\hat{}}$ *n* $\leq$ *x*
using *of-nat-le-iff* [*of numeral i* $\mathbin{\hat{}}$ *n x*, *unfolded of-nat-numeral of-nat-power*] .

lemma *of-nat-le-of-nat-power-cancel-iff* [*simp*]: (*of-nat b*) $\mathbin{\hat{}} w \leq$ *of-nat x* $\longleftrightarrow$ *b* $\mathbin{\hat{}}$ *w* $\leq$ *x*
by (*metis of-nat-le-iff of-nat-power*)

lemma *of-nat-power-le-of-nat-cancel-iff* [*simp*]: *of-nat x* $\leq$ (*of-nat b*) $\mathbin{\hat{}} w \longleftrightarrow x \leq b \mathbin{\hat{}} w$
by (*metis of-nat-le-iff of-nat-power*)

lemma *of-nat-less-of-nat-power-cancel-iff*[*simp*]: (*of-nat b*) ^ *w* < *of-nat x* ⟷ *b* ^ *w* < *x*
by (*metis of-nat-less-iff of-nat-power*)

lemma *of-nat-power-less-of-nat-cancel-iff*[*simp*]: *of-nat x* < (*of-nat b*) ^ *w* ⟷ *x* < *b* ^ *w*
by (*metis of-nat-less-iff of-nat-power*)

lemma *power2-nonneg-ge-1-iff*:
assumes *x* ≥ *0*
shows *x* ^ *2* ≥ *1* ⟷ *x* ≥ *1*
using *assms* **by** (*auto intro*: *power2-le-imp-le*)

lemma *power2-nonneg-gt-1-iff*:
assumes *x* ≥ *0*
shows *x* ^ *2* > *1* ⟷ *x* > *1*
using *assms* **by** (*auto intro*: *power-less-imp-less-base*)

end

Some *nat*-specific lemmas:

lemma *mono-ge2-power-minus-self*:
assumes *k* ≥ *2* **shows** *mono* (λ*m*. *k* ^ *m* − *m*)
unfolding *mono-iff-le-Suc*
proof
fix *n*
have *k* ^ *n* < *k* ^ *Suc n* **using** *power-strict-increasing-iff*[*of k n Suc n*] *assms* **by** *linarith*
thus *k* ^ *n* − *n* ≤ *k* ^ *Suc n* − *Suc n* **by** *linarith*
qed

lemma *self-le-ge2-pow*[*simp*]:
assumes *k* ≥ *2* **shows** *m* ≤ *k* ^ *m*
proof (*induction m*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc m*)
hence *Suc m* ≤ *Suc* (*k* ^ *m*) **by** *simp*
also have ... ≤ *k*^*m* + *k*^*m* **using** *one-le-power*[*of k m*] *assms* **by** *linarith*
also have ... ≤ *k* ∗ *k*^*m* **by** (*metis mult-2 mult-le-mono1*[*OF assms*])
finally show *?case* **by** *simp*
qed

lemma *diff-le-diff-pow*[*simp*]:
assumes *k* ≥ *2* **shows** *m* − *n* ≤ *k* ^ *m* − *k* ^ *n*
proof (*cases n* ≤ *m*)
case *True*
thus *?thesis*

using *monoD*[*OF mono-ge2-power-minus-self*[*OF assms*] *True*] *self-le-ge2-pow*[*OF assms*, *of m*]
by (*simp add*: *le-diff-conv le-diff-conv2*)
qed *auto*

context *linordered-ring-strict*
begin

lemma *sum-squares-eq-zero-iff*: $x * x + y * y = 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*simp add*: *add-nonneg-eq-0-iff*)

lemma *sum-squares-le-zero-iff*: $x * x + y * y \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*simp add*: *le-less not-sum-squares-lt-zero sum-squares-eq-zero-iff*)

lemma *sum-squares-gt-zero-iff*: $0 < x * x + y * y \longleftrightarrow x \neq 0 \vee y \neq 0$
by (*simp add*: *not-le* [*symmetric*] *sum-squares-le-zero-iff*)

end

context *linordered-idom*
begin

lemma *zero-le-power2* [*simp*]: $0 \leq a^2$
by (*simp add*: *power2-eq-square*)

lemma *zero-less-power2* [*simp*]: $0 < a^2 \longleftrightarrow a \neq 0$
by (*force simp add*: *power2-eq-square zero-less-mult-iff linorder-neq-iff*)

lemma *power2-less-0* [*simp*]: $\neg\ a^2 < 0$
by (*force simp add*: *power2-eq-square mult-less-0-iff*)

lemma *power-abs*: $|a \mathbin{\hat{}} n| = |a| \mathbin{\hat{}} n$ — FIXME simp?
by (*induct n*) (*simp-all add*: *abs-mult*)

lemma *power-sgn* [*simp*]: $sgn\ (a \mathbin{\hat{}} n) = sgn\ a \mathbin{\hat{}} n$
by (*induct n*) (*simp-all add*: *sgn-mult*)

lemma *abs-power-minus* [*simp*]: $|(-\ a) \mathbin{\hat{}} n| = |a \mathbin{\hat{}} n|$
by (*simp add*: *power-abs*)

lemma *zero-less-power-abs-iff* [*simp*]: $0 < |a| \mathbin{\hat{}} n \longleftrightarrow a \neq 0 \vee n = 0$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*auto simp*: *zero-less-mult-iff*)
qed

lemma *zero-le-power-abs* [*simp*]: $0 \leq |a| \mathbin{\hat{}} n$
by (*rule zero-le-power* [*OF abs-ge-zero*])

lemma *power2-less-eq-zero-iff* [*simp*]: $a^2 \leq 0 \longleftrightarrow a = 0$
by (*simp add*: *le-less*)

lemma *abs-power2* [*simp*]: $|a^2| = a^2$
by (*simp add*: *power2-eq-square*)

lemma *power2-abs* [*simp*]: $|a|^2 = a^2$
by (*simp add*: *power2-eq-square*)

lemma *odd-power-less-zero*: $a < 0 \Longrightarrow a \mathbin{\hat{}} Suc\ (2 * n) < 0$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have $a \mathbin{\hat{}} Suc\ (2 * Suc\ n) = (a*a) * a \mathbin{\hat{}} Suc(2*n)$
by (*simp add*: *ac-simps power-add power2-eq-square*)
then show *?case*
by (*simp del*: *power-Suc add*: *Suc mult-less-0-iff mult-neg-neg*)
qed

lemma *odd-0-le-power-imp-0-le*: $0 \leq a \mathbin{\hat{}} Suc\ (2 * n) \Longrightarrow 0 \leq a$
using *odd-power-less-zero* [*of a n*]
by (*force simp add*: *linorder-not-less* [*symmetric*])

lemma *zero-le-even-power'*[*simp*]: $0 \leq a \mathbin{\hat{}} (2 * n)$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
have $a \mathbin{\hat{}} (2 * Suc\ n) = (a*a) * a \mathbin{\hat{}} (2*n)$
by (*simp add*: *ac-simps power-add power2-eq-square*)
then show *?case*
by (*simp add*: *Suc zero-le-mult-iff*)
qed

lemma *sum-power2-ge-zero*: $0 \leq x^2 + y^2$
by (*intro add-nonneg-nonneg zero-le-power2*)

lemma *not-sum-power2-lt-zero*: $\neg\ x^2 + y^2 < 0$
unfolding *not-less* **by** (*rule sum-power2-ge-zero*)

lemma *sum-power2-eq-zero-iff*: $x^2 + y^2 = 0 \longleftrightarrow x = 0 \wedge y = 0$
unfolding *power2-eq-square* **by** (*simp add*: *add-nonneg-eq-0-iff*)

lemma *sum-power2-le-zero-iff*: $x^2 + y^2 \leq 0 \longleftrightarrow x = 0 \wedge y = 0$
by (*simp add*: *le-less sum-power2-eq-zero-iff not-sum-power2-lt-zero*)

lemma *sum-power2-gt-zero-iff*: $0 < x^2 + y^2 \longleftrightarrow x \neq 0 \vee y \neq 0$
unfolding *not-le* [*symmetric*] **by** (*simp add*: *sum-power2-le-zero-iff*)

lemma *abs-le-square-iff*: $|x| \leq |y| \longleftrightarrow x^2 \leq y^2$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have $|x|^2 \leq |y|^2$ **by** (*rule power-mono*) *simp*
then show *?rhs* **by** *simp*
next
assume *?rhs*
then show *?lhs*
by (*auto intro*!: *power2-le-imp-le* [*OF - abs-ge-zero*])
qed

lemma *power2-le-iff-abs-le*:
$y \geq 0 \Longrightarrow x^2 \leq y^2 \longleftrightarrow |x| \leq y$
by (*metis abs-le-square-iff abs-of-nonneg*)

lemma *abs-square-le-1*:$x^2 \leq 1 \longleftrightarrow |x| \leq 1$
using *abs-le-square-iff* [*of x 1*] **by** *simp*

lemma *abs-square-eq-1*: $x^2 = 1 \longleftrightarrow |x| = 1$
by (*auto simp add*: *abs-if power2-eq-1-iff*)

lemma *abs-square-less-1*: $x^2 < 1 \longleftrightarrow |x| < 1$
using *abs-square-eq-1* [*of x*] *abs-square-le-1* [*of x*] **by** (*auto simp add*: *le-less*)

lemma *square-le-1*:
assumes $-1 \leq x$ $x \leq 1$
shows $x^2 \leq 1$
using *assms*
by (*metis add.inverse-inverse linear mult-le-one neg-equal-0-iff-equal neg-le-iff-le power2-eq-square power-minus-Bit0*)

lemma *power2-mono*: $|x| \leq |y| \Longrightarrow x \mathbin{\hat{}} 2 \leq y \mathbin{\hat{}} 2$
by (*simp add*: *abs-le-square-iff*)

lemma *power2-strict-mono*:
assumes $|x| < |y|$
shows $x \mathbin{\hat{}} 2 < y \mathbin{\hat{}} 2$
proof −
have $|x| \mathbin{\hat{}} 2 < |y| \mathbin{\hat{}} 2$
by (*rule power-strict-mono*) (*use assms* **in** *auto*)
thus *?thesis*

by *simp*
qed

end

47.3 Miscellaneous rules

context *linordered-semidom*
begin

lemma *self-le-power*: $1 \leq a \Longrightarrow 0 < n \Longrightarrow a \leq a \mathbin{\hat{}} n$
using *power-increasing* [*of* 1 *n a*] *power-one-right* [*of a*] **by** *auto*

lemma *power-le-one-iff*: $0 \leq a \Longrightarrow a \mathbin{\hat{}} n \leq 1 \longleftrightarrow (n = 0 \vee a \leq 1)$
by (*metis* (*mono-tags*) *gr0I nle-le one-le-power power-le-one self-le-power power-0*)

lemma *power-less1-D*: $a \hat{} n < 1 \Longrightarrow a < 1$
using *not-le one-le-power* **by** *blast*

lemma *power-less-one-iff*: $0 \leq a \Longrightarrow a \mathbin{\hat{}} n < 1 \longleftrightarrow (n > 0 \wedge a < 1)$
by (*metis* (*mono-tags*) *power-one power-strict-mono power-less1-D less-le-not-le neq0-conv power-0*)

end

lemma *power2-ge-1-iff*: $x \mathbin{\hat{}} 2 \geq 1 \longleftrightarrow x \geq 1 \vee x \leq (-1 :: 'a :: linordered\text{-}idom)$
using *abs-le-square-iff*[*of* 1 *x*] **by** (*auto simp*: *abs-if split*: *if-splits*)

lemma *power2-less-1-iff*: $x^2 < 1 \longleftrightarrow (-1 :: 'a :: linordered\text{-}idom) < x \wedge x < 1$
using *power2-ge-1-iff* [*of x*] **by** (*auto simp*: *less-le-not-le*)

lemma *power2-gt-1-iff*: $x^2 > 1 \longleftrightarrow x < (-1 :: 'a :: linordered\text{-}idom) \vee x > 1$
using *power2-ge-1-iff* [*of x*] *power2-eq-1-iff* [*of x*] **by** *auto*

lemma (**in** *power*) *power-eq-if*: $p \mathbin{\hat{}} m = (if\ m{=}0\ then\ 1\ else\ p * (p \mathbin{\hat{}} (m - 1)))$
unfolding *One-nat-def* **by** (*cases m*) *simp-all*

lemma (**in** *comm-semiring-1*) *power2-sum*: $(x + y)^2 = x^2 + y^2 + 2 * x * y$
by (*simp add*: *algebra-simps power2-eq-square mult-2-right*)

context *comm-ring-1*
begin

lemma *power2-diff*: $(x - y)^2 = x^2 + y^2 - 2 * x * y$
by (*simp add*: *algebra-simps power2-eq-square mult-2-right*)

lemma *power2-commute*: $(x - y)^2 = (y - x)^2$
by (*simp add*: *algebra-simps power2-eq-square*)

lemma *minus-power-mult-self*: $(-a) \hat{} n * (-a) \hat{} n = a \hat{} (2 * n)$
by (*simp add*: *power-mult-distrib* [*symmetric*])
(*simp add*: *power2-eq-square* [*symmetric*] *power-mult* [*symmetric*])

lemma *minus-one-mult-self* [*simp*]: $(-1) \hat{} n * (-1) \hat{} n = 1$
using *minus-power-mult-self* [*of 1 n*] **by** *simp*

lemma *left-minus-one-mult-self* [*simp*]: $(-1) \hat{} n * ((-1) \hat{} n * a) = a$
by (*simp add*: *mult.assoc* [*symmetric*])

end

Simprules for comparisons where common factors can be cancelled.

lemmas *zero-compare-simps* =
add-strict-increasing add-strict-increasing2 add-increasing
zero-le-mult-iff zero-le-divide-iff
zero-less-mult-iff zero-less-divide-iff
mult-le-0-iff divide-le-0-iff
mult-less-0-iff divide-less-0-iff
zero-le-power2 power2-less-0

47.4 Exponentiation for the Natural Numbers

lemma *nat-one-le-power* [*simp*]: $Suc\ 0 \leq i \Longrightarrow Suc\ 0 \leq i \hat{} n$
by (*rule one-le-power* [*of i n*, *unfolded One-nat-def*])

lemma *nat-zero-less-power-iff* [*simp*]: $x \hat{} n > 0 \longleftrightarrow x > 0 \vee n = 0$
for $x :: nat$
by (*induct n*) *auto*

lemma *nat-power-eq-Suc-0-iff* [*simp*]: $x \hat{} m = Suc\ 0 \longleftrightarrow m = 0 \vee x = Suc\ 0$
by (*induct m*) *auto*

lemma *power-Suc-0* [*simp*]: $Suc\ 0 \hat{} n = Suc\ 0$
by *simp*

Valid for the naturals, but what if $0 < i < 1$? Premises cannot be weakened: consider the case where $i = 0$, $m = 1$ and $n = 0$.

lemma *nat-power-less-imp-less*:
fixes $i :: nat$
assumes *nonneg*: $0 < i$
assumes *less*: $i \hat{} m < i \hat{} n$
shows $m < n$
proof (*cases* $i = 1$)
case *True*
with *less power-one* [**where** $'a = nat$] **show** *?thesis* **by** *simp*
next
case *False*
with *nonneg* **have** $1 < i$ **by** *auto*

from *power-strict-increasing-iff* [*OF this*] *less* **show** *?thesis* **..**
qed

lemma *power-gt-expt*: $n > Suc\ 0 \Longrightarrow n\hat{}k > k$
by (*induction k*) (*auto simp*: *less-trans-Suc n-less-m-mult-n*)

lemma *less-exp* [*simp*]:
$\langle n < 2 \hat{} n \rangle$
by (*simp add*: *power-gt-expt*)

lemma *power-dvd-imp-le*:
fixes $i :: nat$
assumes $i \hat{} m\ dvd\ i \hat{} n\ 1 < i$
shows $m \leq n$
using *assms* **by** (*auto intro*: *power-le-imp-le-exp* [*OF* $\langle 1 < i \rangle$ *dvd-imp-le*])

lemma *dvd-power-iff-le*:
fixes $k{::}nat$
shows $2 \leq k \Longrightarrow ((k \hat{} m)\ dvd\ (k \hat{} n) \longleftrightarrow m \leq n)$
using *le-imp-power-dvd power-dvd-imp-le* **by** *force*

lemma *power2-nat-le-eq-le*: $m^2 \leq n^2 \longleftrightarrow m \leq n$
for $m\ n :: nat$
by (*auto intro*: *power2-le-imp-le power-mono*)

lemma *power2-nat-le-imp-le*:
fixes $m\ n :: nat$
assumes $m^2 \leq n$
shows $m \leq n$
proof (*cases m*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc k*)
show *?thesis*
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have $n < m$ **by** *simp*
with *assms Suc* **show** *False*
by (*simp add*: *power2-eq-square*)
qed
qed

lemma *ex-power-ivl1*: **fixes** $b\ k :: nat$ **assumes** $b \geq 2$
shows $k \geq 1 \Longrightarrow \exists n.\ b\hat{}n \leq k \wedge k < b\hat{}(n+1)$ (**is** - $\Longrightarrow \exists n.\ ?P\ k\ n$)
proof(*induction k*)
case *0* **thus** *?case* **by** *simp*
next
case (*Suc k*)

```
  show ?case
  proof cases
    assume k=0
    hence ?P (Suc k) 0 using assms by simp
    thus ?case ..
  next
    assume k≠0
    with Suc obtain n where IH: ?P k n by auto
    show ?case
    proof (cases k = b^(n+1) - 1)
      case True
      hence ?P (Suc k) (n+1) using assms
        by (simp add: power-less-power-Suc)
      thus ?thesis ..
    next
      case False
      hence ?P (Suc k) n using IH by auto
      thus ?thesis ..
    qed
  qed
qed

lemma ex-power-ivl2: fixes b k :: nat assumes b ≥ 2 k ≥ 2
  shows ∃ n. b^n < k ∧ k ≤ b^(n+1)
proof −
  have 1 ≤ k − 1 using assms(2) by arith
  from ex-power-ivl1[OF assms(1) this]
  obtain n where b ^ n ≤ k − 1 ∧ k − 1 < b ^ (n + 1) ..
  hence b^n < k ∧ k ≤ b^(n+1) using assms by auto
  thus ?thesis ..
qed
```

47.4.1 Cardinality of the Powerset

```
lemma card-UNIV-bool [simp]: card (UNIV :: bool set) = 2
  unfolding UNIV-bool by simp

lemma card-Pow: finite A ⟹ card (Pow A) = 2 ^ card A
proof (induct rule: finite-induct)
  case empty
  show ?case by simp
next
  case (insert x A)
  from ‹x ∉ A› have disjoint: Pow A ∩ insert x ` Pow A = {} by blast
  from ‹x ∉ A› have inj-on: inj-on (insert x) (Pow A)
    unfolding inj-on-def by auto

  have card (Pow (insert x A)) = card (Pow A ∪ insert x ` Pow A)
    by (simp only: Pow-insert)
```

also have ... = *card* (*Pow A*) + *card* (*insert x ' Pow A*)
by (*rule card-Un-disjoint*) (*use ‹finite A› disjoint* **in** *simp-all*)
also from *inj-on* **have** *card* (*insert x ' Pow A*) = *card* (*Pow A*)
by (*rule card-image*)
also have ... + ... = *2* * ... **by** (*simp add*: *mult-2*)
also from *insert*(*3*) **have** ... = *2* ^ *Suc* (*card A*) **by** *simp*
also from *insert*(*1*,*2*) **have** *Suc* (*card A*) = *card* (*insert x A*)
by (*rule card-insert-disjoint* [*symmetric*])
finally show *?case* **.**
qed

47.5 Code generator tweak

code-identifier
code-module *Power* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

48 Big sum and product over finite (non-empty) sets

theory *Groups-Big*
imports *Power Equiv-Relations*
begin

48.1 Generic monoid operation over a set

locale *comm-monoid-set* = *comm-monoid*
begin

48.1.1 Standard sum or product indexed by a finite set

interpretation *comp-fun-commute f*
by *standard* (*simp add*: *fun-eq-iff left-commute*)

interpretation *comp?*: *comp-fun-commute f* ∘ *g*
by (*fact comp-comp-fun-commute*)

definition *F* :: ('*b* ⇒ '*a*) ⇒ '*b set* ⇒ '*a*
where *eq-fold*: *F g A* = *Finite-Set.fold* (*f* ∘ *g*) **1** *A*

lemma *infinite* [*simp*]: ¬ *finite A* ⟹ *F g A* = **1**
by (*simp add*: *eq-fold*)

lemma *empty* [*simp*]: *F g* {} = **1**
by (*simp add*: *eq-fold*)

lemma *insert* [*simp*]: *finite A* ⟹ *x* ∉ *A* ⟹ *F g* (*insert x A*) = *g x* ∗ *F g A*
by (*simp add*: *eq-fold*)

lemma *remove*:
assumes *finite A* **and** $x \in A$
shows $F\ g\ A = g\ x * F\ g\ (A - \{x\})$
proof −
from ‹$x \in A$› **obtain** *B* **where** *B*: $A = \mathit{insert}\ x\ B$ **and** $x \notin B$
by (*auto dest*: *mk-disjoint-insert*)
moreover from ‹*finite A*› *B* **have** *finite B* **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *insert-remove*: *finite A* $\Longrightarrow$ $F\ g\ (\mathit{insert}\ x\ A) = g\ x * F\ g\ (A - \{x\})$
by (*cases* $x \in A$) (*simp-all add*: *remove insert-absorb*)

lemma *insert-if*: *finite A* $\Longrightarrow$ $F\ g\ (\mathit{insert}\ x\ A) = (\mathit{if}\ x \in A\ \mathit{then}\ F\ g\ A\ \mathit{else}\ g\ x * F\ g\ A)$
by (*cases* $x \in A$) (*simp-all add*: *insert-absorb*)

lemma *neutral*: $\forall x{\in}A.\ g\ x = \mathbf{1} \Longrightarrow F\ g\ A = \mathbf{1}$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *neutral-const* [*simp*]: $F\ (\lambda\text{-}.\ \mathbf{1})\ A = \mathbf{1}$
by (*simp add*: *neutral*)

lemma *union-inter*:
assumes *finite A* **and** *finite B*
shows $F\ g\ (A \cup B) * F\ g\ (A \cap B) = F\ g\ A * F\ g\ B$
— The reversed orientation looks more natural, but LOOPS as a simprule!
using *assms*
proof (*induct A*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert x A*)
then show *?case*
by (*auto simp*: *insert-absorb Int-insert-left commute* [*of* - *g x*] *assoc left-commute*)
qed

corollary *union-inter-neutral*:
assumes *finite A* **and** *finite B*
and $\forall x \in A \cap B.\ g\ x = \mathbf{1}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
using *assms* **by** (*simp add*: *union-inter* [*symmetric*] *neutral*)

corollary *union-disjoint*:
assumes *finite A* **and** *finite B*
assumes $A \cap B = \{\}$
shows $F\ g\ (A \cup B) = F\ g\ A * F\ g\ B$
using *assms* **by** (*simp add*: *union-inter-neutral*)

lemma *union-diff2*:
assumes *finite A* **and** *finite B*
shows $F\ g\ (A \cup B) = F\ g\ (A - B) * F\ g\ (B - A) * F\ g\ (A \cap B)$
proof −
have $A \cup B = A - B \cup (B - A) \cup A \cap B$
by *auto*
with *assms* **show** *?thesis*
by *simp* (*subst union-disjoint*, *auto*)+
qed

lemma *subset-diff*:
assumes $B \subseteq A$ **and** *finite A*
shows $F\ g\ A = F\ g\ (A - B) * F\ g\ B$
proof −
from *assms* **have** *finite* $(A - B)$ **by** *auto*
moreover from *assms* **have** *finite B* **by** (*rule finite-subset*)
moreover from *assms* **have** $(A - B) \cap B = \{\}$ **by** *auto*
ultimately have $F\ g\ (A - B \cup B) = F\ g\ (A - B) * F\ g\ B$ **by** (*rule union-disjoint*)
moreover from *assms* **have** $A \cup B = A$ **by** *auto*
ultimately show *?thesis* **by** *simp*
qed

lemma *Int-Diff*:
assumes *finite A*
shows $F\ g\ A = F\ g\ (A \cap B) * F\ g\ (A - B)$
by (*subst subset-diff* [**where** $B = A - B$]) (*auto simp*: *Diff-Diff-Int assms*)

lemma *setdiff-irrelevant*:
assumes *finite A*
shows $F\ g\ (A - \{x.\ g\ x = z\}) = F\ g\ A$
using *assms* **by** (*induct A*) (*simp-all add*: *insert-Diff-if*)

lemma *not-neutral-contains-not-neutral*:
assumes $F\ g\ A \neq \mathbf{1}$
obtains *a* **where** $a \in A$ **and** $g\ a \neq \mathbf{1}$
proof −
from *assms* **have** $\exists\, a{\in}A.\ g\ a \neq \mathbf{1}$
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
then show *?case* **by** *fastforce*
qed
with *that* **show** *thesis* **by** *blast*

qed

lemma *reindex*:
assumes *inj-on h A*
shows *F g* (*h* ‘ *A*) = *F* (*g* ∘ *h*) *A*
proof (*cases finite A*)
case *True*
with *assms* **show** *?thesis*
by (*simp add*: *eq-fold fold-image comp-assoc*)
next
case *False*
with *assms* **have** ¬ *finite* (*h* ‘ *A*) **by** (*blast dest*: *finite-imageD*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *cong* [*fundef-cong*]:
assumes *A* = *B*
assumes *g-h*: ⋀*x*. *x* ∈ *B* ⟹ *g x* = *h x*
shows *F g A* = *F h B*
using *g-h* **unfolding** ‹*A* = *B*›
by (*induct B rule*: *infinite-finite-induct*) *auto*

lemma *cong-simp* [*cong*]:
⟦ *A* = *B*; ⋀*x*. *x* ∈ *B* =*simp*=> *g x* = *h x* ⟧ ⟹ *F* (λ*x*. *g x*) *A* = *F* (λ*x*. *h x*) *B*
by (*rule cong*) (*simp-all add*: *simp-implies-def*)

lemma *reindex-cong*:
assumes *inj-on l B*
assumes *A* = *l* ‘ *B*
assumes ⋀*x*. *x* ∈ *B* ⟹ *g* (*l x*) = *h x*
shows *F g A* = *F h B*
using *assms* **by** (*simp add*: *reindex*)

lemma *image-eq*:
assumes *inj-on g A*
shows *F* (λ*x*. *x*) (*g* ‘ *A*) = *F g A*
using *assms reindex-cong* **by** *fastforce*

lemma *UNION-disjoint*:
assumes *finite I* **and** ∀ *i*∈*I*. *finite* (*A i*)
and ∀ *i*∈*I*. ∀ *j*∈*I*. *i* ≠ *j* ⟶ *A i* ∩ *A j* = {}
shows *F g* (⋃(*A* ‘ *I*)) = *F* (λ*x*. *F g* (*A x*)) *I*
using *assms*
proof (*induction rule*: *finite-induct*)
case (*insert i I*)
then have ∀ *j*∈*I*. *j* ≠ *i*
by *blast*
with *insert.prems* **have** *A i* ∩ ⋃(*A* ‘ *I*) = {}
by *blast*

with *insert* **show** *?case*
by (*simp add*: *union-disjoint*)
qed *auto*

lemma *Union-disjoint*:
assumes $\forall A{\in}C.$ *finite* A $\forall A{\in}C.$ $\forall B{\in}C.$ $A \neq B \longrightarrow A \cap B = \{\}$
shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
proof (*cases finite C*)
case *True*
from *UNION-disjoint* [*OF this assms*] **show** *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **by** (*auto dest*: *finite-UnionD intro*: *infinite*)
qed

lemma *distrib*: $F\ (\lambda x.\ g\ x * h\ x)\ A = F\ g\ A * F\ h\ A$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *assoc commute left-commute*)

lemma *Sigma*:
assumes *finite* A $\forall x{\in}A.$ *finite* $(B\ x)$
shows $F\ (\lambda x.\ F\ (g\ x)\ (B\ x))\ A = F$ (*case-prod g*) (*SIGMA x*:*A*. *B x*)
unfolding *Sigma-def*
proof (*subst UNION-disjoint*)
show $F\ (\lambda x.\ F\ (g\ x)\ (B\ x))\ A = F\ (\lambda x.\ F\ (\lambda(x,\ y).\ g\ x\ y)\ (\bigcup y{\in}B\ x.\ \{(x,\ y)\}))$
A
proof (*rule cong* [*OF refl*])
show $F\ (g\ x)\ (B\ x) = F\ (\lambda(x,\ y).\ g\ x\ y)\ (\bigcup y{\in}B\ x.\ \{(x,\ y)\})$
if $x \in A$ **for** x
using *that assms* **by** (*simp add*: *UNION-disjoint*)
qed
qed (*use assms* **in** *auto*)

lemma *related*:
assumes *Re*: R **1** **1**
and *Rop*: $\forall x1\ y1\ x2\ y2.\ R\ x1\ x2 \wedge R\ y1\ y2 \longrightarrow R\ (x1 * y1)\ (x2 * y2)$
and *fin*: *finite S*
and *R-h-g*: $\forall x{\in}S.\ R\ (h\ x)\ (g\ x)$
shows $R\ (F\ h\ S)\ (F\ g\ S)$
using *fin* **by** (*rule finite-subset-induct*) (*use assms* **in** *auto*)

lemma *mono-neutral-cong-left*:
assumes *finite T*
and $S \subseteq T$
and $\forall i \in T - S.\ h\ i = \mathbf{1}$
and $\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x$
shows $F\ g\ S = F\ h\ T$
proof−
have *eq*: $T = S \cup (T - S)$ **using** ‹$S \subseteq T$› **by** *blast*
have *d*: $S \cap (T - S) = \{\}$ **using** ‹$S \subseteq T$› **by** *blast*

from ‹*finite T*› ‹$S \subseteq T$› **have** *f*: *finite S finite* $(T - S)$
by (*auto intro*: *finite-subset*)
show *?thesis* **using** *assms*(*4*)
by (*simp add*: *union-disjoint* [*OF f d*, *unfolded eq* [*symmetric*]] *neutral* [*OF assms*(*3*)])
qed

lemma *mono-neutral-cong-right*:
finite $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x) \Longrightarrow$
$F\ g\ T = F\ h\ S$
by (*auto intro*!: *mono-neutral-cong-left* [*symmetric*])

lemma *mono-neutral-left*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F\ g\ S = F\ g\ T$
by (*blast intro*: *mono-neutral-cong-left*)

lemma *mono-neutral-right*: *finite* $T \Longrightarrow S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow F\ g\ T = F\ g\ S$
by (*blast intro*!: *mono-neutral-left* [*symmetric*])

lemma *mono-neutral-cong*:
assumes [*simp*]: *finite T finite S*
and ∗: $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = \mathbf{1}$ $\bigwedge i.\ i \in S - T \Longrightarrow g\ i = \mathbf{1}$
and *gh*: $\bigwedge x.\ x \in S \cap T \Longrightarrow g\ x = h\ x$
shows $F\ g\ S = F\ h\ T$
proof−
have $F\ g\ S = F\ g\ (S \cap T)$
by(*rule mono-neutral-right*)(*auto intro*: ∗)
also have $\ldots = F\ h\ (S \cap T)$ **using** *refl gh* **by**(*rule cong*)
also have $\ldots = F\ h\ T$
by(*rule mono-neutral-left*)(*auto intro*: ∗)
finally show *?thesis* **.**
qed

lemma *reindex-bij-betw*: *bij-betw h S T* $\Longrightarrow F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
by (*auto simp*: *bij-betw-def reindex*)

lemma *reindex-bij-witness*:
assumes *witness*:
$\bigwedge a.\ a \in S \Longrightarrow i\ (j\ a) = a$
$\bigwedge a.\ a \in S \Longrightarrow j\ a \in T$
$\bigwedge b.\ b \in T \Longrightarrow j\ (i\ b) = b$
$\bigwedge b.\ b \in T \Longrightarrow i\ b \in S$
assumes *eq*:
$\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
proof −
have *bij-betw j S T*

using *bij-betw-byWitness*[**where** $A{=}S$ **and** $f{=}j$ **and** $f'{=}i$ **and** $A'{=}T$] *witness*
by *auto*
moreover have $F\ g\ S = F\ (\lambda x.\ h\ (j\ x))\ S$
by (*intro cong*) (*auto simp*: *eq*)
ultimately show *?thesis*
by (*simp add*: *reindex-bij-betw*)
qed

lemma *reindex-bij-betw-not-neutral*:
assumes *fin*: *finite* S' *finite* T'
assumes *bij*: *bij-betw* $h\ (S - S')\ (T - T')$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ (h\ a) = z$
$\bigwedge b.\ b \in T' \Longrightarrow g\ b = z$
shows $F\ (\lambda x.\ g\ (h\ x))\ S = F\ g\ T$
proof −
have [*simp*]: *finite* $S \longleftrightarrow$ *finite* T
using *bij-betw-finite*[*OF bij*] *fin* **by** *auto*
show *?thesis*
proof (*cases finite S*)
case *True*
with *nn* **have** $F\ (\lambda x.\ g\ (h\ x))\ S = F\ (\lambda x.\ g\ (h\ x))\ (S - S')$
by (*intro mono-neutral-cong-right*) *auto*
also have $\ldots = F\ g\ (T - T')$
using *bij* **by** (*rule reindex-bij-betw*)
also have $\ldots = F\ g\ T$
using *nn* ‹*finite S*› **by** (*intro mono-neutral-cong-left*) *auto*
finally show *?thesis* **.**
next
case *False*
then show *?thesis* **by** *simp*
qed
qed

lemma *reindex-nontrivial*:
assumes *finite A*
and *nz*: $\bigwedge x\ y.\ x \in A \Longrightarrow y \in A \Longrightarrow x \neq y \Longrightarrow h\ x = h\ y \Longrightarrow g\ (h\ x) = \mathbf{1}$
shows $F\ g\ (h\ `\ A) = F\ (g \circ h)\ A$
proof (*subst reindex-bij-betw-not-neutral* [*symmetric*])
show *bij-betw* $h\ (A - \{x \in A.\ (g \circ h)\ x = \mathbf{1}\})\ (h\ `\ A - h\ `\ \{x \in A.\ (g \circ h)\ x = \mathbf{1}\})$
using *nz* **by** (*auto intro*!: *inj-onI simp*: *bij-betw-def*)
qed (*use* ‹*finite A*› **in** *auto*)

lemma *reindex-bij-witness-not-neutral*:
assumes *fin*: *finite* S' *finite* T'
assumes *witness*:
$\bigwedge a.\ a \in S - S' \Longrightarrow i\ (j\ a) = a$
$\bigwedge a.\ a \in S - S' \Longrightarrow j\ a \in T - T'$

$\bigwedge b.\ b \in T - T' \Longrightarrow j\ (i\ b) = b$
$\bigwedge b.\ b \in T - T' \Longrightarrow i\ b \in S - S'$
assumes *nn*:
$\bigwedge a.\ a \in S' \Longrightarrow g\ a = z$
$\bigwedge b.\ b \in T' \Longrightarrow h\ b = z$
assumes *eq*:
$\bigwedge a.\ a \in S \Longrightarrow h\ (j\ a) = g\ a$
shows $F\ g\ S = F\ h\ T$
proof −
have *bij*: *bij-betw j* $(S - (S' \cap S))\ (T - (T' \cap T))$
using *witness* **by** (*intro bij-betw-byWitness*[**where** $f'{=}i$]) *auto*
have *F-eq*: $F\ g\ S = F\ (\lambda x.\ h\ (j\ x))\ S$
by (*intro cong*) (*auto simp*: *eq*)
show *?thesis*
unfolding *F-eq* **using** *fin nn eq*
by (*intro reindex-bij-betw-not-neutral*[*OF - - bij*]) *auto*
qed

lemma *delta-remove*:
assumes *fS*: *finite S*
shows $F\ (\lambda k.$ *if* $k = a$ *then* $b\ k$ *else* $c\ k)\ S = ($*if* $a \in S$ *then* $b\ a * F\ c\ (S{-}\{a\})$ *else* $F\ c\ (S{-}\{a\}))$
proof −
let *?f* $= (\lambda k.$ *if* $k = a$ *then* $b\ k$ *else* $c\ k)$
show *?thesis*
proof (*cases* $a \in S$)
case *False*
then have $\forall\, k{\in}S.$ *?f* $k = c\ k$ **by** *simp*
with *False* **show** *?thesis* **by** *simp*
next
case *True*
let *?A* $= S - \{a\}$
let *?B* $= \{a\}$
from *True* **have** *eq*: $S =$ *?A* $\cup$ *?B* **by** *blast*
have *dj*: *?A* $\cap$ *?B* $= \{\}$ **by** *simp*
from *fS* **have** *fAB*: *finite ?A finite ?B* **by** *auto*
have F *?f S* $= F$ *?f ?A* $*$ F *?f ?B*
using *union-disjoint* [*OF fAB dj*, *of ?f*, *unfolded eq* [*symmetric*]] **by** *simp*
with *True* **show** *?thesis*
using *comm-monoid-set.remove comm-monoid-set-axioms fS* **by** *fastforce*
qed
qed

lemma *delta* [*simp*]:
assumes *fS*: *finite S*
shows $F\ (\lambda k.$ *if* $k = a$ *then* $b\ k$ *else* $\mathbf{1})\ S = ($*if* $a \in S$ *then* $b\ a$ *else* $\mathbf{1})$
by (*simp add*: *delta-remove* [*OF assms*])

lemma *delta'* [*simp*]:

assumes *fin*: *finite S*
shows *F* (λ*k*. *if a* = *k then b k else* **1**) *S* = (*if a* ∈ *S then b a else* **1**)
using *delta* [*OF fin*, *of a b*, *symmetric*] **by** (*auto intro*: *cong*)

lemma *If-cases*:
fixes *P* :: *'b* ⇒ *bool* **and** *g h* :: *'b* ⇒ *'a*
assumes *fin*: *finite A*
shows *F* (λ*x*. *if P x then h x else g x*) *A* = *F h* (*A* ∩ {*x*. *P x*}) ∗ *F g* (*A* ∩ − {*x*. *P x*})
proof −
have *a*: *A* = *A* ∩ {*x*. *P x*} ∪ *A* ∩ −{*x*. *P x*} (*A* ∩ {*x*. *P x*}) ∩ (*A* ∩ −{*x*. *P x*}) = {}
by *blast*+
from *fin* **have** *f*: *finite* (*A* ∩ {*x*. *P x*}) *finite* (*A* ∩ −{*x*. *P x*}) **by** *auto*
let *?g* = λ*x*. *if P x then h x else g x*
from *union-disjoint* [*OF f a*(*2*), *of ?g*] *a*(*1*) **show** *?thesis*
by (*subst* (*1 2*) *cong*) *simp-all*
qed

lemma *cartesian-product*: *F* (λ*x*. *F* (*g x*) *B*) *A* = *F* (*case-prod g*) (*A* × *B*)
proof (*cases A* = {} ∨ *B* = {})
case *True*
then show *?thesis*
by *auto*
next
case *False*
then have *A* ≠ {} *B* ≠ {} **by** *auto*
show *?thesis*
proof (*cases finite A* ∧ *finite B*)
case *True*
then show *?thesis*
by (*simp add*: *Sigma*)
next
case *False*
then consider *infinite A* | *infinite B* **by** *auto*
then have *infinite* (*A* × *B*)
by *cases* (*use* ‹*A* ≠ {}› ‹*B* ≠ {}› **in** ‹*auto dest*: *finite-cartesian-productD1 finite-cartesian-productD2*›)
then show *?thesis*
using *False* **by** *auto*
qed
qed

lemma *cartesian-product'*:
F g (*A* × *B*) = *F* (λ*x*. *F* (λ*y*. *g* (*x*,*y*)) *B*) *A*
unfolding *cartesian-product* **by** *simp*

lemma *inter-restrict*:

assumes *finite A*
shows $F\ g\ (A \cap B) = F\ (\lambda x.\ \mathit{if}\ x \in B\ \mathit{then}\ g\ x\ \mathit{else}\ \mathbf{1})\ A$
proof −
let $?g = \lambda x.\ \mathit{if}\ x \in A \cap B\ \mathit{then}\ g\ x\ \mathit{else}\ \mathbf{1}$
have $\forall i{\in}A - A \cap B.\ (\mathit{if}\ i \in A \cap B\ \mathit{then}\ g\ i\ \mathit{else}\ \mathbf{1}) = \mathbf{1}$ **by** *simp*
moreover have $A \cap B \subseteq A$ **by** *blast*
ultimately have $F\ ?g\ (A \cap B) = F\ ?g\ A$
using ‹*finite A*› **by** (*intro mono-neutral-left*) *auto*
then show *?thesis* **by** *simp*
qed

lemma *inter-filter*:
$\mathit{finite}\ A \Longrightarrow F\ g\ \{x \in A.\ P\ x\} = F\ (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ g\ x\ \mathit{else}\ \mathbf{1})\ A$
by (*simp add*: *inter-restrict* [*symmetric*, *of A* $\{x.\ P\ x\}$ *g*, *simplified mem-Collect-eq*] *Int-def*)

lemma *Union-comp*:
assumes $\forall A \in B.\ \mathit{finite}\ A$
and $\bigwedge A1\ A2\ x.\ A1 \in B \Longrightarrow A2 \in B \Longrightarrow A1 \neq A2 \Longrightarrow x \in A1 \Longrightarrow x \in A2 \Longrightarrow g\ x = \mathbf{1}$
shows $F\ g\ (\bigcup B) = (F \circ F)\ g\ B$
using *assms*
proof (*induct B rule*: *infinite-finite-induct*)
case (*infinite A*)
then have ¬ *finite* $(\bigcup A)$ **by** (*blast dest*: *finite-UnionD*)
with *infinite* **show** *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert A B*)
then have *finite A finite B finite* $(\bigcup B)$ $A \notin B$
and $\forall x{\in}A \cap \bigcup B.\ g\ x = \mathbf{1}$
and *H*: $F\ g\ (\bigcup B) = (F \circ F)\ g\ B$ **by** *auto*
then have $F\ g\ (A \cup \bigcup B) = F\ g\ A * F\ g\ (\bigcup B)$
by (*simp add*: *union-inter-neutral*)
with ‹*finite B*› ‹$A \notin B$› **show** *?case*
by (*simp add*: *H*)
qed

lemma *swap*: $F\ (\lambda i.\ F\ (g\ i)\ B)\ A = F\ (\lambda j.\ F\ (\lambda i.\ g\ i\ j)\ A)\ B$
unfolding *cartesian-product*
by (*rule reindex-bij-witness* [**where** $i = \lambda(i,\ j).\ (j,\ i)$ **and** $j = \lambda(i,\ j).\ (j,\ i)$]) *auto*

lemma *swap-restrict*:
$\mathit{finite}\ A \Longrightarrow \mathit{finite}\ B \Longrightarrow$
$F\ (\lambda x.\ F\ (g\ x)\ \{y.\ y \in B \wedge R\ x\ y\})\ A = F\ (\lambda y.\ F\ (\lambda x.\ g\ x\ y)\ \{x.\ x \in A \wedge R\ x\ y\})\ B$

by (*simp add*: *inter-filter*) (*rule swap*)

lemma *image-gen*:
assumes *fin*: *finite S*
shows $F\ h\ S = F\ (\lambda y.\ F\ h\ \{x.\ x \in S \wedge g\ x = y\})\ (g\ `\ S)$
proof −
have $\{y.\ y \in g`S \wedge g\ x = y\} = \{g\ x\}$ **if** $x \in S$ **for** x
using *that* **by** *auto*
then have $F\ h\ S = F\ (\lambda x.\ F\ (\lambda y.\ h\ x)\ \{y.\ y \in g`S \wedge g\ x = y\})\ S$
by *simp*
also have $\ldots = F\ (\lambda y.\ F\ h\ \{x.\ x \in S \wedge g\ x = y\})\ (g\ `\ S)$
by (*rule swap-restrict* [*OF fin finite-imageI* [*OF fin*]])
finally show *?thesis* **.**
qed

lemma *group*:
assumes *fS*: *finite S* **and** *fT*: *finite T* **and** *fST*: $g\ `\ S \subseteq T$
shows $F\ (\lambda y.\ F\ h\ \{x.\ x \in S \wedge g\ x = y\})\ T = F\ h\ S$
unfolding *image-gen*[*OF fS*, *of h g*]
by (*auto intro*: *neutral mono-neutral-right*[*OF fT fST*])

lemma *Plus*:
fixes $A :: 'b\ set$ **and** $B :: 'c\ set$
assumes *fin*: *finite A finite B*
shows $F\ g\ (A <+> B) = F\ (g \circ Inl)\ A * F\ (g \circ Inr)\ B$
proof −
have $A <+> B = Inl\ `\ A \cup Inr\ `\ B$ **by** *auto*
moreover from *fin* **have** *finite* $(Inl\ `\ A)$ *finite* $(Inr\ `\ B)$ **by** *auto*
moreover have $Inl\ `\ A \cap Inr\ `\ B = \{\}$ **by** *auto*
moreover have *inj-on Inl A inj-on Inr B* **by** (*auto intro*: *inj-onI*)
ultimately show *?thesis*
using *fin* **by** (*simp add*: *union-disjoint reindex*)
qed

lemma *same-carrier*:
assumes *finite C*
assumes *subset*: $A \subseteq C\ B \subseteq C$
assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}\ \bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
shows $F\ g\ A = F\ h\ B \longleftrightarrow F\ g\ C = F\ h\ C$
proof −
have *finite A* **and** *finite B* **and** *finite* $(C - A)$ **and** *finite* $(C - B)$
using ‹*finite C*› *subset* **by** (*auto elim*: *finite-subset*)
from *subset* **have** [*simp*]: $A - (C - A) = A$ **by** *auto*
from *subset* **have** [*simp*]: $B - (C - B) = B$ **by** *auto*
from *subset* **have** $C = A \cup (C - A)$ **by** *auto*
then have $F\ g\ C = F\ g\ (A \cup (C - A))$ **by** *simp*
also have $\ldots = F\ g\ (A - (C - A)) * F\ g\ (C - A - A) * F\ g\ (A \cap (C - A))$
using ‹*finite A*› ‹*finite* $(C - A)$› **by** (*simp only*: *union-diff2*)
finally have $*$: $F\ g\ C = F\ g\ A$ **using** *trivial* **by** *simp*

from *subset* **have** $C = B \cup (C - B)$ **by** *auto*
then have $F\ h\ C = F\ h\ (B \cup (C - B))$ **by** *simp*
also have $\ldots = F\ h\ (B - (C - B)) * F\ h\ (C - B - B) * F\ h\ (B \cap (C - B))$
using ‹*finite B*› ‹*finite* $(C - B)$› **by** (*simp only*: *union-diff2*)
finally have $F\ h\ C = F\ h\ B$
using *trivial* **by** *simp*
with $*$ **show** *?thesis* **by** *simp*
qed

lemma *same-carrierI*:
assumes *finite C*
assumes *subset*: $A \subseteq C\ B \subseteq C$
assumes *trivial*: $\bigwedge a.\ a \in C - A \Longrightarrow g\ a = \mathbf{1}\ \bigwedge b.\ b \in C - B \Longrightarrow h\ b = \mathbf{1}$
assumes $F\ g\ C = F\ h\ C$
shows $F\ g\ A = F\ h\ B$
using *assms same-carrier* [*of C A B*] **by** *simp*

lemma *eq-general*:
assumes B: $\bigwedge y.\ y \in B \Longrightarrow \exists!x.\ x \in A \wedge h\ x = y$ **and** A: $\bigwedge x.\ x \in A \Longrightarrow h\ x \in B \wedge \gamma(h\ x) = \varphi\ x$
shows $F\ \varphi\ A = F\ \gamma\ B$
proof −
have *eq*: $B = h\ `\ A$
by (*auto dest*: *assms*)
have *h*: *inj-on h A*
using *assms* **by** (*blast intro*: *inj-onI*)
have $F\ \varphi\ A = F\ (\gamma \circ h)\ A$
using A **by** *auto*
also have $\ldots = F\ \gamma\ B$
by (*simp add*: *eq reindex h*)
finally show *?thesis* **.**
qed

lemma *eq-general-inverses*:
assumes B: $\bigwedge y.\ y \in B \Longrightarrow k\ y \in A \wedge h(k\ y) = y$ **and** A: $\bigwedge x.\ x \in A \Longrightarrow h\ x \in B \wedge k(h\ x) = x \wedge \gamma(h\ x) = \varphi\ x$
shows $F\ \varphi\ A = F\ \gamma\ B$
by (*rule eq-general* [**where** *h*=*h*]) (*force intro*: *dest*: *A B*)+

48.1.2 HOL Light variant: sum/product indexed by the non-neutral subset

NB only a subset of the properties above are proved

definition $G :: ['b \Rightarrow 'a, 'b\ set] \Rightarrow 'a$
where $G\ p\ I \equiv$ *if finite* $\{x \in I.\ p\ x \neq \mathbf{1}\}$ *then* $F\ p\ \{x \in I.\ p\ x \neq \mathbf{1}\}$ *else* $\mathbf{1}$

lemma *finite-Collect-op*:
shows $[\![$*finite* $\{i \in I.\ x\ i \neq \mathbf{1}\}$; *finite* $\{i \in I.\ y\ i \neq \mathbf{1}\}]\!] \Longrightarrow$ *finite* $\{i \in I.\ x\ i * y\ i \neq \mathbf{1}\}$

apply (*rule finite-subset* [**where** $B = \{i \in I.\ x\ i \neq \mathbf{1}\} \cup \{i \in I.\ y\ i \neq \mathbf{1}\}$])
using *left-neutral* **by** *force+*

lemma *empty'* [*simp*]: $G\ p\ \{\} = \mathbf{1}$
by (*auto simp*: *G-def*)

lemma *eq-sum* [*simp*]: *finite* $I \Longrightarrow G\ p\ I = F\ p\ I$
by (*auto simp*: *G-def intro*: *mono-neutral-cong-left*)

lemma *insert'* [*simp*]:
assumes *finite* $\{x \in I.\ p\ x \neq \mathbf{1}\}$
shows $G\ p\ (\mathit{insert}\ i\ I) = (\mathit{if}\ i \in I\ \mathit{then}\ G\ p\ I\ \mathit{else}\ p\ i * G\ p\ I)$
proof −
have $\{x.\ x = i \wedge p\ x \neq \mathbf{1} \vee x \in I \wedge p\ x \neq \mathbf{1}\} = (\mathit{if}\ p\ i = \mathbf{1}\ \mathit{then}\ \{x \in I.\ p\ x \neq \mathbf{1}\}\ \mathit{else}\ \mathit{insert}\ i\ \{x \in I.\ p\ x \neq \mathbf{1}\})$
by *auto*
then show *?thesis*
using *assms* **by** (*simp add*: *G-def conj-disj-distribR insert-absorb*)
qed

lemma *distrib-triv'*:
assumes *finite I*
shows $G\ (\lambda i.\ g\ i * h\ i)\ I = G\ g\ I * G\ h\ I$
by (*simp add*: *assms local.distrib*)

lemma *non-neutral'*: $G\ g\ \{x \in I.\ g\ x \neq \mathbf{1}\} = G\ g\ I$
by (*simp add*: *G-def*)

lemma *distrib'*:
assumes *finite* $\{x \in I.\ g\ x \neq \mathbf{1}\}$ *finite* $\{x \in I.\ h\ x \neq \mathbf{1}\}$
shows $G\ (\lambda i.\ g\ i * h\ i)\ I = G\ g\ I * G\ h\ I$
proof −
have $a * a \neq a \Longrightarrow a \neq \mathbf{1}$ **for** a
by *auto*
then have $G\ (\lambda i.\ g\ i * h\ i)\ I = G\ (\lambda i.\ g\ i * h\ i)\ (\{i \in I.\ g\ i \neq \mathbf{1}\} \cup \{i \in I.\ h\ i \neq \mathbf{1}\})$
using *assms* **by** (*force simp*: *G-def finite-Collect-op intro*!: *mono-neutral-cong*)
also have $\ldots = G\ g\ I * G\ h\ I$
proof −
have $F\ g\ (\{i \in I.\ g\ i \neq \mathbf{1}\} \cup \{i \in I.\ h\ i \neq \mathbf{1}\}) = G\ g\ I$
$F\ h\ (\{i \in I.\ g\ i \neq \mathbf{1}\} \cup \{i \in I.\ h\ i \neq \mathbf{1}\}) = G\ h\ I$
by (*auto simp*: *G-def assms intro*: *mono-neutral-right*)
then show *?thesis*
using *assms* **by** (*simp add*: *distrib*)
qed
finally show *?thesis* **.**
qed

lemma *cong'*:

assumes $A = B$
assumes *g-h*: $\bigwedge x.\ x \in B \Longrightarrow g\ x = h\ x$
shows $G\ g\ A = G\ h\ B$
using *assms* **by** (*auto simp*: *G-def cong*: *conj-cong intro*: *cong*)

lemma *mono-neutral-cong-left'*:
assumes $S \subseteq T$
and $\bigwedge i.\ i \in T - S \Longrightarrow h\ i = \mathbf{1}$
and $\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x$
shows $G\ g\ S = G\ h\ T$
proof −
have $*$: $\{x \in S.\ g\ x \neq \mathbf{1}\} = \{x \in T.\ h\ x \neq \mathbf{1}\}$
using *assms* **by** (*metis DiffI subset-eq*)
then have *finite* $\{x \in S.\ g\ x \neq \mathbf{1}\} =$ *finite* $\{x \in T.\ h\ x \neq \mathbf{1}\}$
by *simp*
then show *?thesis*
using *assms* **by** (*auto simp add*: *G-def* $*$ *intro*: *cong*)
qed

lemma *mono-neutral-cong-right'*:
$S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow (\bigwedge x.\ x \in S \Longrightarrow g\ x = h\ x) \Longrightarrow$
$G\ g\ T = G\ h\ S$
by (*auto intro*!: *mono-neutral-cong-left'* [*symmetric*])

lemma *mono-neutral-left'*: $S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow G\ g\ S = G\ g\ T$
by (*blast intro*: *mono-neutral-cong-left'*)

lemma *mono-neutral-right'*: $S \subseteq T \Longrightarrow \forall i \in T - S.\ g\ i = \mathbf{1} \Longrightarrow G\ g\ T = G\ g\ S$
by (*blast intro*!: *mono-neutral-left'* [*symmetric*])

end

48.2 Generalized summation over a set

context *comm-monoid-add*
begin

sublocale *sum*: *comm-monoid-set plus 0*
defines $sum = sum.F$ **and** $sum' = sum.G$ **..**

abbreviation *Sum* (‹$\sum$›)
where $\sum \equiv sum\ (\lambda x.\ x)$

end

Now: lots of fancy syntax. First, $sum\ (\lambda x.\ e)\ A$ is written $\sum x{\in}A.\ e$.

syntax (*ASCII*)
-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=3 notation=‹binder*

SUM››*SUM* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
syntax
-sum :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b*::*comm-monoid-add* (‹(‹*indent=2 notation=‹binder* ∑›› ∑ (-/∈-)./ -)› [*0*, *51*, *10*] *10*)

syntax-consts
-sum ⇌ *sum*

translations — Beware of argument permutation!
∑ *i*∈*A*. *b* ⇌ *CONST sum* (λ*i*. *b*) *A*

Instead of ∑ *x*∈{*x*. *P*}. *e* we introduce the shorter ∑ *x*|*P*. *e*.

syntax (*ASCII*)
-qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=3 notation=‹binder SUM Collect*›› *SUM - |/ -./ -)*› [*0*, *0*, *10*] *10*)
syntax
-qsum :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=2 notation=‹binder* ∑ *Collect*›› ∑ - | (-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-qsum == *sum*
translations
∑ *x*|*P*. *t* => *CONST sum* (λ*x*. *t*) {*x*. *P*}
print-translation ‹
[(***const-syntax*** ‹*sum*›, *K* (*Collect-binder-tr'* ***syntax-const*** ‹*-qsum*›))]
›

48.2.1 Properties in more restricted classes of structures

lemma *sum-Un*:
finite A ⟹ *finite B* ⟹ *sum f* (*A* ∪ *B*) = *sum f A* + *sum f B* − *sum f* (*A* ∩ *B*)
for *f* :: *'b* ⇒ *'a*::*ab-group-add*
by (*subst sum.union-inter* [*symmetric*]) (*auto simp add*: *algebra-simps*)

lemma *sum-Un2*:
assumes *finite* (*A* ∪ *B*)
shows *sum f* (*A* ∪ *B*) = *sum f* (*A* − *B*) + *sum f* (*B* − *A*) + *sum f* (*A* ∩ *B*)
proof −
have *A* ∪ *B* = *A* − *B* ∪ (*B* − *A*) ∪ *A* ∩ *B*
by *auto*
with *assms* **show** *?thesis*
by *simp* (*subst sum.union-disjoint*, *auto*)+
qed

lemma *sum-diff*:
fixes *f* :: *'b* ⇒ *'a*::*ab-group-add*
assumes *finite A B* ⊆ *A*
shows *sum f* (*A* − *B*) = *sum f A* − *sum f B*

using *sum.subset-diff* [*of B A f*] *assms* **by** *simp*

lemma *sum-diff1*:
fixes $f :: 'b \Rightarrow 'a$::*ab-group-add*
assumes *finite A*
shows *sum f* $(A - \{a\})$ = (*if* $a \in A$ *then sum f A* $-$ *f a else sum f A*)
using *assms* **by** (*simp add*: *sum-diff*)

lemma *sum-diff1'-aux*:
fixes $f :: 'a \Rightarrow 'b$::*ab-group-add*
assumes *finite F* $\{i \in I.\ f\ i \neq 0\} \subseteq F$
shows *sum' f* $(I - \{i\})$ = (*if* $i \in I$ *then sum' f I* $-$ *f i else sum' f I*)
using *assms*
proof *induct*
case (*insert x F*)
have *1*: *finite* $\{x \in I.\ f\ x \neq 0\} \Longrightarrow$ *finite* $\{x \in I.\ x \neq i \wedge f\ x \neq 0\}$
by (*erule rev-finite-subset*) *auto*
have *2*: *finite* $\{x \in I.\ x \neq i \wedge f\ x \neq 0\} \Longrightarrow$ *finite* $\{x \in I.\ f\ x \neq 0\}$
apply (*drule finite-insert* [*THEN iffD2*])
by (*erule rev-finite-subset*) *auto*
have *3*: *finite* $\{i \in I.\ f\ i \neq 0\}$
using *finite-subset insert* **by** *blast*
show *?case*
using *insert sum-diff1* [*of* $\{i \in I.\ f\ i \neq 0\}$ *f i*]
by (*auto simp*: *sum.G-def 1 2 3 set-diff-eq conj-ac*)
qed (*simp add*: *sum.G-def*)

lemma *sum-diff1'*:
fixes $f :: 'a \Rightarrow 'b$::*ab-group-add*
assumes *finite* $\{i \in I.\ f\ i \neq 0\}$
shows *sum' f* $(I - \{i\})$ = (*if* $i \in I$ *then sum' f I* $-$ *f i else sum' f I*)
by (*rule sum-diff1'-aux* [*OF assms order-refl*])

lemma (**in** *ordered-comm-monoid-add*) *sum-mono*:
$(\bigwedge i.\ i \in K \Longrightarrow f\ i \leq g\ i) \Longrightarrow (\sum i \in K.\ f\ i) \leq (\sum i \in K.\ g\ i)$
by (*induct K rule*: *infinite-finite-induct*) (*use add-mono* **in** *auto*)

lemma (**in** *ordered-cancel-comm-monoid-add*) *sum-strict-mono-strong*:
assumes *finite A* $a \in A$ *f a* $<$ *g a*
and $\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x$
shows *sum f A* $<$ *sum g A*
proof $-$
have *sum f A* = *f a* + *sum f* $(A - \{a\})$
by (*simp add*: *assms sum.remove*)
also have $\ldots \leq$ *f a* + *sum g* $(A - \{a\})$
using *assms* **by** (*meson DiffD1 add-left-mono sum-mono*)
also have $\ldots <$ *g a* + *sum g* $(A - \{a\})$
using *assms add-less-le-mono* **by** *blast*
also have $\ldots$ = *sum g A*

using *assms* **by** (*intro sum.remove* [*symmetric*])
finally show *?thesis* **.**
qed

lemma (**in** *strict-ordered-comm-monoid-add*) *sum-strict-mono*:
assumes *finite A* $A \neq \{\}$
and $\bigwedge x.\ x \in A \Longrightarrow f\ x < g\ x$
shows $sum\ f\ A < sum\ g\ A$
using *assms*
proof (*induct rule*: *finite-ne-induct*)
case *singleton*
then show *?case* **by** *simp*
next
case *insert*
then show *?case* **by** (*auto simp*: *add-strict-mono*)
qed

lemma *sum-strict-mono-ex1*:
fixes $f\ g :: 'i \Rightarrow 'a$::*ordered-cancel-comm-monoid-add*
assumes *finite A*
and $\forall\, x{\in}A.\ f\ x \leq g\ x$
and $\exists\, a{\in}A.\ f\ a < g\ a$
shows $sum\ f\ A < sum\ g\ A$
proof−
from *assms*(*3*) **obtain** *a* **where** *a*: $a \in A\ f\ a < g\ a$ **by** *blast*
have $sum\ f\ A = sum\ f\ ((A - \{a\}) \cup \{a\})$
by(*simp add*: *insert-absorb*[*OF* ‹$a \in A$›])
also have $\ldots = sum\ f\ (A - \{a\}) + sum\ f\ \{a\}$
using ‹*finite A*› **by**(*subst sum.union-disjoint*) *auto*
also have $sum\ f\ (A - \{a\}) \leq sum\ g\ (A - \{a\})$
by (*rule sum-mono*) (*simp add*: *assms*(*2*))
also from *a* **have** $sum\ f\ \{a\} < sum\ g\ \{a\}$ **by** *simp*
also have $sum\ g\ (A - \{a\}) + sum\ g\ \{a\} = sum\ g((A - \{a\}) \cup \{a\})$
using ‹*finite A*› **by** (*subst sum.union-disjoint*[*symmetric*]) *auto*
also have $\ldots = sum\ g\ A$ **by** (*simp add*: *insert-absorb*[*OF* ‹$a \in A$›])
finally show *?thesis*
by (*auto simp add*: *add-right-mono add-strict-left-mono*)
qed

lemma *sum-mono-inv*:
fixes $f\ g :: 'i \Rightarrow 'a$ :: *ordered-cancel-comm-monoid-add*
assumes *eq*: $sum\ f\ I = sum\ g\ I$
assumes *le*: $\bigwedge i.\ i \in I \Longrightarrow f\ i \leq g\ i$
assumes *i*: $i \in I$
assumes *I*: *finite I*
shows $f\ i = g\ i$
proof (*rule ccontr*)
assume ¬ *?thesis*
with *le*[*OF i*] **have** $f\ i < g\ i$ **by** *simp*

with *i* **have** $\exists\, i{\in}I.\ f\ i < g\ i$ **..**
from *sum-strict-mono-ex1*[*OF I - this*] *le* **have** *sum f I < sum g I*
by *blast*
with *eq* **show** *False* **by** *simp*
qed

lemma *member-le-sum*:
fixes *f* :: *- ⇒ 'b*::{*semiring-1*, *ordered-comm-monoid-add*}
assumes $i \in A$
and *le*: $\bigwedge x.\ x \in A - \{i\} \Longrightarrow 0 \le f\ x$
and *finite A*
shows $f\ i \le$ *sum f A*
proof −
have $f\ i \le$ *sum f* $(A \cap \{i\})$
by (*simp add*: *assms*)
also have ... = $(\sum x{\in}A.$ *if* $x \in \{i\}$ *then f x else 0*)
using *assms sum.inter-restrict* **by** *blast*
also have ... $\le$ *sum f A*
apply (*rule sum-mono*)
apply (*auto simp*: *le*)
done
finally show *?thesis* **.**
qed

lemma *sum-negf*: $(\sum x{\in}A.\ -\ f\ x) = -\ (\sum x{\in}A.\ f\ x)$
for *f* :: *'b ⇒ 'a*::*ab-group-add*
by (*induct A rule*: *infinite-finite-induct*) *auto*

lemma *sum-subtractf*: $(\sum x{\in}A.\ f\ x\ -\ g\ x) = (\sum x{\in}A.\ f\ x)\ -\ (\sum x{\in}A.\ g\ x)$
for *f g* :: *'b ⇒'a*::*ab-group-add*
using *sum.distrib* [*of f − g A*] **by** (*simp add*: *sum-negf*)

lemma *sum-subtractf-nat*:
$(\bigwedge x.\ x \in A \Longrightarrow g\ x \le f\ x) \Longrightarrow (\sum x{\in}A.\ f\ x\ -\ g\ x) = (\sum x{\in}A.\ f\ x)\ -\ (\sum x{\in}A.\ g\ x)$
for *f g* :: *'a ⇒ nat*
by (*induct A rule*: *infinite-finite-induct*) (*auto simp*: *sum-mono*)

context *ordered-comm-monoid-add*
begin

lemma *sum-nonneg*: $(\bigwedge x.\ x \in A \Longrightarrow 0 \le f\ x) \Longrightarrow 0 \le$ *sum f A*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next

case (*insert x F*)
then have *0 + 0 ≤ f x + sum f F* **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed

lemma *sum-nonpos*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \leq 0) \Longrightarrow sum\ f\ A \leq 0$
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then have *f x + sum f F ≤ 0 + 0* **by** (*blast intro*: *add-mono*)
with *insert* **show** *?case* **by** *simp*
qed

lemma *sum-nonneg-eq-0-iff*:
finite A $\Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow 0 \leq f\ x) \Longrightarrow sum\ f\ A = 0 \longleftrightarrow (\forall x{\in}A.\ f\ x = 0)$
by (*induct set*: *finite*) (*simp-all add*: *add-nonneg-eq-0-iff sum-nonneg*)

lemma *sum-nonneg-0*:
finite s $\Longrightarrow (\bigwedge i.\ i \in s \Longrightarrow f\ i \geq 0) \Longrightarrow (\sum\ i \in s.\ f\ i) = 0 \Longrightarrow i \in s \Longrightarrow f\ i = 0$
by (*simp add*: *sum-nonneg-eq-0-iff*)

lemma *sum-nonneg-leq-bound*:
assumes *finite s* $\bigwedge i.\ i \in s \Longrightarrow f\ i \geq 0$ $(\sum i \in s.\ f\ i) = B$ $i \in s$
shows $f\ i \leq B$
proof −
from *assms* **have** $f\ i \leq f\ i + (\sum i \in s - \{i\}.\ f\ i)$
by (*intro add-increasing2 sum-nonneg*) *auto*
also have $\ldots = B$
using *sum.remove*[*of s i f*] *assms* **by** *simp*
finally show *?thesis* **by** *auto*
qed

lemma *sum-mono2*:
assumes *fin*: *finite B*
and *sub*: $A \subseteq B$
and *nn*: $\bigwedge b.\ b \in B{-}A \Longrightarrow 0 \leq f\ b$
shows *sum f A ≤ sum f B*
proof −
have *sum f A ≤ sum f A + sum f* $(B{-}A)$
by (*auto intro*: *add-increasing2* [*OF sum-nonneg*] *nn*)
also from *fin finite-subset*[*OF sub fin*] **have** $\ldots$ *= sum f* $(A \cup (B{-}A))$
by (*simp add*: *sum.union-disjoint del*: *Un-Diff-cancel*)
also from *sub* **have** $A \cup (B{-}A) = B$ **by** *blast*
finally show *?thesis* **.**

qed

lemma *sum-le-included*:
assumes *finite s finite t*
and $\forall\, y{\in}t.\ 0 \le g\ y\ (\forall\, x{\in}s.\ \exists\, y{\in}t.\ i\ y = x \wedge f\ x \le g\ y)$
shows *sum f s* $\le$ *sum g t*
proof −
have *sum f s* $\le$ *sum* $(\lambda y.$ *sum g* $\{x.\ x{\in}t \wedge i\ x = y\})$ *s*
proof (*rule sum-mono*)
fix *y*
assume $y \in s$
with *assms* **obtain** *z* **where** *z*: $z \in t\ y = i\ z\ f\ y \le g\ z$ **by** *auto*
with *assms* **show** $f\ y \le$ *sum g* $\{x \in t.\ i\ x = y\}$ (**is** *?A y* $\le$ *?B y*)
using *order-trans*[*of ?A* (*i z*) *sum g* $\{z\}$ *?B* (*i z*), *intro*]
by (*auto intro*!: *sum-mono2*)
qed
also have ... $\le$ *sum* $(\lambda y.$ *sum g* $\{x.\ x{\in}t \wedge i\ x = y\})$ (*i* ' *t*)
using *assms*(*2*−*4*) **by** (*auto intro*!: *sum-mono2 sum-nonneg*)
also have ... $\le$ *sum g t*
using *assms* **by** (*auto simp*: *sum.image-gen*[*symmetric*])
finally show *?thesis* **.**
qed

end

lemma (**in** *canonically-ordered-monoid-add*) *sum-eq-0-iff* [*simp*]:
finite F $\Longrightarrow$ (*sum f F* = *0*) = $(\forall\, a{\in}F.\ f\ a = 0)$
by (*intro ballI sum-nonneg-eq-0-iff zero-le*)

context *semiring-0*
begin

lemma *sum-distrib-left*: $r * sum\ f\ A = (\sum n{\in}A.\ r * f\ n)$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *algebra-simps*)

lemma *sum-distrib-right*: $sum\ f\ A * r = (\sum n{\in}A.\ f\ n * r)$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *algebra-simps*)

end

lemma *sum-divide-distrib*: $sum\ f\ A\ /\ r = (\sum n{\in}A.\ f\ n\ /\ r)$
for *r* :: *'a*::*field*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next

case *insert*
then show *?case* **by** (*simp add*: *add-divide-distrib*)
qed

lemma *sum-abs*[*iff*]: $|sum\ f\ A| \leq sum\ (\lambda i.\ |f\ i|)\ A$
for $f :: 'a \Rightarrow 'b$::*ordered-ab-group-add-abs*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case *insert*
then show *?case* **by** (*auto intro*: *abs-triangle-ineq order-trans*)
qed

lemma *sum-abs-ge-zero*[*iff*]: $0 \leq sum\ (\lambda i.\ |f\ i|)\ A$
for $f :: 'a \Rightarrow 'b$::*ordered-ab-group-add-abs*
by (*simp add*: *sum-nonneg*)

lemma *abs-sum-abs*[*simp*]: $|\sum a{\in}A.\ |f\ a|| = (\sum a{\in}A.\ |f\ a|)$
for $f :: 'a \Rightarrow 'b$::*ordered-ab-group-add-abs*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
then have $|\sum a{\in}insert\ a\ A.\ |f\ a|| = ||f\ a| + (\sum a{\in}A.\ |f\ a|)|$ **by** *simp*
also from *insert* **have** $\ldots = ||f\ a| + |\sum a{\in}A.\ |f\ a|||$ **by** *simp*
also have $\ldots = |f\ a| + |\sum a{\in}A.\ |f\ a||$ **by** (*simp del*: *abs-of-nonneg*)
also from *insert* **have** $\ldots = (\sum a{\in}insert\ a\ A.\ |f\ a|)$ **by** *simp*
finally show *?case* **.**
qed

lemma *sum-product*:
fixes $f :: 'a \Rightarrow 'b$::*semiring-0*
shows $sum\ f\ A * sum\ g\ B = (\sum i{\in}A.\ \sum j{\in}B.\ f\ i * g\ j)$
by (*simp add*: *sum-distrib-left sum-distrib-right*) (*rule sum.swap*)

lemma *sum-mult-sum-if-inj*:
fixes $f :: 'a \Rightarrow 'b$::*semiring-0*
shows *inj-on* $(\lambda(a,\ b).\ f\ a * g\ b)\ (A \times B) \Longrightarrow$
$sum\ f\ A * sum\ g\ B = sum\ id\ \{f\ a * g\ b\ |a\ b.\ a \in A \wedge b \in B\}$
by(*auto simp*: *sum-product sum.cartesian-product intro*!: *sum.reindex-cong*[*symmetric*])

lemma *sum-SucD*: *sum f A = Suc n ⟹ ∃ a∈A. 0 < f a*
by (*induct A rule*: *infinite-finite-induct*) *auto*

lemma *sum-eq-Suc0-iff*:
finite A ⟹ sum f A = Suc 0 ⟷ (∃ a∈A. f a = Suc 0 ∧ (∀ b∈A. a ≠ b ⟶ f b = 0))
by (*induct A rule*: *finite-induct*) (*auto simp add*: *add-is-1*)

lemmas *sum-eq-1-iff = sum-eq-Suc0-iff*[*simplified One-nat-def*[*symmetric*]]

lemma *sum-Un-nat*:
finite A ⟹ finite B ⟹ sum f (A ∪ B) = sum f A + sum f B − sum f (A ∩ B)
for *f* :: *'a ⇒ nat*
— For the natural numbers, we have subtraction.
by (*subst sum.union-inter* [*symmetric*]) (*auto simp*: *algebra-simps*)

lemma *sum-diff1-nat*: *sum f (A − {a}) = (if a ∈ A then sum f A − f a else sum f A)*
for *f* :: *'a ⇒ nat*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then show *?case*
proof (*cases a ∈ F*)
case *True*
then have *∃ B. F = insert a B ∧ a ∉ B*
by (*auto simp*: *mk-disjoint-insert*)
then show *?thesis* **using** *insert*
by (*auto simp*: *insert-Diff-if*)
qed (*auto*)
qed

lemma *sum-diff-nat*:
fixes *f* :: *'a ⇒ nat*
assumes *finite B* **and** *B ⊆ A*
shows *sum f (A − B) = sum f A − sum f B*
using *assms*
proof *induct*
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
note *IH = ‹F ⊆ A ⟹ sum f (A − F) = sum f A − sum f F›*
from *‹x ∉ F›* *‹insert x F ⊆ A›* **have** *x ∈ A − F* **by** *simp*

then have *A*: *sum f* ((*A* − *F*) − {*x*}) = *sum f* (*A* − *F*) − *f x*
by (*simp add*: *sum-diff1-nat*)
from ‹*insert x F* ⊆ *A*› **have** *F* ⊆ *A* **by** *simp*
with *IH* **have** *sum f* (*A* − *F*) = *sum f A* − *sum f F* **by** *simp*
with *A* **have** *B*: *sum f* ((*A* − *F*) − {*x*}) = *sum f A* − *sum f F* − *f x*
by *simp*
from ‹*x* ∉ *F*› **have** *A* − *insert x F* = (*A* − *F*) − {*x*} **by** *auto*
with *B* **have** *C*: *sum f* (*A* − *insert x F*) = *sum f A* − *sum f F* − *f x*
by *simp*
from ‹*finite F*› ‹*x* ∉ *F*› **have** *sum f* (*insert x F*) = *sum f F* + *f x*
by *simp*
with *C* **have** *sum f* (*A* − *insert x F*) = *sum f A* − *sum f* (*insert x F*)
by *simp*
then show *?case* **by** *simp*
qed

lemma *sum-comp-morphism*:
h 0 = *0* ⟹ (⋀*x y*. *h* (*x* + *y*) = *h x* + *h y*) ⟹ *sum* (*h* ∘ *g*) *A* = *h* (*sum g A*)
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma (**in** *comm-semiring-1*) *dvd-sum*: (⋀*a*. *a* ∈ *A* ⟹ *d dvd f a*) ⟹ *d dvd sum f A*
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma (**in** *ordered-comm-monoid-add*) *sum-pos*:
finite I ⟹ *I* ≠ {} ⟹ (⋀*i*. *i* ∈ *I* ⟹ *0* < *f i*) ⟹ *0* < *sum f I*
by (*induct I rule*: *finite-ne-induct*) (*auto intro*: *add-pos-pos*)

lemma (**in** *ordered-comm-monoid-add*) *sum-pos2*:
assumes *I*: *finite I i* ∈ *I* *0* < *f i* ⋀*i*. *i* ∈ *I* ⟹ *0* ≤ *f i*
shows *0* < *sum f I*
proof −
have *0* < *f i* + *sum f* (*I* − {*i*})
using *assms* **by** (*intro add-pos-nonneg sum-nonneg*) *auto*
also have ... = *sum f I*
using *assms* **by** (*simp add*: *sum.remove*)
finally show *?thesis* **.**
qed

lemma *sum-strict-mono2*:
fixes *f* :: *'a* ⇒ *'b*::*ordered-cancel-comm-monoid-add*
assumes *finite B A* ⊆ *B b* ∈ *B*−*A f b* > *0* **and** ⋀*x*. *x* ∈ *B* ⟹ *f x* ≥ *0*
shows *sum f A* < *sum f B*
proof −
have *B* − *A* ≠ {}
using *assms*(*3*) **by** *blast*
have *sum f* (*B*−*A*) > *0*
by (*rule sum-pos2*) (*use assms* **in** *auto*)
moreover have *sum f B* = *sum f* (*B*−*A*) + *sum f A*

by (*rule sum.subset-diff*) (*use assms* **in** *auto*)
ultimately show *?thesis*
using *add-strict-increasing* **by** *auto*
qed

lemma *sum-cong-Suc*:
assumes $0 \notin A$ $\bigwedge x.\ Suc\ x \in A \Longrightarrow f\ (Suc\ x) = g\ (Suc\ x)$
shows $sum\ f\ A = sum\ g\ A$
proof (*rule sum.cong*)
fix x
assume $x \in A$
with *assms*(*1*) **show** $f\ x = g\ x$
by (*cases x*) (*auto intro*!: *assms*(*2*))
qed *simp-all*

48.2.2 Cardinality as special case of *sum*

lemma *card-eq-sum*: $card\ A = sum\ (\lambda x.\ 1)\ A$
proof −
have $plus \circ (\lambda\text{-}.\ Suc\ 0) = (\lambda\text{-}.\ Suc)$
by (*simp add*: *fun-eq-iff*)
then have $Finite\text{-}Set.fold\ (plus \circ (\lambda\text{-}.\ Suc\ 0)) = Finite\text{-}Set.fold\ (\lambda\text{-}.\ Suc)$
by (*rule arg-cong*)
then have $Finite\text{-}Set.fold\ (plus \circ (\lambda\text{-}.\ Suc\ 0))\ 0\ A = Finite\text{-}Set.fold\ (\lambda\text{-}.\ Suc)\ 0\ A$
by (*blast intro*: *fun-cong*)
then show *?thesis*
by (*simp add*: *card.eq-fold sum.eq-fold*)
qed

context *semiring-1*
begin

lemma *sum-constant* [*simp*]:
$(\sum x \in A.\ y) = of\text{-}nat\ (card\ A) * y$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *algebra-simps*)

context
fixes A
assumes ‹*finite A*›
begin

lemma *sum-of-bool-eq* [*simp*]:
‹$(\sum x \in A.\ of\text{-}bool\ (P\ x)) = of\text{-}nat\ (card\ (A \cap \{x.\ P\ x\}))$› **if** ‹*finite A*›
using ‹*finite A*› **by** *induction simp-all*

lemma *sum-mult-of-bool-eq* [*simp*]:
‹$(\sum x \in A.\ f\ x * of\text{-}bool\ (P\ x)) = (\sum x \in (A \cap \{x.\ P\ x\}).\ f\ x)$›
by (*rule sum.mono-neutral-cong*) (*use* ‹*finite A*› **in** *auto*)

lemma *sum-of-bool-mult-eq* [*simp*]:
‹$(\sum x \in A.\ of\text{-}bool\ (P\ x) * f\ x) = (\sum x \in (A \cap \{x.\ P\ x\}).\ f\ x)$›
by (*rule sum.mono-neutral-cong*) (*use* ‹*finite A*› **in** *auto*)

end

end

lemma *sum-Suc*: *sum* $(\lambda x.\ Suc(f\ x))\ A = sum\ f\ A + card\ A$
using *sum.distrib*[*of f* λ-. *1 A*] **by** *simp*

lemma *sum-bounded-above*:
fixes *K* :: *'a*::{*semiring-1*,*ordered-comm-monoid-add*}
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq K$
shows $sum\ f\ A \leq of\text{-}nat\ (card\ A) * K$
proof (*cases finite A*)
case *True*
then show *?thesis*
using *le sum-mono*[**where** *K*=*A* **and** $g = \lambda x.\ K$] **by** *simp*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *sum-bounded-above-divide*:
fixes *K* :: *'a*::*linordered-field*
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow f\ i \leq K\ /\ of\text{-}nat\ (card\ A)$ **and** *fin*: *finite A* $A \neq \{\}$
shows $sum\ f\ A \leq K$
using *sum-bounded-above* [*of A f K* / *of-nat* (*card A*), *OF le*] *fin* **by** *simp*

lemma *sum-bounded-above-strict*:
fixes *K* :: *'a*::{*ordered-cancel-comm-monoid-add*,*semiring-1*}
assumes $\bigwedge i.\ i{\in}A \Longrightarrow f\ i < K\ card\ A > 0$
shows $sum\ f\ A < of\text{-}nat\ (card\ A) * K$
using *assms sum-strict-mono*[**where** *A*=*A* **and** $g = \lambda x.\ K$]
by (*simp add*: *card-gt-0-iff*)

lemma *sum-bounded-below*:
fixes *K* :: *'a*::{*semiring-1*,*ordered-comm-monoid-add*}
assumes *le*: $\bigwedge i.\ i{\in}A \Longrightarrow K \leq f\ i$
shows $of\text{-}nat\ (card\ A) * K \leq sum\ f\ A$
proof (*cases finite A*)
case *True*
then show *?thesis*
using *le sum-mono*[**where** *K*=*A* **and** $f = \lambda x.\ K$] **by** *simp*
next
case *False*
then show *?thesis* **by** *simp*

qed

lemma *convex-sum-bound-le*:
fixes $x :: 'a \Rightarrow 'b$::*linordered-idom*
assumes *0*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq x\ i$ **and** *1*: $sum\ x\ I = 1$
and δ: $\bigwedge i.\ i \in I \Longrightarrow |a\ i - b| \leq \delta$
shows $|(\sum i{\in}I.\ a\ i * x\ i) - b| \leq \delta$
proof −
have [*simp*]: $(\sum i{\in}I.\ c * x\ i) = c$ **for** *c*
by (*simp flip*: *sum-distrib-left 1*)
then have $|(\sum i{\in}I.\ a\ i * x\ i) - b| = |\sum i{\in}I.\ (a\ i - b) * x\ i|$
by (*simp add*: *sum-subtractf left-diff-distrib*)
also have $\ldots \leq (\sum i{\in}I.\ |(a\ i - b) * x\ i|)$
using *abs-abs abs-of-nonneg* **by** *blast*
also have $\ldots \leq (\sum i{\in}I.\ |(a\ i - b)| * x\ i)$
by (*simp add*: *abs-mult 0*)
also have $\ldots \leq (\sum i{\in}I.\ \delta * x\ i)$
by (*rule sum-mono*) (*use* δ *0 mult-right-mono* **in** *blast*)
also have $\ldots = \delta$
by *simp*
finally show *?thesis* **.**
qed

lemma *card-UN-disjoint*:
assumes *finite I* **and** $\forall i{\in}I.$ *finite* $(A\ i)$
and $\forall i{\in}I.\ \forall j{\in}I.\ i \neq j \longrightarrow A\ i \cap A\ j = \{\}$
shows *card* $(\bigcup (A\ `\ I)) = (\sum i{\in}I.\ card(A\ i))$
proof −
have $(\sum i{\in}I.\ card\ (A\ i)) = (\sum i{\in}I.\ \sum x{\in}A\ i.\ 1)$
by *simp*
with *assms* **show** *?thesis*
by (*simp add*: *card-eq-sum sum.UNION-disjoint del*: *sum-constant*)
qed

lemma *card-Union-disjoint*:
assumes *pairwise disjnt C* **and** *fin*: $\bigwedge A.\ A \in C \Longrightarrow$ *finite A*
shows *card* $(\bigcup C) =$ *sum card C*
proof (*cases finite C*)
case *True*
then show *?thesis*
using *card-UN-disjoint* [*OF True*, *of* $\lambda x.\ x$] *assms*
by (*simp add*: *disjnt-def fin pairwise-def*)
next
case *False*
then show *?thesis*
using *assms card-eq-0-iff finite-UnionD* **by** *fastforce*
qed

lemma *card-Union-le-sum-card-weak*:

fixes $U :: 'a\ set\ set$
assumes $\forall u \in U.\ finite\ u$
shows $card\ (\bigcup U) \leq sum\ card\ U$
proof (*cases finite U*)
case *False*
then show $card\ (\bigcup U) \leq sum\ card\ U$
using *card-eq-0-iff finite-UnionD* **by** *auto*
next
case *True*
then show $card\ (\bigcup U) \leq sum\ card\ U$
proof (*induct U rule*: *finite-induct*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert x F*)
then have $card(\bigcup (insert\ x\ F)) \leq card(x) + card\ (\bigcup F)$ **using** *card-Un-le* **by** *auto*
also have $... \leq card(x) + sum\ card\ F$ **using** *insert.hyps* **by** *auto*
also have $... = sum\ card\ (insert\ x\ F)$ **using** *sum.insert-if* **and** *insert.hyps* **by** *auto*
finally show *?case* **.**
qed
qed

lemma *card-Union-le-sum-card*:
fixes $U :: 'a\ set\ set$
shows $card\ (\bigcup U) \leq sum\ card\ U$
by (*metis Union-upper card.infinite card-Union-le-sum-card-weak finite-subset zero-le*)

lemma *card-UN-le*:
assumes *finite I*
shows $card(\bigcup i{\in}I.\ A\ i) \leq (\sum i{\in}I.\ card(A\ i))$
using *assms*
proof *induction*
case (*insert i I*)
then show *?case*
using *card-Un-le nat-add-left-cancel-le* **by** (*force intro*: *order-trans*)
qed *auto*

lemma *card-quotient-disjoint*:
assumes *finite A inj-on* $(\lambda x.\ \{x\}\ //\ r)\ A$
shows $card\ (A//r) = card\ A$
proof −
have $\forall i{\in}A.\ \forall j{\in}A.\ i \neq j \longrightarrow r\ ``\ \{j\} \neq r\ ``\ \{i\}$
using *assms* **by** (*fastforce simp add*: *quotient-def inj-on-def*)
with *assms* **show** *?thesis*
by (*simp add*: *quotient-def card-UN-disjoint*)
qed

lemma *sum-multicount-gen*:
assumes *finite s finite t* $\forall j{\in}t.$ (*card* $\{i{\in}s.\ R\ i\ j\} = k\ j$)
shows *sum* ($\lambda i.$ (*card* $\{j{\in}t.\ R\ i\ j\}$)) s = *sum* $k\ t$
(**is** *?l* = *?r*)
proof−
have *?l* = *sum* ($\lambda i.$ *sum* ($\lambda x.1$) $\{j{\in}t.\ R\ i\ j\}$) s
by *auto*
also have ... = *?r*
unfolding *sum.swap-restrict* [*OF assms*(*1*−*2*)]
using *assms*(*3*) **by** *auto*
finally show *?thesis* **.**
qed

lemma *sum-multicount*:
assumes *finite S finite T* $\forall j{\in}T.$ (*card* $\{i{\in}S.\ R\ i\ j\} = k$)
shows *sum* ($\lambda i.$ *card* $\{j{\in}T.\ R\ i\ j\}$) $S = k * card\ T$ (**is** *?l* = *?r*)
proof−
have *?l* = *sum* ($\lambda i.$ k) T
by (*rule sum-multicount-gen*) (*auto simp*: *assms*)
also have ... = *?r* **by** (*simp add*: *mult.commute*)
finally show *?thesis* **by** *auto*
qed

lemma *sum-card-image*:
assumes *finite A*
assumes *pairwise* ($\lambda s\ t.$ *disjnt* ($f\ s$) ($f\ t$)) A
shows *sum card* (f ‘ A) = *sum* ($\lambda a.$ *card* ($f\ a$)) A
using *assms*
proof (*induct A*)
case (*insert a A*)
show *?case*
proof *cases*
assume $f\ a = \{\}$
with *insert* **show** *?case*
by (*subst sum.mono-neutral-right*[**where** *S*=f ‘ A]) (*auto simp*: *pairwise-insert*)
next
assume $f\ a \neq \{\}$
then have *sum card* (*insert* ($f\ a$) (f ‘ A)) = *card* ($f\ a$) + *sum card* (f ‘ A)
using *insert*
by (*subst sum.insert*) (*auto simp*: *pairwise-insert*)
with *insert* **show** *?case* **by** (*simp add*: *pairwise-insert*)
qed
qed *simp*

By Jakub Kdzioka:

lemma *sum-fun-comp*:
assumes *finite S finite R g* ‘ $S \subseteq R$
shows $(\sum x \in S.\ f\ (g\ x)) = (\sum y \in R.$ *of-nat* (*card* $\{x \in S.\ g\ x = y\}$) $*\ f\ y)$

proof −
let *?r* = *relation-of* (λ*p q*. *g p* = *g q*) *S*
have *eqv*: *equiv S ?r*
unfolding *relation-of-def* **by** (*auto intro*: *comp-equivI*)
have *finite*: *C* ∈ *S*//*?r* ⟹ *finite C* **for** *C*
by (*fact finite-equiv-class*[*OF* ‹*finite S*› *equiv-type*[*OF* ‹*equiv S ?r*›]])
have *disjoint*: *A* ∈ *S*//*?r* ⟹ *B* ∈ *S*//*?r* ⟹ *A* ≠ *B* ⟹ *A* ∩ *B* = {} **for** *A B*
using *eqv quotient-disj* **by** *blast*

let *?cls* = λ*y*. {*x* ∈ *S*. *y* = *g x*}
have *quot-as-img*: *S*//*?r* = *?cls* ‘ *g* ‘ *S*
by (*auto simp add*: *relation-of-def quotient-def*)
have *cls-inj*: *inj-on ?cls* (*g* ‘ *S*)
by (*auto intro*: *inj-onI*)

have *rest-0*: (∑ *y* ∈ *R* − *g* ‘ *S*. *of-nat* (*card* (*?cls y*)) ∗ *f y*) = *0*
proof −
have *of-nat* (*card* (*?cls y*)) ∗ *f y* = *0* **if** *asm*: *y* ∈ *R* − *g* ‘ *S* **for** *y*
proof −
from *asm* **have** ∗: *?cls y* = {} **by** *auto*
show *?thesis* **unfolding** ∗ **by** *simp*
qed
thus *?thesis* **by** *simp*
qed

have (∑ *x* ∈ *S*. *f* (*g x*)) = (∑ *C* ∈ *S*//*?r*. ∑ *x* ∈ *C*. *f* (*g x*))
using *eqv finite disjoint*
by (*simp flip*: *sum.Union-disjoint*[*simplified*] *add*: *Union-quotient*)
also have ... = (∑ *y* ∈ *g* ‘ *S*. ∑ *x* ∈ *?cls y*. *f* (*g x*))
unfolding *quot-as-img* **by** (*simp add*: *sum.reindex*[*OF cls-inj*])
also have ... = (∑ *y* ∈ *g* ‘ *S*. ∑ *x* ∈ *?cls y*. *f y*)
by *auto*
also have ... = (∑ *y* ∈ *g* ‘ *S*. *of-nat* (*card* (*?cls y*)) ∗ *f y*)
by (*simp flip*: *sum-constant*)
also have ... = (∑ *y* ∈ *R*. *of-nat* (*card* (*?cls y*)) ∗ *f y*)
using *rest-0* **by** (*simp add*: *sum.subset-diff*[*OF* ‹*g* ‘ *S* ⊆ *R*› ‹*finite R*›])
finally show *?thesis*
by (*simp add*: *eq-commute*)
qed

48.2.3 Cardinality of products

lemma *card-SigmaI* [*simp*]:
finite A ⟹ ∀ *a*∈*A*. *finite* (*B a*) ⟹ *card* (*SIGMA x*: *A*. *B x*) = (∑ *a*∈*A*. *card* (*B a*))
by (*simp add*: *card-eq-sum sum.Sigma del*: *sum-constant*)

lemma *card-cartesian-product*: *card* $(A \times B) = card\ A * card\ B$
by (*cases finite* $A \wedge$ *finite* B)
(*auto simp add*: *card-eq-0-iff dest*: *finite-cartesian-productD1 finite-cartesian-productD2*)

lemma *card-cartesian-product-singleton*: *card* $(\{x\} \times A) = card\ A$
by (*simp add*: *card-cartesian-product*)

48.3 Generalized product over a set

context *comm-monoid-mult*
begin

sublocale *prod*: *comm-monoid-set times 1*
defines *prod* = *prod.F* **and** *prod'* = *prod.G* **..**

abbreviation *Prod* (‹$\prod$›)
where $\prod \equiv prod\ (\lambda x.\ x)$

end

syntax (*ASCII*)
-prod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* (‹(‹*indent=4 notation=‹binder PROD››PROD* (-/:-)./ -)› [*0*, *51*, *10*] *10*)
syntax
-prod :: *pttrn* => *'a set* => *'b* => *'b*::*comm-monoid-mult* (‹(‹*indent=2 notation=‹binder* $\prod$ *››*$\prod$ (-/∈-)./ -)› [*0*, *51*, *10*] *10*)

syntax-consts
-prod == *prod*

translations — Beware of argument permutation!
$\prod i{\in}A.\ b$ == *CONST prod* $(\lambda i.\ b)\ A$

Instead of $\prod x{\in}\{x.\ P\}.\ e$ we introduce the shorter $\prod x|P.\ e$.

syntax (*ASCII*)
-qprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=4 notation=‹binder PROD Collect››PROD* - |/ -./ -)› [*0*, *0*, *10*] *10*)
syntax
-qprod :: *pttrn* ⇒ *bool* ⇒ *'a* ⇒ *'a* (‹(‹*indent=2 notation=‹binder* $\prod$ *Collect››*$\prod$ - | (-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-qprod == *prod*
translations
$\prod x|P.\ t$ => *CONST prod* $(\lambda x.\ t)\ \{x.\ P\}$
print-translation ‹
[(***const-syntax*** ‹*prod*›, *K* (*Collect-binder-tr'* ***syntax-const*** ‹*-qprod*›))]
›

context *comm-monoid-mult*
begin

lemma *prod-dvd-prod*: $(\bigwedge a.\ a \in A \Longrightarrow f\ a\ dvd\ g\ a) \Longrightarrow prod\ f\ A\ dvd\ prod\ g\ A$
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** (*auto intro*: *dvdI*)
next
case *empty*
then show *?case* **by** (*auto intro*: *dvdI*)
next
case (*insert a A*)
then have *f a dvd g a* **and** *prod f A dvd prod g A*
by *simp-all*
then obtain *r s* **where** $g\ a = f\ a * r$ **and** $prod\ g\ A = prod\ f\ A * s$
by (*auto elim*!: *dvdE*)
then have $g\ a * prod\ g\ A = f\ a * prod\ f\ A * (r * s)$
by (*simp add*: *ac-simps*)
with *insert.hyps* **show** *?case*
by (*auto intro*: *dvdI*)
qed

lemma *prod-dvd-prod-subset*: *finite B* $\Longrightarrow$ $A \subseteq B \Longrightarrow$ *prod f A dvd prod f B*
by (*auto simp add*: *prod.subset-diff ac-simps intro*: *dvdI*)

end

48.3.1 Properties in more restricted classes of structures

context *linordered-nonzero-semiring*
begin

lemma *prod-ge-1*: $(\bigwedge x.\ x \in A \Longrightarrow 1 \leq f\ x) \Longrightarrow 1 \leq prod\ f\ A$
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
have $1 * 1 \leq f\ x * prod\ f\ F$
by (*rule mult-mono'*) (*use insert* **in** *auto*)
with *insert* **show** *?case* **by** *simp*
qed

lemma *prod-le-1*:
fixes $f :: 'b \Rightarrow 'a$
assumes $\bigwedge x.\ x \in A \Longrightarrow 0 \leq f\ x \wedge f\ x \leq 1$

shows *prod f A* ≤ *1*
using *assms*
proof (*induct A rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert x F*)
then show *?case* **by** (*force simp*: *mult.commute intro*: *dest*: *mult-le-one*)
qed

end

context *comm-semiring-1*
begin

lemma *dvd-prod-eqI* [*intro*]:
assumes *finite A* **and** *a* ∈ *A* **and** *b* = *f a*
shows *b dvd prod f A*
proof −
from ‹*finite A*› **have** *prod f* (*insert a* (*A* − {*a*})) = *f a* ∗ *prod f* (*A* − {*a*})
by (*intro prod.insert*) *auto*
also from ‹*a* ∈ *A*› **have** *insert a* (*A* − {*a*}) = *A*
by *blast*
finally have *prod f A* = *f a* ∗ *prod f* (*A* − {*a*}) **.**
with ‹*b* = *f a*› **show** *?thesis*
by *simp*
qed

lemma *dvd-prodI* [*intro*]: *finite A* ⟹ *a* ∈ *A* ⟹ *f a dvd prod f A*
by *auto*

lemma *prod-zero*:
assumes *finite A* **and** ∃ *a*∈*A*. *f a* = *0*
shows *prod f A* = *0*
using *assms*
proof (*induct A*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
then have *f a* = *0* ∨ (∃ *a*∈*A*. *f a* = *0*) **by** *simp*
then have *f a* ∗ *prod f A* = *0* **by** (*rule disjE*) (*simp-all add*: *insert*)
with *insert* **show** *?case* **by** *simp*
qed

lemma *prod-dvd-prod-subset2*:

assumes *finite B* **and** $A \subseteq B$ **and** $\bigwedge a.\ a \in A \Longrightarrow f\ a\ dvd\ g\ a$
shows *prod f A dvd prod g B*
proof −
from *assms* **have** *prod f A dvd prod g A*
by (*auto intro*: *prod-dvd-prod*)
moreover from *assms* **have** *prod g A dvd prod g B*
by (*auto intro*: *prod-dvd-prod-subset*)
ultimately show *?thesis* **by** (*rule dvd-trans*)
qed

end

lemma (**in** *semidom*) *prod-zero-iff* [*simp*]:
fixes $f :: 'b \Rightarrow 'a$
assumes *finite A*
shows $prod\ f\ A = 0 \longleftrightarrow (\exists\, a{\in}A.\ f\ a = 0)$
using *assms* **by** (*induct A*) (*auto simp*: *no-zero-divisors*)

lemma (**in** *semidom-divide*) *prod-diff1*:
assumes *finite A* **and** $f\ a \neq 0$
shows $prod\ f\ (A - \{a\}) = (if\ a \in A\ then\ prod\ f\ A\ div\ f\ a\ else\ prod\ f\ A)$
proof (*cases* $a \notin A$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *assms* **show** *?thesis*
proof *induct*
case *empty*
then show *?case* **by** *simp*
next
case (*insert b B*)
then show *?case*
proof (*cases* $a = b$)
case *True*
with *insert* **show** *?thesis* **by** *simp*
next
case *False*
with *insert* **have** $a \in B$ **by** *simp*
define *C* **where** $C = B - \{a\}$
with ‹*finite B*› ‹$a \in B$› **have** $B = insert\ a\ C\ finite\ C\ a \notin C$
by *auto*
with *insert* **show** *?thesis*
by (*auto simp add*: *insert-commute ac-simps*)
qed
qed
qed

lemma *prod-uminus*: $(\prod x{\in}A.\ -f\ x :: 'a :: comm\text{-}ring\text{-}1) = (-1)\ \widehat{}\ card\ A\ *$

$(\prod x{\in}A.\ f\ x)$
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *algebra-simps*)

lemma *prod-diff*:
fixes $f :: 'a \Rightarrow 'b :: field$
assumes *finite A B* $\subseteq A$ $\bigwedge x.\ x \in B \Longrightarrow f\ x \neq 0$
shows *prod f* $(A - B) =$ *prod f A* / *prod f B*
by (*metis assms finite-subset nonzero-eq-divide-eq prod.subset-diff prod-zero-iff*)

lemma *sum-zero-power* [*simp*]: $(\sum i{\in}A.\ c\ i * 0\hat{}i) =$ (*if finite A* $\wedge$ $0 \in A$ *then c 0 else 0*)
for $c :: nat \Rightarrow 'a$::*division-ring*
by (*induct A rule*: *infinite-finite-induct*) *auto*

lemma *sum-zero-power'* [*simp*]:
$(\sum i{\in}A.\ c\ i * 0\hat{}i$ / *d i*$) =$ (*if finite A* $\wedge$ $0 \in A$ *then c 0* / *d 0 else 0*)
for $c :: nat \Rightarrow 'a$::*field*
using *sum-zero-power* [*of* $\lambda i.$ *c i* / *d i A*] **by** *auto*

lemma (**in** *field*) *prod-inversef*: *prod* (*inverse* $\circ$ *f*) *A* = *inverse* (*prod f A*)
proof (*cases finite A*)
case *True*
then show *?thesis*
by (*induct A rule*: *finite-induct*) *simp-all*
next
case *False*
then show *?thesis*
by *auto*
qed

lemma (**in** *field*) *prod-dividef*: $(\prod x{\in}A.\ f\ x$ / *g x*$) =$ *prod f A* / *prod g A*
using *prod-inversef* [*of g A*] **by** (*simp add*: *divide-inverse prod.distrib*)

lemma *prod-Un*:
fixes $f :: 'b \Rightarrow 'a :: field$
assumes *finite A* **and** *finite B*
and $\forall x{\in}A \cap B.\ f\ x \neq 0$
shows *prod f* $(A \cup B) =$ *prod f A* $*$ *prod f B* / *prod f* $(A \cap B)$
proof −
from *assms* **have** *prod f A* $*$ *prod f B* = *prod f* $(A \cup B) *$ *prod f* $(A \cap B)$
by (*simp add*: *prod.union-inter* [*symmetric*, *of A B*])
with *assms* **show** *?thesis*
by *simp*
qed

context *linordered-semidom*
begin

lemma *prod-nonneg*: $(\bigwedge a.\ a{\in}A \Longrightarrow 0 \leq f\ a) \Longrightarrow 0 \leq prod\ f\ A$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *prod-pos*: $(\bigwedge a.\ a{\in}A \Longrightarrow 0 < f\ a) \Longrightarrow 0 < prod\ f\ A$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *prod-mono*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i \wedge f\ i \leq g\ i) \Longrightarrow prod\ f\ A \leq prod\ g\ A$
by (*induct A rule*: *infinite-finite-induct*) (*force intro*!: *prod-nonneg mult-mono*)+

Only one needs to be strict

lemma *prod-mono-strict*:
assumes $i \in A\ f\ i < g\ i$
assumes *finite A*
assumes $\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i \wedge f\ i \leq g\ i$
assumes $\bigwedge i.\ i \in A \Longrightarrow 0 < g\ i$
shows $prod\ f\ A < prod\ g\ A$
proof −
have $prod\ f\ A = f\ i * prod\ f\ (A - \{i\})$
using *assms* **by** (*intro prod.remove*)
also have $\ldots \leq f\ i * prod\ g\ (A - \{i\})$
using *assms* **by** (*intro mult-left-mono prod-mono*) *auto*
also have $\ldots < g\ i * prod\ g\ (A - \{i\})$
using *assms* **by** (*intro mult-strict-right-mono prod-pos*) *auto*
also have $\ldots = prod\ g\ A$
using *assms* **by** (*intro prod.remove* [*symmetric*])
finally show *?thesis* **.**
qed

lemma *prod-le-power*:
assumes *A*: $\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i \wedge f\ i \leq n$ *card A* $\leq k$ **and** $n \geq 1$
shows $prod\ f\ A \leq n\ \hat{}\ k$
using *A*
proof (*induction A arbitrary*: *k rule*: *infinite-finite-induct*)
case (*insert i A*)
then obtain k' **where** k': *card A* $\leq k'$ $k = Suc\ k'$
using *Suc-le-D* **by** *force*
have $f\ i * prod\ f\ A \leq n * n\ \hat{}\ k'$
using *insert* ‹$n \geq 1$› k' **by** (*intro prod-nonneg mult-mono*; *force*)
then show *?case*
by (*auto simp*: ‹$k = Suc\ k'$› *insert.hyps*)
qed (*use* ‹$n \geq 1$› **in** *auto*)

end

lemma *prod-mono2*:
fixes $f :: 'a \Rightarrow 'b :: linordered\text{-}idom$
assumes *fin*: *finite B*
and *sub*: $A \subseteq B$

```
    and nn: ⋀b. b ∈ B−A ⟹ 1 ≤ f b
    and A: ⋀a. a ∈ A ⟹ 0 ≤ f a
  shows prod f A ≤ prod f B
proof −
  have prod f A ≤ prod f A * prod f (B−A)
    by (metis prod-ge-1 A mult-le-cancel-left1 nn not-less prod-nonneg)
  also from fin finite-subset[OF sub fin] have ... = prod f (A ∪ (B−A))
    by (simp add: prod.union-disjoint del: Un-Diff-cancel)
  also from sub have A ∪ (B−A) = B by blast
  finally show ?thesis .
qed

lemma less-1-prod:
  fixes f :: 'a ⇒ 'b::linordered-idom
  shows finite I ⟹ I ≠ {} ⟹ (⋀i. i ∈ I ⟹ 1 < f i) ⟹ 1 < prod f I
  by (induct I rule: finite-ne-induct) (auto intro: less-1-mult)

lemma less-1-prod2:
  fixes f :: 'a ⇒ 'b::linordered-idom
  assumes I: finite I i ∈ I 1 < f i ⋀i. i ∈ I ⟹ 1 ≤ f i
  shows 1 < prod f I
proof −
  have 1 < f i * prod f (I − {i})
    using assms
    by (meson DiffD1 leI less-1-mult less-le-trans mult-le-cancel-left1 prod-ge-1)
  also have ... = prod f I
    using assms by (simp add: prod.remove)
  finally show ?thesis .
qed

lemma (in linordered-field) abs-prod: |prod f A| = (∏ x∈A. |f x|)
  by (induct A rule: infinite-finite-induct) (simp-all add: abs-mult)

lemma prod-eq-1-iff [simp]: finite A ⟹ prod f A = 1 ⟷ (∀ a∈A. f a = 1)
  for f :: 'a ⇒ nat
  by (induct A rule: finite-induct) simp-all

lemma prod-pos-nat-iff [simp]: finite A ⟹ prod f A > 0 ⟷ (∀ a∈A. f a > 0)
  for f :: 'a ⇒ nat
  using prod-zero-iff by (simp del: neq0-conv add: zero-less-iff-neq-zero)

lemma prod-constant [simp]: (∏ x∈ A. y) = y ^ card A
  for y :: 'a::comm-monoid-mult
  by (induct A rule: infinite-finite-induct) simp-all

lemma prod-diff-swap:
  fixes f :: 'a ⇒ 'b :: comm-ring-1
  shows prod (λx. f x − g x) A = (−1) ^ card A * prod (λx. g x − f x) A
  using prod.distrib[of λ-. −1 λx. f x − g x A]
```

by *simp*

lemma *prod-power-distrib*: *prod f A ^ n = prod* (λ*x*. (*f x*) *^ n*) *A*
for *f* :: *'a* ⇒ *'b*::*comm-semiring-1*
by (*induct A rule*: *infinite-finite-induct*) (*auto simp add*: *power-mult-distrib*)

lemma *power-inject-exp'*:
assumes *a* ≠ *1 a* > (*0* :: *'a* :: *linordered-semidom*)
shows *a ^ m = a ^ n* ⟷ *m = n*
by (*metis assms not-less-iff-gr-or-eq order-le-less power-decreasing-iff power-inject-exp*)

lemma *power-sum*: *c ^* (∑ *a*∈*A*. *f a*) = (∏ *a*∈*A*. *c ^ f a*)
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *power-add*)

lemma *prod-gen-delta*:
fixes *b* :: *'b* ⇒ *'a*::*comm-monoid-mult*
assumes *fin*: *finite S*
shows *prod* (λ*k*. *if k = a then b k else c*) *S* =
(*if a* ∈ *S then b a* ∗ *c ^* (*card S* − *1*) *else c ^ card S*)
proof −
let *?f* = (λ*k*. *if k*=*a then b k else c*)
show *?thesis*
proof (*cases a* ∈ *S*)
case *False*
then have ∀ *k*∈ *S*. *?f k = c* **by** *simp*
with *False* **show** *?thesis* **by** (*simp add*: *prod-constant*)
next
case *True*
let *?A* = *S* − {*a*}
let *?B* = {*a*}
from *True* **have** *eq*: *S* = *?A* ∪ *?B* **by** *blast*
have *disjoint*: *?A* ∩ *?B* = {} **by** *simp*
from *fin* **have** *fin'*: *finite ?A finite ?B* **by** *auto*
have *f-A0*: *prod ?f ?A = prod* (λ*i*. *c*) *?A*
by (*rule prod.cong*) *auto*
from *fin True* **have** *card-A*: *card ?A = card S* − *1* **by** *auto*
have *f-A1*: *prod ?f ?A = c ^ card ?A*
unfolding *f-A0* **by** (*rule prod-constant*)
have *prod ?f ?A* ∗ *prod ?f ?B = prod ?f S*
using *prod.union-disjoint*[*OF fin' disjoint*, *of ?f*, *unfolded eq*[*symmetric*]]
by *simp*
with *True card-A* **show** *?thesis*
by (*simp add*: *f-A1 field-simps cong add*: *prod.cong cong del*: *if-weak-cong*)
qed
qed

lemma *sum-image-le*:
fixes *g* :: *'a* ⇒ *'b*::*ordered-comm-monoid-add*

assumes *finite I* $\bigwedge i.\ i \in I \Longrightarrow 0 \leq g(f\ i)$
shows *sum g (f ' I)* $\leq$ *sum (g* $\circ$ *f) I*
using *assms*
proof *induction*
case *empty*
then show *?case* **by** *auto*
next
case (*insert i I*)
hence $*$: *sum g (f ' I)* $\leq$ *g (f i)* + *sum g (f ' I)*
sum g (f ' I) $\leq$ *sum (g* $\circ$ *f) I* **using** *add-increasing* **by** *blast+*
have *sum g (f ' insert i I) = sum g (insert (f i) (f ' I))* **by** *simp*
also have ... $\leq$ *g (f i)* + *sum g (f ' I)* **by** (*simp add*: $*$ *insert sum.insert-if*)
also from $*$ **have** ... $\leq$ *g (f i)* + *sum (g* $\circ$ *f) I* **by** (*intro add-left-mono*)
also from *insert* **have** ... = *sum (g* $\circ$ *f) (insert i I)* **by** (*simp add*: *sum.insert-if*)
finally show *?case* **.**
qed

lemma *prod-add*:
fixes *f1 f2* :: *'a* $\Rightarrow$ *'c* :: *comm-semiring-1*
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x + f2\ x) = (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
using *assms*
proof (*induction A rule*: *finite-induct*)
case (*insert x A*)
have $(\sum X{\in}Pow\ (insert\ x\ A).\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x)) =$
$(\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x)) +$
$(\sum X{\in}insert\ x\ `\ (Pow\ A).\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x))$
unfolding *Pow-insert* **by** (*rule sum.union-disjoint*) (*use insert.hyps* **in** *auto*)
also have $(\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x)) =$
$(\sum X{\in}Pow\ A.\ f2\ x * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof (*rule sum.cong*)
fix *X* **assume** *X*: *X* $\in$ *Pow A*
have $(\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ (A{-}X).\ f2\ x) = f2\ x * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x)$
by (*subst prod.insert*) (*use insert.hyps finite-subset*[*of X A*] *X* **in** ‹*auto simp*: *mult-ac*›)
also have *insert x (A* $-$ *X) = insert x A* $-$ *X*
using *insert.hyps X* **by** *auto*
finally show $(\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x) = f2\ x * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x)$ **.**
qed *auto*
also have $(\sum X{\in}insert\ x\ `\ (Pow\ A).\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}X.\ f2\ x)) =$
$(\sum X{\in}Pow\ A.\ (\prod x{\in}insert\ x\ X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}insert\ x\ X.\ f2\ x))$
by (*subst sum.reindex*) (*use insert.hyps* **in** ‹*auto intro*!: *inj-onI simp*: *o-def*›)
also have $(\sum X{\in}Pow\ A.\ (\prod x{\in}insert\ x\ X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}insert\ x\ X.\ f2\ x)) =$

$(\sum X{\in}Pow\ A.\ f1\ x * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof (*rule sum.cong*)
fix *X* **assume** *X*: $X \in Pow\ A$
show $(\prod x{\in}insert\ x\ X.\ f1\ x) * (\prod x{\in}insert\ x\ A{-}insert\ x\ X.\ f2\ x) =$
$f1\ x * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x)$
by (*subst prod.insert*) (*use insert.hyps finite-subset*[*of X A*] *X* **in** *auto*)
qed *auto*
also have $(\sum X{\in}Pow\ A.\ f2\ x * prod\ f1\ X * prod\ f2\ (A - X)) +$
$(\sum X{\in}Pow\ A.\ f1\ x * prod\ f1\ X * prod\ f2\ (A - X)) =$
$(f1\ x + f2\ x) * (\sum X{\in}Pow\ A.\ prod\ f1\ X * prod\ f2\ (A - X))$
by (*simp add*: *algebra-simps flip*: *sum-distrib-left sum-distrib-right*)
finally show *?case*
by (*subst* (*asm*) *insert.IH* [*symmetric*]) (*use insert.hyps* **in** *simp*)
qed *auto*

lemma *prod-diff-conv-sum*:
fixes *f1 f2* :: *'a* ⇒ *'c* :: *comm-ring-1*
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x - f2\ x) = (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ card\ X * (\prod x{\in}X.\ f2\ x) * (\prod x{\in}A{-}X.\ f1\ x))$
proof −
have $(\prod x{\in}A.\ f1\ x - f2\ x) = (\prod x{\in}A.\ -f2\ x + f1\ x)$
by *simp*
also have ... $= (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ -\ f2\ x) * prod\ f1\ (A - X))$
by (*rule prod-add*) *fact+*
also have ... $= (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ card\ X * (\prod x{\in}X.\ f2\ x) * prod\ f1\ (A - X))$
by (*simp add*: *prod-uminus*)
finally show *?thesis* **.**
qed

lemma *prod-diff-conv-sum'*:
fixes *f1 f2* :: *'a* ⇒ *'c* :: *comm-ring-1*
assumes *finite*: *finite A*
shows $(\prod x{\in}A.\ f1\ x - f2\ x) = (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ (card\ A - card\ X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
proof −
have $(\prod x{\in}A.\ f1\ x - f2\ x) = (\prod x{\in}A.\ f1\ x + (-f2\ x))$
by *simp*
also have ... $= (\sum X{\in}Pow\ A.\ (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ -f2\ x))$
by (*rule prod-add*) *fact+*
also have ... $= (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ card\ (A - X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
by (*simp add*: *prod-uminus mult-ac*)
also have ... $= (\sum X{\in}Pow\ A.\ (-1)\ \hat{}\ (card\ A - card\ X) * (\prod x{\in}X.\ f1\ x) * (\prod x{\in}A{-}X.\ f2\ x))$
using *finite-subset*[*OF - assms*] **by** (*intro sum.cong refl*, *subst card-Diff-subset*) *auto*
finally show *?thesis* **.**

qed

end

49 Chain-complete partial orders and their fixpoints

theory *Complete-Partial-Order*
imports *Product-Type*
begin

49.1 Chains

A chain is a totally-ordered set. Chains are parameterized over the order for maximal flexibility, since type classes are not enough.

definition *chain* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a\ set \Rightarrow bool$
where $chain\ ord\ S \longleftrightarrow (\forall x{\in}S.\ \forall y{\in}S.\ ord\ x\ y \vee ord\ y\ x)$

lemma *chainI*:
assumes $\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow ord\ x\ y \vee ord\ y\ x$
shows *chain ord S*
using *assms* **unfolding** *chain-def* **by** *fast*

lemma *chainD*:
assumes *chain ord S* **and** $x \in S$ **and** $y \in S$
shows *ord x y* $\vee$ *ord y x*
using *assms* **unfolding** *chain-def* **by** *fast*

lemma *chainE*:
assumes *chain ord S* **and** $x \in S$ **and** $y \in S$
obtains *ord x y* | *ord y x*
using *assms* **unfolding** *chain-def* **by** *fast*

lemma *chain-empty*: *chain ord* {}
by (*simp add*: *chain-def*)

lemma *chain-equality*: *chain* (=) $A \longleftrightarrow (\forall x{\in}A.\ \forall y{\in}A.\ x = y)$
by (*auto simp add*: *chain-def*)

lemma *chain-subset*: *chain ord A* $\Longrightarrow B \subseteq A \Longrightarrow$ *chain ord B*
by (*rule chainI*) (*blast dest*: *chainD*)

lemma *chain-imageI*:
assumes *chain*: *chain le-a Y*
and *mono*: $\bigwedge x\ y.\ x \in Y \Longrightarrow y \in Y \Longrightarrow$ *le-a x y* $\Longrightarrow$ *le-b* (*f x*) (*f y*)
shows *chain le-b* (*f* ‘ *Y*)
by (*blast intro*: *chainI dest*: *chainD*[*OF chain*] *mono*)

49.2 Chain-complete partial orders

A *ccpo* has a least upper bound for any chain. In particular, the empty set is a chain, so every *ccpo* must have a bottom element.

class *ccpo* = *order* + *Sup* +
  assumes *ccpo-Sup-upper*: *chain* (≤) *A* ⟹ *x* ∈ *A* ⟹ *x* ≤ *Sup A*
  assumes *ccpo-Sup-least*: *chain* (≤) *A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z*
begin

lemma *chain-singleton*: *Complete-Partial-Order.chain* (≤) {*x*}
  by (*rule chainI*) *simp*

lemma *ccpo-Sup-singleton* [*simp*]: ⨆{*x*} = *x*
  by (*rule order.antisym*) (*auto intro*: *ccpo-Sup-least ccpo-Sup-upper simp add*: *chain-singleton*)

49.3 Transfinite iteration of a function

context notes [[*inductive-internals*]]
begin

inductive-set *iterates* :: ('*a* ⇒ '*a*) ⇒ '*a set*
  for *f* :: '*a* ⇒ '*a*
  where
    step: *x* ∈ *iterates f* ⟹ *f x* ∈ *iterates f*
  | *Sup*: *chain* (≤) *M* ⟹ ∀*x*∈*M*. *x* ∈ *iterates f* ⟹ *Sup M* ∈ *iterates f*

end

lemma *iterates-le-f*: *x* ∈ *iterates f* ⟹ *monotone* (≤) (≤) *f* ⟹ *x* ≤ *f x*
  by (*induct x rule*: *iterates.induct*)
    (*force dest*: *monotoneD intro*!: *ccpo-Sup-upper ccpo-Sup-least*)+

lemma *chain-iterates*:
  assumes *f*: *monotone* (≤) (≤) *f*
  shows *chain* (≤) (*iterates f*) (**is** *chain* - *?C*)
proof (*rule chainI*)
  fix *x y*
  assume *x* ∈ *?C y* ∈ *?C*
  then show *x* ≤ *y* ∨ *y* ≤ *x*
  proof (*induct x arbitrary*: *y rule*: *iterates.induct*)
    fix *x y*
    assume *y*: *y* ∈ *?C*
      and *IH*: ⋀*z*. *z* ∈ *?C* ⟹ *x* ≤ *z* ∨ *z* ≤ *x*
    from *y* **show** *f x* ≤ *y* ∨ *y* ≤ *f x*
    proof (*induct y rule*: *iterates.induct*)
      case (*step y*)
      with *IH f* **show** *?case* **by** (*auto dest*: *monotoneD*)

```
    next
      case (Sup M)
      then have chM: chain (≤) M
        and IH': ⋀z. z ∈ M ⟹ f x ≤ z ∨ z ≤ f x by auto
      show f x ≤ Sup M ∨ Sup M ≤ f x
      proof (cases ∃ z∈M. f x ≤ z)
        case True
        then have f x ≤ Sup M
          by (blast intro: ccpo-Sup-upper[OF chM] order-trans)
        then show ?thesis ..
      next
        case False
        with IH' show ?thesis
          by (auto intro: ccpo-Sup-least[OF chM])
      qed
    qed
  next
    case (Sup M y)
    show ?case
    proof (cases ∃ x∈M. y ≤ x)
      case True
      then have y ≤ Sup M
        by (blast intro: ccpo-Sup-upper[OF Sup(1)] order-trans)
      then show ?thesis ..
    next
      case False with Sup
      show ?thesis by (auto intro: ccpo-Sup-least)
    qed
  qed
qed

lemma bot-in-iterates: Sup {} ∈ iterates f
  by (auto intro: iterates.Sup simp add: chain-empty)
```

49.4 Fixpoint combinator

```
definition fixp :: ('a ⇒ 'a) ⇒ 'a
  where fixp f = Sup (iterates f)

lemma iterates-fixp:
  assumes f: monotone (≤) (≤) f
  shows fixp f ∈ iterates f
  unfolding fixp-def
  by (simp add: iterates.Sup chain-iterates f)

lemma fixp-unfold:
  assumes f: monotone (≤) (≤) f
  shows fixp f = f (fixp f)
proof (rule order.antisym)
```

show *fixp f* ≤ *f* (*fixp f*)
by (*intro iterates-le-f iterates-fixp f*)
have *f* (*fixp f*) ≤ *Sup* (*iterates f*)
by (*intro ccpo-Sup-upper chain-iterates f iterates.step iterates-fixp*)
then show *f* (*fixp f*) ≤ *fixp f*
by (*simp only*: *fixp-def*)
qed

lemma *fixp-lowerbound*:
assumes *f*: *monotone* (≤) (≤) *f*
and *z*: *f z* ≤ *z*
shows *fixp f* ≤ *z*
unfolding *fixp-def*
proof (*rule ccpo-Sup-least*[*OF chain-iterates*[*OF f*]])
fix *x*
assume *x* ∈ *iterates f*
then show *x* ≤ *z*
proof (*induct x rule*: *iterates.induct*)
case (*step x*)
from *f* ‹*x* ≤ *z*› **have** *f x* ≤ *f z* **by** (*rule monotoneD*)
also note *z*
finally show *f x* ≤ *z* **.**
next
case (*Sup M*)
then show *?case*
by (*auto intro*: *ccpo-Sup-least*)
qed
qed

end

49.5 Fixpoint induction

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path ccpo*)›

definition *admissible* :: ('*a set* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ ('*a* ⇒ *bool*) ⇒ *bool*
where *admissible lub ord P* ⟷ (∀ *A*. *chain ord A* ⟶ *A* ≠ {} ⟶ (∀ *x*∈*A*. *P x*) ⟶ *P* (*lub A*))

lemma *admissibleI*:
assumes ⋀*A*. *chain ord A* ⟹ *A* ≠ {} ⟹ ∀ *x*∈*A*. *P x* ⟹ *P* (*lub A*)
shows *ccpo.admissible lub ord P*
using *assms* **unfolding** *ccpo.admissible-def* **by** *fast*

lemma *admissibleD*:
assumes *ccpo.admissible lub ord P*
assumes *chain ord A*
assumes *A* ≠ {}
assumes ⋀*x*. *x* ∈ *A* ⟹ *P x*

shows P (*lub* A)
using *assms* **by** (*auto simp*: *ccpo.admissible-def*)

setup ‹*Sign.map-naming Name-Space.parent-path*›

lemma (**in** *ccpo*) *fixp-induct*:
assumes *adm*: *ccpo.admissible Sup* (≤) P
assumes *mono*: *monotone* (≤) (≤) f
assumes *bot*: P (*Sup* {})
assumes *step*: $\bigwedge x.\ P\ x \Longrightarrow P\ (f\ x)$
shows P (*fixp f*)
unfolding *fixp-def*
using *adm chain-iterates*[*OF mono*]
proof (*rule ccpo.admissibleD*)
show *iterates f* ≠ {}
using *bot-in-iterates* **by** *auto*
next
fix x
assume $x \in$ *iterates f*
then show $P\ x$
proof (*induct rule*: *iterates.induct*)
case *prems*: (*step x*)
from *this*(*2*) **show** *?case* **by** (*rule step*)
next
case (*Sup M*)
then show *?case* **by** (*cases* $M = \{\}$) (*auto intro*: *step bot ccpo.admissibleD adm*)
qed
qed

lemma *admissible-True*: *ccpo.admissible lub ord* (λ*x*. *True*)
unfolding *ccpo.admissible-def* **by** *simp*

lemma *admissible-const*: *ccpo.admissible lub ord* (λ*x*. *t*)
by (*auto intro*: *ccpo.admissibleI*)

lemma *admissible-conj*:
assumes *ccpo.admissible lub ord* (λ*x*. $P\ x$)
assumes *ccpo.admissible lub ord* (λ*x*. $Q\ x$)
shows *ccpo.admissible lub ord* (λ*x*. $P\ x \wedge Q\ x$)
using *assms* **unfolding** *ccpo.admissible-def* **by** *simp*

lemma *admissible-all*:
assumes $\bigwedge y.$ *ccpo.admissible lub ord* (λ*x*. $P\ x\ y$)
shows *ccpo.admissible lub ord* (λ*x*. $\forall y.\ P\ x\ y$)
using *assms* **unfolding** *ccpo.admissible-def* **by** *fast*

lemma *admissible-ball*:

assumes $\bigwedge y.\ y \in A \Longrightarrow$ *ccpo.admissible lub ord* $(\lambda x.\ P\ x\ y)$
shows *ccpo.admissible lub ord* $(\lambda x.\ \forall\, y{\in}A.\ P\ x\ y)$
using *assms* **unfolding** *ccpo.admissible-def* **by** *fast*

lemma *chain-compr*: *chain ord A* $\Longrightarrow$ *chain ord* $\{x \in A.\ P\ x\}$
unfolding *chain-def* **by** *fast*

context *ccpo*
begin

lemma *admissible-disj*:
fixes $P\ Q :: 'a \Rightarrow$ *bool*
assumes *P*: *ccpo.admissible Sup* $(\leq)$ $(\lambda x.\ P\ x)$
assumes *Q*: *ccpo.admissible Sup* $(\leq)$ $(\lambda x.\ Q\ x)$
shows *ccpo.admissible Sup* $(\leq)$ $(\lambda x.\ P\ x \vee Q\ x)$
proof (*rule ccpo.admissibleI*)
fix $A :: 'a$ *set*
assume *chain*: *chain* $(\leq)$ *A*
assume *A*: $A \neq \{\}$ **and** *P-Q*: $\forall\, x{\in}A.\ P\ x \vee Q\ x$
have $(\exists\, x{\in}A.\ P\ x) \wedge (\forall\, x{\in}A.\ \exists\, y{\in}A.\ x \leq y \wedge P\ y) \vee (\exists\, x{\in}A.\ Q\ x) \wedge (\forall\, x{\in}A.\ \exists\, y{\in}A.\ x \leq y \wedge Q\ y)$
(**is** *?P* $\vee$ *?Q* **is** *?P1* $\wedge$ *?P2* $\vee$ -)
proof (*rule disjCI*)
assume $\neg$ *?Q*
then consider $\forall\, x{\in}A.\ \neg\ Q\ x \mid a$ **where** $a \in A\ \forall\, y{\in}A.\ a \leq y \longrightarrow \neg\ Q\ y$
by *blast*
then show *?P*
proof *cases*
case *1*
with *P-Q* **have** $\forall\, x{\in}A.\ P\ x$ **by** *blast*
with *A* **show** *?P* **by** *blast*
next
case *2*
note $a = \langle a \in A \rangle$
show *?P*
proof
from *P-Q 2* **have** $*$: $\forall\, y{\in}A.\ a \leq y \longrightarrow P\ y$ **by** *blast*
with *a* **have** *P a* **by** *blast*
with *a* **show** *?P1* **by** *blast*
show *?P2*
proof
fix *x*
assume *x*: $x \in A$
with *chain a* **show** $\exists\, y{\in}A.\ x \leq y \wedge P\ y$
proof (*rule chainE*)
assume *le*: $a \leq x$
with $*$ *a x* **have** *P x* **by** *blast*
with *x le* **show** *?thesis* **by** *blast*
next

```
          assume a ≥ x
          with a ‹P a› show ?thesis by blast
        qed
      qed
    qed
  qed
  moreover
  have Sup A = Sup {x ∈ A. P x} if ⋀x. x∈A ⟹ ∃ y∈A. x ≤ y ∧ P y for P
  proof (rule order.antisym)
    have chain-P: chain (≤) {x ∈ A. P x}
      by (rule chain-compr [OF chain])
    show Sup A ≤ Sup {x ∈ A. P x}
    proof (rule ccpo-Sup-least [OF chain])
      show ⋀x. x ∈ A ⟹ x ≤ ⨆ {x ∈ A. P x}
        by (blast intro: ccpo-Sup-upper[OF chain-P] order-trans dest: that)
    qed
    show Sup {x ∈ A. P x} ≤ Sup A
      apply (rule ccpo-Sup-least [OF chain-P])
      apply (simp add: ccpo-Sup-upper [OF chain])
      done
  qed
  ultimately
  consider ∃ x. x ∈ A ∧ P x Sup A = Sup {x ∈ A. P x}
    | ∃ x. x ∈ A ∧ Q x Sup A = Sup {x ∈ A. Q x}
    by blast
  then show P (Sup A) ∨ Q (Sup A)
  proof cases
    case 1
    then show ?thesis
      using ccpo.admissibleD [OF P chain-compr [OF chain]] by force
  next
    case 2
    then show ?thesis
      using ccpo.admissibleD [OF Q chain-compr [OF chain]] by force
  qed
qed

end

instance complete-lattice ⊆ ccpo
  by standard (fast intro: Sup-upper Sup-least)+

lemma lfp-eq-fixp:
  assumes mono: mono f
  shows lfp f = fixp f
proof (rule order.antisym)
  from mono have f': monotone (≤) (≤) f
    unfolding mono-def monotone-def .
```

show *lfp f* ≤ *fixp f*
by (*rule lfp-lowerbound*, *subst fixp-unfold* [*OF f'*], *rule order-refl*)
show *fixp f* ≤ *lfp f*
by (*rule fixp-lowerbound* [*OF f'*]) (*simp add*: *lfp-fixpoint* [*OF mono*])
qed

hide-const (**open**) *iterates fixp*

end

50 Datatype option

theory *Option*
imports *Lifting*
begin

datatype *'a option* =
None
| *Some* (*the*: *'a*)

datatype-compat *option*

lemma [*case-names None Some*, *cases type*: *option*]:
— for backward compatibility – names of variables differ
(*y* = *None* ⟹ *P*) ⟹ (⋀*a*. *y* = *Some a* ⟹ *P*) ⟹ *P*
by (*rule option.exhaust*)

lemma [*case-names None Some*, *induct type*: *option*]:
— for backward compatibility – names of variables differ
P None ⟹ (⋀*option*. *P* (*Some option*)) ⟹ *P option*
by (*rule option.induct*)

Compatibility:

setup ‹*Sign.mandatory-path option*›
lemmas *inducts* = *option.induct*
lemmas *cases* = *option.case*
setup ‹*Sign.parent-path*›

lemma *not-None-eq* [*iff*]: *x* ≠ *None* ⟷ (∃ *y*. *x* = *Some y*)
by (*induct x*) *auto*

lemma *not-Some-eq* [*iff*]: (∀ *y*. *x* ≠ *Some y*) ⟷ *x* = *None*
by (*induct x*) *auto*

lemma *comp-the-Some*[*simp*]: *the o Some* = *id*
by *auto*

Although it may appear that both of these equalities are helpful only when applied to assumptions, in practice it seems better to give them the uniform

iff attribute.

lemma *inj-Some* [*simp*]: *inj-on Some A*
by (*rule inj-onI*) *simp*

lemma *case-optionE*:
assumes *c*: (*case x of None* ⇒ *P* | *Some y* ⇒ *Q y*)
obtains
(*None*) *x* = *None* **and** *P*
| (*Some*) *y* **where** *x* = *Some y* **and** *Q y*
using *c* **by** (*cases x*) *simp-all*

lemma *split-option-all*: (∀ *x*. *P x*) ⟷ *P None* ∧ (∀ *x*. *P* (*Some x*))
by (*auto intro*: *option.induct*)

lemma *split-option-ex*: (∃ *x*. *P x*) ⟷ *P None* ∨ (∃ *x*. *P* (*Some x*))
using *split-option-all*[*of* λ*x*. ¬ *P x*] **by** *blast*

lemma *UNIV-option-conv*: *UNIV* = *insert None* (*range Some*)
by (*auto intro*: *classical*)

lemma *rel-option-None1* [*simp*]: *rel-option P None x* ⟷ *x* = *None*
by (*cases x*) *simp-all*

lemma *rel-option-None2* [*simp*]: *rel-option P x None* ⟷ *x* = *None*
by (*cases x*) *simp-all*

lemma *option-rel-Some1*: *rel-option A* (*Some x*) *y* ⟷ (∃ *y'*. *y* = *Some y'* ∧ *A x y'*)
by(*cases y*) *simp-all*

lemma *option-rel-Some2*: *rel-option A x* (*Some y*) ⟷ (∃ *x'*. *x* = *Some x'* ∧ *A x' y*)
by(*cases x*) *simp-all*

lemma *rel-option-inf*: *inf* (*rel-option A*) (*rel-option B*) = *rel-option* (*inf A B*)
(**is** *?lhs* = *?rhs*)
proof (*rule antisym*)
show *?lhs* ≤ *?rhs* **by** (*auto elim*: *option.rel-cases*)
show *?rhs* ≤ *?lhs* **by** (*auto elim*: *option.rel-mono-strong*)
qed

lemma *rel-option-reflI*:
(⋀*x*. *x* ∈ *set-option y* ⟹ *P x x*) ⟹ *rel-option P y y*
by (*cases y*) *auto*

50.0.1 Operations

lemma *ospec* [*dest*]: (∀ *x*∈*set-option A*. *P x*) ⟹ *A* = *Some x* ⟹ *P x*
by *simp*

setup ‹*map-theory-claset* (*fn ctxt => ctxt addSD2* (*ospec*, @{*thm ospec*}))›

lemma *elem-set* [*iff*]: $(x \in \textit{set-option}\ xo) = (xo = \textit{Some}\ x)$
by (*cases xo*) *auto*

lemma *set-empty-eq* [*simp*]: $(\textit{set-option}\ xo = \{\}) = (xo = \textit{None})$
by (*cases xo*) *auto*

lemma *map-option-case*: $\textit{map-option}\ f\ y = (\textit{case}\ y\ \textit{of None} \Rightarrow \textit{None} \mid \textit{Some}\ x \Rightarrow \textit{Some}\ (f\ x))$
by (*auto split*: *option.split*)

lemma *map-option-is-None* [*iff*]: $(\textit{map-option}\ f\ \textit{opt} = \textit{None}) = (\textit{opt} = \textit{None})$
by (*simp add*: *map-option-case split*: *option.split*)

lemma *None-eq-map-option-iff* [*iff*]: $\textit{None} = \textit{map-option}\ f\ x \longleftrightarrow x = \textit{None}$
by(*cases x*) *simp-all*

lemma *map-option-eq-Some* [*iff*]: $(\textit{map-option}\ f\ xo = \textit{Some}\ y) = (\exists z.\ xo = \textit{Some}\ z \wedge f\ z = y)$
by (*simp add*: *map-option-case split*: *option.split*)

lemma *map-option-o-case-sum* [*simp*]:
$\textit{map-option}\ f \circ \textit{case-sum}\ g\ h = \textit{case-sum}\ (\textit{map-option}\ f \circ g)\ (\textit{map-option}\ f \circ h)$
by (*rule o-case-sum*)

lemma *map-option-cong*: $x = y \Longrightarrow (\bigwedge a.\ y = \textit{Some}\ a \Longrightarrow f\ a = g\ a) \Longrightarrow \textit{map-option}\ f\ x = \textit{map-option}\ g\ y$
by (*cases x*) *auto*

lemma *map-option-idI*: $(\bigwedge y.\ y \in \textit{set-option}\ x \Longrightarrow f\ y = y) \Longrightarrow \textit{map-option}\ f\ x = x$
by(*cases x*)(*simp-all*)

functor *map-option*: *map-option*
by (*simp-all add*: *option.map-comp fun-eq-iff option.map-id*)

lemma *case-map-option* [*simp*]: $\textit{case-option}\ g\ h\ (\textit{map-option}\ f\ x) = \textit{case-option}\ g\ (h \circ f)\ x$
by (*cases x*) *simp-all*

lemma *None-notin-image-Some* [*simp*]: $\textit{None} \notin \textit{Some}\ `\ A$
by *auto*

lemma *notin-range-Some*: $x \notin \textit{range Some} \longleftrightarrow x = \textit{None}$
by(*cases x*) *auto*

lemma *rel-option-iff*:
rel-option R x y = (case (x, y) of (None, None) ⇒ True
| (Some x, Some y) ⇒ R x y
| - ⇒ False)
by (*auto split*: *prod.split option.split*)

definition *combine-options* :: *('a ⇒ 'a ⇒ 'a) ⇒ 'a option ⇒ 'a option ⇒ 'a option*
where *combine-options f x y =*
(case x of None ⇒ y | Some x ⇒ (case y of None ⇒ Some x | Some y ⇒ Some (f x y)))

lemma *combine-options-simps* [*simp*]:
combine-options f None y = y
combine-options f x None = x
combine-options f (Some a) (Some b) = Some (f a b)
by (*simp-all add*: *combine-options-def split*: *option.splits*)

lemma *combine-options-cases* [*case-names None1 None2 Some*]:
(x = None ⟹ P x y) ⟹ (y = None ⟹ P x y) ⟹
(⋀a b. x = Some a ⟹ y = Some b ⟹ P x y) ⟹ P x y
by (*cases x*; *cases y*) *simp-all*

lemma *combine-options-commute*:
(⋀x y. f x y = f y x) ⟹ combine-options f x y = combine-options f y x
using *combine-options-cases*[*of x*]
by (*induction x y rule*: *combine-options-cases*) *simp-all*

lemma *combine-options-assoc*:
(⋀x y z. f (f x y) z = f x (f y z)) ⟹
combine-options f (combine-options f x y) z =
combine-options f x (combine-options f y z)
by (*auto simp*: *combine-options-def split*: *option.splits*)

lemma *combine-options-left-commute*:
(⋀x y. f x y = f y x) ⟹ (⋀x y z. f (f x y) z = f x (f y z)) ⟹
combine-options f y (combine-options f x z) =
combine-options f x (combine-options f y z)
by (*auto simp*: *combine-options-def split*: *option.splits*)

lemmas *combine-options-ac* =
combine-options-commute combine-options-assoc combine-options-left-commute

context
begin

qualified definition *is-none* :: *'a option ⇒ bool*
where [*code-post*]: *is-none x ⟷ x = None*

lemma *is-none-simps* [*simp*]:
is-none None
¬ *is-none* (*Some x*)
by (*simp-all add*: *is-none-def*)

lemma *is-none-code* [*code*]:
is-none None ⟷ *True*
is-none (*Some x*) ⟷ *False*
by *simp-all*

lemma *rel-option-unfold*:
rel-option R x y ⟷
(*is-none x* ⟷ *is-none y*) ∧ (¬ *is-none x* ⟶ ¬ *is-none y* ⟶ *R* (*the x*) (*the y*))
by (*simp add*: *rel-option-iff split*: *option.split*)

lemma *rel-optionI*:
⟦ *is-none x* ⟷ *is-none y*; ⟦ ¬ *is-none x*; ¬ *is-none y* ⟧ ⟹ *P* (*the x*) (*the y*) ⟧
⟹ *rel-option P x y*
by (*simp add*: *rel-option-unfold*)

lemma *is-none-map-option* [*simp*]: *is-none* (*map-option f x*) ⟷ *is-none x*
by (*simp add*: *is-none-def*)

lemma *the-map-option*: ¬ *is-none x* ⟹ *the* (*map-option f x*) = *f* (*the x*)
by (*auto simp add*: *is-none-def*)

qualified primrec *bind* :: *'a option* ⇒ (*'a* ⇒ *'b option*) ⇒ *'b option*
where
bind-lzero: *bind None f* = *None*
| *bind-lunit*: *bind* (*Some x*) *f* = *f x*

lemma *is-none-bind*: *is-none* (*bind f g*) ⟷ *is-none f* ∨ *is-none* (*g* (*the f*))
by (*cases f*) *simp-all*

lemma *bind-runit*[*simp*]: *bind x Some* = *x*
by (*cases x*) *auto*

lemma *bind-assoc*[*simp*]: *bind* (*bind x f*) *g* = *bind x* (λ*y*. *bind* (*f y*) *g*)
by (*cases x*) *auto*

lemma *bind-rzero*[*simp*]: *bind x* (λ*x*. *None*) = *None*
by (*cases x*) *auto*

qualified lemma *bind-cong*: *x* = *y* ⟹ (⋀*a*. *y* = *Some a* ⟹ *f a* = *g a*) ⟹ *bind x f* = *bind y g*
by (*cases x*) *auto*

lemma *bind-split*: $P\ (\mathit{bind}\ m\ f) \longleftrightarrow (m = \mathit{None} \longrightarrow P\ \mathit{None}) \wedge (\forall v.\ m = \mathit{Some}\ v \longrightarrow P\ (f\ v))$
by (*cases m*) *auto*

lemma *bind-split-asm*: $P\ (\mathit{bind}\ m\ f) \longleftrightarrow \neg\ (m = \mathit{None} \wedge \neg\ P\ \mathit{None} \vee (\exists x.\ m = \mathit{Some}\ x \wedge \neg\ P\ (f\ x)))$
by (*cases m*) *auto*

lemmas *bind-splits* = *bind-split bind-split-asm*

lemma *bind-eq-Some-conv*: $\mathit{bind}\ f\ g = \mathit{Some}\ x \longleftrightarrow (\exists y.\ f = \mathit{Some}\ y \wedge g\ y = \mathit{Some}\ x)$
by (*cases f*) *simp-all*

lemma *bind-eq-None-conv*: $\mathit{Option.bind}\ a\ f = \mathit{None} \longleftrightarrow a = \mathit{None} \vee f\ (\mathit{the}\ a) = \mathit{None}$
by(*cases a*) *simp-all*

lemma *map-option-bind*: $\mathit{map\text{-}option}\ f\ (\mathit{bind}\ x\ g) = \mathit{bind}\ x\ (\mathit{map\text{-}option}\ f \circ g)$
by (*cases x*) *simp-all*

lemma *bind-option-cong*:
$[\![\ x = y;\ \bigwedge z.\ z \in \mathit{set\text{-}option}\ y \Longrightarrow f\ z = g\ z\ ]\!] \Longrightarrow \mathit{bind}\ x\ f = \mathit{bind}\ y\ g$
by (*cases y*) *simp-all*

lemma *bind-option-cong-simp*:
$[\![\ x = y;\ \bigwedge z.\ z \in \mathit{set\text{-}option}\ y =\mathit{simp}=> f\ z = g\ z\ ]\!] \Longrightarrow \mathit{bind}\ x\ f = \mathit{bind}\ y\ g$
unfolding *simp-implies-def* **by** (*rule bind-option-cong*)

lemma *bind-option-cong-code*: $x = y \Longrightarrow \mathit{bind}\ x\ f = \mathit{bind}\ y\ f$
by *simp*

lemma *bind-map-option*: $\mathit{bind}\ (\mathit{map\text{-}option}\ f\ x)\ g = \mathit{bind}\ x\ (g \circ f)$
by(*cases x*) *simp-all*

lemma *set-bind-option* [*simp*]: $\mathit{set\text{-}option}\ (\mathit{bind}\ x\ f) = (\bigcup((\mathit{set\text{-}option} \circ f)\ `\ \mathit{set\text{-}option}\ x))$
by(*cases x*) *auto*

lemma *map-conv-bind-option*: $\mathit{map\text{-}option}\ f\ x = \mathit{Option.bind}\ x\ (\mathit{Some} \circ f)$
by(*cases x*) *simp-all*

end

setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm bind-option-cong-code*})›

context

begin

qualified definition *these* :: *'a option set* ⇒ *'a set*
  where *these A* = *the* ‘ {*x* ∈ *A*. *x* ≠ *None*}

qualified lemma *these-eq* [*code*]:
  ‹*these A* = *the* ‘ (*Set.filter* (*Not* ∘ *Option.is-none*) *A*)›
  by (*simp add*: *these-def Option.is-none-def*)

qualified lemma *these-unfold*:
  ‹*these A* = {*x*. ∃ *y* ∈ *A*. *y* = *Some x*}›
  by (*auto simp add*: *these-def set-eq-iff image-iff*)

lemma *these-empty* [*simp*]: *these* {} = {}
  by (*simp add*: *these-def*)

lemma *these-insert-None* [*simp*]: *these* (*insert None A*) = *these A*
  by (*auto simp add*: *these-def*)

lemma *these-insert-Some* [*simp*]: *these* (*insert* (*Some x*) *A*) = *insert x* (*these A*)
proof −
  have {*y* ∈ *insert* (*Some x*) *A*. *y* ≠ *None*} = *insert* (*Some x*) {*y* ∈ *A*. *y* ≠ *None*}
    by *auto*
  then show *?thesis* **by** (*simp add*: *these-def*)
qed

lemma *in-these-eq*: *x* ∈ *these A* ⟷ *Some x* ∈ *A*
proof
  assume *Some x* ∈ *A*
  then obtain *B* **where** *A* = *insert* (*Some x*) *B* **by** *auto*
  then show *x* ∈ *these A* **by** (*auto simp add*: *these-def intro*!: *image-eqI*)
next
  assume *x* ∈ *these A*
  then show *Some x* ∈ *A* **by** (*auto simp add*: *these-def*)
qed

lemma *these-image-Some-eq* [*simp*]: *these* (*Some* ‘ *A*) = *A*
  by (*auto simp add*: *these-def intro*!: *image-eqI*)

lemma *Some-image-these-eq*: *Some* ‘ *these A* = {*x*∈*A*. *x* ≠ *None*}
  by (*auto simp add*: *these-def image-image intro*!: *image-eqI*)

lemma *these-empty-eq*: *these B* = {} ⟷ *B* = {} ∨ *B* = {*None*}
  by (*auto simp add*: *these-def*)

lemma *these-not-empty-eq*: *these B* ≠ {} ⟷ *B* ≠ {} ∧ *B* ≠ {*None*}
  by (*auto simp add*: *these-empty-eq*)

qualified definition *image-filter* :: (*'a* ⇒ *'b option*) ⇒ *'a set* ⇒ *'b set*

where *image-filter-eq*: *image-filter f A = these (f ' A)*

end

lemma *finite-range-Some*: *finite* (*range* (*Some* :: *'a* ⇒ *'a option*)) = *finite* (*UNIV* :: *'a set*)
by (*auto dest*: *finite-imageD intro*: *inj-Some*)

50.1 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *option-bind-transfer* [*transfer-rule*]:
(*rel-option A* ===> (*A* ===> *rel-option B*) ===> *rel-option B*)
Option.bind Option.bind
unfolding *rel-fun-def split-option-all* **by** *simp*

lemma *pred-option-parametric* [*transfer-rule*]:
((*A* ===> (=)) ===> *rel-option A* ===> (=)) *pred-option pred-option*
by (*rule rel-funI*)+ (*auto simp add*: *rel-option-unfold Option.is-none-def dest*: *rel-funD*)

end

50.1.1 Interaction with finite sets

lemma *finite-option-UNIV* [*simp*]:
finite (*UNIV* :: *'a option set*) = *finite* (*UNIV* :: *'a set*)
by (*auto simp add*: *UNIV-option-conv elim*: *finite-imageD intro*: *inj-Some*)

instance *option* :: (*finite*) *finite*
by *standard* (*simp add*: *UNIV-option-conv*)

50.1.2 Code generator setup

lemma *equal-None-code-unfold* [*code-unfold*]:
HOL.equal x None ⟷ *Option.is-none x*
HOL.equal None = *Option.is-none*
by (*auto simp add*: *equal Option.is-none-def*)

code-printing
type-constructor *option* ⇀
(*SML*) *option*
and (*OCaml*) *option*
and (*Haskell*) *Maybe*
and (*Scala*) !*Option*[(-)]
| **constant** *None* ⇀
(*SML*) *NONE*
and (*OCaml*) *None*

and (*Haskell*) *Nothing*
and (*Scala*) !*None*
| **constant** *Some* ⇀
(*SML*) *SOME*
and (*OCaml*) *Some -*
and (*Haskell*) *Just*
and (*Scala*) *Some*
| **class-instance** *option* :: *equal* ⇀
(*Haskell*) –
| **constant** *HOL.equal* :: *'a option* ⇒ *'a option* ⇒ *bool* ⇀
(*Haskell*) **infix** *4* ==

code-reserved
(*SML*) *option NONE SOME*
and (*OCaml*) *option None Some*
and (*Scala*) *Option None Some*

end

51 Partial Function Definitions

theory *Partial-Function*
imports *Complete-Partial-Order Option*
keywords *partial-function* :: *thy-defn*
begin

named-theorems *partial-function-mono monotonicity rules for partial function definitions*
ML-file ‹*Tools/Function/partial-function.ML*›

lemma (**in** *ccpo*) *in-chain-finite*:
assumes *Complete-Partial-Order.chain* (≤) *A finite A A* ≠ {}
shows ⨆ *A* ∈ *A*
using *assms*(*2*,*1*,*3*)
proof *induction*
case *empty* **thus** *?case* **by** *simp*
next
case (*insert x A*)
note *chain* = ‹*Complete-Partial-Order.chain* (≤) (*insert x A*)›
show *?case*
proof(*cases A* = {})
case *True* **thus** *?thesis* **by** *simp*
next
case *False*
from *chain* **have** *chain'*: *Complete-Partial-Order.chain* (≤) *A*
by(*rule chain-subset*) *blast*
hence ⨆ *A* ∈ *A* **using** *False* **by**(*rule insert.IH*)
show *?thesis*
proof(*cases x* ≤ ⨆ *A*)

case *True*
have $\bigsqcup(\mathit{insert}\ x\ A) \leq \bigsqcup A$ **using** *chain*
by(*rule ccpo-Sup-least*)(*auto simp add*: *True intro*: *ccpo-Sup-upper*[*OF chain*'])
hence $\bigsqcup(\mathit{insert}\ x\ A) = \bigsqcup A$
by(*rule order.antisym*)(*blast intro*: *ccpo-Sup-upper*[*OF chain*] *ccpo-Sup-least*[*OF chain*'])
with ‹$\bigsqcup A \in A$› **show** *?thesis* **by** *simp*
next
case *False*
with *chainD*[*OF chain*, *of x* $\bigsqcup A$] ‹$\bigsqcup A \in A$›
have $\bigsqcup(\mathit{insert}\ x\ A) = x$
by(*auto intro*: *order.antisym ccpo-Sup-least*[*OF chain*] *order-trans*[*OF ccpo-Sup-upper*[*OF chain*']] *ccpo-Sup-upper*[*OF chain*])
thus *?thesis* **by** *simp*
qed
qed
qed

lemma (**in** *ccpo*) *admissible-chfin*:
$(\forall S.\ \mathit{Complete\text{-}Partial\text{-}Order.chain}\ (\leq)\ S \longrightarrow \mathit{finite}\ S)$
$\Longrightarrow \mathit{ccpo.admissible}\ \mathit{Sup}\ (\leq)\ P$
using *in-chain-finite* **by** (*blast intro*: *ccpo.admissibleI*)

51.1 Axiomatic setup

This techical locale constains the requirements for function definitions with ccpo fixed points.

definition $\mathit{fun\text{-}ord}\ \mathit{ord}\ f\ g \longleftrightarrow (\forall x.\ \mathit{ord}\ (f\ x)\ (g\ x))$
definition $\mathit{fun\text{-}lub}\ L\ A = (\lambda x.\ L\ \{y.\ \exists f{\in}A.\ y = f\ x\})$
definition $\mathit{img\text{-}ord}\ f\ \mathit{ord} = (\lambda x\ y.\ \mathit{ord}\ (f\ x)\ (f\ y))$
definition $\mathit{img\text{-}lub}\ f\ g\ \mathit{Lub} = (\lambda A.\ g\ (\mathit{Lub}\ (f\ `\ A)))$

lemma *chain-fun*:
assumes *A*: *chain* (*fun-ord ord*) *A*
shows *chain ord* $\{y.\ \exists f{\in}A.\ y = f\ a\}$ (**is** *chain ord ?C*)
proof (*rule chainI*)
fix *x y* **assume** $x \in ?C\ y \in ?C$
then obtain *f g* **where** *fg*: $f \in A\ g \in A$
and [*simp*]: $x = f\ a\ y = g\ a$ **by** *blast*
from *chainD*[*OF A fg*]
show *ord x y* $\vee$ *ord y x* **unfolding** *fun-ord-def* **by** *auto*
qed

lemma *call-mono*[*partial-function-mono*]: *monotone* (*fun-ord ord*) *ord* $(\lambda f.\ f\ t)$
by (*rule monotoneI*) (*auto simp*: *fun-ord-def*)

lemma *let-mono*[*partial-function-mono*]:
$(\bigwedge x.\ \mathit{monotone}\ \mathit{orda}\ \mathit{ordb}\ (\lambda f.\ b\ f\ x))$

$\Longrightarrow$ *monotone orda ordb* (λf. *Let t* (*b f*))
by (*simp add*: *Let-def*)

lemma *if-mono*[*partial-function-mono*]: *monotone orda ordb F*
$\Longrightarrow$ *monotone orda ordb G*
$\Longrightarrow$ *monotone orda ordb* (λf. *if c then F f else G f*)
unfolding *monotone-def* **by** *simp*

definition *mk-less R* = ($\lambda x\ y$. *R x y* $\wedge$ $\neg$ *R y x*)

locale *partial-function-definitions* =
fixes *leq* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *bool*
fixes *lub* :: *'a set* $\Rightarrow$ *'a*
assumes *leq-refl*: *leq x x*
assumes *leq-trans*: *leq x y* $\Longrightarrow$ *leq y z* $\Longrightarrow$ *leq x z*
assumes *leq-antisym*: *leq x y* $\Longrightarrow$ *leq y x* $\Longrightarrow$ *x* = *y*
assumes *lub-upper*: *chain leq A* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *leq x* (*lub A*)
assumes *lub-least*: *chain leq A* $\Longrightarrow$ ($\bigwedge x$. *x* $\in$ *A* $\Longrightarrow$ *leq x z*) $\Longrightarrow$ *leq* (*lub A*) *z*

lemma *partial-function-lift*:
assumes *partial-function-definitions ord lb*
shows *partial-function-definitions* (*fun-ord ord*) (*fun-lub lb*) (**is** *partial-function-definitions ?ordf ?lubf*)
proof −
interpret *partial-function-definitions ord lb* **by** *fact*

show *?thesis*
proof
fix *x* **show** *?ordf x x*
unfolding *fun-ord-def* **by** (*auto simp*: *leq-refl*)
next
fix *x y z* **assume** *?ordf x y ?ordf y z*
thus *?ordf x z* **unfolding** *fun-ord-def*
by (*force dest*: *leq-trans*)
next
fix *x y* **assume** *?ordf x y ?ordf y x*
thus *x* = *y* **unfolding** *fun-ord-def*
by (*force intro*!: *dest*: *leq-antisym*)
next
fix *A f* **assume** *f*: *f* $\in$ *A* **and** *A*: *chain ?ordf A*
thus *?ordf f* (*?lubf A*)
unfolding *fun-lub-def fun-ord-def*
by (*blast intro*: *lub-upper chain-fun*[*OF A*] *f*)
next
fix *A* :: (*'b* $\Rightarrow$ *'a*) *set* **and** *g* :: *'b* $\Rightarrow$ *'a*
assume *A*: *chain ?ordf A* **and** *g*: $\bigwedge f$. *f* $\in$ *A* $\Longrightarrow$ *?ordf f g*
show *?ordf* (*?lubf A*) *g* **unfolding** *fun-lub-def fun-ord-def*
by (*blast intro*: *lub-least chain-fun*[*OF A*] *dest*: *g*[*unfolded fun-ord-def*])
qed

qed

lemma *ccpo*: **assumes** *partial-function-definitions ord lb*
shows *class.ccpo lb ord* (*mk-less ord*)
using *assms* **unfolding** *partial-function-definitions-def mk-less-def*
by *unfold-locales blast+*

lemma *partial-function-image*:
assumes *partial-function-definitions ord Lub*
assumes *inj*: $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
assumes *inv*: $\bigwedge x.\ f\ (g\ x) = x$
shows *partial-function-definitions* (*img-ord f ord*) (*img-lub f g Lub*)
proof −
let *?iord* = *img-ord f ord*
let *?ilub* = *img-lub f g Lub*

interpret *partial-function-definitions ord Lub* **by** *fact*
show *?thesis*
proof
fix *A x* **assume** *chain ?iord A x* ∈ *A*
then have *chain ord* (*f* ‘ *A*) *f x* ∈ *f* ‘ *A*
by (*auto simp*: *img-ord-def intro*: *chainI dest*: *chainD*)
thus *?iord x* (*?ilub A*)
unfolding *inv img-lub-def img-ord-def* **by** (*rule lub-upper*)
next
fix *A x* **assume** *chain ?iord A*
and *1*: $\bigwedge z.\ z \in A \Longrightarrow$ *?iord z x*
then have *chain ord* (*f* ‘ *A*)
by (*auto simp*: *img-ord-def intro*: *chainI dest*: *chainD*)
thus *?iord* (*?ilub A*) *x*
unfolding *inv img-lub-def img-ord-def*
by (*rule lub-least*) (*auto dest*: *1*[*unfolded img-ord-def*])
qed (*auto simp*: *img-ord-def intro*: *leq-refl dest*: *leq-trans leq-antisym inj*)
qed

context *partial-function-definitions*
begin

abbreviation *le-fun* ≡ *fun-ord leq*
abbreviation *lub-fun* ≡ *fun-lub lub*
abbreviation *fixp-fun* ≡ *ccpo.fixp lub-fun le-fun*
abbreviation *mono-body* ≡ *monotone le-fun leq*
abbreviation *admissible* ≡ *ccpo.admissible lub-fun le-fun*

Interpret manually, to avoid flooding everything with facts about orders

lemma *ccpo*: *class.ccpo lub-fun le-fun* (*mk-less le-fun*)
apply (*rule ccpo*)
apply (*rule partial-function-lift*)
apply (*rule partial-function-definitions-axioms*)

done

The crucial fixed-point theorem

lemma *mono-body-fixp*:
(⋀*x*. *mono-body* (λ*f*. *F f x*)) ⟹ *fixp-fun F* = *F* (*fixp-fun F*)
by (*rule ccpo.fixp-unfold*[*OF ccpo*]) (*auto simp*: *monotone-def fun-ord-def*)

Version with curry/uncurry combinators, to be used by package

lemma *fixp-rule-uc*:
fixes *F* :: *'c* ⇒ *'c* **and**
U :: *'c* ⇒ *'b* ⇒ *'a* **and**
C :: (*'b* ⇒ *'a*) ⇒ *'c*
assumes *mono*: ⋀*x*. *mono-body* (λ*f*. *U* (*F* (*C f*)) *x*)
assumes *eq*: *f* ≡ *C* (*fixp-fun* (λ*f*. *U* (*F* (*C f*))))
assumes *inverse*: ⋀*f*. *C* (*U f*) = *f*
shows *f* = *F f*
proof −
have *f* = *C* (*fixp-fun* (λ*f*. *U* (*F* (*C f*)))) **by** (*simp add*: *eq*)
also have ... = *C* (*U* (*F* (*C* (*fixp-fun* (λ*f*. *U* (*F* (*C f*)))))))
by (*subst mono-body-fixp*[*of* %*f*. *U* (*F* (*C f*)), *OF mono*]) (*rule refl*)
also have ... = *F* (*C* (*fixp-fun* (λ*f*. *U* (*F* (*C f*))))) **by** (*rule inverse*)
also have ... = *F f* **by** (*simp add*: *eq*)
finally show *f* = *F f* **.**
qed

Fixpoint induction rule

lemma *fixp-induct-uc*:
fixes *F* :: *'c* ⇒ *'c*
and *U* :: *'c* ⇒ *'b* ⇒ *'a*
and *C* :: (*'b* ⇒ *'a*) ⇒ *'c*
and *P* :: (*'b* ⇒ *'a*) ⇒ *bool*
assumes *mono*: ⋀*x*. *mono-body* (λ*f*. *U* (*F* (*C f*)) *x*)
and *eq*: *f* ≡ *C* (*fixp-fun* (λ*f*. *U* (*F* (*C f*))))
and *inverse*: ⋀*f*. *U* (*C f*) = *f*
and *adm*: *ccpo.admissible lub-fun le-fun P*
and *bot*: *P* (λ-. *lub* {})
and *step*: ⋀*f*. *P* (*U f*) ⟹ *P* (*U* (*F f*))
shows *P* (*U f*)
unfolding *eq inverse*
proof (*rule ccpo.fixp-induct*[*OF ccpo adm*])
show *monotone le-fun le-fun* (λ*f*. *U* (*F* (*C f*)))
using *mono* **by** (*auto simp*: *monotone-def fun-ord-def*)
next
show *P* (*lub-fun* {})
by (*auto simp*: *bot fun-lub-def*)
next
fix *x*
assume *P x*
then show *P* (*U* (*F* (*C x*)))

using *step*[*of C x*] **by** (*simp add*: *inverse*)
qed

Rules for *monotone le-fun leq*:

lemma *const-mono*[*partial-function-mono*]: *monotone ord leq* (λ*f*. *c*)
by (*rule monotoneI*) (*rule leq-refl*)

end

51.2 Flat interpretation: tailrec and option

definition
flat-ord b x y ⟷ *x* = *b* ∨ *x* = *y*

definition
flat-lub b A = (*if A* ⊆ {*b*} *then b else* (*THE x*. *x* ∈ *A* − {*b*}))

lemma *flat-interpretation*:
partial-function-definitions (*flat-ord b*) (*flat-lub b*)
proof
fix *A x* **assume** *1*: *chain* (*flat-ord b*) *A x* ∈ *A*
show *flat-ord b x* (*flat-lub b A*)
proof *cases*
assume *x* = *b*
thus *?thesis* **by** (*simp add*: *flat-ord-def*)
next
assume *x* ≠ *b*
with *1* **have** *A* − {*b*} = {*x*}
by (*auto elim*: *chainE simp*: *flat-ord-def*)
then have *flat-lub b A* = *x*
by (*auto simp*: *flat-lub-def*)
thus *?thesis* **by** (*auto simp*: *flat-ord-def*)
qed
next
fix *A z* **assume** *A*: *chain* (*flat-ord b*) *A*
and *z*: ⋀*x*. *x* ∈ *A* ⟹ *flat-ord b x z*
show *flat-ord b* (*flat-lub b A*) *z*
proof *cases*
assume *A* ⊆ {*b*}
thus *?thesis*
by (*auto simp*: *flat-lub-def flat-ord-def*)
next
assume *nb*: ¬ *A* ⊆ {*b*}
then obtain *y* **where** *y*: *y* ∈ *A y* ≠ *b* **by** *auto*
with *A* **have** *A* − {*b*} = {*y*}
by (*auto elim*: *chainE simp*: *flat-ord-def*)
with *nb* **have** *flat-lub b A* = *y*
by (*auto simp*: *flat-lub-def*)
with *z y* **show** *?thesis* **by** *auto*

qed
qed (*auto simp*: *flat-ord-def*)

lemma *flat-ordI*: $(x \neq a \Longrightarrow x = y) \Longrightarrow$ *flat-ord a x y*
by(*auto simp add*: *flat-ord-def*)

lemma *flat-ord-antisym*: $[\![$ *flat-ord a x y*; *flat-ord a y x* $]\!] \Longrightarrow x = y$
by(*auto simp add*: *flat-ord-def*)

lemma *antisymp-flat-ord*: *antisymp* (*flat-ord a*)
by(*rule antisympI*)(*auto dest*: *flat-ord-antisym*)

interpretation *tailrec*:
partial-function-definitions flat-ord undefined flat-lub undefined
rewrites *flat-lub undefined* {} ≡ *undefined*
by (*rule flat-interpretation*)(*simp add*: *flat-lub-def*)

interpretation *option*:
partial-function-definitions flat-ord None flat-lub None
rewrites *flat-lub None* {} ≡ *None*
by (*rule flat-interpretation*)(*simp add*: *flat-lub-def*)

abbreviation *tailrec-ord* ≡ *flat-ord undefined*
abbreviation *mono-tailrec* ≡ *monotone* (*fun-ord tailrec-ord*) *tailrec-ord*

lemma *tailrec-admissible*:
ccpo.admissible (*fun-lub* (*flat-lub c*)) (*fun-ord* (*flat-ord c*))
(λ*a*. ∀ *x*. *a x* ≠ *c* ⟶ *P x* (*a x*))
proof(*intro ccpo.admissibleI strip*)
fix *A x*
assume *chain*: *Complete-Partial-Order.chain* (*fun-ord* (*flat-ord c*)) *A*
and *P* [*rule-format*]: ∀ *f*∈*A*. ∀ *x*. *f x* ≠ *c* ⟶ *P x* (*f x*)
and *defined*: *fun-lub* (*flat-lub c*) *A x* ≠ *c*
from *defined* **obtain** *f* **where** *f*: *f* ∈ *A f x* ≠ *c*
by(*auto simp add*: *fun-lub-def flat-lub-def split*: *if-split-asm*)
hence *P x* (*f x*) **by**(*rule P*)
moreover from *chain f* **have** ∀ *f'* ∈ *A*. *f' x* = *c* ∨ *f' x* = *f x*
by(*auto 4 4 simp add*: *Complete-Partial-Order.chain-def flat-ord-def fun-ord-def*)
hence *fun-lub* (*flat-lub c*) *A x* = *f x*
using *f* **by**(*auto simp add*: *fun-lub-def flat-lub-def*)
ultimately show *P x* (*fun-lub* (*flat-lub c*) *A x*) **by** *simp*
qed

lemma *fixp-induct-tailrec*:
fixes *F* :: *'c* ⇒ *'c* **and**
U :: *'c* ⇒ *'b* ⇒ *'a* **and**
C :: (*'b* ⇒ *'a*) ⇒ *'c* **and**
P :: *'b* ⇒ *'a* ⇒ *bool* **and**

$x :: 'b$
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord* (*flat-ord c*)) (*flat-ord c*) ($\lambda f.$ *U* (*F* (*C f*)) *x*)
assumes *eq*: $f \equiv$ *C* (*ccpo.fixp* (*fun-lub* (*flat-lub c*)) (*fun-ord* (*flat-ord c*)) ($\lambda f.$ *U* (*F* (*C f*))))
assumes *inverse2*: $\bigwedge f.$ *U* (*C f*) = *f*
assumes *step*: $\bigwedge f\ x\ y.$ ($\bigwedge x\ y.$ *U f x* = *y* $\Longrightarrow$ *y* $\neq$ *c* $\Longrightarrow$ *P x y*) $\Longrightarrow$ *U* (*F f*) *x* = *y* $\Longrightarrow$ *y* $\neq$ *c* $\Longrightarrow$ *P x y*
assumes *result*: *U f x* = *y*
assumes *defined*: *y* $\neq$ *c*
shows *P x y*
proof −
have $\forall\, x\ y.$ *U f x* = *y* $\longrightarrow$ *y* $\neq$ *c* $\longrightarrow$ *P x y*
by(*rule partial-function-definitions.fixp-induct-uc*[*OF flat-interpretation*, *of* - *U F C*, *OF mono eq inverse2*])
(*auto intro*: *step tailrec-admissible simp add*: *fun-lub-def flat-lub-def*)
thus *?thesis* **using** *result defined* **by** *blast*
qed

lemma *admissible-image*:
assumes *pfun*: *partial-function-definitions le lub*
assumes *adm*: *ccpo.admissible lub le* (*P* ∘ *g*)
assumes *inj*: $\bigwedge x\ y.$ *f x* = *f y* $\Longrightarrow$ *x* = *y*
assumes *inv*: $\bigwedge x.$ *f* (*g x*) = *x*
shows *ccpo.admissible* (*img-lub f g lub*) (*img-ord f le*) *P*
proof (*rule ccpo.admissibleI*)
fix *A* **assume** *chain* (*img-ord f le*) *A*
then have *ch'*: *chain le* (*f* ‘ *A*)
by (*auto simp*: *img-ord-def intro*: *chainI dest*: *chainD*)
assume *A* $\neq$ {}
assume *P-A*: $\forall\, x{\in}A.$ *P x*
have (*P* ∘ *g*) (*lub* (*f* ‘ *A*)) **using** *adm ch'*
proof (*rule ccpo.admissibleD*)
fix *x* **assume** *x* ∈ *f* ‘ *A*
with *P-A* **show** (*P* ∘ *g*) *x* **by** (*auto simp*: *inj*[*OF inv*])
qed(*simp add*: ‹*A* $\neq$ {}›)
thus *P* (*img-lub f g lub A*) **unfolding** *img-lub-def* **by** *simp*
qed

lemma *admissible-fun*:
assumes *pfun*: *partial-function-definitions le lub*
assumes *adm*: $\bigwedge x.$ *ccpo.admissible lub le* (*Q x*)
shows *ccpo.admissible* (*fun-lub lub*) (*fun-ord le*) ($\lambda f.\ \forall\, x.$ *Q x* (*f x*))
proof (*rule ccpo.admissibleI*)
fix *A* :: ('*b* ⇒ '*a*) *set*
assume *Q*: $\forall\, f{\in}A.\ \forall\, x.$ *Q x* (*f x*)
assume *ch*: *chain* (*fun-ord le*) *A*
assume *A* $\neq$ {}
hence *non-empty*: $\bigwedge a.$ {*y*. $\exists\, f{\in}A.$ *y* = *f a*} $\neq$ {} **by** *auto*

```
  show ∀x. Q x (fun-lub lub A x)
    unfolding fun-lub-def
    by (rule allI, rule ccpo.admissibleD[OF adm chain-fun[OF ch] non-empty])
      (auto simp: Q)
qed


abbreviation option-ord ≡ flat-ord None
abbreviation mono-option ≡ monotone (fun-ord option-ord) option-ord

lemma bind-mono[partial-function-mono]:
assumes mf: mono-option B and mg: ⋀y. mono-option (λf. C y f)
shows mono-option (λf. Option.bind (B f) (λy. C y f))
proof (rule monotoneI)
  fix f g :: 'a ⇒ 'b option assume fg: fun-ord option-ord f g
  with mf
  have option-ord (B f) (B g) by (rule monotoneD[of - - - f g])
  then have option-ord (Option.bind (B f) (λy. C y f)) (Option.bind (B g) (λy.
C y f))
    unfolding flat-ord-def by auto
  also from mg
  have ⋀y'. option-ord (C y' f) (C y' g)
    by (rule monotoneD) (rule fg)
  then have option-ord (Option.bind (B g) (λy'. C y' f)) (Option.bind (B g) (λy'.
C y' g))
    unfolding flat-ord-def by (cases B g) auto
  finally (option.leq-trans)
  show option-ord (Option.bind (B f) (λy. C y f)) (Option.bind (B g) (λy'. C y'
g)) .
qed

lemma flat-lub-in-chain:
  assumes ch: chain (flat-ord b) A
  assumes lub: flat-lub b A = a
  shows a = b ∨ a ∈ A
proof (cases A ⊆ {b})
  case True
  then have flat-lub b A = b unfolding flat-lub-def by simp
  with lub show ?thesis by simp
next
  case False
  then obtain c where c ∈ A and c ≠ b by auto
  { fix z assume z ∈ A
    from chainD[OF ch ‹c ∈ A› this] have z = c ∨ z = b
      unfolding flat-ord-def using ‹c ≠ b› by auto }
  with False have A − {b} = {c} by auto
  with False have flat-lub b A = c by (auto simp: flat-lub-def)
  with ‹c ∈ A› lub show ?thesis by simp
qed
```

lemma *option-admissible*: *option.admissible* (%(*f*::*'a* ⇒ *'b option*).
(∀ *x y*. *f x* = *Some y* ⟶ *P x y*))
proof (*rule ccpo.admissibleI*)
fix *A* :: (*'a* ⇒ *'b option*) *set*
assume *ch*: *chain option.le-fun A*
and *IH*: ∀ *f*∈*A*. ∀ *x y*. *f x* = *Some y* ⟶ *P x y*
from *ch* **have** *ch'*: ⋀*x*. *chain option-ord* {*y*. ∃ *f*∈*A*. *y* = *f x*} **by** (*rule chain-fun*)
show ∀ *x y*. *option.lub-fun A x* = *Some y* ⟶ *P x y*
proof (*intro allI impI*)
fix *x y* **assume** *option.lub-fun A x* = *Some y*
from *flat-lub-in-chain*[*OF ch' this*[*unfolded fun-lub-def*]]
have *Some y* ∈ {*y*. ∃ *f*∈*A*. *y* = *f x*} **by** *simp*
then have ∃ *f*∈*A*. *f x* = *Some y* **by** *auto*
with *IH* **show** *P x y* **by** *auto*
qed
qed

lemma *fixp-induct-option*:
fixes *F* :: *'c* ⇒ *'c* **and**
U :: *'c* ⇒ *'b* ⇒ *'a option* **and**
C :: (*'b* ⇒ *'a option*) ⇒ *'c* **and**
P :: *'b* ⇒ *'a* ⇒ *bool*
assumes *mono*: ⋀*x*. *mono-option* (λ*f*. *U* (*F* (*C f*)) *x*)
assumes *eq*: *f* ≡ *C* (*ccpo.fixp* (*fun-lub* (*flat-lub None*)) (*fun-ord option-ord*) (λ*f*. *U* (*F* (*C f*))))
assumes *inverse2*: ⋀*f*. *U* (*C f*) = *f*
assumes *step*: ⋀*f x y*. (⋀*x y*. *U f x* = *Some y* ⟹ *P x y*) ⟹ *U* (*F f*) *x* = *Some y* ⟹ *P x y*
assumes *defined*: *U f x* = *Some y*
shows *P x y*
using *step defined option.fixp-induct-uc*[*of U F C*, *OF mono eq inverse2 option-admissible*]
unfolding *fun-lub-def flat-lub-def* **by**(*auto 9 2*)

declaration ‹*Partial-Function.init tailrec* ***term*** ‹*tailrec.fixp-fun*›
term ‹*tailrec.mono-body*› @{*thm tailrec.fixp-rule-uc*} @{*thm tailrec.fixp-induct-uc*}
(*SOME* @{*thm fixp-induct-tailrec*[*where c* = *undefined*]})›

declaration ‹*Partial-Function.init option* ***term*** ‹*option.fixp-fun*›
term ‹*option.mono-body*› @{*thm option.fixp-rule-uc*} @{*thm option.fixp-induct-uc*}
(*SOME* @{*thm fixp-induct-option*})›

hide-const (**open**) *chain*

end

theory *Argo*

imports *HOL*
begin

ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-expr.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-term.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-lit.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-proof.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-rewr.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-cls.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-common.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-cc.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-simplex.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-thy.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-heap.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-cdcl.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-core.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-clausify.ML*›
ML-file ‹~~/*src*/*Tools*/*Argo*/*argo-solver.ML*›

ML-file ‹*Tools*/*Argo*/*argo-tactic.ML*›

end

52 Reconstructing external resolution proofs for propositional logic

theory *SAT*
imports *Argo*
begin

ML-file ‹*Tools*/*prop-logic.ML*›
ML-file ‹*Tools*/*sat-solver.ML*›
ML-file ‹*Tools*/*sat.ML*›
ML-file ‹*Tools*/*Argo*/*argo-sat-solver.ML*›

method-setup *sat* = ‹*Scan.succeed* (*SIMPLE-METHOD′ o SAT.sat-tac*)›
SAT solver

method-setup *satx* = ‹*Scan.succeed* (*SIMPLE-METHOD′ o SAT.satx-tac*)›
SAT solver (*with definitional CNF*)

end

53 Function Definitions and Termination Proofs

theory *Fun-Def*
imports *Basic-BNF-LFPs Partial-Function SAT*
keywords

function termination :: *thy-goal-defn* **and**
fun fun-cases :: *thy-defn*
begin

53.1 Definitions with default value

definition *THE-default* :: $'a \Rightarrow ('a \Rightarrow bool) \Rightarrow 'a$
where *THE-default d P* = (*if* $(\exists!x.\ P\ x)$ *then* (*THE x. P x*) *else d*)

lemma *THE-defaultI'*: $\exists!x.\ P\ x \Longrightarrow P\ (THE\text{-}default\ d\ P)$
by (*simp add*: *theI' THE-default-def*)

lemma *THE-default1-equality*: $\exists!x.\ P\ x \Longrightarrow P\ a \Longrightarrow THE\text{-}default\ d\ P = a$
by (*simp add*: *the1-equality THE-default-def*)

lemma *THE-default-none*: $\neg\ (\exists!x.\ P\ x) \Longrightarrow THE\text{-}default\ d\ P = d$
by (*simp add*: *THE-default-def*)

lemma *fundef-ex1-existence*:
assumes *f-def*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $G\ x\ (f\ x)$
apply (*simp only*: *f-def*)
apply (*rule THE-defaultI'*)
apply (*rule ex1*)
done

lemma *fundef-ex1-uniqueness*:
assumes *f-def*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
assumes *elm*: $G\ x\ (h\ x)$
shows $h\ x = f\ x$
by (*auto simp add*: *f-def ex1 elm THE-default1-equality*[*symmetric*])

lemma *fundef-ex1-iff*:
assumes *f-def*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *ex1*: $\exists!y.\ G\ x\ y$
shows $(G\ x\ y) = (f\ x = y)$
by (*auto simp add*: *ex1 f-def THE-default1-equality THE-defaultI'*)

lemma *fundef-default-value*:
assumes *f-def*: $f \equiv (\lambda x::'a.\ THE\text{-}default\ (d\ x)\ (\lambda y.\ G\ x\ y))$
assumes *graph*: $\bigwedge x\ y.\ G\ x\ y \Longrightarrow D\ x$
assumes $\neg\ D\ x$
shows $f\ x = d\ x$
proof −
have $\neg(\exists\, y.\ G\ x\ y)$

proof
assume $\exists y.\ G\ x\ y$
then have $D\ x$ **using** *graph* **..**
with ‹$\neg\ D\ x$› **show** *False* **..**
qed
then have $\neg(\exists!y.\ G\ x\ y)$ **by** (*rule THE-default-none*)
qed

definition *in-rel-def*[*simp*]: *in-rel R x y* ≡ $(x,\ y) \in R$

lemma *wf-in-rel*: *wf R* ⟹ *wfp* (*in-rel R*)
by (*simp add*: *wfp-def*)

ML-file ‹*Tools/Function/function-core.ML*›
ML-file ‹*Tools/Function/mutual.ML*›
ML-file ‹*Tools/Function/pattern-split.ML*›
ML-file ‹*Tools/Function/relation.ML*›
ML-file ‹*Tools/Function/function-elims.ML*›

method-setup *relation* = ‹
Args.term >> (fn t => fn ctxt => SIMPLE-METHOD' (Function-Relation.relation-infer-tac ctxt t))
› *prove termination using a user−specified wellfounded relation*

ML-file ‹*Tools/Function/function.ML*›
ML-file ‹*Tools/Function/pat-completeness.ML*›

method-setup *pat-completeness* = ‹
Scan.succeed (SIMPLE-METHOD' o Pat-Completeness.pat-completeness-tac)
› *prove completeness of (co)datatype patterns*

ML-file ‹*Tools/Function/fun.ML*›
ML-file ‹*Tools/Function/induction-schema.ML*›

method-setup *induction-schema* = ‹
Scan.succeed (CONTEXT-TACTIC oo Induction-Schema.induction-schema-tac)
› *prove an induction principle*

53.2 Measure functions

inductive *is-measure* :: $('a \Rightarrow nat) \Rightarrow bool$
where *is-measure-trivial*: *is-measure f*

named-theorems *measure-function rules that guide the heuristic generation of measure functions*
ML-file ‹*Tools/Function/measure-functions.ML*›

lemma *measure-size*[*measure-function*]: *is-measure size*
by (*rule is-measure-trivial*)

lemma *measure-fst*[*measure-function*]: *is-measure f* ⟹ *is-measure* (λ*p*. *f* (*fst p*))
by (*rule is-measure-trivial*)

lemma *measure-snd*[*measure-function*]: *is-measure f* ⟹ *is-measure* (λ*p*. *f* (*snd p*))
by (*rule is-measure-trivial*)

ML-file ‹*Tools/Function/lexicographic-order.ML*›

method-setup *lexicographic-order* = ‹
Method.sections clasimp-modifiers >>
(*K* (*SIMPLE-METHOD o Lexicographic-Order.lexicographic-order-tac false*))
› *termination prover for lexicographic orderings*

53.3 Congruence rules

lemma *let-cong* [*fundef-cong*]: *M* = *N* ⟹ (⋀*x*. *x* = *N* ⟹ *f x* = *g x*) ⟹ *Let M f* = *Let N g*
unfolding *Let-def* **by** *blast*

lemmas [*fundef-cong*] =
if-cong image-cong
bex-cong ball-cong imp-cong map-option-cong Option.bind-cong

lemma *split-cong* [*fundef-cong*]:
(⋀*x y*. (*x*, *y*) = *q* ⟹ *f x y* = *g x y*) ⟹ *p* = *q* ⟹ *case-prod f p* = *case-prod g q*
by (*auto simp*: *split-def*)

lemma *comp-cong* [*fundef-cong*]: *f* (*g x*) = *f'* (*g' x'*) ⟹ (*f* ∘ *g*) *x* = (*f'* ∘ *g'*) *x'*
by (*simp only*: *o-apply*)

53.4 Simp rules for termination proofs

declare
trans-less-add1[*termination-simp*]
trans-less-add2[*termination-simp*]
trans-le-add1[*termination-simp*]
trans-le-add2[*termination-simp*]
less-imp-le-nat[*termination-simp*]
le-imp-less-Suc[*termination-simp*]

lemma *size-prod-simp*[*termination-simp*]: *size-prod f g p* = *f* (*fst p*) + *g* (*snd p*) + *Suc 0*
by (*induct p*) *auto*

53.5 Decomposition

lemma *less-by-empty*: $A = \{\} \Longrightarrow A \subseteq B$
  and *union-comp-emptyL*: $A\ O\ C = \{\} \Longrightarrow B\ O\ C = \{\} \Longrightarrow (A \cup B)\ O\ C = \{\}$
  and *union-comp-emptyR*: $A\ O\ B = \{\} \Longrightarrow A\ O\ C = \{\} \Longrightarrow A\ O\ (B \cup C) = \{\}$
  and *wf-no-loop*: $R\ O\ R = \{\} \Longrightarrow wf\ R$
  by (*auto simp add*: *wf-comp-self* [*of R*])

53.6 Reduction pairs

definition *reduction-pair* $P \longleftrightarrow wf\ (fst\ P) \wedge fst\ P\ O\ snd\ P \subseteq fst\ P$

lemma *reduction-pairI*[*intro*]: $wf\ R \Longrightarrow R\ O\ S \subseteq R \Longrightarrow$ *reduction-pair* (R, S)
  by (*auto simp*: *reduction-pair-def*)

lemma *reduction-pair-lemma*:
  assumes *rp*: *reduction-pair P*
  assumes $R \subseteq fst\ P$
  assumes $S \subseteq snd\ P$
  assumes *wf S*
  shows *wf* $(R \cup S)$
proof −
  from *rp* ‹$S \subseteq snd\ P$› **have** *wf* $(fst\ P)$ $fst\ P\ O\ S \subseteq fst\ P$
    unfolding *reduction-pair-def* **by** *auto*
  with ‹*wf S*› **have** *wf* $(fst\ P \cup S)$
    by (*auto intro*: *wf-union-compatible*)
  moreover from ‹$R \subseteq fst\ P$› **have** $R \cup S \subseteq fst\ P \cup S$ **by** *auto*
  ultimately show *?thesis* **by** (*rule wf-subset*)
qed

definition *rp-inv-image* = $(\lambda(R,S)\ f.\ (inv\text{-}image\ R\ f,\ inv\text{-}image\ S\ f))$

lemma *rp-inv-image-rp*: *reduction-pair P* $\Longrightarrow$ *reduction-pair* (*rp-inv-image P f*)
  unfolding *reduction-pair-def rp-inv-image-def split-def* **by** *force*

53.7 Concrete orders for SCNP termination proofs

definition *pair-less* = *less-than* $<*lex*>$ *less-than*
definition *pair-leq* = *pair-less*$^=$
definition *max-strict* = *max-ext pair-less*
definition *max-weak* = *max-ext pair-leq* $\cup$ $\{(\{\}, \{\})\}$
definition *min-strict* = *min-ext pair-less*
definition *min-weak* = *min-ext pair-leq* $\cup$ $\{(\{\}, \{\})\}$

lemma *wf-pair-less*[*simp*]: *wf pair-less*
  by (*auto simp*: *pair-less-def*)

lemma *total-pair-less* [*iff*]: *total-on A pair-less* **and** *trans-pair-less* [*iff*]: *trans pair-less*
  by (*auto simp*: *total-on-def pair-less-def*)

Introduction rules for *pair-less/pair-leq*

lemma *pair-leqI1*: $a < b \Longrightarrow ((a,\ s),\ (b,\ t)) \in$ *pair-leq*
and *pair-leqI2*: $a \leq b \Longrightarrow s \leq t \Longrightarrow ((a,\ s),\ (b,\ t)) \in$ *pair-leq*
and *pair-lessI1*: $a < b \Longrightarrow ((a,\ s),\ (b,\ t)) \in$ *pair-less*
and *pair-lessI2*: $a \leq b \Longrightarrow s < t \Longrightarrow ((a,\ s),\ (b,\ t)) \in$ *pair-less*
by (*auto simp*: *pair-leq-def pair-less-def*)

lemma *pair-less-iff1* [*simp*]: $((x,y),\ (x,z)) \in$ *pair-less* $\longleftrightarrow$ $y<z$
by (*simp add*: *pair-less-def*)

Introduction rules for max

lemma *smax-emptyI*: *finite* $Y \Longrightarrow Y \neq \{\} \Longrightarrow (\{\},\ Y) \in$ *max-strict*
and *smax-insertI*:
$y \in Y \Longrightarrow (x,\ y) \in$ *pair-less* $\Longrightarrow (X,\ Y) \in$ *max-strict* $\Longrightarrow$ (*insert* $x\ X,\ Y) \in$ *max-strict*
and *wmax-emptyI*: *finite* $X \Longrightarrow (\{\},\ X) \in$ *max-weak*
and *wmax-insertI*:
$y \in YS \Longrightarrow (x,\ y) \in$ *pair-leq* $\Longrightarrow (XS,\ YS) \in$ *max-weak* $\Longrightarrow$ (*insert* $x\ XS,\ YS) \in$ *max-weak*
by (*auto simp*: *max-strict-def max-weak-def elim*!: *max-ext.cases*)

Introduction rules for min

lemma *smin-emptyI*: $X \neq \{\} \Longrightarrow (X,\ \{\}) \in$ *min-strict*
and *smin-insertI*:
$x \in XS \Longrightarrow (x,\ y) \in$ *pair-less* $\Longrightarrow (XS,\ YS) \in$ *min-strict* $\Longrightarrow$ (XS, *insert* y $YS) \in$ *min-strict*
and *wmin-emptyI*: $(X,\ \{\}) \in$ *min-weak*
and *wmin-insertI*:
$x \in XS \Longrightarrow (x,\ y) \in$ *pair-leq* $\Longrightarrow (XS,\ YS) \in$ *min-weak* $\Longrightarrow$ (XS, *insert* y $YS) \in$ *min-weak*
by (*auto simp*: *min-strict-def min-weak-def min-ext-def*)

Reduction Pairs.

lemma *max-ext-compat*:
assumes $R\ O\ S \subseteq R$
shows *max-ext* $R\ O$ (*max-ext* $S \cup \{(\{\},\ \{\})\}) \subseteq$ *max-ext* R
proof −
have $\bigwedge X\ Y\ Z.\ (X,\ Y) \in$ *max-ext* $R \Longrightarrow (Y,\ Z) \in$ *max-ext* $S \Longrightarrow (X,\ Z) \in$ *max-ext* R
proof −
fix $X\ Y\ Z$
assume $(X,Y) \in$*max-ext* R
$(Y,\ Z) \in$*max-ext* S
then have $*$: *finite* X *finite* Y *finite* Z $Y \neq \{\}$ $Z \neq \{\}$
$(\bigwedge x.\ x \in X \Longrightarrow \exists\, y \in Y.\ (x,\ y) \in R)$
$(\bigwedge y.\ y \in Y \Longrightarrow \exists\, z \in Z.\ (y,\ z) \in S)$
by (*auto elim*: *max-ext.cases*)
moreover have $\bigwedge x.\ x \in X \Longrightarrow \exists\, z \in Z.\ (x,\ z) \in R$

```
    proof −
      fix x
      assume x∈X
      then obtain y where 1: y∈Y (x, y)∈R
        using ∗ by auto
      then obtain z where z∈Z (y, z)∈S
        using ∗ by auto
      then show ∃ z∈Z. (x, z)∈R
        using assms 1 by (auto elim: max-ext.cases)
    qed
    ultimately show (X,Z)∈max-ext R
      by auto
  qed
  then show max-ext R O (max-ext S ∪ {({}, {})}) ⊆ max-ext R
    by auto
qed

lemma max-rpair-set: reduction-pair (max-strict, max-weak)
  unfolding max-strict-def max-weak-def
  apply (intro reduction-pairI max-ext-wf)
   apply simp
  apply (rule max-ext-compat)
  apply (auto simp: pair-less-def pair-leq-def)
  done

lemma min-ext-compat:
  assumes R O S ⊆ R
  shows min-ext R O (min-ext S ∪ {({},{})}) ⊆ min-ext R
proof −
  have ⋀X Y Z z. ∀ y∈Y. ∃ x∈X. (x, y) ∈ R ⟹ ∀ z∈Z. ∃ y∈Y. (y, z) ∈ S
  ⟹ z ∈ Z ⟹ ∃ x∈X. (x, z) ∈ R
  proof −
    fix X Y Z z
    assume ∗: ∀ y∈Y. ∃ x∈X. (x, y) ∈ R
      ∀ z∈Z. ∃ y∈Y. (y, z) ∈ S
      z∈Z
    then obtain y′ where 1: y′∈Y (y′, z) ∈ S
      by auto
    then obtain x′ where 2: x′∈X (x′, y′) ∈ R
      using ∗ by auto
    show ∃ x∈X. (x, z) ∈ R
      using 1 2 assms by auto
  qed
  then show ?thesis
    using assms by (auto simp: min-ext-def)
qed

lemma min-rpair-set: reduction-pair (min-strict, min-weak)
  unfolding min-strict-def min-weak-def
```

apply (*intro reduction-pairI min-ext-wf*)
apply *simp*
apply (*rule min-ext-compat*)
apply (*auto simp*: *pair-less-def pair-leq-def*)
done

53.8 Yet more induction principles on the natural numbers

lemma *nat-descend-induct* [*case-names base descend*]:
fixes $P :: nat \Rightarrow bool$
assumes *H1*: $\bigwedge k.\ k > n \Longrightarrow P\ k$
assumes *H2*: $\bigwedge k.\ k \leq n \Longrightarrow (\bigwedge i.\ i > k \Longrightarrow P\ i) \Longrightarrow P\ k$
shows $P\ m$
using *assms* **by** *induction-schema* (*force intro*!: *wf-measure* [*of* $\lambda k.\ Suc\ n - k$])+

lemma *induct-nat-012*[*case-names 0 1 ge2*]:
$P\ 0 \Longrightarrow P\ (Suc\ 0) \Longrightarrow (\bigwedge n.\ P\ n \Longrightarrow P\ (Suc\ n) \Longrightarrow P\ (Suc\ (Suc\ n))) \Longrightarrow P\ n$
proof (*induction-schema*, *pat-completeness*)
show *wf* (*Wellfounded.measure id*)
by *simp*
qed *auto*

53.9 Tool setup

ML-file ‹*Tools/Function/termination.ML*›
ML-file ‹*Tools/Function/scnp-solve.ML*›
ML-file ‹*Tools/Function/scnp-reconstruct.ML*›
ML-file ‹*Tools/Function/fun-cases.ML*›

ML-val — setup inactive
‹
Context.theory-map (*Function-Common.set-termination-prover*
(*K* (*ScnpReconstruct.decomp-scnp-tac* [*ScnpSolve.MAX*, *ScnpSolve.MIN*, *ScnpSolve.MS*])))
›

end

54 The Integers as Equivalence Classes over Pairs of Natural Numbers

theory *Int*
imports *Quotient Groups-Big Fun-Def*
begin

54.1 Definition of integers as a quotient type

definition *intrel* :: $(nat \times nat) \Rightarrow (nat \times nat) \Rightarrow bool$
where *intrel* = $(\lambda(x,\ y)\ (u,\ v).\ x + v = u + y)$

lemma *intrel-iff* [*simp*]: *intrel* (x, y) $(u, v) \longleftrightarrow x + v = u + y$
by (*simp add*: *intrel-def*)

quotient-type *int* = *nat* × *nat* / *intrel*
morphisms *Rep-Integ Abs-Integ*
proof (*rule equivpI*)
show *reflp intrel* **by** (*auto simp*: *reflp-def*)
show *symp intrel* **by** (*auto simp*: *symp-def*)
show *transp intrel* **by** (*auto simp*: *transp-def*)
qed

54.2 Integers form a commutative ring

instantiation *int* :: *comm-ring-1*
begin

lift-definition *zero-int* :: *int* **is** $(0, 0)$ **.**

lift-definition *one-int* :: *int* **is** $(1, 0)$ **.**

lift-definition *plus-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x, y)\ (u, v).\ (x + u, y + v)$
by *clarsimp*

lift-definition *uminus-int* :: *int* ⇒ *int*
is $\lambda(x, y).\ (y, x)$
by *clarsimp*

lift-definition *minus-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x, y)\ (u, v).\ (x + v, y + u)$
by *clarsimp*

lift-definition *times-int* :: *int* ⇒ *int* ⇒ *int*
is $\lambda(x, y)\ (u, v).\ (x*u + y*v, x*v + y*u)$
proof (*unfold intrel-def*, *clarify*)
fix *s t u v w x y z* :: *nat*
assume $s + v = u + t$ **and** $w + z = y + x$
then have $(s + v) * w + (u + t) * x + u * (w + z) + v * (y + x) =$
$(u + t) * w + (s + v) * x + u * (y + x) + v * (w + z)$
by *simp*
then show $(s * w + t * x) + (u * z + v * y) = (u * y + v * z) + (s * x + t *$
$w)$
by (*simp add*: *algebra-simps*)
qed

instance
by *standard* (*transfer*; *clarsimp simp*: *algebra-simps*)+

end

abbreviation *int* :: *nat* ⇒ *int*
  where *int* ≡ *of-nat*

lemma *int-def*: *int n* = *Abs-Integ* (*n*, *0*)
  by (*induct n*) (*simp add*: *zero-int.abs-eq*, *simp add*: *one-int.abs-eq plus-int.abs-eq*)

lemma *int-transfer* [*transfer-rule*]:
  includes *lifting-syntax*
  shows *rel-fun* (=) *pcr-int* (λ*n*. (*n*, *0*)) *int*
  by (*simp add*: *rel-fun-def int.pcr-cr-eq cr-int-def int-def*)

lemma *int-diff-cases*: **obtains** (*diff*) *m n* **where** *z* = *int m* − *int n*
  by *transfer clarsimp*

54.3 Integers are totally ordered

instantiation *int* :: *linorder*
begin

lift-definition *less-eq-int* :: *int* ⇒ *int* ⇒ *bool*
  is λ(*x*, *y*) (*u*, *v*). *x* + *v* ≤ *u* + *y*
  by *auto*

lift-definition *less-int* :: *int* ⇒ *int* ⇒ *bool*
  is λ(*x*, *y*) (*u*, *v*). *x* + *v* < *u* + *y*
  by *auto*

instance
  by *standard* (*transfer*, *force*)+

end

instantiation *int* :: *distrib-lattice*
begin

definition (*inf* :: *int* ⇒ *int* ⇒ *int*) = *min*

definition (*sup* :: *int* ⇒ *int* ⇒ *int*) = *max*

instance
  by *standard* (*auto simp add*: *inf-int-def sup-int-def max-min-distrib2*)

end

54.4 Ordering properties of arithmetic operations

instance *int* :: *ordered-cancel-ab-semigroup-add*
proof

fix *i j k* :: *int*
show $i \leq j \Longrightarrow k + i \leq k + j$
by *transfer clarsimp*
qed

Strict Monotonicity of Multiplication.

Strict, in 1st argument; proof is by induction on $k > 0$.

lemma *zmult-zless-mono2-lemma*: $i < j \Longrightarrow 0 < k \Longrightarrow int\ k * i < int\ k * j$
for *i j* :: *int*
proof (*induct k*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc k*)
then show *?case*
by (*cases* $k = 0$) (*simp-all add*: *distrib-right add-strict-mono*)
qed

lemma *zero-le-imp-eq-int*:
assumes $k \geq (0::int)$ **shows** $\exists n.\ k = int\ n$
proof −
have $b \leq a \Longrightarrow \exists n::nat.\ a = n + b$ **for** *a b*
using *exI*[*of* - $a - b$] **by** *simp*
with *assms* **show** *?thesis*
by *transfer auto*
qed

lemma *zero-less-imp-eq-int*:
assumes $k > (0::int)$ **shows** $\exists n>0.\ k = int\ n$
proof −
have $b < a \Longrightarrow \exists n::nat.\ n>0 \wedge a = n + b$ **for** *a b*
using *exI*[*of* - $a - b$] **by** *simp*
with *assms* **show** *?thesis*
by *transfer auto*
qed

lemma *zmult-zless-mono2*: $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
for *i j k* :: *int*
by (*drule zero-less-imp-eq-int*) (*auto simp add*: *zmult-zless-mono2-lemma*)

The integers form an ordered integral domain.

instantiation *int* :: *linordered-idom*
begin

definition *zabs-def*: $|i::int| = (if\ i < 0\ then - i\ else\ i)$

definition *zsgn-def*: $sgn\ (i::int) = (if\ i = 0\ then\ 0\ else\ if\ 0 < i\ then\ 1\ else - 1)$

instance
proof
  fix *i j k* :: *int*
  show $i < j \Longrightarrow 0 < k \Longrightarrow k * i < k * j$
    by (*rule zmult-zless-mono2*)
  show $|i| = (\mathit{if}\ i < 0\ \mathit{then}\ -i\ \mathit{else}\ i)$
    by (*simp only*: *zabs-def*)
  show *sgn* (*i*::*int*) = (*if i=0 then 0 else if 0<i then 1 else − 1*)
    by (*simp only*: *zsgn-def*)
qed

end

instance *int* :: *discrete-linordered-semidom*
proof
  fix *k l* :: *int*
  show ‹$k < l \longleftrightarrow k + 1 \leq l$› (**is** ‹*?P* $\longleftrightarrow$ *?Q*›)
  proof
    assume *?Q*
    then show *?P*
      by *simp*
  next
    assume *?P*
    then have ‹$l - k > 0$›
      by *simp*
    with *zero-less-imp-eq-int* **obtain** *n* **where** ‹$l - k = \mathit{int}\ n$›
      by *blast*
    then have ‹$n > 0$›
      using ‹$l - k > 0$› **by** *simp*
    then have ‹$n \geq 1$›
      by *simp*
    then have ‹$\mathit{int}\ n \geq \mathit{int}\ 1$›
      by (*rule of-nat-mono*)
    with ‹$l - k = \mathit{int}\ n$› **show** *?Q*
      by (*simp add*: *algebra-simps*)
  qed
qed

lemma *zless-imp-add1-zle*: $w < z \Longrightarrow w + 1 \leq z$
  for *w z* :: *int*
  by *transfer clarsimp*

lemma *zless-iff-Suc-zadd*: $w < z \longleftrightarrow (\exists\, n.\ z = w + \mathit{int}\ (\mathit{Suc}\ n))$
  for *w z* :: *int*
proof −
  have $\bigwedge a\ b\ c\ d.\ a + d < c + b \Longrightarrow \exists\, n.\ c + b = \mathit{Suc}\ (a + n + d)$
  proof −
    fix *a b c d* :: *nat*
    assume $a + d < c + b$

```
    then have c + b = Suc (a + (c + b - Suc (a + d)) + d)
      by arith
    then show ∃n. c + b = Suc (a + n + d)
      by (rule exI)
  qed
  then show ?thesis
    by transfer auto
qed

lemma zabs-less-one-iff [simp]: |z| < 1 ⟷ z = 0 (is ?lhs ⟷ ?rhs)
  for z :: int
proof
  assume ?rhs
  then show ?lhs by simp
next
  assume ?lhs
  with zless-imp-add1-zle [of |z| 1] have |z| + 1 ≤ 1 by simp
  then have |z| ≤ 0 by simp
  then show ?rhs by simp
qed
```

54.5 Embedding of the Integers into any *ring-1*: *of-int*

```
context ring-1
begin

lift-definition of-int :: int ⇒ 'a
  is λ(i, j). of-nat i - of-nat j
  by (clarsimp simp add: diff-eq-eq eq-diff-eq diff-add-eq
      of-nat-add [symmetric] simp del: of-nat-add)

lemma of-int-0 [simp]: of-int 0 = 0
  by transfer simp

lemma of-int-1 [simp]: of-int 1 = 1
  by transfer simp

lemma of-int-add [simp]: of-int (w + z) = of-int w + of-int z
  by transfer (clarsimp simp add: algebra-simps)

lemma of-int-minus [simp]: of-int (- z) = - (of-int z)
  by (transfer fixing: uminus) clarsimp

lemma of-int-diff [simp]: of-int (w - z) = of-int w - of-int z
  using of-int-add [of w - z] by simp

lemma of-int-mult [simp]: of-int (w*z) = of-int w * of-int z
  by (transfer fixing: times) (clarsimp simp add: algebra-simps)
```

lemma *mult-of-int-commute*: *of-int x * y = y * of-int x*
by (*transfer fixing*: *times*) (*auto simp*: *algebra-simps mult-of-nat-commute*)

Collapse nested embeddings.

lemma *of-int-of-nat-eq* [*simp*]: *of-int (int n) = of-nat n*
by (*induct n*) *auto*

lemma *of-int-numeral* [*simp*, *code-post*]: *of-int (numeral k) = numeral k*
by (*simp add*: *of-nat-numeral* [*symmetric*] *of-int-of-nat-eq* [*symmetric*])

lemma *of-int-neg-numeral* [*code-post*]: *of-int (- numeral k) = - numeral k*
by *simp*

lemma *of-int-power* [*simp*]: *of-int (z ^ n) = of-int z ^ n*
by (*induct n*) *simp-all*

lemma *of-int-of-bool* [*simp*]:
of-int (of-bool P) = of-bool P
by *auto*

end

context *ring-char-0*
begin

lemma *of-int-eq-iff* [*simp*]: *of-int w = of-int z ⟷ w = z*
by *transfer* (*clarsimp simp add*: *algebra-simps of-nat-add* [*symmetric*] *simp del*: *of-nat-add*)

Special cases where either operand is zero.

lemma *of-int-eq-0-iff* [*simp*]: *of-int z = 0 ⟷ z = 0*
using *of-int-eq-iff* [*of z 0*] **by** *simp*

lemma *of-int-0-eq-iff* [*simp*]: *0 = of-int z ⟷ z = 0*
using *of-int-eq-iff* [*of 0 z*] **by** *simp*

lemma *of-int-eq-1-iff* [*iff*]: *of-int z = 1 ⟷ z = 1*
using *of-int-eq-iff* [*of z 1*] **by** *simp*

lemma *of-nat-of-int-iff*: *of-int i = of-nat n ⟷ i = of-nat n of-nat n = of-int i ⟷ i = of-nat n*
by (*metis of-int-eq-iff of-int-of-nat-eq*)+

lemma *numeral-power-eq-of-int-cancel-iff* [*simp*]:
numeral x ^ n = of-int y ⟷ numeral x ^ n = y
using *of-int-eq-iff*[*of numeral x ^ n y*, *unfolded of-int-numeral of-int-power*] **.**

lemma *of-int-eq-numeral-power-cancel-iff* [*simp*]:
of-int y = numeral x ^ n ⟷ y = numeral x ^ n

using *numeral-power-eq-of-int-cancel-iff* [*of x n y*] **by** (*metis* (*mono-tags*))

lemma *neg-numeral-power-eq-of-int-cancel-iff* [*simp*]:
(− *numeral x*) ^ *n* = *of-int y* ⟷ (− *numeral x*) ^ *n* = *y*
using *of-int-eq-iff*[*of* (− *numeral x*) ^ *n y*]
by *simp*

lemma *of-int-eq-neg-numeral-power-cancel-iff* [*simp*]:
of-int y = (− *numeral x*) ^ *n* ⟷ *y* = (− *numeral x*) ^ *n*
using *neg-numeral-power-eq-of-int-cancel-iff*[*of x n y*] **by** (*metis* (*mono-tags*))

lemma *of-int-eq-of-int-power-cancel-iff*[*simp*]: (*of-int b*) ^ *w* = *of-int x* ⟷ *b* ^ *w* = *x*
by (*metis of-int-power of-int-eq-iff*)

lemma *of-int-power-eq-of-int-cancel-iff*[*simp*]: *of-int x* = (*of-int b*) ^ *w* ⟷ *x* = *b* ^ *w*
by (*metis of-int-eq-of-int-power-cancel-iff*)

end

context *linordered-idom*
begin

Every *linordered-idom* has characteristic zero.

subclass *ring-char-0* **..**

lemma *of-int-le-iff* [*simp*]: *of-int w* ≤ *of-int z* ⟷ *w* ≤ *z*
by (*transfer fixing*: *less-eq*)
(*clarsimp simp add*: *algebra-simps of-nat-add* [*symmetric*] *simp del*: *of-nat-add*)

lemma *of-int-less-iff* [*simp*]: *of-int w* < *of-int z* ⟷ *w* < *z*
by (*simp add*: *less-le order-less-le*)

lemma *of-int-0-le-iff* [*simp*]: *0* ≤ *of-int z* ⟷ *0* ≤ *z*
using *of-int-le-iff* [*of 0 z*] **by** *simp*

lemma *of-int-le-0-iff* [*simp*]: *of-int z* ≤ *0* ⟷ *z* ≤ *0*
using *of-int-le-iff* [*of z 0*] **by** *simp*

lemma *of-int-0-less-iff* [*simp*]: *0* < *of-int z* ⟷ *0* < *z*
using *of-int-less-iff* [*of 0 z*] **by** *simp*

lemma *of-int-less-0-iff* [*simp*]: *of-int z* < *0* ⟷ *z* < *0*
using *of-int-less-iff* [*of z 0*] **by** *simp*

lemma *of-int-1-le-iff* [*simp*]: *1* ≤ *of-int z* ⟷ *1* ≤ *z*
using *of-int-le-iff* [*of 1 z*] **by** *simp*

lemma *of-int-le-1-iff* [*simp*]: *of-int z* $\leq$ *1* $\longleftrightarrow$ *z* $\leq$ *1*
using *of-int-le-iff* [*of z 1*] **by** *simp*

lemma *of-int-1-less-iff* [*simp*]: *1* $<$ *of-int z* $\longleftrightarrow$ *1* $<$ *z*
using *of-int-less-iff* [*of 1 z*] **by** *simp*

lemma *of-int-less-1-iff* [*simp*]: *of-int z* $<$ *1* $\longleftrightarrow$ *z* $<$ *1*
using *of-int-less-iff* [*of z 1*] **by** *simp*

lemma *of-int-pos*: *z* $>$ *0* $\Longrightarrow$ *of-int z* $>$ *0*
by *simp*

lemma *of-int-nonneg*: *z* $\geq$ *0* $\Longrightarrow$ *of-int z* $\geq$ *0*
by *simp*

lemma *of-int-abs* [*simp*]: *of-int* $|x|$ = $|$*of-int x*$|$
by (*auto simp add*: *abs-if*)

lemma *of-int-lessD*:
assumes $|$*of-int n*$|$ $<$ *x*
shows *n* = *0* $\vee$ *x* $>$ *1*
proof (*cases n* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $|n|$ $\neq$ *0* **by** *simp*
then have $|n|$ $>$ *0* **by** *simp*
then have $|n|$ $\geq$ *1*
using *zless-imp-add1-zle* [*of 0* $|n|$] **by** *simp*
then have $|$*of-int n*$|$ $\geq$ *1*
unfolding *of-int-1-le-iff* [*of* $|n|$, *symmetric*] **by** *simp*
then have *1* $<$ *x* **using** *assms* **by** (*rule le-less-trans*)
then show *?thesis* **..**
qed

lemma *of-int-leD*:
assumes $|$*of-int n*$|$ $\leq$ *x*
shows *n* = *0* $\vee$ *1* $\leq$ *x*
proof (*cases n* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $|n|$ $\neq$ *0* **by** *simp*
then have $|n|$ $>$ *0* **by** *simp*
then have $|n|$ $\geq$ *1*
using *zless-imp-add1-zle* [*of 0* $|n|$] **by** *simp*
then have $|$*of-int n*$|$ $\geq$ *1*

unfolding *of-int-1-le-iff* [*of* |*n*|, *symmetric*] **by** *simp*
then have $1 \leq x$ **using** *assms* **by** (*rule order-trans*)
then show *?thesis* **..**
qed

lemma *numeral-power-le-of-int-cancel-iff* [*simp*]:
$numeral\ x\ \hat{}\ n \leq of\text{-}int\ a \longleftrightarrow numeral\ x\ \hat{}\ n \leq a$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-le-iff*)

lemma *of-int-le-numeral-power-cancel-iff* [*simp*]:
$of\text{-}int\ a \leq numeral\ x\ \hat{}\ n \longleftrightarrow a \leq numeral\ x\ \hat{}\ n$
by (*metis* (*mono-tags*) *local.numeral-power-eq-of-int-cancel-iff of-int-le-iff*)

lemma *numeral-power-less-of-int-cancel-iff* [*simp*]:
$numeral\ x\ \hat{}\ n < of\text{-}int\ a \longleftrightarrow numeral\ x\ \hat{}\ n < a$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-less-iff*)

lemma *of-int-less-numeral-power-cancel-iff* [*simp*]:
$of\text{-}int\ a < numeral\ x\ \hat{}\ n \longleftrightarrow a < numeral\ x\ \hat{}\ n$
by (*metis* (*mono-tags*) *local.of-int-eq-numeral-power-cancel-iff of-int-less-iff*)

lemma *neg-numeral-power-le-of-int-cancel-iff* [*simp*]:
$(-\ numeral\ x)\ \hat{}\ n \leq of\text{-}int\ a \longleftrightarrow (-\ numeral\ x)\ \hat{}\ n \leq a$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-neg-numeral of-int-power*)

lemma *of-int-le-neg-numeral-power-cancel-iff* [*simp*]:
$of\text{-}int\ a \leq (-\ numeral\ x)\ \hat{}\ n \longleftrightarrow a \leq (-\ numeral\ x)\ \hat{}\ n$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-neg-numeral of-int-power*)

lemma *neg-numeral-power-less-of-int-cancel-iff* [*simp*]:
$(-\ numeral\ x)\ \hat{}\ n < of\text{-}int\ a \longleftrightarrow (-\ numeral\ x)\ \hat{}\ n < a$
using *of-int-less-iff*[*of* $(-\ numeral\ x)\ \hat{}\ n\ a$]
by *simp*

lemma *of-int-less-neg-numeral-power-cancel-iff* [*simp*]:
$of\text{-}int\ a < (-\ numeral\ x)\ \hat{}\ n \longleftrightarrow a < (-\ numeral\ x{::}int)\ \hat{}\ n$
using *of-int-less-iff*[*of* $a\ (-\ numeral\ x)\ \hat{}\ n$]
by *simp*

lemma *of-int-le-of-int-power-cancel-iff*[*simp*]: $(of\text{-}int\ b)\ \hat{}\ w \leq of\text{-}int\ x \longleftrightarrow b\ \hat{}\ w \leq x$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-power*)

lemma *of-int-power-le-of-int-cancel-iff*[*simp*]: $of\text{-}int\ x \leq (of\text{-}int\ b)\ \hat{}\ w \longleftrightarrow x \leq b\ \hat{}\ w$
by (*metis* (*mono-tags*) *of-int-le-iff of-int-power*)

lemma *of-int-less-of-int-power-cancel-iff*[*simp*]: $(of\text{-}int\ b)\ \hat{}\ w < of\text{-}int\ x \longleftrightarrow b\ \hat{}\ w < x$

by (*metis* (*mono-tags*) *of-int-less-iff of-int-power*)

lemma *of-int-power-less-of-int-cancel-iff*[*simp*]: *of-int x* < (*of-int b*) ^ *w*⟷ *x* < *b* ^ *w*
by (*metis* (*mono-tags*) *of-int-less-iff of-int-power*)

lemma *of-int-max*: *of-int* (*max x y*) = *max* (*of-int x*) (*of-int y*)
by (*auto simp*: *max-def*)

lemma *of-int-min*: *of-int* (*min x y*) = *min* (*of-int x*) (*of-int y*)
by (*auto simp*: *min-def*)

end

context *division-ring*
begin

lemmas *mult-inverse-of-int-commute* =
mult-commute-imp-mult-inverse-commute[*OF mult-of-int-commute*]

end

Comparisons involving *of-int*.

lemma *of-int-eq-numeral-iff* [*iff*]: *of-int z* = (*numeral n* :: *'a*::*ring-char-0*) ⟷ *z* = *numeral n*
using *of-int-eq-iff* **by** *fastforce*

lemma *of-int-le-numeral-iff* [*simp*]:
of-int z ≤ (*numeral n* :: *'a*::*linordered-idom*) ⟷ *z* ≤ *numeral n*
using *of-int-le-iff* [*of z numeral n*] **by** *simp*

lemma *of-int-numeral-le-iff* [*simp*]:
(*numeral n* :: *'a*::*linordered-idom*) ≤ *of-int z* ⟷ *numeral n* ≤ *z*
using *of-int-le-iff* [*of numeral n*] **by** *simp*

lemma *of-int-less-numeral-iff* [*simp*]:
of-int z < (*numeral n* :: *'a*::*linordered-idom*) ⟷ *z* < *numeral n*
using *of-int-less-iff* [*of z numeral n*] **by** *simp*

lemma *of-int-numeral-less-iff* [*simp*]:
(*numeral n* :: *'a*::*linordered-idom*) < *of-int z* ⟷ *numeral n* < *z*
using *of-int-less-iff* [*of numeral n z*] **by** *simp*

lemma *of-nat-less-of-int-iff*: (*of-nat n*::*'a*::*linordered-idom*) < *of-int x* ⟷ *int n* < *x*
by (*metis of-int-of-nat-eq of-int-less-iff*)

lemma *of-int-eq-id* [*simp*]: *of-int* = *id*
proof

show *of-int z = id z* **for** *z*
by (*cases z rule*: *int-diff-cases*) *simp*
qed

instance *int* :: *no-top*
proof
fix *x*::*int*
have *x < x + 1*
by *simp*
then show ∃ *y*. *x < y*
by (*rule exI*)
qed

instance *int* :: *no-bot*
proof
fix *x*::*int*
have *x − 1< x*
by *simp*
then show ∃ *y*. *y < x*
by (*rule exI*)
qed

54.6 Magnitude of an Integer, as a Natural Number: *nat*

lift-definition *nat* :: *int* ⇒ *nat* **is** λ(*x*, *y*). *x − y*
by *auto*

lemma *nat-int* [*simp*]: *nat* (*int n*) = *n*
by *transfer simp*

lemma *int-nat-eq* [*simp*]: *int* (*nat z*) = (*if 0 ≤ z then z else 0*)
by *transfer clarsimp*

lemma *nat-0-le*: *0 ≤ z* ⟹ *int* (*nat z*) = *z*
by *simp*

lemma *nat-le-0* [*simp*]: *z ≤ 0* ⟹ *nat z = 0*
by *transfer clarsimp*

lemma *nat-le-eq-zle*: *0 < w ∨ 0 ≤ z* ⟹ *nat w ≤ nat z* ⟷ *w ≤ z*
by *transfer* (*clarsimp*, *arith*)

An alternative condition is *0 ≤ w*.

lemma *nat-mono-iff*: *0 < z* ⟹ *nat w < nat z* ⟷ *w < z*
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *nat-less-eq-zless*: *0 ≤ w* ⟹ *nat w < nat z* ⟷ *w < z*
by (*simp add*: *nat-le-eq-zle linorder-not-le* [*symmetric*])

lemma *zless-nat-conj* [*simp*]: *nat w* < *nat z* ⟷ *0* < *z* ∧ *w* < *z*
by *transfer* (*clarsimp*, *arith*)

lemma *nonneg-int-cases*:
assumes *0* ≤ *k*
obtains *n* **where** *k* = *int n*
proof −
from *assms* **have** *k* = *int* (*nat k*)
by *simp*
then show *thesis*
by (*rule that*)
qed

lemma *pos-int-cases*:
assumes *0* < *k*
obtains *n* **where** *k* = *int n* **and** *n* > *0*
proof −
from *assms* **have** *0* ≤ *k*
by *simp*
then obtain *n* **where** *k* = *int n*
by (*rule nonneg-int-cases*)
moreover have *n* > *0*
using ‹*k* = *int n*› *assms* **by** *simp*
ultimately show *thesis*
by (*rule that*)
qed

lemma *nonpos-int-cases*:
assumes *k* ≤ *0*
obtains *n* **where** *k* = − *int n*
proof −
from *assms* **have** − *k* ≥ *0*
by *simp*
then obtain *n* **where** − *k* = *int n*
by (*rule nonneg-int-cases*)
then have *k* = − *int n*
by *simp*
then show *thesis*
by (*rule that*)
qed

lemma *neg-int-cases*:
assumes *k* < *0*
obtains *n* **where** *k* = − *int n* **and** *n* > *0*
proof −
from *assms* **have** − *k* > *0*
by *simp*
then obtain *n* **where** − *k* = *int n* **and** − *k* > *0*
by (*blast elim*: *pos-int-cases*)

then have $k = - \ int\ n$ **and** $n > 0$
by *simp-all*
then show *thesis*
by (*rule that*)
qed

lemma *nat-eq-iff*: $nat\ w = m \longleftrightarrow (if\ 0 \leq w\ then\ w = int\ m\ else\ m = 0)$
by *transfer* (*clarsimp simp add*: *le-imp-diff-is-add*)

lemma *nat-eq-iff2*: $m = nat\ w \longleftrightarrow (if\ 0 \leq w\ then\ w = int\ m\ else\ m = 0)$
using *nat-eq-iff* [*of w m*] **by** *auto*

lemma *nat-0* [*simp*]: $nat\ 0 = 0$
by (*simp add*: *nat-eq-iff*)

lemma *nat-1* [*simp*]: $nat\ 1 = Suc\ 0$
by (*simp add*: *nat-eq-iff*)

lemma *nat-numeral* [*simp*]: $nat\ (numeral\ k) = numeral\ k$
by (*simp add*: *nat-eq-iff*)

lemma *nat-neg-numeral* [*simp*]: $nat\ (- \ numeral\ k) = 0$
by *simp*

lemma *nat-2*: $nat\ 2 = Suc\ (Suc\ 0)$
by *simp*

lemma *nat-less-iff*: $0 \leq w \Longrightarrow nat\ w < m \longleftrightarrow w < of\text{-}nat\ m$
by *transfer* (*clarsimp*, *arith*)

lemma *nat-le-iff*: $nat\ x \leq n \longleftrightarrow x \leq int\ n$
by *transfer* (*clarsimp simp add*: *le-diff-conv*)

lemma *nat-mono*: $x \leq y \Longrightarrow nat\ x \leq nat\ y$
by *transfer auto*

lemma *nat-0-iff* [*simp*]: $nat\ i = 0 \longleftrightarrow i \leq 0$
for $i :: int$
by *transfer clarsimp*

lemma *int-eq-iff*: $of\text{-}nat\ m = z \longleftrightarrow m = nat\ z \wedge 0 \leq z$
by (*auto simp add*: *nat-eq-iff2*)

lemma *zero-less-nat-eq* [*simp*]: $0 < nat\ z \longleftrightarrow 0 < z$
using *zless-nat-conj* [*of 0*] **by** *auto*

lemma *nat-add-distrib*: $0 \leq z \Longrightarrow 0 \leq z' \Longrightarrow nat\ (z + z') = nat\ z + nat\ z'$
by *transfer clarsimp*

lemma *nat-diff-distrib'*: $0 \leq x \Longrightarrow 0 \leq y \Longrightarrow nat\ (x - y) = nat\ x - nat\ y$
by *transfer clarsimp*

lemma *nat-diff-distrib*: $0 \leq z' \Longrightarrow z' \leq z \Longrightarrow nat\ (z - z') = nat\ z - nat\ z'$
by (*rule nat-diff-distrib'*) *auto*

lemma *nat-zminus-int* [*simp*]: $nat\ (-\ int\ n) = 0$
by *transfer simp*

lemma *le-nat-iff*: $k \geq 0 \Longrightarrow n \leq nat\ k \longleftrightarrow int\ n \leq k$
by *transfer auto*

lemma *zless-nat-eq-int-zless*: $m < nat\ z \longleftrightarrow int\ m < z$
by *transfer* (*clarsimp simp add*: *less-diff-conv*)

lemma (**in** *ring-1*) *of-nat-nat* [*simp*]: $0 \leq z \Longrightarrow of\text{-}nat\ (nat\ z) = of\text{-}int\ z$
by *transfer* (*clarsimp simp add*: *of-nat-diff*)

lemma *diff-nat-numeral* [*simp*]: $(numeral\ v :: nat) - numeral\ v' = nat\ (numeral\ v - numeral\ v')$
by (*simp only*: *nat-diff-distrib' zero-le-numeral nat-numeral*)

lemma *nat-abs-triangle-ineq*:
$nat\ |k + l| \leq nat\ |k| + nat\ |l|$
by (*simp add*: *nat-add-distrib* [*symmetric*] *nat-le-eq-zle abs-triangle-ineq*)

lemma *nat-of-bool* [*simp*]:
$nat\ (of\text{-}bool\ P) = of\text{-}bool\ P$
by *auto*

lemma *split-nat* [*linarith-split*]: $P\ (nat\ i) \longleftrightarrow ((\forall n.\ i = int\ n \longrightarrow P\ n) \wedge (i < 0 \longrightarrow P\ 0))$
(**is** $?P = (?L \wedge ?R)$)
for $i :: int$
proof (*cases* $i < 0$)
case *True*
then show *?thesis*
by *auto*
next
case *False*
have $?P = ?L$
proof
assume *?P*
then show *?L* **using** *False* **by** *auto*
next
assume *?L*
moreover from *False* **have** $int\ (nat\ i) = i$
by (*simp add*: *not-less*)
ultimately show *?P*

by *simp*
qed
with *False* **show** *?thesis* **by** *simp*
qed

lemma *all-nat*: $(\forall x.\ P\ x) \longleftrightarrow (\forall x{\geq}0.\ P\ (nat\ x))$
by (*auto split*: *split-nat*)

lemma *ex-nat*: $(\exists x.\ P\ x) \longleftrightarrow (\exists x.\ 0 \leq x \wedge P\ (nat\ x))$
proof
assume $\exists x.\ P\ x$
then obtain x **where** $P\ x$ **..**
then have $int\ x \geq 0 \wedge P\ (nat\ (int\ x))$ **by** *simp*
then show $\exists x{\geq}0.\ P\ (nat\ x)$ **..**
next
assume $\exists x{\geq}0.\ P\ (nat\ x)$
then show $\exists x.\ P\ x$ **by** *auto*
qed

For termination proofs:

lemma *measure-function-int*[*measure-function*]: *is-measure* $(nat \circ abs)$ **..**

54.7 Lemmas about the Function *of-nat* and Orderings

lemma *negative-zless-0*: $-\ (int\ (Suc\ n)) < (0 :: int)$
by (*simp add*: *order-less-le del*: *of-nat-Suc*)

lemma *negative-zless* [*iff*]: $-\ (int\ (Suc\ n)) < int\ m$
by (*rule negative-zless-0* [*THEN order-less-le-trans*], *simp*)

lemma *negative-zle-0*: $-\ int\ n \leq 0$
by (*simp add*: *minus-le-iff*)

lemma *negative-zle* [*iff*]: $-\ int\ n \leq int\ m$
by (*rule order-trans* [*OF negative-zle-0 of-nat-0-le-iff*])

lemma *not-zle-0-negative* [*simp*]: $\neg\ 0 \leq -\ int\ (Suc\ n)$
by (*subst le-minus-iff*) (*simp del*: *of-nat-Suc*)

lemma *int-zle-neg*: $int\ n \leq -\ int\ m \longleftrightarrow n = 0 \wedge m = 0$
by *transfer simp*

lemma *not-int-zless-negative* [*simp*]: $\neg\ int\ n < -\ int\ m$
by (*simp add*: *linorder-not-less*)

lemma *negative-eq-positive* [*simp*]: $-\ int\ n = of\text{-}nat\ m \longleftrightarrow n = 0 \wedge m = 0$
by (*force simp add*: *order-eq-iff* [*of* $-$ *of-nat n*] *int-zle-neg*)

lemma *zle-iff-zadd*: $w \leq z \longleftrightarrow (\exists n.\ z = w + int\ n)$

```
  (is ?lhs ⟷ ?rhs)
proof
  assume ?rhs
  then show ?lhs by auto
next
  assume ?lhs
  then have 0 ≤ z − w by simp
  then obtain n where z − w = int n
    using zero-le-imp-eq-int [of z − w] by blast
  then have z = w + int n by simp
  then show ?rhs ..
qed

lemma zadd-int-left: int m + (int n + z) = int (m + n) + z
  by simp

lemma negD:
  assumes x < 0 shows ∃ n. x = − (int (Suc n))
proof −
  have ⋀a b. a < b ⟹ ∃ n. Suc (a + n) = b
  proof −
    fix a b:: nat
    assume a < b
    then have Suc (a + (b − Suc a)) = b
      by arith
    then show ∃ n. Suc (a + n) = b
      by (rule exI)
  qed
  with assms show ?thesis
    by transfer auto
qed
```

54.8 Cases and induction

Now we replace the case analysis rule by a more conventional one: whether an integer is negative or not.

This version is symmetric in the two subgoals.

```
lemma int-cases2 [case-names nonneg nonpos, cases type: int]:
  (⋀n. z = int n ⟹ P) ⟹ (⋀n. z = − (int n) ⟹ P) ⟹ P
  by (cases z < 0) (auto simp add: linorder-not-less dest!: negD nat-0-le [THEN
sym])
```

This is the default, with a negative case.

```
lemma int-cases [case-names nonneg neg, cases type: int]:
  assumes pos: ⋀n. z = int n ⟹ P and neg: ⋀n. z = − (int (Suc n)) ⟹ P
  shows P
proof (cases z < 0)
  case True
```

with *neg* **show** *?thesis*
by (*blast dest*!: *negD*)
next
case *False*
with *pos* **show** *?thesis*
by (*force simp add*: *linorder-not-less dest*: *nat-0-le* [*THEN sym*])
qed

lemma *int-cases3* [*case-names zero pos neg*]:
fixes k :: *int*
assumes $k = 0 \Longrightarrow P$ **and** $\bigwedge n.\ k = \mathit{int}\ n \Longrightarrow n > 0 \Longrightarrow P$
and $\bigwedge n.\ k = -\ \mathit{int}\ n \Longrightarrow n > 0 \Longrightarrow P$
shows P
proof (*cases k 0*::*int rule*: *linorder-cases*)
case *equal*
with *assms*(*1*) **show** P **by** *simp*
next
case *greater*
then have $*$: *nat* $k > 0$ **by** *simp*
moreover from $*$ **have** $k = \mathit{int}\ (\mathit{nat}\ k)$ **by** *auto*
ultimately show P **using** *assms*(*2*) **by** *blast*
next
case *less*
then have $*$: *nat* $(-\ k) > 0$ **by** *simp*
moreover from $*$ **have** $k = -\ \mathit{int}\ (\mathit{nat}\ (-\ k))$ **by** *auto*
ultimately show P **using** *assms*(*3*) **by** *blast*
qed

lemma *int-of-nat-induct* [*case-names nonneg neg*, *induct type*: *int*]:
$(\bigwedge n.\ P\ (\mathit{int}\ n)) \Longrightarrow (\bigwedge n.\ P\ (-\ (\mathit{int}\ (\mathit{Suc}\ n)))) \Longrightarrow P\ z$
by (*cases z*) *auto*

lemma *sgn-mult-dvd-iff* [*simp*]:
$\mathit{sgn}\ r * l\ \mathit{dvd}\ k \longleftrightarrow l\ \mathit{dvd}\ k \wedge (r = 0 \longrightarrow k = 0)$ **for** $k\ l\ r$ :: *int*
by (*cases r rule*: *int-cases3*) *auto*

lemma *mult-sgn-dvd-iff* [*simp*]:
$l * \mathit{sgn}\ r\ \mathit{dvd}\ k \longleftrightarrow l\ \mathit{dvd}\ k \wedge (r = 0 \longrightarrow k = 0)$ **for** $k\ l\ r$ :: *int*
using *sgn-mult-dvd-iff* [*of r l k*] **by** (*simp add*: *ac-simps*)

lemma *dvd-sgn-mult-iff* [*simp*]:
$l\ \mathit{dvd}\ \mathit{sgn}\ r * k \longleftrightarrow l\ \mathit{dvd}\ k \vee r = 0$ **for** $k\ l\ r$ :: *int*
by (*cases r rule*: *int-cases3*) *simp-all*

lemma *dvd-mult-sgn-iff* [*simp*]:
$l\ \mathit{dvd}\ k * \mathit{sgn}\ r \longleftrightarrow l\ \mathit{dvd}\ k \vee r = 0$ **for** $k\ l\ r$ :: *int*
using *dvd-sgn-mult-iff* [*of l r k*] **by** (*simp add*: *ac-simps*)

lemma *int-sgnE*:

fixes k :: *int*
obtains n **and** l **where** $k = sgn\ l * int\ n$
proof −
have $k = sgn\ k * int\ (nat\ |k|)$
by (*simp add*: *sgn-mult-abs*)
then show *?thesis* **..**
qed

54.8.1 Binary comparisons

Preliminaries

lemma *le-imp-0-less*:
fixes z :: *int*
assumes *le*: $0 \leq z$
shows $0 < 1 + z$
proof −
have $0 \leq z$ **by** *fact*
also have $\ldots < z + 1$ **by** (*rule less-add-one*)
also have $\ldots = 1 + z$ **by** (*simp add*: *ac-simps*)
finally show $0 < 1 + z$ **.**
qed

lemma *odd-less-0-iff*: $1 + z + z < 0 \longleftrightarrow z < 0$
for z :: *int*
proof (*cases z*)
case (*nonneg n*)
then show *?thesis*
by (*simp add*: *linorder-not-less add.assoc add-increasing le-imp-0-less* [*THEN order-less-imp-le*])
next
case (*neg n*)
then show *?thesis*
by (*simp del*: *of-nat-Suc of-nat-add of-nat-1*
add: *algebra-simps of-nat-1* [**where** *'a=int*, *symmetric*] *of-nat-add* [*symmetric*])
qed

54.8.2 Comparisons, for Ordered Rings

lemma *odd-nonzero*: $1 + z + z \neq 0$
for z :: *int*
proof (*cases z*)
case (*nonneg n*)
have *le*: $0 \leq z + z$
by (*simp add*: *nonneg add-increasing*)
then show *?thesis*
using *le-imp-0-less* [*OF le*] **by** (*auto simp*: *ac-simps*)
next
case (*neg n*)
show *?thesis*

proof
assume *eq*: $1 + z + z = 0$
have $0 < 1 + (int\ n + int\ n)$
by (*simp add*: *le-imp-0-less add-increasing*)
also have $\ldots = -(1 + z + z)$
by (*simp add*: *neg add.assoc* [*symmetric*])
also have $\ldots = 0$ **by** (*simp add*: *eq*)
finally have $0<0$ **..**
then show *False* **by** *blast*
qed
qed

54.9 The Set of Integers

context *ring-1*
begin

definition *Ints* :: *'a set* (‹$\mathbb{Z}$›)
where $\mathbb{Z} = range\ of\text{-}int$

lemma *Ints-of-int* [*simp*]: *of-int z* $\in \mathbb{Z}$
by (*simp add*: *Ints-def*)

lemma *Ints-of-nat* [*simp*]: *of-nat n* $\in \mathbb{Z}$
using *Ints-of-int* [*of of-nat n*] **by** *simp*

lemma *Ints-0* [*simp*]: $0 \in \mathbb{Z}$
using *Ints-of-int* [*of 0*] **by** *simp*

lemma *Ints-1* [*simp*]: $1 \in \mathbb{Z}$
using *Ints-of-int* [*of 1*] **by** *simp*

lemma *Ints-numeral* [*simp*]: *numeral n* $\in \mathbb{Z}$
by (*subst of-nat-numeral* [*symmetric*], *rule Ints-of-nat*)

lemma *Ints-add* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a + b \in \mathbb{Z}$
by (*force simp add*: *Ints-def simp flip*: *of-int-add intro*: *range-eqI*)

lemma *Ints-minus* [*simp*]: $a \in \mathbb{Z} \Longrightarrow -a \in \mathbb{Z}$
by (*force simp add*: *Ints-def simp flip*: *of-int-minus intro*: *range-eqI*)

lemma *minus-in-Ints-iff* [*simp*]: $-x \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
using *Ints-minus*[*of x*] *Ints-minus*[*of* $-x$] **by** *auto*

lemma *Ints-diff* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a - b \in \mathbb{Z}$
by (*force simp add*: *Ints-def simp flip*: *of-int-diff intro*: *range-eqI*)

lemma *Ints-mult* [*simp*]: $a \in \mathbb{Z} \Longrightarrow b \in \mathbb{Z} \Longrightarrow a * b \in \mathbb{Z}$
by (*force simp add*: *Ints-def simp flip*: *of-int-mult intro*: *range-eqI*)

lemma *Ints-power* [*simp*]: $a \in \mathbb{Z} \Longrightarrow a \mathbin{\hat{}} n \in \mathbb{Z}$
by (*induct n*) *simp-all*

lemma *Ints-cases* [*cases set*: *Ints*]:
assumes $q \in \mathbb{Z}$
obtains (*of-int*) z **where** $q = \textit{of-int}\ z$
unfolding *Ints-def*
proof −
from ‹$q \in \mathbb{Z}$› **have** $q \in \textit{range of-int}$ **unfolding** *Ints-def* **.**
then obtain z **where** $q = \textit{of-int}\ z$ **..**
then show *thesis* **..**
qed

lemma *Ints-induct* [*case-names of-int*, *induct set*: *Ints*]:
$q \in \mathbb{Z} \Longrightarrow (\bigwedge z.\ P\ (\textit{of-int}\ z)) \Longrightarrow P\ q$
by (*rule Ints-cases*) *auto*

lemma *Nats-subset-Ints*: $\mathbb{N} \subseteq \mathbb{Z}$
unfolding *Nats-def Ints-def*
by (*rule subsetI*, *elim imageE*, *hypsubst*, *subst of-int-of-nat-eq*[*symmetric*], *rule imageI*) *simp-all*

lemma *Nats-altdef1*: $\mathbb{N} = \{\textit{of-int}\ n \mid n.\ n \geq 0\}$
proof (*intro subsetI equalityI*)
fix $x :: 'a$
assume $x \in \{\textit{of-int}\ n \mid n.\ n \geq 0\}$
then obtain n **where** $x = \textit{of-int}\ n\ n \geq 0$
by (*auto elim*!: *Ints-cases*)
then have $x = \textit{of-nat}\ (\textit{nat}\ n)$
by (*subst of-nat-nat*) *simp-all*
then show $x \in \mathbb{N}$
by *simp*
next
fix $x :: 'a$
assume $x \in \mathbb{N}$
then obtain n **where** $x = \textit{of-nat}\ n$
by (*auto elim*!: *Nats-cases*)
then have $x = \textit{of-int}\ (\textit{int}\ n)$ **by** *simp*
also have $\textit{int}\ n \geq 0$ **by** *simp*
then have $\textit{of-int}\ (\textit{int}\ n) \in \{\textit{of-int}\ n \mid n.\ n \geq 0\}$ **by** *blast*
finally show $x \in \{\textit{of-int}\ n \mid n.\ n \geq 0\}$ **.**
qed

end

lemma *Ints-sum* [*intro*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Z}) \Longrightarrow \textit{sum}\ f\ A \in \mathbb{Z}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *Ints-prod* [*intro*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Z}) \Longrightarrow prod\ f\ A \in \mathbb{Z}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma (**in** *linordered-idom*) *Ints-abs* [*simp*]:
shows $a \in \mathbb{Z} \Longrightarrow abs\ a \in \mathbb{Z}$
by (*auto simp*: *abs-if*)

lemma (**in** *linordered-idom*) *Nats-altdef2*: $\mathbb{N} = \{n \in \mathbb{Z}.\ n \geq 0\}$
proof (*intro subsetI equalityI*)
fix $x :: 'a$
assume $x \in \{n \in \mathbb{Z}.\ n \geq 0\}$
then obtain n **where** $x = of\text{-}int\ n\ n \geq 0$
by (*auto elim*!: *Ints-cases*)
then have $x = of\text{-}nat\ (nat\ n)$
by (*subst of-nat-nat*) *simp-all*
then show $x \in \mathbb{N}$
by *simp*
qed (*auto elim*!: *Nats-cases*)

lemma (**in** *idom-divide*) *of-int-divide-in-Ints*:
of-int a div of-int b $\in \mathbb{Z}$ **if** *b dvd a*
proof −
from *that* **obtain** *c* **where** $a = b * c$ **..**
then show *?thesis*
by (*cases of-int b = 0*) *simp-all*
qed

The premise involving $\mathbb{Z}$ prevents $a = 1\ /\ (2::'a)$.

lemma *Ints-double-eq-0-iff*:
fixes $a :: 'a$::*ring-char-0*
assumes *in-Ints*: $a \in \mathbb{Z}$
shows $a + a = 0 \longleftrightarrow a = 0$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof −
from *in-Ints* **have** $a \in range\ of\text{-}int$
unfolding *Ints-def* [*symmetric*] **.**
then obtain *z* **where** *a*: $a = of\text{-}int\ z$ **..**
show *?thesis*
proof
assume *?rhs*
then show *?lhs* **by** *simp*
next
assume *?lhs*
with *a* **have** $of\text{-}int\ (z + z) = (of\text{-}int\ 0 :: 'a)$ **by** *simp*
then have $z + z = 0$ **by** (*simp only*: *of-int-eq-iff*)
then have $z = 0$ **by** (*simp only*: *double-zero*)
with *a* **show** *?rhs* **by** *simp*
qed
qed

lemma *Ints-odd-nonzero*:
fixes a :: *'a*::*ring-char-0*
assumes *in-Ints*: $a \in \mathbb{Z}$
shows $1 + a + a \neq 0$
proof −
from *in-Ints* **have** $a \in$ *range of-int*
unfolding *Ints-def* [*symmetric*] **.**
then obtain z **where** *a*: $a =$ *of-int* z **..**
show *?thesis*
proof
assume $1 + a + a = 0$
with a **have** *of-int* $(1 + z + z) = (\textit{of-int } 0 :: 'a)$ **by** *simp*
then have $1 + z + z = 0$ **by** (*simp only*: *of-int-eq-iff*)
with *odd-nonzero* **show** *False* **by** *blast*
qed
qed

lemma *Nats-numeral* [*simp*]: *numeral* $w \in \mathbb{N}$
using *of-nat-in-Nats* [*of numeral* w] **by** *simp*

lemma *Ints-odd-less-0*:
fixes a :: *'a*::*linordered-idom*
assumes *in-Ints*: $a \in \mathbb{Z}$
shows $1 + a + a < 0 \longleftrightarrow a < 0$
proof −
from *in-Ints* **have** $a \in$ *range of-int*
unfolding *Ints-def* [*symmetric*] **.**
then obtain z **where** *a*: $a =$ *of-int* z **..**
with a **have** $1 + a + a < 0 \longleftrightarrow$ *of-int* $(1 + z + z) < (\textit{of-int } 0 :: 'a)$
by *simp*
also have $\ldots \longleftrightarrow z < 0$
by (*simp only*: *of-int-less-iff odd-less-0-iff*)
also have $\ldots \longleftrightarrow a < 0$
by (*simp add*: *a*)
finally show *?thesis* **.**
qed

lemma *add-in-Ints-iff-left* [*simp*]: $x \in \mathbb{Z} \Longrightarrow x + y \in \mathbb{Z} \longleftrightarrow y \in \mathbb{Z}$
by (*metis Ints-add Ints-diff add-diff-cancel-left'*)

lemma *add-in-Ints-iff-right* [*simp*]: $y \in \mathbb{Z} \Longrightarrow x + y \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
by (*subst add.commute*) *auto*

lemma *diff-in-Ints-iff-left* [*simp*]: $x \in \mathbb{Z} \Longrightarrow x - y \in \mathbb{Z} \longleftrightarrow y \in \mathbb{Z}$
by (*metis Ints-diff add-in-Ints-iff-left diff-add-cancel*)

lemma *diff-in-Ints-iff-right* [*simp*]: $y \in \mathbb{Z} \Longrightarrow x - y \in \mathbb{Z} \longleftrightarrow x \in \mathbb{Z}$
by (*metis Ints-minus diff-in-Ints-iff-left minus-diff-eq*)

lemmas [*simp*] = *minus-in-Ints-iff*

lemma *fraction-not-in-Ints*:
assumes ¬(*n dvd m*) *n* ≠ *0*
shows *of-int m / of-int n* ∉ (**ℤ** :: *'a* :: {*division-ring*,*ring-char-0*} *set*)
proof
assume *of-int m / (of-int n* :: *'a*) ∈ **ℤ**
then obtain *k* **where** *of-int m / of-int n* = (*of-int k* :: *'a*) **by** (*elim Ints-cases*)
with *assms* **have** *of-int m* = (*of-int* (*k* ∗ *n*) :: *'a*) **by** (*auto simp add*: *field-split-simps*)
hence *m* = *k* ∗ *n* **by** (*subst* (*asm*) *of-int-eq-iff*)
hence *n dvd m* **by** *simp*
with *assms*(*1*) **show** *False* **by** *contradiction*
qed

lemma *of-int-div-of-int-in-Ints-iff*:
(*of-int n / of-int m* :: *'a* :: {*division-ring*,*ring-char-0*}) ∈ **ℤ** ⟷ *m* = *0* ∨ *m dvd n*
proof
assume ∗: *m* = *0* ∨ *m dvd n*
have *of-int n / of-int m* ∈ (**ℤ** :: *'a set*) **if** *m* ≠ *0 m dvd n*
proof −
from ‹*m dvd n*› **obtain** *k* **where** *n* = *m* ∗ *k*
by (*elim dvdE*)
hence *n* = *k* ∗ *m*
by (*simp add*: *mult.commute*)
hence *of-int n / (of-int m* :: *'a*) = *of-int k*
using ‹*m* ≠ *0*› **by** (*simp add*: *field-simps*)
also have ... ∈ **ℤ**
by *auto*
finally show *?thesis* **.**
qed
with ∗ **show** *of-int n / of-int m* ∈ (**ℤ** :: *'a set*)
by (*cases m* = *0*) *auto*
next
assume ∗: (*of-int n / of-int m* :: *'a*) ∈ **ℤ**
thus *m* = *0* ∨ *m dvd n*
using *fraction-not-in-Ints*[*of m n*, **where** *?'a* = *'a*] **by** *auto*
qed

lemma *fraction-numeral-not-in-Ints* [*simp*]:
assumes ¬(*numeral b* :: *int*) *dvd numeral a*
shows *numeral a / numeral b* ∉ (**ℤ** :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
using *fraction-not-in-Ints*[*of numeral b numeral a*, **where** *?'a* = *'a*] *assms* **by** *simp*

54.10 *sum* and *prod*

context *semiring-1*

begin

lemma *of-nat-sum* [*simp*]:
of-nat (*sum f A*) = ($\sum$ *x*∈*A*. *of-nat* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

context *ring-1*
begin

lemma *of-int-sum* [*simp*]:
of-int (*sum f A*) = ($\sum$ *x*∈*A*. *of-int* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

context *comm-semiring-1*
begin

lemma *of-nat-prod* [*simp*]:
of-nat (*prod f A*) = ($\prod$ *x*∈*A*. *of-nat* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

context *comm-ring-1*
begin

lemma *of-int-prod* [*simp*]:
of-int (*prod f A*) = ($\prod$ *x*∈*A*. *of-int* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

54.11 Setting up simplification procedures

ML-file ‹*Tools/int-arith.ML*›

declaration ‹*K* (
Lin-Arith.add-discrete-type ***type-name*** ‹*Int.int*›
#> Lin-Arith.add-lessD @{*thm zless-imp-add1-zle*}
#> Lin-Arith.add-inj-thms @{*thms of-nat-le-iff* [*THEN iffD2*] *of-nat-eq-iff* [*THEN iffD2*]}
#> Lin-Arith.add-inj-const (***const-name*** ‹*of-nat*›, ***typ*** ‹*nat* ⇒ *int*›)
#> Lin-Arith.add-simps
@{*thms of-int-0 of-int-1 of-int-add of-int-mult of-int-numeral of-int-neg-numeral nat-0 nat-1 diff-nat-numeral nat-numeral*
neg-less-iff-less

True-implies-equals
distrib-left [*where a = numeral v for v*]
distrib-left [*where a = − numeral v for v*]
div-by-1 div-0
times-divide-eq-right times-divide-eq-left
minus-divide-left [*THEN sym*] *minus-divide-right* [*THEN sym*]
add-divide-distrib diff-divide-distrib
of-int-minus of-int-diff
of-int-of-nat-eq}
#> *Lin-Arith.add-simprocs* [*Int-Arith.zero-one-idom-simproc*]
)›

simproc-setup *fast-arith*
((*m*::'*a*::*linordered-idom*) < *n* |
(*m*::'*a*::*linordered-idom*) ≤ *n* |
(*m*::'*a*::*linordered-idom*) = *n*) =
‹*K Lin-Arith.simproc*›

54.12 More Inequality Reasoning

lemma *zless-add1-eq*: *w* < *z* + *1* ⟷ *w* < *z* ∨ *w* = *z*
for *w z* :: *int*
by *arith*

lemma *add1-zle-eq*: *w* + *1* ≤ *z* ⟷ *w* < *z*
for *w z* :: *int*
by *arith*

lemma *zle-diff1-eq* [*simp*]: *w* ≤ *z* − *1* ⟷ *w* < *z*
for *w z* :: *int*
by *arith*

lemma *zle-add1-eq-le* [*simp*]: *w* < *z* + *1* ⟷ *w* ≤ *z*
for *w z* :: *int*
by *arith*

lemma *int-one-le-iff-zero-less*: *1* ≤ *z* ⟷ *0* < *z*
for *z* :: *int*
by *arith*

lemma *Ints-nonzero-abs-ge1*:
fixes *x*:: '*a* :: *linordered-idom*
assumes *x* ∈ *Ints x* ≠ *0*
shows *1* ≤ *abs x*
proof (*rule Ints-cases* [*OF* ‹*x* ∈ *Ints*›])
fix *z*::*int*
assume *x* = *of-int z*
with ‹*x* ≠ *0*›
show *1* ≤ |*x*|

apply (*auto simp*: *abs-if*)
by (*metis diff-0 of-int-1 of-int-le-iff of-int-minus zle-diff1-eq*)
qed

lemma *Ints-nonzero-abs-less1*:
fixes *x*:: *'a* :: *linordered-idom*
shows ⟦*x* ∈ *Ints*; *abs x* < *1*⟧ ⟹ *x* = *0*
using *Ints-nonzero-abs-ge1* [*of x*] **by** *auto*

lemma *Ints-eq-abs-less1*:
fixes *x*:: *'a* :: *linordered-idom*
shows ⟦*x* ∈ *Ints*; *y* ∈ *Ints*⟧ ⟹ *x* = *y* ⟷ *abs* (*x*−*y*) < *1*
using *eq-iff-diff-eq-0* **by** (*fastforce intro*: *Ints-nonzero-abs-less1*)

54.13 The functions *nat* and *int*

Simplify the term *w* + − *z*.

lemma *one-less-nat-eq* [*simp*]: *Suc 0* < *nat z* ⟷ *1* < *z*
using *zless-nat-conj* [*of 1 z*] **by** *auto*

lemma *int-eq-iff-numeral* [*simp*]:
int m = *numeral v* ⟷ *m* = *numeral v*
by (*simp add*: *int-eq-iff*)

lemma *nat-abs-int-diff*:
nat |*int a* − *int b*| = (*if a* ≤ *b then b* − *a else a* − *b*)
by *auto*

lemma *nat-int-add*: *nat* (*int a* + *int b*) = *a* + *b*
by *auto*

context *ring-1*
begin

lemma *of-int-of-nat* [*nitpick-simp*]:
of-int k = (*if k* < *0 then* − *of-nat* (*nat* (− *k*)) *else of-nat* (*nat k*))
proof (*cases k* < *0*)
case *True*
then have *0* ≤ − *k* **by** *simp*
then have *of-nat* (*nat* (− *k*)) = *of-int* (− *k*) **by** (*rule of-nat-nat*)
with *True* **show** *?thesis* **by** *simp*
next
case *False*
then show *?thesis* **by** (*simp add*: *not-less*)
qed

end

lemma *transfer-rule-of-int*:

```
  includes lifting-syntax
  fixes R :: 'a::ring-1 ⇒ 'b::ring-1 ⇒ bool
  assumes [transfer-rule]: R 0 0 R 1 1
    (R ===> R ===> R) (+) (+)
    (R ===> R) uminus uminus
  shows ((=) ===> R) of-int of-int
proof −
  note assms
  note transfer-rule-of-nat [transfer-rule]
  have [transfer-rule]: ((=) ===> R) of-nat of-nat
    by transfer-prover
  show ?thesis
    by (unfold of-int-of-nat [abs-def]) transfer-prover
qed

lemma nat-mult-distrib:
  fixes z z' :: int
  assumes 0 ≤ z
  shows nat (z * z') = nat z * nat z'
proof (cases 0 ≤ z')
  case False
  with assms have z * z' ≤ 0
    by (simp add: not-le mult-le-0-iff)
  then have nat (z * z') = 0 by simp
  moreover from False have nat z' = 0 by simp
  ultimately show ?thesis by simp
next
  case True
  with assms have ge-0: z * z' ≥ 0 by (simp add: zero-le-mult-iff)
  show ?thesis
    by (rule injD [of of-nat :: nat ⇒ int, OF inj-of-nat])
      (simp only: of-nat-mult of-nat-nat [OF True]
        of-nat-nat [OF assms] of-nat-nat [OF ge-0], simp)
qed

lemma nat-mult-distrib-neg:
  assumes z ≤ (0::int) shows nat (z * z') = nat (− z) * nat (− z') (is ?L = ?R)
proof −
  have ?L = nat (− z * − z')
    using assms by auto
  also have ... = ?R
    by (rule nat-mult-distrib) (use assms in auto)
  finally show ?thesis .
qed

lemma nat-abs-mult-distrib: nat |w * z| = nat |w| * nat |z|
  by (cases z = 0 ∨ w = 0)
    (auto simp add: abs-if nat-mult-distrib [symmetric]
      nat-mult-distrib-neg [symmetric] mult-less-0-iff)
```

lemma *int-in-range-abs* [*simp*]: *int n* ∈ *range abs*
proof (*rule range-eqI*)
  show *int n* = |*int n*| **by** *simp*
qed

lemma *range-abs-Nats* [*simp*]: *range abs* = (ℕ :: *int set*)
proof −
  have |*k*| ∈ ℕ **for** *k* :: *int*
    by (*cases k*) *simp-all*
  moreover have *k* ∈ *range abs* **if** *k* ∈ ℕ **for** *k* :: *int*
    using *that* **by** *induct simp*
  ultimately show *?thesis* **by** *blast*
qed

lemma *Suc-nat-eq-nat-zadd1*: *0* ≤ *z* ⟹ *Suc* (*nat z*) = *nat* (*1* + *z*)
  for *z* :: *int*
  by (*rule sym*) (*simp add*: *nat-eq-iff*)

lemma *diff-nat-eq-if*:
  nat z − *nat z'* =
    (*if z'* < *0 then nat z*
     else
      let d = *z* − *z'*
      in if d < *0 then 0 else nat d*)
  by (*simp add*: *Let-def nat-diff-distrib* [*symmetric*])

lemma *nat-numeral-diff-1* [*simp*]: *numeral v* − (*1*::*nat*) = *nat* (*numeral v* − *1*)
  using *diff-nat-numeral* [*of v Num.One*] **by** *simp*

54.14 Induction principles for int

Well-founded segments of the integers.

definition *int-ge-less-than* :: *int* ⇒ (*int* × *int*) *set*
  where *int-ge-less-than d* = {(*z'*, *z*). *d* ≤ *z'* ∧ *z'* < *z*}

lemma *wf-int-ge-less-than*: *wf* (*int-ge-less-than d*)
proof −
  have *int-ge-less-than d* ⊆ *measure* (λ*z*. *nat* (*z* − *d*))
    by (*auto simp add*: *int-ge-less-than-def*)
  then show *?thesis*
    by (*rule wf-subset* [*OF wf-measure*])
qed

This variant looks odd, but is typical of the relations suggested by RankFinder.

definition *int-ge-less-than2* :: *int* ⇒ (*int* × *int*) *set*
  where *int-ge-less-than2 d* = {(*z'*,*z*). *d* ≤ *z* ∧ *z'* < *z*}

lemma *wf-int-ge-less-than2*: *wf* (*int-ge-less-than2 d*)
proof −
  have *int-ge-less-than2 d* ⊆ *measure* (λ*z*. *nat* (*1* + *z* − *d*))
    by (*auto simp add*: *int-ge-less-than2-def*)
  then show *?thesis*
    by (*rule wf-subset* [*OF wf-measure*])
qed

theorem *int-ge-induct* [*case-names base step*, *induct set*: *int*]:
  fixes *i* :: *int*
  assumes *ge*: *k* ≤ *i*
    and *base*: *P k*
    and *step*: ⋀*i*. *k* ≤ *i* ⟹ *P i* ⟹ *P* (*i* + *1*)
  shows *P i*
proof −
  have ⋀*i*::*int*. *n* = *nat* (*i* − *k*) ⟹ *k* ≤ *i* ⟹ *P i* **for** *n*
  proof (*induct n*)
    case *0*
    then have *i* = *k* **by** *arith*
    with *base* **show** *P i* **by** *simp*
  next
    case (*Suc n*)
    then have *n* = *nat* ((*i* − *1*) − *k*) **by** *arith*
    moreover have *k*: *k* ≤ *i* − *1* **using** *Suc.prems* **by** *arith*
    ultimately have *P* (*i* − *1*) **by** (*rule Suc.hyps*)
    from *step* [*OF k this*] **show** *?case* **by** *simp*
  qed
  with *ge* **show** *?thesis* **by** *fast*
qed

theorem *int-gr-induct* [*case-names base step*, *induct set*: *int*]:
  fixes *i k* :: *int*
  assumes *k* < *i P* (*k* + *1*) ⋀*i*. *k* < *i* ⟹ *P i* ⟹ *P* (*i* + *1*)
  shows *P i*
proof −
  have *k*+*1* ≤ *i*
    using *assms* **by** *auto*
  then show *?thesis*
    by (*induction i rule*: *int-ge-induct*) (*auto simp*: *assms*)
qed

theorem *int-le-induct* [*consumes 1*, *case-names base step*]:
  fixes *i k* :: *int*
  assumes *le*: *i* ≤ *k*
    and *base*: *P k*
    and *step*: ⋀*i*. *i* ≤ *k* ⟹ *P i* ⟹ *P* (*i* − *1*)

```
  shows P i
proof −
  have ⋀i::int. n = nat(k−i) ⟹ i ≤ k ⟹ P i for n
  proof (induct n)
    case 0
    then have i = k by arith
    with base show P i by simp
  next
    case (Suc n)
    then have n = nat (k − (i + 1)) by arith
    moreover have k: i + 1 ≤ k using Suc.prems by arith
    ultimately have P (i + 1) by (rule Suc.hyps)
    from step[OF k this] show ?case by simp
  qed
  with le show ?thesis by fast
qed

theorem int-less-induct [consumes 1, case-names base step]:
  fixes i k :: int
  assumes i < k P (k − 1) ⋀i. i < k ⟹ P i ⟹ P (i − 1)
  shows P i
proof −
  have i ≤ k−1
    using assms by auto
  then show ?thesis
    by (induction i rule: int-le-induct) (auto simp: assms)
qed

theorem int-induct [case-names base step1 step2]:
  fixes k :: int
  assumes base: P k
    and step1: ⋀i. k ≤ i ⟹ P i ⟹ P (i + 1)
    and step2: ⋀i. k ≥ i ⟹ P i ⟹ P (i − 1)
  shows P i
proof −
  have i ≤ k ∨ i ≥ k by arith
  then show ?thesis
  proof
    assume i ≥ k
    then show ?thesis
      using base by (rule int-ge-induct) (fact step1)
  next
    assume i ≤ k
    then show ?thesis
      using base by (rule int-le-induct) (fact step2)
  qed
qed
```

54.15 Intermediate value theorems

lemma *nat-ivt-aux*:
  $[\![\forall i{<}n.\ |f\ (Suc\ i) - f\ i| \leq 1; f\ 0 \leq k; k \leq f\ n]\!] \Longrightarrow \exists i \leq n.\ f\ i = k$
  for $m\ n :: nat$ **and** $k :: int$
proof (*induct n*)
  case (*Suc n*)
  show *?case*
  proof (*cases* $k = f\ (Suc\ n)$)
    case *False*
    with *Suc* **have** $k \leq f\ n$
      by *auto*
    with *Suc* **show** *?thesis*
      by (*auto simp add*: *abs-if split*: *if-split-asm intro*: *le-SucI*)
  qed (*use Suc* **in** *auto*)
qed *auto*

lemma *nat-intermed-int-val*:
  fixes $m\ n :: nat$ **and** $k :: int$
  assumes $\forall i.\ m \leq i \wedge i < n \longrightarrow |f\ (Suc\ i) - f\ i| \leq 1\ m \leq n\ f\ m \leq k\ k \leq f\ n$
  shows $\exists i.\ m \leq i \wedge i \leq n \wedge f\ i = k$
proof −
  obtain i **where** $i \leq n - m\ k = f\ (m + i)$
    using *nat-ivt-aux* [*of* $n - m\ f \circ plus\ m\ k$] *assms* **by** *auto*
  with *assms* **show** *?thesis*
    using *exI*[*of* - $m + i$] **by** *auto*
qed

lemma *nat0-intermed-int-val*:
  $\exists i{\leq}n.\ f\ i = k$
  if $\forall i{<}n.\ |f\ (i + 1) - f\ i| \leq 1\ f\ 0 \leq k\ k \leq f\ n$
  for $n :: nat$ **and** $k :: int$
  using *nat-intermed-int-val* [*of 0 n f k*] *that* **by** *auto*

54.16 Products and 1, by T. M. Rasmussen

lemma *abs-zmult-eq-1*:
  fixes $m\ n :: int$
  assumes *mn*: $|m * n| = 1$
  shows $|m| = 1$
proof −
  from *mn* **have** *0*: $m \neq 0\ n \neq 0$ **by** *auto*
  have $\neg\ 2 \leq |m|$
  proof
    assume $2 \leq |m|$
    then have $2 * |n| \leq |m| * |n|$ **by** (*simp add*: *mult-mono 0*)
    also have $\ldots = |m * n|$ **by** (*simp add*: *abs-mult*)
    also from *mn* **have** $\ldots = 1$ **by** *simp*
    finally have $2 * |n| \leq 1$ **.**
    with *0* **show** *False* **by** *arith*

qed
with *0* **show** *?thesis* **by** *auto*
qed

lemma *pos-zmult-eq-1-iff-lemma*: $m * n = 1 \Longrightarrow m = 1 \vee m = -1$
for *m n* :: *int*
using *abs-zmult-eq-1* [*of m n*] **by** *arith*

lemma *pos-zmult-eq-1-iff*:
fixes *m n* :: *int*
assumes $0 < m$
shows $m * n = 1 \longleftrightarrow m = 1 \wedge n = 1$
proof −
from *assms* **have** $m * n = 1 \Longrightarrow m = 1$
by (*auto dest*: *pos-zmult-eq-1-iff-lemma*)
then show *?thesis*
by (*auto dest*: *pos-zmult-eq-1-iff-lemma*)
qed

lemma *zmult-eq-1-iff*: $m * n = 1 \longleftrightarrow (m = 1 \wedge n = 1) \vee (m = -1 \wedge n = -1)$ (**is** *?L = ?R*)
for *m n* :: *int*
proof
assume *L*: *?L* **show** *?R*
using *pos-zmult-eq-1-iff-lemma* [*OF L*] *L* **by** *force*
qed *auto*

lemma *zmult-eq-neg1-iff*: $a * b = (-1 :: int) \longleftrightarrow a = 1 \wedge b = -1 \vee a = -1 \wedge b = 1$
using *zmult-eq-1-iff*[*of a* $-b$] **by** *auto*

lemma *infinite-UNIV-int* [*simp*]: ¬ *finite* (*UNIV*::*int set*)
proof
assume *finite* (*UNIV*::*int set*)
moreover have *inj* (λ*i*::*int*. *2* ∗ *i*)
by (*rule injI*) *simp*
ultimately have *surj* (λ*i*::*int*. *2* ∗ *i*)
by (*rule finite-UNIV-inj-surj*)
then obtain *i* :: *int* **where** *1* = *2* ∗ *i* **by** (*rule surjE*)
then show *False* **by** (*simp add*: *pos-zmult-eq-1-iff*)
qed

54.17 The divides relation

lemma *zdvd-antisym-nonneg*: $0 \leq m \Longrightarrow 0 \leq n \Longrightarrow m$ *dvd* $n \Longrightarrow n$ *dvd* $m \Longrightarrow m = n$
for *m n* :: *int*
by (*auto simp add*: *dvd-def mult.assoc zero-le-mult-iff zmult-eq-1-iff*)

lemma *zdvd-antisym-abs*:
fixes $a\ b :: int$
assumes $a\ dvd\ b$ **and** $b\ dvd\ a$
shows $|a| = |b|$
proof (*cases* $a = 0$)
case *True*
with *assms* **show** *?thesis* **by** *simp*
next
case *False*
from ‹$a\ dvd\ b$› **obtain** k **where** k: $b = a * k$
unfolding *dvd-def* **by** *blast*
from ‹$b\ dvd\ a$› **obtain** k' **where** k': $a = b * k'$
unfolding *dvd-def* **by** *blast*
from $k\ k'$ **have** $a = a * k * k'$ **by** *simp*
with *mult-cancel-left1*[**where** $c{=}a$ **and** $b{=}k{*}k'$] **have** kk': $k * k' = 1$
using ‹$a \neq 0$› **by** (*simp add*: *mult.assoc*)
then have $k = 1 \wedge k' = 1 \vee k = -1 \wedge k' = -1$
by (*simp add*: *zmult-eq-1-iff*)
with $k\ k'$ **show** *?thesis* **by** *auto*
qed

lemma *zdvd-zdiffD*: $k\ dvd\ m - n \Longrightarrow k\ dvd\ n \Longrightarrow k\ dvd\ m$
for $k\ m\ n :: int$
using *dvd-add-right-iff* [*of* $k - n\ m$] **by** *simp*

lemma *zdvd-reduce*: $k\ dvd\ n + k * m \longleftrightarrow k\ dvd\ n$
for $k\ m\ n :: int$
using *dvd-add-times-triv-right-iff* [*of* $k\ n\ m$] **by** (*simp add*: *ac-simps*)

lemma *dvd-imp-le-int*:
fixes $d\ i :: int$
assumes $i \neq 0$ **and** $d\ dvd\ i$
shows $|d| \leq |i|$
proof −
from ‹$d\ dvd\ i$› **obtain** k **where** $i = d * k$ **..**
with ‹$i \neq 0$› **have** $k \neq 0$ **by** *auto*
then have $1 \leq |k|$ **and** $0 \leq |d|$ **by** *auto*
then have $|d| * 1 \leq |d| * |k|$ **by** (*rule mult-left-mono*)
with ‹$i = d * k$› **show** *?thesis* **by** (*simp add*: *abs-mult*)
qed

lemma *zdvd-not-zless*:
fixes $m\ n :: int$
assumes $0 < m$ **and** $m < n$
shows ¬ $n\ dvd\ m$
proof
from *assms* **have** $0 < n$ **by** *auto*
assume $n\ dvd\ m$ **then obtain** k **where** k: $m = n * k$ **..**
with ‹$0 < m$› **have** $0 < n * k$ **by** *auto*

```
  with ‹0 < n› have 0 < k by (simp add: zero-less-mult-iff)
  with k ‹0 < n› ‹m < n› have n * k < n * 1 by simp
  with ‹0 < n› ‹0 < k› show False unfolding mult-less-cancel-left by auto
qed

lemma zdvd-mult-cancel:
  fixes k m n :: int
  assumes d: k * m dvd k * n
    and k ≠ 0
  shows m dvd n
proof −
  from d obtain h where h: k * n = k * m * h
    unfolding dvd-def by blast
  have n = m * h
  proof (rule ccontr)
    assume ¬ ?thesis
    with ‹k ≠ 0› have k * n ≠ k * (m * h) by simp
    with h show False
      by (simp add: mult.assoc)
  qed
  then show ?thesis by simp
qed

lemma int-dvd-int-iff [simp]:
  int m dvd int n ⟷ m dvd n
proof −
  have m dvd n if int n = int m * k for k
  proof (cases k)
    case (nonneg q)
    with that have n = m * q
      by (simp del: of-nat-mult add: of-nat-mult [symmetric])
    then show ?thesis ..
  next
    case (neg q)
    with that have int n = int m * (− int (Suc q))
      by simp
    also have ... = − (int m * int (Suc q))
      by (simp only: mult-minus-right)
    also have ... = − int (m * Suc q)
      by (simp only: of-nat-mult [symmetric])
    finally have − int (m * Suc q) = int n ..
    then show ?thesis
      by (simp only: negative-eq-positive) auto
  qed
  then show ?thesis by (auto simp add: dvd-def)
qed

lemma dvd-nat-abs-iff [simp]:
  n dvd nat |k| ⟷ int n dvd k
```

proof −
have *n dvd nat* |*k*| ⟷ *int n dvd int* (*nat* |*k*|)
by (*simp only*: *int-dvd-int-iff*)
then show *?thesis*
by *simp*
qed

lemma *nat-abs-dvd-iff* [*simp*]:
nat |*k*| *dvd n* ⟷ *k dvd int n*
proof −
have *nat* |*k*| *dvd n* ⟷ *int* (*nat* |*k*|) *dvd int n*
by (*simp only*: *int-dvd-int-iff*)
then show *?thesis*
by *simp*
qed

lemma *zdvd1-eq* [*simp*]: *x dvd 1* ⟷ |*x*| = *1* (**is** *?lhs* ⟷ *?rhs*)
for *x* :: *int*
proof
assume *?lhs*
then have *nat* |*x*| *dvd nat* |*1*|
by (*simp only*: *nat-abs-dvd-iff*) *simp*
then have *nat* |*x*| = *1*
by *simp*
then show *?rhs*
by (*cases x* < *0*) *simp-all*
next
assume *?rhs*
then have *x* = *1* ∨ *x* = − *1*
by *auto*
then show *?lhs*
by (*auto intro*: *dvdI*)
qed

lemma *zdvd-mult-cancel1*:
fixes *m* :: *int*
assumes *mp*: *m* ≠ *0*
shows *m* ∗ *n dvd m* ⟷ |*n*| = *1*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?rhs*
then show *?lhs*
by (*cases n* > *0*) (*auto simp add*: *minus-equation-iff*)
next
assume *?lhs*
then have *m* ∗ *n dvd m* ∗ *1* **by** *simp*
from *zdvd-mult-cancel*[*OF this mp*] **show** *?rhs*
by (*simp only*: *zdvd1-eq*)
qed

lemma *nat-dvd-iff*: *nat z dvd m* $\longleftrightarrow$ (*if* $0 \leq z$ *then z dvd int m else m* $= 0$)
using *nat-abs-dvd-iff* [*of z m*] **by** (*cases* $z \geq 0$) *auto*

lemma *eq-nat-nat-iff*: $0 \leq z \Longrightarrow 0 \leq z' \Longrightarrow$ *nat z* $=$ *nat z'* $\longleftrightarrow z = z'$
by (*auto elim*: *nonneg-int-cases*)

lemma *nat-power-eq*: $0 \leq z \Longrightarrow$ *nat* $(z \hat{} n) =$ *nat z* $\hat{}$ *n*
by (*induct n*) (*simp-all add*: *nat-mult-distrib*)

lemma *numeral-power-eq-nat-cancel-iff* [*simp*]:
numeral x $\hat{}$ *n* $=$ *nat y* $\longleftrightarrow$ *numeral x* $\hat{}$ *n* $= y$
using *nat-eq-iff2* **by** *auto*

lemma *nat-eq-numeral-power-cancel-iff* [*simp*]:
nat y $=$ *numeral x* $\hat{}$ *n* $\longleftrightarrow y =$ *numeral x* $\hat{}$ *n*
using *numeral-power-eq-nat-cancel-iff*[*of x n y*]
by (*metis* (*mono-tags*))

lemma *numeral-power-le-nat-cancel-iff* [*simp*]:
numeral x $\hat{}$ *n* $\leq$ *nat a* $\longleftrightarrow$ *numeral x* $\hat{}$ *n* $\leq a$
using *nat-le-eq-zle*[*of numeral x* $\hat{}$ *n a*]
by (*auto simp*: *nat-power-eq*)

lemma *nat-le-numeral-power-cancel-iff* [*simp*]:
nat a $\leq$ *numeral x* $\hat{}$ *n* $\longleftrightarrow a \leq$ *numeral x* $\hat{}$ *n*
by (*simp add*: *nat-le-iff*)

lemma *numeral-power-less-nat-cancel-iff* [*simp*]:
numeral x $\hat{}$ *n* $<$ *nat a* $\longleftrightarrow$ *numeral x* $\hat{}$ *n* $< a$
using *nat-less-eq-zless*[*of numeral x* $\hat{}$ *n a*]
by (*auto simp*: *nat-power-eq*)

lemma *nat-less-numeral-power-cancel-iff* [*simp*]:
nat a $<$ *numeral x* $\hat{}$ *n* $\longleftrightarrow a <$ *numeral x* $\hat{}$ *n*
using *nat-less-eq-zless*[*of a numeral x* $\hat{}$ *n*]
by (*cases* $a < 0$) (*auto simp*: *nat-power-eq less-le-trans*[**where** *y*=*0*])

lemma *zdvd-imp-le*: $z \leq n$ **if** *z dvd n* $0 < n$ **for** *n z* :: *int*
proof (*cases n*)
case (*nonneg n*)
show *?thesis*
by (*cases z*) (*use nonneg dvd-imp-le that* **in** *auto*)
qed (*use that* **in** *auto*)

lemma *zdvd-period*:
fixes *a d* :: *int*
assumes *a dvd d*
shows *a dvd* $(x + t) \longleftrightarrow$ *a dvd* $((x + c * d) + t)$

(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof −
from *assms* **have** *a dvd* $(x + t) \longleftrightarrow$ *a dvd* $((x + t) + c * d)$
by (*simp add*: *dvd-add-left-iff*)
then show *?thesis*
by (*simp add*: *ac-simps*)
qed

lemma *fraction-numeral-not-in-Ints'* [*simp*]:
assumes $b \neq$ *Num.One*
shows *1 / numeral b* $\notin (\mathbb{Z} :: 'a :: \{$*division-ring*, *ring-char-0*$\}$ *set*)
proof −
have ∗: ¬*numeral b dvd* (*1* :: *int*)
using *assms* **by** *simp*
have *of-int 1 / of-int* (*numeral b*) $\notin (\mathbb{Z} :: 'a$ *set*)
by (*rule fraction-not-in-Ints*) (*use* ∗ **in** *auto*)
thus *?thesis*
by *simp*
qed

54.18 Powers with integer exponents

The following allows writing powers with an integer exponent. While the type signature is very generic, most theorems will assume that the underlying type is a division ring or a field.

The notation 'powi' is inspired by the 'powr' notation for real/complex exponentiation.

definition *power-int* :: $'a :: \{$*inverse*, *power*$\} \Rightarrow$ *int* $\Rightarrow$ $'a$ (**infixr** ‹*powi*› *80*) **where**
power-int x n = (*if n* $\geq$ *0 then x* ^ *nat n else inverse x* ^ (*nat* (−*n*)))

lemma *power-int-0-right* [*simp*]: *power-int x 0* = *1*
and *power-int-1-right* [*simp*]:
power-int (*y* :: $'a :: \{$*power*, *inverse*, *monoid-mult*$\}$) *1* = *y*
and *power-int-minus1-right* [*simp*]:
power-int (*y* :: $'a :: \{$*power*, *inverse*, *monoid-mult*$\}$) (−*1*) = *inverse y*
by (*simp-all add*: *power-int-def*)

lemma *power-int-of-nat* [*simp*]: *power-int x* (*int n*) = *x* ^ *n*
by (*simp add*: *power-int-def*)

lemma *power-int-numeral* [*simp*]: *power-int x* (*numeral n*) = *x* ^ *numeral n*
by (*simp add*: *power-int-def*)

lemma *powi-numeral-reduce*: *x powi numeral n* = *x* ∗ *x powi int* (*pred-numeral n*)
by (*simp add*: *numeral-eq-Suc*)

lemma *powi-minus-numeral-reduce*: *x powi* − (*numeral n*) = *inverse x* ∗ *x powi* − *int*(*pred-numeral n*)

by (*simp add*: *numeral-eq-Suc power-int-def*)

lemma *int-cases4* [*case-names nonneg neg*]:
fixes m :: *int*
obtains n **where** $m = int\ n \mid n$ **where** $n > 0\ m = -int\ n$
proof (*cases* $m \geq 0$)
case *True*
thus *?thesis* **using** *that*(*1*)[*of nat m*] **by** *auto*
next
case *False*
thus *?thesis* **using** *that*(*2*)[*of nat* $(-m)$] **by** *auto*
qed

context
assumes *SORT-CONSTRAINT*('*a*::*division-ring*)
begin

lemma *power-int-minus*: *power-int* (*x*::'*a*) $(-n)$ = *inverse* (*power-int x n*)
by (*auto simp*: *power-int-def power-inverse*)

lemma *power-int-minus-divide*: *power-int* (*x*::'*a*) $(-n)$ = *1* / (*power-int x n*)
by (*simp add*: *divide-inverse power-int-minus*)

lemma *power-int-eq-0-iff* [*simp*]: *power-int* (*x*::'*a*) *n* = *0* $\longleftrightarrow$ *x* = *0* $\wedge$ *n* $\neq$ *0*
by (*auto simp*: *power-int-def*)

lemma *power-int-0-left-if*: *power-int* (*0* :: '*a*) *m* = (*if m* = *0 then 1 else 0*)
by (*auto simp*: *power-int-def*)

lemma *power-int-0-left* [*simp*]: *m* $\neq$ *0* $\Longrightarrow$ *power-int* (*0* :: '*a*) *m* = *0*
by (*simp add*: *power-int-0-left-if*)

lemma *power-int-1-left* [*simp*]: *power-int 1 n* = (*1* :: '*a* :: *division-ring*)
by (*auto simp*: *power-int-def*)

lemma *power-diff-conv-inverse*: *x* $\neq$ *0* $\Longrightarrow$ *m* $\leq$ *n* $\Longrightarrow$ (*x* :: '*a*) ^ (*n* − *m*) = *x* ^ *n* ∗ *inverse x* ^ *m*
by (*simp add*: *field-simps flip*: *power-add*)

lemma *power-mult-inverse-distrib*: *x* ^ *m* ∗ *inverse* (*x* :: '*a*) = *inverse x* ∗ *x* ^ *m*
proof (*cases x* = *0*)
case [*simp*]: *False*
show *?thesis*
proof (*cases m*)
case (*Suc m'*)
have *x* ^ *Suc m'* ∗ *inverse x* = *x* ^ *m'*
by (*subst power-Suc2*) (*auto simp*: *mult.assoc*)
also have ... = *inverse x* ∗ *x* ^ *Suc m'*

```
    by (subst power-Suc) (auto simp: mult.assoc [symmetric])
   finally show ?thesis using Suc by simp
  qed auto
qed auto

lemma power-mult-power-inverse-commute:
  x ^ m * inverse (x :: 'a) ^ n = inverse x ^ n * x ^ m
proof (induction n)
  case (Suc n)
  have x ^ m * inverse x ^ Suc n = (x ^ m * inverse x ^ n) * inverse x
    by (simp only: power-Suc2 mult.assoc)
  also have x ^ m * inverse x ^ n = inverse x ^ n * x ^ m
    by (rule Suc)
  also have ... * inverse x = (inverse x ^ n * inverse x) * x ^ m
    by (simp add: mult.assoc power-mult-inverse-distrib)
  also have ... = inverse x ^ (Suc n) * x ^ m
    by (simp only: power-Suc2)
  finally show ?case .
qed auto

lemma power-int-add:
  assumes x ≠ 0 ∨ m + n ≠ 0
  shows   power-int (x::'a) (m + n) = power-int x m * power-int x n
proof (cases x = 0)
  case True
  thus ?thesis using assms by (auto simp: power-int-0-left-if)
next
  case [simp]: False
  show ?thesis
  proof (cases m n rule: int-cases4 [case-product int-cases4])
    case (nonneg-nonneg a b)
    thus ?thesis
      by (auto simp: power-int-def nat-add-distrib power-add)
  next
    case (nonneg-neg a b)
    thus ?thesis
      by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
                     power-mult-power-inverse-commute)
  next
    case (neg-nonneg a b)
    thus ?thesis
      by (auto simp: power-int-def nat-diff-distrib not-le power-diff-conv-inverse
                     power-mult-power-inverse-commute)
  next
    case (neg-neg a b)
    thus ?thesis
     by (auto simp: power-int-def nat-add-distrib add.commute simp flip: power-add)
  qed
qed
```

lemma *power-int-add-1*:
assumes $x \neq 0 \vee m \neq -1$
shows *power-int* $(x::'a)$ $(m + 1) =$ *power-int* x $m * x$
using *assms* **by** (*subst power-int-add*) *auto*

lemma *power-int-add-1'*:
assumes $x \neq 0 \vee m \neq -1$
shows *power-int* $(x::'a)$ $(m + 1) = x *$ *power-int* x m
using *assms* **by** (*subst add.commute*, *subst power-int-add*) *auto*

lemma *power-int-commutes*: *power-int* $(x :: 'a)$ $n * x = x *$ *power-int* x n
by (*cases* $x = 0$) (*auto simp flip*: *power-int-add-1 power-int-add-1'*)

lemma *power-int-inverse* [*field-simps*, *field-split-simps*, *divide-simps*]:
power-int (*inverse* $(x :: 'a)$) $n =$ *inverse* (*power-int* x n)
by (*auto simp*: *power-int-def power-inverse*)

lemma *power-int-mult*: *power-int* $(x :: 'a)$ $(m * n) =$ *power-int* (*power-int* x m) n
by (*auto simp*: *power-int-def zero-le-mult-iff simp flip*: *power-mult power-inverse nat-mult-distrib*)

lemma *power-int-power*: $(a \mathbin{\hat{}} b :: 'a ::$ *division-ring*$)$ *powi* $c = a$ *powi* (*int* $b * c$)
by (*subst power-int-mult*) *simp*

lemma *power-int-power'*: $(a$ *powi* $b :: 'a ::$ *division-ring*$) \mathbin{\hat{}} c = a$ *powi* $(b *$ *int* $c)$
by (*simp add*: *power-int-mult*)

lemma *power-int-nonneg-exp*: $n \geq 0 \Longrightarrow x$ *powi* $n = x \mathbin{\hat{}}$ *nat* n
by (*simp add*: *power-int-def*)

end

context
assumes *SORT-CONSTRAINT*$('a::$*field*$)$
begin

lemma *power-int-diff*:
assumes $x \neq 0 \vee m \neq n$
shows *power-int* $(x::'a)$ $(m - n) =$ *power-int* x m / *power-int* x n
using *power-int-add*[*of* x m $-n$] *assms* **by** (*auto simp*: *field-simps power-int-minus*)

lemma *power-int-minus-mult*: $x \neq 0 \vee n \neq 0 \Longrightarrow$ *power-int* $(x :: 'a)$ $(n - 1) *$ $x =$ *power-int* x n
by (*auto simp flip*: *power-int-add-1*)

lemma *power-int-mult-distrib*: *power-int* $(x * y :: 'a)$ $m =$ *power-int* x $m *$ *power-int* y m
by (*auto simp*: *power-int-def power-mult-distrib*)

lemmas *power-int-mult-distrib-numeral1* = *power-int-mult-distrib* [**where** *x* = *numeral w* **for** *w*, *simp*]
lemmas *power-int-mult-distrib-numeral2* = *power-int-mult-distrib* [**where** *y* = *numeral w* **for** *w*, *simp*]

lemma *power-int-divide-distrib*: *power-int* (*x* / *y* :: *'a*) *m* = *power-int x m* / *power-int y m*
using *power-int-mult-distrib*[*of x inverse y m*] **unfolding** *power-int-inverse*
by (*simp add*: *field-simps*)

end

lemma *power-int-add-numeral* [*simp*]:
power-int x (*numeral m*) ∗ *power-int x* (*numeral n*) = *power-int x* (*numeral* (*m* + *n*))
for *x* :: *'a* :: *division-ring*
by (*simp add*: *power-int-add* [*symmetric*])

lemma *power-int-add-numeral2* [*simp*]:
power-int x (*numeral m*) ∗ (*power-int x* (*numeral n*) ∗ *b*) = *power-int x* (*numeral* (*m* + *n*)) ∗ *b*
for *x* :: *'a* :: *division-ring*
by (*simp add*: *mult.assoc* [*symmetric*])

lemma *power-int-mult-numeral* [*simp*]:
power-int (*power-int x* (*numeral m*)) (*numeral n*) = *power-int x* (*numeral* (*m* ∗ *n*))
for *x* :: *'a* :: *division-ring*
by (*simp only*: *numeral-mult power-int-mult*)

lemma *power-int-not-zero*: (*x* :: *'a* :: *division-ring*) ≠ *0* ∨ *n* = *0* ⟹ *power-int x n* ≠ *0*
by (*subst power-int-eq-0-iff*) *auto*

lemma *power-int-one-over* [*field-simps*, *field-split-simps*, *divide-simps*]:
power-int (*1* / *x* :: *'a* :: *division-ring*) *n* = *1* / *power-int x n*
using *power-int-inverse*[*of x*] **by** (*simp add*: *divide-inverse*)

context
assumes *SORT-CONSTRAINT*(*'a* :: *linordered-field*)
begin

lemma *power-int-numeral-neg-numeral* [*simp*]:
power-int (*numeral m*) (−*numeral n*) = (*inverse* (*numeral* (*Num.pow m n*)) :: *'a*)
by (*simp add*: *power-int-minus*)

lemma *zero-less-power-int* [*simp*]: $0 < (x :: 'a) \Longrightarrow 0 < \textit{power-int}\ x\ n$
by (*auto simp*: *power-int-def*)

lemma *zero-le-power-int* [*simp*]: $0 \leq (x :: 'a) \Longrightarrow 0 \leq \textit{power-int}\ x\ n$
by (*auto simp*: *power-int-def*)

lemma *power-int-mono*: $(x :: 'a) \leq y \Longrightarrow n \geq 0 \Longrightarrow 0 \leq x \Longrightarrow \textit{power-int}\ x\ n \leq \textit{power-int}\ y\ n$
by (*cases n rule*: *int-cases4*) (*auto intro*: *power-mono*)

lemma *one-le-power-int* [*simp*]: $1 \leq (x :: 'a) \Longrightarrow n \geq 0 \Longrightarrow 1 \leq \textit{power-int}\ x\ n$
using *power-int-mono* [*of 1 x n*] **by** *simp*

lemma *power-int-le-one*: $0 \leq (x :: 'a) \Longrightarrow n \geq 0 \Longrightarrow x \leq 1 \Longrightarrow \textit{power-int}\ x\ n \leq 1$
using *power-int-mono* [*of x 1 n*] **by** *simp*

lemma *power-int-le-imp-le-exp*:
assumes *gt1*: $1 < (x :: 'a :: \textit{linordered-field})$
assumes $\textit{power-int}\ x\ m \leq \textit{power-int}\ x\ n\ n \geq 0$
shows $m \leq n$
proof (*cases* $m < 0$)
case *True*
with ‹$n \geq 0$› **show** *?thesis* **by** *simp*
next
case *False*
with *assms* **have** $x \mathbin{\hat{}} \textit{nat}\ m \leq x \mathbin{\hat{}} \textit{nat}\ n$
by (*simp add*: *power-int-def*)
from *gt1* **and** *this* **show** *?thesis*
using *False* ‹$n \geq 0$› **by** *auto*
qed

lemma *power-int-le-imp-less-exp*:
assumes *gt1*: $1 < (x :: 'a :: \textit{linordered-field})$
assumes $\textit{power-int}\ x\ m < \textit{power-int}\ x\ n\ n \geq 0$
shows $m < n$
proof (*cases* $m < 0$)
case *True*
with ‹$n \geq 0$› **show** *?thesis* **by** *simp*
next
case *False*
with *assms* **have** $x \mathbin{\hat{}} \textit{nat}\ m < x \mathbin{\hat{}} \textit{nat}\ n$
by (*simp add*: *power-int-def*)
from *gt1* **and** *this* **show** *?thesis*
using *False* ‹$n \geq 0$› **by** *auto*
qed

lemma *power-int-strict-mono*:

$(a :: 'a :: linordered\text{-}field) < b \Longrightarrow 0 \leq a \Longrightarrow 0 < n \Longrightarrow power\text{-}int\ a\ n < power\text{-}int\ b\ n$
by (*auto simp*: *power-int-def intro*!: *power-strict-mono*)

lemma *power-int-mono-iff* [*simp*]:
fixes $a\ b :: 'a :: linordered\text{-}field$
shows $[\![a \geq 0;\ b \geq 0;\ n > 0]\!] \Longrightarrow power\text{-}int\ a\ n \leq power\text{-}int\ b\ n \longleftrightarrow a \leq b$
by (*auto simp*: *power-int-def intro*!: *power-strict-mono*)

lemma *power-int-strict-increasing*:
fixes $a :: 'a :: linordered\text{-}field$
assumes $n < N\ 1 < a$
shows *power-int a N > power-int a n*
proof −
have ∗: $a \mathbin{\hat{}} nat\ (N - n) > a \mathbin{\hat{}} 0$
using *assms* **by** (*intro power-strict-increasing*) *auto*
have *power-int a N = power-int a n ∗ power-int a (N − n)*
using *assms* **by** (*simp flip*: *power-int-add*)
also have ... *> power-int a n ∗ 1*
using *assms* ∗
by (*intro mult-strict-left-mono zero-less-power-int*) (*auto simp*: *power-int-def*)
finally show *?thesis* **by** *simp*
qed

lemma *power-int-increasing*:
fixes $a :: 'a :: linordered\text{-}field$
assumes $n \leq N\ a \geq 1$
shows *power-int a N ≥ power-int a n*
proof −
have ∗: $a \mathbin{\hat{}} nat\ (N - n) \geq a \mathbin{\hat{}} 0$
using *assms* **by** (*intro power-increasing*) *auto*
have *power-int a N = power-int a n ∗ power-int a (N − n)*
using *assms* **by** (*simp flip*: *power-int-add*)
also have ... *≥ power-int a n ∗ 1*
using *assms* ∗ **by** (*intro mult-left-mono*) (*auto simp*: *power-int-def*)
finally show *?thesis* **by** *simp*
qed

lemma *power-int-strict-decreasing*:
fixes $a :: 'a :: linordered\text{-}field$
assumes $n < N\ 0 < a\ a < 1$
shows *power-int a N < power-int a n*
proof −
have ∗: $a \mathbin{\hat{}} nat\ (N - n) < a \mathbin{\hat{}} 0$
using *assms* **by** (*intro power-strict-decreasing*) *auto*
have *power-int a N = power-int a n ∗ power-int a (N − n)*
using *assms* **by** (*simp flip*: *power-int-add*)
also have ... *< power-int a n ∗ 1*
using *assms* ∗

```
    by (intro mult-strict-left-mono zero-less-power-int) (auto simp: power-int-def)
  finally show ?thesis by simp
qed

lemma power-int-decreasing:
  fixes a :: 'a :: linordered-field
  assumes n ≤ N 0 ≤ a a ≤ 1 a ≠ 0 ∨ N ≠ 0 ∨ n = 0
  shows   power-int a N ≤ power-int a n
proof (cases a = 0)
  case False
  have *: a ^ nat (N − n) ≤ a ^ 0
    using assms by (intro power-decreasing) auto
  have power-int a N = power-int a n * power-int a (N − n)
    using assms False by (simp flip: power-int-add)
  also have ... ≤ power-int a n * 1
    using assms * by (intro mult-left-mono) (auto simp: power-int-def)
  finally show ?thesis by simp
qed (use assms in ‹auto simp: power-int-0-left-if›)

lemma one-less-power-int: 1 < (a :: 'a) ⟹ 0 < n ⟹ 1 < power-int a n
  using power-int-strict-increasing[of 0 n a] by simp

lemma power-int-abs: |power-int a n :: 'a| = power-int |a| n
  by (auto simp: power-int-def power-abs)

lemma power-int-sgn [simp]: sgn (power-int a n :: 'a) = power-int (sgn a) n
  by (auto simp: power-int-def)

lemma abs-power-int-minus [simp]: |power-int (− a) n :: 'a| = |power-int a n|
  by (simp add: power-int-abs)

lemma power-int-strict-antimono:
  assumes (a :: 'a :: linordered-field) < b 0 < a n < 0
  shows   power-int a n > power-int b n
proof −
  have inverse (power-int a (−n)) > inverse (power-int b (−n))
   using assms by (intro less-imp-inverse-less power-int-strict-mono zero-less-power-int)
auto
  thus ?thesis by (simp add: power-int-minus)
qed

lemma power-int-antimono:
  assumes (a :: 'a :: linordered-field) ≤ b 0 < a n < 0
  shows   power-int a n ≥ power-int b n
  using power-int-strict-antimono[of a b n] assms by (cases a = b) auto

end
```

54.19 Finiteness of intervals

lemma *finite-interval-int1* [*iff*]: *finite* $\{i :: int.\ a \leq i \wedge i \leq b\}$
proof (*cases* $a \leq b$)
  case *True*
  then show *?thesis*
  proof (*induct b rule*: *int-ge-induct*)
    case *base*
    have $\{i.\ a \leq i \wedge i \leq a\} = \{a\}$ **by** *auto*
    then show *?case* **by** *simp*
  next
    case (*step b*)
    then have $\{i.\ a \leq i \wedge i \leq b + 1\} = \{i.\ a \leq i \wedge i \leq b\} \cup \{b + 1\}$ **by** *auto*
    with *step* **show** *?case* **by** *simp*
  qed
next
  case *False*
  then show *?thesis*
    by (*metis* (*lifting*, *no-types*) *Collect-empty-eq finite.emptyI order-trans*)
qed

lemma *finite-interval-int2* [*iff*]: *finite* $\{i :: int.\ a \leq i \wedge i < b\}$
  by (*rule rev-finite-subset*[*OF finite-interval-int1*[*of a b*]]) *auto*

lemma *finite-interval-int3* [*iff*]: *finite* $\{i :: int.\ a < i \wedge i \leq b\}$
  by (*rule rev-finite-subset*[*OF finite-interval-int1*[*of a b*]]) *auto*

lemma *finite-interval-int4* [*iff*]: *finite* $\{i :: int.\ a < i \wedge i < b\}$
  by (*rule rev-finite-subset*[*OF finite-interval-int1*[*of a b*]]) *auto*

54.20 Configuration of the code generator

Constructors

definition $Pos :: num \Rightarrow int$
  where [*simp*, *code-abbrev*]: $Pos = numeral$

definition $Neg :: num \Rightarrow int$
  where [*simp*, *code-abbrev*]: $Neg\ n = -\ (Pos\ n)$

code-datatype *0*::*int Pos Neg*

Auxiliary operations.

definition $dup :: int \Rightarrow int$
  where [*simp*]: $dup\ k = k + k$

lemma *dup-code* [*code*]:
  $dup\ 0 = 0$
  $dup\ (Pos\ n) = Pos\ (Num.Bit0\ n)$
  $dup\ (Neg\ n) = Neg\ (Num.Bit0\ n)$

by (*simp-all add*: *numeral-Bit0*)

definition *sub* :: *num* ⇒ *num* ⇒ *int*
where [*simp*]: *sub m n* = *numeral m* − *numeral n*

lemma *sub-code* [*code*]:
sub Num.One Num.One = *0*
sub (*Num.Bit0 m*) *Num.One* = *Pos* (*Num.BitM m*)
sub (*Num.Bit1 m*) *Num.One* = *Pos* (*Num.Bit0 m*)
sub Num.One (*Num.Bit0 n*) = *Neg* (*Num.BitM n*)
sub Num.One (*Num.Bit1 n*) = *Neg* (*Num.Bit0 n*)
sub (*Num.Bit0 m*) (*Num.Bit0 n*) = *dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit1 n*) = *dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit0 n*) = *dup* (*sub m n*) + *1*
sub (*Num.Bit0 m*) (*Num.Bit1 n*) = *dup* (*sub m n*) − *1*
by (*simp-all only*: *sub-def dup-def numeral.simps Pos-def Neg-def numeral-BitM*)

lemma *sub-BitM-One-eq*:
‹(*Num.sub* (*Num.BitM n*) *num.One*) = *2* ∗ (*Num.sub n Num.One* :: *int*)›
by (*cases n*) *simp-all*

Implementations.

lemma *one-int-code* [*code*]: *1* = *Pos Num.One*
by *simp*

lemma *plus-int-code* [*code*]:
k + *0* = *k*
0 + *l* = *l*
Pos m + *Pos n* = *Pos* (*m* + *n*)
Pos m + *Neg n* = *sub m n*
Neg m + *Pos n* = *sub n m*
Neg m + *Neg n* = *Neg* (*m* + *n*)
for *k l* :: *int*
by *simp-all*

lemma *uminus-int-code* [*code*]:
uminus 0 = (*0*::*int*)
uminus (*Pos m*) = *Neg m*
uminus (*Neg m*) = *Pos m*
by *simp-all*

lemma *minus-int-code* [*code*]:
k − *0* = *k*
0 − *l* = *uminus l*
Pos m − *Pos n* = *sub m n*
Pos m − *Neg n* = *Pos* (*m* + *n*)
Neg m − *Pos n* = *Neg* (*m* + *n*)
Neg m − *Neg n* = *sub n m*
for *k l* :: *int*

by *simp-all*

lemma *times-int-code* [*code*]:
*k * 0 = 0*
*0 * l = 0*
*Pos m * Pos n = Pos (m * n)*
*Pos m * Neg n = Neg (m * n)*
*Neg m * Pos n = Neg (m * n)*
*Neg m * Neg n = Pos (m * n)*
for *k l* :: *int*
by *simp-all*

instantiation *int* :: *equal*
begin

definition *HOL.equal k l* ⟷ *k = (l*::*int)*

instance
by *standard* (*rule equal-int-def*)

end

lemma *equal-int-code* [*code*]:
HOL.equal 0 (0::*int)* ⟷ *True*
HOL.equal 0 (Pos l) ⟷ *False*
HOL.equal 0 (Neg l) ⟷ *False*
HOL.equal (Pos k) 0 ⟷ *False*
HOL.equal (Pos k) (Pos l) ⟷ *HOL.equal k l*
HOL.equal (Pos k) (Neg l) ⟷ *False*
HOL.equal (Neg k) 0 ⟷ *False*
HOL.equal (Neg k) (Pos l) ⟷ *False*
HOL.equal (Neg k) (Neg l) ⟷ *HOL.equal k l*
by (*auto simp add*: *equal*)

lemma *equal-int-refl* [*code nbe*]: *HOL.equal k k* ⟷ *True*
for *k* :: *int*
by (*fact equal-refl*)

lemma *less-eq-int-code* [*code*]:
0 ≤ (0::*int)* ⟷ *True*
0 ≤ Pos l ⟷ *True*
0 ≤ Neg l ⟷ *False*
Pos k ≤ 0 ⟷ *False*
Pos k ≤ Pos l ⟷ *k ≤ l*
Pos k ≤ Neg l ⟷ *False*
Neg k ≤ 0 ⟷ *True*
Neg k ≤ Pos l ⟷ *True*
Neg k ≤ Neg l ⟷ *l ≤ k*
by *simp-all*

lemma *less-int-code* [*code*]:
0 < (0::int) ⟷ False
0 < Pos l ⟷ True
0 < Neg l ⟷ False
Pos k < 0 ⟷ False
Pos k < Pos l ⟷ k < l
Pos k < Neg l ⟷ False
Neg k < 0 ⟷ True
Neg k < Pos l ⟷ True
Neg k < Neg l ⟷ l < k
by *simp-all*

lemma *nat-code* [*code*]:
nat (Int.Neg k) = 0
nat 0 = 0
nat (Int.Pos k) = nat-of-num k
by (*simp-all add*: *nat-of-num-numeral*)

lemma (**in** *ring-1*) *of-int-code* [*code*]:
of-int (Int.Neg k) = − numeral k
of-int 0 = 0
of-int (Int.Pos k) = numeral k
by *simp-all*

Serializer setup.

code-identifier
code-module *Int* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

quickcheck-params [*default-type = int*]

hide-const (**open**) *Pos Neg sub dup*

De-register *int* as a quotient type:

lifting-update *int.lifting*
lifting-forget *int.lifting*

54.21 Duplicates

lemmas *int-sum = of-nat-sum* [**where** *'a=int*]
lemmas *int-prod = of-nat-prod* [**where** *'a=int*]
lemmas *zle-int = of-nat-le-iff* [**where** *'a=int*]
lemmas *int-int-eq = of-nat-eq-iff* [**where** *'a=int*]
lemmas *nonneg-eq-int = nonneg-int-cases*
lemmas *double-eq-0-iff = double-zero*

lemmas *int-distrib =*
distrib-right [*of z1 z2 w*]
distrib-left [*of w z1 z2*]

left-diff-distrib [*of z1 z2 w*]
right-diff-distrib [*of w z1 z2*]
for *z1 z2 w* :: *int*

end

55 Big infimum (minimum) and supremum (maximum) over finite (non-empty) sets

theory *Lattices-Big*
imports *Groups-Big Option*
begin

55.1 Generic lattice operations over a set

55.1.1 Without neutral element

locale *semilattice-set* = *semilattice*
begin

interpretation *comp-fun-idem f*
by *standard* (*simp-all add*: *fun-eq-iff left-commute*)

definition *F* :: *'a set* ⇒ *'a*
where
eq-fold': *F A* = *the* (*Finite-Set.fold* (λ*x y*. *Some* (*case y of None* ⇒ *x* | *Some z* ⇒ *f x z*)) *None A*)

lemma *eq-fold*:
assumes *finite A*
shows *F* (*insert x A*) = *Finite-Set.fold f x A*
proof (*rule sym*)
let *?f* = λ*x y*. *Some* (*case y of None* ⇒ *x* | *Some z* ⇒ *f x z*)
interpret *comp-fun-idem ?f*
by *standard* (*simp-all add*: *fun-eq-iff commute left-commute split*: *option.split*)
from *assms* **show** *Finite-Set.fold f x A* = *F* (*insert x A*)
proof *induct*
case *empty* **then show** *?case* **by** (*simp add*: *eq-fold'*)
next
case (*insert y B*) **then show** *?case* **by** (*simp add*: *insert-commute* [*of x*] *eq-fold'*)
qed
qed

lemma *singleton* [*simp*]:
F {*x*} = *x*
by (*simp add*: *eq-fold*)

lemma *insert-not-elem*:
assumes *finite A* **and** *x* ∉ *A* **and** *A* ≠ {}

shows $F\ (\mathit{insert}\ x\ A) = x * F\ A$
proof −
from ‹$A \neq \{\}$› **obtain** b **where** $b \in A$ **by** *blast*
then obtain B **where** $*$: $A = \mathit{insert}\ b\ B\ b \notin B$ **by** (*blast dest*: *mk-disjoint-insert*)
with ‹*finite* A› **and** ‹$x \notin A$›
have *finite* (*insert* x B) **and** $b \notin \mathit{insert}\ x\ B$ **by** *auto*
then have $F\ (\mathit{insert}\ b\ (\mathit{insert}\ x\ B)) = x * F\ (\mathit{insert}\ b\ B)$
by (*simp add*: *eq-fold*)
then show *?thesis* **by** (*simp add*: $*$ *insert-commute*)
qed

lemma *in-idem*:
assumes *finite* A **and** $x \in A$
shows $x * F\ A = F\ A$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
with ‹*finite* A› **show** *?thesis* **using** ‹$x \in A$›
by (*induct* A *rule*: *finite-ne-induct*) (*auto simp add*: *ac-simps insert-not-elem*)
qed

lemma *insert* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $F\ (\mathit{insert}\ x\ A) = x * F\ A$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert-absorb in-idem insert-not-elem*)

lemma *union*:
assumes *finite* A $A \neq \{\}$ **and** *finite* B $B \neq \{\}$
shows $F\ (A \cup B) = F\ A * F\ B$
using *assms* **by** (*induct* A *rule*: *finite-ne-induct*) (*simp-all add*: *ac-simps*)

lemma *remove*:
assumes *finite* A **and** $x \in A$
shows $F\ A = (\mathit{if}\ A - \{x\} = \{\}\ \mathit{then}\ x\ \mathit{else}\ x * F\ (A - \{x\}))$
proof −
from *assms* **obtain** B **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$ **by** (*blast dest*: *mk-disjoint-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *insert-remove*:
assumes *finite* A
shows $F\ (\mathit{insert}\ x\ A) = (\mathit{if}\ A - \{x\} = \{\}\ \mathit{then}\ x\ \mathit{else}\ x * F\ (A - \{x\}))$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert-absorb remove*)

lemma *subset*:
assumes *finite* A $B \neq \{\}$ **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
proof −
from *assms* **have** $A \neq \{\}$ **and** *finite* B **by** (*auto dest*: *finite-subset*)

with *assms* **show** *?thesis* **by** (*simp add*: *union* [*symmetric*] *Un-absorb1*)
qed

lemma *closed*:
assumes *finite A* $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
using ‹*finite A*› ‹$A \neq \{\}$› **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **then show** *?case* **by** *simp*
next
case *insert* **with** *elem* **show** *?case* **by** *force*
qed

lemma *hom-commute*:
assumes *hom*: $\bigwedge x\ y.\ h\ (x * y) = h\ x * h\ y$
and *N*: *finite N* $N \neq \{\}$
shows $h\ (F\ N) = F\ (h\ `\ N)$
using *N* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** *simp*
next
case (*insert n N*)
then have $h\ (F\ (\mathit{insert}\ n\ N)) = h\ (n * F\ N)$ **by** *simp*
also have $\ldots = h\ n * h\ (F\ N)$ **by** (*rule hom*)
also have $h\ (F\ N) = F\ (h\ `\ N)$ **by** (*rule insert*)
also have $h\ n * \ldots = F\ (\mathit{insert}\ (h\ n)\ (h\ `\ N))$
using *insert* **by** *simp*
also have $\mathit{insert}\ (h\ n)\ (h\ `\ N) = h\ `\ \mathit{insert}\ n\ N$ **by** *simp*
finally show *?case* **.**
qed

lemma *infinite*: $\neg$ *finite A* $\Longrightarrow F\ A = \mathit{the\ None}$
unfolding *eq-fold′* **by** (*cases finite* (*UNIV*::′*a set*)) (*auto intro*: *finite-subset fold-infinite*)

end

locale *semilattice-order-set* = *binary?*: *semilattice-order* + *semilattice-set*
begin

lemma *bounded-iff*:
assumes *finite A* **and** $A \neq \{\}$
shows $x \leq F\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
using *assms* **by** (*induct rule*: *finite-ne-induct*) *simp-all*

lemma *boundedI*:
assumes *finite A*
assumes $A \neq \{\}$
assumes $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
shows $x \leq F\ A$
using *assms* **by** (*simp add*: *bounded-iff*)

lemma *boundedE*:
assumes *finite A* **and** $A \neq \{\}$ **and** $x \leq F\ A$
obtains $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
using *assms* **by** (*simp add*: *bounded-iff*)

lemma *coboundedI*:
assumes *finite A*
and $a \in A$
shows $F\ A \leq a$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$a \in A$› **show** *?thesis*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** (*simp add*: *refl*)
next
case (*insert x B*)
from *insert* **have** $a = x \vee a \in B$ **by** *simp*
then show *?case* **using** *insert* **by** (*auto intro*: *coboundedI2*)
qed
qed

lemma *subset-imp*:
assumes $A \subseteq B$ **and** $A \neq \{\}$ **and** *finite B*
shows $F\ B \leq F\ A$
proof (*cases* $A = B$)
case *True* **then show** *?thesis* **by** (*simp add*: *refl*)
next
case *False*
have *B*: $B = A \cup (B - A)$ **using** ‹$A \subseteq B$› **by** *blast*
then have $F\ B = F\ (A \cup (B - A))$ **by** *simp*
also have $\ldots = F\ A * F\ (B - A)$ **using** *False assms* **by** (*subst union*) (*auto intro*: *finite-subset*)
also have $\ldots \leq F\ A$ **by** *simp*
finally show *?thesis* **.**
qed

end

55.1.2 With neutral element

locale *semilattice-neutr-set* = *semilattice-neutr*
begin

interpretation *comp-fun-idem f*
by *standard* (*simp-all add*: *fun-eq-iff left-commute*)

definition $F :: 'a\ set \Rightarrow 'a$
where

eq-fold: $F\ A = \mathit{Finite\text{-}Set.fold}\ f\ \mathbf{1}\ A$

lemma *infinite* [*simp*]:
$\neg\ \mathit{finite}\ A \Longrightarrow F\ A = \mathbf{1}$
by (*simp add*: *eq-fold*)

lemma *empty* [*simp*]:
$F\ \{\} = \mathbf{1}$
by (*simp add*: *eq-fold*)

lemma *insert* [*simp*]:
assumes *finite A*
shows $F\ (\mathit{insert}\ x\ A) = x * F\ A$
using *assms* **by** (*simp add*: *eq-fold*)

lemma *in-idem*:
assumes *finite A* **and** $x \in A$
shows $x * F\ A = F\ A$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
with ‹*finite A*› **show** *?thesis* **using** ‹$x \in A$›
by (*induct A rule*: *finite-ne-induct*) (*auto simp add*: *ac-simps*)
qed

lemma *union*:
assumes *finite A* **and** *finite B*
shows $F\ (A \cup B) = F\ A * F\ B$
using *assms* **by** (*induct A*) (*simp-all add*: *ac-simps*)

lemma *remove*:
assumes *finite A* **and** $x \in A$
shows $F\ A = x * F\ (A - \{x\})$
proof −
from *assms* **obtain** *B* **where** $A = \mathit{insert}\ x\ B$ **and** $x \notin B$ **by** (*blast dest*: *mk-disjoint-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *insert-remove*:
assumes *finite A*
shows $F\ (\mathit{insert}\ x\ A) = x * F\ (A - \{x\})$
using *assms* **by** (*cases* $x \in A$) (*simp-all add*: *insert-absorb remove*)

lemma *subset*:
assumes *finite A* **and** $B \subseteq A$
shows $F\ B * F\ A = F\ A$
proof −
from *assms* **have** *finite B* **by** (*auto dest*: *finite-subset*)
with *assms* **show** *?thesis* **by** (*simp add*: *union* [*symmetric*] *Un-absorb1*)

qed

lemma *closed*:
assumes *finite A* $A \neq \{\}$ **and** *elem*: $\bigwedge x\ y.\ x * y \in \{x, y\}$
shows $F\ A \in A$
using ‹*finite A*› ‹$A \neq \{\}$› **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **then show** *?case* **by** *simp*
next
case *insert* **with** *elem* **show** *?case* **by** *force*
qed

end

locale *semilattice-order-neutr-set* = *binary?*: *semilattice-neutr-order* + *semilattice-neutr-set*
begin

lemma *bounded-iff*:
assumes *finite A*
shows $x \leq F\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
using *assms* **by** (*induct A*) *simp-all*

lemma *boundedI*:
assumes *finite A*
assumes $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
shows $x \leq F\ A$
using *assms* **by** (*simp add*: *bounded-iff*)

lemma *boundedE*:
assumes *finite A* **and** $x \leq F\ A$
obtains $\bigwedge a.\ a \in A \Longrightarrow x \leq a$
using *assms* **by** (*simp add*: *bounded-iff*)

lemma *coboundedI*:
assumes *finite A*
and $a \in A$
shows $F\ A \leq a$
proof −
from *assms* **have** $A \neq \{\}$ **by** *auto*
from ‹*finite A*› ‹$A \neq \{\}$› ‹$a \in A$› **show** *?thesis*
proof (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case* **by** (*simp add*: *refl*)
next
case (*insert x B*)
from *insert* **have** $a = x \vee a \in B$ **by** *simp*
then show *?case* **using** *insert* **by** (*auto intro*: *coboundedI2*)
qed
qed

lemma *subset-imp*:
assumes $A \subseteq B$ **and** *finite* B
shows $F\ B \leq F\ A$
proof (*cases* $A = B$)
case *True* **then show** *?thesis* **by** (*simp add*: *refl*)
next
case *False*
have *B*: $B = A \cup (B - A)$ **using** ‹$A \subseteq B$› **by** *blast*
then have $F\ B = F\ (A \cup (B - A))$ **by** *simp*
also have $\ldots = F\ A * F\ (B - A)$ **using** *False assms* **by** (*subst union*) (*auto intro*: *finite-subset*)
also have $\ldots \leq F\ A$ **by** *simp*
finally show *?thesis* **.**
qed

end

55.2 Lattice operations on finite sets

context *semilattice-inf*
begin

sublocale *Inf-fin*: *semilattice-order-set inf less-eq less*
defines
Inf-fin (‹$\bigsqcap_{fin}$ -› [*900*] *900*) = *Inf-fin.F* **..**

end

context *semilattice-sup*
begin

sublocale *Sup-fin*: *semilattice-order-set sup greater-eq greater*
defines
Sup-fin (‹$\bigsqcup_{fin}$ -› [*900*] *900*) = *Sup-fin.F* **..**

end

55.3 Infimum and Supremum over non-empty sets

context *lattice*
begin

lemma *Inf-fin-le-Sup-fin* [*simp*]:
assumes *finite* A **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A \leq \bigsqcup_{fin} A$
proof −
from ‹$A \neq \{\}$› **obtain** a **where** $a \in A$ **by** *blast*
with ‹*finite* A› **have** $\bigsqcap_{fin} A \leq a$ **by** (*rule Inf-fin.coboundedI*)
moreover from ‹*finite* A› ‹$a \in A$› **have** $a \leq \bigsqcup_{fin} A$ **by** (*rule Sup-fin.coboundedI*)
ultimately show *?thesis* **by** (*rule order-trans*)

qed

lemma *sup-Inf-absorb* [*simp*]:
finite $A \Longrightarrow a \in A \Longrightarrow \bigsqcap_{fin} A \sqcup a = a$
by (*rule sup-absorb2*) (*rule Inf-fin.coboundedI*)

lemma *inf-Sup-absorb* [*simp*]:
finite $A \Longrightarrow a \in A \Longrightarrow a \sqcap \bigsqcup_{fin} A = a$
by (*rule inf-absorb1*) (*rule Sup-fin.coboundedI*)

end

context *distrib-lattice*
begin

lemma *sup-Inf1-distrib*:
assumes *finite A*
and $A \neq \{\}$
shows *sup x* $(\bigsqcap_{fin} A) = \bigsqcap_{fin} \{sup\ x\ a | a.\ a \in A\}$
using *assms* **by** (*simp add*: *image-def Inf-fin.hom-commute* [**where** *h*=*sup x*, *OF sup-inf-distrib1*])
(*rule arg-cong* [**where** *f*=*Inf-fin*], *blast*)

lemma *sup-Inf2-distrib*:
assumes *A*: *finite A* $A \neq \{\}$ **and** *B*: *finite B* $B \neq \{\}$
shows *sup* $(\bigsqcap_{fin} A)$ $(\bigsqcap_{fin} B) = \bigsqcap_{fin} \{sup\ a\ b | a\ b.\ a \in A \wedge b \in B\}$
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **then show** *?case*
by (*simp add*: *sup-Inf1-distrib* [*OF B*])
next
case (*insert x A*)
have *finB*: *finite* $\{sup\ x\ b\ |b.\ b \in B\}$
by (*rule finite-surj* [**where** $f = sup\ x$, *OF B*(*1*)], *auto*)
have *finAB*: *finite* $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\}$
proof −
have $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} = (\bigcup a{\in}A.\ \bigcup b{\in}B.\ \{sup\ a\ b\})$
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: $\{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\} \neq \{\}$ **using** *insert B* **by** *blast*
have *sup* $(\bigsqcap_{fin}(insert\ x\ A))$ $(\bigsqcap_{fin} B) = sup\ (inf\ x\ (\bigsqcap_{fin} A))\ (\bigsqcap_{fin} B)$
using *insert* **by** *simp*
also have $\ldots = inf\ (sup\ x\ (\bigsqcap_{fin} B))\ (sup\ (\bigsqcap_{fin} A)\ (\bigsqcap_{fin} B))$ **by**(*rule sup-inf-distrib2*)
also have $\ldots = inf\ (\bigsqcap_{fin} \{sup\ x\ b | b.\ b \in B\})\ (\bigsqcap_{fin} \{sup\ a\ b | a\ b.\ a \in A \wedge b \in B\})$
using *insert* **by**(*simp add*:*sup-Inf1-distrib*[*OF B*])
also have $\ldots = \bigsqcap_{fin}(\{sup\ x\ b\ |b.\ b \in B\} \cup \{sup\ a\ b\ |a\ b.\ a \in A \wedge b \in B\})$
(**is** - = $\bigsqcap_{fin}$ *?M*)
using *B insert*

by (*simp add*: *Inf-fin.union* [*OF finB - finAB ne*])
also have *?M* = {*sup a b* |*a b*. *a* ∈ *insert x A* ∧ *b* ∈ *B*}
by *blast*
finally show *?case* .
qed

lemma *inf-Sup1-distrib*:
assumes *finite A* **and** *A* ≠ {}
shows *inf x* (⨅$_{fin}$*A*) = ⨅$_{fin}${*inf x a*|*a*. *a* ∈ *A*}
using *assms* **by** (*simp add*: *image-def Sup-fin.hom-commute* [**where** *h*=*inf x*, *OF inf-sup-distrib1*])
(*rule arg-cong* [**where** *f*=*Sup-fin*], *blast*)

lemma *inf-Sup2-distrib*:
assumes *A*: *finite A A* ≠ {} **and** *B*: *finite B B* ≠ {}
shows *inf* (⨅$_{fin}$*A*) (⨅$_{fin}$*B*) = ⨅$_{fin}${*inf a b*|*a b*. *a* ∈ *A* ∧ *b* ∈ *B*}
using *A* **proof** (*induct rule*: *finite-ne-induct*)
case *singleton* **thus** *?case*
by(*simp add*: *inf-Sup1-distrib* [*OF B*])
next
case (*insert x A*)
have *finB*: *finite* {*inf x b* |*b*. *b* ∈ *B*}
by(*rule finite-surj*[**where** *f* = %*b*. *inf x b*, *OF B*(*1*)], *auto*)
have *finAB*: *finite* {*inf a b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*}
proof −
have {*inf a b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*} = (⋃ *a*∈*A*. ⋃ *b*∈*B*. {*inf a b*})
by *blast*
thus *?thesis* **by**(*simp add*: *insert*(*1*) *B*(*1*))
qed
have *ne*: {*inf a b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*} ≠ {} **using** *insert B* **by** *blast*
have *inf* (⨅$_{fin}$(*insert x A*)) (⨅$_{fin}$*B*) = *inf* (*sup x* (⨅$_{fin}$*A*)) (⨅$_{fin}$*B*)
using *insert* **by** *simp*
also have ... = *sup* (*inf x* (⨅$_{fin}$*B*)) (*inf* (⨅$_{fin}$*A*) (⨅$_{fin}$*B*)) **by**(*rule inf-sup-distrib2*)
also have ... = *sup* (⨅$_{fin}${*inf x b*|*b*. *b* ∈ *B*}) (⨅$_{fin}${*inf a b*|*a b*. *a* ∈ *A* ∧ *b* ∈ *B*})
using *insert* **by**(*simp add*:*inf-Sup1-distrib*[*OF B*])
also have ... = ⨅$_{fin}$({*inf x b* |*b*. *b* ∈ *B*} ∪ {*inf a b* |*a b*. *a* ∈ *A* ∧ *b* ∈ *B*})
(**is** - = ⨅$_{fin}$ *?M*)
using *B insert*
by (*simp add*: *Sup-fin.union* [*OF finB - finAB ne*])
also have *?M* = {*inf a b* |*a b*. *a* ∈ *insert x A* ∧ *b* ∈ *B*}
by *blast*
finally show *?case* .
qed

end

context *complete-lattice*
begin

lemma *Inf-fin-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcap_{fin} A = \bigsqcap A$
proof −
from *assms* **obtain** *b B* **where** $A = insert\ b\ B$ **and** *finite B* **by** *auto*
then show *?thesis*
by (*simp add*: *Inf-fin.eq-fold inf-Inf-fold-inf inf.commute* [*of b*])
qed

lemma *Sup-fin-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows $\bigsqcup_{fin} A = \bigsqcup A$
proof −
from *assms* **obtain** *b B* **where** $A = insert\ b\ B$ **and** *finite B* **by** *auto*
then show *?thesis*
by (*simp add*: *Sup-fin.eq-fold sup-Sup-fold-sup sup.commute* [*of b*])
qed

end

55.4 Minimum and Maximum over non-empty sets

context *linorder*
begin

sublocale *Min*: *semilattice-order-set min less-eq less*
+ *Max*: *semilattice-order-set max greater-eq greater*
defines
Min = *Min.F* **and** *Max* = *Max.F* **..**

end

syntax
-MIN1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder MIN››MIN -./ -*)› [*0*, *10*] *10*)
-MIN :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder MIN››MIN -∈-./ -*)› [*0*, *0*, *10*] *10*)
-MAX1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder MAX››MAX -./ -*)› [*0*, *10*] *10*)
-MAX :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder MAX››MAX -∈-./ -*)› [*0*, *0*, *10*] *10*)

syntax-consts
-MIN1 -MIN ⇌ *Min* **and**
-MAX1 -MAX ⇌ *Max*

translations
MIN x y. f ⇌ *MIN x. MIN y. f*

MIN x. f $\rightleftharpoons$ *CONST Min* (*CONST range* (λ*x. f*))
MIN x∈A. f $\rightleftharpoons$ *CONST Min* ((λ*x. f*) ‘ *A*)
MAX x y. f $\rightleftharpoons$ *MAX x. MAX y. f*
MAX x. f $\rightleftharpoons$ *CONST Max* (*CONST range* (λ*x. f*))
MAX x∈A. f $\rightleftharpoons$ *CONST Max* ((λ*x. f*) ‘ *A*)

An aside: *Min*/*Max* on linear orders as special case of *Inf-fin*/*Sup-fin*

lemma *Inf-fin-Min*:
Inf-fin = (*Min* :: *'a*::{*semilattice-inf*, *linorder*} *set* ⇒ *'a*)
by (*simp add*: *Inf-fin-def Min-def inf-min*)

lemma *Sup-fin-Max*:
Sup-fin = (*Max* :: *'a*::{*semilattice-sup*, *linorder*} *set* ⇒ *'a*)
by (*simp add*: *Sup-fin-def Max-def sup-max*)

context *linorder*
begin

lemma *dual-min*:
ord.min greater-eq = *max*
by (*auto simp add*: *ord.min-def max-def fun-eq-iff*)

lemma *dual-max*:
ord.max greater-eq = *min*
by (*auto simp add*: *ord.max-def min-def fun-eq-iff*)

lemma *dual-Min*:
linorder.Min greater-eq = *Max*
proof −
interpret *dual*: *linorder greater-eq greater* **by** (*fact dual-linorder*)
show *?thesis* **by** (*simp add*: *dual.Min-def dual-min Max-def*)
qed

lemma *dual-Max*:
linorder.Max greater-eq = *Min*
proof −
interpret *dual*: *linorder greater-eq greater* **by** (*fact dual-linorder*)
show *?thesis* **by** (*simp add*: *dual.Max-def dual-max Min-def*)
qed

lemmas *Min-singleton* = *Min.singleton*
lemmas *Max-singleton* = *Max.singleton*
lemmas *Min-insert* = *Min.insert*
lemmas *Max-insert* = *Max.insert*
lemmas *Min-Un* = *Min.union*
lemmas *Max-Un* = *Max.union*
lemmas *hom-Min-commute* = *Min.hom-commute*
lemmas *hom-Max-commute* = *Max.hom-commute*

lemma *Min-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Min\ A \in A$
using *assms* **by** (*auto simp add*: *min-def Min.closed*)

lemma *Max-in* [*simp*]:
assumes *finite A* **and** $A \neq \{\}$
shows $Max\ A \in A$
using *assms* **by** (*auto simp add*: *max-def Max.closed*)

lemma *Min-insert2*:
assumes *finite A* **and** *min*: $\bigwedge b.\ b \in A \Longrightarrow a \leq b$
shows $Min\ (insert\ a\ A) = a$
proof (*cases* $A = \{\}$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with ‹*finite A*› **have** $Min\ (insert\ a\ A) = min\ a\ (Min\ A)$
by *simp*
moreover from ‹*finite A*› ‹$A \neq \{\}$› *min* **have** $a \leq Min\ A$ **by** *simp*
ultimately show *?thesis* **by** (*simp add*: *min.absorb1*)
qed

lemma *Max-insert2*:
assumes *finite A* **and** *max*: $\bigwedge b.\ b \in A \Longrightarrow b \leq a$
shows $Max\ (insert\ a\ A) = a$
proof (*cases* $A = \{\}$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with ‹*finite A*› **have** $Max\ (insert\ a\ A) = max\ a\ (Max\ A)$
by *simp*
moreover from ‹*finite A*› ‹$A \neq \{\}$› *max* **have** $Max\ A \leq a$ **by** *simp*
ultimately show *?thesis* **by** (*simp add*: *max.absorb1*)
qed

lemma *Max-const*[*simp*]: $[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Max\ ((\lambda\text{-}.\ c)\ `\ A) = c$
using *Max-in image-is-empty* **by** *blast*

lemma *Min-const*[*simp*]: $[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Min\ ((\lambda\text{-}.\ c)\ `\ A) = c$
using *Min-in image-is-empty* **by** *blast*

lemma *Min-le* [*simp*]:
assumes *finite A* **and** $x \in A$
shows $Min\ A \leq x$
using *assms* **by** (*fact Min.coboundedI*)

lemma *Max-ge* [*simp*]:
assumes *finite A* **and** $x \in A$
shows $x \leq Max\ A$
using *assms* **by** (*fact Max.coboundedI*)

lemma *Min-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \geq x$
and $x \in A$
shows $Min\ A = x$
proof (*rule order.antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Min\ A \geq x$ **by** *simp*
next
from *assms* **show** $x \geq Min\ A$ **by** *simp*
qed

lemma *Max-eqI*:
assumes *finite A*
assumes $\bigwedge y.\ y \in A \Longrightarrow y \leq x$
and $x \in A$
shows $Max\ A = x$
proof (*rule order.antisym*)
from ‹$x \in A$› **have** $A \neq \{\}$ **by** *auto*
with *assms* **show** $Max\ A \leq x$ **by** *simp*
next
from *assms* **show** $x \leq Max\ A$ **by** *simp*
qed

lemma *eq-Min-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow m = Min\ A \longleftrightarrow m \in A \wedge (\forall\, a \in A.\ m \leq a)$
by (*meson Min-in Min-le order.antisym*)

lemma *Min-eq-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Min\ A = m \longleftrightarrow m \in A \wedge (\forall\, a \in A.\ m \leq a)$
by (*meson Min-in Min-le order.antisym*)

lemma *eq-Max-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow m = Max\ A \longleftrightarrow m \in A \wedge (\forall\, a \in A.\ a \leq m)$
by (*meson Max-in Max-ge order.antisym*)

lemma *Max-eq-iff*:
$[\![\ finite\ A;\ A \neq \{\}\ ]\!] \Longrightarrow Max\ A = m \longleftrightarrow m \in A \wedge (\forall\, a \in A.\ a \leq m)$
by (*meson Max-in Max-ge order.antisym*)

context
fixes $A :: 'a\ set$
assumes *fin-nonempty*: *finite A* $A \neq \{\}$
begin

lemma *Min-ge-iff* [*simp*]:
$x \leq Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x \leq a)$
using *fin-nonempty* **by** (*fact Min.bounded-iff*)

lemma *Max-le-iff* [*simp*]:
$Max\ A \leq x \longleftrightarrow (\forall\, a{\in}A.\ a \leq x)$
using *fin-nonempty* **by** (*fact Max.bounded-iff*)

lemma *Min-gr-iff* [*simp*]:
$x < Min\ A \longleftrightarrow (\forall\, a{\in}A.\ x < a)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) *simp-all*

lemma *Max-less-iff* [*simp*]:
$Max\ A < x \longleftrightarrow (\forall\, a{\in}A.\ a < x)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) *simp-all*

lemma *Min-le-iff*:
$Min\ A \leq x \longleftrightarrow (\exists\, a{\in}A.\ a \leq x)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) (*simp-all add*: *min-le-iff-disj*)

lemma *Max-ge-iff*:
$x \leq Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x \leq a)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) (*simp-all add*: *le-max-iff-disj*)

lemma *Min-less-iff*:
$Min\ A < x \longleftrightarrow (\exists\, a{\in}A.\ a < x)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) (*simp-all add*: *min-less-iff-disj*)

lemma *Max-gr-iff*:
$x < Max\ A \longleftrightarrow (\exists\, a{\in}A.\ x < a)$
using *fin-nonempty* **by** (*induct rule*: *finite-ne-induct*) (*simp-all add*: *less-max-iff-disj*)

end

Handy results about *Max* and *Min* by Chelsea Edmonds

lemma *obtains-Max*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $Max\ A = x$
using *assms Max-in* **by** *blast*

lemma *obtains-MAX*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $Max\ (f\ `\ A) = f\ x$
using *obtains-Max*
by (*metis* (*mono-tags*, *opaque-lifting*) *assms*(*1*) *assms*(*2*) *empty-is-image finite-imageI image-iff*)

lemma *obtains-Min*:

assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $Min\ A = x$
using *assms Min-in* **by** *blast*

lemma *obtains-MIN*:
assumes *finite A* **and** $A \neq \{\}$
obtains x **where** $x \in A$ **and** $Min\ (f\ `\ A) = f\ x$
using *obtains-Min assms empty-is-image finite-imageI image-iff*
by (*metis* (*mono-tags*, *opaque-lifting*))

lemma *Max-eq-if*:
assumes *finite A* *finite B* $\forall\, a{\in}A.\ \exists\, b{\in}B.\ a \leq b$ $\forall\, b{\in}B.\ \exists\, a{\in}A.\ b \leq a$
shows $Max\ A = Max\ B$
proof *cases*
assume $A = \{\}$ **thus** *?thesis* **using** *assms* **by** *simp*
next
assume $A \neq \{\}$ **thus** *?thesis* **using** *assms*
by(*blast intro*: *order.antisym Max-in Max-ge-iff*[*THEN iffD2*])
qed

lemma *Min-antimono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Min\ N \leq Min\ M$
using *assms* **by** (*fact Min.subset-imp*)

lemma *Max-mono*:
assumes $M \subseteq N$ **and** $M \neq \{\}$ **and** *finite N*
shows $Max\ M \leq Max\ N$
using *assms* **by** (*fact Max.subset-imp*)

lemma *mono-Min-commute*:
assumes *mono f*
assumes *finite A* **and** $A \neq \{\}$
shows $f\ (Min\ A) = Min\ (f\ `\ A)$
proof (*rule linorder-class.Min-eqI* [*symmetric*])
from ‹*finite A*› **show** *finite* $(f\ `\ A)$ **by** *simp*
from *assms* **show** $f\ (Min\ A) \in f\ `\ A$ **by** *simp*
fix x
assume $x \in f\ `\ A$
then obtain y **where** $y \in A$ **and** $x = f\ y$ **..**
with *assms* **have** $Min\ A \leq y$ **by** *auto*
with ‹*mono f*› **have** $f\ (Min\ A) \leq f\ y$ **by** (*rule monoE*)
with ‹$x = f\ y$› **show** $f\ (Min\ A) \leq x$ **by** *simp*
qed

lemma *mono-Max-commute*:
assumes *mono f*
assumes *finite A* **and** $A \neq \{\}$
shows $f\ (Max\ A) = Max\ (f\ `\ A)$

proof (*rule linorder-class.Max-eqI* [*symmetric*])
from ‹*finite A*› **show** *finite* (*f* ‘ *A*) **by** *simp*
from *assms* **show** *f* (*Max A*) ∈ *f* ‘ *A* **by** *simp*
fix *x*
assume *x* ∈ *f* ‘ *A*
then obtain *y* **where** *y* ∈ *A* **and** *x* = *f y* **..**
with *assms* **have** *y* ≤ *Max A* **by** *auto*
with ‹*mono f*› **have** *f y* ≤ *f* (*Max A*) **by** (*rule monoE*)
with ‹*x* = *f y*› **show** *x* ≤ *f* (*Max A*) **by** *simp*
qed

lemma *finite-linorder-max-induct* [*consumes 1*, *case-names empty insert*]:
assumes *fin*: *finite A*
and *empty*: *P* {}
and *insert*: ⋀*b A*. *finite A* ⟹ ∀ *a*∈*A*. *a* < *b* ⟹ *P A* ⟹ *P* (*insert b A*)
shows *P A*
using *fin empty insert*
proof (*induct rule*: *finite-psubset-induct*)
case (*psubset A*)
have *IH*: ⋀*B*. ⟦*B* < *A*; *P* {}; (⋀*A b*. ⟦*finite A*; ∀ *a*∈*A*. *a*<*b*; *P A*⟧ ⟹ *P* (*insert b A*))⟧ ⟹ *P B* **by** *fact*
have *fin*: *finite A* **by** *fact*
have *empty*: *P* {} **by** *fact*
have *step*: ⋀*b A*. ⟦*finite A*; ∀ *a*∈*A*. *a* < *b*; *P A*⟧ ⟹ *P* (*insert b A*) **by** *fact*
show *P A*
proof (*cases A* = {})
assume *A* = {}
then show *P A* **using** ‹*P* {}› **by** *simp*
next
let *?B* = *A* − {*Max A*}
let *?A* = *insert* (*Max A*) *?B*
have *finite ?B* **using** ‹*finite A*› **by** *simp*
assume *A* ≠ {}
with ‹*finite A*› **have** *Max A* ∈ *A* **by** *auto*
then have *A*: *?A* = *A* **using** *insert-Diff-single insert-absorb* **by** *auto*
then have *P ?B* **using** ‹*P* {}› *step IH* [*of ?B*] **by** *blast*
moreover
have ∀ *a*∈*?B*. *a* < *Max A* **using** *Max-ge* [*OF* ‹*finite A*›] **by** *fastforce*
ultimately show *P A* **using** *A insert-Diff-single step* [*OF* ‹*finite ?B*›] **by** *fastforce*
qed
qed

lemma *finite-linorder-min-induct* [*consumes 1*, *case-names empty insert*]:
⟦*finite A*; *P* {}; ⋀*b A*. ⟦*finite A*; ∀ *a*∈*A*. *b* < *a*; *P A*⟧ ⟹ *P* (*insert b A*)⟧ ⟹ *P A*
by (*rule linorder.finite-linorder-max-induct* [*OF dual-linorder*])

lemma *finite-ranking-induct*[*consumes 1*, *case-names empty insert*]:

fixes $f :: 'b \Rightarrow 'a$
assumes *finite S*
assumes $P\ \{\}$
assumes $\bigwedge x\ S.$ *finite* $S \Longrightarrow (\bigwedge y.\ y \in S \Longrightarrow f\ y \leq f\ x) \Longrightarrow P\ S \Longrightarrow P$ (*insert* x S)
shows $P\ S$
using ‹*finite S*›
proof (*induction rule*: *finite-psubset-induct*)
case (*psubset A*)
{
assume $A \neq \{\}$
hence $f\ `\ A \neq \{\}$ **and** *finite* $(f\ `\ A)$
using *psubset finite-image-iff* **by** *simp+*
then obtain a **where** $f\ a = Max\ (f\ `\ A)$ **and** $a \in A$
by (*metis Max-in*[*of* $f\ `\ A$] *imageE*)
then have $P\ (A - \{a\})$
using *psubset*(*2*) [*of* ‹$A - \{a\}$›] **by** *auto*
moreover
have $\bigwedge y.\ y \in A \Longrightarrow f\ y \leq f\ a$
using ‹$f\ a = Max\ (f\ `\ A)$› ‹*finite* $(f\ `\ A)$› **by** *simp*
ultimately
have *?case*
by (*metis* ‹$a \in A$› *DiffD1 insert-Diff assms*(*3*) *finite-Diff psubset.hyps*)
}
thus *?case*
using *assms*(*2*) **by** *blast*
qed

lemma *Least-Min*:
assumes *finite* $\{a.\ P\ a\}$ **and** $\exists\, a.\ P\ a$
shows $(LEAST\ a.\ P\ a) = Min\ \{a.\ P\ a\}$
proof −
{ fix $A :: 'a\ set$
assume *A*: *finite A* $A \neq \{\}$
have $(LEAST\ a.\ a \in A) = Min\ A$
using *A* **proof** (*induct A rule*: *finite-ne-induct*)
case *singleton* **show** *?case* **by** (*rule Least-equality*) *simp-all*
next
case (*insert a A*)
have $(LEAST\ b.\ b = a \vee b \in A) = min\ a\ (LEAST\ a.\ a \in A)$
by (*auto intro*!: *Least-equality simp add*: *min-def not-le Min-le-iff insert.hyps dest*!: *less-imp-le*)
with *insert* **show** *?case* **by** *simp*
qed
} from *this* [*of* $\{a.\ P\ a\}$] *assms* **show** *?thesis* **by** *simp*
qed

lemma *Greatest-Max*:
assumes *finite* $\{a.\ P\ a\}$ **and** $\exists\, a.\ P\ a$

```
  shows (GREATEST a. P a) = Max {a. P a}
proof −
  { fix A :: 'a set
    assume A: finite A A ≠ {}
    have (GREATEST a. a ∈ A) = Max A
    using A proof (induct A rule: finite-ne-induct)
      case singleton show ?case by (rule Greatest-equality) simp-all
    next
      case (insert a A)
      have (GREATEST b. b = a ∨ b ∈ A) = max a (GREATEST a. a ∈ A)
        by (auto intro!: Greatest-equality simp add: max-def not-le insert.hyps)
      with insert show ?case by simp
    qed
  } from this [of {a. P a}] assms show ?thesis by simp
qed

lemma infinite-growing:
  assumes X ≠ {}
  assumes ∗: ⋀x. x ∈ X ⟹ ∃ y∈X. y > x
  shows ¬ finite X
proof
  assume finite X
  with ‹X ≠ {}› have Max X ∈ X ∀ x∈X. x ≤ Max X
    by auto
  with ∗[of Max X] show False
    by auto
qed

end

lemma sum-le-card-Max: finite A ⟹ sum f A ≤ card A ∗ Max (f ‘ A)
using sum-bounded-above[of A f Max (f ‘ A)] by simp

lemma card-Min-le-sum: finite A ⟹ card A ∗ Min (f ‘ A) ≤ sum f A
using sum-bounded-below[of A Min (f ‘ A) f] by simp

context linordered-ab-semigroup-add
begin

lemma Min-add-commute:
  fixes k
  assumes finite S and S ≠ {}
  shows Min ((λx. f x + k) ‘ S) = Min(f ‘ S) + k
proof −
  have m: ⋀x y. min x y + k = min (x+k) (y+k)
    by (simp add: min-def order.antisym add-right-mono)
  have (λx. f x + k) ‘ S = (λy. y + k) ‘ (f ‘ S) by auto
  also have Min ... = Min (f ‘ S) + k
    using assms hom-Min-commute [of λy. y+k f ‘ S, OF m, symmetric] by simp
```

finally show *?thesis* **by** *simp*
qed

lemma *Max-add-commute*:
fixes *k*
assumes *finite S* **and** $S \neq \{\}$
shows *Max* $((\lambda x.\ f\ x + k)\ `\ S) = Max(f\ `\ S) + k$
proof −
have *m*: $\bigwedge x\ y.\ max\ x\ y + k = max\ (x+k)\ (y+k)$
by (*simp add*: *max-def order.antisym add-right-mono*)
have $(\lambda x.\ f\ x + k)\ `\ S = (\lambda y.\ y + k)\ `\ (f\ `\ S)$ **by** *auto*
also have *Max* ... = *Max* $(f\ `\ S) + k$
using *assms hom-Max-commute* [*of* $\lambda y.\ y+k\ f\ `\ S$, *OF m*, *symmetric*] **by** *simp*
finally show *?thesis* **by** *simp*
qed

end

context *linordered-ab-group-add*
begin

lemma *minus-Max-eq-Min* [*simp*]:
finite $S \Longrightarrow S \neq \{\} \Longrightarrow -\ Max\ S = Min\ (uminus\ `\ S)$
by (*induct S rule*: *finite-ne-induct*) (*simp-all add*: *minus-max-eq-min*)

lemma *minus-Min-eq-Max* [*simp*]:
finite $S \Longrightarrow S \neq \{\} \Longrightarrow -\ Min\ S = Max\ (uminus\ `\ S)$
by (*induct S rule*: *finite-ne-induct*) (*simp-all add*: *minus-min-eq-max*)

end

context *complete-linorder*
begin

lemma *Min-Inf*:
assumes *finite A* **and** $A \neq \{\}$
shows *Min A* = *Inf A*
proof −
from *assms* **obtain** *b B* **where** *A* = *insert b B* **and** *finite B* **by** *auto*
then show *?thesis*
by (*simp add*: *Min.eq-fold complete-linorder-inf-min* [*symmetric*] *inf-Inf-fold-inf inf.commute* [*of b*])
qed

lemma *Max-Sup*:
assumes *finite A* **and** $A \neq \{\}$
shows *Max A* = *Sup A*
proof −
from *assms* **obtain** *b B* **where** *A* = *insert b B* **and** *finite B* **by** *auto*

then show *?thesis*
by (*simp add*: *Max.eq-fold complete-linorder-sup-max* [*symmetric*] *sup-Sup-fold-sup sup.commute* [*of b*])
qed

end

lemma *disjnt-ge-max*:
‹*disjnt X Y*› **if** ‹*finite Y*› ‹$\bigwedge x.\ x \in X \Longrightarrow x > Max\ Y$›
using *that* **by** (*auto simp add*: *disjnt-def*) (*use Max-less-iff* **in** *blast*)

55.5 An aside: code generation for *LEAST* and *GREATEST*

context
begin

qualified definition *Least* :: ‹'*a*::*linorder set* ⇒ '*a*› — only for code generation
where *Least-eq* [*code-abbrev*, *simp*]: ‹*Least S* = (*LEAST x*. *x* ∈ *S*)›

qualified lemma *Least-filter-eq* [*code-abbrev*]:
‹*Least* (*Set.filter P S*) = (*LEAST x*. *x* ∈ *S* ∧ *P x*)›
by *simp*

qualified definition *Least-abort* :: ‹'*a set* ⇒ '*a*::*linorder*›
where ‹*Least-abort* = *Least*›

qualified lemma *Least-code* [*code abort*: *Lattices-Big.Least-abort*, *code*]:
‹*Least A* = (*if finite A* ⟶ *Set.is-empty A then Least-abort A else Min A*)›
using *Least-Min* [*of* ‹λ*x*. *x* ∈ *A*›] **by** (*auto simp add*: *Least-abort-def*)

qualified definition *Greatest* :: ‹'*a*::*linorder set* ⇒ '*a*› — only for code generation
where *Greatest-eq* [*code-abbrev*, *simp*]: ‹*Greatest S* = (*GREATEST x*. *x* ∈ *S*)›

qualified lemma *Greatest-filter-eq* [*code-abbrev*]:
‹*Greatest* (*Set.filter P S*) = (*GREATEST x*. *x* ∈ *S* ∧ *P x*)›
by *simp*

qualified definition *Greatest-abort* :: ‹'*a set* ⇒ '*a*::*linorder*›
where ‹*Greatest-abort* = *Greatest*›

qualified lemma *Greatest-code* [*code abort*: *Lattices-Big.Greatest-abort*, *code*]:
‹*Greatest A* = (*if finite A* ⟶ *Set.is-empty A then Greatest-abort A else Max A*)›
using *Greatest-Max* [*of* ‹λ*x*. *x* ∈ *A*›] **by** (*auto simp add*: *Greatest-abort-def*)

end

55.6 Arg Min

context *ord*
begin

definition *is-arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'b \Rightarrow bool$ **where**
$is\text{-}arg\text{-}min\ f\ P\ x = (P\ x \wedge \neg(\exists\, y.\ P\ y \wedge f\ y < f\ x))$

definition *arg-min* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow bool) \Rightarrow 'b$ **where**
$arg\text{-}min\ f\ P = (SOME\ x.\ is\text{-}arg\text{-}min\ f\ P\ x)$

definition *arg-min-on* :: $('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'b$ **where**
$arg\text{-}min\text{-}on\ f\ S = arg\text{-}min\ f\ (\lambda x.\ x \in S)$

end

syntax
-arg-min :: $('b \Rightarrow 'a) \Rightarrow pttrn \Rightarrow bool \Rightarrow 'b$
(‹(‹indent=3 notation=‹binder ARG-MIN››ARG′-MIN - -./ -)› [1000, 0, 10] 10)
syntax-consts
-arg-min ⇌ *arg-min*
translations
ARG-MIN f x. P ⇌ *CONST arg-min f* (λx. *P*)

lemma *is-arg-min-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: linorder$
shows $is\text{-}arg\text{-}min\ f\ P\ x = (P\ x \wedge (\forall\, y.\ P\ y \longrightarrow f\ x \leq f\ y))$
by(*auto simp add*: *is-arg-min-def*)

lemma *is-arg-min-antimono*: **fixes** $f :: 'a \Rightarrow ('b :: order)$
shows $[\![\ is\text{-}arg\text{-}min\ f\ P\ x;\ f\ y \leq f\ x;\ P\ y\ ]\!] \Longrightarrow is\text{-}arg\text{-}min\ f\ P\ y$
by (*simp add*: *order.order-iff-strict is-arg-min-def*)

lemma *arg-minI*:
$[\![\ P\ x;$
$\bigwedge y.\ P\ y \Longrightarrow \neg\ f\ y < f\ x;$
$\bigwedge x.\ [\![\ P\ x; \forall\, y.\ P\ y \longrightarrow \neg\ f\ y < f\ x\ ]\!] \Longrightarrow Q\ x\ ]\!]$
$\Longrightarrow Q\ (arg\text{-}min\ f\ P)$
unfolding *arg-min-def is-arg-min-def*
by (*blast intro*!: *someI2-ex*)

lemma *arg-min-equality*:
$[\![\ P\ k; \bigwedge x.\ P\ x \Longrightarrow f\ k \leq f\ x\ ]\!] \Longrightarrow f\ (arg\text{-}min\ f\ P) = f\ k$
for $f :: - \Rightarrow 'a :: order$
by (*rule arg-minI*; *force simp*: *not-less less-le-not-le*)

lemma *wf-linord-ex-has-least*:
$[\![\ wf\ r; \forall\, x\ y.\ (x, y) \in r^+ \longleftrightarrow (y, x) \notin r^*; P\ k\ ]\!]$
$\Longrightarrow \exists\, x.\ P\ x \wedge (\forall\, y.\ P\ y \longrightarrow (m\ x, m\ y) \in r^*)$
by (*force dest*!: *wf-trancl* [*THEN wf-eq-minimal* [*THEN iffD1*, *THEN spec*], **where** $x = m$ ‘ *Collect P*])

lemma *ex-has-least-nat*: $P\ k \Longrightarrow \exists\, x.\ P\ x \wedge (\forall\, y.\ P\ y \longrightarrow m\ x \leq m\ y)$

for $m :: 'a \Rightarrow nat$
unfolding *pred-nat-trancl-eq-le* [*symmetric*]
apply (*rule wf-pred-nat* [*THEN wf-linord-ex-has-least*])
apply (*simp add*: *less-eq linorder-not-le pred-nat-trancl-eq-le*)
by *assumption*

lemma *arg-min-nat-lemma*:
$P\ k \Longrightarrow P(\mathit{arg\text{-}min}\ m\ P) \wedge (\forall y.\ P\ y \longrightarrow m\ (\mathit{arg\text{-}min}\ m\ P) \leq m\ y)$
for $m :: 'a \Rightarrow nat$
unfolding *arg-min-def is-arg-min-linorder*
apply (*rule someI-ex*)
apply (*erule ex-has-least-nat*)
done

lemmas *arg-min-natI* = *arg-min-nat-lemma* [*THEN conjunct1*]

lemma *is-arg-min-arg-min-nat*: **fixes** $m :: 'a \Rightarrow nat$
shows $P\ x \Longrightarrow \mathit{is\text{-}arg\text{-}min}\ m\ P\ (\mathit{arg\text{-}min}\ m\ P)$
by (*metis arg-min-nat-lemma is-arg-min-linorder*)

lemma *arg-min-nat-le*: $P\ x \Longrightarrow m\ (\mathit{arg\text{-}min}\ m\ P) \leq m\ x$
for $m :: 'a \Rightarrow nat$
by (*rule arg-min-nat-lemma* [*THEN conjunct2*, *THEN spec*, *THEN mp*])

lemma *ex-min-if-finite*:
⟦ *finite S*; $S \neq \{\}$ ⟧ $\Longrightarrow \exists m{\in}S.\ \neg(\exists x{\in}S.\ x < (m::'a::order))$
by(*induction rule*: *finite.induct*) (*auto intro*: *order.strict-trans*)

lemma *ex-is-arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: order$
shows ⟦ *finite S*; $S \neq \{\}$ ⟧ $\Longrightarrow \exists x.\ \mathit{is\text{-}arg\text{-}min}\ f\ (\lambda x.\ x \in S)\ x$
unfolding *is-arg-min-def*
using *ex-min-if-finite*[*of f ' S*]
by *auto*

lemma *arg-min-SOME-Min*:
finite S $\Longrightarrow$ *arg-min-on f S* = (*SOME y*. $y \in S \wedge f\ y = \mathit{Min}(f\ `\ S)$)
unfolding *arg-min-on-def arg-min-def is-arg-min-linorder*
apply(*rule arg-cong*[**where** $f = \mathit{Eps}$])
apply (*auto simp*: *fun-eq-iff intro*: *Min-eqI*[*symmetric*])
done

lemma *arg-min-if-finite*: **fixes** $f :: 'a \Rightarrow 'b :: order$
assumes *finite S* $S \neq \{\}$
shows *arg-min-on f S* $\in S$ **and** $\neg(\exists x{\in}S.\ f\ x < f\ (\mathit{arg\text{-}min\text{-}on}\ f\ S))$
using *ex-is-arg-min-if-finite*[*OF assms*, *of f*]
unfolding *arg-min-on-def arg-min-def is-arg-min-def*
by(*auto dest*!: *someI-ex*)

lemma *arg-min-least*: **fixes** $f :: 'a \Rightarrow 'b :: linorder$

shows $[\![$ *finite* S; $S \neq \{\}$; $y \in S$ $]\!] \Longrightarrow f(\textit{arg-min-on}\ f\ S) \leq f\ y$
by(*simp add*: *arg-min-SOME-Min inv-into-def2*[*symmetric*] *f-inv-into-f*)

lemma *arg-min-inj-eq*: **fixes** $f :: 'a \Rightarrow 'b :: \textit{order}$
shows $[\![$ *inj-on* f $\{x.\ P\ x\}$; $P\ a$; $\forall y.\ P\ y \longrightarrow f\ a \leq f\ y$ $]\!] \Longrightarrow \textit{arg-min}\ f\ P = a$
apply(*simp add*: *arg-min-def is-arg-min-def*)
apply(*rule someI2*[*of - a*])
apply (*simp add*: *less-le-not-le*)
by (*metis inj-on-eq-iff less-le mem-Collect-eq*)

55.7 Arg Max

context *ord*
begin

definition *is-arg-max* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \textit{bool}) \Rightarrow 'b \Rightarrow \textit{bool}$ **where**
$\textit{is-arg-max}\ f\ P\ x = (P\ x \wedge \neg(\exists y.\ P\ y \wedge f\ y > f\ x))$

definition *arg-max* :: $('b \Rightarrow 'a) \Rightarrow ('b \Rightarrow \textit{bool}) \Rightarrow 'b$ **where**
$\textit{arg-max}\ f\ P = (\textit{SOME}\ x.\ \textit{is-arg-max}\ f\ P\ x)$

definition *arg-max-on* :: $('b \Rightarrow 'a) \Rightarrow 'b\ \textit{set} \Rightarrow 'b$ **where**
$\textit{arg-max-on}\ f\ S = \textit{arg-max}\ f\ (\lambda x.\ x \in S)$

end

syntax
-arg-max :: $('b \Rightarrow 'a) \Rightarrow \textit{pttrn} \Rightarrow \textit{bool} \Rightarrow 'a$
(‹(‹*indent=3 notation=‹binder ARG-MAX›› ARG'-MAX - -./ -*)› [*1000*, *0*, *10*] *10*)
syntax-consts
-arg-max $\rightleftharpoons$ *arg-max*
translations
ARG-MAX f x. P $\rightleftharpoons$ *CONST arg-max f* $(\lambda x.\ P)$

lemma *is-arg-max-linorder*: **fixes** $f :: 'a \Rightarrow 'b :: \textit{linorder}$
shows $\textit{is-arg-max}\ f\ P\ x = (P\ x \wedge (\forall y.\ P\ y \longrightarrow f\ x \geq f\ y))$
by(*auto simp add*: *is-arg-max-def*)

lemma *arg-maxI*:
$P\ x \Longrightarrow$
$(\bigwedge y.\ P\ y \Longrightarrow \neg\ f\ y > f\ x) \Longrightarrow$
$(\bigwedge x.\ P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow \neg\ f\ y > f\ x \Longrightarrow Q\ x) \Longrightarrow$
$Q\ (\textit{arg-max}\ f\ P)$
unfolding *arg-max-def is-arg-max-def*
by (*blast intro*!: *someI2-ex elim*:)

lemma *arg-max-equality*:
$[\![\ P\ k;\ \bigwedge x.\ P\ x \Longrightarrow f\ x \leq f\ k\ ]\!] \Longrightarrow f\ (\textit{arg-max}\ f\ P) = f\ k$

for $f :: - \Rightarrow 'a$::*order*
apply (*rule arg-maxI* [**where** $f = f$])
apply *assumption*
apply (*simp add*: *less-le-not-le*)
by (*metis le-less*)

lemma *ex-has-greatest-nat-lemma*:
$P\ k \Longrightarrow \forall x.\ P\ x \longrightarrow (\exists y.\ P\ y \wedge \neg\ f\ y \leq f\ x) \Longrightarrow \exists y.\ P\ y \wedge \neg\ f\ y < f\ k + n$
for $f :: 'a \Rightarrow nat$
by (*induct n*) (*force simp*: *le-Suc-eq*)+

lemma *ex-has-greatest-nat*:
assumes $P\ k$
and $\forall y.\ P\ y \longrightarrow (f :: 'a \Rightarrow nat)\ y < b$
shows $\exists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ x)$
proof (*rule ccontr*)
assume $\nexists x.\ P\ x \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ x)$
then have $\forall x.\ P\ x \longrightarrow (\exists y.\ P\ y \wedge \neg\ f\ y \leq f\ x)$
by *auto*
then have $\exists y.\ P\ y \wedge \neg\ f\ y < f\ k + (b - f\ k)$
using *assms ex-has-greatest-nat-lemma*[*of P k f b − f k*]
by *blast*
then show *False*
using *assms* **by** *auto*
qed

lemma *arg-max-nat-lemma*:
$[\![\ P\ k;\ \forall y.\ P\ y \longrightarrow f\ y < b\ ]\!]$
$\Longrightarrow P\ (arg\text{-}max\ f\ P) \wedge (\forall y.\ P\ y \longrightarrow f\ y \leq f\ (arg\text{-}max\ f\ P))$
for $f :: 'a \Rightarrow nat$
unfolding *arg-max-def is-arg-max-linorder*
by (*rule someI-ex*) (*metis ex-has-greatest-nat*)

lemmas *arg-max-natI* = *arg-max-nat-lemma* [*THEN conjunct1*]

lemma *arg-max-nat-le*: $P\ x \Longrightarrow \forall y.\ P\ y \longrightarrow f\ y < b \Longrightarrow f\ x \leq f\ (arg\text{-}max\ f\ P)$
for $f :: 'a \Rightarrow nat$
using *arg-max-nat-lemma* **by** *metis*

end

56 Division in euclidean (semi)rings

theory *Euclidean-Rings*
imports *Int Lattices-Big*
begin

56.1 Euclidean (semi)rings with explicit division and remainder

class *euclidean-semiring* = *semidom-modulo* +
  fixes *euclidean-size* :: $'a \Rightarrow nat$
  assumes *size-0* [*simp*]: *euclidean-size 0* = *0*
  assumes *mod-size-less*:
    $b \neq 0 \Longrightarrow$ *euclidean-size* (*a mod b*) < *euclidean-size b*
  assumes *size-mult-mono*:
    $b \neq 0 \Longrightarrow$ *euclidean-size a* $\leq$ *euclidean-size* (*a* $*$ *b*)
begin

lemma *euclidean-size-eq-0-iff* [*simp*]:
  euclidean-size b = *0* $\longleftrightarrow$ *b* = *0*
proof
  assume *b* = *0*
  then show *euclidean-size b* = *0*
    by *simp*
next
  assume *euclidean-size b* = *0*
  show *b* = *0*
  proof (*rule ccontr*)
    assume *b* $\neq$ *0*
    with *mod-size-less* **have** *euclidean-size* (*b mod b*) < *euclidean-size b* **.**
    with ‹*euclidean-size b* = *0*› **show** *False*
      by *simp*
  qed
qed

lemma *euclidean-size-greater-0-iff* [*simp*]:
  euclidean-size b > *0* $\longleftrightarrow$ *b* $\neq$ *0*
  using *euclidean-size-eq-0-iff* [*symmetric*, *of b*] **by** *safe simp*

lemma *size-mult-mono'*: *b* $\neq$ *0* $\Longrightarrow$ *euclidean-size a* $\leq$ *euclidean-size* (*b* $*$ *a*)
  by (*subst mult.commute*) (*rule size-mult-mono*)

lemma *dvd-euclidean-size-eq-imp-dvd*:
  assumes *a* $\neq$ *0* **and** *euclidean-size a* = *euclidean-size b*
    and *b dvd a*
  shows *a dvd b*
proof (*rule ccontr*)
  assume ¬ *a dvd b*
  hence *b mod a* $\neq$ *0* **using** *mod-0-imp-dvd* [*of b a*] **by** *blast*
  then have *b mod a* $\neq$ *0* **by** (*simp add*: *mod-eq-0-iff-dvd*)
  from ‹*b dvd a*› **have** *b dvd b mod a* **by** (*simp add*: *dvd-mod-iff*)
  then obtain *c* **where** *b mod a* = *b* $*$ *c* **unfolding** *dvd-def* **by** *blast*
    with ‹*b mod a* $\neq$ *0*› **have** *c* $\neq$ *0* **by** *auto*
  with ‹*b mod a* = *b* $*$ *c*› **have** *euclidean-size* (*b mod a*) $\geq$ *euclidean-size b*
    using *size-mult-mono* **by** *force*
  moreover from ‹¬ *a dvd b*› **and** ‹*a* $\neq$ *0*›

```
  have euclidean-size (b mod a) < euclidean-size a
    using mod-size-less by blast
  ultimately show False using ‹euclidean-size a = euclidean-size b›
    by simp
qed

lemma euclidean-size-times-unit:
  assumes is-unit a
  shows   euclidean-size (a * b) = euclidean-size b
proof (rule antisym)
  from assms have [simp]: a ≠ 0 by auto
  thus euclidean-size (a * b) ≥ euclidean-size b by (rule size-mult-mono')
  from assms have is-unit (1 div a) by simp
  hence 1 div a ≠ 0 by (intro notI) simp-all
  hence euclidean-size (a * b) ≤ euclidean-size ((1 div a) * (a * b))
    by (rule size-mult-mono')
  also from assms have (1 div a) * (a * b) = b
    by (simp add: algebra-simps unit-div-mult-swap)
  finally show euclidean-size (a * b) ≤ euclidean-size b .
qed

lemma euclidean-size-unit:
  is-unit a ⟹ euclidean-size a = euclidean-size 1
  using euclidean-size-times-unit [of a 1] by simp

lemma unit-iff-euclidean-size:
  is-unit a ⟷ euclidean-size a = euclidean-size 1 ∧ a ≠ 0
proof safe
  assume A: a ≠ 0 and B: euclidean-size a = euclidean-size 1
  show is-unit a
    by (rule dvd-euclidean-size-eq-imp-dvd [OF A B]) simp-all
qed (auto intro: euclidean-size-unit)

lemma euclidean-size-times-nonunit:
  assumes a ≠ 0 b ≠ 0 ¬ is-unit a
  shows   euclidean-size b < euclidean-size (a * b)
proof (rule ccontr)
  assume ¬euclidean-size b < euclidean-size (a * b)
  with size-mult-mono'[OF assms(1), of b]
    have eq: euclidean-size (a * b) = euclidean-size b by simp
  have a * b dvd b
    by (rule dvd-euclidean-size-eq-imp-dvd [OF - eq])
       (use assms in simp-all)
  hence a * b dvd 1 * b by simp
  with ‹b ≠ 0› have is-unit a by (subst (asm) dvd-times-right-cancel-iff)
  with assms(3) show False by contradiction
qed

lemma dvd-imp-size-le:
```

assumes *a dvd b b* $\neq$ *0*
shows *euclidean-size a* $\leq$ *euclidean-size b*
using *assms* **by** (*auto simp*: *size-mult-mono*)

lemma *dvd-proper-imp-size-less*:
assumes *a dvd b* $\neg$ *b dvd a b* $\neq$ *0*
shows *euclidean-size a* $<$ *euclidean-size b*
proof $-$
from *assms*(*1*) **obtain** *c* **where** *b* = *a* $*$ *c* **by** (*erule dvdE*)
hence *z*: *b* = *c* $*$ *a* **by** (*simp add*: *mult.commute*)
from *z assms* **have** $\neg$*is-unit c* **by** (*auto simp*: *mult.commute mult-unit-dvd-iff*)
with *z assms* **show** *?thesis*
by (*auto intro*!: *euclidean-size-times-nonunit*)
qed

lemma *unit-imp-mod-eq-0*:
a mod b = *0* **if** *is-unit b*
using *that* **by** (*simp add*: *mod-eq-0-iff-dvd unit-imp-dvd*)

lemma *mod-eq-self-iff-div-eq-0*:
a mod b = *a* $\longleftrightarrow$ *a div b* = *0* (**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?P*
with *div-mult-mod-eq* [*of a b*] **show** *?Q*
by *auto*
next
assume *?Q*
with *div-mult-mod-eq* [*of a b*] **show** *?P*
by *simp*
qed

lemma *coprime-mod-left-iff* [*simp*]:
coprime (*a mod b*) *b* $\longleftrightarrow$ *coprime a b* **if** *b* $\neq$ *0*
by (*rule iffI*; *rule coprimeI*)
(*use that* **in** ‹*auto dest*!: *dvd-mod-imp-dvd coprime-common-divisor simp add*: *dvd-mod-iff*›)

lemma *coprime-mod-right-iff* [*simp*]:
coprime a (*b mod a*) $\longleftrightarrow$ *coprime a b* **if** *a* $\neq$ *0*
using *that coprime-mod-left-iff* [*of a b*] **by** (*simp add*: *ac-simps*)

end

class *euclidean-ring* = *idom-modulo* + *euclidean-semiring*
begin

lemma *dvd-diff-commute* [*ac-simps*]:
a dvd c $-$ *b* $\longleftrightarrow$ *a dvd b* $-$ *c*
proof $-$

have *a dvd c − b ⟷ a dvd (c − b) * − 1*
by (*subst dvd-mult-unit-iff*) *simp-all*
then show *?thesis*
by *simp*
qed

end

56.2 Euclidean (semi)rings with cancel rules

class *euclidean-semiring-cancel = euclidean-semiring +*
assumes *div-mult-self1* [*simp*]: *b ≠ 0 ⟹ (a + c * b) div b = c + a div b*
and *div-mult-mult1* [*simp*]: *c ≠ 0 ⟹ (c * a) div (c * b) = a div b*
begin

lemma *div-mult-self2* [*simp*]:
assumes *b ≠ 0*
shows *(a + b * c) div b = c + a div b*
using *assms div-mult-self1* [*of b a c*] **by** (*simp add*: *mult.commute*)

lemma *div-mult-self3* [*simp*]:
assumes *b ≠ 0*
shows *(c * b + a) div b = c + a div b*
using *assms* **by** (*simp add*: *add.commute*)

lemma *div-mult-self4* [*simp*]:
assumes *b ≠ 0*
shows *(b * c + a) div b = c + a div b*
using *assms* **by** (*simp add*: *add.commute*)

lemma *mod-mult-self1* [*simp*]: *(a + c * b) mod b = a mod b*
proof (*cases b = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
have *a + c * b = (a + c * b) div b * b + (a + c * b) mod b*
by (*simp add*: *div-mult-mod-eq*)
also from *False div-mult-self1* [*of b a c*] **have**
*... = (c + a div b) * b + (a + c * b) mod b*
by (*simp add*: *algebra-simps*)
finally have *a = a div b * b + (a + c * b) mod b*
by (*simp add*: *add.commute* [*of a*] *add.assoc distrib-right*)
then have *a div b * b + (a + c * b) mod b = a div b * b + a mod b*
by (*simp add*: *div-mult-mod-eq*)
then show *?thesis* **by** *simp*
qed

lemma *mod-mult-self2* [*simp*]:
*(a + b * c) mod b = a mod b*

by (*simp add*: *mult.commute* [*of b*])

lemma *mod-mult-self3* [*simp*]:
(*c* ∗ *b* + *a*) *mod b* = *a mod b*
by (*simp add*: *add.commute*)

lemma *mod-mult-self4* [*simp*]:
(*b* ∗ *c* + *a*) *mod b* = *a mod b*
by (*simp add*: *add.commute*)

lemma *mod-mult-self1-is-0* [*simp*]:
b ∗ *a mod b* = *0*
using *mod-mult-self2* [*of 0 b a*] **by** *simp*

lemma *mod-mult-self2-is-0* [*simp*]:
a ∗ *b mod b* = *0*
using *mod-mult-self1* [*of 0 a b*] **by** *simp*

lemma *div-add-self1*:
assumes *b* ≠ *0*
shows (*b* + *a*) *div b* = *a div b* + *1*
using *assms div-mult-self1* [*of b a 1*] **by** (*simp add*: *add.commute*)

lemma *div-add-self2*:
assumes *b* ≠ *0*
shows (*a* + *b*) *div b* = *a div b* + *1*
using *assms div-add-self1* [*of b a*] **by** (*simp add*: *add.commute*)

lemma *mod-add-self1* [*simp*]:
(*b* + *a*) *mod b* = *a mod b*
using *mod-mult-self1* [*of a 1 b*] **by** (*simp add*: *add.commute*)

lemma *mod-add-self2* [*simp*]:
(*a* + *b*) *mod b* = *a mod b*
using *mod-mult-self1* [*of a 1 b*] **by** *simp*

lemma *mod-div-trivial* [*simp*]:
a mod b div b = *0*
proof (*cases b* = *0*)
assume *b* = *0*
thus *?thesis* **by** *simp*
next
assume *b* ≠ *0*
hence *a div b* + *a mod b div b* = (*a mod b* + *a div b* ∗ *b*) *div b*
by (*rule div-mult-self1* [*symmetric*])
also have ... = *a div b*
by (*simp only*: *mod-div-mult-eq*)
also have ... = *a div b* + *0*
by *simp*

finally show *?thesis*
by (*rule add-left-imp-eq*)
qed

lemma *mod-mod-trivial* [*simp*]:
a mod b mod b = a mod b
proof −
have *a mod b mod b = (a mod b + a div b * b) mod b*
by (*simp only*: *mod-mult-self1*)
also have ... = *a mod b*
by (*simp only*: *mod-div-mult-eq*)
finally show *?thesis* **.**
qed

lemma *mod-mod-cancel*:
assumes *c dvd b*
shows *a mod b mod c = a mod c*
proof −
from ‹*c dvd b*› **obtain** *k* **where** *b = c * k*
by (*rule dvdE*)
have *a mod b mod c = a mod (c * k) mod c*
by (*simp only*: ‹*b = c * k*›)
also have ... = *(a mod (c * k) + a div (c * k) * k * c) mod c*
by (*simp only*: *mod-mult-self1*)
also have ... = *(a div (c * k) * (c * k) + a mod (c * k)) mod c*
by (*simp only*: *ac-simps*)
also have ... = *a mod c*
by (*simp only*: *div-mult-mod-eq*)
finally show *?thesis* **.**
qed

lemma *div-mult-mult2* [*simp*]:
*c ≠ 0 ⟹ (a * c) div (b * c) = a div b*
by (*drule div-mult-mult1*) (*simp add*: *mult.commute*)

lemma *div-mult-mult1-if* [*simp*]:
*(c * a) div (c * b) = (if c = 0 then 0 else a div b)*
by *simp-all*

lemma *mod-mult-mult1*:
*(c * a) mod (c * b) = c * (a mod b)*
proof (*cases c = 0*)
case *True* **then show** *?thesis* **by** *simp*
next
case *False*
from *div-mult-mod-eq*
have *((c * a) div (c * b)) * (c * b) + (c * a) mod (c * b) = c * a* **.**
with *False* **have** *c * ((a div b) * b + a mod b) + (c * a) mod (c * b)*
*= c * a + c * (a mod b)* **by** (*simp add*: *algebra-simps*)

with *div-mult-mod-eq* **show** *?thesis* **by** *simp*
qed

lemma *mod-mult-mult2*:
(*a* ∗ *c*) *mod* (*b* ∗ *c*) = (*a mod b*) ∗ *c*
using *mod-mult-mult1* [*of c a b*] **by** (*simp add*: *mult.commute*)

lemma *mult-mod-left*: (*a mod b*) ∗ *c* = (*a* ∗ *c*) *mod* (*b* ∗ *c*)
by (*fact mod-mult-mult2* [*symmetric*])

lemma *mult-mod-right*: *c* ∗ (*a mod b*) = (*c* ∗ *a*) *mod* (*c* ∗ *b*)
by (*fact mod-mult-mult1* [*symmetric*])

lemma *dvd-mod*: *k dvd m* ⟹ *k dvd n* ⟹ *k dvd* (*m mod n*)
unfolding *dvd-def* **by** (*auto simp add*: *mod-mult-mult1*)

lemma *div-plus-div-distrib-dvd-left*:
c dvd a ⟹ (*a* + *b*) *div c* = *a div c* + *b div c*
by (*cases c* = *0*) *auto*

lemma *div-plus-div-distrib-dvd-right*:
c dvd b ⟹ (*a* + *b*) *div c* = *a div c* + *b div c*
using *div-plus-div-distrib-dvd-left* [*of c b a*]
by (*simp add*: *ac-simps*)

lemma *sum-div-partition*:
‹(∑ *a*∈*A*. *f a*) *div b* = (∑ *a*∈*A* ∩ {*a*. *b dvd f a*}. *f a div b*) + (∑ *a*∈*A* ∩ {*a*. ¬ *b dvd f a*}. *f a*) *div b*›
if ‹*finite A*›
proof −
have ‹*A* = *A* ∩ {*a*. *b dvd f a*} ∪ *A* ∩ {*a*. ¬ *b dvd f a*}›
by *auto*
then have ‹(∑ *a*∈*A*. *f a*) = (∑ *a*∈*A* ∩ {*a*. *b dvd f a*} ∪ *A* ∩ {*a*. ¬ *b dvd f a*}. *f a*)›
by *simp*
also have ‹. . . = (∑ *a*∈*A* ∩ {*a*. *b dvd f a*}. *f a*) + (∑ *a*∈*A* ∩ {*a*. ¬ *b dvd f a*}. *f a*)›
using ‹*finite A*› **by** (*auto intro*: *sum.union-inter-neutral*)
finally have ∗: ‹*sum f A* = *sum f* (*A* ∩ {*a*. *b dvd f a*}) + *sum f* (*A* ∩ {*a*. ¬ *b dvd f a*})› **.**
define *B* **where** *B*: ‹*B* = *A* ∩ {*a*. *b dvd f a*}›
with ‹*finite A*› **have** ‹*finite B*› **and** ‹*a* ∈ *B* ⟹ *b dvd f a*› **for** *a*
by *simp-all*
then have ‹(∑ *a*∈*B*. *f a*) *div b* = (∑ *a*∈*B*. *f a div b*)› **and** ‹*b dvd* (∑ *a*∈*B*. *f a*)›
by *induction* (*simp-all add*: *div-plus-div-distrib-dvd-left*)
then show *?thesis* **using** ∗
by (*simp add*: *B div-plus-div-distrib-dvd-left*)
qed

named-theorems *mod-simps*

Addition respects modular equivalence.

lemma *mod-add-left-eq* [*mod-simps*]:
  $(a \bmod c + b) \bmod c = (a + b) \bmod c$
proof −
  have $(a + b) \bmod c = (a \operatorname{div} c * c + a \bmod c + b) \bmod c$
    by (*simp only*: *div-mult-mod-eq*)
  also have $\ldots = (a \bmod c + b + a \operatorname{div} c * c) \bmod c$
    by (*simp only*: *ac-simps*)
  also have $\ldots = (a \bmod c + b) \bmod c$
    by (*rule mod-mult-self1*)
  finally show *?thesis*
    by (*rule sym*)
qed

lemma *mod-add-right-eq* [*mod-simps*]:
  $(a + b \bmod c) \bmod c = (a + b) \bmod c$
  using *mod-add-left-eq* [*of b c a*] **by** (*simp add*: *ac-simps*)

lemma *mod-add-eq*:
  $(a \bmod c + b \bmod c) \bmod c = (a + b) \bmod c$
  by (*simp add*: *mod-add-left-eq mod-add-right-eq*)

lemma *mod-sum-eq* [*mod-simps*]:
  $(\sum i{\in}A.\ f\ i \bmod a) \bmod a = sum\ f\ A \bmod a$
proof (*induct A rule*: *infinite-finite-induct*)
  case (*insert i A*)
  then have $(\sum i{\in}insert\ i\ A.\ f\ i \bmod a) \bmod a$
    $= (f\ i \bmod a + (\sum i{\in}A.\ f\ i \bmod a)) \bmod a$
    by *simp*
  also have $\ldots = (f\ i + (\sum i{\in}A.\ f\ i \bmod a) \bmod a) \bmod a$
    by (*simp add*: *mod-simps*)
  also have $\ldots = (f\ i + (\sum i{\in}A.\ f\ i) \bmod a) \bmod a$
    by (*simp add*: *insert.hyps*)
  finally show *?case*
    by (*simp add*: *insert.hyps mod-simps*)
qed *simp-all*

lemma *mod-add-cong*:
  assumes $a \bmod c = a' \bmod c$
  assumes $b \bmod c = b' \bmod c$
  shows $(a + b) \bmod c = (a' + b') \bmod c$
proof −
  have $(a \bmod c + b \bmod c) \bmod c = (a' \bmod c + b' \bmod c) \bmod c$
    unfolding *assms* **..**
  then show *?thesis*
    by (*simp add*: *mod-add-eq*)
qed

Multiplication respects modular equivalence.

lemma *mod-mult-left-eq* [*mod-simps*]:
  $((a \bmod c) * b) \bmod c = (a * b) \bmod c$
proof −
  have $(a * b) \bmod c = ((a \operatorname{div} c * c + a \bmod c) * b) \bmod c$
    by (*simp only*: *div-mult-mod-eq*)
  also have $\ldots = (a \bmod c * b + a \operatorname{div} c * b * c) \bmod c$
    by (*simp only*: *algebra-simps*)
  also have $\ldots = (a \bmod c * b) \bmod c$
    by (*rule mod-mult-self1*)
  finally show *?thesis*
    by (*rule sym*)
qed

lemma *mod-mult-right-eq* [*mod-simps*]:
  $(a * (b \bmod c)) \bmod c = (a * b) \bmod c$
  using *mod-mult-left-eq* [*of b c a*] **by** (*simp add*: *ac-simps*)

lemma *mod-mult-eq*:
  $((a \bmod c) * (b \bmod c)) \bmod c = (a * b) \bmod c$
  by (*simp add*: *mod-mult-left-eq mod-mult-right-eq*)

lemma *mod-prod-eq* [*mod-simps*]:
  $(\prod i{\in}A.\ f\ i \bmod a) \bmod a = prod\ f\ A \bmod a$
proof (*induct A rule*: *infinite-finite-induct*)
  case (*insert i A*)
  then have $(\prod i{\in}insert\ i\ A.\ f\ i \bmod a) \bmod a$
    $= (f\ i \bmod a * (\prod i{\in}A.\ f\ i \bmod a)) \bmod a$
    by *simp*
  also have $\ldots = (f\ i * ((\prod i{\in}A.\ f\ i \bmod a) \bmod a)) \bmod a$
    by (*simp add*: *mod-simps*)
  also have $\ldots = (f\ i * ((\prod i{\in}A.\ f\ i) \bmod a)) \bmod a$
    by (*simp add*: *insert.hyps*)
  finally show *?case*
    by (*simp add*: *insert.hyps mod-simps*)
qed *simp-all*

lemma *mod-mult-cong*:
  assumes $a \bmod c = a' \bmod c$
  assumes $b \bmod c = b' \bmod c$
  shows $(a * b) \bmod c = (a' * b') \bmod c$
proof −
  have $(a \bmod c * (b \bmod c)) \bmod c = (a' \bmod c * (b' \bmod c)) \bmod c$
    unfolding *assms* **..**
  then show *?thesis*
    by (*simp add*: *mod-mult-eq*)
qed

Exponentiation respects modular equivalence.

lemma *power-mod* [*mod-simps*]:
  $((a\ mod\ b)\ \hat{}\ n)\ mod\ b = (a\ \hat{}\ n)\ mod\ b$
proof (*induct n*)
  case *0*
  then show *?case* **by** *simp*
next
  case (*Suc n*)
  have $(a\ mod\ b)\ \hat{}\ Suc\ n\ mod\ b = (a\ mod\ b) * ((a\ mod\ b)\ \hat{}\ n\ mod\ b)\ mod\ b$
    by (*simp add*: *mod-mult-right-eq*)
  with *Suc* **show** *?case*
    by (*simp add*: *mod-mult-left-eq mod-mult-right-eq*)
qed

lemma *power-diff-power-eq*:
  $‹a\ \hat{}\ m\ div\ a\ \hat{}\ n = (if\ n \leq m\ then\ a\ \hat{}\ (m - n)\ else\ 1\ div\ a\ \hat{}\ (n - m))›$
    if $‹a \neq 0›$
proof (*cases* $‹n \leq m›$)
  case *True*
  with *that power-diff* [*symmetric*, *of a n m*] **show** *?thesis* **by** *simp*
next
  case *False*
  then obtain *q* **where** *n*: $‹n = m + Suc\ q›$
    by (*auto simp add*: *not-le dest*: *less-imp-Suc-add*)
  then have $‹a\ \hat{}\ m\ div\ a\ \hat{}\ n = (a\ \hat{}\ m * 1)\ div\ (a\ \hat{}\ m * a\ \hat{}\ Suc\ q)›$
    by (*simp add*: *power-add ac-simps*)
  moreover from *that* **have** $‹a\ \hat{}\ m \neq 0›$
    by *simp*
  ultimately have $‹a\ \hat{}\ m\ div\ a\ \hat{}\ n = 1\ div\ a\ \hat{}\ Suc\ q›$
    by (*subst* (*asm*) *div-mult-mult1*) *simp*
  with *False n* **show** *?thesis*
    by *simp*
qed

end

class *euclidean-ring-cancel* = *euclidean-ring* + *euclidean-semiring-cancel*
begin

subclass *idom-divide* **..**

lemma *div-minus-minus* [*simp*]: $(-a)\ div\ (-b) = a\ div\ b$
  using *div-mult-mult1* [*of* $-1\ a\ b$] **by** *simp*

lemma *mod-minus-minus* [*simp*]: $(-a)\ mod\ (-b) = -(a\ mod\ b)$
  using *mod-mult-mult1* [*of* $-1\ a\ b$] **by** *simp*

lemma *div-minus-right*: $a\ div\ (-b) = (-a)\ div\ b$
  using *div-minus-minus* [*of* $-a\ b$] **by** *simp*

lemma *mod-minus-right*: *a mod (− b) = − ((− a) mod b)*
using *mod-minus-minus* [*of − a b*] **by** *simp*

lemma *div-minus1-right* [*simp*]: *a div (− 1) = − a*
using *div-minus-right* [*of a 1*] **by** *simp*

lemma *mod-minus1-right* [*simp*]: *a mod (− 1) = 0*
using *mod-minus-right* [*of a 1*] **by** *simp*

Negation respects modular equivalence.

lemma *mod-minus-eq* [*mod-simps*]:
(− (a mod b)) mod b = (− a) mod b
proof −
have *(− a) mod b = (− (a div b * b + a mod b)) mod b*
by (*simp only*: *div-mult-mod-eq*)
also have *... = (− (a mod b) + − (a div b) * b) mod b*
by (*simp add*: *ac-simps*)
also have *... = (− (a mod b)) mod b*
by (*rule mod-mult-self1*)
finally show *?thesis*
by (*rule sym*)
qed

lemma *mod-minus-cong*:
assumes *a mod b = a′ mod b*
shows *(− a) mod b = (− a′) mod b*
proof −
have *(− (a mod b)) mod b = (− (a′ mod b)) mod b*
unfolding *assms* **..**
then show *?thesis*
by (*simp add*: *mod-minus-eq*)
qed

Subtraction respects modular equivalence.

lemma *mod-diff-left-eq* [*mod-simps*]:
(a mod c − b) mod c = (a − b) mod c
using *mod-add-cong* [*of a c a mod c − b − b*]
by *simp*

lemma *mod-diff-right-eq* [*mod-simps*]:
(a − b mod c) mod c = (a − b) mod c
using *mod-add-cong* [*of a c a − b − (b mod c)*] *mod-minus-cong* [*of b mod c c b*]
by *simp*

lemma *mod-diff-eq*:
(a mod c − b mod c) mod c = (a − b) mod c
using *mod-add-cong* [*of a c a mod c − b − (b mod c)*] *mod-minus-cong* [*of b mod c c b*]

by *simp*

lemma *mod-diff-cong*:
assumes $a\ mod\ c = a'\ mod\ c$
assumes $b\ mod\ c = b'\ mod\ c$
shows $(a - b)\ mod\ c = (a' - b')\ mod\ c$
using *assms mod-add-cong* [*of a c a' − b − b'*] *mod-minus-cong* [*of b c b'*]
by *simp*

lemma *minus-mod-self2* [*simp*]:
$(a - b)\ mod\ b = a\ mod\ b$
using *mod-diff-right-eq* [*of a b b*]
by (*simp add*: *mod-diff-right-eq*)

lemma *minus-mod-self1* [*simp*]:
$(b - a)\ mod\ b = -\ a\ mod\ b$
using *mod-add-self2* [*of − a b*] **by** *simp*

lemma *mod-eq-dvd-iff*:
$a\ mod\ c = b\ mod\ c \longleftrightarrow c\ dvd\ a - b$ (**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?P*
then have $(a\ mod\ c - b\ mod\ c)\ mod\ c = 0$
by *simp*
then show *?Q*
by (*simp add*: *dvd-eq-mod-eq-0 mod-simps*)
next
assume *?Q*
then obtain *d* **where** *d*: $a - b = c * d$ **..**
then have $a = c * d + b$
by (*simp add*: *algebra-simps*)
then show *?P* **by** *simp*
qed

lemma *mod-eqE*:
assumes $a\ mod\ c = b\ mod\ c$
obtains *d* **where** $b = a + c * d$
proof −
from *assms* **have** $c\ dvd\ a - b$
by (*simp add*: *mod-eq-dvd-iff*)
then obtain *d* **where** $a - b = c * d$ **..**
then have $b = a + c * -\ d$
by (*simp add*: *algebra-simps*)
with *that* **show** *thesis* **.**
qed

lemma *invertible-coprime*:
coprime a c **if** $a * b\ mod\ c = 1$
by (*rule coprimeI*) (*use that dvd-mod-iff* [*of - c a * b*] **in** *auto*)

end

56.3 Uniquely determined division

class *unique-euclidean-semiring* = *euclidean-semiring* +
  assumes *euclidean-size-mult*: ‹*euclidean-size* (*a* ∗ *b*) = *euclidean-size a* ∗ *euclidean-size b*›
  fixes *division-segment* :: ‹'*a* ⇒ '*a*›
  assumes *is-unit-division-segment* [*simp*]: ‹*is-unit* (*division-segment a*)›
    and *division-segment-mult*:
    ‹*a* ≠ *0* ⟹ *b* ≠ *0* ⟹ *division-segment* (*a* ∗ *b*) = *division-segment a* ∗ *division-segment b*›
    and *division-segment-mod*:
    ‹*b* ≠ *0* ⟹ ¬ *b dvd a* ⟹ *division-segment* (*a mod b*) = *division-segment b*›
  assumes *div-bounded*:
    ‹*b* ≠ *0* ⟹ *division-segment r* = *division-segment b*
    ⟹ *euclidean-size r* < *euclidean-size b*
    ⟹ (*q* ∗ *b* + *r*) *div b* = *q*›
begin

lemma *division-segment-not-0* [*simp*]:
  ‹*division-segment a* ≠ *0*›
  using *is-unit-division-segment* [*of a*] *is-unitE* [*of* ‹*division-segment a*›] **by** *blast*

lemma *euclidean-relationI* [*case-names by0 divides euclidean-relation*]:
  ‹(*a div b*, *a mod b*) = (*q*, *r*)›
    if *by0*: ‹*b* = *0* ⟹ *q* = *0* ∧ *r* = *a*›
    and *divides*: ‹*b* ≠ *0* ⟹ *b dvd a* ⟹ *r* = *0* ∧ *a* = *q* ∗ *b*›
    and *euclidean-relation*: ‹*b* ≠ *0* ⟹ ¬ *b dvd a* ⟹ *division-segment r* = *division-segment b*
      ∧ *euclidean-size r* < *euclidean-size b* ∧ *a* = *q* ∗ *b* + *r*›
proof (*cases* ‹*b* = *0*›)
  case *True*
  with *by0* **show** *?thesis*
    by *simp*
next
  case *False*
  show *?thesis*
  proof (*cases* ‹*b dvd a*›)
    case *True*
    with ‹*b* ≠ *0*› *divides*
    show *?thesis*
      by *simp*
  next
    case *False*
    with ‹*b* ≠ *0*› *euclidean-relation*
    have ‹*division-segment r* = *division-segment b*›
      ‹*euclidean-size r* < *euclidean-size b*› ‹*a* = *q* ∗ *b* + *r*›

```
    by simp-all
  from ‹b ≠ 0› ‹division-segment r = division-segment b›
    ‹euclidean-size r < euclidean-size b›
  have ‹(q * b + r) div b = q›
    by (rule div-bounded)
  with ‹a = q * b + r›
  have ‹q = a div b›
    by simp
  from ‹a = q * b + r›
  have ‹a div b * b + a mod b = q * b + r›
    by (simp add: div-mult-mod-eq)
  with ‹q = a div b›
  have ‹q * b + a mod b = q * b + r›
    by simp
  then have ‹r = a mod b›
    by simp
  with ‹q = a div b›
  show ?thesis
    by simp
 qed
qed

subclass euclidean-semiring-cancel
proof
 fix a b c
 assume ‹b ≠ 0›
 have ‹((a + c * b) div b, (a + c * b) mod b) = (c + a div b, a mod b)›
 proof (induction rule: euclidean-relationI)
  case by0
  with ‹b ≠ 0›
  show ?case
    by simp
 next
  case divides
  then show ?case
    by (simp add: algebra-simps dvd-add-left-iff)
 next
  case euclidean-relation
  then have ‹¬ b dvd a›
    by (simp add: dvd-add-left-iff)
  have ‹a mod b + (b * c + b * (a div b)) = b * c + ((a div b) * b + a mod b)›
    by (simp add: ac-simps)
  with ‹b ≠ 0› have *: ‹a mod b + (b * c + b * (a div b)) = b * c + a›
    by (simp add: div-mult-mod-eq)
  from ‹¬ b dvd a› euclidean-relation show ?case
    by (simp-all add: algebra-simps division-segment-mod mod-size-less *)
 qed
 then show ‹(a + c * b) div b = c + a div b›
  by simp
```

next
fix *a b c*
assume ‹*c ≠ 0*›
have ‹((*c * a*) *div* (*c * b*), (*c * a*) *mod* (*c * b*)) = (*a div b*, *c * *(*a mod b*))›
proof (*induction rule*: *euclidean-relationI*)
case *by0*
with ‹*c ≠ 0*› **show** *?case*
by *simp*
next
case *divides*
then show *?case*
by (*auto simp add*: *algebra-simps*)
next
case *euclidean-relation*
then have ‹*b ≠ 0*› ‹*a mod b ≠ 0*›
by (*simp-all add*: *mod-eq-0-iff-dvd*)
have ‹*c * *(*a mod b*) + *b * *(*c * *(*a div b*)) = *c * *((*a div b*) * *b* + *a mod b*)›
by (*simp add*: *algebra-simps*)
with ‹*b ≠ 0*› **have** *: ‹*c * *(*a mod b*) + *b * *(*c * *(*a div b*)) = *c * a*›
by (*simp add*: *div-mult-mod-eq*)
from ‹*b ≠ 0*› ‹*c ≠ 0*› **have** ‹*euclidean-size c * euclidean-size* (*a mod b*)
< *euclidean-size c * euclidean-size b*›
using *mod-size-less* [*of b a*] **by** *simp*
with *euclidean-relation* ‹*b ≠ 0*› ‹*a mod b ≠ 0*› **show** *?case*
by (*simp add*: *algebra-simps division-segment-mult division-segment-mod euclidean-size-mult* *)
qed
then show ‹(*c * a*) *div* (*c * b*) = *a div b*›
by *simp*
qed

lemma *div-eq-0-iff*:
‹*a div b = 0* ⟷ *euclidean-size a < euclidean-size b* ∨ *b = 0*› (**is** - ⟷ *?P*)
if ‹*division-segment a = division-segment b*›
proof (*cases* ‹*a = 0* ∨ *b = 0*›)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have ‹*a ≠ 0*› ‹*b ≠ 0*›
by *simp-all*
have ‹*a div b = 0* ⟷ *euclidean-size a < euclidean-size b*›
proof
assume ‹*a div b = 0*›
then have ‹*a mod b = a*›
using *div-mult-mod-eq* [*of a b*] **by** *simp*
with ‹*b ≠ 0*› *mod-size-less* [*of b a*]
show ‹*euclidean-size a < euclidean-size b*›
by *simp*

```
  next
    assume ‹euclidean-size a < euclidean-size b›
    have ‹(a div b, a mod b) = (0, a)›
    proof (induction rule: euclidean-relationI)
      case by0
      show ?case
        by simp
    next
      case divides
      with ‹euclidean-size a < euclidean-size b› show ?case
        using dvd-imp-size-le [of b a] ‹a ≠ 0› by simp
    next
      case euclidean-relation
      with ‹euclidean-size a < euclidean-size b› that
      show ?case
        by simp
    qed
    then show ‹a div b = 0›
      by simp
  qed
  with ‹b ≠ 0› show ?thesis
    by simp
qed

lemma div-mult1-eq:
  ‹(a * b) div c = a * (b div c) + a * (b mod c) div c›
proof −
  have *: ‹(a * b) mod c + (a * (c * (b div c)) + c * (a * (b mod c) div c)) = a
* b› (is ‹?A + (?B + ?C) = -›)
  proof −
    have ‹?A = a * (b mod c) mod c›
      by (simp add: mod-mult-right-eq)
    then have ‹?C + ?A = a * (b mod c)›
      by (simp add: mult-div-mod-eq)
    then have ‹?B + (?C + ?A) = a * (c * (b div c) + (b mod c))›
      by (simp add: algebra-simps)
    also have ‹... = a * b›
      by (simp add: mult-div-mod-eq)
    finally show ?thesis
      by (simp add: algebra-simps)
  qed
  have ‹((a * b) div c, (a * b) mod c) = (a * (b div c) + a * (b mod c) div c, (a
* b) mod c)›
  proof (induction rule: euclidean-relationI)
    case by0
    then show ?case by simp
  next
    case divides
    with * show ?case
```

```
    by (simp add: algebra-simps)
  next
    case euclidean-relation
    with * show ?case
      by (simp add: division-segment-mod mod-size-less algebra-simps)
  qed
  then show ?thesis
    by simp
qed

lemma div-add1-eq:
  ‹(a + b) div c = a div c + b div c + (a mod c + b mod c) div c›
proof −
  have *: ‹(a + b) mod c + (c * (a div c) + (c * (b div c) + c * ((a mod c + b
mod c) div c))) = a + b›
    (is ‹?A + (?B + (?C + ?D)) = -›)
  proof −
    have ‹?A + (?B + (?C + ?D)) = ?A + ?D + (?B + ?C)›
      by (simp add: ac-simps)
    also have ‹?A + ?D = (a mod c + b mod c) mod c + ?D›
      by (simp add: mod-add-eq)
    also have ‹... = a mod c + b mod c›
      by (simp add: mod-mult-div-eq)
    finally have ‹?A + (?B + (?C + ?D)) = (a mod c + ?B) + (b mod c + ?C)›
      by (simp add: ac-simps)
    then show ?thesis
      by (simp add: mod-mult-div-eq)
  qed
  have ‹((a + b) div c, (a + b) mod c) = (a div c + b div c + (a mod c + b mod
c) div c, (a + b) mod c)›
  proof (induction rule: euclidean-relationI)
    case by0
    then show ?case
      by simp
  next
    case divides
    with * show ?case
      by (simp add: algebra-simps)
  next
    case euclidean-relation
    with * show ?case
      by (simp add: division-segment-mod mod-size-less algebra-simps)
  qed
  then show ?thesis
    by simp
qed

end
```

class *unique-euclidean-ring* = *euclidean-ring* + *unique-euclidean-semiring*
begin

subclass *euclidean-ring-cancel* **..**

end

56.4 Division on *nat*

instantiation *nat* :: *normalization-semidom*
begin

definition *normalize-nat* :: ‹*nat* ⇒ *nat*›
  where [*simp*]: ‹*normalize* = (*id* :: *nat* ⇒ *nat*)›

definition *unit-factor-nat* :: ‹*nat* ⇒ *nat*›
  where ‹*unit-factor n* = *of-bool* (*n* > *0*)› **for** *n* :: *nat*

lemma *unit-factor-simps* [*simp*]:
  ‹*unit-factor 0* = (*0*::*nat*)›
  ‹*unit-factor* (*Suc n*) = *1*›
  by (*simp-all add*: *unit-factor-nat-def*)

definition *divide-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
  where ‹*m div n* = (*if n* = *0 then 0 else Max* {*k*. *k* ∗ *n* ≤ *m*})› **for** *m n* :: *nat*

instance
  by *standard* (*auto simp add*: *divide-nat-def ac-simps unit-factor-nat-def intro*: *Max-eqI*)

end

lemma *coprime-Suc-0-left* [*simp*]:
  coprime (*Suc 0*) *n*
  using *coprime-1-left* [*of n*] **by** *simp*

lemma *coprime-Suc-0-right* [*simp*]:
  coprime n (*Suc 0*)
  using *coprime-1-right* [*of n*] **by** *simp*

lemma *coprime-common-divisor-nat*: *coprime a b* ⟹ *x dvd a* ⟹ *x dvd b* ⟹ *x* = *1*
  for *a b* :: *nat*
  by (*drule coprime-common-divisor* [*of - - x*]) *simp-all*

instantiation *nat* :: *unique-euclidean-semiring*
begin

definition *euclidean-size-nat* :: ‹*nat* ⇒ *nat*›

where [*simp*]: ‹*euclidean-size-nat* = *id*›

definition *division-segment-nat* :: ‹*nat* ⇒ *nat*›
where [*simp*]: ‹*division-segment n* = *1*› **for** *n* :: *nat*

definition *modulo-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*m mod n* = *m* − (*m div n* ∗ *n*)› **for** *m n* :: *nat*

instance proof
fix *m n* :: *nat*
have *ex*: ∃ *k*. *k* ∗ *n* ≤ *l* **for** *l* :: *nat*
by (*rule exI* [*of* - *0*]) *simp*
have *fin*: *finite* {*k*. *k* ∗ *n* ≤ *l*} **if** *n* > *0* **for** *l*
proof −
from *that* **have** {*k*. *k* ∗ *n* ≤ *l*} ⊆ {*k*. *k* ≤ *l*}
by (*cases n*) *auto*
then show *?thesis*
by (*rule finite-subset*) *simp*
qed
have *mult-div-unfold*: *n* ∗ (*m div n*) = *Max* {*l*. *l* ≤ *m* ∧ *n dvd l*}
proof (*cases n* = *0*)
case *True*
moreover have {*l*. *l* = *0* ∧ *l* ≤ *m*} = {*0*::*nat*}
by *auto*
ultimately show *?thesis*
by *simp*
next
case *False*
with *ex* [*of m*] *fin* **have** *n* ∗ *Max* {*k*. *k* ∗ *n* ≤ *m*} = *Max* (*times n* ‘ {*k*. *k* ∗ *n* ≤ *m*})
by (*auto simp add*: *nat-mult-max-right intro*: *hom-Max-commute*)
also have *times n* ‘ {*k*. *k* ∗ *n* ≤ *m*} = {*l*. *l* ≤ *m* ∧ *n dvd l*}
by (*auto simp add*: *ac-simps elim*!: *dvdE*)
finally show *?thesis*
using *False* **by** (*simp add*: *divide-nat-def ac-simps*)
qed
have *less-eq*: *m div n* ∗ *n* ≤ *m*
by (*auto simp add*: *mult-div-unfold ac-simps intro*: *Max.boundedI*)
then show *m div n* ∗ *n* + *m mod n* = *m*
by (*simp add*: *modulo-nat-def*)
assume *n* ≠ *0*
show *euclidean-size* (*m mod n*) < *euclidean-size n*
proof −
have *m* < *Suc* (*m div n*) ∗ *n*
proof (*rule ccontr*)
assume ¬ *m* < *Suc* (*m div n*) ∗ *n*
then have *Suc* (*m div n*) ∗ *n* ≤ *m*
by (*simp add*: *not-less*)
moreover from ‹*n* ≠ *0*› **have** *Max* {*k*. *k* ∗ *n* ≤ *m*} < *Suc* (*m div n*)

```
      by (simp add: divide-nat-def)
    with ‹n ≠ 0› ex fin have ⋀k. k * n ≤ m ⟹ k < Suc (m div n)
      by auto
    ultimately have Suc (m div n) < Suc (m div n)
      by blast
    then show False
      by simp
  qed
  with ‹n ≠ 0› show ?thesis
    by (simp add: modulo-nat-def)
 qed
 show euclidean-size m ≤ euclidean-size (m * n)
  using ‹n ≠ 0› by (cases n) simp-all
 fix q r :: nat
 show (q * n + r) div n = q if euclidean-size r < euclidean-size n
 proof −
  from that have r < n
    by simp
  have k ≤ q if k * n ≤ q * n + r for k
  proof (rule ccontr)
    assume ¬ k ≤ q
    then have q < k
      by simp
    then obtain l where k = Suc (q + l)
      by (auto simp add: less-iff-Suc-add)
    with ‹r < n› that show False
      by (simp add: algebra-simps)
  qed
  with ‹n ≠ 0› ex fin show ?thesis
    by (auto simp add: divide-nat-def Max-eq-iff)
 qed
qed simp-all

end

lemma euclidean-relation-natI [case-names by0 divides euclidean-relation]:
 ‹(m div n, m mod n) = (q, r)›
  if by0: ‹n = 0 ⟹ q = 0 ∧ r = m›
  and divides: ‹n > 0 ⟹ n dvd m ⟹ r = 0 ∧ m = q * n›
  and euclidean-relation: ‹n > 0 ⟹ ¬ n dvd m ⟹ r < n ∧ m = q * n + r›
for m n q r :: nat
 by (rule euclidean-relationI) (use that in simp-all)

lemma div-nat-eqI:
 ‹m div n = q› if ‹n * q ≤ m› and ‹m < n * Suc q› for m n q :: nat
proof −
 have ‹(m div n, m mod n) = (q, m − n * q)›
 proof (induction rule: euclidean-relation-natI)
  case by0
```

```
    with that show ?case
      by simp
  next
    case divides
    from ‹n dvd m› obtain s where ‹m = n * s› ..
    with ‹n > 0› that have ‹s < Suc q›
      by (simp only: mult-less-cancel1)
    with ‹m = n * s› ‹n > 0› that have ‹q = s›
      by simp
    with ‹m = n * s› show ?case
      by (simp add: ac-simps)
  next
    case euclidean-relation
    with that show ?case
      by (simp add: ac-simps)
  qed
  then show ?thesis
    by simp
qed

lemma mod-nat-eqI:
  ‹m mod n = r› if ‹r < n› and ‹r ≤ m› and ‹n dvd m - r› for m n r :: nat
proof -
  have ‹(m div n, m mod n) = ((m - r) div n, r)›
  proof (induction rule: euclidean-relation-natI)
    case by0
    with that show ?case
      by simp
  next
    case divides
    from that dvd-minus-add [of r ‹m› 1 n]
    have ‹n dvd m + (n - r)›
      by simp
    with divides have ‹n dvd n - r›
      by (simp add: dvd-add-right-iff)
    then have ‹n ≤ n - r›
      by (rule dvd-imp-le) (use ‹r < n› in simp)
    with ‹n > 0› have ‹r = 0›
      by simp
    with ‹n > 0› that show ?case
      by simp
  next
    case euclidean-relation
    with that show ?case
      by (simp add: ac-simps)
  qed
  then show ?thesis
    by simp
qed
```

Tool support

```
ML ‹
structure Cancel-Div-Mod-Nat = Cancel-Div-Mod
(
  val div-name = const-name ‹divide›;
  val mod-name = const-name ‹modulo›;
  val mk-binop = HOLogic.mk-binop;
  val dest-plus = HOLogic.dest-bin const-name ‹Groups.plus› HOLogic.natT;
  val mk-sum = Arith-Data.mk-sum;
  fun dest-sum tm =
    if HOLogic.is-zero tm then []
    else
      (case try HOLogic.dest-Suc tm of
        SOME t => HOLogic.Suc-zero :: dest-sum t
      | NONE =>
          (case try dest-plus tm of
            SOME (t, u) => dest-sum t @ dest-sum u
          | NONE => [tm]));

  val div-mod-eqs = map mk-meta-eq @{thms cancel-div-mod-rules};

  val prove-eq-sums = Arith-Data.prove-conv2 all-tac
    (Arith-Data.simp-all-tac @{thms add-0-left add-0-right ac-simps})
)
›

simproc-setup cancel-div-mod-nat ((m::nat) + n) =
  ‹K Cancel-Div-Mod-Nat.proc›

lemma div-mult-self-is-m [simp]:
  m * n div n = m if n > 0 for m n :: nat
  using that by simp

lemma div-mult-self1-is-m [simp]:
  n * m div n = m if n > 0 for m n :: nat
  using that by simp

lemma mod-less-divisor [simp]:
  m mod n < n if n > 0 for m n :: nat
  using mod-size-less [of n m] that by simp

lemma mod-le-divisor [simp]:
  m mod n ≤ n if n > 0 for m n :: nat
  using that by (auto simp add: le-less)

lemma div-times-less-eq-dividend [simp]:
  m div n * n ≤ m for m n :: nat
  by (simp add: minus-mod-eq-div-mult [symmetric])
```

lemma *times-div-less-eq-dividend* [*simp*]:
  $n * (m \ div \ n) \leq m$ **for** $m \ n :: nat$
  using *div-times-less-eq-dividend* [*of m n*]
  by (*simp add*: *ac-simps*)

lemma *dividend-less-div-times*:
  $m < n + (m \ div \ n) * n$ **if** $0 < n$ **for** $m \ n :: nat$
proof −
  from *that* **have** $m \ mod \ n < n$
    by *simp*
  then show *?thesis*
    by (*simp add*: *minus-mod-eq-div-mult* [*symmetric*])
qed

lemma *dividend-less-times-div*:
  $m < n + n * (m \ div \ n)$ **if** $0 < n$ **for** $m \ n :: nat$
  using *dividend-less-div-times* [*of n m*] *that*
  by (*simp add*: *ac-simps*)

lemma *mod-Suc-le-divisor* [*simp*]:
  $m \ mod \ Suc \ n \leq n$
  using *mod-less-divisor* [*of Suc n m*] **by** *arith*

lemma *mod-less-eq-dividend* [*simp*]:
  $m \ mod \ n \leq m$ **for** $m \ n :: nat$
proof (*rule add-leD2*)
  from *div-mult-mod-eq* **have** $m \ div \ n * n + m \ mod \ n = m$ **.**
  then show $m \ div \ n * n + m \ mod \ n \leq m$ **by** *auto*
qed

lemma
  div-less [*simp*]: $m \ div \ n = 0$
  and *mod-less* [*simp*]: $m \ mod \ n = m$
  if $m < n$ **for** $m \ n :: nat$
  using *that* **by** (*auto intro*: *div-nat-eqI mod-nat-eqI*)

lemma *split-div*:
  ‹$P \ (m \ div \ n) \longleftrightarrow$
    $(n = 0 \longrightarrow P \ 0) \wedge$
    $(n \neq 0 \longrightarrow (\forall i \ j. \ j < n \wedge m = n * i + j \longrightarrow P \ i))$› (**is** *?div*)
  and *split-mod*:
  ‹$Q \ (m \ mod \ n) \longleftrightarrow$
    $(n = 0 \longrightarrow Q \ m) \wedge$
    $(n \neq 0 \longrightarrow (\forall i \ j. \ j < n \wedge m = n * i + j \longrightarrow Q \ j))$› (**is** *?mod*)
  for $m \ n :: nat$
proof −
  have ∗: ‹$R \ (m \ div \ n) \ (m \ mod \ n) \longleftrightarrow$
    $(n = 0 \longrightarrow R \ 0 \ m) \wedge$
    $(n \neq 0 \longrightarrow (\forall i \ j. \ j < n \wedge m = n * i + j \longrightarrow R \ i \ j))$› **for** R

```
    by (cases ‹n = 0›) auto
  from * [of ‹λq -. P q›] show ?div .
  from * [of ‹λ- r. Q r›] show ?mod .
qed

declare split-div [of - - ‹numeral n›, linarith-split] for n
declare split-mod [of - - ‹numeral n›, linarith-split] for n

lemma split-div':
  P (m div n) ⟷ n = 0 ∧ P 0 ∨ (∃ q. (n * q ≤ m ∧ m < n * Suc q) ∧ P q)
proof (cases n = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  then have n * q ≤ m ∧ m < n * Suc q ⟷ m div n = q for q
    by (auto intro: div-nat-eqI dividend-less-times-div)
  then show ?thesis
    by auto
qed

lemma le-div-geq:
  m div n = Suc ((m − n) div n) if 0 < n and n ≤ m for m n :: nat
proof −
  from ‹n ≤ m› obtain q where m = n + q
    by (auto simp add: le-iff-add)
  with ‹0 < n› show ?thesis
    by (simp add: div-add-self1)
qed

lemma le-mod-geq:
  m mod n = (m − n) mod n if n ≤ m for m n :: nat
proof −
  from ‹n ≤ m› obtain q where m = n + q
    by (auto simp add: le-iff-add)
  then show ?thesis
    by simp
qed

lemma div-if:
  m div n = (if m < n ∨ n = 0 then 0 else Suc ((m − n) div n))
  by (simp add: le-div-geq)

lemma mod-if:
  m mod n = (if m < n then m else (m − n) mod n) for m n :: nat
  by (simp add: le-mod-geq)

lemma div-eq-0-iff:
```

m div n = 0 ⟷ m < n ∨ n = 0 **for** *m n :: nat*
by (*simp add*: *div-eq-0-iff*)

lemma *div-greater-zero-iff*:
m div n > 0 ⟷ n ≤ m ∧ n > 0 **for** *m n :: nat*
using *div-eq-0-iff* [*of m n*] **by** *auto*

lemma *mod-greater-zero-iff-not-dvd*:
m mod n > 0 ⟷ ¬ n dvd m **for** *m n :: nat*
by (*simp add*: *dvd-eq-mod-eq-0*)

lemma *dvd-pos-nat*: *n > 0 ⟹ m dvd n ⟹ m > 0*
for *m n :: nat*
by *auto*

lemma *dvd-div-ge-1*:
fixes *a b :: nat*
assumes *a ≥ 1 b dvd a*
shows *a div b ≥ 1*
proof −
from ‹*b dvd a*› **obtain** *c* **where** *a = b * c* **..**
with ‹*a ≥ 1*› **show** *?thesis* **by** *simp*
qed

lemma *dvd-nat-bounds*:
fixes *n p :: nat*
assumes *p > 0 n dvd p*
shows *n > 0 ∧ n ≤ p*
using *assms* **by** (*simp add*: *dvd-pos-nat dvd-imp-le*)

lemma *div-by-Suc-0* [*simp*]:
m div Suc 0 = m
using *div-by-1* [*of m*] **by** *simp*

lemma *mod-by-Suc-0* [*simp*]:
m mod Suc 0 = 0
using *mod-by-1* [*of m*] **by** *simp*

lemma *div2-Suc-Suc* [*simp*]:
Suc (*Suc m*) *div 2 = Suc* (*m div 2*)
by (*simp add*: *numeral-2-eq-2 le-div-geq*)

lemma *Suc-n-div-2-gt-zero* [*simp*]:
0 < Suc n div 2 **if** *n > 0* **for** *n :: nat*
using *that* **by** (*cases n*) *simp-all*

lemma *div-2-gt-zero* [*simp*]:
0 < n div 2 **if** *Suc 0 < n* **for** *n :: nat*
using *that Suc-n-div-2-gt-zero* [*of n − 1*] **by** *simp*

lemma *mod2-Suc-Suc* [*simp*]:
Suc (*Suc m*) *mod* 2 = *m mod* 2
by (*simp add*: *numeral-2-eq-2 le-mod-geq*)

lemma *add-self-div-2* [*simp*]:
(*m* + *m*) *div* 2 = *m* **for** *m* :: *nat*
by (*simp add*: *mult-2* [*symmetric*])

lemma *add-self-mod-2* [*simp*]:
(*m* + *m*) *mod* 2 = 0 **for** *m* :: *nat*
by (*simp add*: *mult-2* [*symmetric*])

lemma *mod2-gr-0* [*simp*]:
0 < *m mod* 2 ⟷ *m mod* 2 = 1 **for** *m* :: *nat*
proof −
have *m mod* 2 < 2
by (*rule mod-less-divisor*) *simp*
then have *m mod* 2 = 0 ∨ *m mod* 2 = 1
by *arith*
then show *?thesis*
by *auto*
qed

lemma *mod-Suc-eq* [*mod-simps*]:
Suc (*m mod n*) *mod n* = *Suc m mod n*
proof −
have (*m mod n* + 1) *mod n* = (*m* + 1) *mod n*
by (*simp only*: *mod-simps*)
then show *?thesis*
by *simp*
qed

lemma *mod-Suc-Suc-eq* [*mod-simps*]:
Suc (*Suc* (*m mod n*)) *mod n* = *Suc* (*Suc m*) *mod n*
proof −
have (*m mod n* + 2) *mod n* = (*m* + 2) *mod n*
by (*simp only*: *mod-simps*)
then show *?thesis*
by *simp*
qed

lemma
Suc-mod-mult-self1 [*simp*]: *Suc* (*m* + *k* ∗ *n*) *mod n* = *Suc m mod n*
and *Suc-mod-mult-self2* [*simp*]: *Suc* (*m* + *n* ∗ *k*) *mod n* = *Suc m mod n*
and *Suc-mod-mult-self3* [*simp*]: *Suc* (*k* ∗ *n* + *m*) *mod n* = *Suc m mod n*
and *Suc-mod-mult-self4* [*simp*]: *Suc* (*n* ∗ *k* + *m*) *mod n* = *Suc m mod n*
by (*subst mod-Suc-eq* [*symmetric*], *simp add*: *mod-simps*)+

lemma *Suc-0-mod-eq* [*simp*]:
Suc 0 mod n = of-bool (n ≠ Suc 0)
by (*cases n*) *simp-all*

lemma *div-mult2-eq*:
‹*m div (n * q) = (m div n) div q*› (**is** *?Q*)
and *mod-mult2-eq*:
‹*m mod (n * q) = n * (m div n mod q) + m mod n*› (**is** *?R*)
for *m n q* :: *nat*
proof −
have ‹(*m div (n * q)*, *m mod (n * q)*) = ((*m div n*) *div q*, *n * (m div n mod q) + m mod n*)›
proof (*induction rule*: *euclidean-relation-natI*)
case *by0*
then show *?case*
by *auto*
next
case *divides*
from ‹*n * q dvd m*› **obtain** *t* **where** ‹*m = n * q * t*› **..**
with ‹*n * q > 0*› **show** *?case*
by (*simp add*: *algebra-simps*)
next
case *euclidean-relation*
then have ‹*n > 0*› ‹*q > 0*›
by *simp-all*
from ‹*n > 0*› **have** ‹*m mod n < n*›
by (*rule mod-less-divisor*)
from ‹*q > 0*› **have** ‹*m div n mod q < q*›
by (*rule mod-less-divisor*)
then obtain *s* **where** ‹*q = Suc (m div n mod q + s)*›
by (*blast dest*: *less-imp-Suc-add*)
moreover have ‹*m mod n + n * (m div n mod q) < n * Suc (m div n mod q + s)*›
using ‹*m mod n < n*› **by** (*simp add*: *add-mult-distrib2*)
ultimately have ‹*m mod n + n * (m div n mod q) < n * q*›
by *simp*
then show *?case*
by (*simp add*: *algebra-simps flip*: *add-mult-distrib2*)
qed
then show *?Q* **and** *?R*
by *simp-all*
qed

lemma *div-le-mono*:
m div k ≤ n div k **if** *m ≤ n* **for** *m n k* :: *nat*
proof −
from *that* **obtain** *q* **where** *n = m + q*
by (*auto simp add*: *le-iff-add*)
then show *?thesis*

```
    by (simp add: div-add1-eq [of m q k])
qed
```

Antimonotonicity of (*div*) in second argument

```
lemma div-le-mono2:
  k div n ≤ k div m if 0 < m and m ≤ n for m n k :: nat
using that proof (induct k arbitrary: m rule: less-induct)
  case (less k)
  show ?case
  proof (cases n ≤ k)
    case False
    then show ?thesis
      by simp
  next
    case True
    have (k − n) div n ≤ (k − m) div n
      using less.prems
      by (blast intro: div-le-mono diff-le-mono2)
    also have ... ≤ (k − m) div m
      using ‹n ≤ k› less.prems less.hyps [of k − m m]
      by simp
    finally show ?thesis
      using ‹n ≤ k› less.prems
      by (simp add: le-div-geq)
  qed
qed

lemma div-le-dividend [simp]:
  m div n ≤ m for m n :: nat
  using div-le-mono2 [of 1 n m] by (cases n = 0) simp-all

lemma div-less-dividend [simp]:
  m div n < m if 1 < n and 0 < m for m n :: nat
using that proof (induct m rule: less-induct)
  case (less m)
  show ?case
  proof (cases n < m)
    case False
    with less show ?thesis
      by (cases n = m) simp-all
  next
    case True
    then show ?thesis
      using less.hyps [of m − n] less.prems
      by (simp add: le-div-geq)
  qed
qed

lemma div-eq-dividend-iff:
```

```
  m div n = m ⟷ n = 1 if m > 0 for m n :: nat
proof
  assume n = 1
  then show m div n = m
    by simp
next
  assume P: m div n = m
  show n = 1
  proof (rule ccontr)
    have n ≠ 0
      by (rule ccontr) (use that P in auto)
    moreover assume n ≠ 1
    ultimately have n > 1
      by simp
    with that have m div n < m
      by simp
    with P show False
      by simp
  qed
qed

lemma less-mult-imp-div-less:
  m div n < i if m < i * n for m n i :: nat
proof −
  from that have i * n > 0
    by (cases i * n = 0) simp-all
  then have i > 0 and n > 0
    by simp-all
  have m div n * n ≤ m
    by simp
  then have m div n * n < i * n
    using that by (rule le-less-trans)
  with ‹n > 0› show ?thesis
    by simp
qed

lemma div-less-iff-less-mult:
  ‹m div q < n ⟷ m < n * q› (is ‹?P ⟷ ?Q›)
  if ‹q > 0› for m n q :: nat
proof
  assume ?Q then show ?P
    by (rule less-mult-imp-div-less)
next
  assume ?P
  then obtain h where ‹n = Suc (m div q + h)›
    using less-natE by blast
  moreover have ‹m < m + (Suc h * q − m mod q)›
    using that by (simp add: trans-less-add1)
  ultimately show ?Q
```

```
    by (simp add: algebra-simps flip: minus-mod-eq-mult-div)
qed

lemma less-eq-div-iff-mult-less-eq:
  ‹m ≤ n div q ⟷ m * q ≤ n› if ‹q > 0› for m n q :: nat
  using div-less-iff-less-mult [of q n m] that by auto

lemma div-Suc:
  ‹Suc m div n = (if Suc m mod n = 0 then Suc (m div n) else m div n)›
proof (cases ‹n = 0 ∨ n = 1›)
  case True
  then show ?thesis by auto
next
  case False
  then have ‹n > 1›
    by simp
  then have ‹Suc m div n = m div n + Suc (m mod n) div n›
    using div-add1-eq [of m 1 n] by simp
  also have ‹Suc (m mod n) div n = of-bool (n dvd Suc m)›
  proof (cases ‹n dvd Suc m›)
    case False
    moreover have ‹Suc (m mod n) ≠ n›
    proof (rule ccontr)
      assume ‹¬ Suc (m mod n) ≠ n›
      then have ‹m mod n = n - Suc 0›
        by simp
      with ‹n > 1› have ‹(m + 1) mod n = 0›
        by (subst mod-add-left-eq [symmetric]) simp
      then have ‹n dvd Suc m›
        by auto
      with False show False ..
    qed
    moreover have ‹Suc (m mod n) ≤ n›
      using ‹n > 1› by (simp add: Suc-le-eq)
    ultimately show ?thesis
      by (simp add: div-eq-0-iff)
  next
    case True
    then obtain q where q: ‹Suc m = n * q› ..
    moreover have ‹q > 0› by (rule ccontr)
      (use q in simp)
    ultimately have ‹m mod n = n - Suc 0›
      using ‹n > 1› mult-le-cancel1 [of n ‹Suc 0› q]
      by (auto intro: mod-nat-eqI)
    with True ‹n > 1› show ?thesis
      by simp
  qed
  finally show ?thesis
    by (simp add: mod-greater-zero-iff-not-dvd)
```

qed

lemma *mod-Suc*:
  ‹*Suc m mod n* = (*if Suc* (*m mod n*) = *n then 0 else Suc* (*m mod n*))›
proof (*cases* ‹*n* = *0*›)
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  moreover have ‹*Suc m mod n* = *Suc* (*m mod n*) *mod n*›
    by (*simp add*: *mod-simps*)
  ultimately show *?thesis*
    by (*auto intro*!: *mod-nat-eqI intro*: *neq-le-trans simp add*: *Suc-le-eq*)
qed

lemma *Suc-times-mod-eq*:
  Suc (*m* ∗ *n*) *mod m* = *1* **if** *Suc 0* < *m*
  using *that* **by** (*simp add*: *mod-Suc*)

lemma *Suc-times-numeral-mod-eq* [*simp*]:
  Suc (*numeral k* ∗ *n*) *mod numeral k* = *1* **if** *numeral k* ≠ (*1*::*nat*)
  by (*rule Suc-times-mod-eq*) (*use that* **in** *simp*)

lemma *Suc-div-le-mono* [*simp*]:
  m div n ≤ *Suc m div n*
  by (*simp add*: *div-le-mono*)

These lemmas collapse some needless occurrences of Suc: at least three Sucs, since two and fewer are rewritten back to Suc again! We already have some rules to simplify operands smaller than 3.

lemma *div-Suc-eq-div-add3* [*simp*]:
  m div Suc (*Suc* (*Suc n*)) = *m div* (*3* + *n*)
  by (*simp add*: *Suc3-eq-add-3*)

lemma *mod-Suc-eq-mod-add3* [*simp*]:
  m mod Suc (*Suc* (*Suc n*)) = *m mod* (*3* + *n*)
  by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-div-eq-add3-div*:
  Suc (*Suc* (*Suc m*)) *div n* = (*3* + *m*) *div n*
  by (*simp add*: *Suc3-eq-add-3*)

lemma *Suc-mod-eq-add3-mod*:
  Suc (*Suc* (*Suc m*)) *mod n* = (*3* + *m*) *mod n*
  by (*simp add*: *Suc3-eq-add-3*)

lemmas *Suc-div-eq-add3-div-numeral* [*simp*] =
  Suc-div-eq-add3-div [*of* - *numeral v*] **for** *v*

lemmas *Suc-mod-eq-add3-mod-numeral* [*simp*] =
Suc-mod-eq-add3-mod [*of - numeral v*] **for** *v*

lemma (**in** *field-char-0*) *of-nat-div*:
of-nat (*m div n*) = ((*of-nat m* − *of-nat* (*m mod n*)) / *of-nat n*)
proof −
have *of-nat* (*m div n*) = ((*of-nat* (*m div n* ∗ *n* + *m mod n*) − *of-nat* (*m mod n*)) / *of-nat n* :: ′*a*)
unfolding *of-nat-add* **by** (*cases n* = *0*) *simp-all*
then show *?thesis*
by *simp*
qed

An "induction" law for modulus arithmetic.

lemma *mod-induct* [*consumes 3*, *case-names step*]:
P m **if** *P n* **and** *n* < *p* **and** *m* < *p*
and *step*: ⋀*n*. *n* < *p* ⟹ *P n* ⟹ *P* (*Suc n mod p*)
using ‹*m* < *p*› **proof** (*induct m*)
case *0*
show *?case*
proof (*rule ccontr*)
assume ¬ *P 0*
from ‹*n* < *p*› **have** *0* < *p*
by *simp*
from ‹*n* < *p*› **obtain** *m* **where** *0* < *m* **and** *p* = *n* + *m*
by (*blast dest*: *less-imp-add-positive*)
with ‹*P n*› **have** *P* (*p* − *m*)
by *simp*
moreover have ¬ *P* (*p* − *m*)
using ‹*0* < *m*› **proof** (*induct m*)
case *0*
then show *?case*
by *simp*
next
case (*Suc m*)
show *?case*
proof
assume *P*: *P* (*p* − *Suc m*)
with ‹¬ *P 0*› **have** *Suc m* < *p*
by (*auto intro*: *ccontr*)
then have *Suc* (*p* − *Suc m*) = *p* − *m*
by *arith*
moreover from ‹*0* < *p*› **have** *p* − *Suc m* < *p*
by *arith*
with *P step* **have** *P* ((*Suc* (*p* − *Suc m*)) *mod p*)
by *blast*
ultimately show *False*
using ‹¬ *P 0*› *Suc.hyps* **by** (*cases m* = *0*) *simp-all*

```
      qed
    qed
    ultimately show False
      by blast
  qed
next
  case (Suc m)
  then have m < p and mod: Suc m mod p = Suc m
    by simp-all
  from ‹m < p› have P m
    by (rule Suc.hyps)
  with ‹m < p› have P (Suc m mod p)
    by (rule step)
  with mod show ?case
    by simp
qed

lemma funpow-mod-eq:
  ‹(f ^^ (m mod n)) x = (f ^^ m) x› if ‹(f ^^ n) x = x›
proof −
  have ‹(f ^^ m) x = (f ^^ (m mod n + m div n * n)) x›
    by simp
  also have ‹... = (f ^^ (m mod n)) (((f ^^ n) ^^ (m div n)) x)›
    by (simp only: funpow-add funpow-mult ac-simps) simp
  also have ‹((f ^^ n) ^^ q) x = x› for q
    by (induction q) (use ‹(f ^^ n) x = x› in simp-all)
  finally show ?thesis
    by simp
qed

lemma mod-eq-dvd-iff-nat:
  ‹m mod q = n mod q ⟷ q dvd m − n› (is ‹?P ⟷ ?Q›)
    if ‹m ≥ n› for m n q :: nat
proof
  assume ?Q
  then obtain s where ‹m − n = q * s› ..
  with that have ‹m = q * s + n›
    by simp
  then show ?P
    by simp
next
  assume ?P
  have ‹m − n = m div q * q + m mod q − (n div q * q + n mod q)›
    by simp
  also have ‹... = q * (m div q − n div q)›
    by (simp only: algebra-simps ‹?P›)
  finally show ?Q ..
qed
```

lemma *mod-eq-iff-dvd-symdiff-nat*:
‹*m mod q* = *n mod q* ⟷ *q dvd nat* |*int m* − *int n*|›
by (*auto simp add*: *abs-if mod-eq-dvd-iff-nat nat-diff-distrib dest*: *sym intro*: *sym*)

lemma *mod-eq-nat1E*:
fixes *m n q* :: *nat*
assumes *m mod q* = *n mod q* **and** *m* ≥ *n*
obtains *s* **where** *m* = *n* + *q* ∗ *s*
proof −
from *assms* **have** *q dvd m* − *n*
by (*simp add*: *mod-eq-dvd-iff-nat*)
then obtain *s* **where** *m* − *n* = *q* ∗ *s* **..**
with ‹*m* ≥ *n*› **have** *m* = *n* + *q* ∗ *s*
by *simp*
with *that* **show** *thesis* **.**
qed

lemma *mod-eq-nat2E*:
fixes *m n q* :: *nat*
assumes *m mod q* = *n mod q* **and** *n* ≥ *m*
obtains *s* **where** *n* = *m* + *q* ∗ *s*
using *assms mod-eq-nat1E* [*of n q m*] **by** (*auto simp add*: *ac-simps*)

lemma *nat-mod-eq-iff*:
(*x*::*nat*) *mod n* = *y mod n* ⟷ (∃ *q1 q2*. *x* + *n* ∗ *q1* = *y* + *n* ∗ *q2*) (**is** *?lhs* = *?rhs*)
proof
assume *H*: *x mod n* = *y mod n*
{ assume *xy*: *x* ≤ *y*
from *H* **have** *th*: *y mod n* = *x mod n* **by** *simp*
from *mod-eq-nat1E* [*OF th xy*] **obtain** *q* **where** *y* = *x* + *n* ∗ *q* **.**
then have *x* + *n* ∗ *q* = *y* + *n* ∗ *0*
by *simp*
then have ∃ *q1 q2*. *x* + *n* ∗ *q1* = *y* + *n* ∗ *q2*
by *blast*
}
moreover
{ assume *xy*: *y* ≤ *x*
from *mod-eq-nat1E* [*OF H xy*] **obtain** *q* **where** *x* = *y* + *n* ∗ *q* **.**
then have *x* + *n* ∗ *0* = *y* + *n* ∗ *q*
by *simp*
then have ∃ *q1 q2*. *x* + *n* ∗ *q1* = *y* + *n* ∗ *q2*
by *blast*
}
ultimately show *?rhs* **using** *linear*[*of x y*] **by** *blast*
next
assume *?rhs* **then obtain** *q1 q2* **where** *q12*: *x* + *n* ∗ *q1* = *y* + *n* ∗ *q2* **by** *blast*
hence (*x* + *n* ∗ *q1*) *mod n* = (*y* + *n* ∗ *q2*) *mod n* **by** *simp*
thus *?lhs* **by** *simp*

qed

56.5 Division on *int*

The following specification of integer division rounds towards minus infinity and is advocated by Donald Knuth. See [5] for an overview and terminology of different possibilities to specify integer division; there division rounding towards minus infinitiy is named "F-division".

56.5.1 Basic instantiation

instantiation *int* :: {*normalization-semidom*, *idom-modulo*}
begin

definition *normalize-int* :: ‹*int* ⇒ *int*›
  where [*simp*]: ‹*normalize* = (*abs* :: *int* ⇒ *int*)›

definition *unit-factor-int* :: ‹*int* ⇒ *int*›
  where [*simp*]: ‹*unit-factor* = (*sgn* :: *int* ⇒ *int*)›

definition *divide-int* :: ‹*int* ⇒ *int* ⇒ *int*›
  where ‹*k div l* = (*sgn k* ∗ *sgn l* ∗ *int* (*nat* |*k*| *div nat* |*l*|)
    − *of-bool* (*l* ≠ *0* ∧ *sgn k* ≠ *sgn l* ∧ ¬ *l dvd k*))›

lemma *divide-int-unfold*:
  ‹(*sgn k* ∗ *int m*) *div* (*sgn l* ∗ *int n*) = (*sgn k* ∗ *sgn l* ∗ *int* (*m div n*)
    − *of-bool* ((*k* = *0* ⟷ *m* = *0*) ∧ *l* ≠ *0* ∧ *n* ≠ *0* ∧ *sgn k* ≠ *sgn l* ∧ ¬ *n dvd m*))›
  by (*simp add*: *divide-int-def sgn-mult nat-mult-distrib abs-mult sgn-eq-0-iff ac-simps*)

definition *modulo-int* :: ‹*int* ⇒ *int* ⇒ *int*›
  where ‹*k mod l* = *sgn k* ∗ *int* (*nat* |*k*| *mod nat* |*l*|) + *l* ∗ *of-bool* (*sgn k* ≠ *sgn l* ∧ ¬ *l dvd k*)›

lemma *modulo-int-unfold*:
  ‹(*sgn k* ∗ *int m*) *mod* (*sgn l* ∗ *int n*) =
    sgn k ∗ *int* (*m mod* (*of-bool* (*l* ≠ *0*) ∗ *n*)) + (*sgn l* ∗ *int n*) ∗ *of-bool* ((*k* = *0* ⟷ *m* = *0*) ∧ *sgn k* ≠ *sgn l* ∧ ¬ *n dvd m*)›
  by (*auto simp add*: *modulo-int-def sgn-mult abs-mult*)

instance proof
  fix *k* :: *int* **show** *k div 0* = *0*
  by (*simp add*: *divide-int-def*)
next
  fix *k l* :: *int*
  assume *l* ≠ *0*
  obtain *n m* **and** *s t* **where** *k*: *k* = *sgn s* ∗ *int n* **and** *l*: *l* = *sgn t* ∗ *int m*
    by (*blast intro*: *int-sgnE elim*: *that*)
  then have *k* ∗ *l* = *sgn* (*s* ∗ *t*) ∗ *int* (*n* ∗ *m*)

by (*simp add*: *ac-simps sgn-mult*)
with *k l* ‹*l* ≠ *0*› **show** *k* ∗ *l div l* = *k*
by (*simp only*: *divide-int-unfold*)
(*auto simp add*: *algebra-simps sgn-mult sgn-1-pos sgn-0-0*)
next
fix *k l* :: *int*
obtain *n m* **and** *s t* **where** *k* = *sgn s* ∗ *int n* **and** *l* = *sgn t* ∗ *int m*
by (*blast intro*: *int-sgnE elim*: *that*)
then show *k div l* ∗ *l* + *k mod l* = *k*
by (*simp add*: *divide-int-unfold modulo-int-unfold algebra-simps modulo-nat-def of-nat-diff*)
qed (*auto simp add*: *sgn-mult mult-sgn-abs abs-eq-iff'*)

end

lemma *of-int-div*: *b dvd a* ⟹ *of-int* (*a div b*) = (*of-int a* / *of-int b* :: *'a* :: *field-char-0*)
by (*elim dvdE*) (*auto simp*: *divide-simps mult-ac*)

56.5.2 Algebraic foundations

lemma *coprime-int-iff* [*simp*]:
coprime (*int m*) (*int n*) ⟷ *coprime m n* (**is** *?P* ⟷ *?Q*)
proof
assume *?P*
show *?Q*
proof (*rule coprimeI*)
fix *q*
assume *q dvd m q dvd n*
then have *int q dvd int m int q dvd int n*
by *simp-all*
with ‹*?P*› **have** *is-unit* (*int q*)
by (*rule coprime-common-divisor*)
then show *is-unit q*
by *simp*
qed
next
assume *?Q*
show *?P*
proof (*rule coprimeI*)
fix *k*
assume *k dvd int m k dvd int n*
then have *nat* |*k*| *dvd m nat* |*k*| *dvd n*
by *simp-all*
with ‹*?Q*› **have** *is-unit* (*nat* |*k*|)
by (*rule coprime-common-divisor*)
then show *is-unit k*
by *simp*
qed

qed

lemma *coprime-abs-left-iff* [*simp*]:
coprime |*k*| *l* ⟷ *coprime k l* **for** *k l* :: *int*
using *coprime-normalize-left-iff* [*of k l*] **by** *simp*

lemma *coprime-abs-right-iff* [*simp*]:
coprime k |*l*| ⟷ *coprime k l* **for** *k l* :: *int*
using *coprime-abs-left-iff* [*of l k*] **by** (*simp add*: *ac-simps*)

lemma *coprime-nat-abs-left-iff* [*simp*]:
coprime (*nat* |*k*|) *n* ⟷ *coprime k* (*int n*)
proof −
define *m* **where** *m* = *nat* |*k*|
then have |*k*| = *int m*
by *simp*
moreover have *coprime k* (*int n*) ⟷ *coprime* |*k*| (*int n*)
by *simp*
ultimately show *?thesis*
by *simp*
qed

lemma *coprime-nat-abs-right-iff* [*simp*]:
coprime n (*nat* |*k*|) ⟷ *coprime* (*int n*) *k*
using *coprime-nat-abs-left-iff* [*of k n*] **by** (*simp add*: *ac-simps*)

lemma *coprime-common-divisor-int*: *coprime a b* ⟹ *x dvd a* ⟹ *x dvd b* ⟹ |*x*|
= *1*
for *a b* :: *int*
by (*drule coprime-common-divisor* [*of - - x*]) *simp-all*

56.5.3 Basic conversions

lemma *div-abs-eq-div-nat*:
|*k*| *div* |*l*| = *int* (*nat* |*k*| *div nat* |*l*|)
by (*auto simp add*: *divide-int-def*)

lemma *div-eq-div-abs*:
‹*k div l* = *sgn k* ∗ *sgn l* ∗ (|*k*| *div* |*l*|)
− *of-bool* (*l* ≠ *0* ∧ *sgn k* ≠ *sgn l* ∧ ¬ *l dvd k*)›
for *k l* :: *int*
by (*simp add*: *divide-int-def* [*of k l*] *div-abs-eq-div-nat*)

lemma *div-abs-eq*:
‹|*k*| *div* |*l*| = *sgn k* ∗ *sgn l* ∗ (*k div l* + *of-bool* (*sgn k* ≠ *sgn l* ∧ ¬ *l dvd k*))›
for *k l* :: *int*
by (*simp add*: *div-eq-div-abs* [*of k l*] *ac-simps*)

lemma *mod-abs-eq-div-nat*:

$|k|\ mod\ |l| = int\ (nat\ |k|\ mod\ nat\ |l|)$
by (*simp add*: *modulo-int-def*)

lemma *mod-eq-mod-abs*:
$\langle k\ mod\ l = sgn\ k * (|k|\ mod\ |l|) + l * of\text{-}bool\ (sgn\ k \neq sgn\ l \wedge \neg\ l\ dvd\ k)\rangle$
for $k\ l :: int$
by (*simp add*: *modulo-int-def* [*of k l*] *mod-abs-eq-div-nat*)

lemma *mod-abs-eq*:
$\langle |k|\ mod\ |l| = sgn\ k * (k\ mod\ l - l * of\text{-}bool\ (sgn\ k \neq sgn\ l \wedge \neg\ l\ dvd\ k))\rangle$
for $k\ l :: int$
by (*auto simp*: *mod-eq-mod-abs* [*of k l*])

lemma *div-sgn-abs-cancel*:
fixes $k\ l\ v :: int$
assumes $v \neq 0$
shows $(sgn\ v * |k|)\ div\ (sgn\ v * |l|) = |k|\ div\ |l|$
using *assms* **by** (*simp add*: *sgn-mult abs-mult sgn-0-0*
divide-int-def [*of sgn v * |k| sgn v * |l|*] *flip*: *div-abs-eq-div-nat*)

lemma *div-eq-sgn-abs*:
fixes $k\ l\ v :: int$
assumes $sgn\ k = sgn\ l$
shows $k\ div\ l = |k|\ div\ |l|$
using *assms* **by** (*auto simp add*: *div-abs-eq*)

lemma *div-dvd-sgn-abs*:
fixes $k\ l :: int$
assumes $l\ dvd\ k$
shows $k\ div\ l = (sgn\ k * sgn\ l) * (|k|\ div\ |l|)$
using *assms* **by** (*auto simp add*: *div-abs-eq ac-simps*)

lemma *div-noneq-sgn-abs*:
fixes $k\ l :: int$
assumes $l \neq 0$
assumes $sgn\ k \neq sgn\ l$
shows $k\ div\ l = -\ (|k|\ div\ |l|) - of\text{-}bool\ (\neg\ l\ dvd\ k)$
using *assms* **by** (*auto simp add*: *div-abs-eq ac-simps sgn-0-0 dest*!: *sgn-not-eq-imp*)

56.5.4 Euclidean division

instantiation *int* :: *unique-euclidean-ring*
begin

definition *euclidean-size-int* :: $int \Rightarrow nat$
where [*simp*]: *euclidean-size-int* $= (nat \circ abs :: int \Rightarrow nat)$

definition *division-segment-int* :: $int \Rightarrow int$
where *division-segment-int* $k = (if\ k \geq 0\ then\ 1\ else\ -\ 1)$

lemma *division-segment-eq-sgn*:
division-segment k = sgn k **if** *k ≠ 0* **for** *k :: int*
using *that* **by** (*simp add*: *division-segment-int-def*)

lemma *abs-division-segment* [*simp*]:
|division-segment k| = 1 **for** *k :: int*
by (*simp add*: *division-segment-int-def*)

lemma *abs-mod-less*:
|k mod l| < |l| **if** *l ≠ 0* **for** *k l :: int*
proof −
obtain *n m* **and** *s t* **where** *k = sgn s * int n* **and** *l = sgn t * int m*
by (*blast intro*: *int-sgnE elim*: *that*)
with *that* **show** *?thesis*
by (*auto simp add*: *modulo-int-unfold abs-mult mod-greater-zero-iff-not-dvd*
simp flip: *right-diff-distrib dest*!: *sgn-not-eq-imp*)
(*simp add*: *sgn-0-0*)
qed

lemma *sgn-mod*:
sgn (k mod l) = sgn l **if** *l ≠ 0* ¬ *l dvd k* **for** *k l :: int*
proof −
obtain *n m* **and** *s t* **where** *k = sgn s * int n* **and** *l = sgn t * int m*
by (*blast intro*: *int-sgnE elim*: *that*)
with *that* **show** *?thesis*
by (*auto simp add*: *modulo-int-unfold sgn-mult mod-greater-zero-iff-not-dvd*
simp flip: *right-diff-distrib dest*!: *sgn-not-eq-imp*)
qed

instance proof
fix *k l :: int*
show *division-segment (k mod l) = division-segment l* **if**
l ≠ 0 **and** ¬ *l dvd k*
using *that* **by** (*simp add*: *division-segment-eq-sgn dvd-eq-mod-eq-0 sgn-mod*)
next
fix *l q r :: int*
obtain *n m* **and** *s t*
where *l*: *l = sgn s * int n* **and** *q*: *q = sgn t * int m*
by (*blast intro*: *int-sgnE elim*: *that*)
assume ‹*l ≠ 0*›
with *l* **have** *s ≠ 0* **and** *n > 0*
by (*simp-all add*: *sgn-0-0*)
assume *division-segment r = division-segment l*
moreover have *r = sgn r * |r|*
by (*simp add*: *sgn-mult-abs*)
moreover define *u* **where** *u = nat |r|*
ultimately have *r = sgn l * int u*
using *division-segment-eq-sgn* ‹*l ≠ 0*› **by** (*cases r = 0*) *simp-all*

with *l* ‹*n* > *0*› **have** *r*: *r* = *sgn s* ∗ *int u*
by (*simp add*: *sgn-mult*)
assume *euclidean-size r* < *euclidean-size l*
with *l r* ‹*s* ≠ *0*› **have** *u* < *n*
by (*simp add*: *abs-mult*)
show (*q* ∗ *l* + *r*) *div l* = *q*
proof (*cases q* = *0* ∨ *r* = *0*)
case *True*
then show *?thesis*
proof
assume *q* = *0*
then show *?thesis*
using *l r* ‹*u* < *n*› **by** (*simp add*: *divide-int-unfold*)
next
assume *r* = *0*
from ‹*r* = *0*› **have** ∗: *q* ∗ *l* + *r* = *sgn* (*t* ∗ *s*) ∗ *int* (*n* ∗ *m*)
using *q l* **by** (*simp add*: *ac-simps sgn-mult*)
from ‹*s* ≠ *0*› ‹*n* > *0*› **show** *?thesis*
by (*simp only*: ∗, *simp only*: ∗ *q l divide-int-unfold*)
(*auto simp add*: *sgn-mult ac-simps*)
qed
next
case *False*
with *q r* **have** *t* ≠ *0* **and** *m* > *0* **and** *s* ≠ *0* **and** *u* > *0*
by (*simp-all add*: *sgn-0-0*)
moreover from ‹*0* < *m*› ‹*u* < *n*› **have** *u* ≤ *m* ∗ *n*
using *mult-le-less-imp-less* [*of 1 m u n*] **by** *simp*
ultimately have ∗: *q* ∗ *l* + *r* = *sgn* (*s* ∗ *t*)
∗ *int* (*if t* < *0 then m* ∗ *n* − *u else m* ∗ *n* + *u*)
using *l q r*
by (*simp add*: *sgn-mult algebra-simps of-nat-diff*)
have (*m* ∗ *n* − *u*) *div n* = *m* − *1* **if** *u* > *0*
using ‹*0* < *m*› ‹*u* < *n*› *that*
by (*auto intro*: *div-nat-eqI simp add*: *algebra-simps*)
moreover have *n dvd m* ∗ *n* − *u* ⟷ *n dvd u*
using ‹*u* ≤ *m* ∗ *n*› *dvd-diffD1* [*of n m* ∗ *n u*]
by *auto*
ultimately show *?thesis*
using ‹*s* ≠ *0*› ‹*m* > *0*› ‹*u* > *0*› ‹*u* < *n*› ‹*u* ≤ *m* ∗ *n*›
by (*simp only*: ∗, *simp only*: *l q divide-int-unfold*)
(*auto simp add*: *sgn-mult sgn-0-0 sgn-1-pos algebra-simps dest*: *dvd-imp-le*)
qed
qed (*use mult-le-mono2* [*of 1*] **in** ‹*auto simp add*: *division-segment-int-def not-le zero-less-mult-iff mult-less-0-iff abs-mult sgn-mult abs-mod-less sgn-mod nat-mult-distrib*›)

end

lemma *euclidean-relation-intI* [*case-names by0 divides euclidean-relation*]:
‹(*k div l*, *k mod l*) = (*q*, *r*)›

if *by0'*: ‹$l = 0 \Longrightarrow q = 0 \wedge r = k$›
and *divides'*: ‹$l \neq 0 \Longrightarrow l$ *dvd* $k \Longrightarrow r = 0 \wedge k = q * l$›
and *euclidean-relation'*: ‹$l \neq 0 \Longrightarrow \neg\ l$ *dvd* $k \Longrightarrow$ *sgn* $r =$ *sgn* l
$\wedge\ |r| < |l| \wedge k = q * l + r$› **for** $k\ l$ :: *int*
proof (*induction rule*: *euclidean-relationI*)
case *by0*
then show *?case*
by (*rule by0'*)
next
case *divides*
then show *?case*
by (*rule divides'*)
next
case *euclidean-relation*
with *euclidean-relation'* **have** ‹*sgn* $r =$ *sgn* l› ‹$|r| < |l|$› ‹$k = q * l + r$›
by *simp-all*
from ‹*sgn* $r =$ *sgn* l› ‹$l \neq 0$› **have** ‹*division-segment* $r =$ *division-segment* l›
by (*simp add*: *division-segment-int-def sgn-if split*: *if-splits*)
with ‹$|r| < |l|$› ‹$k = q * l + r$›
show *?case*
by *simp*
qed

56.5.5 Trivial reduction steps

lemma *div-pos-pos-trivial* [*simp*]:
k *div* $l = 0$ **if** $k \geq 0$ **and** $k < l$ **for** $k\ l$ :: *int*
using *that* **by** (*simp add*: *unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def*)

lemma *mod-pos-pos-trivial* [*simp*]:
k *mod* $l = k$ **if** $k \geq 0$ **and** $k < l$ **for** $k\ l$ :: *int*
using *that* **by** (*simp add*: *mod-eq-self-iff-div-eq-0*)

lemma *div-neg-neg-trivial* [*simp*]:
k *div* $l = 0$ **if** $k \leq 0$ **and** $l < k$ **for** $k\ l$ :: *int*
using *that* **by** (*cases* $k = 0$) (*simp*, *simp add*: *unique-euclidean-semiring-class.div-eq-0-iff division-segment-int-def*)

lemma *mod-neg-neg-trivial* [*simp*]:
k *mod* $l = k$ **if** $k \leq 0$ **and** $l < k$ **for** $k\ l$ :: *int*
using *that* **by** (*simp add*: *mod-eq-self-iff-div-eq-0*)

lemma
div-pos-neg-trivial: ‹k *div* $l = -\ 1$› (**is** *?Q*)
and *mod-pos-neg-trivial*: ‹k *mod* $l = k + l$› (**is** *?R*)
if ‹$0 < k$› **and** ‹$k + l \leq 0$› **for** $k\ l$ :: *int*
proof −
from *that* **have** ‹$l < 0$›
by *simp*

```
  have ‹(k div l, k mod l) = (− 1, k + l)›
  proof (induction rule: euclidean-relation-intI)
    case by0
    with ‹l < 0› show ?case
      by simp
  next
    case divides
    from ‹l dvd k› obtain j where ‹k = l * j› ..
    with ‹l < 0› ‹0 < k› have ‹j < 0›
      by (simp add: zero-less-mult-iff)
    moreover from ‹k + l ≤ 0› ‹k = l * j› have ‹l * (j + 1) ≤ 0›
      by (simp add: algebra-simps)
    with ‹l < 0› have ‹j + 1 ≥ 0›
      by (simp add: mult-le-0-iff)
    with ‹j < 0› have ‹j = − 1›
      by simp
    with ‹k = l * j› show ?case
      by simp
  next
    case euclidean-relation
    with ‹k + l ≤ 0› have ‹k + l < 0›
      by (auto simp add: less-le add-eq-0-iff)
    with ‹0 < k› show ?case
      by simp
  qed
  then show ?Q and ?R
    by simp-all
qed
```

There is neither *div-neg-pos-trivial* nor *mod-neg-pos-trivial* because *0 div l = 0* would supersede it.

56.5.6 More uniqueness rules

```
lemma
  fixes a b q r :: int
  assumes ‹a = b * q + r› ‹0 ≤ r› ‹r < b›
  shows int-div-pos-eq:
      ‹a div b = q› (is ?Q)
    and int-mod-pos-eq:
      ‹a mod b = r› (is ?R)
proof −
  have ‹(a div b, a mod b) = (q, r)›
    by (induction rule: euclidean-relation-intI)
      (use assms in ‹auto simp add: ac-simps dvd-add-left-iff sgn-1-pos le-less dest:
zdvd-imp-le›)
  then show ?Q and ?R
    by simp-all
qed
```

lemma *int-div-neg-eq*:
‹*a div b = q*› **if** ‹*a = b * q + r*› ‹*r ≤ 0*› ‹*b < r*› **for** *a b q r :: int*
using *that int-div-pos-eq* [*of a ‹− b› ‹− q› ‹− r›*] **by** *simp-all*

lemma *int-mod-neg-eq*:
‹*a mod b = r*› **if** ‹*a = b * q + r*› ‹*r ≤ 0*› ‹*b < r*› **for** *a b q r :: int*
using *that int-div-neg-eq* [*of a b q r*] **by** *simp*

56.5.7 Laws for unary minus

lemma *zmod-zminus1-not-zero*:
fixes *k l :: int*
shows *− k mod l ≠ 0 ⟹ k mod l ≠ 0*
by (*simp add*: *mod-eq-0-iff-dvd*)

lemma *zmod-zminus2-not-zero*:
fixes *k l :: int*
shows *k mod − l ≠ 0 ⟹ k mod l ≠ 0*
by (*simp add*: *mod-eq-0-iff-dvd*)

lemma *zdiv-zminus1-eq-if*:
‹(*− a*) *div b* = (*if a mod b = 0 then − (a div b) else − (a div b) − 1*)›
if ‹*b ≠ 0*› **for** *a b :: int*
using *that sgn-not-eq-imp* [*of b ‹− a›*]
by (*cases ‹a = 0›*) (*auto simp add*: *div-eq-div-abs* [*of ‹− a› b*] *div-eq-div-abs* [*of a b*] *sgn-eq-0-iff*)

lemma *zdiv-zminus2-eq-if*:
‹*a div* (*− b*) = (*if a mod b = 0 then − (a div b) else − (a div b) − 1*)›
if ‹*b ≠ 0*› **for** *a b :: int*
using *that* **by** (*auto simp add*: *zdiv-zminus1-eq-if div-minus-right*)

lemma *zmod-zminus1-eq-if*:
‹(*− a*) *mod b* = (*if a mod b = 0 then 0 else b − (a mod b*))›
for *a b :: int*
by (*cases ‹b = 0›*)
(*auto simp flip*: *minus-div-mult-eq-mod simp add*: *zdiv-zminus1-eq-if algebra-simps*)

lemma *zmod-zminus2-eq-if*:
‹*a mod* (*− b*) = (*if a mod b = 0 then 0 else* (*a mod b*) *− b*)›
for *a b :: int*
by (*auto simp add*: *zmod-zminus1-eq-if mod-minus-right*)

56.5.8 Borders

lemma *pos-mod-bound* [*simp*]:
k mod l < l **if** *l > 0* **for** *k l :: int*
proof −
obtain *m* **and** *s* **where** *k = sgn s * int m*

```
    by (rule int-sgnE)
  moreover from that obtain n where l = sgn 1 * int n
    by (cases l) simp-all
  moreover from this that have n > 0
    by simp
  ultimately show ?thesis
    by (simp only: modulo-int-unfold)
      (auto simp add: mod-greater-zero-iff-not-dvd sgn-1-pos)
qed

lemma neg-mod-bound [simp]:
  l < k mod l if l < 0 for k l :: int
proof −
  obtain m and s where k = sgn s * int m
    by (rule int-sgnE)
  moreover from that obtain q where l = sgn (− 1) * int (Suc q)
    by (cases l) simp-all
  moreover define n where n = Suc q
  then have Suc q = n
    by simp
  ultimately show ?thesis
    by (simp only: modulo-int-unfold)
      (auto simp add: mod-greater-zero-iff-not-dvd sgn-1-neg)
qed

lemma pos-mod-sign [simp]:
  0 ≤ k mod l if l > 0 for k l :: int
proof −
  obtain m and s where k = sgn s * int m
    by (rule int-sgnE)
  moreover from that obtain n where l = sgn 1 * int n
    by (cases l) auto
  moreover from this that have n > 0
    by simp
  ultimately show ?thesis
    by (simp only: modulo-int-unfold) (auto simp add: sgn-1-pos)
qed

lemma neg-mod-sign [simp]:
  k mod l ≤ 0 if l < 0 for k l :: int
proof −
  obtain m and s where k = sgn s * int m
    by (rule int-sgnE)
  moreover from that obtain q where l = sgn (− 1) * int (Suc q)
    by (cases l) simp-all
  moreover define n where n = Suc q
  then have Suc q = n
    by simp
  moreover have ‹int (m mod n) ≤ int n›
```

using ‹*Suc q = n*› **by** *simp*
then have ‹*sgn s * int (m mod n) ≤ int n*›
by (*cases s* ‹*0::int*› *rule*: *linorder-cases*) *simp-all*
ultimately show *?thesis*
by (*simp only*: *modulo-int-unfold*) *auto*
qed

56.5.9 Splitting Rules for div and mod

lemma *split-zdiv*:
‹*P (n div k) ⟷*
(k = 0 ⟶ P 0) ∧
*(0 < k ⟶ (∀ i j. 0 ≤ j ∧ j < k ∧ n = k * i + j ⟶ P i)) ∧*
*(k < 0 ⟶ (∀ i j. k < j ∧ j ≤ 0 ∧ n = k * i + j ⟶ P i))*› (**is** *?div*)
and *split-zmod*:
‹*Q (n mod k) ⟷*
(k = 0 ⟶ Q n) ∧
*(0 < k ⟶ (∀ i j. 0 ≤ j ∧ j < k ∧ n = k * i + j ⟶ Q j)) ∧*
*(k < 0 ⟶ (∀ i j. k < j ∧ j ≤ 0 ∧ n = k * i + j ⟶ Q j))*› (**is** *?mod*)
for *n k :: int*
proof −
have *∗*: ‹*R (n div k) (n mod k) ⟷*
(k = 0 ⟶ R 0 n) ∧
*(0 < k ⟶ (∀ i j. 0 ≤ j ∧ j < k ∧ n = k * i + j ⟶ R i j)) ∧*
*(k < 0 ⟶ (∀ i j. k < j ∧ j ≤ 0 ∧ n = k * i + j ⟶ R i j))*› **for** *R*
by (*cases* ‹*k = 0*›)
(*auto simp add*: *linorder-class.neq-iff*)
from *∗* [*of* ‹*λq -. P q*›] **show** *?div* **.**
from *∗* [*of* ‹*λ- r. Q r*›] **show** *?mod* **.**
qed

Enable (lin)arith to deal with (*div*) and (*mod*) when these are applied to some constant that is of the form *numeral k*:

declare *split-zdiv* [*of - -* ‹*numeral n*›, *linarith-split*] **for** *n*
declare *split-zdiv* [*of - -* ‹*− numeral n*›, *linarith-split*] **for** *n*
declare *split-zmod* [*of - -* ‹*numeral n*›, *linarith-split*] **for** *n*
declare *split-zmod* [*of - -* ‹*− numeral n*›, *linarith-split*] **for** *n*

lemma *zdiv-eq-0-iff*:
i div k = 0 ⟷ k = 0 ∨ 0 ≤ i ∧ i < k ∨ i ≤ 0 ∧ k < i (**is** *?L = ?R*)
for *i k :: int*
proof
assume *?L*
moreover have *?L ⟶ ?R*
by (*rule split-zdiv* [*THEN iffD2*]) *simp*
ultimately show *?R*
by *blast*
next
assume *?R* **then show** *?L*

by *auto*
qed

lemma *zmod-trivial-iff*:
fixes *i k* :: *int*
shows *i mod k* = *i* ⟷ *k* = *0* ∨ *0* ≤ *i* ∧ *i* < *k* ∨ *i* ≤ *0* ∧ *k* < *i*
proof −
have *i mod k* = *i* ⟷ *i div k* = *0*
using *div-mult-mod-eq* [*of i k*] **by** *safe auto*
with *zdiv-eq-0-iff*
show *?thesis*
by *simp*
qed

56.5.10 Algebraic rewrites

lemma *zdiv-zmult2-eq*: ‹*a div* (*b* ∗ *c*) = (*a div b*) *div c*› (**is** *?Q*)
and *zmod-zmult2-eq*: ‹*a mod* (*b* ∗ *c*) = *b* ∗ (*a div b mod c*) + *a mod b*› (**is** *?P*)
if ‹*c* ≥ *0*› **for** *a b c* :: *int*
proof −
have ∗: ‹(*a div* (*b* ∗ *c*), *a mod* (*b* ∗ *c*)) = ((*a div b*) *div c*, *b* ∗ (*a div b mod c*) + *a mod b*)›
if ‹*b* > *0*› **for** *a b*
proof (*induction rule*: *euclidean-relationI*)
case *by0*
then show *?case* **by** *auto*
next
case *divides*
then obtain *d* **where** ‹*a* = *b* ∗ *c* ∗ *d*›
by *blast*
with *divides that* **show** *?case*
by (*simp add*: *ac-simps*)
next
case *euclidean-relation*
with ‹*b* > *0*› ‹*c* ≥ *0*› **have** ‹*0* < *c*› ‹*b* > *0*›
by *simp-all*
then have ‹*a mod b* < *b*›
by *simp*
moreover have ‹*1* ≤ *c* − *a div b mod c*›
using ‹*c* > *0*› **by** (*simp add*: *int-one-le-iff-zero-less*)
ultimately have ‹*a mod b* ∗ *1* < *b* ∗ (*c* − *a div b mod c*)›
by (*rule mult-less-le-imp-less*) (*use* ‹*b* > *0*› **in** *simp-all*)
with ‹*0* < *b*› ‹*0* < *c*› **show** *?case*
by (*simp add*: *division-segment-int-def algebra-simps flip*: *minus-mod-eq-mult-div*)
qed
show *?Q*
proof (*cases* ‹*b* ≥ *0*›)
case *True*
with ∗ [*of b a*] **show** *?thesis*

```
    by (cases ‹b = 0›) simp-all
  next
    case False
    with * [of ‹- b› ‹- a›] show ?thesis
      by simp
  qed
  show ?P
  proof (cases ‹b ≥ 0›)
    case True
    with * [of b a] show ?thesis
      by (cases ‹b = 0›) simp-all
  next
    case False
    with * [of ‹- b› ‹- a›] show ?thesis
      by simp
  qed
qed

lemma zdiv-zmult2-eq':
  ‹k div (l * j) = ((sgn j * k) div l) div |j|› for k l j :: int
proof -
  have ‹k div (l * j) = (sgn j * k) div (sgn j * (l * j))›
    by (simp add: sgn-0-0)
  also have ‹sgn j * (l * j) = l * |j|›
    by (simp add: mult.left-commute [of - l] abs-sgn) (simp add: ac-simps)
  also have ‹(sgn j * k) div (l * |j|) = ((sgn j * k) div l) div |j|›
    by (simp add: zdiv-zmult2-eq)
  finally show ?thesis .
qed

lemma half-nonnegative-int-iff [simp]:
  ‹k div 2 ≥ 0 ⟷ k ≥ 0› for k :: int
  by auto

lemma half-negative-int-iff [simp]:
  ‹k div 2 < 0 ⟷ k < 0› for k :: int
  by auto
```

56.5.11 Distributive laws for conversions.

```
lemma zdiv-int:
  ‹int (m div n) = int m div int n›
  by (cases ‹m = 0›) (auto simp add: divide-int-def)

lemma zmod-int:
  ‹int (m mod n) = int m mod int n›
  by (cases ‹m = 0›) (auto simp add: modulo-int-def)

lemma nat-div-distrib:
```

‹*nat* (*x div y*) = *nat x div nat y*› **if** ‹*0* ≤ *x*›
using *that* **by** (*simp add*: *divide-int-def sgn-if*)

lemma *nat-div-distrib'*:
‹*nat* (*x div y*) = *nat x div nat y*› **if** ‹*0* ≤ *y*›
using *that* **by** (*simp add*: *divide-int-def sgn-if*)

lemma *nat-mod-distrib*: — Fails if y<0: the LHS collapses to (nat z) but the RHS doesn't
‹*nat* (*x mod y*) = *nat x mod nat y*› **if** ‹*0* ≤ *x*› ‹*0* ≤ *y*›
using *that* **by** (*simp add*: *modulo-int-def sgn-if*)

56.5.12 Monotonicity in the First Argument (Dividend)

lemma *zdiv-mono1*:
‹*a div b* ≤ *a' div b*›
if ‹*a* ≤ *a'*› ‹*0* < *b*›
for *a b b'* :: *int*
proof −
from ‹*a* ≤ *a'*› **have** ‹*b* ∗ (*a div b*) + *a mod b* ≤ *b* ∗ (*a' div b*) + *a' mod b*›
by *simp*
then have ‹*b* ∗ (*a div b*) ≤ (*a' mod b* − *a mod b*) + *b* ∗ (*a' div b*)›
by (*simp add*: *algebra-simps*)
moreover have ‹*a' mod b* < *b* + *a mod b*›
by (*rule less-le-trans* [*of* - *b*]) (*use* ‹*0* < *b*› **in** *simp-all*)
ultimately have ‹*b* ∗ (*a div b*) < *b* ∗ (*1* + *a' div b*)›
by (*simp add*: *distrib-left*)
with ‹*0* < *b*› **have** ‹*a div b* < *1* + *a' div b*›
by (*simp add*: *mult-less-cancel-left*)
then show *?thesis*
by *simp*
qed

lemma *zdiv-mono1-neg*:
‹*a' div b* ≤ *a div b*›
if ‹*a* ≤ *a'*› ‹*b* < *0*›
for *a a' b* :: *int*
using *that zdiv-mono1* [*of* ‹− *a'*› ‹− *a*› ‹− *b*›] **by** *simp*

56.5.13 Monotonicity in the Second Argument (Divisor)

lemma *zdiv-mono2*:
‹*a div b* ≤ *a div b'*› **if** ‹*0* ≤ *a*› ‹*0* < *b'*› ‹*b'* ≤ *b*› **for** *a b b'* :: *int*
proof −
define *q q' r r'* **where** ∗∗: ‹*q* = *a div b*› ‹*q'* = *a div b'*› ‹*r* = *a mod b*› ‹*r'* = *a mod b'*›
then have ∗: ‹*b* ∗ *q* + *r* = *b'* ∗ *q'* + *r'*› ‹*0* ≤ *b'* ∗ *q'* + *r'*›
‹*r'* < *b'*› ‹*0* ≤ *r*› ‹*0* < *b'*› ‹*b'* ≤ *b*›
using *that* **by** *simp-all*
have ‹*0* < *b'* ∗ (*q'* + *1*)›

using $*$ **by** (*simp add*: *distrib-left*)
with $*$ **have** ‹$0 \leq q'$›
by (*simp add*: *zero-less-mult-iff*)
moreover have ‹$b * q = r' - r + b' * q'$›
using $*$ **by** *linarith*
ultimately have ‹$b * q < b * (q' + 1)$›
using *mult-right-mono* $*$ **unfolding** *distrib-left* **by** *fastforce*
with $*$ **have** ‹$q \leq q'$›
by (*simp add*: *mult-less-cancel-left-pos*)
with $**$ **show** *?thesis*
by *simp*
qed

lemma *zdiv-mono2-neg*:
‹a *div* $b' \leq a$ *div* b› **if** ‹$a < 0$› ‹$0 < b'$› ‹$b' \leq b$› **for** $a\ b\ b' :: int$
proof −
define $q\ q'\ r\ r'$ **where** $**$: ‹$q = a$ *div* b› ‹$q' = a$ *div* b'› ‹$r = a$ *mod* b› ‹$r' = a$ *mod* b'›
then have $*$: ‹$b * q + r = b' * q' + r'$› ‹$b' * q' + r' < 0$›
‹$r < b$› ‹$0 \leq r'$› ‹$0 < b'$› ‹$b' \leq b$›
using *that* **by** *simp-all*
have ‹$b' * q' < 0$›
using $*$ **by** *linarith*
with $*$ **have** ‹$q' \leq 0$›
by (*simp add*: *mult-less-0-iff*)
have ‹$b * q' \leq b' * q'$›
by (*simp add*: ‹$q' \leq 0$› $*$ *mult-right-mono-neg*)
then have $b * q' < b * (q + 1)$
using $*$ **by** (*simp add*: *distrib-left*)
then have ‹$q' \leq q$›
using $*$ **by** (*simp add*: *mult-less-cancel-left*)
then show *?thesis*
by (*simp add*: $**$)
qed

56.5.14 Quotients of Signs

lemma *div-eq-minus1*:
‹$0 < b \Longrightarrow -1$ *div* $b = -1$› **for** $b :: int$
by (*simp add*: *divide-int-def*)

lemma *zmod-minus1*:
‹$0 < b \Longrightarrow -1$ *mod* $b = b - 1$› **for** $b :: int$
by (*auto simp add*: *modulo-int-def*)

lemma *minus-mod-int-eq*:
‹$-k$ *mod* $l = l - 1 - (k - 1)$ *mod* l› **if** ‹$l \geq 0$› **for** $k\ l :: int$
proof (*cases* ‹$l = 0$›)
case *True*

```
  then show ?thesis
    by simp
next
  case False
  with that have ‹l > 0›
    by simp
  then show ?thesis
  proof (cases ‹l dvd k›)
    case True
    then obtain j where ‹k = l * j› ..
    moreover have ‹(l * j mod l - 1) mod l = l - 1›
      using ‹l > 0› by (simp add: zmod-minus1)
    then have ‹(l * j - 1) mod l = l - 1›
      by (simp only: mod-simps)
    ultimately show ?thesis
      by simp
  next
    case False
    moreover have 1: ‹0 < k mod l›
      using ‹0 < l› False le-less by fastforce
    moreover have 2: ‹k mod l < 1 + l›
      using ‹0 < l› pos-mod-bound[of l k] by linarith
    from 1 2 ‹l > 0› have ‹(k mod l - 1) mod l = k mod l - 1›
      by (simp add: zmod-trivial-iff)
    ultimately show ?thesis
      by (simp only: zmod-zminus1-eq-if)
        (simp add: mod-eq-0-iff-dvd algebra-simps mod-simps)
  qed
qed

lemma div-neg-pos-less0:
  ‹a div b < 0› if ‹a < 0› ‹0 < b› for a b :: int
proof -
  have a div b ≤ - 1 div b
    using zdiv-mono1 that by auto
  also have ... ≤ -1
    by (simp add: that(2) div-eq-minus1)
  finally show ?thesis
    by force
qed

lemma div-nonneg-neg-le0:
  ‹a div b ≤ 0› if ‹0 ≤ a› ‹b < 0› for a b :: int
  using that by (auto dest: zdiv-mono1-neg)

lemma div-nonpos-pos-le0:
  ‹a div b ≤ 0› if ‹a ≤ 0› ‹0 < b› for a b :: int
  using that by (auto dest: zdiv-mono1)
```

Now for some equivalences of the form *a div b >=< 0* $\longleftrightarrow$ ... conditional

upon the sign of *a* or *b*. There are many more. They should all be simp rules unless that causes too much search.

lemma *pos-imp-zdiv-nonneg-iff*:
‹*0* ≤ *a div b* ⟷ *0* ≤ *a*›
if ‹*0* < *b*› **for** *a b* :: *int*
proof
assume ‹*0* ≤ *a div b*›
show ‹*0* ≤ *a*›
proof (*rule ccontr*)
assume ‹¬ *0* ≤ *a*›
then have ‹*a* < *0*›
by *simp*
then have ‹*a div b* < *0*›
using *that* **by** (*rule div-neg-pos-less0*)
with ‹*0* ≤ *a div b*› **show** *False*
by *simp*
qed
next
assume *0* ≤ *a*
then have *0 div b* ≤ *a div b*
using *zdiv-mono1 that* **by** *blast*
then show *0* ≤ *a div b*
by *auto*
qed

lemma *neg-imp-zdiv-nonneg-iff*:
‹*0* ≤ *a div b* ⟷ *a* ≤ *0*› **if** ‹*b* < *0*› **for** *a b* :: *int*
using *that pos-imp-zdiv-nonneg-iff* [*of* ‹− *b*› ‹− *a*›] **by** *simp*

lemma *pos-imp-zdiv-pos-iff*:
‹*0* < (*i*::*int*) *div k* ⟷ *k* ≤ *i*› **if** ‹*0* < *k*› **for** *i k* :: *int*
using *that pos-imp-zdiv-nonneg-iff* [*of k i*] *zdiv-eq-0-iff* [*of i k*] **by** *arith*

lemma *pos-imp-zdiv-neg-iff*:
‹*a div b* < *0* ⟷ *a* < *0*› **if** ‹*0* < *b*› **for** *a b* :: *int*
— But not (*a div b* ≤ *0*) = (*a* ≤ *0*); consider *a* = *1*, *b* = *2* when *a div b* = *0*.
using *that* **by** (*simp add*: *pos-imp-zdiv-nonneg-iff flip*: *linorder-not-le*)

lemma *neg-imp-zdiv-neg-iff*:
— But not (*a div b* ≤ *0*) = (*0* ≤ *a*); consider *a* = − *1*, *b* = − *2* when *a div b* = *0*.
‹*a div b* < *0* ⟷ *0* < *a*› **if** ‹*b* < *0*› **for** *a b* :: *int*
using *that* **by** (*simp add*: *neg-imp-zdiv-nonneg-iff flip*: *linorder-not-le*)

lemma *nonneg1-imp-zdiv-pos-iff*:
‹*a div b* > *0* ⟷ *a* ≥ *b* ∧ *b* > *0*› **if** ‹*0* ≤ *a*› **for** *a b* :: *int*
proof −
have *0* < *a div b* ⟹ *b* ≤ *a*
using *div-pos-pos-trivial*[*of a b*] *that* **by** *arith*

```
  moreover have 0 < a div b ⟹ b > 0
    using that div-nonneg-neg-le0[of a b] by (cases b=0; force)
  moreover have b ≤ a ∧ 0 < b ⟹ 0 < a div b
    using int-one-le-iff-zero-less[of a div b] zdiv-mono1[of b a b] by simp
  ultimately show ?thesis
    by blast
qed

lemma zmod-le-nonneg-dividend:
  ‹m mod k ≤ m› if ‹(m::int) ≥ 0› for m k :: int
proof −
  from that have ‹m > 0 ∨ m = 0›
    by auto
  then show ?thesis proof
    assume ‹m = 0› then show ?thesis
      by simp
  next
    assume ‹m > 0› then show ?thesis
    proof (cases k ‹0::int› rule: linorder-cases)
      case less
      moreover define l where ‹l = − k›
      ultimately have ‹l > 0›
        by simp
      with ‹m > 0› have ‹int (nat m mod nat l) ≤ m›
        by (simp flip: le-nat-iff)
      then have ‹int (nat m mod nat l) − l ≤ m›
        using ‹l > 0› by simp
      with ‹m > 0› ‹l > 0› show ?thesis
        by (simp add: modulo-int-def l-def flip: le-nat-iff)
    qed (simp-all add: modulo-int-def flip: le-nat-iff)
  qed
qed

lemma sgn-div-eq-sgn-mult:
  ‹sgn (k div l) = of-bool (k div l ≠ 0) * sgn (k * l)›
  for k l :: int
proof (cases ‹k div l = 0›)
  case True
  then show ?thesis
    by simp
next
  case False
  have ‹0 ≤ |k| div |l|›
    by (cases ‹l = 0›) (simp-all add: pos-imp-zdiv-nonneg-iff)
  then have ‹|k| div |l| ≠ 0 ⟷ 0 < |k| div |l|›
    by (simp add: less-le)
  also have ‹... ⟷ |k| ≥ |l|›
    using False nonneg1-imp-zdiv-pos-iff by auto
  finally have *: ‹|k| div |l| ≠ 0 ⟷ |l| ≤ |k|› .
```

show *?thesis*
using ‹*0 ≤ |k| div |l|*› *False*
by (*auto simp add*: *div-eq-div-abs* [*of k l*] *div-eq-sgn-abs* [*of k l*]
sgn-mult sgn-1-pos sgn-1-neg sgn-eq-0-iff nonneg1-imp-zdiv-pos-iff ∗ *dest*: *sgn-not-eq-imp*)
qed

56.5.15 Further properties

lemma *div-int-pos-iff*:
k div l ≥ 0 ⟷ k = 0 ∨ l = 0 ∨ k ≥ 0 ∧ l ≥ 0
∨ k < 0 ∧ l < 0
for *k l* :: *int*
proof (*cases k = 0 ∨ l = 0*)
case *False*
then have ∗: *k ≠ 0 l ≠ 0*
by *auto*
then have *0 ≤ k div l ⟹ ¬ k < 0 ⟹ 0 ≤ l*
by (*meson neg-imp-zdiv-neg-iff not-le not-less-iff-gr-or-eq*)
then show *?thesis*
using ∗ **by** (*auto simp add*: *pos-imp-zdiv-nonneg-iff neg-imp-zdiv-nonneg-iff*)
qed *auto*

lemma *mod-int-pos-iff*:
‹*k mod l ≥ 0 ⟷ l dvd k ∨ l = 0 ∧ k ≥ 0 ∨ l > 0*›
for *k l* :: *int*
proof (*cases l > 0*)
case *False*
then show *?thesis*
by (*simp add*: *dvd-eq-mod-eq-0*) (*use neg-mod-sign* [*of l k*] **in** ‹*auto simp add*:
le-less not-less›)
qed *auto*

lemma *abs-div*:
‹|*x div y*| = |*x*| *div* |*y*|› **if** ‹*y dvd x*› **for** *x y* :: *int*
using *that* **by** (*cases* ‹*y = 0*›) (*auto simp add*: *abs-mult*)

lemma *int-power-div-base*:
‹*k ^ m div k = k ^ (m − Suc 0)*› **if** ‹*0 < m*› ‹*0 < k*› **for** *k* :: *int*
using *that* **by** (*cases m*) *simp-all*

lemma *int-div-less-self*:
‹*x div k < x*› **if** ‹*0 < x*› ‹*1 < k*› **for** *x k* :: *int*
proof −
from *that* **have** ‹*nat* (*x div k*) = *nat x div nat k*›
by (*simp add*: *nat-div-distrib*)
also from *that* **have** ‹*nat x div nat k < nat x*›
by *simp*
finally show *?thesis*
by *simp*

qed

lemma *int-div-le-self*:
  ‹*x div k* ≤ *x*› **if** ‹*0* < *x*› **for** *x k* :: *int*
  by (*metis div-by-1 int-div-less-self less-le-not-le nle-le nonneg1-imp-zdiv-pos-iff order.trans that*)

56.5.16 Computing *div* and *mod* by shifting

lemma *div-pos-geq*:
  ‹*k div l* = (*k* − *l*) *div l* + *1*› **if** ‹*0* < *l*› ‹*l* ≤ *k*› **for** *k l* :: *int*
proof −
  have *k* = (*k* − *l*) + *l* **by** *simp*
  then obtain *j* **where** *k*: *k* = *j* + *l* **..**
  with *that* **show** *?thesis* **by** (*simp add*: *div-add-self2*)
qed

lemma *mod-pos-geq*:
  ‹*k mod l* = (*k* − *l*) *mod l*› **if** ‹*0* < *l*› ‹*l* ≤ *k*› **for** *k l* :: *int*
proof −
  have *k* = (*k* − *l*) + *l* **by** *simp*
  then obtain *j* **where** *k*: *k* = *j* + *l* **..**
  with *that* **show** *?thesis* **by** *simp*
qed

lemma *pos-zdiv-mult-2*: ‹(*1* + *2* ∗ *b*) *div* (*2* ∗ *a*) = *b div a*› (**is** *?Q*)
  and *pos-zmod-mult-2*: ‹(*1* + *2* ∗ *b*) *mod* (*2* ∗ *a*) = *1* + *2* ∗ (*b mod a*)› (**is** *?R*)
  if ‹*0* ≤ *a*› **for** *a b* :: *int*
proof −
  have ‹((*1* + *2* ∗ *b*) *div* (*2* ∗ *a*), (*1* + *2* ∗ *b*) *mod* (*2* ∗ *a*)) = (*b div a*, *1* + *2* ∗ (*b mod a*))›
  proof (*induction rule*: *euclidean-relation-intI*)
    case *by0*
    then show *?case*
      by *simp*
  next
    case *divides*
    have ‹*2 dvd* (*2* ∗ *a*)›
      by *simp*
    then have ‹*2 dvd* (*1* + *2* ∗ *b*)›
      using ‹*2* ∗ *a dvd 1* + *2* ∗ *b*› **by** (*rule dvd-trans*)
    then have ‹*2 dvd* (*1* + *b* ∗ *2*)›
      by (*simp add*: *ac-simps*)
    then have ‹*is-unit* (*2* :: *int*)›
      by *simp*
    then show *?case*
      by *simp*
  next
    case *euclidean-relation*

```
    with that have ‹a > 0›
      by simp
    moreover have ‹b mod a < a›
      using ‹a > 0› by simp
    then have ‹1 + 2 * (b mod a) < 2 * a›
      by simp
    moreover have ‹2 * (b mod a) + a * (2 * (b div a)) = 2 * (b div a * a + b
mod a)›
      by (simp only: algebra-simps)
    moreover have ‹0 ≤ 2 * (b mod a)›
      using ‹a > 0› by simp
    ultimately show ?case
      by (simp add: algebra-simps)
  qed
  then show ?Q and ?R
    by simp-all
qed

lemma neg-zdiv-mult-2: ‹(1 + 2 * b) div (2 * a) = (b + 1) div a› (is ?Q)
  and neg-zmod-mult-2: ‹(1 + 2 * b) mod (2 * a) = 2 * ((b + 1) mod a) - 1›
(is ?R)
  if ‹a ≤ 0› for a b :: int
proof -
  have ‹((1 + 2 * b) div (2 * a), (1 + 2 * b) mod (2 * a)) = ((b + 1) div a, 2
* ((b + 1) mod a) - 1)›
  proof (induction rule: euclidean-relation-intI)
    case by0
    then show ?case
      by simp
  next
    case divides
    have ‹2 dvd (2 * a)›
      by simp
    then have ‹2 dvd (1 + 2 * b)›
      using ‹2 * a dvd 1 + 2 * b› by (rule dvd-trans)
    then have ‹2 dvd (1 + b * 2)›
      by (simp add: ac-simps)
    then have ‹is-unit (2 :: int)›
      by simp
    then show ?case
      by simp
  next
    case euclidean-relation
    with that have ‹a < 0›
      by simp
    moreover have ‹(b + 1) mod a > a›
      using ‹a < 0› by simp
    then have ‹2 * ((b + 1) mod a) > 1 + 2 * a›
      by simp
```

```
    moreover have ‹((1 + b) mod a) ≤ 0›
      using ‹a < 0› by simp
    then have ‹2 * ((1 + b) mod a) ≤ 0›
      by simp
    moreover have ‹2 * ((1 + b) mod a) + a * (2 * ((1 + b) div a)) =
      2 * ((1 + b) div a * a + (1 + b) mod a)›
      by (simp only: algebra-simps)
    ultimately show ?case
      by (simp add: algebra-simps sgn-mult abs-mult)
  qed
  then show ?Q and ?R
    by simp-all
qed

lemma zdiv-numeral-Bit0 [simp]:
  ‹numeral (Num.Bit0 v) div numeral (Num.Bit0 w) =
    numeral v div (numeral w :: int)›
  unfolding numeral.simps unfolding mult-2 [symmetric]
  by (rule div-mult-mult1) simp

lemma zdiv-numeral-Bit1 [simp]:
  ‹numeral (Num.Bit1 v) div numeral (Num.Bit0 w) =
    (numeral v div (numeral w :: int))›
  unfolding numeral.simps
  unfolding mult-2 [symmetric] add.commute [of - 1]
  by (rule pos-zdiv-mult-2) simp

lemma zmod-numeral-Bit0 [simp]:
  ‹numeral (Num.Bit0 v) mod numeral (Num.Bit0 w) =
    (2::int) * (numeral v mod numeral w)›
  unfolding numeral-Bit0 [of v] numeral-Bit0 [of w]
  unfolding mult-2 [symmetric] by (rule mod-mult-mult1)

lemma zmod-numeral-Bit1 [simp]:
  ‹numeral (Num.Bit1 v) mod numeral (Num.Bit0 w) =
    2 * (numeral v mod numeral w) + (1::int)›
  unfolding numeral-Bit1 [of v] numeral-Bit0 [of w]
  unfolding mult-2 [symmetric] add.commute [of - 1]
  by (rule pos-zmod-mult-2) simp
```

56.6 Code generation

```
context
begin

qualified definition divmod-nat :: nat ⇒ nat ⇒ nat × nat
  where divmod-nat m n = (m div n, m mod n)

qualified lemma divmod-nat-if [code]:
```

divmod-nat m n = (*if n* = *0* ∨ *m* < *n then* (*0*, *m*) *else*
let (*q*, *r*) = *divmod-nat* (*m* − *n*) *n in* (*Suc q*, *r*))
by (*simp add*: *divmod-nat-def prod-eq-iff case-prod-beta not-less le-div-geq le-mod-geq*)

qualified lemma [*code*]:
m div n = *fst* (*divmod-nat m n*)
m mod n = *snd* (*divmod-nat m n*)
by (*simp-all add*: *divmod-nat-def*)

end

code-identifier
code-module *Euclidean-Rings* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

57 Parity in rings and semirings

theory *Parity*
imports *Euclidean-Rings*
begin

57.1 Ring structures with parity and *even*/*odd* predicates

class *semiring-parity* = *comm-semiring-1* + *semiring-modulo* +
assumes *mod-2-eq-odd*: ‹*a mod 2* = *of-bool* (¬ *2 dvd a*)›
and *odd-one* [*simp*]: ‹¬ *2 dvd 1*›
and *even-half-succ-eq* [*simp*]: ‹*2 dvd a* ⟹ (*1* + *a*) *div 2* = *a div 2*›
begin

abbreviation *even* :: *'a* ⇒ *bool*
where ‹*even a* ≡ *2 dvd a*›

abbreviation *odd* :: *'a* ⇒ *bool*
where ‹*odd a* ≡ ¬ *2 dvd a*›

end

class *ring-parity* = *ring* + *semiring-parity*
begin

subclass *comm-ring-1* **..**

end

instance *nat* :: *semiring-parity*
by *standard* (*auto simp add*: *dvd-eq-mod-eq-0*)

instance *int* :: *ring-parity*
by *standard* (*auto simp add*: *dvd-eq-mod-eq-0*)

context *semiring-parity*
begin

lemma *evenE* [*elim?*]:
assumes ‹*even a*›
obtains *b* **where** ‹*a* = *2* ∗ *b*›
using *assms* **by** (*rule dvdE*)

lemma *oddE* [*elim?*]:
assumes ‹*odd a*›
obtains *b* **where** ‹*a* = *2* ∗ *b* + *1*›
proof −
have ‹*a* = *2* ∗ (*a div 2*) + *a mod 2*›
by (*simp add*: *mult-div-mod-eq*)
with *assms* **have** ‹*a* = *2* ∗ (*a div 2*) + *1*›
by (*simp add*: *mod-2-eq-odd*)
then show *thesis* **..**
qed

lemma *of-bool-odd-eq-mod-2*:
‹*of-bool* (*odd a*) = *a mod 2*›
by (*simp add*: *mod-2-eq-odd*)

lemma *odd-of-bool-self* [*simp*]:
‹*odd* (*of-bool p*) ⟷ *p*›
by (*cases p*) *simp-all*

lemma *not-mod-2-eq-0-eq-1* [*simp*]:
‹*a mod 2* ≠ *0* ⟷ *a mod 2* = *1*›
by (*simp add*: *mod-2-eq-odd*)

lemma *not-mod-2-eq-1-eq-0* [*simp*]:
‹*a mod 2* ≠ *1* ⟷ *a mod 2* = *0*›
by (*simp add*: *mod-2-eq-odd*)

lemma *even-iff-mod-2-eq-zero*:
‹*2 dvd a* ⟷ *a mod 2* = *0*›
by (*simp add*: *mod-2-eq-odd*)

lemma *odd-iff-mod-2-eq-one*:
‹¬ *2 dvd a* ⟷ *a mod 2* = *1*›
by (*simp add*: *mod-2-eq-odd*)

lemma *even-mod-2-iff* [*simp*]:
‹*even* (*a mod 2*) ⟷ *even a*›
by (*simp add*: *mod-2-eq-odd*)

lemma *mod2-eq-if*:
a mod 2 = (if even a then 0 else 1)
by (*simp add*: *mod-2-eq-odd*)

lemma *zero-mod-two-eq-zero* [*simp*]:
‹*0 mod 2 = 0*›
by (*simp add*: *mod-2-eq-odd*)

lemma *one-mod-two-eq-one* [*simp*]:
‹*1 mod 2 = 1*›
by (*simp add*: *mod-2-eq-odd*)

lemma *parity-cases* [*case-names even odd*]:
assumes ‹*even a* ⟹ *a mod 2 = 0* ⟹ *P*›
assumes ‹*odd a* ⟹ *a mod 2 = 1* ⟹ *P*›
shows *P*
using *assms* **by** (*auto simp add*: *mod-2-eq-odd*)

lemma *even-zero* [*simp*]:
‹*even 0*›
by (*fact dvd-0-right*)

lemma *odd-even-add*:
even (a + b) **if** *odd a* **and** *odd b*
proof −
from *that* **obtain** *c d* **where** *a = 2 * c + 1* **and** *b = 2 * d + 1*
by (*blast elim*: *oddE*)
then have *a + b = 2 * c + 2 * d + (1 + 1)*
by (*simp only*: *ac-simps*)
also have *... = 2 * (c + d + 1)*
by (*simp add*: *algebra-simps*)
finally show *?thesis* **..**
qed

lemma *even-add* [*simp*]:
even (a + b) ⟷ (*even a* ⟷ *even b*)
by (*auto simp add*: *dvd-add-right-iff dvd-add-left-iff odd-even-add*)

lemma *odd-add* [*simp*]:
odd (a + b) ⟷ ¬ (*odd a* ⟷ *odd b*)
by *simp*

lemma *even-plus-one-iff* [*simp*]:
even (a + 1) ⟷ *odd a*
by (*auto simp add*: *dvd-add-right-iff intro*: *odd-even-add*)

lemma *even-mult-iff* [*simp*]:
*even (a * b)* ⟷ *even a* ∨ *even b* (**is** *?P* ⟷ *?Q*)

proof
assume *?Q*
then show *?P*
by *auto*
next
assume *?P*
show *?Q*
proof (*rule ccontr*)
assume ¬ (*even a* ∨ *even b*)
then have *odd a* **and** *odd b*
by *auto*
then obtain *r s* **where** *a* = *2* ∗ *r* + *1* **and** *b* = *2* ∗ *s* + *1*
by (*blast elim*: *oddE*)
then have *a* ∗ *b* = (*2* ∗ *r* + *1*) ∗ (*2* ∗ *s* + *1*)
by *simp*
also have ... = *2* ∗ (*2* ∗ *r* ∗ *s* + *r* + *s*) + *1*
by (*simp add*: *algebra-simps*)
finally have *odd* (*a* ∗ *b*)
by *simp*
with ‹*?P*› **show** *False*
by *auto*
qed
qed

lemma *even-numeral* [*simp*]: *even* (*numeral* (*Num.Bit0 n*))
proof −
have *even* (*2* ∗ *numeral n*)
unfolding *even-mult-iff* **by** *simp*
then have *even* (*numeral n* + *numeral n*)
unfolding *mult-2* **.**
then show *?thesis*
unfolding *numeral.simps* **.**
qed

lemma *odd-numeral* [*simp*]: *odd* (*numeral* (*Num.Bit1 n*))
proof
assume *even* (*numeral* (*num.Bit1 n*))
then have *even* (*numeral n* + *numeral n* + *1*)
unfolding *numeral.simps* **.**
then have *even* (*2* ∗ *numeral n* + *1*)
unfolding *mult-2* **.**
then have *2 dvd numeral n* ∗ *2* + *1*
by (*simp add*: *ac-simps*)
then have *2 dvd 1*
using *dvd-add-times-triv-left-iff* [*of 2 numeral n 1*] **by** *simp*
then show *False* **by** *simp*
qed

lemma *odd-numeral-BitM* [*simp*]:

‹*odd* (*numeral* (*Num.BitM w*))›
by (*cases w*) *simp-all*

lemma *even-power* [*simp*]: *even* (*a* ^ *n*) ⟷ *even a* ∧ *n* > *0*
by (*induct n*) *auto*

lemma *even-prod-iff*:
‹*even* (*prod f A*) ⟷ (∃ *a*∈*A*. *even* (*f a*))› **if** ‹*finite A*›
using *that* **by** (*induction A*) *simp-all*

lemma *even-half-maybe-succ-eq* [*simp*]:
‹*even a* ⟹ (*of-bool b* + *a*) *div 2* = *a div 2*›
by *simp*

lemma *even-half-maybe-succ′-eq* [*simp*]:
‹*even a* ⟹ (*b mod 2* + *a*) *div 2* = *a div 2*›
by (*simp add*: *mod2-eq-if*)

lemma *mask-eq-sum-exp*:
‹*2* ^ *n* − *1* = (∑ *m*∈{*q*. *q* < *n*}. *2* ^ *m*)›
proof −
have ∗: ‹{*q*. *q* < *Suc m*} = *insert m* {*q*. *q* < *m*}› **for** *m*
by *auto*
have ‹*2* ^ *n* = (∑ *m*∈{*q*. *q* < *n*}. *2* ^ *m*) + *1*›
by (*induction n*) (*simp-all add*: *ac-simps mult-2* ∗)
then have ‹*2* ^ *n* − *1* = (∑ *m*∈{*q*. *q* < *n*}. *2* ^ *m*) + *1* − *1*›
by *simp*
then show *?thesis*
by *simp*
qed

lemma (**in** −) *mask-eq-sum-exp-nat*:
‹*2* ^ *n* − *Suc 0* = (∑ *m*∈{*q*. *q* < *n*}. *2* ^ *m*)›
using *mask-eq-sum-exp* [**where** *?′a* = *nat*] **by** *simp*

end

context *ring-parity*
begin

lemma *even-minus*:
even (− *a*) ⟷ *even a*
by (*fact dvd-minus-iff*)

lemma *even-diff* [*simp*]:
even (*a* − *b*) ⟷ *even* (*a* + *b*)
using *even-add* [*of a* − *b*] **by** *simp*

end

57.2 Instance for *nat*

lemma *even-Suc-Suc-iff* [*simp*]:
even (*Suc* (*Suc n*)) ⟷ *even n*
using *dvd-add-triv-right-iff* [*of 2 n*] **by** *simp*

lemma *even-Suc* [*simp*]: *even* (*Suc n*) ⟷ *odd n*
using *even-plus-one-iff* [*of n*] **by** *simp*

lemma *even-diff-nat* [*simp*]:
even (*m* − *n*) ⟷ *m* < *n* ∨ *even* (*m* + *n*) **for** *m n* :: *nat*
proof (*cases n* ≤ *m*)
case *True*
then have *m* − *n* + *n* ∗ *2* = *m* + *n* **by** (*simp add*: *mult-2-right*)
moreover have *even* (*m* − *n*) ⟷ *even* (*m* − *n* + *n* ∗ *2*) **by** *simp*
ultimately have *even* (*m* − *n*) ⟷ *even* (*m* + *n*) **by** (*simp only*:)
then show *?thesis* **by** *auto*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *odd-pos*:
odd n ⟹ *0* < *n* **for** *n* :: *nat*
by (*auto elim*: *oddE*)

lemma *Suc-double-not-eq-double*:
Suc (*2* ∗ *m*) ≠ *2* ∗ *n*
proof
assume *Suc* (*2* ∗ *m*) = *2* ∗ *n*
moreover have *odd* (*Suc* (*2* ∗ *m*)) **and** *even* (*2* ∗ *n*)
by *simp-all*
ultimately show *False* **by** *simp*
qed

lemma *double-not-eq-Suc-double*:
2 ∗ *m* ≠ *Suc* (*2* ∗ *n*)
using *Suc-double-not-eq-double* [*of n m*] **by** *simp*

lemma *odd-Suc-minus-one* [*simp*]: *odd n* ⟹ *Suc* (*n* − *Suc 0*) = *n*
by (*auto elim*: *oddE*)

lemma *even-Suc-div-two* [*simp*]:
even n ⟹ *Suc n div 2* = *n div 2*
by *auto*

lemma *odd-Suc-div-two* [*simp*]:
odd n ⟹ *Suc n div 2* = *Suc* (*n div 2*)
by (*auto elim*: *oddE*)

lemma *odd-two-times-div-two-nat* [*simp*]:
assumes *odd n*
shows *2 * (n div 2) = n - (1 :: nat)*
proof −
from *assms* **have** *2 * (n div 2) + 1 = n*
by (*auto elim*: *oddE*)
then have *Suc (2 * (n div 2)) - 1 = n - 1*
by *simp*
then show *?thesis*
by *simp*
qed

lemma *not-mod2-eq-Suc-0-eq-0* [*simp*]:
n mod 2 ≠ Suc 0 ⟷ n mod 2 = 0
using *not-mod-2-eq-1-eq-0* [*of n*] **by** *simp*

lemma *odd-card-imp-not-empty*:
‹*A ≠ {}*› **if** ‹*odd (card A)*›
using *that* **by** *auto*

lemma *nat-induct2* [*case-names 0 1 step*]:
assumes *P 0 P 1* **and** *step*: ⋀*n*::*nat*. *P n ⟹ P (n + 2)*
shows *P n*
proof (*induct n rule*: *less-induct*)
case (*less n*)
show *?case*
proof (*cases n < Suc (Suc 0)*)
case *True*
then show *?thesis*
using *assms* **by** (*auto simp*: *less-Suc-eq*)
next
case *False*
then obtain *k* **where** *k*: *n = Suc (Suc k)*
by (*force simp*: *not-less nat-le-iff-add*)
then have *k<n*
by *simp*
with *less assms* **have** *P (k+2)*
by *blast*
then show *?thesis*
by (*simp add*: *k*)
qed
qed

lemma *mod-double-nat*:
‹*n mod (2 * m) = n mod m ∨ n mod (2 * m) = n mod m + m*›
for *m n* :: *nat*
by (*cases* ‹*even (n div m)*›)
(*simp-all add*: *mod-mult2-eq ac-simps even-iff-mod-2-eq-zero*)

context *semiring-parity*
begin

lemma *even-sum-iff*:
  ‹*even* (*sum f A*) ⟷ *even* (*card* {*a*∈*A*. *odd* (*f a*)})› **if** ‹*finite A*›
using *that* **proof** (*induction A*)
  case *empty*
  then show *?case*
    by *simp*
next
  case (*insert a A*)
  moreover have ‹{*b* ∈ *insert a A*. *odd* (*f b*)} = (*if odd* (*f a*) *then* {*a*} *else* {}) ∪
{*b* ∈ *A*. *odd* (*f b*)}›
    by *auto*
  ultimately show *?case*
    by *simp*
qed

lemma *even-mask-iff* [*simp*]:
  ‹*even* (2 ^ *n* − 1) ⟷ *n* = 0›
proof (*cases* ‹*n* = 0›)
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  then have ‹{*a*. *a* = 0 ∧ *a* < *n*} = {0}›
    by *auto*
  then show *?thesis*
    by (*auto simp add*: *mask-eq-sum-exp even-sum-iff*)
qed

lemma *even-of-nat-iff* [*simp*]:
  even (*of-nat n*) ⟷ *even n*
  by (*induction n*) *simp-all*

end

57.3 Parity and powers

context *ring-1*
begin

lemma *power-minus-even* [*simp*]: *even n* ⟹ (− *a*) ^ *n* = *a* ^ *n*
  by (*auto elim*: *evenE*)

lemma *power-minus-odd* [*simp*]: *odd n* ⟹ (− *a*) ^ *n* = − (*a* ^ *n*)
  by (*auto elim*: *oddE*)

lemma *uminus-power-if*:
(− *a*) ^ *n* = (*if even n then a* ^ *n else* − (*a* ^ *n*))
by *auto*

lemma *neg-one-even-power* [*simp*]: *even n* ⟹ (− *1*) ^ *n* = *1*
by *simp*

lemma *neg-one-odd-power* [*simp*]: *odd n* ⟹ (− *1*) ^ *n* = − *1*
by *simp*

lemma *neg-one-power-add-eq-neg-one-power-diff*: *k* ≤ *n* ⟹ (− *1*) ^ (*n* + *k*) = (− *1*) ^ (*n* − *k*)
by (*cases even* (*n* + *k*)) *auto*

lemma *minus-one-power-iff*: (− *1*) ^ *n* = (*if even n then 1 else* − *1*)
by (*induct n*) *auto*

end

context *linordered-idom*
begin

lemma *zero-le-even-power*: *even n* ⟹ *0* ≤ *a* ^ *n*
by (*auto elim*: *evenE*)

lemma *zero-le-odd-power*: *odd n* ⟹ *0* ≤ *a* ^ *n* ⟷ *0* ≤ *a*
by (*auto simp add*: *power-even-eq zero-le-mult-iff elim*: *oddE*)

lemma *zero-le-power-eq*: *0* ≤ *a* ^ *n* ⟷ *even n* ∨ *odd n* ∧ *0* ≤ *a*
by (*auto simp add*: *zero-le-even-power zero-le-odd-power*)

lemma *zero-less-power-eq*: *0* < *a* ^ *n* ⟷ *n* = *0* ∨ *even n* ∧ *a* ≠ *0* ∨ *odd n* ∧ *0* < *a*
proof −
have [*simp*]: *0* = *a* ^ *n* ⟷ *a* = *0* ∧ *n* > *0*
unfolding *power-eq-0-iff* [*of a n*, *symmetric*] **by** *blast*
show *?thesis*
unfolding *less-le zero-le-power-eq* **by** *auto*
qed

lemma *power-less-zero-eq* [*simp*]: *a* ^ *n* < *0* ⟷ *odd n* ∧ *a* < *0*
unfolding *not-le* [*symmetric*] *zero-le-power-eq* **by** *auto*

lemma *power-le-zero-eq*: *a* ^ *n* ≤ *0* ⟷ *n* > *0* ∧ (*odd n* ∧ *a* ≤ *0* ∨ *even n* ∧ *a* = *0*)
unfolding *not-less* [*symmetric*] *zero-less-power-eq* **by** *auto*

lemma *power-even-abs*: *even n* ⟹ |*a*| ^ *n* = *a* ^ *n*
using *power-abs* [*of a n*] **by** (*simp add*: *zero-le-even-power*)

lemma *power-mono-even*:
assumes *even n* **and** $|a| \leq |b|$
shows $a \hat{} n \leq b \hat{} n$
proof −
have $0 \leq |a|$ **by** *auto*
with ‹$|a| \leq |b|$› **have** $|a| \hat{} n \leq |b| \hat{} n$
by (*rule power-mono*)
with ‹*even n*› **show** *?thesis*
by (*simp add*: *power-even-abs*)
qed

lemma *power-mono-odd*:
assumes *odd n* **and** $a \leq b$
shows $a \hat{} n \leq b \hat{} n$
proof (*cases* $b < 0$)
case *True*
with ‹$a \leq b$› **have** $- b \leq - a$ **and** $0 \leq - b$ **by** *auto*
then have $(- b) \hat{} n \leq (- a) \hat{} n$ **by** (*rule power-mono*)
with ‹*odd n*› **show** *?thesis* **by** *simp*
next
case *False*
then have $0 \leq b$ **by** *auto*
show *?thesis*
proof (*cases* $a < 0$)
case *True*
then have $n \neq 0$ **and** $a \leq 0$ **using** ‹*odd n*› [*THEN odd-pos*] **by** *auto*
then have $a \hat{} n \leq 0$ **unfolding** *power-le-zero-eq* **using** ‹*odd n*› **by** *auto*
moreover from ‹$0 \leq b$› **have** $0 \leq b \hat{} n$ **by** *auto*
ultimately show *?thesis* **by** *auto*
next
case *False*
then have $0 \leq a$ **by** *auto*
with ‹$a \leq b$› **show** *?thesis*
using *power-mono* **by** *auto*
qed
qed

Simplify, when the exponent is a numeral

lemma *zero-le-power-eq-numeral* [*simp*]:
$0 \leq a \hat{} \mathit{numeral}\ w \longleftrightarrow \mathit{even}\ (\mathit{numeral}\ w :: \mathit{nat}) \vee \mathit{odd}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge 0 \leq a$
by (*fact zero-le-power-eq*)

lemma *zero-less-power-eq-numeral* [*simp*]:
$0 < a \hat{} \mathit{numeral}\ w \longleftrightarrow$
$\mathit{numeral}\ w = (0 :: \mathit{nat}) \vee$
$\mathit{even}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge a \neq 0 \vee$
$\mathit{odd}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge 0 < a$

by (*fact zero-less-power-eq*)

lemma *power-le-zero-eq-numeral* [*simp*]:
$a \mathbin{\hat{}} \mathit{numeral}\ w \leq 0 \longleftrightarrow$
$(0 :: \mathit{nat}) < \mathit{numeral}\ w \wedge$
$(\mathit{odd}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge a \leq 0 \vee \mathit{even}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge a = 0)$
by (*fact power-le-zero-eq*)

lemma *power-less-zero-eq-numeral* [*simp*]:
$a \mathbin{\hat{}} \mathit{numeral}\ w < 0 \longleftrightarrow \mathit{odd}\ (\mathit{numeral}\ w :: \mathit{nat}) \wedge a < 0$
by (*fact power-less-zero-eq*)

lemma *power-even-abs-numeral* [*simp*]:
$\mathit{even}\ (\mathit{numeral}\ w :: \mathit{nat}) \Longrightarrow |a| \mathbin{\hat{}} \mathit{numeral}\ w = a \mathbin{\hat{}} \mathit{numeral}\ w$
by (*fact power-even-abs*)

end

57.4 Instance for *int*

lemma *even-diff-iff*:
$\mathit{even}\ (k - l) \longleftrightarrow \mathit{even}\ (k + l)$ **for** $k\ l :: \mathit{int}$
by (*fact even-diff*)

lemma *even-abs-add-iff*:
$\mathit{even}\ (|k| + l) \longleftrightarrow \mathit{even}\ (k + l)$ **for** $k\ l :: \mathit{int}$
by *simp*

lemma *even-add-abs-iff*:
$\mathit{even}\ (k + |l|) \longleftrightarrow \mathit{even}\ (k + l)$ **for** $k\ l :: \mathit{int}$
by *simp*

lemma *even-nat-iff*: $0 \leq k \Longrightarrow \mathit{even}\ (\mathit{nat}\ k) \longleftrightarrow \mathit{even}\ k$
by (*simp add*: *even-of-nat-iff* [*of nat k*, **where** *?'a* = *int*, *symmetric*])

context
assumes *SORT-CONSTRAINT*('*a*::*division-ring*)
begin

lemma *power-int-minus-left*:
$\mathit{power\text{-}int}\ (-a :: 'a)\ n = (\mathit{if}\ \mathit{even}\ n\ \mathit{then}\ \mathit{power\text{-}int}\ a\ n\ \mathit{else}\ -\mathit{power\text{-}int}\ a\ n)$
by (*auto simp*: *power-int-def minus-one-power-iff even-nat-iff*)

lemma *power-int-minus-left-even* [*simp*]: $\mathit{even}\ n \Longrightarrow \mathit{power\text{-}int}\ (-a :: 'a)\ n = \mathit{power\text{-}int}\ a\ n$
by (*simp add*: *power-int-minus-left*)

lemma *power-int-minus-left-odd* [*simp*]: $\mathit{odd}\ n \Longrightarrow \mathit{power\text{-}int}\ (-a :: 'a)\ n = -\mathit{power\text{-}int}\ a\ n$

by (*simp add*: *power-int-minus-left*)

lemma *power-int-minus-left-distrib*:
NO-MATCH (−1) *x* ⟹ *power-int* (−*a* :: 'a) *n* = *power-int* (−1) *n* ∗ *power-int* *a* *n*
by (*simp add*: *power-int-minus-left*)

lemma *power-int-minus-one-minus*: *power-int* (−1 :: 'a) (−*n*) = *power-int* (−1) *n*
by (*simp add*: *power-int-minus-left*)

lemma *power-int-minus-one-diff-commute*: *power-int* (−1 :: 'a) (*a* − *b*) = *power-int* (−1) (*b* − *a*)
by (*subst power-int-minus-one-minus* [*symmetric*]) *auto*

lemma *power-int-minus-one-mult-self* [*simp*]:
power-int (−1 :: 'a) *m* ∗ *power-int* (−1) *m* = 1
by (*simp add*: *power-int-minus-left*)

lemma *power-int-minus-one-mult-self'* [*simp*]:
power-int (−1 :: 'a) *m* ∗ (*power-int* (−1) *m* ∗ *b*) = *b*
by (*simp add*: *power-int-minus-left*)

end

57.5 Special case: euclidean rings structurally containing the natural numbers

class *linordered-euclidean-semiring* = *discrete-linordered-semidom* + *unique-euclidean-semiring* +
assumes *of-nat-div*: *of-nat* (*m* *div* *n*) = *of-nat* *m* *div* *of-nat* *n*
and *division-segment-of-nat* [*simp*]: *division-segment* (*of-nat* *n*) = 1
and *division-segment-euclidean-size* [*simp*]: *division-segment* *a* ∗ *of-nat* (*euclidean-size* *a*) = *a*
begin

lemma *division-segment-eq-iff*:
a = *b* **if** *division-segment* *a* = *division-segment* *b*
and *euclidean-size* *a* = *euclidean-size* *b*
using *that* *division-segment-euclidean-size* [*of a*] **by** *simp*

lemma *euclidean-size-of-nat* [*simp*]:
euclidean-size (*of-nat* *n*) = *n*
proof −
have *division-segment* (*of-nat* *n*) ∗ *of-nat* (*euclidean-size* (*of-nat* *n*)) = *of-nat* *n*
by (*fact division-segment-euclidean-size*)
then show *?thesis* **by** *simp*
qed

lemma *of-nat-euclidean-size*:
of-nat (*euclidean-size a*) = *a div division-segment a*
proof −
have *of-nat* (*euclidean-size a*) = *division-segment a* ∗ *of-nat* (*euclidean-size a*) *div division-segment a*
by (*subst nonzero-mult-div-cancel-left*) *simp-all*
also have ... = *a div division-segment a*
by *simp*
finally show *?thesis* **.**
qed

lemma *division-segment-1* [*simp*]:
division-segment 1 = *1*
using *division-segment-of-nat* [*of 1*] **by** *simp*

lemma *division-segment-numeral* [*simp*]:
division-segment (*numeral k*) = *1*
using *division-segment-of-nat* [*of numeral k*] **by** *simp*

lemma *euclidean-size-1* [*simp*]:
euclidean-size 1 = *1*
using *euclidean-size-of-nat* [*of 1*] **by** *simp*

lemma *euclidean-size-numeral* [*simp*]:
euclidean-size (*numeral k*) = *numeral k*
using *euclidean-size-of-nat* [*of numeral k*] **by** *simp*

lemma *of-nat-dvd-iff*:
of-nat m dvd of-nat n ⟷ *m dvd n* (**is** *?P* ⟷ *?Q*)
proof (*cases m* = *0*)
case *True*
then show *?thesis*
by *simp*
next
case *False*
show *?thesis*
proof
assume *?Q*
then show *?P*
by *auto*
next
assume *?P*
with *False* **have** *of-nat n* = *of-nat n div of-nat m* ∗ *of-nat m*
by *simp*
then have *of-nat n* = *of-nat* (*n div m* ∗ *m*)
by (*simp add*: *of-nat-div*)
then have *n* = *n div m* ∗ *m*
by (*simp only*: *of-nat-eq-iff*)
then have *n* = *m* ∗ (*n div m*)

```
    by (simp add: ac-simps)
  then show ?Q ..
 qed
qed

lemma of-nat-mod:
 of-nat (m mod n) = of-nat m mod of-nat n
proof −
 have of-nat m div of-nat n ∗ of-nat n + of-nat m mod of-nat n = of-nat m
   by (simp add: div-mult-mod-eq)
 also have of-nat m = of-nat (m div n ∗ n + m mod n)
   by simp
 finally show ?thesis
   by (simp only: of-nat-div of-nat-mult of-nat-add) simp
qed

lemma one-div-two-eq-zero [simp]:
 1 div 2 = 0
proof −
 from of-nat-div [symmetric] have of-nat 1 div of-nat 2 = of-nat 0
   by (simp only:) simp
 then show ?thesis
   by simp
qed

lemma one-mod-2-pow-eq [simp]:
 1 mod (2 ^ n) = of-bool (n > 0)
proof −
 have 1 mod (2 ^ n) = of-nat (1 mod (2 ^ n))
   using of-nat-mod [of 1 2 ^ n] by simp
 also have ... = of-bool (n > 0)
   by simp
 finally show ?thesis .
qed

lemma one-div-2-pow-eq [simp]:
 1 div (2 ^ n) = of-bool (n = 0)
 using div-mult-mod-eq [of 1 2 ^ n] by auto

lemma div-mult2-eq′:
 ‹a div (of-nat m ∗ of-nat n) = a div of-nat m div of-nat n›
proof (cases ‹m = 0 ∨ n = 0›)
 case True
 then show ?thesis
   by auto
next
 case False
 then have ‹m > 0› ‹n > 0›
   by simp-all
```

show *?thesis*
proof (*cases* ‹*of-nat m * of-nat n dvd a*›)
case *True*
then obtain *b* **where** ‹*a = (of-nat m * of-nat n) * b*› **..**
then have ‹*a = of-nat m * (of-nat n * b)*›
by (*simp add*: *ac-simps*)
then show *?thesis*
by *simp*
next
case *False*
define *q* **where** ‹*q = a div (of-nat m * of-nat n)*›
define *r* **where** ‹*r = a mod (of-nat m * of-nat n)*›
from ‹*m > 0*› ‹*n > 0*› ‹¬ *of-nat m * of-nat n dvd a*› *r-def* **have** *division-segment r = 1*
using *division-segment-of-nat* [*of m * n*] **by** (*simp add*: *division-segment-mod*)
with *division-segment-euclidean-size* [*of r*]
have *of-nat* (*euclidean-size r*) = *r*
by *simp*
have *a mod* (*of-nat m * of-nat n*) *div* (*of-nat m * of-nat n*) = *0*
by *simp*
with ‹*m > 0*› ‹*n > 0*› *r-def* **have** *r div* (*of-nat m * of-nat n*) = *0*
by *simp*
with ‹*of-nat* (*euclidean-size r*) = *r*›
have *of-nat* (*euclidean-size r*) *div* (*of-nat m * of-nat n*) = *0*
by *simp*
then have *of-nat* (*euclidean-size r div* (*m * n*)) = *0*
by (*simp add*: *of-nat-div*)
then have *of-nat* (*euclidean-size r div m div n*) = *0*
by (*simp add*: *div-mult2-eq*)
with ‹*of-nat* (*euclidean-size r*) = *r*› **have** *r div of-nat m div of-nat n = 0*
by (*simp add*: *of-nat-div*)
with ‹*m > 0*› ‹*n > 0*› *q-def*
have *q* = (*r div of-nat m + q * of-nat n * of-nat m div of-nat m*) *div of-nat n*
by *simp*
moreover have ‹*a = q * (of-nat m * of-nat n) + r*›
by (*simp add*: *q-def r-def div-mult-mod-eq*)
ultimately show ‹*a div* (*of-nat m * of-nat n*) = *a div of-nat m div of-nat n*›
using *q-def* [*symmetric*] *div-plus-div-distrib-dvd-right* [*of* ‹*of-nat m*› ‹*q * (of-nat m * of-nat n)*› *r*]
by (*simp add*: *ac-simps*)
qed
qed

lemma *mod-mult2-eq'*:
a mod (*of-nat m * of-nat n*) = *of-nat m * (a div of-nat m mod of-nat n*) + *a mod of-nat m*
proof −
have *a div* (*of-nat m * of-nat n*) * (*of-nat m * of-nat n*) + *a mod* (*of-nat m * of-nat n*) = *a div of-nat m div of-nat n * of-nat n * of-nat m* + (*a div of-nat m*

mod of-nat n ∗ of-nat m + a mod of-nat m)
by (*simp add*: *combine-common-factor div-mult-mod-eq*)
moreover have *a div of-nat m div of-nat n ∗ of-nat n ∗ of-nat m = of-nat n ∗ of-nat m ∗* (*a div of-nat m div of-nat n*)
by (*simp add*: *ac-simps*)
ultimately show *?thesis*
by (*simp add*: *div-mult2-eq′ mult-commute*)
qed

lemma *div-mult2-numeral-eq*:
a div numeral k div numeral l = a div numeral (*k ∗ l*) (**is** *?A = ?B*)
proof −
have *?A = a div of-nat* (*numeral k*) *div of-nat* (*numeral l*)
by *simp*
also have ... = *a div* (*of-nat* (*numeral k*) ∗ *of-nat* (*numeral l*))
by (*fact div-mult2-eq′* [*symmetric*])
also have ... = *?B*
by *simp*
finally show *?thesis* **.**
qed

lemma *numeral-Bit0-div-2*:
numeral (*num.Bit0 n*) *div 2 = numeral n*
proof −
have *numeral* (*num.Bit0 n*) = *numeral n + numeral n*
by (*simp only*: *numeral.simps*)
also have ... = *numeral n ∗ 2*
by (*simp add*: *mult-2-right*)
finally have *numeral* (*num.Bit0 n*) *div 2 = numeral n ∗ 2 div 2*
by *simp*
also have ... = *numeral n*
by (*rule nonzero-mult-div-cancel-right*) *simp*
finally show *?thesis* **.**
qed

lemma *numeral-Bit1-div-2*:
numeral (*num.Bit1 n*) *div 2 = numeral n*
proof −
have *numeral* (*num.Bit1 n*) = *numeral n + numeral n + 1*
by (*simp only*: *numeral.simps*)
also have ... = *numeral n ∗ 2 + 1*
by (*simp add*: *mult-2-right*)
finally have *numeral* (*num.Bit1 n*) *div 2 =* (*numeral n ∗ 2 + 1*) *div 2*
by *simp*
also have ... = *numeral n ∗ 2 div 2 + 1 div 2*
using *dvd-triv-right* **by** (*rule div-plus-div-distrib-dvd-left*)
also have ... = *numeral n ∗ 2 div 2*
by *simp*
also have ... = *numeral n*

```
    by (rule nonzero-mult-div-cancel-right) simp
  finally show ?thesis .
qed

lemma exp-mod-exp:
  ‹2 ^ m mod 2 ^ n = of-bool (m < n) * 2 ^ m›
proof −
  have ‹(2::nat) ^ m mod 2 ^ n = of-bool (m < n) * 2 ^ m› (is ‹?lhs = ?rhs›)
    by (auto simp add: linorder-class.not-less monoid-mult-class.power-add dest!:
le-Suc-ex)
  then have ‹of-nat ?lhs = of-nat ?rhs›
    by simp
  then show ?thesis
    by (simp add: of-nat-mod)
qed

lemma mask-mod-exp:
  ‹(2 ^ n − 1) mod 2 ^ m = 2 ^ min m n − 1›
proof −
  have ‹(2 ^ n − 1) mod 2 ^ m = 2 ^ min m n − (1::nat)› (is ‹?lhs = ?rhs›)
  proof (cases ‹n ≤ m›)
    case True
    then show ?thesis
      by (simp add: Suc-le-lessD)
  next
    case False
    then have ‹m < n›
      by simp
    then obtain q where n: ‹n = Suc q + m›
      by (auto dest: less-imp-Suc-add)
    then have ‹min m n = m›
      by simp
    moreover have ‹(2::nat) ^ m ≤ 2 * 2 ^ q * 2 ^ m›
      using mult-le-mono1 [of 1 ‹2 * 2 ^ q› ‹2 ^ m›] by simp
    with n have ‹2 ^ n − 1 = (2 ^ Suc q − 1) * 2 ^ m + (2 ^ m − (1::nat))›
      by (simp add: monoid-mult-class.power-add algebra-simps)
    ultimately show ?thesis
      by (simp only: euclidean-semiring-cancel-class.mod-mult-self3) simp
  qed
  then have ‹of-nat ?lhs = of-nat ?rhs›
    by simp
  then show ?thesis
    by (simp add: of-nat-mod of-nat-diff)
qed

lemma of-bool-half-eq-0 [simp]:
  ‹of-bool b div 2 = 0›
  by simp
```

lemma *of-nat-mod-double*:
  ‹*of-nat n mod* (*2* ∗ *of-nat m*) = *of-nat n mod of-nat m* ∨ *of-nat n mod* (*2* ∗ *of-nat m*) = *of-nat n mod of-nat m* + *of-nat m*›
 by (*simp add*: *mod-double-nat flip*: *of-nat-mod of-nat-add of-nat-mult of-nat-numeral*)

end

instance *nat* :: *linordered-euclidean-semiring*
 by *standard simp-all*

instance *int* :: *linordered-euclidean-semiring*
 by *standard* (*auto simp add*: *divide-int-def division-segment-int-def elim*: *contrapos-np*)

context *linordered-euclidean-semiring*
begin

subclass *semiring-parity*
proof
 show ‹*a mod 2* = *of-bool* (¬ *2 dvd a*)› **for** *a*
 proof (*cases* ‹*2 dvd a*›)
  case *True*
  then show *?thesis*
   by (*simp add*: *dvd-eq-mod-eq-0*)
 next
  case *False*
  have *eucl*: *euclidean-size* (*a mod 2*) = *1*
  proof (*rule Orderings.order-antisym*)
   show *euclidean-size* (*a mod 2*) ≤ *1*
    using *mod-size-less* [*of 2 a*] **by** *simp*
   show *1* ≤ *euclidean-size* (*a mod 2*)
    using ‹¬ *2 dvd a*› **by** (*simp add*: *Suc-le-eq dvd-eq-mod-eq-0*)
  qed
  from ‹¬ *2 dvd a*› **have** ¬ *of-nat 2 dvd division-segment a* ∗ *of-nat* (*euclidean-size a*)
   by *simp*
  then have ¬ *of-nat 2 dvd of-nat* (*euclidean-size a*)
   by (*auto simp only*: *dvd-mult-unit-iff' is-unit-division-segment*)
  then have ¬ *2 dvd euclidean-size a*
   using *of-nat-dvd-iff* [*of 2*] **by** *simp*
  then have *euclidean-size a mod 2* = *1*
   by (*simp add*: *semidom-modulo-class.dvd-eq-mod-eq-0*)
  then have *of-nat* (*euclidean-size a mod 2*) = *of-nat 1*
   by *simp*
  then have *of-nat* (*euclidean-size a*) *mod 2* = *1*
   by (*simp add*: *of-nat-mod*)
  from ‹¬ *2 dvd a*› *eucl*
  have *a mod 2* = *1*
   by (*auto intro*: *division-segment-eq-iff simp add*: *division-segment-mod*)

```
    with ‹¬ 2 dvd a› show ?thesis
      by simp
  qed
  show ‹¬ is-unit 2›
  proof
    assume ‹is-unit 2›
    then have ‹of-nat 2 dvd of-nat 1›
      by simp
    then have ‹is-unit (2::nat)›
      by (simp only: of-nat-dvd-iff)
    then show False
      by simp
  qed
  show ‹(1 + a) div 2 = a div 2› if ‹2 dvd a› for a
    using that by auto
qed

lemma even-succ-div-two [simp]:
  even a ⟹ (a + 1) div 2 = a div 2
  by (cases a = 0) (auto elim!: evenE dest: mult-not-zero)

lemma odd-succ-div-two [simp]:
  odd a ⟹ (a + 1) div 2 = a div 2 + 1
  by (auto elim!: oddE simp add: add.assoc)

lemma even-two-times-div-two:
  even a ⟹ 2 * (a div 2) = a
  by (fact dvd-mult-div-cancel)

lemma odd-two-times-div-two-succ [simp]:
  odd a ⟹ 2 * (a div 2) + 1 = a
  using mult-div-mod-eq [of 2 a]
  by (simp add: even-iff-mod-2-eq-zero)

lemma coprime-left-2-iff-odd [simp]:
  coprime 2 a ⟷ odd a
proof
  assume odd a
  show coprime 2 a
  proof (rule coprimeI)
    fix b
    assume b dvd 2 b dvd a
    then have b dvd a mod 2
      by (auto intro: dvd-mod)
    with ‹odd a› show is-unit b
      by (simp add: mod-2-eq-odd)
  qed
next
  assume coprime 2 a
```

```
  show odd a
  proof (rule notI)
    assume even a
    then obtain b where a = 2 * b ..
    with ‹coprime 2 a› have coprime 2 (2 * b)
      by simp
    moreover have ¬ coprime 2 (2 * b)
      by (rule not-coprimeI [of 2]) simp-all
    ultimately show False
      by blast
  qed
qed

lemma coprime-right-2-iff-odd [simp]:
  coprime a 2 ⟷ odd a
  using coprime-left-2-iff-odd [of a] by (simp add: ac-simps)

end

lemma minus-1-mod-2-eq [simp]:
  ‹- 1 mod 2 = (1::int)›
  by (simp add: mod-2-eq-odd)

lemma minus-1-div-2-eq [simp]:
  - 1 div 2 = - (1::int)
proof -
  from div-mult-mod-eq [of - 1 :: int 2]
  have - 1 div 2 * 2 = - 1 * (2 :: int)
    using add-implies-diff by fastforce
  then show ?thesis
    using mult-right-cancel [of 2 - 1 div 2 - (1 :: int)] by simp
qed

context linordered-euclidean-semiring
begin

lemma even-decr-exp-div-exp-iff':
  ‹even ((2 ^ m - 1) div 2 ^ n) ⟷ m ≤ n›
proof -
  have ‹even ((2 ^ m - 1) div 2 ^ n) ⟷ even (of-nat ((2 ^ m - Suc 0) div 2 ^
n))›
    by (simp only: of-nat-div) (simp add: of-nat-diff)
  also have ‹... ⟷ even ((2 ^ m - Suc 0) div 2 ^ n)›
    by simp
  also have ‹... ⟷ m ≤ n›
  proof (cases ‹m ≤ n›)
    case True
    then show ?thesis
      by (simp add: Suc-le-lessD)
```

next
case *False*
then obtain *r* **where** *r*: ‹*m* = *n* + *Suc r*›
using *less-imp-Suc-add* **by** *fastforce*
from *r* **have** ‹{*q*. *q* < *m*} ∩ {*q*. 2 ^ *n* dvd (2::*nat*) ^ *q*} = {*q*. *n* ≤ *q* ∧ *q* < *m*}›
by (*auto simp add*: *dvd-power-iff-le*)
moreover from *r* **have** ‹{*q*. *q* < *m*} ∩ {*q*. ¬ 2 ^ *n* dvd (2::*nat*) ^ *q*} = {*q*. *q* < *n*}›
by (*auto simp add*: *dvd-power-iff-le*)
moreover from *False* **have** ‹{*q*. *n* ≤ *q* ∧ *q* < *m* ∧ *q* ≤ *n*} = {*n*}›
by *auto*
then have ‹*odd* ((∑ *a*∈{*q*. *n* ≤ *q* ∧ *q* < *m*}. 2 ^ *a* *div* (2::*nat*) ^ *n*) + *sum* ((^) 2) {*q*. *q* < *n*} *div* 2 ^ *n*)›
by (*simp-all add*: *euclidean-semiring-cancel-class.power-diff-power-eq semiring-parity-class.even-sum-iff*
linorder-class.not-less mask-eq-sum-exp-nat [*symmetric*])
ultimately have ‹*odd* (*sum* ((^) (2::*nat*)) {*q*. *q* < *m*} *div* 2 ^ *n*)›
by (*subst euclidean-semiring-cancel-class.sum-div-partition*) *simp-all*
with *False* **show** *?thesis*
by (*simp add*: *mask-eq-sum-exp-nat*)
qed
finally show *?thesis* **.**
qed

end

57.6 Generic symbolic computations

The following type class contains everything necessary to formulate a division algorithm in ring structures with numerals, restricted to its positive segments.

class *linordered-euclidean-semiring-division* = *linordered-euclidean-semiring* +
fixes *divmod* :: ‹*num* ⇒ *num* ⇒ 'a × 'a›
and *divmod-step* :: ‹'a ⇒ 'a × 'a ⇒ 'a × 'a› — These are conceptually definitions but force generated code to be monomorphic wrt. particular instances of this class which yields a significant speedup.
assumes *divmod-def*: ‹*divmod m n* = (*numeral m div numeral n*, *numeral m mod numeral n*)›
and *divmod-step-def* [*simp*]: ‹*divmod-step l* (*q*, *r*) =
(*if euclidean-size l* ≤ *euclidean-size r then* (2 * *q* + 1, *r* − *l*)
else (2 * *q*, *r*))› — This is a formulation of one step (referring to one digit position) in school-method division: compare the dividend at the current digit position with the remainder from previous division steps and evaluate accordingly.
begin

lemma *fst-divmod*:
‹*fst* (*divmod m n*) = *numeral m div numeral n*›
by (*simp add*: *divmod-def*)

lemma *snd-divmod*:
‹*snd* (*divmod m n*) = *numeral m mod numeral n*›
by (*simp add*: *divmod-def*)

Following a formulation of school-method division. If the divisor is smaller than the dividend, terminate. If not, shift the dividend to the right until termination occurs and then reiterate single division steps in the opposite direction.

lemma *divmod-divmod-step*:
‹*divmod m n* = (*if m* < *n then* (*0*, *numeral m*)
else divmod-step (*numeral n*) (*divmod m* (*Num.Bit0 n*)))›
proof (*cases* ‹*m* < *n*›)
case *True*
then show *?thesis*
by (*simp add*: *prod-eq-iff fst-divmod snd-divmod flip*: *of-nat-numeral of-nat-div of-nat-mod*)
next
case *False*
define *r s t* **where** ‹*r* = (*numeral m* :: *nat*)› ‹*s* = (*numeral n* :: *nat*)› ‹*t* = *2* ∗ *s*›
then have ∗: ‹*numeral m* = *of-nat r*› ‹*numeral n* = *of-nat s*› ‹*numeral* (*num.Bit0 n*) = *of-nat t*›
and ‹¬ *s* ≤ *r mod s*›
by (*simp-all add*: *linorder-class.not-le*)
have *t*: ‹*2* ∗ (*r div t*) = *r div s* − *r div s mod 2*›
‹*r mod t* = *s* ∗ (*r div s mod 2*) + *r mod s*›
by (*simp add*: *Rings.minus-mod-eq-mult-div Groups.mult.commute* [*of 2*] *Euclidean-Rings.div-mult2-eq* ‹*t* = *2* ∗ *s*›)
(*use mod-mult2-eq* [*of r s 2*] **in** ‹*simp add*: *ac-simps* ‹*t* = *2* ∗ *s*››)
have *rs*: ‹*r div s mod 2* = *0* ∨ *r div s mod 2* = *Suc 0*›
by *auto*
from ‹¬ *s* ≤ *r mod s*› **have** ‹*s* ≤ *r mod t* ⟹
r div s = *Suc* (*2* ∗ (*r div t*)) ∧
r mod s = *r mod t* − *s*›
using *rs*
by (*auto simp add*: *t*)
moreover have ‹*r mod t* < *s* ⟹
r div s = *2* ∗ (*r div t*) ∧
r mod s = *r mod t*›
using *rs*
by (*auto simp add*: *t*)
ultimately show *?thesis*
by (*simp add*: *divmod-def prod-eq-iff split-def Let-def*
not-less mod-eq-0-iff-dvd Rings.mod-eq-0-iff-dvd False not-le ∗)
(*simp add*: *flip*: *of-nat-numeral of-nat-mult add.commute* [*of 1*] *of-nat-div of-nat-mod of-nat-Suc of-nat-diff*)
qed

The division rewrite proper – first, trivial results involving *1*

lemma *divmod-trivial* [*simp*]:
divmod m Num.One = (numeral m, 0)
divmod num.One (num.Bit0 n) = (0, Numeral1)
divmod num.One (num.Bit1 n) = (0, Numeral1)
using *divmod-divmod-step* [*of Num.One*] **by** (*simp-all add*: *divmod-def*)

Division by an even number is a right-shift

lemma *divmod-cancel* [*simp*]:
‹*divmod (Num.Bit0 m) (Num.Bit0 n) = (case divmod m n of (q, r) ⇒ (q, 2 * r))*› (**is** *?P*)
‹*divmod (Num.Bit1 m) (Num.Bit0 n) = (case divmod m n of (q, r) ⇒ (q, 2 * r + 1))*› (**is** *?Q*)
proof −
define *r s* **where** ‹*r = (numeral m :: nat)*› ‹*s = (numeral n :: nat)*›
then have ∗: ‹*numeral m = of-nat r*› ‹*numeral n = of-nat s*›
‹*numeral (num.Bit0 m) = of-nat (2 * r)*› ‹*numeral (num.Bit0 n) = of-nat (2 * s)*›
‹*numeral (num.Bit1 m) = of-nat (Suc (2 * r))*›
by *simp-all*
have ∗∗: ‹*Suc (2 * r) div 2 = r*›
by *simp*
show *?P* **and** *?Q*
by (*simp-all add*: *divmod-def* ∗)
(*simp-all flip*: *of-nat-numeral of-nat-div of-nat-mod of-nat-mult add.commute* [*of 1*] *of-nat-Suc*
add: *Euclidean-Rings.mod-mult-mult1 div-mult2-eq* [*of - 2*] *mod-mult2-eq* [*of - 2*] ∗∗)
qed

The really hard work

lemma *divmod-steps* [*simp*]:
divmod (num.Bit0 m) (num.Bit1 n) =
(if m ≤ n then (0, numeral (num.Bit0 m))
else divmod-step (numeral (num.Bit1 n))
(divmod (num.Bit0 m)
(num.Bit0 (num.Bit1 n))))
divmod (num.Bit1 m) (num.Bit1 n) =
(if m < n then (0, numeral (num.Bit1 m))
else divmod-step (numeral (num.Bit1 n))
(divmod (num.Bit1 m)
(num.Bit0 (num.Bit1 n))))
by (*simp-all add*: *divmod-divmod-step*)

lemmas *divmod-algorithm-code = divmod-trivial divmod-cancel divmod-steps*

Special case: divisibility

definition *divides-aux* :: *'a × 'a ⇒ bool*

where
divides-aux qr ⟷ *snd qr = 0*

lemma *divides-aux-eq* [*simp*]:
divides-aux (*q*, *r*) ⟷ *r = 0*
by (*simp add*: *divides-aux-def*)

lemma *dvd-numeral-simp* [*simp*]:
numeral m dvd numeral n ⟷ *divides-aux* (*divmod n m*)
by (*simp add*: *divmod-def mod-eq-0-iff-dvd*)

Generic computation of quotient and remainder

lemma *numeral-div-numeral* [*simp*]:
numeral k div numeral l = fst (*divmod k l*)
by (*simp add*: *fst-divmod*)

lemma *numeral-mod-numeral* [*simp*]:
numeral k mod numeral l = snd (*divmod k l*)
by (*simp add*: *snd-divmod*)

lemma *one-div-numeral* [*simp*]:
1 div numeral n = fst (*divmod num.One n*)
by (*simp add*: *fst-divmod*)

lemma *one-mod-numeral* [*simp*]:
1 mod numeral n = snd (*divmod num.One n*)
by (*simp add*: *snd-divmod*)

end

instantiation *nat* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-nat* :: *num* ⇒ *num* ⇒ *nat* × *nat*
where
divmod'-nat-def: *divmod-nat m n* = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-nat* :: *nat* ⇒ *nat* × *nat* ⇒ *nat* × *nat*
where
divmod-step-nat l qr = (*let* (*q*, *r*) = *qr*
in if r ≥ *l then* (*2* ∗ *q* + *1*, *r* − *l*)
else (*2* ∗ *q*, *r*))

instance
by *standard* (*simp-all add*: *divmod'-nat-def divmod-step-nat-def*)

end

declare *divmod-algorithm-code* [**where** *?'a = nat*, *code*]

lemma *Suc-0-div-numeral* [*simp*]:
‹*Suc 0 div numeral Num.One = 1*›
‹*Suc 0 div numeral (Num.Bit0 n) = 0*›
‹*Suc 0 div numeral (Num.Bit1 n) = 0*›
by *simp-all*

lemma *Suc-0-mod-numeral* [*simp*]:
‹*Suc 0 mod numeral Num.One = 0*›
‹*Suc 0 mod numeral (Num.Bit0 n) = 1*›
‹*Suc 0 mod numeral (Num.Bit1 n) = 1*›
by *simp-all*

instantiation *int* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-int* :: *num* ⇒ *num* ⇒ *int* × *int*
where
divmod-int m n = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-int* :: *int* ⇒ *int* × *int* ⇒ *int* × *int*
where
divmod-step-int l qr = (*let* (*q*, *r*) = *qr*
in if |*l*| ≤ |*r*| *then* (2 ∗ *q* + 1, *r* − *l*)
else (2 ∗ *q*, *r*))

instance
by *standard* (*auto simp add*: *divmod-int-def divmod-step-int-def*)

end

declare *divmod-algorithm-code* [**where** *?'a = int*, *code*]

context
begin

qualified definition *adjust-div* :: *int* × *int* ⇒ *int*
where
adjust-div qr = (*let* (*q*, *r*) = *qr in q* + *of-bool* (*r* ≠ 0))

qualified lemma *adjust-div-eq* [*simp*, *code*]:
adjust-div (*q*, *r*) = *q* + *of-bool* (*r* ≠ 0)
by (*simp add*: *adjust-div-def*)

qualified definition *adjust-mod* :: *num* ⇒ *int* ⇒ *int*
where
[*simp*]: *adjust-mod l r* = (*if r* = 0 *then* 0 *else numeral l* − *r*)

lemma *minus-numeral-div-numeral* [*simp*]:
  − *numeral m div numeral n* = − (*adjust-div* (*divmod m n*) :: *int*)
proof −
  have *int* (*fst* (*divmod m n*)) = *fst* (*divmod m n*)
    by (*simp only*: *fst-divmod divide-int-def*) *auto*
  then show *?thesis*
   by (*auto simp add*: *split-def Let-def adjust-div-def divides-aux-def divide-int-def*)
qed

lemma *minus-numeral-mod-numeral* [*simp*]:
  − *numeral m mod numeral n* = *adjust-mod n* (*snd* (*divmod m n*) :: *int*)
proof (*cases snd* (*divmod m n*) = (*0*::*int*))
  case *True*
  then show *?thesis*
    by (*simp add*: *mod-eq-0-iff-dvd divides-aux-def*)
next
  case *False*
  then have *int* (*snd* (*divmod m n*)) = *snd* (*divmod m n*) **if** *snd* (*divmod m n*) ≠ (*0*::*int*)
    by (*simp only*: *snd-divmod modulo-int-def*) *auto*
  then show *?thesis*
    by (*simp add*: *divides-aux-def adjust-div-def*)
      (*simp add*: *divides-aux-def modulo-int-def*)
qed

lemma *numeral-div-minus-numeral* [*simp*]:
  numeral m div − *numeral n* = − (*adjust-div* (*divmod m n*) :: *int*)
proof −
  have *int* (*fst* (*divmod m n*)) = *fst* (*divmod m n*)
    by (*simp only*: *fst-divmod divide-int-def*) *auto*
  then show *?thesis*
   by (*auto simp add*: *split-def Let-def adjust-div-def divides-aux-def divide-int-def*)
qed

lemma *numeral-mod-minus-numeral* [*simp*]:
  numeral m mod − *numeral n* = − *adjust-mod n* (*snd* (*divmod m n*) :: *int*)
proof (*cases snd* (*divmod m n*) = (*0*::*int*))
  case *True*
  then show *?thesis*
    by (*simp add*: *mod-eq-0-iff-dvd divides-aux-def*)
next
  case *False*
  then have *int* (*snd* (*divmod m n*)) = *snd* (*divmod m n*) **if** *snd* (*divmod m n*) ≠ (*0*::*int*)
    by (*simp only*: *snd-divmod modulo-int-def*) *auto*
  then show *?thesis*
    by (*simp add*: *divides-aux-def adjust-div-def*)
      (*simp add*: *divides-aux-def modulo-int-def*)
qed

lemma *minus-one-div-numeral* [*simp*]:
− 1 div numeral n = − (adjust-div (divmod Num.One n) :: int)
using *minus-numeral-div-numeral* [*of Num.One n*] **by** *simp*

lemma *minus-one-mod-numeral* [*simp*]:
− 1 mod numeral n = adjust-mod n (snd (divmod Num.One n) :: int)
using *minus-numeral-mod-numeral* [*of Num.One n*] **by** *simp*

lemma *one-div-minus-numeral* [*simp*]:
1 div − numeral n = − (adjust-div (divmod Num.One n) :: int)
using *numeral-div-minus-numeral* [*of Num.One n*] **by** *simp*

lemma *one-mod-minus-numeral* [*simp*]:
1 mod − numeral n = − adjust-mod n (snd (divmod Num.One n) :: int)
using *numeral-mod-minus-numeral* [*of Num.One n*] **by** *simp*

lemma [*code*]:
fixes *k :: int*
shows
k div 0 = 0
k mod 0 = k
0 div k = 0
0 mod k = 0
k div Int.Pos Num.One = k
k mod Int.Pos Num.One = 0
k div Int.Neg Num.One = − k
k mod Int.Neg Num.One = 0
Int.Pos m div Int.Pos n = (fst (divmod m n) :: int)
Int.Pos m mod Int.Pos n = (snd (divmod m n) :: int)
Int.Neg m div Int.Pos n = − (adjust-div (divmod m n) :: int)
Int.Neg m mod Int.Pos n = adjust-mod n (snd (divmod m n) :: int)
Int.Pos m div Int.Neg n = − (adjust-div (divmod m n) :: int)
Int.Pos m mod Int.Neg n = − adjust-mod n (snd (divmod m n) :: int)
Int.Neg m div Int.Neg n = (fst (divmod m n) :: int)
Int.Neg m mod Int.Neg n = − (snd (divmod m n) :: int)
by *simp-all*

end

lemma *divmod-BitM-2-eq* [*simp*]:
‹divmod (Num.BitM m) (Num.Bit0 Num.One) = (numeral m − 1, (1 :: int))›
by (*cases m*) *simp-all*

57.6.1 Computation by simplification

lemma *euclidean-size-nat-less-eq-iff*:
‹euclidean-size m ≤ euclidean-size n ⟷ m ≤ n› **for** *m n :: nat*
by *simp*

lemma *euclidean-size-int-less-eq-iff*:
‹*euclidean-size k* ≤ *euclidean-size l* ⟷ |*k*| ≤ |*l*|› **for** *k l* :: *int*
by *auto*

simproc-setup *numeral-divmod*
(*0 div 0* :: *'a* :: *linordered-euclidean-semiring-division* | *0 mod 0* :: *'a* :: *linordered-euclidean-semiring-division*
|
0 div 1 :: *'a* :: *linordered-euclidean-semiring-division* | *0 mod 1* :: *'a* :: *linordered-euclidean-semiring-division*
|
0 div − 1 :: *int* | *0 mod − 1* :: *int* |
0 div numeral b :: *'a* :: *linordered-euclidean-semiring-division* | *0 mod numeral b*
:: *'a* :: *linordered-euclidean-semiring-division* |
0 div − numeral b :: *int* | *0 mod − numeral b* :: *int* |
1 div 0 :: *'a* :: *linordered-euclidean-semiring-division* | *1 mod 0* :: *'a* :: *linordered-euclidean-semiring-division*
|
1 div 1 :: *'a* :: *linordered-euclidean-semiring-division* | *1 mod 1* :: *'a* :: *linordered-euclidean-semiring-division*
|
1 div − 1 :: *int* | *1 mod − 1* :: *int* |
1 div numeral b :: *'a* :: *linordered-euclidean-semiring-division* | *1 mod numeral b*
:: *'a* :: *linordered-euclidean-semiring-division* |
1 div − numeral b :: *int* |*1 mod − numeral b* :: *int* |
− 1 div 0 :: *int* | *− 1 mod 0* :: *int* | *− 1 div 1* :: *int* | *− 1 mod 1* :: *int* |
− 1 div − 1 :: *int* | *− 1 mod − 1* :: *int* | *− 1 div numeral b* :: *int* | *− 1 mod*
numeral b :: *int* |
− 1 div − numeral b :: *int* | *− 1 mod − numeral b* :: *int* |
numeral a div 0 :: *'a* :: *linordered-euclidean-semiring-division* | *numeral a mod*
0 :: *'a* :: *linordered-euclidean-semiring-division* |
numeral a div 1 :: *'a* :: *linordered-euclidean-semiring-division* | *numeral a mod*
1 :: *'a* :: *linordered-euclidean-semiring-division* |
numeral a div − 1 :: *int* | *numeral a mod − 1* :: *int* |
numeral a div numeral b :: *'a* :: *linordered-euclidean-semiring-division* | *numeral*
a mod numeral b :: *'a* :: *linordered-euclidean-semiring-division* |
numeral a div − numeral b :: *int* | *numeral a mod − numeral b* :: *int* |
− numeral a div 0 :: *int* | *− numeral a mod 0* :: *int* |
− numeral a div 1 :: *int* | *− numeral a mod 1* :: *int* |
− numeral a div − 1 :: *int* | *− numeral a mod − 1* :: *int* |
− numeral a div numeral b :: *int* | *− numeral a mod numeral b* :: *int* |
− numeral a div − numeral b :: *int* | *− numeral a mod − numeral b* :: *int*) = ‹
let
val if-cong = *the* (*Code.get-case-cong* ***theory const-name*** ‹*If*›);
fun successful-rewrite ctxt ct =
let
val thm = *Simplifier.rewrite ctxt ct*
in if Thm.is-reflexive thm then NONE else SOME thm end;
val simps = @{*thms div-0 mod-0 div-by-0 mod-by-0 div-by-1 mod-by-1*
one-div-numeral one-mod-numeral minus-one-div-numeral minus-one-mod-numeral
one-div-minus-numeral one-mod-minus-numeral
numeral-div-numeral numeral-mod-numeral minus-numeral-div-numeral mi-

nus-numeral-mod-numeral
numeral-div-minus-numeral numeral-mod-minus-numeral
div-minus-minus mod-minus-minus Parity.adjust-div-eq of-bool-eq one-neq-zero
numeral-neq-zero neg-equal-0-iff-equal arith-simps arith-special divmod-trivial
divmod-cancel divmod-steps divmod-step-def fst-conv snd-conv numeral-One
case-prod-beta rel-simps Parity.adjust-mod-def div-minus1-right mod-minus1-right
minus-minus numeral-times-numeral mult-zero-right mult-1-right
euclidean-size-nat-less-eq-iff euclidean-size-int-less-eq-iff diff-nat-numeral nat-numeral}
@ [@{*lemma 0 = 0 ⟷ True by simp*}];
val simpset =
HOL-ss |> Simplifier.simpset-map ***context***
(*Simplifier.add-cong if-cong #> fold Simplifier.add-simp simps*);
in K (*fn ctxt => successful-rewrite* (*Simplifier.put-simpset simpset ctxt*)) *end*
› — There is space for improvement here: the calculation itself could be carried out outside the logic, and a generic simproc (simplifier setup) for generic calculation would be helpful.

57.7 Computing congruences modulo *2 ^ q*

context *linordered-euclidean-semiring-division*
begin

lemma *cong-exp-iff-simps*:
numeral n mod numeral Num.One = 0
⟷ *True*
numeral (*Num.Bit0 n*) *mod numeral* (*Num.Bit0 q*) = *0*
⟷ *numeral n mod numeral q = 0*
numeral (*Num.Bit1 n*) *mod numeral* (*Num.Bit0 q*) = *0*
⟷ *False*
numeral m mod numeral Num.One = (*numeral n mod numeral Num.One*)
⟷ *True*
numeral Num.One mod numeral (*Num.Bit0 q*) = (*numeral Num.One mod numeral* (*Num.Bit0 q*))
⟷ *True*
numeral Num.One mod numeral (*Num.Bit0 q*) = (*numeral* (*Num.Bit0 n*) *mod numeral* (*Num.Bit0 q*))
⟷ *False*
numeral Num.One mod numeral (*Num.Bit0 q*) = (*numeral* (*Num.Bit1 n*) *mod numeral* (*Num.Bit0 q*))
⟷ (*numeral n mod numeral q*) = *0*
numeral (*Num.Bit0 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral Num.One mod numeral* (*Num.Bit0 q*))
⟷ *False*
numeral (*Num.Bit0 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral* (*Num.Bit0 n*) *mod numeral* (*Num.Bit0 q*))
⟷ *numeral m mod numeral q* = (*numeral n mod numeral q*)
numeral (*Num.Bit0 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral* (*Num.Bit1 n*) *mod numeral* (*Num.Bit0 q*))
⟷ *False*

numeral (*Num.Bit1 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral Num.One mod numeral* (*Num.Bit0 q*))
⟷ (*numeral m mod numeral q*) = *0*
numeral (*Num.Bit1 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral* (*Num.Bit0 n*) *mod numeral* (*Num.Bit0 q*))
⟷ *False*
numeral (*Num.Bit1 m*) *mod numeral* (*Num.Bit0 q*) = (*numeral* (*Num.Bit1 n*) *mod numeral* (*Num.Bit0 q*))
⟷ *numeral m mod numeral q* = (*numeral n mod numeral q*)
by (*auto simp add*: *case-prod-beta dest*: *arg-cong* [*of* - - *even*])

end

code-identifier
code-module *Parity* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

lemmas *even-of-nat* = *even-of-nat-iff*

end

58 Combination and Cancellation Simprocs for Numeral Expressions

theory *Numeral-Simprocs*
imports *Parity*
begin

ML-file ‹~~/*src/Provers/Arith/assoc-fold.ML*›
ML-file ‹~~/*src/Provers/Arith/cancel-numerals.ML*›
ML-file ‹~~/*src/Provers/Arith/combine-numerals.ML*›
ML-file ‹~~/*src/Provers/Arith/cancel-numeral-factor.ML*›
ML-file ‹~~/*src/Provers/Arith/extract-common-term.ML*›

lemmas *semiring-norm* =
Let-def arith-simps diff-nat-numeral rel-simps
if-False if-True
add-Suc add-numeral-left
add-neg-numeral-left mult-numeral-left
numeral-One [*symmetric*] *uminus-numeral-One* [*symmetric*] *Suc-eq-plus1*
eq-numeral-iff-iszero not-iszero-Numeral1

For *combine-numerals*

lemma *left-add-mult-distrib*: *i*u* + (*j*u* + *k*) = (*i*+*j*)*u* + (*k*::*nat*)
by (*simp add*: *add-mult-distrib*)

For *cancel-numerals*

lemma *nat-diff-add-eq1*:

$j <= (i::nat) ==> ((i*u + m) - (j*u + n)) = (((i-j)*u + m) - n)$
by (*simp split*: *nat-diff-split add*: *add-mult-distrib*)

lemma *nat-diff-add-eq2*:
$i <= (j::nat) ==> ((i*u + m) - (j*u + n)) = (m - ((j-i)*u + n))$
by (*simp split*: *nat-diff-split add*: *add-mult-distrib*)

lemma *nat-eq-add-iff1*:
$j <= (i::nat) ==> (i*u + m = j*u + n) = ((i-j)*u + m = n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-eq-add-iff2*:
$i <= (j::nat) ==> (i*u + m = j*u + n) = (m = (j-i)*u + n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-less-add-iff1*:
$j <= (i::nat) ==> (i*u + m < j*u + n) = ((i-j)*u + m < n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-less-add-iff2*:
$i <= (j::nat) ==> (i*u + m < j*u + n) = (m < (j-i)*u + n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-le-add-iff1*:
$j <= (i::nat) ==> (i*u + m <= j*u + n) = ((i-j)*u + m <= n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

lemma *nat-le-add-iff2*:
$i <= (j::nat) ==> (i*u + m <= j*u + n) = (m <= (j-i)*u + n)$
by (*auto split*: *nat-diff-split simp add*: *add-mult-distrib*)

For *cancel-numeral-factors*

lemma *nat-mult-le-cancel1*: $(0::nat) < k ==> (k*m <= k*n) = (m<=n)$
by *auto*

lemma *nat-mult-less-cancel1*: $(0::nat) < k ==> (k*m < k*n) = (m<n)$
by *auto*

lemma *nat-mult-eq-cancel1*: $(0::nat) < k ==> (k*m = k*n) = (m=n)$
by *auto*

lemma *nat-mult-div-cancel1*: $(0::nat) < k ==> (k*m)\ div\ (k*n) = (m\ div\ n)$
by *auto*

lemma *nat-mult-dvd-cancel-disj*[*simp*]:
$(k*m)\ dvd\ (k*n) = (k=0 \vee m\ dvd\ (n::nat))$
by (*auto simp*: *dvd-eq-mod-eq-0 mod-mult-mult1*)

lemma *nat-mult-dvd-cancel1*: $0 < k \Longrightarrow (k*m)\ dvd\ (k*n::nat) = (m\ dvd\ n)$

by(*auto*)

For *cancel-factor*

lemmas *nat-mult-le-cancel-disj* = *mult-le-cancel1*

lemmas *nat-mult-less-cancel-disj* = *mult-less-cancel1*

lemma *nat-mult-eq-cancel-disj*:
fixes *k m n* :: *nat*
shows $k * m = k * n \longleftrightarrow k = 0 \vee m = n$
by (*fact mult-cancel-left*)

lemma *nat-mult-div-cancel-disj*:
fixes *k m n* :: *nat*
shows $(k * m)\ div\ (k * n) = (if\ k = 0\ then\ 0\ else\ m\ div\ n)$
by (*fact div-mult-mult1-if*)

lemma *numeral-times-minus-swap*:
fixes *x*:: *'a*::*comm-ring-1* **shows** $numeral\ w * -x = x * -\ numeral\ w$
by (*simp add*: *ac-simps*)

lemma *minus-times-numeral-swap*:
fixes *x*:: *'a*::*comm-ring-1* **shows** $-x * numeral\ w = x * -\ numeral\ w$
by (*simp add*: *ac-simps*)

ML-file ‹*Tools/numeral-simprocs.ML*›

simproc-setup *semiring-assoc-fold*
$((a::'a::comm\text{-}semiring\text{-}1\text{-}cancel) * b) =$
‹*K Numeral-Simprocs.assoc-fold*›

simproc-setup *int-combine-numerals*
$((i::'a::comm\text{-}ring\text{-}1) + j \mid (i::'a::comm\text{-}ring\text{-}1) - j) =$
‹*K Numeral-Simprocs.combine-numerals*›

simproc-setup *field-combine-numerals*
$((i::'a::\{field,ring\text{-}char\text{-}0\}) + j$
$\mid (i::'a::\{field,ring\text{-}char\text{-}0\}) - j) =$
‹*K Numeral-Simprocs.field-combine-numerals*›

simproc-setup *inteq-cancel-numerals*
$((l::'a::comm\text{-}ring\text{-}1) + m = n$
$\mid (l::'a::comm\text{-}ring\text{-}1) = m + n$
$\mid (l::'a::comm\text{-}ring\text{-}1) - m = n$
$\mid (l::'a::comm\text{-}ring\text{-}1) = m - n$
$\mid (l::'a::comm\text{-}ring\text{-}1) * m = n$
$\mid (l::'a::comm\text{-}ring\text{-}1) = m * n$
$\mid -\ (l::'a::comm\text{-}ring\text{-}1) = m$

$|(l::'a::comm\text{-}ring\text{-}1) = - m) =$
$\langle K\ Numeral\text{-}Simprocs.eq\text{-}cancel\text{-}numerals\rangle$

simproc-setup *intless-cancel-numerals*
$((l::'a::linordered\text{-}idom) + m < n$
$|(l::'a::linordered\text{-}idom) < m + n$
$|(l::'a::linordered\text{-}idom) - m < n$
$|(l::'a::linordered\text{-}idom) < m - n$
$|(l::'a::linordered\text{-}idom) * m < n$
$|(l::'a::linordered\text{-}idom) < m * n$
$|- (l::'a::linordered\text{-}idom) < m$
$|(l::'a::linordered\text{-}idom) < - m) =$
$\langle K\ Numeral\text{-}Simprocs.less\text{-}cancel\text{-}numerals\rangle$

simproc-setup *intle-cancel-numerals*
$((l::'a::linordered\text{-}idom) + m \leq n$
$|(l::'a::linordered\text{-}idom) \leq m + n$
$|(l::'a::linordered\text{-}idom) - m \leq n$
$|(l::'a::linordered\text{-}idom) \leq m - n$
$|(l::'a::linordered\text{-}idom) * m \leq n$
$|(l::'a::linordered\text{-}idom) \leq m * n$
$|- (l::'a::linordered\text{-}idom) \leq m$
$|(l::'a::linordered\text{-}idom) \leq - m) =$
$\langle K\ Numeral\text{-}Simprocs.le\text{-}cancel\text{-}numerals\rangle$

simproc-setup *ring-eq-cancel-numeral-factor*
$((l::'a::\{idom,ring\text{-}char\text{-}0\}) * m = n$
$|(l::'a::\{idom,ring\text{-}char\text{-}0\}) = m * n) =$
$\langle K\ Numeral\text{-}Simprocs.eq\text{-}cancel\text{-}numeral\text{-}factor\rangle$

simproc-setup *ring-less-cancel-numeral-factor*
$((l::'a::linordered\text{-}idom) * m < n$
$|(l::'a::linordered\text{-}idom) < m * n) =$
$\langle K\ Numeral\text{-}Simprocs.less\text{-}cancel\text{-}numeral\text{-}factor\rangle$

simproc-setup *ring-le-cancel-numeral-factor*
$((l::'a::linordered\text{-}idom) * m <= n$
$|(l::'a::linordered\text{-}idom) <= m * n) =$
$\langle K\ Numeral\text{-}Simprocs.le\text{-}cancel\text{-}numeral\text{-}factor\rangle$

simproc-setup *int-div-cancel-numeral-factors*
$(((l::'a::\{euclidean\text{-}semiring\text{-}cancel,comm\text{-}ring\text{-}1,ring\text{-}char\text{-}0\}) * m)\ div\ n$
$|(l::'a::\{euclidean\text{-}semiring\text{-}cancel,comm\text{-}ring\text{-}1,ring\text{-}char\text{-}0\})\ div\ (m * n)) =$
$\langle K\ Numeral\text{-}Simprocs.div\text{-}cancel\text{-}numeral\text{-}factor\rangle$

simproc-setup *divide-cancel-numeral-factor*
$(((l::'a::\{field,ring\text{-}char\text{-}0\}) * m) / n$
$|(l::'a::\{field,ring\text{-}char\text{-}0\}) / (m * n)$

|((*numeral v*)::'*a*::{*field*,*ring-char-0*}) / (*numeral w*)) =
‹*K Numeral-Simprocs.divide-cancel-numeral-factor*›

simproc-setup *ring-eq-cancel-factor*
((*l*::'*a*::*idom*) * *m* = *n* | (*l*::'*a*::*idom*) = *m* * *n*) =
‹*K Numeral-Simprocs.eq-cancel-factor*›

simproc-setup *linordered-ring-le-cancel-factor*
((*l*::'*a*::*linordered-idom*) * *m* <= *n*
|(*l*::'*a*::*linordered-idom*) <= *m* * *n*) =
‹*K Numeral-Simprocs.le-cancel-factor*›

simproc-setup *linordered-ring-less-cancel-factor*
((*l*::'*a*::*linordered-idom*) * *m* < *n*
|(*l*::'*a*::*linordered-idom*) < *m* * *n*) =
‹*K Numeral-Simprocs.less-cancel-factor*›

simproc-setup *int-div-cancel-factor*
(((*l*::'*a*::*euclidean-semiring-cancel*) * *m*) *div* *n*
|(*l*::'*a*::*euclidean-semiring-cancel*) *div* (*m* * *n*)) =
‹*K Numeral-Simprocs.div-cancel-factor*›

simproc-setup *int-mod-cancel-factor*
(((*l*::'*a*::*euclidean-semiring-cancel*) * *m*) *mod* *n*
|(*l*::'*a*::*euclidean-semiring-cancel*) *mod* (*m* * *n*)) =
‹*K Numeral-Simprocs.mod-cancel-factor*›

simproc-setup *dvd-cancel-factor*
(((*l*::'*a*::*idom*) * *m*) *dvd* *n*
|(*l*::'*a*::*idom*) *dvd* (*m* * *n*)) =
‹*K Numeral-Simprocs.dvd-cancel-factor*›

simproc-setup *divide-cancel-factor*
(((*l*::'*a*::*field*) * *m*) / *n*
|(*l*::'*a*::*field*) / (*m* * *n*)) =
‹*K Numeral-Simprocs.divide-cancel-factor*›

ML-file ‹*Tools/nat-numeral-simprocs.ML*›

simproc-setup *nat-combine-numerals*
((*i*::*nat*) + *j* | *Suc* (*i* + *j*)) =
‹*K Nat-Numeral-Simprocs.combine-numerals*›

simproc-setup *nateq-cancel-numerals*
((*l*::*nat*) + *m* = *n* | (*l*::*nat*) = *m* + *n* |
(*l*::*nat*) * *m* = *n* | (*l*::*nat*) = *m* * *n* |
Suc *m* = *n* | *m* = *Suc* *n*) =
‹*K Nat-Numeral-Simprocs.eq-cancel-numerals*›

simproc-setup *natless-cancel-numerals*
((*l*::*nat*) + *m* < *n* | (*l*::*nat*) < *m* + *n* |
(*l*::*nat*) * *m* < *n* | (*l*::*nat*) < *m* * *n* |
Suc m < *n* | *m* < *Suc n*) =
‹*K Nat-Numeral-Simprocs.less-cancel-numerals*›

simproc-setup *natle-cancel-numerals*
((*l*::*nat*) + *m* ≤ *n* | (*l*::*nat*) ≤ *m* + *n* |
(*l*::*nat*) * *m* ≤ *n* | (*l*::*nat*) ≤ *m* * *n* |
Suc m ≤ *n* | *m* ≤ *Suc n*) =
‹*K Nat-Numeral-Simprocs.le-cancel-numerals*›

simproc-setup *natdiff-cancel-numerals*
(((*l*::*nat*) + *m*) − *n* | (*l*::*nat*) − (*m* + *n*) |
(*l*::*nat*) * *m* − *n* | (*l*::*nat*) − *m* * *n* |
Suc m − *n* | *m* − *Suc n*) =
‹*K Nat-Numeral-Simprocs.diff-cancel-numerals*›

simproc-setup *nat-eq-cancel-numeral-factor*
((*l*::*nat*) * *m* = *n* | (*l*::*nat*) = *m* * *n*) =
‹*K Nat-Numeral-Simprocs.eq-cancel-numeral-factor*›

simproc-setup *nat-less-cancel-numeral-factor*
((*l*::*nat*) * *m* < *n* | (*l*::*nat*) < *m* * *n*) =
‹*K Nat-Numeral-Simprocs.less-cancel-numeral-factor*›

simproc-setup *nat-le-cancel-numeral-factor*
((*l*::*nat*) * *m* <= *n* | (*l*::*nat*) <= *m* * *n*) =
‹*K Nat-Numeral-Simprocs.le-cancel-numeral-factor*›

simproc-setup *nat-div-cancel-numeral-factor*
(((*l*::*nat*) * *m*) *div* *n* | (*l*::*nat*) *div* (*m* * *n*)) =
‹*K Nat-Numeral-Simprocs.div-cancel-numeral-factor*›

simproc-setup *nat-dvd-cancel-numeral-factor*
(((*l*::*nat*) * *m*) *dvd* *n* | (*l*::*nat*) *dvd* (*m* * *n*)) =
‹*K Nat-Numeral-Simprocs.dvd-cancel-numeral-factor*›

simproc-setup *nat-eq-cancel-factor*
((*l*::*nat*) * *m* = *n* | (*l*::*nat*) = *m* * *n*) =
‹*K Nat-Numeral-Simprocs.eq-cancel-factor*›

simproc-setup *nat-less-cancel-factor*
((*l*::*nat*) * *m* < *n* | (*l*::*nat*) < *m* * *n*) =
‹*K Nat-Numeral-Simprocs.less-cancel-factor*›

simproc-setup *nat-le-cancel-factor*
((*l*::*nat*) * *m* <= *n* | (*l*::*nat*) <= *m* * *n*) =
‹*K Nat-Numeral-Simprocs.le-cancel-factor*›

simproc-setup *nat-div-cancel-factor*
(((*l*::*nat*) * *m*) *div n* | (*l*::*nat*) *div* (*m* * *n*)) =
‹*K Nat-Numeral-Simprocs.div-cancel-factor*›

simproc-setup *nat-dvd-cancel-factor*
(((*l*::*nat*) * *m*) *dvd n* | (*l*::*nat*) *dvd* (*m* * *n*)) =
‹*K Nat-Numeral-Simprocs.dvd-cancel-factor*›

declaration ‹
K (*Lin-Arith.add-simprocs*
[***simproc*** ‹*semiring-assoc-fold*›,
simproc ‹*int-combine-numerals*›,
simproc ‹*inteq-cancel-numerals*›,
simproc ‹*intless-cancel-numerals*›,
simproc ‹*intle-cancel-numerals*›,
simproc ‹*field-combine-numerals*›,
simproc ‹*nat-combine-numerals*›,
simproc ‹*nateq-cancel-numerals*›,
simproc ‹*natless-cancel-numerals*›,
simproc ‹*natle-cancel-numerals*›,
simproc ‹*natdiff-cancel-numerals*›,
Numeral-Simprocs.field-divide-cancel-numeral-factor])
›

end

59 Semiring normalization

theory *Semiring-Normalization*
imports *Numeral-Simprocs*
begin

Prelude

class *comm-semiring-1-cancel-crossproduct* = *comm-semiring-1-cancel* +
assumes *crossproduct-eq*: *w* * *y* + *x* * *z* = *w* * *z* + *x* * *y* ⟷ *w* = *x* ∨ *y* = *z*
begin

lemma *crossproduct-noteq*:
a ≠ *b* ∧ *c* ≠ *d* ⟷ *a* * *c* + *b* * *d* ≠ *a* * *d* + *b* * *c*
by (*simp add*: *crossproduct-eq*)

lemma *add-scale-eq-noteq*:
r ≠ *0* ⟹ *a* = *b* ∧ *c* ≠ *d* ⟹ *a* + *r* * *c* ≠ *b* + *r* * *d*
proof (*rule notI*)
assume *nz*: *r*≠ *0* **and** *cnd*: *a* = *b* ∧ *c*≠*d*
and *eq*: *a* + (*r* * *c*) = *b* + (*r* * *d*)
have (*0* * *d*) + (*r* * *c*) = (*0* * *c*) + (*r* * *d*)
using *add-left-imp-eq eq mult-zero-left* **by** (*simp add*: *cnd*)

then show *False* **using** *crossproduct-eq* [*of 0 d*] *nz cnd* **by** *simp*
qed

lemma *add-0-iff*:
$b = b + a \longleftrightarrow a = 0$
using *add-left-imp-eq* [*of b a 0*] **by** *auto*

end

subclass (**in** *idom*) *comm-semiring-1-cancel-crossproduct*
proof
fix *w x y z*
show $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$
proof
assume $w * y + x * z = w * z + x * y$
then have $w * y + x * z - w * z - x * y = 0$ **by** (*simp add*: *algebra-simps*)
then have $w * (y - z) - x * (y - z) = 0$ **by** (*simp add*: *algebra-simps*)
then have $(y - z) * (w - x) = 0$ **by** (*simp add*: *algebra-simps*)
then have $y - z = 0 \vee w - x = 0$ **by** (*rule divisors-zero*)
then show $w = x \vee y = z$ **by** *auto*
qed (*auto simp add*: *ac-simps*)
qed

instance *nat* :: *comm-semiring-1-cancel-crossproduct*
proof
fix *w x y z* :: *nat*
have *aux*: $\bigwedge y\ z.\ y < z \Longrightarrow w * y + x * z = w * z + x * y \Longrightarrow w = x$
proof −
fix *y z* :: *nat*
assume $y < z$ **then have** $\exists k.\ z = y + k \wedge k \neq 0$ **by** (*intro exI* [*of - z − y*]) *auto*
then obtain *k* **where** $z = y + k$ **and** $k \neq 0$ **by** *blast*
assume $w * y + x * z = w * z + x * y$
then have $(w * y + x * y) + x * k = (w * y + x * y) + w * k$ **by** (*simp add*: ‹$z = y + k$› *algebra-simps*)
then have $x * k = w * k$ **by** *simp*
then show $w = x$ **using** ‹$k \neq 0$› **by** *simp*
qed
show $w * y + x * z = w * z + x * y \longleftrightarrow w = x \vee y = z$
by (*auto simp add*: *neq-iff dest*!: *aux*)
qed

Semiring normalization proper

ML-file ‹*Tools/semiring-normalizer.ML*›

context *comm-semiring-1*
begin

lemma *semiring-normalization-rules* [*no-atp*]:

$(a * m) + (b * m) = (a + b) * m$
$(a * m) + m = (a + 1) * m$
$m + (a * m) = (a + 1) * m$
$m + m = (1 + 1) * m$
$0 + a = a$
$a + 0 = a$
$a * b = b * a$
$(a + b) * c = (a * c) + (b * c)$
$0 * a = 0$
$a * 0 = 0$
$1 * a = a$
$a * 1 = a$
$(lx * ly) * (rx * ry) = (lx * rx) * (ly * ry)$
$(lx * ly) * (rx * ry) = lx * (ly * (rx * ry))$
$(lx * ly) * (rx * ry) = rx * ((lx * ly) * ry)$
$(lx * ly) * rx = (lx * rx) * ly$
$(lx * ly) * rx = lx * (ly * rx)$
$lx * (rx * ry) = (lx * rx) * ry$
$lx * (rx * ry) = rx * (lx * ry)$
$(a + b) + (c + d) = (a + c) + (b + d)$
$(a + b) + c = a + (b + c)$
$a + (c + d) = c + (a + d)$
$(a + b) + c = (a + c) + b$
$a + c = c + a$
$a + (c + d) = (a + c) + d$
$(x \hat{} p) * (x \hat{} q) = x \hat{} (p + q)$
$x * (x \hat{} q) = x \hat{} (Suc\ q)$
$(x \hat{} q) * x = x \hat{} (Suc\ q)$
$x * x = x^2$
$(x * y) \hat{} q = (x \hat{} q) * (y \hat{} q)$
$(x \hat{} p) \hat{} q = x \hat{} (p * q)$
$x \hat{} 0 = 1$
$x \hat{} 1 = x$
$x * (y + z) = (x * y) + (x * z)$
$x \hat{} (Suc\ q) = x * (x \hat{} q)$
$x \hat{} (2*n) = (x \hat{} n) * (x \hat{} n)$
by (*simp-all add*: *algebra-simps power-add power2-eq-square power-mult-distrib power-mult del*: *one-add-one*)

local-setup ‹
Semiring-Normalizer.declare @{*thm comm-semiring-1-axioms*}
{*semiring* = ([***term*** ‹$x + y$›, ***term*** ‹$x * y$›, ***term*** ‹$x \hat{} n$›, ***term*** ‹0›, ***term*** ‹1›],
@{*thms semiring-normalization-rules*}),
ring = ([], []),
field = ([], []),
idom = [],
ideal = []}
›

end

context *comm-ring-1*
begin

lemma *ring-normalization-rules* [*no-atp*]:
$- x = (- 1) * x$
$x - y = x + (- y)$
by *simp-all*

local-setup ‹
Semiring-Normalizer.declare @{*thm comm-ring-1-axioms*}
{*semiring* = ([***term*** ‹$x + y$›, ***term*** ‹$x * y$›, ***term*** ‹$x \hat{} n$›, ***term*** ‹0›, ***term*** ‹1›],
@{*thms semiring-normalization-rules*}),
ring = ([***term*** ‹$x - y$›, ***term*** ‹$- x$›], @{*thms ring-normalization-rules*}),
field = ([], []),
idom = [],
ideal = []}
›

end

context *comm-semiring-1-cancel-crossproduct*
begin

local-setup ‹
Semiring-Normalizer.declare @{*thm comm-semiring-1-cancel-crossproduct-axioms*}
{*semiring* = ([***term*** ‹$x + y$›, ***term*** ‹$x * y$›, ***term*** ‹$x \hat{} n$›, ***term*** ‹0›, ***term*** ‹1›],
@{*thms semiring-normalization-rules*}),
ring = ([], []),
field = ([], []),
idom = @{*thms crossproduct-noteq add-scale-eq-noteq*},
ideal = []}
›

end

context *idom*
begin

local-setup ‹
Semiring-Normalizer.declare @{*thm idom-axioms*}
{*semiring* = ([***term*** ‹$x + y$›, ***term*** ‹$x * y$›, ***term*** ‹$x \hat{} n$›, ***term*** ‹0›, ***term*** ‹1›],
@{*thms semiring-normalization-rules*}),
ring = ([***term*** ‹$x - y$›, ***term*** ‹$- x$›], @{*thms ring-normalization-rules*}),
field = ([], []),
idom = @{*thms crossproduct-noteq add-scale-eq-noteq*},
ideal = @{*thms right-minus-eq add-0-iff*}}
›

end

context *field*
begin

local-setup ‹
Semiring-Normalizer.declare @{*thm field-axioms*}
{*semiring* = ([***term*** ‹*x* + *y*›, ***term*** ‹*x* ∗ *y*›, ***term*** ‹*x* ^ *n*›, ***term*** ‹*0*›, ***term*** ‹*1*›],
@{*thms semiring-normalization-rules*}),
ring = ([***term*** ‹*x* − *y*›, ***term*** ‹− *x*›], @{*thms ring-normalization-rules*}),
field = ([***term*** ‹*x* / *y*›, ***term*** ‹*inverse x*›], @{*thms divide-inverse inverse-eq-divide*}),
idom = @{*thms crossproduct-noteq add-scale-eq-noteq*},
ideal = @{*thms right-minus-eq add-0-iff*}}
›

end

code-identifier
code-module *Semiring-Normalization* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

60 Groebner bases

theory *Groebner-Basis*
imports *Semiring-Normalization Parity*
begin

60.1 Groebner Bases

lemmas *bool-simps* = *simp-thms*(*1*−*34*) — FIXME move to *HOL.HOL*

lemma *nnf-simps*: — FIXME shadows fact binding in *HOL.HOL*
(¬(*P* ∧ *Q*)) = (¬*P* ∨ ¬*Q*) (¬(*P* ∨ *Q*)) = (¬*P* ∧ ¬*Q*)
(*P* ⟶ *Q*) = (¬*P* ∨ *Q*)
(*P* = *Q*) = ((*P* ∧ *Q*) ∨ (¬*P* ∧ ¬ *Q*)) (¬ ¬(*P*)) = *P*
by *blast*+

lemma *dnf*:
(*P* ∧ (*Q* ∨ *R*)) = ((*P*∧*Q*) ∨ (*P*∧*R*))
((*Q* ∨ *R*) ∧ *P*) = ((*Q*∧*P*) ∨ (*R*∧*P*))
(*P* ∧ *Q*) = (*Q* ∧ *P*)
(*P* ∨ *Q*) = (*Q* ∨ *P*)
by *blast*+

lemmas *weak-dnf-simps* = *dnf bool-simps*

lemma *PFalse*:
  $P \equiv False \Longrightarrow \neg\ P$
  $\neg\ P \Longrightarrow (P \equiv False)$
 by *auto*

named-theorems *algebra pre−simplification rules for algebraic methods*
ML-file ‹*Tools/groebner.ML*›

method-setup *algebra* = ‹
 let
  fun keyword k = Scan.lift (Args.$$$ k −− Args.colon) >> K ()
  val addN = add
  val delN = del
  val any-keyword = keyword addN || keyword delN
  val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
 in
  Scan.optional (keyword addN |−− thms) [] −−
   Scan.optional (keyword delN |−− thms) [] >>
  (fn (add-ths, del-ths) => fn ctxt =>
   SIMPLE-METHOD' (Groebner.algebra-tac add-ths del-ths ctxt))
 end
› *solve polynomial equations over (semi)rings and ideal membership problems using Groebner bases*

declare *dvd-def*[*algebra*]
declare *mod-eq-0-iff-dvd*[*algebra*]
declare *mod-div-trivial*[*algebra*]
declare *mod-mod-trivial*[*algebra*]
declare *div-by-0*[*algebra*]
declare *mod-by-0*[*algebra*]
declare *mult-div-mod-eq*[*algebra*]
declare *div-minus-minus*[*algebra*]
declare *mod-minus-minus*[*algebra*]
declare *div-minus-right*[*algebra*]
declare *mod-minus-right*[*algebra*]
declare *div-0*[*algebra*]
declare *mod-0*[*algebra*]
declare *mod-by-1*[*algebra*]
declare *div-by-1*[*algebra*]
declare *mod-minus1-right*[*algebra*]
declare *div-minus1-right*[*algebra*]
declare *mod-mult-self2-is-0*[*algebra*]
declare *mod-mult-self1-is-0*[*algebra*]

lemma *zmod-eq-0-iff* [*algebra*]:
 ‹*m mod d = 0* ⟷ (∃ *q*. *m = d * q*)› **for** *m d* :: *int*
 by (*auto simp add*: *mod-eq-0-iff-dvd*)

declare *dvd-0-left-iff*[*algebra*]

declare *zdvd1-eq*[*algebra*]
declare *mod-eq-dvd-iff*[*algebra*]
declare *nat-mod-eq-iff*[*algebra*]

context *semiring-parity*
begin

declare *even-mult-iff* [*algebra*]
declare *even-power* [*algebra*]

end

context *ring-parity*
begin

declare *even-minus* [*algebra*]

end

declare *even-Suc* [*algebra*]
declare *even-diff-nat* [*algebra*]

end

61 Set intervals

theory *Set-Interval*
imports *Parity*
begin

lemma *card-2-iff*: *card S = 2* $\longleftrightarrow$ ($\exists$ *x y. S* = {*x*,*y*} $\wedge$ *x* $\neq$ *y*)
by (*auto simp*: *card-Suc-eq numeral-eq-Suc*)

lemma *card-2-iff'*: *card S = 2* $\longleftrightarrow$ ($\exists$ *x*$\in$*S*. $\exists$ *y*$\in$*S*. *x* $\neq$ *y* $\wedge$ ($\forall$ *z*$\in$*S*. *z* = *x* $\vee$ *z* = *y*))
by (*auto simp*: *card-Suc-eq numeral-eq-Suc*)

lemma *card-3-iff*: *card S = 3* $\longleftrightarrow$ ($\exists$ *x y z. S* = {*x*,*y*,*z*} $\wedge$ *x* $\neq$ *y* $\wedge$ *y* $\neq$ *z* $\wedge$ *x* $\neq$ *z*)
by (*fastforce simp*: *card-Suc-eq numeral-eq-Suc*)

context *ord*
begin

definition
lessThan :: *'a* $\Rightarrow$ *'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{..<-})›)
where
{..<*u*} == {*x*. *x* < *u*}

definition
atMost :: *'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{..-})›)
where
{..*u*} == {*x*. *x* ≤ *u*}

definition
greaterThan :: *'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-<..})›)
where
{*l*<..} == {*x*. *l*<*x*}

definition
atLeast :: *'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-..})›)
where
{*l*..} == {*x*. *l*≤*x*}

definition
greaterThanLessThan :: *'a ⇒ 'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-/<..<-})›) **where**
{*l*<..<*u*} == {*l*<..} ∩ {..<*u*}

definition
atLeastLessThan :: *'a ⇒ 'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-/..<-})›) **where**
{*l*..<*u*} == {*l*..} ∩ {..<*u*}

definition
greaterThanAtMost :: *'a ⇒ 'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-/<..-})›) **where**
{*l*<..*u*} == {*l*<..} ∩ {..*u*}

definition
atLeastAtMost :: *'a ⇒ 'a ⇒ 'a set* (‹(‹*indent=1 notation=‹mixfix set interval›*›{-/..-})›) **where**
{*l*..*u*} == {*l*..} ∩ {..*u*}

end

A note of warning when using {..<*n*} on type *nat*: it is equivalent to {*0*..<*n*} but some lemmas involving {*m*..<*n*} may not exist in {..<*n*}-form as well.

syntax (*ASCII*)
-UNION-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder UN›*›*UN -<=-./ -)*› [*0*, *0*, *10*] *10*)
-UNION-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder UN›*›*UN -<-./ -)*› [*0*, *0*, *10*] *10*)
-INTER-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder INT›*›*INT -<=-./ -)*› [*0*, *0*, *10*] *10*)
-INTER-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder INT›*›*INT -<-./ -)*› [*0*, *0*, *10*] *10*)

syntax (*latex* **output**)
-UNION-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(*3*⋃(‹*unbreakable*›- ≤ -)/ -)› [*0*, *0*, *10*] *10*)
-UNION-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(*3*⋃(‹*unbreakable*›- < -)/ -)› [*0*, *0*, *10*] *10*)
-INTER-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(*3*⋂(‹*unbreakable*›- ≤ -)/ -)› [*0*, *0*, *10*] *10*)
-INTER-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(*3*⋂(‹*unbreakable*›- < -)/ -)› [*0*, *0*, *10*] *10*)

syntax
-UNION-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder* ⋃››⋃-≤-./ -)› [*0*, *0*, *10*] *10*)
-UNION-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder* ⋃››⋃-<-./ -)› [*0*, *0*, *10*] *10*)
-INTER-le :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder* ⋂››⋂-≤-./ -)› [*0*, *0*, *10*] *10*)
-INTER-less :: *'a ⇒ 'a ⇒ 'b set ⇒ 'b set* (‹(‹*indent=3 notation=‹binder* ⋂››⋂-<-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-UNION-le -UNION-less ⇌ *Union* **and**
-INTER-le -INTER-less ⇌ *Inter*

translations
⋃*i*≤*n*. *A* ⇌ ⋃*i*∈{..*n*}. *A*
⋃*i*<*n*. *A* ⇌ ⋃*i*∈{..<*n*}. *A*
⋂*i*≤*n*. *A* ⇌ ⋂*i*∈{..*n*}. *A*
⋂*i*<*n*. *A* ⇌ ⋂*i*∈{..<*n*}. *A*

61.1 Various equivalences

lemma (**in** *ord*) *lessThan-iff* [*iff*]: (*i* ∈ *lessThan k*) = (*i*<*k*)
by (*simp add*: *lessThan-def*)

lemma *Compl-lessThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*lessThan k* = *atLeast k*
by (*auto simp add*: *lessThan-def atLeast-def*)

lemma *single-Diff-lessThan* [*simp*]: !!*k*:: *'a*::*preorder*. {*k*} − *lessThan k* = {*k*}
by *auto*

lemma (**in** *ord*) *greaterThan-iff* [*iff*]: (*i* ∈ *greaterThan k*) = (*k*<*i*)
by (*simp add*: *greaterThan-def*)

lemma *Compl-greaterThan* [*simp*]:
!!*k*:: *'a*::*linorder*. −*greaterThan k* = *atMost k*
by (*auto simp add*: *greaterThan-def atMost-def*)

lemma *Compl-atMost* [*simp*]: !!*k*:: *'a*::*linorder*. −*atMost k* = *greaterThan k*
by (*metis Compl-greaterThan double-complement*)

lemma (**in** *ord*) *atLeast-iff* [*iff*]: (*i* ∈ *atLeast k*) = (*k*<=*i*)
by (*simp add*: *atLeast-def*)

lemma *Compl-atLeast* [*simp*]: !!*k*:: *'a*::*linorder*. −*atLeast k* = *lessThan k*
by (*auto simp add*: *lessThan-def atLeast-def*)

lemma (**in** *ord*) *atMost-iff* [*iff*]: (*i* ∈ *atMost k*) = (*i*<=*k*)
by (*simp add*: *atMost-def*)

lemma *atMost-Int-atLeast*: !!*n*:: *'a*::*order*. *atMost n* ∩ *atLeast n* = {*n*}
by (*blast intro*: *order-antisym*)

lemma (**in** *linorder*) *lessThan-Int-lessThan*: { *a* <..} ∩ { *b* <..} = { *max a b* <..}
by *auto*

lemma (**in** *linorder*) *greaterThan-Int-greaterThan*: {..< *a*} ∩ {..< *b*} = {..< *min a b*}
by *auto*

61.2 Logical Equivalences for Set Inclusion and Equality

lemma *atLeast-empty-triv* [*simp*]: {{}..} = *UNIV*
by *auto*

lemma *atMost-UNIV-triv* [*simp*]: {..*UNIV*} = *UNIV*
by *auto*

lemma *atLeast-subset-iff* [*iff*]:
(*atLeast x* ⊆ *atLeast y*) = (*y* ≤ (*x*::*'a*::*preorder*))
by (*blast intro*: *order-trans*)

lemma *atLeast-eq-iff* [*iff*]:
(*atLeast x* = *atLeast y*) = (*x* = (*y*::*'a*::*order*))
by (*blast intro*: *order-antisym order-trans*)

lemma *greaterThan-subset-iff* [*iff*]:
(*greaterThan x* ⊆ *greaterThan y*) = (*y* ≤ (*x*::*'a*::*linorder*))
unfolding *greaterThan-def* **by** (*auto simp*: *linorder-not-less* [*symmetric*])

lemma *greaterThan-eq-iff* [*iff*]:
(*greaterThan x* = *greaterThan y*) = (*x* = (*y*::*'a*::*linorder*))
by (*auto simp*: *elim*!: *equalityE*)

lemma *atMost-subset-iff* [*iff*]: (*atMost x* ⊆ *atMost y*) = (*x* ≤ (*y*::*'a*::*preorder*))
by (*blast intro*: *order-trans*)

lemma *atMost-eq-iff* [*iff*]: (*atMost x* = *atMost y*) = (*x* = (*y*::'*a*::*order*))
by (*blast intro*: *order-antisym order-trans*)

lemma *lessThan-subset-iff* [*iff*]:
(*lessThan x* ⊆ *lessThan y*) = (*x* ≤ (*y*::'*a*::*linorder*))
unfolding *lessThan-def* **by** (*auto simp*: *linorder-not-less* [*symmetric*])

lemma *lessThan-eq-iff* [*iff*]:
(*lessThan x* = *lessThan y*) = (*x* = (*y*::'*a*::*linorder*))
by (*auto simp*: *elim*!: *equalityE*)

lemma *lessThan-strict-subset-iff*:
fixes *m n* :: '*a*::*linorder*
shows {..<*m*} < {..<*n*} ⟷ *m* < *n*
by (*metis leD lessThan-subset-iff linorder-linear not-less-iff-gr-or-eq psubset-eq*)

lemma (**in** *linorder*) *Ici-subset-Ioi-iff*: {*a* ..} ⊆ {*b* <..} ⟷ *b* < *a*
by *auto*

lemma (**in** *linorder*) *Iic-subset-Iio-iff*: {.. *a*} ⊆ {..< *b*} ⟷ *a* < *b*
by *auto*

lemma (**in** *preorder*) *Ioi-le-Ico*: {*a* <..} ⊆ {*a* ..}
by (*auto intro*: *less-imp-le*)

61.3 Two-sided intervals

context *ord*
begin

lemma *greaterThanLessThan-iff* [*simp*]: (*i* ∈ {*l*<..<*u*}) = (*l* < *i* ∧ *i* < *u*)
by (*simp add*: *greaterThanLessThan-def*)

lemma *atLeastLessThan-iff* [*simp*]: (*i* ∈ {*l*..<*u*}) = (*l* ≤ *i* ∧ *i* < *u*)
by (*simp add*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-iff* [*simp*]: (*i* ∈ {*l*<..*u*}) = (*l* < *i* ∧ *i* ≤ *u*)
by (*simp add*: *greaterThanAtMost-def*)

lemma *atLeastAtMost-iff* [*simp*]: (*i* ∈ {*l*..*u*}) = (*l* ≤ *i* ∧ *i* ≤ *u*)
by (*simp add*: *atLeastAtMost-def*)

The above four lemmas could be declared as iffs. Unfortunately this breaks many proofs. Since it only helps blast, it is better to leave them alone.

lemma *greaterThanLessThan-eq*: { *a* <..< *b*} = { *a* <..} ∩ {..< *b* }
by *auto*

lemma (**in** *order*) *atLeastLessThan-eq-atLeastAtMost-diff*:

$\{a..<b\} = \{a..b\} - \{b\}$
by (*auto simp add*: *atLeastLessThan-def atLeastAtMost-def*)

lemma (**in** *order*) *greaterThanAtMost-eq-atLeastAtMost-diff*:
$\{a<..b\} = \{a..b\} - \{a\}$
by (*auto simp add*: *greaterThanAtMost-def atLeastAtMost-def*)

end

61.3.1 Emptyness, singletons, subset

context *preorder*
begin

lemma *atLeastatMost-empty-iff*[*simp*]:
$\{a..b\} = \{\} \longleftrightarrow (\neg\ a \leq b)$
by *auto* (*blast intro*: *order-trans*)

lemma *atLeastatMost-empty-iff2*[*simp*]:
$\{\} = \{a..b\} \longleftrightarrow (\neg\ a \leq b)$
by *auto* (*blast intro*: *order-trans*)

lemma *atLeastLessThan-empty-iff*[*simp*]:
$\{a..<b\} = \{\} \longleftrightarrow (\neg\ a < b)$
by *auto* (*blast intro*: *le-less-trans*)

lemma *atLeastLessThan-empty-iff2*[*simp*]:
$\{\} = \{a..<b\} \longleftrightarrow (\neg\ a < b)$
by *auto* (*blast intro*: *le-less-trans*)

lemma *greaterThanAtMost-empty-iff*[*simp*]: $\{k<..l\} = \{\} \longleftrightarrow \neg\ k < l$
by *auto* (*blast intro*: *less-le-trans*)

lemma *greaterThanAtMost-empty-iff2*[*simp*]: $\{\} = \{k<..l\} \longleftrightarrow \neg\ k < l$
by *auto* (*blast intro*: *less-le-trans*)

lemma *atLeastatMost-subset-iff*[*simp*]:
$\{a..b\} \leq \{c..d\} \longleftrightarrow (\neg\ a \leq b) \vee c \leq a \wedge b \leq d$
unfolding *atLeastAtMost-def atLeast-def atMost-def*
by (*blast intro*: *order-trans*)

lemma *atLeastatMost-psubset-iff*:
$\{a..b\} < \{c..d\} \longleftrightarrow$
$((\neg\ a \leq b) \vee c \leq a \wedge b \leq d \wedge (c < a \vee b < d)) \wedge c \leq d$
by(*simp add*: *psubset-eq set-eq-iff less-le-not-le*)(*blast intro*: *order-trans*)

lemma *atLeastAtMost-subseteq-atLeastLessThan-iff*:
$\{a..b\} \subseteq \{c\ ..<\ d\} \longleftrightarrow (a \leq b \longrightarrow c \leq a \wedge b < d)$
by *auto* (*blast intro*: *local.order-trans local.le-less-trans elim*:)+

lemma *Icc-subset-Ici-iff*[*simp*]:
$\{l..h\} \subseteq \{l'..\} = (\neg\ l{\leq}h \vee l{\geq}l')$
by(*auto simp*: *subset-eq intro*: *order-trans*)

lemma *Icc-subset-Iic-iff*[*simp*]:
$\{l..h\} \subseteq \{..h'\} = (\neg\ l{\leq}h \vee h{\leq}h')$
by(*auto simp*: *subset-eq intro*: *order-trans*)

lemma *not-Ici-eq-empty*[*simp*]: $\{l..\} \neq \{\}$
by(*auto simp*: *set-eq-iff*)

lemma *not-Iic-eq-empty*[*simp*]: $\{..h\} \neq \{\}$
by(*auto simp*: *set-eq-iff*)

lemmas *not-empty-eq-Ici-eq-empty*[*simp*] = *not-Ici-eq-empty*[*symmetric*]
lemmas *not-empty-eq-Iic-eq-empty*[*simp*] = *not-Iic-eq-empty*[*symmetric*]

end

context *order*
begin

lemma *atLeastatMost-empty*[*simp*]: $b < a \Longrightarrow \{a..b\} = \{\}$
and *atLeastatMost-empty'*[*simp*]: $\neg\ a \leq b \Longrightarrow \{a..b\} = \{\}$
by(*auto simp*: *atLeastAtMost-def atLeast-def atMost-def*)

lemma *atLeastLessThan-empty*[*simp*]:
$b \leq a \Longrightarrow \{a..{<}b\} = \{\}$
by(*auto simp*: *atLeastLessThan-def*)

lemma *greaterThanAtMost-empty*[*simp*]: $l \leq k \Longrightarrow \{k{<}..l\} = \{\}$
by(*auto simp*:*greaterThanAtMost-def greaterThan-def atMost-def*)

lemma *greaterThanLessThan-empty*[*simp*]:$l \leq k \Longrightarrow \{k{<}..{<}l\} = \{\}$
by(*auto simp*:*greaterThanLessThan-def greaterThan-def lessThan-def*)

lemma *atLeastAtMost-singleton* [*simp*]: $\{a..a\} = \{a\}$
by (*auto simp add*: *atLeastAtMost-def atMost-def atLeast-def*)

lemma *atLeastAtMost-singleton'*: $a = b \Longrightarrow \{a\ ..\ b\} = \{a\}$ **by** *simp*

lemma *Icc-eq-Icc*[*simp*]:
$\{l..h\} = \{l'..h'\} = (l{=}l' \wedge h{=}h' \vee \neg\ l{\leq}h \wedge \neg\ l'{\leq}h')$
by (*simp add*: *order-class.order.eq-iff*) (*auto intro*: *order-trans*)

lemma (**in** *linorder*) *Ico-eq-Ico*:
$\{l..{<}h\} = \{l'..{<}h'\} = (l{=}l' \wedge h{=}h' \vee \neg\ l{<}h \wedge \neg\ l'{<}h')$
by (*metis atLeastLessThan-empty-iff2 nle-le not-less ord.atLeastLessThan-iff*)

lemma *atLeastAtMost-singleton-iff*[*simp*]:
$\{a\ ..\ b\} = \{c\} \longleftrightarrow a = b \wedge b = c$
proof
assume $\{a..b\} = \{c\}$
hence $*$: $\neg\ (\neg\ a \leq b)$ **unfolding** *atLeastatMost-empty-iff*[*symmetric*] **by** *simp*
with ‹$\{a..b\} = \{c\}$› **have** $c \leq a \wedge b \leq c$ **by** *auto*
with $*$ **show** $a = b \wedge b = c$ **by** *auto*
qed *simp*

Quantifiers

lemma *ex-interval-simps*:
$(\exists\, x \in \{..<u\}.\ P\ x) \longleftrightarrow (\exists\, x{<}u.\ P\ x)$
$(\exists\, x \in \{..u\}.\ P\ x) \longleftrightarrow (\exists\, x{\leq}u.\ P\ x)$
$(\exists\, x \in \{l<..\}.\ P\ x) \longleftrightarrow (\exists\, x{>}l.\ P\ x)$
$(\exists\, x \in \{l..\}.\ P\ x) \longleftrightarrow (\exists\, x{\geq}l.\ P\ x)$
$(\exists\, x \in \{l<..<u\}.\ P\ x) \longleftrightarrow (\exists\, x.\ l{<}x \wedge x{<}u \wedge P\ x)$
$(\exists\, x \in \{l..<u\}.\ P\ x) \longleftrightarrow (\exists\, x.\ l{\leq}x \wedge x{<}u \wedge P\ x)$
$(\exists\, x \in \{l<..u\}.\ P\ x) \longleftrightarrow (\exists\, x.\ l{<}x \wedge x{\leq}u \wedge P\ x)$
$(\exists\, x \in \{l..u\}.\ P\ x) \longleftrightarrow (\exists\, x.\ l{\leq}x \wedge x{\leq}u \wedge P\ x)$
by *auto*

lemma *all-interval-simps*:
$(\forall\, x \in \{..<u\}.\ P\ x) \longleftrightarrow (\forall\, x{<}u.\ P\ x)$
$(\forall\, x \in \{..u\}.\ P\ x) \longleftrightarrow (\forall\, x{\leq}u.\ P\ x)$
$(\forall\, x \in \{l<..\}.\ P\ x) \longleftrightarrow (\forall\, x{>}l.\ P\ x)$
$(\forall\, x \in \{l..\}.\ P\ x) \longleftrightarrow (\forall\, x{\geq}l.\ P\ x)$
$(\forall\, x \in \{l<..<u\}.\ P\ x) \longleftrightarrow (\forall\, x.\ l{<}x \longrightarrow x{<}u \longrightarrow P\ x)$
$(\forall\, x \in \{l..<u\}.\ P\ x) \longleftrightarrow (\forall\, x.\ l{\leq}x \longrightarrow x{<}u \longrightarrow P\ x)$
$(\forall\, x \in \{l<..u\}.\ P\ x) \longleftrightarrow (\forall\, x.\ l{<}x \longrightarrow x{\leq}u \longrightarrow P\ x)$
$(\forall\, x \in \{l..u\}.\ P\ x) \longleftrightarrow (\forall\, x.\ l{\leq}x \longrightarrow x{\leq}u \longrightarrow P\ x)$
by *auto*

The following results generalise their namesakes in *HOL.Nat* to intervals

lemma *lift-Suc-mono-le-ivl*:
assumes *mono*: $\bigwedge n.\ n{\in}N \Longrightarrow f\ n \leq f\ (Suc\ n)$
and $n \leq n'$ **and** *subN*: $\{n..<n'\} \subseteq N$
shows $f\ n \leq f\ n'$
proof (*cases* $n < n'$)
case *True*
then show *?thesis*
using *subN*
proof (*induction* $n\ n'$ *rule*: *less-Suc-induct*)
case (*1 i*)
then show *?case*
by (*simp add*: *mono subsetD*)
next
case (*2 i j k*)
have $f\ i \leq f\ j\ f\ j \leq f\ k$

```
    using 2 by force+
  then show ?case by auto
 qed
next
 case False
 with ‹n ≤ n'› show ?thesis by auto
qed

lemma lift-Suc-antimono-le-ivl:
 assumes mono: ⋀n. n∈N ⟹ f n ≥ f (Suc n)
   and n ≤ n' and subN: {n..<n'} ⊆ N
 shows f n ≥ f n'
proof (cases n < n')
 case True
 then show ?thesis
   using subN
 proof (induction n n' rule: less-Suc-induct)
   case (1 i)
   then show ?case
     by (simp add: mono subsetD)
 next
   case (2 i j k)
   have f i ≥ f j f j ≥ f k
     using 2 by force+
   then show ?case by auto
 qed
next
 case False
 with ‹n ≤ n'› show ?thesis by auto
qed

lemma lift-Suc-mono-less-ivl:
 assumes mono: ⋀n. n∈N ⟹ f n < f (Suc n)
   and n < n' and subN: {n..<n'} ⊆ N
 shows f n < f n'
 using ‹n < n'›
 using subN
proof (induction n n' rule: less-Suc-induct)
 case (1 i)
 then show ?case
   by (simp add: mono subsetD)
next
 case (2 i j k)
 have f i < f j f j < f k
   using 2 by force+
 then show ?case by auto
qed

end
```

context *no-top*
begin

lemma *not-UNIV-le-Icc*[*simp*]: ¬ *UNIV* ⊆ {*l*..*h*}
using *gt-ex*[*of h*] **by**(*auto simp*: *subset-eq less-le-not-le*)

lemma *not-UNIV-le-Iic*[*simp*]: ¬ *UNIV* ⊆ {..*h*}
using *gt-ex*[*of h*] **by**(*auto simp*: *subset-eq less-le-not-le*)

lemma *not-Ici-le-Icc*[*simp*]: ¬ {*l*..} ⊆ {*l'*..*h'*}
using *gt-ex*[*of h'*]
by(*auto simp*: *subset-eq less-le*)(*blast dest*:*antisym-conv intro*: *order-trans*)

lemma *not-Ici-le-Iic*[*simp*]: ¬ {*l*..} ⊆ {..*h'*}
using *gt-ex*[*of h'*]
by(*auto simp*: *subset-eq less-le*)(*blast dest*:*antisym-conv intro*: *order-trans*)

end

context *no-bot*
begin

lemma *not-UNIV-le-Ici*[*simp*]: ¬ *UNIV* ⊆ {*l*..}
using *lt-ex*[*of l*] **by**(*auto simp*: *subset-eq less-le-not-le*)

lemma *not-Iic-le-Icc*[*simp*]: ¬ {..*h*} ⊆ {*l'*..*h'*}
using *lt-ex*[*of l'*]
by(*auto simp*: *subset-eq less-le*)(*blast dest*:*antisym-conv intro*: *order-trans*)

lemma *not-Iic-le-Ici*[*simp*]: ¬ {..*h*} ⊆ {*l'*..}
using *lt-ex*[*of l'*]
by(*auto simp*: *subset-eq less-le*)(*blast dest*:*antisym-conv intro*: *order-trans*)

end

context *no-top*
begin

lemma *not-UNIV-eq-Icc*[*simp*]: ¬ *UNIV* = {*l'*..*h'*}
using *gt-ex*[*of h'*] **by**(*auto simp*: *set-eq-iff less-le-not-le*)

lemmas *not-Icc-eq-UNIV*[*simp*] = *not-UNIV-eq-Icc*[*symmetric*]

lemma *not-UNIV-eq-Iic*[*simp*]: ¬ *UNIV* = {..*h'*}
using *gt-ex*[*of h'*] **by**(*auto simp*: *set-eq-iff less-le-not-le*)

lemmas *not-Iic-eq-UNIV*[*simp*] = *not-UNIV-eq-Iic*[*symmetric*]

lemma *not-Icc-eq-Ici*[*simp*]: ¬ {*l*..*h*} = {*l'*..}
unfolding *atLeastAtMost-def* **using** *not-Ici-le-Iic*[*of l'*] **by** *blast*

lemmas *not-Ici-eq-Icc*[*simp*] = *not-Icc-eq-Ici*[*symmetric*]

lemma *not-Iic-eq-Ici*[*simp*]: ¬ {..*h*} = {*l'*..}
using *not-Ici-le-Iic*[*of l' h*] **by** *blast*

lemmas *not-Ici-eq-Iic*[*simp*] = *not-Iic-eq-Ici*[*symmetric*]

end

context *no-bot*
begin

lemma *not-UNIV-eq-Ici*[*simp*]: ¬ *UNIV* = {*l'*..}
using *lt-ex*[*of l'*] **by**(*auto simp*: *set-eq-iff* *less-le-not-le*)

lemmas *not-Ici-eq-UNIV*[*simp*] = *not-UNIV-eq-Ici*[*symmetric*]

lemma *not-Icc-eq-Iic*[*simp*]: ¬ {*l*..*h*} = {..*h'*}
unfolding *atLeastAtMost-def* **using** *not-Iic-le-Ici*[*of h'*] **by** *blast*

lemmas *not-Iic-eq-Icc*[*simp*] = *not-Icc-eq-Iic*[*symmetric*]

end

context *dense-linorder*
begin

lemma *greaterThanLessThan-empty-iff*[*simp*]:
{ *a* <..< *b* } = {} ⟷ *b* ≤ *a*
using *dense*[*of a b*] **by** (*cases a* < *b*) *auto*

lemma *greaterThanLessThan-empty-iff2*[*simp*]:
{} = { *a* <..< *b* } ⟷ *b* ≤ *a*
using *dense*[*of a b*] **by** (*cases a* < *b*) *auto*

lemma *atLeastLessThan-subseteq-atLeastAtMost-iff*:
{*a* ..< *b*} ⊆ { *c* .. *d* } ⟷ (*a* < *b* ⟶ *c* ≤ *a* ∧ *b* ≤ *d*)
using *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanAtMost-subseteq-atLeastAtMost-iff*:

$\{a <.. b\} \subseteq \{\ c .. d\ \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense*[*of a min c b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanLessThan-subseteq-atLeastAtMost-iff*:
$\{a <..< b\} \subseteq \{\ c .. d\ \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense*[*of a min c b*] *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanLessThan-subseteq-greaterThanLessThan*:
$\{a <..< b\} \subseteq \{c <..< d\} \longleftrightarrow (a < b \longrightarrow a \geq c \wedge b \leq d)$
using *dense*[*of a min c b*] *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanAtMost-subseteq-atLeastLessThan-iff*:
$\{a <.. b\} \subseteq \{\ c ..< d\ \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b < d)$
using *dense*[*of a min c b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanLessThan-subseteq-atLeastLessThan-iff*:
$\{a <..< b\} \subseteq \{\ c ..< d\ \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense*[*of a min c b*] *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

lemma *greaterThanLessThan-subseteq-greaterThanAtMost-iff*:
$\{a <..< b\} \subseteq \{\ c <.. d\ \} \longleftrightarrow (a < b \longrightarrow c \leq a \wedge b \leq d)$
using *dense*[*of a min c b*] *dense*[*of max a d b*]
by (*force simp*: *subset-eq Ball-def not-less*[*symmetric*])

end

context *no-top*
begin

lemma *greaterThan-non-empty*[*simp*]: $\{x <..\} \neq \{\}$
using *gt-ex*[*of x*] **by** *auto*

end

context *no-bot*
begin

lemma *lessThan-non-empty*[*simp*]: $\{..< x\} \neq \{\}$
using *lt-ex*[*of x*] **by** *auto*

end

lemma (**in** *linorder*) *atLeastLessThan-subset-iff*:
$\{a..<b\} \subseteq \{c..<d\} \Longrightarrow b \leq a \vee c \leq a \wedge b \leq d$

proof (*cases* $a < b$)
case *True*
assume *assm*: $\{a..<b\} \subseteq \{c..<d\}$
then have *1*: $c \leq a \wedge a \leq d$
using *True* **by** (*auto simp add*: *subset-eq Ball-def*)
then have *2*: $b \leq d$
using *assm* **by** (*auto simp add*: *subset-eq*)
from *1 2* **show** *?thesis*
by *simp*
qed (*auto*)

lemma *atLeastLessThan-inj*:
fixes $a\ b\ c\ d :: 'a::linorder$
assumes *eq*: $\{a\ ..<\ b\} = \{c\ ..<\ d\}$ **and** $a < b\ c < d$
shows $a = c\ b = d$
using *assms* **by** (*metis atLeastLessThan-subset-iff eq less-le-not-le antisym-conv2 subset-refl*)+

lemma *atLeastLessThan-eq-iff*:
fixes $a\ b\ c\ d :: 'a::linorder$
assumes $a < b\ c < d$
shows $\{a\ ..<\ b\} = \{c\ ..<\ d\} \longleftrightarrow a = c \wedge b = d$
using *atLeastLessThan-inj assms* **by** *auto*

lemma (**in** *linorder*) *Ioc-inj*:
$\langle\{a <..\ b\} = \{c <..\ d\} \longleftrightarrow (b \leq a \wedge d \leq c) \vee a = c \wedge b = d\rangle$ (**is** $\langle ?P \longleftrightarrow ?Q\rangle$)
proof
assume *?Q*
then show *?P*
by *auto*
next
assume *?P*
then have $\langle a < x \wedge x \leq b \longleftrightarrow c < x \wedge x \leq d\rangle$ **for** x
by (*simp add*: *set-eq-iff*)
from *this* [*of a*] *this* [*of b*] *this* [*of c*] *this* [*of d*] **show** *?Q*
by *auto*
qed

lemma (**in** *order*) *Iio-Int-singleton*: $\{..<k\} \cap \{x\} = (\textit{if } x < k \textit{ then } \{x\} \textit{ else } \{\})$
by *auto*

lemma (**in** *linorder*) *Ioc-subset-iff*: $\{a<..b\} \subseteq \{c<..d\} \longleftrightarrow (b \leq a \vee c \leq a \wedge b \leq d)$
by (*auto simp*: *subset-eq Ball-def*) (*metis less-le not-less*)

lemma (**in** *order-bot*) *atLeast-eq-UNIV-iff*: $\{x..\} = UNIV \longleftrightarrow x = bot$
by (*auto simp*: *set-eq-iff intro*: *le-bot*)

lemma (**in** *order-top*) *atMost-eq-UNIV-iff*: $\{..x\} = UNIV \longleftrightarrow x = top$

by (*auto simp*: *set-eq-iff intro*: *top-le*)

lemma (**in** *bounded-lattice*) *atLeastAtMost-eq-UNIV-iff*:
  $\{x..y\} = UNIV \longleftrightarrow (x = bot \wedge y = top)$
by (*auto simp*: *set-eq-iff intro*: *top-le le-bot*)

lemma *Iio-eq-empty-iff*: $\{..< n::'a::\{linorder, order\text{-}bot\}\} = \{\} \longleftrightarrow n = bot$
  by (*auto simp*: *set-eq-iff not-less le-bot*)

lemma *lessThan-empty-iff*: $\{..< n::nat\} = \{\} \longleftrightarrow n = 0$
  by (*simp add*: *Iio-eq-empty-iff bot-nat-def*)

lemma *mono-image-least*:
  assumes *f-mono*: *mono f* **and** *f-img*: $f\ `\ \{m\ ..<\ n\} = \{m'\ ..<\ n'\}$ $m < n$
  shows $f\ m = m'$
proof −
  from *f-img* **have** $\{m'\ ..<\ n'\} \neq \{\}$
    by (*metis atLeastLessThan-empty-iff image-is-empty*)
  with *f-img* **have** $m' \in f\ `\ \{m\ ..<\ n\}$ **by** *auto*
  then obtain *k* **where** $f\ k = m'$ $m \leq k$ **by** *auto*
  moreover have $m' \leq f\ m$ **using** *f-img* **by** *auto*
  ultimately show $f\ m = m'$
    using *f-mono* **by** (*auto elim*: *monoE*[**where** *x*=*m* **and** *y*=*k*])
qed

61.4 Infinite intervals

context *dense-linorder*
begin

lemma *infinite-Ioo*:
  assumes $a < b$
  shows ¬ *finite* $\{a<..<b\}$
proof
  assume *fin*: *finite* $\{a<..<b\}$
  moreover have *ne*: $\{a<..<b\} \neq \{\}$
    using ‹$a < b$› **by** *auto*
  ultimately have $a < Max\ \{a <..< b\}$ $Max\ \{a <..< b\} < b$
    using *Max-in*[*of* $\{a <..< b\}$] **by** *auto*
  then obtain *x* **where** $Max\ \{a <..< b\} < x$ $x < b$
    using *dense*[*of Max* $\{a<..<b\}$ *b*] **by** *auto*
  then have $x \in \{a <..< b\}$
    using ‹$a < Max\ \{a <..< b\}$› **by** *auto*
  then have $x \leq Max\ \{a <..< b\}$
    using *fin* **by** *auto*
  with ‹$Max\ \{a <..< b\} < x$› **show** *False* **by** *auto*
qed

lemma *infinite-Icc*: $a < b \Longrightarrow$ ¬ *finite* $\{a\ ..\ b\}$

using *greaterThanLessThan-subseteq-atLeastAtMost-iff*[*of a b a b*] *infinite-Ioo*[*of a b*]
by (*auto dest*: *finite-subset*)

lemma *infinite-Ico*: $a < b \Longrightarrow \neg$ *finite* $\{a ..< b\}$
using *greaterThanLessThan-subseteq-atLeastLessThan-iff*[*of a b a b*] *infinite-Ioo*[*of a b*]
by (*auto dest*: *finite-subset*)

lemma *infinite-Ioc*: $a < b \Longrightarrow \neg$ *finite* $\{a <.. b\}$
using *greaterThanLessThan-subseteq-greaterThanAtMost-iff*[*of a b a b*] *infinite-Ioo*[*of a b*]
by (*auto dest*: *finite-subset*)

lemma *infinite-Ioo-iff* [*simp*]: *infinite* $\{a<..<b\} \longleftrightarrow a < b$
using *not-less-iff-gr-or-eq* **by** (*fastforce simp*: *infinite-Ioo*)

lemma *infinite-Icc-iff* [*simp*]: *infinite* $\{a .. b\} \longleftrightarrow a < b$
using *not-less-iff-gr-or-eq* **by** (*fastforce simp*: *infinite-Icc*)

lemma *infinite-Ico-iff* [*simp*]: *infinite* $\{a..<b\} \longleftrightarrow a < b$
using *not-less-iff-gr-or-eq* **by** (*fastforce simp*: *infinite-Ico*)

lemma *infinite-Ioc-iff* [*simp*]: *infinite* $\{a<..b\} \longleftrightarrow a < b$
using *not-less-iff-gr-or-eq* **by** (*fastforce simp*: *infinite-Ioc*)

end

lemma *infinite-Iio*: $\neg$ *finite* $\{..< a :: 'a :: \{no\text{-}bot, linorder\}\}$
proof
assume *finite* $\{..< a\}$
then have $*$: $\bigwedge x.\ x < a \Longrightarrow Min\ \{..< a\} \leq x$
by *auto*
obtain x **where** $x < a$
using *lt-ex* **by** *auto*

obtain y **where** $y < Min\ \{..< a\}$
using *lt-ex* **by** *auto*
also have $Min\ \{..< a\} \leq x$
using ‹$x < a$› **by** *fact*
also note ‹$x < a$›
finally have $Min\ \{..< a\} \leq y$
by *fact*
with ‹$y < Min\ \{..< a\}$› **show** *False* **by** *auto*
qed

lemma *infinite-Iic*: $\neg$ *finite* $\{..\ a :: 'a :: \{no\text{-}bot, linorder\}\}$
using *infinite-Iio*[*of a*] *finite-subset*[*of* $\{..< a\}$ $\{..\ a\}$]
by (*auto simp*: *subset-eq less-imp-le*)

lemma *infinite-Ioi*: ¬ *finite* {*a* :: *'a* :: {*no-top*, *linorder*} *<..*}
proof
assume *finite* {*a <..*}
then have ∗: ⋀*x*. *a* < *x* ⟹ *x* ≤ *Max* {*a <..*}
by *auto*

obtain *y* **where** *Max* {*a <..*} < *y*
using *gt-ex* **by** *auto*

obtain *x* **where** *x*: *a* < *x*
using *gt-ex* **by** *auto*
also from *x* **have** *x* ≤ *Max* {*a <..*}
by *fact*
also note ‹*Max* {*a <..*} < *y*›
finally have *y* ≤ *Max* { *a <..*}
by *fact*
with ‹*Max* {*a <..*} < *y*› **show** *False* **by** *auto*
qed

lemma *infinite-Ici*: ¬ *finite* {*a* :: *'a* :: {*no-top*, *linorder*} *..*}
using *infinite-Ioi*[*of a*] *finite-subset*[*of* {*a <..*} {*a ..*}]
by (*auto simp*: *subset-eq less-imp-le*)

61.4.1 Intersection

context *linorder*
begin

lemma *Icc-minus-Ico* [*simp*]:
assumes *a* ≤ *b*
shows {*a..b'*} − {*a..<b*} = {*b..b'*}
using *assms* **by** *auto*

lemma *Icc-minus-Ioc* [*simp*]:
assumes *a* ≤ *b*
shows {*a'..b*} − {*a<..b*} = {*a'..a*}
using *assms* **by** *auto*

lemma *Icc-minus-Ioo* [*simp*]:
assumes *a* ≤ *b*
shows {*a..b*} − {*a<..<b*} = {*a*,*b*}
using *assms* **by** *auto*

lemma *Int-atLeastAtMost*[*simp*]: {*a..b*} ∩ {*c..d*} = {*max a c .. min b d*}
by *auto*

lemma *Int-atLeastAtMostR1*[*simp*]: {*..b*} ∩ {*c..d*} = {*c .. min b d*}
by *auto*

lemma *Int-atLeastAtMostR2*[*simp*]: $\{a..\} \cap \{c..d\} = \{max\ a\ c\ ..\ d\}$
by *auto*

lemma *Int-atLeastAtMostL1*[*simp*]: $\{a..b\} \cap \{..d\} = \{a\ ..\ min\ b\ d\}$
by *auto*

lemma *Int-atLeastAtMostL2*[*simp*]: $\{a..b\} \cap \{c..\} = \{max\ a\ c\ ..\ b\}$
by *auto*

lemma *Int-atLeastLessThan*[*simp*]: $\{a..<b\} \cap \{c..<d\} = \{max\ a\ c\ ..<\ min\ b\ d\}$
by *auto*

lemma *Int-greaterThanAtMost*[*simp*]: $\{a<..b\} \cap \{c<..d\} = \{max\ a\ c\ <..\ min\ b\ d\}$
by *auto*

lemma *Int-greaterThanLessThan*[*simp*]: $\{a<..<b\} \cap \{c<..<d\} = \{max\ a\ c\ <..<\ min\ b\ d\}$
by *auto*

lemma *Int-atMost*[*simp*]: $\{..a\} \cap \{..b\} = \{..\ min\ a\ b\}$
by (*auto simp*: *min-def*)

lemma *Ioc-disjoint*: $\{a<..b\} \cap \{c<..d\} = \{\} \longleftrightarrow b \leq a \vee d \leq c \vee b \leq c \vee d \leq a$
by *auto*

end

context *complete-lattice*
begin

lemma
shows *Sup-atLeast*[*simp*]: $Sup\ \{x\ ..\} = top$
and *Sup-greaterThanAtLeast*[*simp*]: $x < top \Longrightarrow Sup\ \{x\ <..\} = top$
and *Sup-atMost*[*simp*]: $Sup\ \{..\ y\} = y$
and *Sup-atLeastAtMost*[*simp*]: $x \leq y \Longrightarrow Sup\ \{\ x\ ..\ y\} = y$
and *Sup-greaterThanAtMost*[*simp*]: $x < y \Longrightarrow Sup\ \{\ x\ <..\ y\} = y$
by (*auto intro*!: *Sup-eqI*)

lemma
shows *Inf-atMost*[*simp*]: $Inf\ \{..\ x\} = bot$
and *Inf-atMostLessThan*[*simp*]: $top < x \Longrightarrow Inf\ \{..<\ x\} = bot$
and *Inf-atLeast*[*simp*]: $Inf\ \{x\ ..\} = x$
and *Inf-atLeastAtMost*[*simp*]: $x \leq y \Longrightarrow Inf\ \{\ x\ ..\ y\} = x$
and *Inf-atLeastLessThan*[*simp*]: $x < y \Longrightarrow Inf\ \{\ x\ ..<\ y\} = x$
by (*auto intro*!: *Inf-eqI*)

end

lemma
fixes $x\ y :: 'a :: \{complete\text{-}lattice, dense\text{-}linorder\}$
shows *Sup-lessThan*[*simp*]: $Sup\ \{..< y\} = y$
and *Sup-atLeastLessThan*[*simp*]: $x < y \Longrightarrow Sup\ \{ x\ ..< y\} = y$
and *Sup-greaterThanLessThan*[*simp*]: $x < y \Longrightarrow Sup\ \{ x <..< y\} = y$
and *Inf-greaterThan*[*simp*]: $Inf\ \{x <..\} = x$
and *Inf-greaterThanAtMost*[*simp*]: $x < y \Longrightarrow Inf\ \{ x <.. y\} = x$
and *Inf-greaterThanLessThan*[*simp*]: $x < y \Longrightarrow Inf\ \{ x <..< y\} = x$
by (*auto intro*!: *Inf-eqI Sup-eqI intro*: *dense-le dense-le-bounded dense-ge dense-ge-bounded*)

61.5 Intervals of natural numbers

61.5.1 The Constant *lessThan*

lemma *lessThan-0* [*simp*]: $lessThan\ (0::nat) = \{\}$
by (*simp add*: *lessThan-def*)

lemma *lessThan-Suc*: $lessThan\ (Suc\ k) = insert\ k\ (lessThan\ k)$
by (*simp add*: *lessThan-def less-Suc-eq*, *blast*)

The following proof is convenient in induction proofs where new elements get indices at the beginning. So it is used to transform $\{..<Suc\ n\}$ to *0* and $\{..<n\}$.

lemma *zero-notin-Suc-image* [*simp*]: $0 \notin Suc\ `\ A$
by *auto*

lemma *lessThan-Suc-eq-insert-0*: $\{..<Suc\ n\} = insert\ 0\ (Suc\ `\ \{..<n\})$
by (*auto simp*: *image-iff less-Suc-eq-0-disj*)

lemma *lessThan-Suc-atMost*: $lessThan\ (Suc\ k) = atMost\ k$
by (*simp add*: *lessThan-def atMost-def less-Suc-eq-le*)

lemma *atMost-Suc-eq-insert-0*: $\{..\ Suc\ n\} = insert\ 0\ (Suc\ `\ \{..\ n\})$
unfolding *lessThan-Suc-atMost*[*symmetric*] *lessThan-Suc-eq-insert-0*[*of Suc n*]
..

lemma *UN-lessThan-UNIV*: $(\bigcup m::nat.\ lessThan\ m) = UNIV$
by *blast*

61.5.2 The Constant *greaterThan*

lemma *greaterThan-0*: $greaterThan\ 0 = range\ Suc$
unfolding *greaterThan-def*
by (*blast dest*: *gr0-conv-Suc* [*THEN iffD1*])

lemma *greaterThan-Suc*: $greaterThan\ (Suc\ k) = greaterThan\ k - \{Suc\ k\}$
unfolding *greaterThan-def*
by (*auto elim*: *linorder-neqE*)

lemma *INT-greaterThan-UNIV*: $(\bigcap m{::}nat.\ greaterThan\ m) = \{\}$
by *blast*

61.5.3 The Constant *atLeast*

lemma *atLeast-0* [*simp*]: $atLeast\ (0{::}nat) = UNIV$
by (*unfold atLeast-def UNIV-def*, *simp*)

lemma *atLeast-Suc*: $atLeast\ (Suc\ k) = atLeast\ k - \{k\}$
unfolding *atLeast-def* **by** (*auto simp*: *order-le-less Suc-le-eq*)

lemma *atLeast-Suc-greaterThan*: $atLeast\ (Suc\ k) = greaterThan\ k$
by (*auto simp add*: *greaterThan-def atLeast-def less-Suc-eq-le*)

lemma *UN-atLeast-UNIV*: $(\bigcup m{::}nat.\ atLeast\ m) = UNIV$
by *blast*

61.5.4 The Constant *atMost*

lemma *atMost-0* [*simp*]: $atMost\ (0{::}nat) = \{0\}$
by (*simp add*: *atMost-def*)

lemma *atMost-Suc*: $atMost\ (Suc\ k) = insert\ (Suc\ k)\ (atMost\ k)$
unfolding *atMost-def* **by** (*auto simp add*: *less-Suc-eq order-le-less*)

lemma *UN-atMost-UNIV*: $(\bigcup m{::}nat.\ atMost\ m) = UNIV$
by *blast*

61.5.5 The Constant *atLeastLessThan*

The orientation of the following 2 rules is tricky. The lhs is defined in terms of the rhs. Hence the chosen orientation makes sense in this theory — the reverse orientation complicates proofs (eg nontermination). But outside, when the definition of the lhs is rarely used, the opposite orientation seems preferable because it reduces a specific concept to a more general one.

lemma *atLeast0LessThan* [*code-abbrev*]: $\{0{::}nat..<n\} = \{..<n\}$
by(*simp add*:*lessThan-def atLeastLessThan-def*)

lemma *atLeast0AtMost* [*code-abbrev*]: $\{0..n{::}nat\} = \{..n\}$
by(*simp add*:*atMost-def atLeastAtMost-def*)

lemma *lessThan-atLeast0*: $\{..<n\} = \{0{::}nat..<n\}$
by (*simp add*: *atLeast0LessThan*)

lemma *atMost-atLeast0*: $\{..n\} = \{0{::}nat..n\}$
by (*simp add*: *atLeast0AtMost*)

lemma *atLeastLessThan0*: $\{m..<0{::}nat\} = \{\}$

by (*simp add*: *atLeastLessThan-def*)

lemma *atLeast0-lessThan-Suc*: $\{0..<Suc\ n\} = insert\ n\ \{0..<n\}$
by (*simp add*: *atLeast0LessThan lessThan-Suc*)

lemma *atLeast0-lessThan-Suc-eq-insert-0*: $\{0..<Suc\ n\} = insert\ 0\ (Suc\ `\ \{0..<n\})$
by (*simp add*: *atLeast0LessThan lessThan-Suc-eq-insert-0*)

61.5.6 The Constant *atLeastAtMost*

lemma *Icc-eq-insert-lb-nat*: $m \leq n \Longrightarrow \{m..n\} = insert\ m\ \{Suc\ m..n\}$
by *auto*

lemma *atLeast0-atMost-Suc*:
$\{0..Suc\ n\} = insert\ (Suc\ n)\ \{0..n\}$
by (*simp add*: *atLeast0AtMost atMost-Suc*)

lemma *atLeast0-atMost-Suc-eq-insert-0*:
$\{0..Suc\ n\} = insert\ 0\ (Suc\ `\ \{0..n\})$
by (*simp add*: *atLeast0AtMost atMost-Suc-eq-insert-0*)

61.5.7 Intervals of nats with *Suc*

Not a simprule because the RHS is too messy.

lemma *atLeastLessThanSuc*:
$\{m..<Suc\ n\} = (if\ m \leq n\ then\ insert\ n\ \{m..<n\}\ else\ \{\})$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThan-singleton* [*simp*]: $\{m..<Suc\ m\} = \{m\}$
by (*auto simp add*: *atLeastLessThan-def*)

lemma *atLeastLessThanSuc-atLeastAtMost*: $\{l..<Suc\ u\} = \{l..u\}$
by (*simp add*: *lessThan-Suc-atMost atLeastAtMost-def atLeastLessThan-def*)

lemma *atLeastSucAtMost-greaterThanAtMost*: $\{Suc\ l..u\} = \{l<..u\}$
by (*simp add*: *atLeast-Suc-greaterThan atLeastAtMost-def*
greaterThanAtMost-def)

lemma *atLeastSucLessThan-greaterThanLessThan*: $\{Suc\ l..<u\} = \{l<..<u\}$
by (*simp add*: *atLeast-Suc-greaterThan atLeastLessThan-def*
greaterThanLessThan-def)

lemma *atLeastAtMostSuc-conv*: $m \leq Suc\ n \Longrightarrow \{m..Suc\ n\} = insert\ (Suc\ n)\ \{m..n\}$
by *auto*

lemma *atLeastAtMost-insertL*: $m \leq n \Longrightarrow insert\ m\ \{Suc\ m..n\} = \{m\ ..n\}$
by *auto*

The analogous result is useful on *int*:

lemma *atLeastAtMostPlus1-int-conv*:
m ≤ *1+n* ⟹ {*m*..*1+n*} = *insert* (*1+n*) {*m*..*n*::*int*}
by (*auto intro*: *set-eqI*)

lemma *atLeastLessThan-add-Un*: *i* ≤ *j* ⟹ {*i*..<*j+k*} = {*i*..<*j*} ∪ {*j*..<*j+k*::*nat*}
by (*induct k*) (*simp-all add*: *atLeastLessThanSuc*)

61.5.8 Intervals and numerals

lemma *lessThan-nat-numeral*: — Evaluation for specific numerals
lessThan (*numeral k* :: *nat*) = *insert* (*pred-numeral k*) (*lessThan* (*pred-numeral k*))
by (*simp add*: *numeral-eq-Suc lessThan-Suc*)

lemma *atMost-nat-numeral*: — Evaluation for specific numerals
atMost (*numeral k* :: *nat*) = *insert* (*numeral k*) (*atMost* (*pred-numeral k*))
by (*simp add*: *numeral-eq-Suc atMost-Suc*)

lemma *atLeastLessThan-nat-numeral*: — Evaluation for specific numerals
atLeastLessThan m (*numeral k* :: *nat*) =
(*if m* ≤ (*pred-numeral k*) *then insert* (*pred-numeral k*) (*atLeastLessThan m* (*pred-numeral k*))
else {})
by (*simp add*: *numeral-eq-Suc atLeastLessThanSuc*)

61.5.9 Image

context *linordered-semidom*
begin

lemma *image-add-atLeast*[*simp*]: *plus k* ‘ {*i*..} = {*k* + *i*..}
proof −
have *n* = *k* + (*n* − *k*) **if** *i* + *k* ≤ *n* **for** *n*
proof −
have *n* = (*n* − (*k* + *i*)) + (*k* + *i*) **using** *that*
by (*metis add-commute le-add-diff-inverse*)
then show *n* = *k* + (*n* − *k*)
by (*metis local.add-diff-cancel-left′ add-assoc add-commute*)
qed
then show *?thesis*
by (*fastforce simp*: *add-le-imp-le-diff add.commute*)
qed

lemma *image-add-atLeastAtMost* [*simp*]:
plus k ‘ {*i*..*j*} = {*i* + *k*..*j* + *k*} (**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
by (*auto simp add*: *ac-simps*)

next
show *?B* ⊆ *?A*
proof
fix *n*
assume *n* ∈ *?B*
then have *i* ≤ *n* − *k*
by (*simp add*: *add-le-imp-le-diff*)
have *n* = *n* − *k* + *k*
proof −
from ‹*n* ∈ *?B*› **have** *n* = *n* − (*i* + *k*) + (*i* + *k*)
by *simp*
also have ... = *n* − *k* − *i* + *i* + *k*
by (*simp add*: *algebra-simps*)
also have ... = *n* − *k* + *k*
using ‹*i* ≤ *n* − *k*› **by** *simp*
finally show *?thesis* **.**
qed
moreover have *n* − *k* ∈ {*i*..*j*}
using ‹*n* ∈ *?B*›
by (*auto simp*: *add-le-imp-le-diff add-le-add-imp-diff-le*)
ultimately show *n* ∈ *?A*
by (*simp add*: *ac-simps*)
qed
qed

lemma *image-add-atLeastAtMost′* [*simp*]:
(λ*n*. *n* + *k*) ‘ {*i*..*j*} = {*i* + *k*..*j* + *k*}
by (*simp add*: *add.commute* [*of* - *k*])

lemma *image-add-atLeastLessThan* [*simp*]:
plus k ‘ {*i*..<*j*} = {*i* + *k*..<*j* + *k*}
by (*simp add*: *image-set-diff atLeastLessThan-eq-atLeastAtMost-diff ac-simps*)

lemma *image-add-atLeastLessThan′* [*simp*]:
(λ*n*. *n* + *k*) ‘ {*i*..<*j*} = {*i* + *k*..<*j* + *k*}
by (*simp add*: *add.commute* [*of* - *k*])

lemma *image-add-greaterThanAtMost*[*simp*]: (+) *c* ‘ {*a*<..*b*} = {*c* + *a*<..*c* + *b*}
by (*simp add*: *image-set-diff greaterThanAtMost-eq-atLeastAtMost-diff ac-simps*)

end

context *ordered-ab-group-add*
begin

lemma
fixes *x* :: *′a*
shows *image-uminus-greaterThan*[*simp*]: *uminus* ‘ {*x*<..} = {..<−*x*}
and *image-uminus-atLeast*[*simp*]: *uminus* ‘ {*x*..} = {..−*x*}

proof *safe*
fix y **assume** $y < -x$
hence $*$: $x < -y$ **using** *neg-less-iff-less*[*of* $-y$ x] **by** *simp*
have $-(-y) \in$ *uminus* ` $\{x<..\}$
by (*rule imageI*) (*simp add*: $*$)
thus $y \in$ *uminus* ` $\{x<..\}$ **by** *simp*
next
fix y **assume** $y \leq -x$
have $-(-y) \in$ *uminus* ` $\{x..\}$
by (*rule imageI*) (*use* ‹$y \leq -x$›[*THEN le-imp-neg-le*] **in** ‹*simp*›)
thus $y \in$ *uminus* ` $\{x..\}$ **by** *simp*
qed *simp-all*

lemma
fixes $x :: 'a$
shows *image-uminus-lessThan*[*simp*]: *uminus* ` $\{..<x\} = \{-x<..\}$
and *image-uminus-atMost*[*simp*]: *uminus* ` $\{..x\} = \{-x..\}$
proof −
have *uminus* ` $\{..<x\} =$ *uminus* ` *uminus* ` $\{-x<..\}$
and *uminus* ` $\{..x\} =$ *uminus* ` *uminus* ` $\{-x..\}$ **by** *simp-all*
thus *uminus* ` $\{..<x\} = \{-x<..\}$ **and** *uminus* ` $\{..x\} = \{-x..\}$
by (*simp-all add*: *image-image*
del: *image-uminus-greaterThan image-uminus-atLeast*)
qed

lemma
fixes $x :: 'a$
shows *image-uminus-atLeastAtMost*[*simp*]: *uminus* ` $\{x..y\} = \{-y..-x\}$
and *image-uminus-greaterThanAtMost*[*simp*]: *uminus* ` $\{x<..y\} = \{-y..<-x\}$
and *image-uminus-atLeastLessThan*[*simp*]: *uminus* ` $\{x..<y\} = \{-y<..-x\}$
and *image-uminus-greaterThanLessThan*[*simp*]: *uminus* ` $\{x<..<y\} = \{-y<..<-x\}$
by (*simp-all add*: *atLeastAtMost-def greaterThanAtMost-def atLeastLessThan-def*
greaterThanLessThan-def image-Int[*OF inj-uminus*] *Int-commute*)

lemma *image-add-atMost*[*simp*]: $(+)$ c ` $\{..a\} = \{..c + a\}$
by (*auto intro*!: *image-eqI*[**where** $x{=}x - c$ **for** x] *simp*: *algebra-simps*)

end

lemma *image-Suc-atLeastAtMost* [*simp*]:
Suc ` $\{i..j\} = \{$*Suc i*..*Suc j*$\}$
using *image-add-atLeastAtMost* [*of 1 i j*]
by (*simp only*: *plus-1-eq-Suc*) *simp*

lemma *image-Suc-atLeastLessThan* [*simp*]:
Suc ` $\{i..<j\} = \{$*Suc i*..<*Suc j*$\}$
using *image-add-atLeastLessThan* [*of 1 i j*]
by (*simp only*: *plus-1-eq-Suc*) *simp*

corollary *image-Suc-atMost*:
Suc ‘ $\{..n\} = \{1..Suc\ n\}$
by (*simp add*: *atMost-atLeast0 atLeastLessThanSuc-atLeastAtMost*)

corollary *image-Suc-lessThan*:
Suc ‘ $\{..<n\} = \{1..n\}$
by (*simp add*: *lessThan-atLeast0 atLeastLessThanSuc-atLeastAtMost*)

lemma *image-diff-atLeastAtMost* [*simp*]:
fixes $d::'a::linordered\text{-}idom$ **shows** $((-)\ d\ ‘\ \{a..b\}) = \{d-b..d-a\}$
proof
show $\{d - b..d - a\} \subseteq (-)\ d\ ‘\ \{a..b\}$
proof
fix x
assume $x \in \{d - b..d - a\}$
then have $d - x \in \{a..b\}$ **and** $x = d - (d - x)$
by *auto*
then show $x \in (-)\ d\ ‘\ \{a..b\}$
by (*rule rev-image-eqI*)
qed
qed(*auto*)

lemma *image-diff-atLeastLessThan* [*simp*]:
fixes $a\ b\ c::'a::linordered\text{-}idom$
shows $(-)\ c\ ‘\ \{a..<b\} = \{c - b<..c - a\}$
proof −
have $(-)\ c\ ‘\ \{a..<b\} = (+)\ c\ ‘\ uminus\ ‘\ \{a\ ..<b\}$
unfolding *image-image* **by** *simp*
also have $\ldots = \{c - b<..c - a\}$ **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-greaterThanAtMost*[*simp*]:
fixes $a\ b\ c::'a::linordered\text{-}idom$
shows $(-)\ c\ ‘\ \{a<..b\} = \{c - b..<c - a\}$
proof −
have $(-)\ c\ ‘\ \{a<..b\} = (+)\ c\ ‘\ uminus\ ‘\ \{a<..b\}$
unfolding *image-image* **by** *simp*
also have $\ldots = \{c - b..<c - a\}$ **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-atLeast*[*simp*]:
fixes $a\ c::'a::linordered\text{-}idom$
shows $(-)\ c\ ‘\ \{a..\} = \{..c - a\}$
proof −
have $(-)\ c\ ‘\ \{a..\} = (+)\ c\ ‘\ uminus\ ‘\ \{a\ ..\}$
unfolding *image-image* **by** *simp*
also have $\ldots = \{..c - a\}$ **by** *simp*

finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-AtMost*[*simp*]:
fixes *b c*::*'a*::*linordered-idom*
shows (−) *c* ‘ {..*b*} = {*c* − *b*..}
proof −
have (−) *c* ‘ {..*b*} = (+) *c* ‘ *uminus* ‘ {..*b*}
unfolding *image-image* **by** *simp*
also have ... = {*c* − *b*..} **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *image-minus-const-atLeastAtMost'* [*simp*]:
(λ*t*. *t*−*d*)‘{*a*..*b*} = {*a*−*d*..*b*−*d*} **for** *d*::*'a*::*linordered-idom*
by (*metis* (*no-types*, *lifting*) *diff-conv-add-uminus image-add-atLeastAtMost' image-cong*)

context *linordered-field*
begin

lemma *image-mult-atLeastAtMost* [*simp*]:
((∗) *d* ‘ {*a*..*b*}) = {*d*∗*a*..*d*∗*b*} **if** *d*>*0*
using *that*
by (*auto simp*: *field-simps mult-le-cancel-right intro*: *rev-image-eqI* [**where** *x*=*x*/*d* **for** *x*])

lemma *image-divide-atLeastAtMost* [*simp*]:
((λ*c*. *c* / *d*) ‘ {*a*..*b*}) = {*a*/*d*..*b*/*d*} **if** *d*>*0*
proof −
from *that* **have** *inverse d* > *0*
by *simp*
with *image-mult-atLeastAtMost* [*of inverse d a b*]
have (∗) (*inverse d*) ‘ {*a*..*b*} = {*inverse d* ∗ *a*..*inverse d* ∗ *b*}
by *blast*
moreover have (∗) (*inverse d*) = (λ*c*. *c* / *d*)
by (*simp add*: *fun-eq-iff field-simps*)
ultimately show *?thesis*
by *simp*
qed

lemma *image-mult-atLeastAtMost-if*:
(∗) *c* ‘ {*x* .. *y*} =
(*if c* > *0 then* {*c* ∗ *x* .. *c* ∗ *y*} *else if x* ≤ *y then* {*c* ∗ *y* .. *c* ∗ *x*} *else* {})
proof (*cases c* = *0* ∨ *x* > *y*)
case *True*
then show *?thesis*
by *auto*
next

```
  case False
  then have x ≤ y
    by auto
  from False consider c < 0| c > 0
    by (auto simp add: neq-iff)
  then show ?thesis
  proof cases
    case 1
    have (∗) c ‘ {x..y} = {c ∗ y..c ∗ x}
    proof (rule set-eqI)
      fix d
      from 1 have inj (λz. z / c)
        by (auto intro: injI)
      then have d ∈ (∗) c ‘ {x..y} ⟷ d / c ∈ (λz. z div c) ‘ (∗) c ‘ {x..y}
        by (subst inj-image-mem-iff) simp-all
      also have ... ⟷ d / c ∈ {x..y}
        using 1 by (simp add: image-image)
      also have ... ⟷ d ∈ {c ∗ y..c ∗ x}
        by (auto simp add: field-simps 1)
      finally show d ∈ (∗) c ‘ {x..y} ⟷ d ∈ {c ∗ y..c ∗ x} .
    qed
    with ‹x ≤ y› show ?thesis
      by auto
  qed (simp add: mult-left-mono-neg)
qed

lemma image-mult-atLeastAtMost-if ′:
  (λx. x ∗ c) ‘ {x..y} =
    (if x ≤ y then if c > 0 then {x ∗ c .. y ∗ c} else {y ∗ c .. x ∗ c} else {})
  using image-mult-atLeastAtMost-if [of c x y] by (auto simp add: ac-simps)

lemma image-affinity-atLeastAtMost:
  ((λx. m ∗ x + c) ‘ {a..b}) = (if {a..b} = {} then {}
          else if 0 ≤ m then {m ∗ a + c .. m ∗ b + c}
          else {m ∗ b + c .. m ∗ a + c})
proof −
  have ∗: (λx. m ∗ x + c) = ((λx. x + c) ∘ (∗) m)
    by (simp add: fun-eq-iff)
 show ?thesis by (simp only: ∗ image-comp [symmetric] image-mult-atLeastAtMost-if)
    (auto simp add: mult-le-cancel-left)
qed

lemma image-affinity-atLeastAtMost-diff:
  ((λx. m∗x − c) ‘ {a..b}) = (if {a..b}={} then {}
          else if 0 ≤ m then {m∗a − c .. m∗b − c}
          else {m∗b − c .. m∗a − c})
  using image-affinity-atLeastAtMost [of m −c a b]
  by simp
```

lemma *image-affinity-atLeastAtMost-div*:
((λx. x/m + c) ' {a..b}) = (*if* {a..b}={} *then* {}
else if 0 ≤ m *then* {a/m + c .. b/m + c}
else {b/m + c .. a/m + c})
using *image-affinity-atLeastAtMost* [*of inverse m c a b*]
by (*simp add*: *field-class.field-divide-inverse algebra-simps inverse-eq-divide*)

lemma *image-affinity-atLeastAtMost-div-diff*:
((λx. x/m − c) ' {a..b}) = (*if* {a..b}={} *then* {}
else if 0 ≤ m *then* {a/m − c .. b/m − c}
else {b/m − c .. a/m − c})
using *image-affinity-atLeastAtMost-diff* [*of inverse m c a b*]
by (*simp add*: *field-class.field-divide-inverse algebra-simps inverse-eq-divide*)

end

lemma *atLeast1-lessThan-eq-remove0*:
{*Suc 0*..<n} = {..<n} − {*0*}
by *auto*

lemma *atLeast1-atMost-eq-remove0*:
{*Suc 0*..n} = {..n} − {*0*}
by *auto*

lemma *image-add-int-atLeastLessThan*:
(λx. x + (l::*int*)) ' {*0*..<u−l} = {l..<u}
by *safe auto*

lemma *image-minus-const-atLeastLessThan-nat*:
fixes c :: *nat*
shows (λi. i − c) ' {x ..< y} =
(*if* c < y *then* {x − c ..< y − c} *else if* x < y *then* {*0*} *else* {})
(**is** - = *?right*)
proof *safe*
fix a **assume** a: a ∈ *?right*
show a ∈ (λi. i − c) ' {x ..< y}
proof *cases*
assume c < y **with** a **show** *?thesis*
by (*auto intro*!: *image-eqI*[*of* - - a + c])
next
assume ¬ c < y **with** a **show** *?thesis*
by (*auto intro*!: *image-eqI*[*of* - - x] *split*: *if-split-asm*)
qed
qed *auto*

lemma *image-int-atLeastLessThan*:
int ' {a..<b} = {*int* a..<*int* b}
by (*auto intro*!: *image-eqI* [**where** x = *nat* x **for** x])

lemma *image-int-atLeastAtMost*:
int ' $\{a..b\} = \{int\ a..int\ b\}$
by (*auto intro*!: *image-eqI* [**where** $x = nat\ x$ **for** x])

61.5.10 Finiteness

lemma *finite-lessThan* [*iff*]: **fixes** $k :: nat$ **shows** *finite* $\{..<k\}$
by (*induct k*) (*simp-all add*: *lessThan-Suc*)

lemma *finite-atMost* [*iff*]: **fixes** $k :: nat$ **shows** *finite* $\{..k\}$
by (*induct k*) (*simp-all add*: *atMost-Suc*)

lemma *finite-greaterThanLessThan* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l<..<u\}$
by (*simp add*: *greaterThanLessThan-def*)

lemma *finite-atLeastLessThan* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l..<u\}$
by (*simp add*: *atLeastLessThan-def*)

lemma *finite-greaterThanAtMost* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l<..u\}$
by (*simp add*: *greaterThanAtMost-def*)

lemma *finite-atLeastAtMost* [*iff*]:
fixes $l :: nat$ **shows** *finite* $\{l..u\}$
by (*simp add*: *atLeastAtMost-def*)

A bounded set of natural numbers is finite.

lemma *bounded-nat-set-is-finite*: $(\forall i{\in}N.\ i < (n::nat)) \Longrightarrow$ *finite N*
by (*rule finite-subset* [*OF* - *finite-lessThan*]) *auto*

A set of natural numbers is finite iff it is bounded.

lemma *finite-nat-set-iff-bounded*:
finite$(N::nat\ set) = (\exists m.\ \forall n{\in}N.\ n{<}m)$ (**is** *?F* = *?B*)
proof
assume *f*:*?F* **show** *?B*
using *Max-ge*[*OF* ‹*?F*›, *simplified less-Suc-eq-le*[*symmetric*]] **by** *blast*
next
assume *?B* **show** *?F* **using** ‹*?B*› **by**(*blast intro*:*bounded-nat-set-is-finite*)
qed

lemma *finite-nat-set-iff-bounded-le*: *finite*$(N::nat\ set) = (\exists m.\ \forall n{\in}N.\ n{\leq}m)$
unfolding *finite-nat-set-iff-bounded*
by (*blast dest*:*less-imp-le-nat le-imp-less-Suc*)

lemma *finite-less-ub*:
$\bigwedge f::nat{\Rightarrow}nat.\ (!!n.\ n \leq f\ n) \Longrightarrow$ *finite* $\{n.\ f\ n \leq u\}$
by (*rule finite-subset*[*of* - $\{..u\}$])

(*auto intro*: *order-trans*)

lemma *bounded-Max-nat*:
fixes $P :: nat \Rightarrow bool$
assumes x: $P\ x$ **and** M: $\bigwedge x.\ P\ x \Longrightarrow x \leq M$
obtains m **where** $P\ m\ \bigwedge x.\ P\ x \Longrightarrow x \leq m$
proof −
have *finite* $\{x.\ P\ x\}$
using M *finite-nat-set-iff-bounded-le* **by** *auto*
then have $Max\ \{x.\ P\ x\} \in \{x.\ P\ x\}$
using *Max-in* x **by** *auto*
then show *?thesis*
by (*simp add*: ‹*finite* $\{x.\ P\ x\}$› *that*)
qed

Any subset of an interval of natural numbers the size of the subset is exactly that interval.

lemma *subset-card-intvl-is-intvl*:
assumes $A \subseteq \{k..<k + card\ A\}$
shows $A = \{k..<k + card\ A\}$
proof (*cases finite A*)
case *True*
from *this* **and** *assms* **show** *?thesis*
proof (*induct A rule*: *finite-linorder-max-induct*)
case *empty* **thus** *?case* **by** *auto*
next
case (*insert b A*)
hence ∗: $b \notin A$ **by** *auto*
with *insert* **have** $A \leq \{k..<k + card\ A\}$ **and** $b = k + card\ A$
by *fastforce*+
with *insert* ∗ **show** *?case* **by** *auto*
qed
next
case *False*
with *assms* **show** *?thesis* **by** *simp*
qed

61.5.11 Proving Inclusions and Equalities between Unions

lemma *UN-le-eq-Un0*:
$(\bigcup i{\leq}n{::}nat.\ M\ i) = (\bigcup i{\in}\{1..n\}.\ M\ i) \cup M\ 0$ (**is** *?A* = *?B*)
proof
show *?A* ⊆ *?B*
proof
fix x **assume** $x \in$ *?A*
then obtain i **where** i: $i{\leq}n\ x \in M\ i$ **by** *auto*
show $x \in$ *?B*
proof(*cases i*)
case *0* **with** i **show** *?thesis* **by** *simp*

```
    next
      case (Suc j) with i show ?thesis by auto
    qed
  qed
next
  show ?B ⊆ ?A by fastforce
qed

lemma UN-le-add-shift:
  (⋃ i≤n::nat. M(i+k)) = (⋃ i∈{k..n+k}. M i) (is ?A = ?B)
proof
  show ?A ⊆ ?B by fastforce
next
  show ?B ⊆ ?A
  proof
    fix x assume x ∈ ?B
    then obtain i where i: i ∈ {k..n+k} x ∈ M(i) by auto
    hence i−k≤n ∧ x ∈ M((i−k)+k) by auto
    thus x ∈ ?A by blast
  qed
qed

lemma UN-le-add-shift-strict:
  (⋃ i<n::nat. M(i+k)) = (⋃ i∈{k..<n+k}. M i) (is ?A = ?B)
proof
  show ?B ⊆ ?A
  proof
    fix x assume x ∈ ?B
    then obtain i where i: i ∈ {k..<n+k} x ∈ M(i) by auto
    then have i − k < n ∧ x ∈ M((i−k) + k) by auto
    then show x ∈ ?A using UN-le-add-shift by blast
  qed
qed (fastforce)

lemma UN-UN-finite-eq: (⋃ n::nat. ⋃ i∈{0..<n}. A i) = (⋃ n. A n)
  by (auto simp add: atLeast0LessThan)

lemma UN-finite-subset:
  (⋀ n::nat. (⋃ i∈{0..<n}. A i) ⊆ C) ⟹ (⋃ n. A n) ⊆ C
  by (subst UN-UN-finite-eq [symmetric]) blast

lemma UN-finite2-subset:
  assumes ⋀ n::nat. (⋃ i∈{0..<n}. A i) ⊆ (⋃ i∈{0..<n + k}. B i)
  shows (⋃ n. A n) ⊆ (⋃ n. B n)
proof (rule UN-finite-subset, rule subsetI)
  fix n and a
  from assms have (⋃ i∈{0..<n}. A i) ⊆ (⋃ i∈{0..<n + k}. B i) .
  moreover assume a ∈ (⋃ i∈{0..<n}. A i)
  ultimately have a ∈ (⋃ i∈{0..<n + k}. B i) by blast
```

then show $a \in (\bigcup i.\ B\ i)$ **by** (*auto simp add*: *UN-UN-finite-eq*)
qed

lemma *UN-finite2-eq*:
assumes $(\bigwedge n{::}nat.\ (\bigcup i{\in}\{0..{<}n\}.\ A\ i) = (\bigcup i{\in}\{0..{<}n+k\}.\ B\ i))$
shows $(\bigcup n.\ A\ n) = (\bigcup n.\ B\ n)$
proof (*rule subset-antisym* [*OF UN-finite-subset UN-finite2-subset*])
fix n
show $\bigcup\ (A\ `\ \{0..{<}n\}) \subseteq (\bigcup n.\ B\ n)$
using *assms* **by** *auto*
next
fix n
show $\bigcup\ (B\ `\ \{0..{<}n\}) \subseteq \bigcup\ (A\ `\ \{0..{<}n+k\})$
using *assms* **by** (*force simp add*: *atLeastLessThan-add-Un* [*of 0*])+
qed

61.5.12 Cardinality

lemma *card-lessThan* [*simp*]: *card* $\{..{<}u\} = u$
by (*induct u*, *simp-all add*: *lessThan-Suc*)

lemma *card-atMost* [*simp*]: *card* $\{..u\} = Suc\ u$
by (*simp add*: *lessThan-Suc-atMost* [*THEN sym*])

lemma *card-atLeastLessThan* [*simp*]: *card* $\{l..{<}u\} = u - l$
proof −
have $(\lambda x.\ x + l)\ `\ \{..{<}u - l\} \subseteq \{l..{<}u\}$
by *auto*
moreover have $\{l..{<}u\} \subseteq (\lambda x.\ x + l)\ `\ \{..{<}u{-}l\}$
proof
fix x
assume $*$: $x \in \{l..{<}u\}$
then have $x - l \in \{..{<}\ u\ {-}l\}$
by *auto*
then have $(x - l) + l \in (\lambda x.\ x + l)\ `\ \{..{<}\ u\ {-}l\}$
by *auto*
then show $x \in (\lambda x.\ x + l)\ `\ \{..{<}u - l\}$
using $*$ **by** *auto*
qed
ultimately have $\{l..{<}u\} = (\lambda x.\ x + l)\ `\ \{..{<}u{-}l\}$
by *auto*
then have *card* $\{l..{<}u\} =$ *card* $\{..{<}u{-}l\}$
by (*simp add*: *card-image inj-on-def*)
then show *?thesis*
by *simp*
qed

lemma *card-atLeastAtMost* [*simp*]: *card* $\{l..u\} = Suc\ u - l$
by (*subst atLeastLessThanSuc-atLeastAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanAtMost* [*simp*]: *card* {*l*<..*u*} = *u* − *l*
by (*subst atLeastSucAtMost-greaterThanAtMost* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan* [*simp*]: *card* {*l*<..<*u*} = *u* − *Suc l*
by (*subst atLeastSucLessThan-greaterThanLessThan* [*THEN sym*], *simp*)

lemma *subset-eq-atLeast0-lessThan-finite*:
fixes *n* :: *nat*
assumes *N* ⊆ {*0*..<*n*}
shows *finite N*
using *assms finite-atLeastLessThan* **by** (*rule finite-subset*)

lemma *subset-eq-atLeast0-atMost-finite*:
fixes *n* :: *nat*
assumes *N* ⊆ {*0*..*n*}
shows *finite N*
using *assms finite-atLeastAtMost* **by** (*rule finite-subset*)

lemma *ex-bij-betw-nat-finite*:
finite M ⟹ ∃ *h*. *bij-betw h* {*0*..<*card M*} *M*
apply(*drule finite-imp-nat-seg-image-inj-on*)
apply(*auto simp*:*atLeast0LessThan*[*symmetric*] *lessThan-def*[*symmetric*] *card-image bij-betw-def*)
done

lemma *ex-bij-betw-finite-nat*:
finite M ⟹ ∃ *h*. *bij-betw h M* {*0*..<*card M*}
by (*blast dest*: *ex-bij-betw-nat-finite bij-betw-inv*)

lemma *finite-same-card-bij*:
finite A ⟹ *finite B* ⟹ *card A* = *card B* ⟹ ∃ *h*. *bij-betw h A B*
apply(*drule ex-bij-betw-finite-nat*)
apply(*drule ex-bij-betw-nat-finite*)
apply(*auto intro*!:*bij-betw-trans*)
done

lemma *ex-bij-betw-nat-finite-1*:
finite M ⟹ ∃ *h*. *bij-betw h* {*1* .. *card M*} *M*
by (*rule finite-same-card-bij*) *auto*

lemma *bij-betw-iff-card*:
assumes *finite A finite B*
shows (∃ *f*. *bij-betw f A B*) ⟷ (*card A* = *card B*)
proof
assume *card A* = *card B*
moreover obtain *f* **where** *bij-betw f A* {*0* ..< *card A*}
using *assms ex-bij-betw-finite-nat* **by** *blast*
moreover obtain *g* **where** *bij-betw g* {*0* ..< *card B*} *B*

using *assms ex-bij-betw-nat-finite* **by** *blast*
ultimately have *bij-betw* $(g \circ f)$ A B
by (*auto simp*: *bij-betw-trans*)
thus $(\exists f.$ *bij-betw* f A $B)$ **by** *blast*
qed (*auto simp*: *bij-betw-same-card*)

lemma *subset-eq-atLeast0-lessThan-card*:
fixes n :: *nat*
assumes $N \subseteq \{0..<n\}$
shows *card* $N \leq n$
proof −
from *assms finite-lessThan* **have** *card* $N \leq$ *card* $\{0..<n\}$
using *card-mono* **by** *blast*
then show *?thesis* **by** *simp*
qed

Relational version of *card-inj-on-le*:

lemma *card-le-if-inj-on-rel*:
assumes *finite* B
$\bigwedge a.\ a \in A \Longrightarrow \exists\, b.\ b \in B \wedge r\ a\ b$
$\bigwedge a1\ a2\ b.\ [\![\ a1 \in A;\ \ a2 \in A;\ \ b \in B;\ \ r\ a1\ b;\ \ r\ a2\ b\ ]\!] \Longrightarrow a1 = a2$
shows *card* $A \leq$ *card* B
proof −
let *?P* $= \lambda a\ b.\ b \in B \wedge r\ a\ b$
let *?f* $= \lambda a.$ *SOME* $b.$ *?P* $a\ b$
have *1*: *?f* ' $A \subseteq B$ **by** (*auto intro*: *someI2-ex*[*OF assms*(*2*)])
have *inj-on ?f A*
unfolding *inj-on-def*
proof *safe*
fix *a1 a2* **assume** *asms*: *a1* $\in$ *A a2* $\in$ *A ?f a1* $=$ *?f a2*
have *0*: *?f a1* $\in$ *B* **using** *1* ‹*a1* $\in$ *A*› **by** *blast*
have *1*: *r a1* (*?f a1*) **using** *someI-ex*[*OF assms*(*2*)[*OF* ‹*a1* $\in$ *A*›]] **by** *blast*
have *2*: *r a2* (*?f a1*) **using** *someI-ex*[*OF assms*(*2*)[*OF* ‹*a2* $\in$ *A*›]] *asms*(*3*) **by** *auto*
show *a1* $=$ *a2* **using** *assms*(*3*)[*OF asms*(*1*,*2*) *0 1 2*] **.**
qed
with *1* **show** *?thesis* **using** *card-inj-on-le*[*of ?f A B*] *assms*(*1*) **by** *simp*
qed

lemma *inj-on-funpow-least*:
‹*inj-on* $(\lambda k.\ (f\ ^{\frown\frown}\ k)\ s)$ $\{0..<n\}$›
if ‹$(f\ ^{\frown\frown}\ n)\ s = s$› ‹$\bigwedge m.\ 0 < m \Longrightarrow m < n \Longrightarrow (f\ ^{\frown\frown}\ m)\ s \neq s$›
proof −
{ fix $k\ l$ **assume** A: $k < n\ l < n\ k \neq l\ (f\ ^{\frown\frown}\ k)\ s = (f\ ^{\frown\frown}\ l)\ s$
define $k'\ l'$ **where** $k' =$ *min* $k\ l$ **and** $l' =$ *max* $k\ l$
with A **have** A': $k' < l'\ (f\ ^{\frown\frown}\ k')\ s = (f\ ^{\frown\frown}\ l')\ s\ l' < n$
by (*auto simp*: *min-def max-def*)

have $s = (f\ ^{\frown\frown}\ ((n - l') + l'))\ s$ **using** *that* ‹$l' < n$› **by** *simp*

```
    also have ... = (f ^^ (n - l')) ((f ^^ l') s) by (simp add: funpow-add)
    also have (f ^^ l') s = (f ^^ k') s by (simp add: A')
   also have (f ^^ (n - l')) ... = (f ^^ (n - l' + k')) s by (simp add: funpow-add)
    finally have (f ^^ (n - l' + k')) s = s by simp
    moreover have n - l' + k' < n 0 < n - l' + k' using A' by linarith+
    ultimately have False using that(2) by auto
  }
  then show ?thesis by (intro inj-onI) auto
qed
```

61.6 Intervals of integers

```
lemma atLeastLessThanPlusOne-atLeastAtMost-int: {l..<u+1} = {l..(u::int)}
  by (auto simp add: atLeastAtMost-def atLeastLessThan-def)

lemma atLeastPlusOneAtMost-greaterThanAtMost-int: {l+1..u} = {l<..(u::int)}
  by (auto simp add: atLeastAtMost-def greaterThanAtMost-def)

lemma atLeastPlusOneLessThan-greaterThanLessThan-int:
    {l+1..<u} = {l<..<u::int}
  by (auto simp add: atLeastLessThan-def greaterThanLessThan-def)
```

61.6.1 Finiteness

```
lemma image-atLeastZeroLessThan-int:
  assumes 0 ≤ u
  shows {(0::int)..<u} = int ` {..<nat u}
  unfolding image-def lessThan-def
proof
  show {0..<u} ⊆ {y. ∃x∈{x. x < nat u}. y = int x}
  proof
    fix x
    assume x ∈ {0..<u}
    then have x = int (nat x) and  nat x < nat u
      by (auto simp add: zless-nat-eq-int-zless [THEN sym])
    then have ∃xa<nat u. x = int xa
      using exI[of - (nat x)] by simp
    then show x ∈ {y. ∃x∈{x. x < nat u}. y = int x}
      by simp
  qed
qed (auto)


lemma finite-atLeastZeroLessThan-int: finite {(0::int)..<u}
proof (cases 0 ≤ u)
  case True
  then show ?thesis
    by (auto simp: image-atLeastZeroLessThan-int)
qed auto
```

lemma *finite-atLeastLessThan-int* [*iff*]: *finite* {*l*..<*u*::*int*}
by (*simp only*: *image-add-int-atLeastLessThan* [*symmetric*, *of l*] *finite-imageI finite-atLeastZeroLessThan-int*)

lemma *finite-atLeastAtMost-int* [*iff*]: *finite* {*l*..(*u*::*int*)}
by (*subst atLeastLessThanPlusOne-atLeastAtMost-int* [*THEN sym*], *simp*)

lemma *finite-greaterThanAtMost-int* [*iff*]: *finite* {*l*<..(*u*::*int*)}
by (*subst atLeastPlusOneAtMost-greaterThanAtMost-int* [*THEN sym*], *simp*)

lemma *finite-greaterThanLessThan-int* [*iff*]: *finite* {*l*<..<*u*::*int*}
by (*subst atLeastPlusOneLessThan-greaterThanLessThan-int* [*THEN sym*], *simp*)

61.6.2 Cardinality

lemma *card-atLeastZeroLessThan-int*: *card* {(*0*::*int*)..<*u*} = *nat u*
proof (*cases 0* ≤ *u*)
case *True*
then show *?thesis*
by (*auto simp*: *image-atLeastZeroLessThan-int card-image inj-on-def*)
qed *auto*

lemma *card-atLeastLessThan-int* [*simp*]: *card* {*l*..<*u*} = *nat* (*u* − *l*)
proof −
have *card* {*l*..<*u*} = *card* {*0*..<*u*−*l*}
apply (*subst image-add-int-atLeastLessThan* [*symmetric*])
apply (*rule card-image*)
apply (*simp add*: *inj-on-def*)
done
then show *?thesis*
by (*simp add*: *card-atLeastZeroLessThan-int*)
qed

lemma *card-atLeastAtMost-int* [*simp*]: *card* {*l*..*u*} = *nat* (*u* − *l* + *1*)
apply (*subst atLeastLessThanPlusOne-atLeastAtMost-int* [*THEN sym*])
apply (*auto simp add*: *algebra-simps*)
done

lemma *card-greaterThanAtMost-int* [*simp*]: *card* {*l*<..*u*} = *nat* (*u* − *l*)
by (*subst atLeastPlusOneAtMost-greaterThanAtMost-int* [*THEN sym*], *simp*)

lemma *card-greaterThanLessThan-int* [*simp*]: *card* {*l*<..<*u*} = *nat* (*u* − (*l* + *1*))
by (*subst atLeastPlusOneLessThan-greaterThanLessThan-int* [*THEN sym*], *simp*)

lemma *finite-M-bounded-by-nat*: *finite* {*k*. *P k* ∧ *k* < (*i*::*nat*)}
proof −
have {*k*. *P k* ∧ *k* < *i*} ⊆ {..<*i*} **by** *auto*
with *finite-lessThan*[*of i*] **show** *?thesis* **by** (*simp add*: *finite-subset*)
qed

lemma *card-less*:
assumes *zero-in-M*: $0 \in M$
shows *card* $\{k \in M.\ k < Suc\ i\} \neq 0$
proof −
from *zero-in-M* **have** $\{k \in M.\ k < Suc\ i\} \neq \{\}$ **by** *auto*
with *finite-M-bounded-by-nat* **show** *?thesis* **by** (*auto simp add*: *card-eq-0-iff*)
qed

lemma *card-less-Suc2*:
assumes $0 \notin M$ **shows** *card* $\{k.\ Suc\ k \in M \wedge k < i\} = card\ \{k \in M.\ k < Suc\ i\}$
proof −
have ∗: $[\![j \in M; j < Suc\ i]\!] \Longrightarrow j - Suc\ 0 < i \wedge Suc\ (j - Suc\ 0) \in M \wedge Suc\ 0 \leq j$ **for** *j*
by (*cases j*) (*use assms* **in** *auto*)
show *?thesis*
proof (*rule card-bij-eq*)
show *inj-on Suc* $\{k.\ Suc\ k \in M \wedge k < i\}$
by *force*
show *inj-on* $(\lambda x.\ x - Suc\ 0)\ \{k \in M.\ k < Suc\ i\}$
by (*rule inj-on-diff-nat*) (*use* ∗ **in** *blast*)
qed (*use* ∗ **in** *auto*)
qed

lemma *card-less-Suc*:
assumes $0 \in M$
shows $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = card\ \{k \in M.\ k < Suc\ i\}$
proof −
have $Suc\ (card\ \{k.\ Suc\ k \in M \wedge k < i\}) = Suc\ (card\ \{k.\ Suc\ k \in M - \{0\} \wedge k < i\})$
by *simp*
also have ... = $Suc\ (card\ \{k \in M - \{0\}.\ k < Suc\ i\})$
apply (*subst card-less-Suc2*)
using *assms* **by** *auto*
also have ... = $Suc\ (card\ (\{k \in M.\ k < Suc\ i\} - \{0\}))$
by (*force intro*: *arg-cong* [**where** *f*=*card*])
also have ... = $card\ (insert\ 0\ (\{k \in M.\ k < Suc\ i\} - \{0\}))$
by (*simp add*: *card.insert-remove*)
also have ... = $card\ \{k \in M.\ k < Suc\ i\}$
using *assms*
by (*force simp add*: *intro*: *arg-cong* [**where** *f*=*card*])
finally show *?thesis*.
qed

lemma *card-le-Suc-Max*: *finite* $S \Longrightarrow card\ S \leq Suc\ (Max\ S)$
proof (*rule classical*)
assume *finite S* **and** ¬ $Suc\ (Max\ S) \geq card\ S$
then have $Suc\ (Max\ S) < card\ S$

by *simp*
with ‹*finite S*› **have** $S \subseteq \{0..Max\ S\}$
by *auto*
hence *card* $S \leq$ *card* $\{0..Max\ S\}$
by (*intro card-mono*; *auto*)
thus *card* $S \leq Suc\ (Max\ S)$
by *simp*
qed

lemma *finite-countable-subset*:
assumes *finite* A **and** A: $A \subseteq (\bigcup i{::}nat.\ B\ i)$
obtains n **where** $A \subseteq (\bigcup i{<}n.\ B\ i)$
proof −
obtain f **where** f: $\bigwedge x.\ x \in A \Longrightarrow x \in B(f\ x)$
by (*metis in-mono UN-iff A*)
define n **where** $n = Suc\ (Max\ (f`A))$
have *finite* $(f\ `\ A)$
by (*simp add*: ‹*finite A*›)
then have $A \subseteq (\bigcup i{<}n.\ B\ i)$
unfolding *UN-iff f n-def subset-iff*
by (*meson Max-ge f imageI le-imp-less-Suc lessThan-iff*)
then show *?thesis* **..**
qed

lemma *finite-countable-equals*:
assumes *finite* A $A = (\bigcup i{::}nat.\ B\ i)$
obtains n **where** $A = (\bigcup i{<}n.\ B\ i)$
proof −
obtain n **where** $A \subseteq (\bigcup i{<}n.\ B\ i)$
proof (*rule finite-countable-subset*)
show $A \subseteq \bigcup\ (range\ B)$
by (*force simp*: *assms*)
qed (*use assms* **in** *auto*)
with *that* **show** *?thesis*
by (*force simp*: *assms*)
qed

61.7 Lemmas useful with the summation operator sum

For examples, see Algebra/poly/UnivPoly2.thy

61.7.1 Disjoint Unions

Singletons and open intervals

lemma *ivl-disj-un-singleton*:
$\{l{::}'a{::}linorder\} \cup \{l{<}..\} = \{l..\}$
$\{..{<}u\} \cup \{u{::}'a{::}linorder\} = \{..u\}$
$(l{::}'a{::}linorder) < u \Longrightarrow \{l\} \cup \{l{<}..{<}u\} = \{l..{<}u\}$

$(l::'a::linorder) < u \Longrightarrow \{l<..<u\} \cup \{u\} = \{l<..u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{l\} \cup \{l<..u\} = \{l..u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{l..<u\} \cup \{u\} = \{l..u\}$
by *auto*

One- and two-sided intervals

lemma *ivl-disj-un-one*:
$(l::'a::linorder) < u \Longrightarrow \{..l\} \cup \{l<..<u\} = \{..<u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{..<l\} \cup \{l..<u\} = \{..<u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{..l\} \cup \{l<..u\} = \{..u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{..<l\} \cup \{l..u\} = \{..u\}$
$(l::'a::linorder) \leq u \Longrightarrow \{l<..u\} \cup \{u<..\} = \{l<..\}$
$(l::'a::linorder) < u \Longrightarrow \{l<..<u\} \cup \{u..\} = \{l<..\}$
$(l::'a::linorder) \leq u \Longrightarrow \{l..u\} \cup \{u<..\} = \{l..\}$
$(l::'a::linorder) \leq u \Longrightarrow \{l..<u\} \cup \{u..\} = \{l..\}$
by *auto*

Two- and two-sided intervals

lemma *ivl-disj-un-two*:
$[\![(l::'a::linorder) < m;\ m \leq u]\!] \Longrightarrow \{l<..<m\} \cup \{m..<u\} = \{l<..<u\}$
$[\![(l::'a::linorder) \leq m;\ m < u]\!] \Longrightarrow \{l<..m\} \cup \{m<..<u\} = \{l<..<u\}$
$[\![(l::'a::linorder) \leq m;\ m \leq u]\!] \Longrightarrow \{l..<m\} \cup \{m..<u\} = \{l..<u\}$
$[\![(l::'a::linorder) \leq m;\ m < u]\!] \Longrightarrow \{l..m\} \cup \{m<..<u\} = \{l..<u\}$
$[\![(l::'a::linorder) < m;\ m \leq u]\!] \Longrightarrow \{l<..<m\} \cup \{m..u\} = \{l<..u\}$
$[\![(l::'a::linorder) \leq m;\ m \leq u]\!] \Longrightarrow \{l<..m\} \cup \{m<..u\} = \{l<..u\}$
$[\![(l::'a::linorder) \leq m;\ m \leq u]\!] \Longrightarrow \{l..<m\} \cup \{m..u\} = \{l..u\}$
$[\![(l::'a::linorder) \leq m;\ m \leq u]\!] \Longrightarrow \{l..m\} \cup \{m<..u\} = \{l..u\}$
by *auto*

lemma *ivl-disj-un-two-touch*:
$[\![(l::'a::linorder) < m;\ m < u]\!] \Longrightarrow \{l<..m\} \cup \{m..<u\} = \{l<..<u\}$
$[\![(l::'a::linorder) \leq m;\ m < u]\!] \Longrightarrow \{l..m\} \cup \{m..<u\} = \{l..<u\}$
$[\![(l::'a::linorder) < m;\ m \leq u]\!] \Longrightarrow \{l<..m\} \cup \{m..u\} = \{l<..u\}$
$[\![(l::'a::linorder) \leq m;\ m \leq u]\!] \Longrightarrow \{l..m\} \cup \{m..u\} = \{l..u\}$
by *auto*

lemmas *ivl-disj-un = ivl-disj-un-singleton ivl-disj-un-one ivl-disj-un-two ivl-disj-un-two-touch*

61.7.2 Disjoint Intersections

One- and two-sided intervals

lemma *ivl-disj-int-one*:
$\{..l::'a::order\} \cap \{l<..<u\} = \{\}$
$\{..<l\} \cap \{l..<u\} = \{\}$
$\{..l\} \cap \{l<..u\} = \{\}$
$\{..<l\} \cap \{l..u\} = \{\}$
$\{l<..u\} \cap \{u<..\} = \{\}$
$\{l<..<u\} \cap \{u..\} = \{\}$

$\{l..u\} \cap \{u<..\} = \{\}$
$\{l..<u\} \cap \{u..\} = \{\}$
by *auto*

Two- and two-sided intervals

lemma *ivl-disj-int-two*:
$\{l::'a::order<..<m\} \cap \{m..<u\} = \{\}$
$\{l<..m\} \cap \{m<..<u\} = \{\}$
$\{l..<m\} \cap \{m..<u\} = \{\}$
$\{l..m\} \cap \{m<..<u\} = \{\}$
$\{l<..<m\} \cap \{m..u\} = \{\}$
$\{l<..m\} \cap \{m<..u\} = \{\}$
$\{l..<m\} \cap \{m..u\} = \{\}$
$\{l..m\} \cap \{m<..u\} = \{\}$
by *auto*

lemmas *ivl-disj-int* = *ivl-disj-int-one ivl-disj-int-two*

61.7.3 Some Differences

lemma *ivl-diff*[*simp*]:
$i \le n \Longrightarrow \{i..<m\} - \{i..<n\} = \{n..<(m::'a::linorder)\}$
by(*auto*)

lemma (**in** *linorder*) *lessThan-minus-lessThan* [*simp*]:
$\{..< n\} - \{..< m\} = \{m ..< n\}$
by *auto*

lemma (**in** *linorder*) *atLeastAtMost-diff-ends*:
$\{a..b\} - \{a, b\} = \{a<..<b\}$
by *auto*

61.7.4 Some Subset Conditions

lemma *ivl-subset* [*simp*]: $(\{i..<j\} \subseteq \{m..<n\}) = (j \le i \vee m \le i \wedge j \le (n::'a::linorder))$
using *linorder-class.le-less-linear*[*of i n*]
by *safe* (*force intro*: *leI*)+

61.8 Generic big monoid operation over intervals

context *semiring-char-0*
begin

lemma *inj-on-of-nat* [*simp*]:
inj-on of-nat N
by (*rule inj-onI*) *simp*

lemma *bij-betw-of-nat* [*simp*]:
bij-betw of-nat N A $\longleftrightarrow$ *of-nat* ' *N* = *A*
by (*simp add*: *bij-betw-def*)

lemma *Nats-infinite*: *infinite* (**ℕ** :: *'a set*)
by (*metis Nats-def finite-imageD infinite-UNIV-char-0 inj-on-of-nat*)

end

context *comm-monoid-set*
begin

lemma *atLeastLessThan-reindex*:
F g {*h m..<h n*} = *F* (*g* ∘ *h*) {*m..<n*}
if *bij-betw h* {*m..<n*} {*h m..<h n*} **for** *m n* ::*nat*
proof −
from *that* **have** *inj-on h* {*m..<n*} **and** *h* ' {*m..<n*} = {*h m..<h n*}
by (*simp-all add*: *bij-betw-def*)
then show *?thesis*
using *reindex* [*of h* {*m..<n*} *g*] **by** *simp*
qed

lemma *atLeastAtMost-reindex*:
F g {*h m..h n*} = *F* (*g* ∘ *h*) {*m..n*}
if *bij-betw h* {*m..n*} {*h m..h n*} **for** *m n* ::*nat*
proof −
from *that* **have** *inj-on h* {*m..n*} **and** *h* ' {*m..n*} = {*h m..h n*}
by (*simp-all add*: *bij-betw-def*)
then show *?thesis*
using *reindex* [*of h* {*m..n*} *g*] **by** *simp*
qed

lemma *atLeastLessThan-shift-bounds*:
F g {*m* + *k..<n* + *k*} = *F* (*g* ∘ *plus k*) {*m..<n*}
for *m n k* :: *nat*
using *atLeastLessThan-reindex* [*of plus k m n g*]
by (*simp add*: *ac-simps*)

lemma *atLeastAtMost-shift-bounds*:
F g {*m* + *k..n* + *k*} = *F* (*g* ∘ *plus k*) {*m..n*}
for *m n k* :: *nat*
using *atLeastAtMost-reindex* [*of plus k m n g*]
by (*simp add*: *ac-simps*)

lemma *atLeast-Suc-lessThan-Suc-shift*:
F g {*Suc m..<Suc n*} = *F* (*g* ∘ *Suc*) {*m..<n*}
using *atLeastLessThan-shift-bounds* [*of - - 1*]
by (*simp add*: *plus-1-eq-Suc*)

lemma *atLeast-Suc-atMost-Suc-shift*:
F g {*Suc m..Suc n*} = *F* (*g* ∘ *Suc*) {*m..n*}
using *atLeastAtMost-shift-bounds* [*of - - 1*]

by (*simp add*: *plus-1-eq-Suc*)

lemma *atLeast-atMost-pred-shift*:
F (*g* ∘ (λ*n*. *n* − *Suc* 0)) {*Suc m*..*Suc n*} = *F g* {*m*..*n*}
unfolding *atLeast-Suc-atMost-Suc-shift* **by** *simp*

lemma *atLeast-lessThan-pred-shift*:
F (*g* ∘ (λ*n*. *n* − *Suc* 0)) {*Suc m*..<*Suc n*} = *F g* {*m*..<*n*}
unfolding *atLeast-Suc-lessThan-Suc-shift* **by** *simp*

lemma *atLeast-int-lessThan-int-shift*:
F g {*int m*..<*int n*} = *F* (*g* ∘ *int*) {*m*..<*n*}
by (*rule atLeastLessThan-reindex*)
(*simp add*: *image-int-atLeastLessThan*)

lemma *atLeast-int-atMost-int-shift*:
F g {*int m*..*int n*} = *F* (*g* ∘ *int*) {*m*..*n*}
by (*rule atLeastAtMost-reindex*)
(*simp add*: *image-int-atLeastAtMost*)

lemma *atLeast0-lessThan-Suc*:
F g {0..<*Suc n*} = *F g* {0..<*n*} ∗ *g n*
by (*simp add*: *atLeast0-lessThan-Suc ac-simps*)

lemma *atLeast0-atMost-Suc*:
F g {0..*Suc n*} = *F g* {0..*n*} ∗ *g* (*Suc n*)
by (*simp add*: *atLeast0-atMost-Suc ac-simps*)

lemma *atLeast0-lessThan-Suc-shift*:
F g {0..<*Suc n*} = *g* 0 ∗ *F* (*g* ∘ *Suc*) {0..<*n*}
by (*simp add*: *atLeast0-lessThan-Suc-eq-insert-0 atLeast-Suc-lessThan-Suc-shift*)

lemma *atLeast0-atMost-Suc-shift*:
F g {0..*Suc n*} = *g* 0 ∗ *F* (*g* ∘ *Suc*) {0..*n*}
by (*simp add*: *atLeast0-atMost-Suc-eq-insert-0 atLeast-Suc-atMost-Suc-shift*)

lemma *atLeast-Suc-lessThan*:
F g {*m*..<*n*} = *g m* ∗ *F g* {*Suc m*..<*n*} **if** *m* < *n*
proof −
from *that* **have** {*m*..<*n*} = *insert m* {*Suc m*..<*n*}
by *auto*
then show *?thesis* **by** *simp*
qed

lemma *atLeast-Suc-atMost*:
F g {*m*..*n*} = *g m* ∗ *F g* {*Suc m*..*n*} **if** *m* ≤ *n*
proof −
from *that* **have** {*m*..*n*} = *insert m* {*Suc m*..*n*}
by *auto*

then show *?thesis* **by** *simp*
qed

lemma *ivl-cong*:
$a = c \Longrightarrow b = d \Longrightarrow (\bigwedge x.\ c \leq x \Longrightarrow x < d \Longrightarrow g\ x = h\ x)$
$\Longrightarrow F\ g\ \{a..<b\} = F\ h\ \{c..<d\}$
by (*rule cong*) *simp-all*

lemma *atLeastLessThan-shift-0*:
fixes $m\ n\ p$ :: *nat*
shows $F\ g\ \{m..<n\} = F\ (g \circ plus\ m)\ \{0..<n - m\}$
using *atLeastLessThan-shift-bounds* [*of g 0 m n − m*]
by (*cases* $m \leq n$) *simp-all*

lemma *atLeastAtMost-shift-0*:
fixes $m\ n\ p$ :: *nat*
assumes $m \leq n$
shows $F\ g\ \{m..n\} = F\ (g \circ plus\ m)\ \{0..n - m\}$
using *assms atLeastAtMost-shift-bounds* [*of g 0 m n − m*] **by** *simp*

lemma *atLeastLessThan-concat*:
fixes $m\ n\ p$ :: *nat*
shows $m \leq n \Longrightarrow n \leq p \Longrightarrow F\ g\ \{m..<n\} * F\ g\ \{n..<p\} = F\ g\ \{m..<p\}$
by (*simp add*: *union-disjoint* [*symmetric*] *ivl-disj-un*)

lemma *atLeastLessThan-rev*:
$F\ g\ \{n..<m\} = F\ (\lambda i.\ g\ (m + n - Suc\ i))\ \{n..<m\}$
by (*rule reindex-bij-witness* [**where** $i=\lambda i.\ m + n - Suc\ i$ **and** $j=\lambda i.\ m + n -$
Suc i], *auto*)

lemma *atLeastAtMost-rev*:
fixes $n\ m$ :: *nat*
shows $F\ g\ \{n..m\} = F\ (\lambda i.\ g\ (m + n - i))\ \{n..m\}$
by (*rule reindex-bij-witness* [**where** $i=\lambda i.\ m + n - i$ **and** $j=\lambda i.\ m + n - i$])
auto

lemma *atLeastLessThan-rev-at-least-Suc-atMost*:
$F\ g\ \{n..<m\} = F\ (\lambda i.\ g\ (m + n - i))\ \{Suc\ n..m\}$
unfolding *atLeastLessThan-rev* [*of g n m*]
by (*cases m*) (*simp-all add*: *atLeast-Suc-atMost-Suc-shift atLeastLessThanSuc-atLeastAtMost*)

end

61.9 Summation indexed over intervals

syntax (*ASCII*)
-from-to-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder SUM*››*SUM - =*
-..-./ -)› [*0,0,0,10*] *10*)
-from-upto-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder SUM*››*SUM -*

= -..<-./ -)› [0,0,0,10] 10)
-upt-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation=‹binder SUM››SUM* -<-./ -)› [0,0,10] 10)
-upto-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation=‹binder SUM››SUM* -<=-./ -)› [0,0,10] 10)

syntax (*latex-sum* **output**)
-from-to-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(3$\sum_{-\ =\ -}$ -)› [0,0,0,10] 10)
-from-upto-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(3$\sum_{-\ =\ -}^{<-}$ -)› [0,0,0,10] 10)
-upt-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(3$\sum_{-\ <\ -}$ -)› [0,0,10] 10)
-upto-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(3$\sum_{-\ \leq\ -}$ -)› [0,0,10] 10)

syntax
-from-to-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* $\sum$ ››$\sum$ - = -..-./ -)› [0,0,0,10] 10)
-from-upto-sum :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* $\sum$ ››$\sum$ - = -..<-./ -)› [0,0,0,10] 10)
-upt-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* $\sum$ ››$\sum$ -<-./ -)› [0,0,10] 10)
-upto-sum :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder* $\sum$ ››$\sum$ -≤-./ -)› [0,0,10] 10)

syntax-consts
-from-to-sum -from-upto-sum -upt-sum -upto-sum == *sum*

translations
$\sum x{=}a..b.\ t$ == *CONST sum* ($\lambda x.\ t$) {*a..b*}
$\sum x{=}a..{<}b.\ t$ == *CONST sum* ($\lambda x.\ t$) {*a..<b*}
$\sum i{\leq}n.\ t$ == *CONST sum* ($\lambda i.\ t$) {*..n*}
$\sum i{<}n.\ t$ == *CONST sum* ($\lambda i.\ t$) {*..<n*}

The above introduces some pretty alternative syntaxes for summation over intervals:

Old	New	LaTeX
$\sum x{\in}\{a..b\}.\ e$	$\sum x = a..b.\ e$	$\sum_{x\ =\ a}^{b}\ e$
$\sum x{\in}\{a..{<}b\}.\ e$	$\sum x = a..{<}b.\ e$	$\sum_{x\ =\ a}^{<b}\ e$
$\sum x{\in}\{..b\}.\ e$	$\sum x{\leq}b.\ e$	$\sum_{x\ \leq\ b}\ e$
$\sum x{\in}\{..{<}b\}.\ e$	$\sum x{<}b.\ e$	$\sum_{x\ <\ b}\ e$

The left column shows the term before introduction of the new syntax, the middle column shows the new (default) syntax, and the right column shows a special syntax. The latter is only meaningful for latex output and has to be activated explicitly by setting the print mode to *latex-sum* (e.g. via *mode* = *latex-sum* in antiquotations). It is not the default LaTeX output because

it only works well with italic-style formulae, not tt-style.

Note that for uniformity on *nat* it is better to use $\sum x = 0..<n.\ e$ rather than $\sum x{<}n.\ e$: *sum* may not provide all lemmas available for $\{m..<n\}$ also in the special form for $\{..<n\}$.

This congruence rule should be used for sums over intervals as the standard theorem *sum.cong* does not work well with the simplifier who adds the unsimplified premise $x \in B$ to the context.

context *comm-monoid-set*
begin

lemma *zero-middle*:
  assumes *1* $\leq$ *p k* $\leq$ *p*
  shows *F* (*λj. if j < k then g j else if j = k then* **1** *else h* (*j − Suc 0*)) {*..p*}
      = *F* (*λj. if j < k then g j else h j*) {*..p − Suc 0*} (**is** *?lhs = ?rhs*)
proof −
  have [*simp*]: {*..p − Suc 0*} ∩ {*j. j < k*} = {*..<k*} {*..p − Suc 0*} ∩ − {*j. j < k*}
= {*k..p − Suc 0*}
    using *assms* **by** *auto*
  have *?lhs = F g* {*..<k*} ∗ *F* (*λj. if j = k then* **1** *else h* (*j − Suc 0*)) {*k..p*}
    using *union-disjoint* [*of* {*..<k*} {*k..p*}] *assms*
    by (*simp add*: *ivl-disj-int-one ivl-disj-un-one*)
  also have ... = *F g* {*..<k*} ∗ *F* (*λj. h* (*j − Suc 0*)) {*Suc k..p*}
    by (*simp add*: *atLeast-Suc-atMost* [*of k p*] *assms*)
  also have ... = *F g* {*..<k*} ∗ *F h* {*k .. p − Suc 0*}
    using *reindex* [*of Suc* {*k..p − Suc 0*}] *assms* **by** *simp*
  also have ... = *?rhs*
    by (*simp add*: *If-cases*)
  finally show *?thesis* **.**
qed

lemma *atMost-Suc* [*simp*]:
  F g {*..Suc n*} = *F g* {*..n*} ∗ *g* (*Suc n*)
  by (*simp add*: *atMost-Suc ac-simps*)

lemma *lessThan-Suc* [*simp*]:
  F g {*..<Suc n*} = *F g* {*..<n*} ∗ *g n*
  by (*simp add*: *lessThan-Suc ac-simps*)

lemma *cl-ivl-Suc* [*simp*]:
  F g {*m..Suc n*} = (*if Suc n < m then* **1** *else F g* {*m..n*} ∗ *g*(*Suc n*))
  by (*auto simp*: *ac-simps atLeastAtMostSuc-conv*)

lemma *op-ivl-Suc* [*simp*]:
  F g {*m..<Suc n*} = (*if n < m then* **1** *else F g* {*m..<n*} ∗ *g*(*n*))
  by (*auto simp*: *ac-simps atLeastLessThanSuc*)

lemma *head*:

fixes n :: *nat*
assumes *mn*: $m \leq n$
shows $F\ g\ \{m..n\} = g\ m * F\ g\ \{m<..n\}$ (**is** *?lhs* = *?rhs*)
proof −
from *mn*
have $\{m..n\} = \{m\} \cup \{m<..n\}$
by (*auto intro*: *ivl-disj-un-singleton*)
hence *?lhs* = $F\ g\ (\{m\} \cup \{m<..n\})$
by (*simp add*: *atLeast0LessThan*)
also have ... = *?rhs* **by** *simp*
finally show *?thesis* **.**
qed

lemma *last-plus*:
fixes *n*::*nat* **shows** $m \leq n \Longrightarrow F\ g\ \{m..n\} = g\ n * F\ g\ \{m..<n\}$
by (*cases n*) (*auto simp*: *atLeastLessThanSuc-atLeastAtMost commute*)

lemma *head-if*:
fixes n :: *nat*
shows $F\ g\ \{m..n\} = (if\ n < m\ then\ \mathbf{1}\ else\ \ F\ g\ \{m..<n\} * g(n))$
by (*simp add*: *commute last-plus*)

lemma *ub-add-nat*:
assumes (*m*::*nat*) ≤ *n* + *1*
shows $F\ g\ \{m..n + p\} = F\ g\ \{m..n\} * F\ g\ \{n + 1..n + p\}$
proof−
have $\{m\ ..\ n+p\} = \{m..n\} \cup \{n+1..n+p\}$ **using** ‹$m \leq n+1$› **by** *auto*
thus *?thesis* **by** (*auto simp*: *ivl-disj-int union-disjoint atLeastSucAtMost-greaterThanAtMost*)
qed

lemma *nat-group*:
fixes *k*::*nat* **shows** $F\ (\lambda m.\ F\ g\ \{m * k\ ..<\ m*k + k\})\ \{..<n\} = F\ g\ \{..<\ n * k\}$
proof (*cases k*)
case (*Suc l*)
then have *k* > *0*
by *auto*
then show *?thesis*
by (*induct n*) (*simp-all add*: *atLeastLessThan-concat add.commute atLeast0LessThan*[*symmetric*])
qed *auto*

lemma *triangle-reindex*:
fixes n :: *nat*
shows $F\ (\lambda(i,j).\ g\ i\ j)\ \{(i,j).\ i+j < n\} = F\ (\lambda k.\ F\ (\lambda i.\ g\ i\ (k - i))\ \{..k\})$
$\{..<n\}$
apply (*simp add*: *Sigma*)
apply (*rule reindex-bij-witness*[**where** *i*=λ(*i*, *j*). (*i*+*j*, *i*) **and** *i*=λ(*k*, *i*). (*i*, *k* −
i)])
apply *auto*
done

lemma *triangle-reindex-eq*:
fixes $n :: nat$
shows $F\ (\lambda(i,j).\ g\ i\ j)\ \{(i,j).\ i+j \leq n\} = F\ (\lambda k.\ F\ (\lambda i.\ g\ i\ (k - i))\ \{..k\})\ \{..n\}$
using *triangle-reindex* [*of g Suc n*]
by (*simp only*: *Nat.less-Suc-eq-le lessThan-Suc-atMost*)

lemma *nat-diff-reindex*: $F\ (\lambda i.\ g\ (n - Suc\ i))\ \{..<n\} = F\ g\ \{..<n\}$
by (*rule reindex-bij-witness*[**where** $i=\lambda i.\ n - Suc\ i$ **and** $j=\lambda i.\ n - Suc\ i$]) *auto*

lemma *shift-bounds-nat-ivl*:
$F\ g\ \{m+k..<n+k\} = F\ (\lambda i.\ g(i + k))\{m..<n::nat\}$
by (*induct n*, *auto simp*: *atLeastLessThanSuc*)

lemma *shift-bounds-cl-nat-ivl*:
$F\ g\ \{m+k..n+k\} = F\ (\lambda i.\ g(i + k))\{m..n::nat\}$
by (*rule reindex-bij-witness*[**where** $i=\lambda i.\ i + k$ **and** $j=\lambda i.\ i - k$]) *auto*

corollary *shift-bounds-cl-Suc-ivl*:
$F\ g\ \{Suc\ m..Suc\ n\} = F\ (\lambda i.\ g(Suc\ i))\{m..n\}$
by (*simp add*: *shift-bounds-cl-nat-ivl*[**where** $k=Suc\ 0$, *simplified*])

corollary *Suc-reindex-ivl*: $m \leq n \Longrightarrow F\ g\ \{m..n\} * g\ (Suc\ n) = g\ m * F\ (\lambda i.\ g\ (Suc\ i))\ \{m..n\}$
by (*simp add*: *assoc atLeast-Suc-atMost flip*: *shift-bounds-cl-Suc-ivl*)

corollary *shift-bounds-Suc-ivl*:
$F\ g\ \{Suc\ m..<Suc\ n\} = F\ (\lambda i.\ g(Suc\ i))\{m..<n\}$
by (*simp add*: *shift-bounds-nat-ivl*[**where** $k=Suc\ 0$, *simplified*])

lemma *atMost-Suc-shift*:
shows $F\ g\ \{..Suc\ n\} = g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\}$
proof (*induct n*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc n*) **note** *IH* = *this*
have $F\ g\ \{..Suc\ (Suc\ n)\} = F\ g\ \{..Suc\ n\} * g\ (Suc\ (Suc\ n))$
by (*rule atMost-Suc*)
also have $F\ g\ \{..Suc\ n\} = g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\}$
by (*rule IH*)
also have $g\ 0 * F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\} * g\ (Suc\ (Suc\ n)) = g\ 0 * (F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\} * g\ (Suc\ (Suc\ n)))$
by (*rule assoc*)
also have $F\ (\lambda i.\ g\ (Suc\ i))\ \{..n\} * g\ (Suc\ (Suc\ n)) = F\ (\lambda i.\ g\ (Suc\ i))\ \{..Suc\ n\}$
by (*rule atMost-Suc* [*symmetric*])
finally show *?case* **.**
qed

lemma *lessThan-Suc-shift*:
*F g {..<Suc n} = g 0 * F (λi. g (Suc i)) {..<n}*
by (*induction n*) (*simp-all add*: *ac-simps*)

lemma *atMost-shift*:
*F g {..n} = g 0 * F (λi. g (Suc i)) {..<n}*
by (*metis atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost*

atLeastSucAtMost-greaterThanAtMost le0 head shift-bounds-Suc-ivl)

lemma *nested-swap*:
F (λi. F (λj. a i j) {0..<i}) {0..n} = F (λj. F (λi. a i j) {Suc j..n}) {0..<n}
by (*induction n*) (*auto simp*: *distrib*)

lemma *nested-swap′*:
F (λi. F (λj. a i j) {..<i}) {..n} = F (λj. F (λi. a i j) {Suc j..n}) {..<n}
by (*induction n*) (*auto simp*: *distrib*)

lemma *atLeast1-atMost-eq*:
F g {Suc 0..n} = F (λk. g (Suc k)) {..<n}
proof −
have *F g {Suc 0..n} = F g (Suc ‘ {..<n})*
by (*simp add*: *image-Suc-lessThan*)
also have *... = F (λk. g (Suc k)) {..<n}*
by (*simp add*: *reindex*)
finally show *?thesis* **.**
qed

lemma *atLeastLessThan-Suc*: *a ≤ b ⟹ F g {a..<Suc b} = F g {a..<b} * g b*
by (*simp add*: *atLeastLessThanSuc commute*)

lemma *nat-ivl-Suc′*:
assumes *m ≤ Suc n*
shows *F g {m..Suc n} = g (Suc n) * F g {m..n}*
proof −
from *assms* **have** *{m..Suc n} = insert (Suc n) {m..n}* **by** *auto*
also have *F g ... = g (Suc n) * F g {m..n}* **by** *simp*
finally show *?thesis* **.**
qed

lemma *in-pairs*: *F g {2*m..Suc(2*n)} = F (λi. g(2*i) * g(Suc(2*i))) {m..n}*
proof (*induction n*)
case *0*
show *?case*
by (*cases m=0*) *auto*
next
case (*Suc n*)
then show *?case*
by (*auto simp*: *assoc split*: *if-split-asm*)

qed

lemma *in-pairs-0*: $F\ g\ \{..Suc(2{*}n)\} = F\ (\lambda i.\ g(2{*}i) * g(Suc(2{*}i)))\ \{..n\}$
using *in-pairs* [*of* - *0 n*] **by** (*simp add*: *atLeast0AtMost*)

end

lemma *card-sum-le-nat-sum*: $\sum\ \{0..<card\ S\} \leq \sum\ S$
proof (*cases finite S*)
case *True*
then show *?thesis*
proof (*induction card S arbitrary*: *S*)
case (*Suc x*)
then have $Max\ S \geq x$ **using** *card-le-Suc-Max* **by** *fastforce*
let $?S' = S - \{Max\ S\}$
from *Suc* **have** $Max\ S \in S$ **by** (*auto intro*: *Max-in*)
hence *cards*: $card\ S = Suc\ (card\ ?S')$
using ‹*finite S*› **by** (*intro card.remove*; *auto*)
hence $\sum\ \{0..<card\ ?S'\} \leq \sum\ ?S'$
using *Suc* **by** (*intro Suc*; *auto*)

hence $\sum\ \{0..<card\ ?S'\} + x \leq \sum\ ?S' + Max\ S$
using ‹$Max\ S \geq x$› **by** *simp*
also have $... = \sum\ S$
using *sum.remove*[*OF* ‹*finite S*› ‹$Max\ S \in S$›, **where** $g{=}\lambda x.\ x$]
by *simp*
finally show *?case*
using *cards Suc* **by** *auto*
qed *simp*
qed *simp*

The triangular double-sum exchange, used for associativity and the key partial-sum estimate.

lemma *sum-triangle-exchange*:
fixes $f :: nat \Rightarrow nat \Rightarrow 'c{::}comm\text{-}monoid\text{-}add$
shows $(\sum d{\leq}N.\ \sum i{\leq}d.\ f\ i\ (d - i)) = (\sum i{\leq}N.\ \sum e{\leq}N - i.\ f\ i\ e)$
proof −
have *L*: $(\sum d{\leq}N.\ \sum i{\leq}d.\ f\ i\ (d - i)) =$
$(\sum (d,\ i){\in}(SIGMA\ d{:}\{..N\}.\ \{..d\}).\ f\ i\ (d - i))$
by (*rule sum.Sigma*[*OF finite-atMost*]) *simp*
also have $\ldots = (\sum (i,\ e){\in}(SIGMA\ i{:}\{..N\}.\ \{..N - i\}).\ f\ i\ e)$
proof (*rule sum.reindex-bij-witness*[**where** $i = \lambda(i,\ e).\ (i + e,\ i)$ **and** $j = \lambda(d,\ i).\ (i,\ d - i)$])
fix *a* **assume** *a*: $a \in (SIGMA\ d{:}\{..N\}.\ \{..d\})$
obtain *d i* **where** *di*: $a = (d,\ i)$ **by** (*cases a*)
have *dN*: $d \leq N$ **and** *idd*: $i \leq d$ **using** *a di* **by** *auto*
show (*case case a of* $(d,\ i) \Rightarrow (i,\ d - i)$ *of* $(i,\ e) \Rightarrow (i + e,\ i)) = a$
using *di idd* **by** *simp*
show (*case a of* $(d,\ i) \Rightarrow (i,\ d - i)) \in (SIGMA\ i{:}\{..N\}.\ \{..N - i\})$

```
    using di dN idd by auto
  next
    fix b assume b: b ∈ (SIGMA i:{..N}. {..N − i})
    obtain i e where ie: b = (i, e) by (cases b)
    have iN: i ≤ N and eN: e ≤ N − i using b ie by auto
    show (case case b of (i, e) ⇒ (i + e, i) of (d, i) ⇒ (i, d − i)) = b
      using ie by simp
    show (case b of (i, e) ⇒ (i + e, i)) ∈ (SIGMA d:{..N}. {..d})
      using ie iN eN by auto
  qed auto
  also have ... = (∑ i≤N. ∑ e≤N − i. f i e)
    by (intro sum.Sigma [symmetric]) auto
  finally show ?thesis .
qed

lemma sum-natinterval-diff:
  fixes f:: nat ⇒ ('a::ab-group-add)
  shows  sum (λk. f k − f(k + 1)) {(m::nat) .. n} =
          (if m ≤ n then f m − f(n + 1) else 0)
by (induct n, auto simp add: algebra-simps not-le le-Suc-eq)

lemma sum-diff-nat-ivl:
  fixes f :: nat ⇒ 'a::ab-group-add
  shows ⟦ m ≤ n; n ≤ p ⟧ ⟹ sum f {m..<p} − sum f {m..<n} = sum f {n..<p}
  using sum.atLeastLessThan-concat [of m n p f,symmetric]
  by (simp add: ac-simps)

lemma sum-diff-distrib: ∀ x. Q x ≤ P x  ⟹ (∑ x<n. P x) − (∑ x<n. Q x) =
(∑ x<n. P x − Q x :: nat)
  by (subst sum-subtractf-nat) auto
```

61.9.1 Shifting bounds

```
context comm-monoid-add
begin

context
  fixes f :: nat ⇒ 'a
  assumes f 0 = 0
begin

lemma sum-shift-lb-Suc0-0-upt:
  sum f {Suc 0..<k} = sum f {0..<k}
proof (cases k)
  case 0
  then show ?thesis
    by simp
next
  case (Suc k)
```

```
  moreover have {0..<Suc k} = insert 0 {Suc 0..<Suc k}
    by auto
  ultimately show ?thesis
    using ‹f 0 = 0› by simp
qed

lemma sum-shift-lb-Suc0-0: sum f {Suc 0..k} = sum f {0..k}
proof (cases k)
  case 0
  with ‹f 0 = 0› show ?thesis
    by simp
next
  case (Suc k)
  moreover have {0..Suc k} = insert 0 {Suc 0..Suc k}
    by auto
  ultimately show ?thesis
    using ‹f 0 = 0› by simp
qed

end

end

lemma sum-Suc-diff:
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ Suc n
  shows (∑ i = m..n. f(Suc i) − f i) = f (Suc n) − f m
using assms by (induct n) (auto simp: le-Suc-eq)

lemma sum-Suc-diff':
  fixes f :: nat ⇒ 'a::ab-group-add
  assumes m ≤ n
  shows (∑ i = m..<n. f (Suc i) − f i) = f n − f m
using assms by (induct n) (auto simp: le-Suc-eq)

lemma sum-diff-split:
  fixes f:: nat ⇒ 'a::ab-group-add
  assumes m ≤ n
  shows (∑ i≤n. f i) − (∑ i<m. f i) = (∑ i≤n − m. f(n − i))
proof −
  have ⋀i. i ≤ n−m ⟹ ∃ k≥m. k ≤ n ∧ i = n−k
    by (metis Nat.le-diff-conv2 add.commute ‹m≤n› diff-diff-cancel diff-le-self or-
der.trans)
  then have eq: {..n−m} = (−)n ‘ {m..n}
    by force
  have inj: inj-on ((−)n) {m..n}
    by (auto simp: inj-on-def)
  have (∑ i≤n − m. f(n − i)) = (∑ i=m..n. f i)
    by (simp add: eq sum.reindex-cong [OF inj])
```

also have ... = $(\sum i{\le}n.\ f\ i) - (\sum i{<}m.\ f\ i)$
using *sum-diff-nat-ivl*[*of 0 m Suc n f*] *assms*
by (*simp only*: *atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost*)
finally show *?thesis* **by** *metis*
qed

lemma *prod-divide-nat-ivl*:
fixes f :: *nat* ⇒ 'a::*idom-divide*
shows ⟦ $m \le n$; $n \le p$; *prod f* {*m*..<*n*} ≠ *0*⟧ ⟹ *prod f* {*m*..<*p*} *div prod f* {*m*..<*n*} = *prod f* {*n*..<*p*}
using *prod.atLeastLessThan-concat* [*of m n p f*,*symmetric*]
by (*simp add*: *ac-simps*)

lemma *prod-divide-split*:
fixes *f*:: *nat* ⇒ 'a::*idom-divide*
assumes $m \le n$ *prod f* {..<*m*} ≠ *0*
shows (*prod f* {..*n*}) *div* (*prod f* {..<*m*}) = *prod* (λ*i*. *f*(*n* − *i*)) {..*n* − *m*}
proof −
have ⋀*i*. $i \le n{-}m \Longrightarrow \exists k{\ge}m.\ k \le n \wedge i = n{-}k$
by (*metis Nat.le-diff-conv2 add.commute* ‹*m*≤*n*› *diff-diff-cancel diff-le-self order.trans*)
then have *eq*: {..*n*−*m*} = (−)*n* ' {*m*..*n*}
by *force*
have *inj*: *inj-on* ((−)*n*) {*m*..*n*}
by (*auto simp*: *inj-on-def*)
have *prod* (λ*i*. *f*(*n* − *i*)) {..*n* − *m*} = *prod f* {*m*..*n*}
by (*simp add*: *eq prod.reindex-cong* [*OF inj*])
also have ... = *prod f* {..*n*} *div prod f* {..<*m*}
using *prod-divide-nat-ivl*[*of 0 m Suc n f*] *assms*
by (*force simp*: *atLeast0AtMost atLeast0LessThan atLeastLessThanSuc-atLeastAtMost*)
finally show *?thesis* **by** *metis*
qed

61.9.2 Telescoping sums

lemma *sum-telescope*:
fixes *f*::*nat* ⇒ 'a::*ab-group-add*
shows *sum* (λ*i*. *f i* − *f* (*Suc i*)) {.. *i*} = *f 0* − *f* (*Suc i*)
by (*induct i*) *simp-all*

lemma *sum-telescope''*:
assumes $m \le n$
shows $(\sum k{\in}\{Suc\ m..n\}.\ f\ k - f\ (k - 1)) = f\ n - (f\ m :: 'a :: ab\text{-}group\text{-}add)$
by (*rule dec-induct*[*OF assms*]) (*simp-all add*: *algebra-simps*)

lemma *sum-lessThan-telescope*:
$(\sum n{<}m.\ f\ (Suc\ n) - f\ n :: 'a :: ab\text{-}group\text{-}add) = f\ m - f\ 0$
by (*induction m*) (*simp-all add*: *algebra-simps*)

lemma *sum-lessThan-telescope'*:
(∑ *n*<*m*. *f n* − *f* (*Suc n*) :: *'a* :: *ab-group-add*) = *f* 0 − *f m*
by (*induction m*) (*simp-all add*: *algebra-simps*)

61.9.3 The formula for geometric sums

lemma *sum-power2*: (∑ *i*=0..<*k*. (2::*nat*)^*i*) = 2^*k*−1
by (*induction k*) (*auto simp*: *mult-2*)

lemma *geometric-sum*:
assumes *x* ≠ 1
shows (∑ *i*<*n*. *x* ^ *i*) = (*x* ^ *n* − 1) / (*x* − 1::*'a*::*field*)
proof −
from *assms* **obtain** *y* **where** *y* = *x* − 1 **and** *y* ≠ 0 **by** *simp-all*
moreover have (∑ *i*<*n*. (*y* + 1) ^ *i*) = ((*y* + 1) ^ *n* − 1) / *y*
by (*induct n*) (*simp-all add*: *field-simps* ‹*y* ≠ 0›)
ultimately show *?thesis* **by** *simp*
qed

lemma *geometric-sum-less*:
fixes *x* :: *'a*::*linordered-field*
assumes 0 < *x x* < 1 *finite S*
shows (∑ *i*∈*S*. *x* ^ *i*) < 1 / (1 − *x*)
proof −
define *n* **where** *n* ≡ *Suc* (*Max S*)
have (∑ *i*∈*S*. *x* ^ *i*) ≤ (∑ *i*<*n*. *x* ^ *i*)
unfolding *n-def* **using** *assms* **by** (*fastforce intro*!: *sum-mono2 le-imp-less-Suc*)
also have ... = (1 − *x* ^ *n*) / (1 − *x*)
using *assms* **by** (*simp add*: *geometric-sum field-simps*)
also have ... < 1 / (1−*x*)
using *assms* **by** (*simp add*: *field-simps power-Suc-less*)
finally show *?thesis* **.**
qed

lemma *diff-power-eq-sum*:
fixes *y* :: *'a*::{*comm-ring*,*monoid-mult*}
shows
x ^ (*Suc n*) − *y* ^ (*Suc n*) =
(*x* − *y*) * (∑ *p*<*Suc n*. (*x* ^ *p*) * *y* ^ (*n* − *p*))
proof (*induct n*)
case (*Suc n*)
have *x* ^ *Suc* (*Suc n*) − *y* ^ *Suc* (*Suc n*) = *x* * (*x* * *x*^*n*) − *y* * (*y* * *y* ^ *n*)
by *simp*
also have ... = *y* * (*x* ^ (*Suc n*) − *y* ^ (*Suc n*)) + (*x* − *y*) * (*x* * *x*^*n*)
by (*simp add*: *algebra-simps*)
also have ... = *y* * ((*x* − *y*) * (∑ *p*<*Suc n*. (*x* ^ *p*) * *y* ^ (*n* − *p*))) + (*x* − *y*)
* (*x* * *x*^*n*)
by (*simp only*: *Suc*)
also have ... = (*x* − *y*) * (*y* * (∑ *p*<*Suc n*. (*x* ^ *p*) * *y* ^ (*n* − *p*))) + (*x* − *y*)

$* (x * x\hat{}n)$
by (*simp only*: *mult.left-commute*)
also have ... = $(x - y) * (\sum p{<}Suc\ (Suc\ n).\ x\ \hat{}\ p * y\ \hat{}\ (Suc\ n - p))$
by (*simp add*: *field-simps Suc-diff-le sum-distrib-right sum-distrib-left*)
finally show *?case* **.**
qed *simp*

corollary *power-diff-sumr2*: — *COMPLEX-POLYFUN* in HOL Light
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
shows $x\hat{}n - y\hat{}n = (x - y) * (\sum i{<}n.\ y\hat{}(n - Suc\ i) * x\hat{}i)$
using *diff-power-eq-sum*[*of x n − 1 y*]
by (*cases n = 0*) (*simp-all add*: *field-simps*)

lemma *power-diff-1-eq*:
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
shows $x\hat{}n - 1 = (x - 1) * (\sum i{<}n.\ (x\hat{}i))$
using *diff-power-eq-sum* [*of x - 1*]
by (*cases n*) *auto*

lemma *one-diff-power-eq'*:
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
shows $1 - x\hat{}n = (1 - x) * (\sum i{<}n.\ x\hat{}(n - Suc\ i))$
using *diff-power-eq-sum* [*of 1 - x*]
by (*cases n*) *auto*

lemma *one-diff-power-eq*:
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
shows $1 - x\hat{}n = (1 - x) * (\sum i{<}n.\ x\hat{}i)$
by (*metis one-diff-power-eq' sum.nat-diff-reindex*)

lemma *sum-gp-basic*:
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
shows $(1 - x) * (\sum i{\leq}n.\ x\hat{}i) = 1 - x\hat{}Suc\ n$
by (*simp only*: *one-diff-power-eq lessThan-Suc-atMost*)

lemma *sum-power-shift*:
fixes x :: *'a*::{*comm-ring*,*monoid-mult*}
assumes $m \leq n$
shows $(\sum i{=}m..n.\ x\hat{}i) = x\hat{}m * (\sum i{\leq}n-m.\ x\hat{}i)$
proof −
have $(\sum i{=}m..n.\ x\hat{}i) = x\hat{}m * (\sum i{=}m..n.\ x\hat{}(i-m))$
by (*simp add*: *sum-distrib-left power-add* [*symmetric*])
also have $(\sum i{=}m..n.\ x\hat{}(i-m)) = (\sum i{\leq}n-m.\ x\hat{}i)$
using ‹$m \leq n$› **by** (*intro sum.reindex-bij-witness*[**where** $j{=}\lambda i.\ i - m$ **and** $i{=}\lambda i.\ i + m$]) *auto*
finally show *?thesis* **.**
qed

lemma *sum-gp-multiplied*:

fixes $x :: 'a::\{comm\text{-}ring,monoid\text{-}mult\}$
assumes $m \leq n$
shows $(1 - x) * (\sum i{=}m..n.\ x\hat{}i) = x\hat{}m - x\hat{}Suc\ n$
proof −
have $(1 - x) * (\sum i{=}m..n.\ x\hat{}i) = x\hat{}m * (1 - x) * (\sum i{\leq}n{-}m.\ x\hat{}i)$
by (*metis mult.assoc mult.commute assms sum-power-shift*)
also have ... $= x\hat{}m * (1 - x\hat{}Suc(n{-}m))$
by (*metis mult.assoc sum-gp-basic*)
also have ... $= x\hat{}m - x\hat{}Suc\ n$
using *assms*
by (*simp add*: *algebra-simps*) (*metis le-add-diff-inverse power-add*)
finally show *?thesis* **.**
qed

lemma *sum-gp*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{=}m..n.\ x\hat{}i) =$
(*if* $n < m$ *then* 0
else if $x = 1$ *then of-nat*$((n + 1) - m)$
else $(x\hat{}m - x\hat{}Suc\ n) / (1 - x)$)
proof (*cases* $n < m$)
case *False*
assume $*$: $\neg\ n < m$
then show *?thesis*
proof (*cases* $x = 1$)
case *False*
assume $x \neq 1$
then have *not-zero*: $1 - x \neq 0$
by *auto*
have $(1 - x) * (\sum i{=}m..n.\ x\hat{}i) = x\ \hat{}\ m - x * x\ \hat{}\ n$
using *sum-gp-multiplied* [*of m n x*] $*$ **by** *auto*
then have $(\sum i{=}m..n.\ x\hat{}i) = (x\ \hat{}\ m - x * x\ \hat{}\ n) / (1 - x)$
using *nonzero-divide-eq-eq mult.commute not-zero*
by *metis*
then show *?thesis*
by *auto*
qed (*auto*)
qed (*auto*)

61.9.4 Geometric progressions

lemma *sum-gp0*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{\leq}n.\ x\hat{}i) = (if\ x = 1\ then\ of\text{-}nat(n + 1)\ else\ (1 - x\hat{}Suc\ n) / (1 - x))$
using *sum-gp-basic*[*of x n*]
by (*simp add*: *mult.commute field-split-simps*)

lemma *sum-power-add*:

fixes $x :: 'a::\{comm\text{-}ring,monoid\text{-}mult\}$
shows $(\sum i{\in}I.\ x\hat{}(m+i)) = x\hat{}m * (\sum i{\in}I.\ x\hat{}i)$
by (*simp add*: *sum-distrib-left power-add*)

lemma *sum-gp-offset*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{=}m..m+n.\ x\hat{}i) =$
$(if\ x = 1\ then\ of\text{-}nat\ n + 1\ else\ x\hat{}m * (1 - x\hat{}Suc\ n) / (1 - x))$
using *sum-gp* [*of x m m+n*]
by (*auto simp*: *power-add algebra-simps*)

lemma *sum-gp-strict*:
fixes $x :: 'a::\{comm\text{-}ring,division\text{-}ring\}$
shows $(\sum i{<}n.\ x\hat{}i) = (if\ x = 1\ then\ of\text{-}nat\ n\ else\ (1 - x\hat{}n) / (1 - x))$
by (*induct n*) (*auto simp*: *algebra-simps field-split-simps*)

61.9.5 The formulae for arithmetic sums

context *comm-semiring-1*
begin

lemma *double-gauss-sum*:
$2 * (\sum i = 0..n.\ of\text{-}nat\ i) = of\text{-}nat\ n * (of\text{-}nat\ n + 1)$
by (*induct n*) (*simp-all add*: *sum.atLeast0-atMost-Suc algebra-simps left-add-twice*)

lemma *double-gauss-sum-from-Suc-0*:
$2 * (\sum i = Suc\ 0..n.\ of\text{-}nat\ i) = of\text{-}nat\ n * (of\text{-}nat\ n + 1)$
proof −
have *sum of-nat* $\{Suc\ 0..n\} = sum\ of\text{-}nat\ (insert\ 0\ \{Suc\ 0..n\})$
by *simp*
also have $\ldots = sum\ of\text{-}nat\ \{0..n\}$
by (*cases n*) (*simp-all add*: *atLeast0-atMost-Suc-eq-insert-0*)
finally show *?thesis*
by (*simp add*: *double-gauss-sum*)
qed

lemma *double-arith-series*:
$2 * (\sum i = 0..n.\ a + of\text{-}nat\ i * d) = (of\text{-}nat\ n + 1) * (2 * a + of\text{-}nat\ n * d)$
proof −
have $(\sum i = 0..n.\ a + of\text{-}nat\ i * d) = ((\sum i = 0..n.\ a) + (\sum i = 0..n.\ of\text{-}nat\ i * d))$
by (*rule sum.distrib*)
also have $\ldots = (of\text{-}nat\ (Suc\ n) * a + d * (\sum i = 0..n.\ of\text{-}nat\ i))$
by (*simp add*: *sum-distrib-left algebra-simps*)
finally show *?thesis*
by (*simp add*: *algebra-simps double-gauss-sum left-add-twice*)
qed

end

context *linordered-euclidean-semiring*
begin

lemma *gauss-sum*:
$(\sum i = 0..n.\ \textit{of-nat}\ i) = \textit{of-nat}\ n * (\textit{of-nat}\ n + 1)\ \textit{div}\ 2$
using *double-gauss-sum* [*of n*, *symmetric*] **by** *simp*

lemma *gauss-sum-from-Suc-0*:
$(\sum i = \textit{Suc}\ 0..n.\ \textit{of-nat}\ i) = \textit{of-nat}\ n * (\textit{of-nat}\ n + 1)\ \textit{div}\ 2$
using *double-gauss-sum-from-Suc-0* [*of n*, *symmetric*] **by** *simp*

lemma *arith-series*:
$(\sum i = 0..n.\ a + \textit{of-nat}\ i * d) = (\textit{of-nat}\ n + 1) * (2 * a + \textit{of-nat}\ n * d)\ \textit{div}\ 2$
using *double-arith-series* [*of a d n*, *symmetric*] **by** *simp*

end

lemma *gauss-sum-nat*:
$\sum \{0..n\} = (n * \textit{Suc}\ n)\ \textit{div}\ 2$
using *gauss-sum* [*of n*, **where** *?'a* = *nat*] **by** *simp*

lemma *arith-series-nat*:
$(\sum i = 0..n.\ a + i * d) = \textit{Suc}\ n * (2 * a + n * d)\ \textit{div}\ 2$
using *arith-series* [*of a d n*] **by** *simp*

lemma *Sum-Icc-int*:
$\sum \{m..n\} = (n * (n + 1) - m * (m - 1))\ \textit{div}\ 2$
if $m \leq n$ **for** *m n* :: *int*
using *that* **proof** (*induct i* ≡ *nat* (*n* − *m*) *arbitrary*: *m n*)
case *0*
then have $m = n$
by *arith*
then show *?case*
by (*simp add*: *algebra-simps mult-2* [*symmetric*])
next
case (*Suc i*)
have *0*: $i = \textit{nat}((n-1) - m)$ $m \leq n-1$ **using** *Suc*(*2*,*3*) **by** *arith*+
have $\sum \{m..n\} = \sum \{m..1+(n-1)\}$ **by** *simp*
also have ... $= \sum \{m..n-1\} + n$ **using** ‹$m \leq n$›
by(*subst atLeastAtMostPlus1-int-conv*) *simp-all*
also have ... $= ((n-1)*(n-1+1) - m*(m-1))\ \textit{div}\ 2 + n$
by(*simp add*: *Suc*(*1*)[*OF 0*])
also have ... $= ((n-1)*(n-1+1) - m*(m-1) + 2*n)\ \textit{div}\ 2$ **by** *simp*
also have ... $= (n*(n+1) - m*(m-1))\ \textit{div}\ 2$
by (*simp add*: *algebra-simps mult-2-right*)
finally show *?case* **.**
qed

lemma *Sum-Icc-nat*:
$\sum\{m..n\} = (n * (n + 1) - m * (m - 1))$ *div 2* **for** *m n* :: *nat*
proof (*cases* $m \leq n$)
case *True*
then have $*$: $m * (m - 1) \leq n * (n + 1)$
by (*meson diff-le-self order-trans le-add1 mult-le-mono*)
have *int* $(\sum\{m..n\}) = (\sum\{int\ m..int\ n\})$
by (*simp add*: *sum.atLeast-int-atMost-int-shift*)
also have $\ldots$ $= (int\ n * (int\ n + 1) - int\ m * (int\ m - 1))$ *div 2*
using ‹$m \leq n$› **by** (*simp add*: *Sum-Icc-int*)
also have $\ldots$ $= int\ ((n * (n + 1) - m * (m - 1))$ *div 2*)
using *le-square* $*$ **by** (*simp add*: *algebra-simps of-nat-div of-nat-diff*)
finally show *?thesis*
by (*simp only*: *of-nat-eq-iff*)
next
case *False*
then show *?thesis*
by (*auto dest*: *less-imp-Suc-add simp add*: *not-le algebra-simps*)
qed

lemma *Sum-Ico-nat*:
$\sum\{m..<n\} = (n * (n - 1) - m * (m - 1))$ *div 2* **for** *m n* :: *nat*
by (*cases n*) (*simp-all add*: *atLeastLessThanSuc-atLeastAtMost Sum-Icc-nat*)

61.9.6 Division remainder

lemma *range-mod*:
fixes *n* :: *nat*
assumes $n > 0$
shows *range* $(\lambda m.\ m\ mod\ n) = \{0..<n\}$ (**is** *?A* = *?B*)
proof (*rule set-eqI*)
fix *m*
show $m \in ?A \longleftrightarrow m \in ?B$
proof
assume $m \in ?A$
with *assms* **show** $m \in ?B$
by *auto*
next
assume $m \in ?B$
moreover have $m\ mod\ n \in ?A$
by (*rule rangeI*)
ultimately show $m \in ?A$
by *simp*
qed
qed

61.10 Products indexed over intervals

syntax (*ASCII*)

-from-to-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder PROD*››*PROD* - = -..-./ -)› [*0*,*0*,*0*,*10*] *10*)
-from-upto-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder PROD*››*PROD* - = -..<-./ -)› [*0*,*0*,*0*,*10*] *10*)
-upt-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder PROD*››*PROD* -<-./ -)› [*0*,*0*,*10*] *10*)
-upto-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*notation*=‹*binder PROD*››*PROD* -<=-./ -)› [*0*,*0*,*10*] *10*)

syntax (*latex-prod* **output**)
-from-to-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(*3*$\prod_{-=-}$ -)› [*0*,*0*,*0*,*10*] *10*)
-from-upto-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(*3*$\prod_{-=-}^{<-}$ -)› [*0*,*0*,*0*,*10*] *10*)
-upt-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(*3*$\prod_{-<-}$ -)› [*0*,*0*,*10*] *10*)
-upto-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b*
(‹(*3*$\prod_{-\leq-}$ -)› [*0*,*0*,*10*] *10*)

syntax
-from-to-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3* *notation*=‹*binder* ∏››∏ - = -..-./ -)› [*0*,*0*,*0*,*10*] *10*)
-from-upto-prod :: *idt* ⇒ *'a* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3* *notation*=‹*binder* ∏››∏ - = -..<-./ -)› [*0*,*0*,*0*,*10*] *10*)
-upt-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3* *notation*=‹*binder* ∏››∏ -<-./ -)› [*0*,*0*,*10*] *10*)
-upto-prod :: *idt* ⇒ *'a* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3* *notation*=‹*binder* ∏››∏ -≤-./ -)› [*0*,*0*,*10*] *10*)

syntax-consts
-from-to-prod -from-upto-prod -upt-prod -upto-prod ⇌ *prod*

translations
∏*x*=*a*..*b*. *t* ⇌ *CONST prod* (λ*x*. *t*) {*a*..*b*}
∏*x*=*a*..<*b*. *t* ⇌ *CONST prod* (λ*x*. *t*) {*a*..<*b*}
∏*i*≤*n*. *t* ⇌ *CONST prod* (λ*i*. *t*) {..*n*}
∏*i*<*n*. *t* ⇌ *CONST prod* (λ*i*. *t*) {..<*n*}

lemma *prod-int-plus-eq*: *prod int* {*i*..*i*+*j*} = ∏{*int i*..*int* (*i*+*j*)}
by (*induct j*) (*auto simp add*: *atLeastAtMostSuc-conv atLeastAtMostPlus1-int-conv*)

lemma *prod-int-eq*: *prod int* {*i*..*j*} = ∏{*int i*..*int j*}
proof (*cases i* ≤ *j*)
case *True*
then show *?thesis*
by (*metis le-iff-add prod-int-plus-eq*)
next
case *False*
then show *?thesis*

by *auto*
qed

61.10.1 Telescoping products

lemma *prod-telescope*:
fixes f::*nat* ⇒ 'a::*field*
assumes $\bigwedge i.\ i{\le}n \Longrightarrow f\ (Suc\ i) \neq 0$
shows $(\prod i{\le}n.\ f\ i\ /\ f\ (Suc\ i)) = f\ 0\ /\ f\ (Suc\ n)$
using *assms* **by** (*induction n*) *auto*

lemma *prod-telescope''*:
fixes f::*nat* ⇒ 'a::*field*
assumes $m \le n$
assumes $\bigwedge i.\ i \in \{m..n\} \Longrightarrow f\ i \neq 0$
shows $(\prod i = Suc\ m..n.\ f\ i\ /\ f\ (i - 1)) = f\ n\ /\ f\ m$
by (*rule dec-induct*[*OF* ‹$m \le n$›]) (*auto simp add*: *assms*)

lemma *prod-lessThan-telescope*:
fixes f::*nat* ⇒ 'a::*field*
assumes $\bigwedge i.\ i{\le}n \Longrightarrow f\ i \neq 0$
shows $(\prod i{<}n.\ f\ (Suc\ i)\ /\ f\ i) = f\ n\ /\ f\ 0$
using *assms* **by** (*induction n*) *auto*

lemma *prod-lessThan-telescope'*:
fixes f::*nat* ⇒ 'a::*field*
assumes $\bigwedge i.\ i{\le}n \Longrightarrow f\ i \neq 0$
shows $(\prod i{<}n.\ f\ i\ /\ f\ (Suc\ i)) = f\ 0\ /\ f\ n$
using *assms* **by** (*induction n*) *auto*

61.11 Efficient folding over intervals

function *fold-atLeastAtMost-nat* **where**
[*simp del*]: *fold-atLeastAtMost-nat f a* (b::*nat*) *acc* =
(*if a* > *b then acc else fold-atLeastAtMost-nat f* (*a*+*1*) *b* (*f a acc*))
by *pat-completeness auto*
termination by (*relation measure* (λ(-,*a*,*b*,-). *Suc b* − *a*)) *auto*

lemma *fold-atLeastAtMost-nat*:
assumes *comp-fun-commute f*
shows *fold-atLeastAtMost-nat f a b acc* = *Finite-Set.fold f acc* {*a*..*b*}
using *assms*
proof (*induction f a b acc rule*: *fold-atLeastAtMost-nat.induct*, *goal-cases*)
case (*1 f a b acc*)
interpret *comp-fun-commute f* **by** *fact*
show *?case*
proof (*cases a* > *b*)
case *True*
thus *?thesis* **by** (*subst fold-atLeastAtMost-nat.simps*) *auto*
next

```
    case False
    with 1 show ?thesis
      by (subst fold-atLeastAtMost-nat.simps)
         (auto simp: atLeastAtMost-insertL[symmetric] fold-fun-left-comm)
  qed
qed

lemma sum-atLeastAtMost-code:
  sum f {a..b} = fold-atLeastAtMost-nat (λa acc. f a + acc) a b 0
proof −
  have comp-fun-commute (λa. (+) (f a))
    by unfold-locales (auto simp: o-def add-ac)
  thus ?thesis
    by (simp add: sum.eq-fold fold-atLeastAtMost-nat o-def)
qed

lemma prod-atLeastAtMost-code:
  prod f {a..b} = fold-atLeastAtMost-nat (λa acc. f a * acc) a b 1
proof −
  have comp-fun-commute (λa. (*) (f a))
    by unfold-locales (auto simp: o-def mult-ac)
  thus ?thesis
    by (simp add: prod.eq-fold fold-atLeastAtMost-nat o-def)
qed



lemma pairs-le-eq-Sigma: {(i, j). i + j ≤ m} = Sigma (atMost m) (λr. atMost
(m − r))
  for m :: nat
  by auto

lemma sum-up-index-split: (∑ k≤m + n. f k) = (∑ k≤m. f k) + (∑ k = Suc
m..m + n. f k)
  by (metis atLeast0AtMost Suc-eq-plus1 le0 sum.ub-add-nat)

lemma Sigma-interval-disjoint: (SIGMA i:A. {..v i}) ∩ (SIGMA i:A.{v i<..w})
= {}
  for w :: 'a::order
  by auto

lemma product-atMost-eq-Un: A × {..m} = (SIGMA i:A.{..m − i}) ∪ (SIGMA
i:A.{m − i<..m})
  for m :: nat
  by auto

lemma polynomial-product:
  fixes x :: 'a::idom
  assumes m: ⋀i. i > m ⟹ a i = 0
```

and n: $\bigwedge j.\ j > n \Longrightarrow b\ j = 0$
shows $(\sum i{\leq}m.\ (a\ i) * x \mathbin{\hat{}} i) * (\sum j{\leq}n.\ (b\ j) * x \mathbin{\hat{}} j) =$
$(\sum r{\leq}m + n.\ (\sum k{\leq}r.\ (a\ k) * (b\ (r - k))) * x \mathbin{\hat{}} r)$
proof −
have $\bigwedge i\ j.\ [\![m + n - i < j;\ a\ i \neq 0]\!] \Longrightarrow b\ j = 0$
by (*meson le-add-diff leI le-less-trans m n*)
then have §: $(\sum (i,j){\in}(SIGMA\ i{:}\{..m{+}n\}.\ \{m{+}n - i{<}..m{+}n\}).\ a\ i * x \mathbin{\hat{}} i * (b\ j * x \mathbin{\hat{}} j)) = 0$
by (*clarsimp simp add*: *sum-Un Sigma-interval-disjoint intro*!: *sum.neutral*)
have $(\sum i{\leq}m.\ (a\ i) * x \mathbin{\hat{}} i) * (\sum j{\leq}n.\ (b\ j) * x \mathbin{\hat{}} j) = (\sum i{\leq}m.\ \sum j{\leq}n.\ (a\ i * x \mathbin{\hat{}} i) * (b\ j * x \mathbin{\hat{}} j))$
by (*rule sum-product*)
also have ... $= (\sum i{\leq}m + n.\ \sum j{\leq}n + m.\ a\ i * x \mathbin{\hat{}} i * (b\ j * x \mathbin{\hat{}} j))$
using *assms* **by** (*auto simp*: *sum-up-index-split*)
also have ... $= (\sum r{\leq}m + n.\ \sum j{\leq}m + n - r.\ a\ r * x \mathbin{\hat{}} r * (b\ j * x \mathbin{\hat{}} j))$
by (*simp add*: *add-ac sum.Sigma product-atMost-eq-Un sum-Un Sigma-interval-disjoint* §)
also have ... $= (\sum (i,j){\in}\{(i,j).\ i{+}j \leq m{+}n\}.\ (a\ i * x \mathbin{\hat{}} i) * (b\ j * x \mathbin{\hat{}} j))$
by (*auto simp*: *pairs-le-eq-Sigma sum.Sigma*)
also have ... $= (\sum k{\leq}m + n.\ \sum i{\leq}k.\ a\ i * x \mathbin{\hat{}} i * (b\ (k - i) * x \mathbin{\hat{}} (k - i)))$
by (*rule sum.triangle-reindex-eq*)
also have ... $= (\sum r{\leq}m + n.\ (\sum k{\leq}r.\ (a\ k) * (b\ (r - k))) * x \mathbin{\hat{}} r)$
by (*auto simp*: *algebra-simps sum-distrib-left simp flip*: *power-add intro*!: *sum.cong*)
finally show *?thesis* **.**
qed

end

62 Decision Procedure for Presburger Arithmetic

theory *Presburger*
imports *Groebner-Basis Set-Interval*
begin

ML-file ‹*Tools/Qelim/qelim.ML*›
ML-file ‹*Tools/Qelim/cooper-procedure.ML*›

62.1 The $-\infty$ and $+\infty$ Properties

lemma *minf*:
$[\![\exists (z::'a{::}linorder).\forall x{<}z.\ P\ x = P'\ x;\ \exists z.\forall x{<}z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x{<}z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z::'a{::}linorder).\forall x{<}z.\ P\ x = P'\ x;\ \exists z.\forall x{<}z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x{<}z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z::'a{::}\{linorder\}).\forall x{<}z.(x = t) = False$
$\exists (z::'a{::}\{linorder\}).\forall x{<}z.(x \neq t) = True$
$\exists (z::'a{::}\{linorder\}).\forall x{<}z.(x < t) = True$
$\exists (z::'a{::}\{linorder\}).\forall x{<}z.(x \leq t) = True$
$\exists (z::'a{::}\{linorder\}).\forall x{<}z.(x > t) = False$

$\exists (z::'a::\{linorder\}).\forall x<z.(x \geq t) = False$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})<z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})<z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x<z.\ F = F$
proof *safe*
fix *z1 z2*
assume $\forall x<z1.\ P\ x = P'\ x$ **and** $\forall x<z2.\ Q\ x = Q'\ x$
then have $\forall x < min\ z1\ z2.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
by *simp*
then show $\exists z.\ \forall x<z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
by *blast*
next
fix *z1 z2*
assume $\forall x<z1.\ P\ x = P'\ x$ **and** $\forall x<z2.\ Q\ x = Q'\ x$
then have $\forall x < min\ z1\ z2.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
by *simp*
then show $\exists z.\ \forall x<z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
by *blast*
next
have $\forall x<t.\ x \leq t$
by *fastforce*
then show $\exists z.\ \forall x<z.\ (x \leq t) = True$
by *auto*
next
have $\forall x<t.\ \neg\ t < x$
by *fastforce*
then show $\exists z.\ \forall x<z.\ (t < x) = False$
by *auto*
next
have $\forall x<t.\ \neg\ t \leq x$
by *fastforce*
then show $\exists z.\ \forall x<z.\ (t \leq x) = False$
by *auto*
qed *auto*

lemma *pinf*:
$[\![\exists (z::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
$[\![\exists (z::'a::linorder).\forall x>z.\ P\ x = P'\ x;\ \exists z.\forall x>z.\ Q\ x = Q'\ x]\!]$
$\Longrightarrow \exists z.\forall x>z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
$\exists (z::'a::\{linorder\}).\forall x>z.(x = t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \neq t) = True$
$\exists (z::'a::\{linorder\}).\forall x>z.(x < t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \leq t) = False$
$\exists (z::'a::\{linorder\}).\forall x>z.(x > t) = True$
$\exists (z::'a::\{linorder\}).\forall x>z.(x \geq t) = True$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})>z.\ (d\ dvd\ x + s) = (d\ dvd\ x + s)$
$\exists z.\forall (x::'b::\{linorder,plus,Rings.dvd\})>z.\ (\neg\ d\ dvd\ x + s) = (\neg\ d\ dvd\ x + s)$
$\exists z.\forall x>z.\ F = F$

proof *safe*
fix *z1 z2*
assume $\forall x{>}z1.\ P\ x = P'\ x$ **and** $\forall x{>}z2.\ Q\ x = Q'\ x$
then have $\forall x > max\ z1\ z2.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
by *simp*
then show $\exists z.\ \forall x{>}z.\ (P\ x \wedge Q\ x) = (P'\ x \wedge Q'\ x)$
by *blast*
next
fix *z1 z2*
assume $\forall x{>}z1.\ P\ x = P'\ x$ **and** $\forall x{>}z2.\ Q\ x = Q'\ x$
then have $\forall x > max\ z1\ z2.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
by *simp*
then show $\exists z.\ \forall x{>}z.\ (P\ x \vee Q\ x) = (P'\ x \vee Q'\ x)$
by *blast*
next
have $\forall x{>}t.\ \neg\ x < t$
by *fastforce*
then show $\exists z.\ \forall x{>}z.\ x < t = False$
by *blast*
next
have $\forall x{>}t.\ \neg\ x \leq t$
by *fastforce*
then show $\exists z.\ \forall x{>}z.\ x \leq t = False$
by *blast*
next
have $\forall x{>}t.\ t \leq x$
by *fastforce*
then show $\exists z.\ \forall x{>}z.\ t \leq x = True$
by *blast*
qed *auto*

lemma *inf-period*:
$[\![\forall x\ k.\ P\ x = P\ (x - k{*}D); \forall x\ k.\ Q\ x = Q\ (x - k{*}D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \wedge Q\ x) = (P\ (x - k{*}D) \wedge Q\ (x - k{*}D))$
$[\![\forall x\ k.\ P\ x = P\ (x - k{*}D); \forall x\ k.\ Q\ x = Q\ (x - k{*}D)]\!]$
$\Longrightarrow \forall x\ k.\ (P\ x \vee Q\ x) = (P\ (x - k{*}D) \vee Q\ (x - k{*}D))$
$(d{::}'a{::}\{comm\text{-}ring, Rings.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (d\ dvd\ x + t) = (d\ dvd\ (x - k{*}D) + t)$
$(d{::}'a{::}\{comm\text{-}ring, Rings.dvd\})\ dvd\ D \Longrightarrow \forall x\ k.\ (\neg d\ dvd\ x + t) = (\neg d\ dvd\ (x - k{*}D) + t)$
$\forall x\ k.\ F = F$
apply (*auto elim*!: *dvdE simp add*: *algebra-simps*)
unfolding *mult.assoc* [*symmetric*] *distrib-right* [*symmetric*] *left-diff-distrib* [*symmetric*]
unfolding *dvd-def mult.commute* [*of d*]
by *auto*

62.2 The A and B sets

lemma *bset*:

$[\![\forall x.(\forall j \in \{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)$;
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x - D) \wedge Q\ (x - D))$
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow P\ x \longrightarrow P(x - D)$;
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow Q\ x \longrightarrow Q(x - D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (P\ x \vee Q\ x) \longrightarrow (P(x - D) \vee Q\ (x - D))$
$[\![D{>}0;\ t - 1{\in}\ B]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t))$
$[\![D{>}0\ ;\ t \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x > t) \longrightarrow (x - D > t))$
$[\![D{>}0\ ;\ t - 1 \in B]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \geq t) \longrightarrow (x - D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (\neg d\ dvd\ x{+}t) \longrightarrow (\neg\ d\ dvd\ (x - D) + t))$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow F \longrightarrow F$

proof (*blast*, *blast*)
assume *dp*: $D > 0$ **and** *tB*: $t - 1{\in}\ B$
show $(\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x = t) \longrightarrow (x - D = t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*1*],*erule ballE*[**where** *x*=*t* − *1*])
apply *algebra* **using** *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **and** *tB*: $t \in B$
show $(\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \neq t) \longrightarrow (x - D \neq t))$
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*D*],*erule ballE*[**where** *x*=*t*])
apply *algebra*
using *dp tB* **by** *simp-all*
next
assume *dp*: $D > 0$ **thus** $(\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x < t) \longrightarrow (x - D < t))$ **by** *arith*
next
assume *dp*: $D > 0$ **thus** $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j) \longrightarrow (x \leq t) \longrightarrow (x - D \leq t)$ **by** *arith*
next
assume *dp*: $D > 0$ **and** *tB*:$t \in B$
{fix *x* **assume** *nob*: $\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j$ **and** *g*: $x > t$ **and** *ng*: $\neg\ (x - D) > t$
hence $x - t \leq D$ **and** $1 \leq x - t$ **by** *simp+*
hence $\exists j \in \{1 \,..\, D\}.\ x - t = j$ **by** *auto*

hence $\exists j \in \{1 \,..\, D\}.\ x = t + j$ **by** (*simp add*: *algebra-simps*)
with *nob tB* **have** *False* **by** *simp*}
thus $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (x > t) \longrightarrow (x - D > t)$ **by** *blast*
next
assume *dp*: $D > 0$ **and** *tB*:$t - 1 \in B$
{**fix** x **assume** *nob*: $\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j$ **and** *g*: $x \geq t$ **and** *ng*: $\neg\ (x - D) \geq t$
hence $x - (t - 1) \leq D$ **and** $1 \leq x - (t - 1)$ **by** *simp+*
hence $\exists j \in \{1 \,..\, D\}.\ x - (t - 1) = j$ **by** *auto*
hence $\exists j \in \{1 \,..\, D\}.\ x = (t - 1) + j$ **by** (*simp add*: *algebra-simps*)
with *nob tB* **have** *False* **by** *simp*}
thus $\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (x \geq t) \longrightarrow (x - D \geq t)$ **by** *blast*
next
assume *d*: *d dvd D*
{**fix** x **assume** *H*: *d dvd* $x + t$ **with** *d* **have** *d dvd* $(x - D) + t$ **by** *algebra*}
thus $\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x - D) + t)$ **by** *simp*
next
assume *d*: *d dvd D*
{**fix** x **assume** *H*: $\neg(d\ dvd\ x + t)$ **with** *d* **have** $\neg\ d\ dvd\ (x - D) + t$
by (*clarsimp simp add*: *dvd-def*,*erule-tac x*= *ka* + *k* **in** *allE*,*simp add*: *algebra-simps*)}
thus $\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}B.\ x \neq b + j){\longrightarrow}\ (\neg d\ dvd\ x{+}t) \longrightarrow (\neg d\ dvd\ (x - D) + t)$ **by** *auto*
qed *blast*

lemma *aset*:
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j) \longrightarrow (P\ x \wedge Q\ x) \longrightarrow (P(x + D) \wedge Q\ (x + D))$
$[\![\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ P\ x \longrightarrow P(x + D)\ ;$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ Q\ x \longrightarrow Q(x + D)]\!] \Longrightarrow$
$\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (P\ x \vee Q\ x) \longrightarrow (P(x + D) \vee Q\ (x + D))$
$[\![D{>}0;\ t + 1{\in}\ A]\!] \Longrightarrow (\forall x.(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x = t) \longrightarrow (x + D = t))$
$[\![D{>}0\ ;\ t \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \neq t) \longrightarrow (x + D \neq t))$
$[\![D{>}0;\ t{\in}\ A]\!] \Longrightarrow (\forall (x{::}int).\ (\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x < t) \longrightarrow (x + D < t))$
$[\![D{>}0;\ t + 1 \in A]\!] \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \leq t) \longrightarrow (x + D \leq t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x > t) \longrightarrow (x + D > t))$
$D{>}0 \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (x \geq t) \longrightarrow (x + D \geq t))$
$d\ dvd\ D \Longrightarrow (\forall (x{::}int).(\forall j{\in}\{1 \,..\, D\}.\ \forall b{\in}A.\ x \neq b - j){\longrightarrow}\ (d\ dvd\ x{+}t) \longrightarrow (d\ dvd\ (x + D) + t))$

d dvd D $\Longrightarrow$($\forall$ (*x*::*int*).($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ ($\neg$*d dvd x*+*t*) $\longrightarrow$ ($\neg$ *d dvd* (*x* + *D*) + *t*))
$\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*) $\longrightarrow$ *F* $\longrightarrow$ *F*
proof (*blast*, *blast*)
assume *dp*: *D* > *0* **and** *tA*: *t* + *1* $\in$ *A*
show ($\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* = *t*) $\longrightarrow$ (*x* + *D* = *t*))
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*1*],*erule ballE*[**where** *x*=*t* + *1*])
using *dp tA* **by** *simp-all*
next
assume *dp*: *D* > *0* **and** *tA*: *t* $\in$ *A*
show ($\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* $\neq$ *t*) $\longrightarrow$ (*x* + *D* $\neq$ *t*))
apply (*rule allI*, *rule impI*,*erule ballE*[**where** *x*=*D*],*erule ballE*[**where** *x*=*t*])
using *dp tA* **by** *simp-all*
next
assume *dp*: *D* > *0* **thus** ($\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* > *t*) $\longrightarrow$ (*x* + *D* > *t*)) **by** *arith*
next
assume *dp*: *D* > *0* **thus** $\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* $\geq$ *t*) $\longrightarrow$ (*x* + *D* $\geq$ *t*) **by** *arith*
next
assume *dp*: *D* > *0* **and** *tA*:*t* $\in$ *A*
{fix *x* **assume** *nob*: $\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j* **and** *g*: *x* < *t* **and** *ng*: $\neg$ (*x* + *D*) < *t*
hence *t* − *x* $\leq$ *D* **and** *1* $\leq$ *t* − *x* **by** *simp*+
hence $\exists$ *j* $\in$ {*1 .. D*}. *t* − *x* = *j* **by** *auto*
hence $\exists$ *j* $\in$ {*1 .. D*}. *x* = *t* − *j* **by** (*auto simp add*: *algebra-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* < *t*) $\longrightarrow$ (*x* + *D* < *t*) **by** *blast*
next
assume *dp*: *D* > *0* **and** *tA*:*t* + *1*$\in$ *A*
{fix *x* **assume** *nob*: $\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j* **and** *g*: *x* $\leq$ *t* **and** *ng*: $\neg$ (*x* + *D*) $\leq$ *t*
hence (*t* + *1*) − *x* $\leq$ *D* **and** *1* $\leq$ (*t* + *1*) − *x* **by** (*simp-all add*: *algebra-simps*)
hence $\exists$ *j* $\in$ {*1 .. D*}. (*t* + *1*) − *x* = *j* **by** *auto*
hence $\exists$ *j* $\in$ {*1 .. D*}. *x* = (*t* + *1*) − *j* **by** (*auto simp add*: *algebra-simps*)
with *nob tA* **have** *False* **by** *simp***}**
thus $\forall$ *x*.($\forall$ *j*$\in${*1 .. D*}. $\forall$ *b*$\in$*A*. *x* $\neq$ *b* − *j*)$\longrightarrow$ (*x* $\leq$ *t*) $\longrightarrow$ (*x* + *D* $\leq$ *t*) **by** *blast*
next
assume *d*: *d dvd D*
have $\bigwedge$*x*. *d dvd x* + *t* $\Longrightarrow$ *d dvd x* + *D* + *t*
proof −
fix *x*
assume *H*: *d dvd x* + *t*
then obtain *ka* **where** *x* + *t* = *d* ∗ *ka*
unfolding *dvd-def* **by** *blast*
moreover from *d* **obtain** *k* **where** ∗:*D* = *d* ∗ *k*
unfolding *dvd-def* **by** *blast*
ultimately have *x* + *d* ∗ *k* + *t* = *d* ∗ (*ka* + *k*)

by (*simp add*: *algebra-simps*)
then show *d dvd* (*x* + *D*) + *t*
using ∗ **unfolding** *dvd-def* **by** *blast*
qed
thus ∀ (*x*::*int*).(∀ *j*∈{*1* .. *D*}. ∀ *b*∈*A*. *x* ≠ *b* − *j*)⟶ (*d dvd x*+*t*) ⟶ (*d dvd* (*x* + *D*) + *t*) **by** *simp*
next
assume *d*: *d dvd D*
{fix *x* **assume** *H*: ¬(*d dvd x* + *t*) **with** *d* **have** ¬*d dvd* (*x* + *D*) + *t*
using *dvd-add-left-iff*[*OF d*, *of x*+*t*] **by** (*simp add*: *algebra-simps*)**}**
thus ∀ (*x*::*int*).(∀ *j*∈{*1* .. *D*}. ∀ *b*∈*A*. *x* ≠ *b* − *j*)⟶ (¬*d dvd x*+*t*) ⟶ (¬*d dvd* (*x* + *D*) + *t*) **by** *auto*
qed *blast*

62.3 Cooper's Theorem $-\infty$ and $+\infty$ Version

62.3.1 First some trivial facts about periodic sets or predicates

lemma *periodic-finite-ex*:
assumes *dpos*: (*0*::*int*) < *d* **and** *modd*: ∀ *x k*. *P x* = *P*(*x* − *k*∗*d*)
shows (∃ *x*. *P x*) = (∃ *j* ∈ {*1*..*d*}. *P j*)
(**is** *?LHS* = *?RHS*)
proof
assume *?LHS*
then obtain *x* **where** *P*: *P x* **..**
have *x mod d* = *x* − (*x div d*)∗*d* **by**(*simp add*:*mult-div-mod-eq* [*symmetric*] *ac-simps eq-diff-eq*)
hence *Pmod*: *P x* = *P*(*x mod d*) **using** *modd* **by** *simp*
show *?RHS*
proof (*cases*)
assume *x mod d* = *0*
hence *P 0* **using** *P Pmod* **by** *simp*
moreover have *P 0* = *P*(*0* − (−*1*)∗*d*) **using** *modd* **by** *blast*
ultimately have *P d* **by** *simp*
moreover have *d* ∈ {*1*..*d*} **using** *dpos* **by** *simp*
ultimately show *?RHS* **..**
next
assume *not0*: *x mod d* ≠ *0*
have *P*(*x mod d*) **using** *dpos P Pmod* **by** *simp*
moreover have *x mod d* ∈ {*1*..*d*}
proof −
from *dpos* **have** *0* ≤ *x mod d* **by**(*rule pos-mod-sign*)
moreover from *dpos* **have** *x mod d* < *d* **by**(*rule pos-mod-bound*)
ultimately show *?thesis* **using** *not0* **by** *simp*
qed
ultimately show *?RHS* **..**
qed
qed *auto*

62.3.2 The $-\infty$ Version

lemma *decr-lemma*: $0 < (d::int) \Longrightarrow x - (|x - z| + 1) * d < z$
by (*induct rule*: *int-gr-induct*) (*simp-all add*: *int-distrib*)

lemma *incr-lemma*: $0 < (d::int) \Longrightarrow z < x + (|x - z| + 1) * d$
by (*induct rule*: *int-gr-induct*) (*simp-all add*: *int-distrib*)

lemma *decr-mult-lemma*:
assumes *dpos*: $(0::int) < d$ **and** *minus*: $\forall x.\ P\ x \longrightarrow P(x - d)$ **and** *knneg*: $0 <= k$
shows $\forall x.\ P\ x \longrightarrow P(x - k*d)$
using *knneg*
proof (*induct rule*:*int-ge-induct*)
case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix *x*
have $P\ x \longrightarrow P\ (x - i * d)$ **using** *step.hyps* **by** *blast*
also have $\ldots \longrightarrow P(x - (i + 1) * d)$ **using** *minus*[*THEN spec*, *of* $x - i * d$]
by (*simp add*: *algebra-simps*)
ultimately have $P\ x \longrightarrow P(x - (i + 1) * d)$ **by** *blast***}**
thus *?case* **..**
qed

lemma *minusinfinity*:
assumes *dpos*: $0 < d$ **and**
P1eqP1: $\forall x\ k.\ P1\ x = P1(x - k*d)$ **and** *ePeqP1*: $\exists z::int.\ \forall x.\ x < z \longrightarrow (P\ x = P1\ x)$
shows $(\exists x.\ P1\ x) \longrightarrow (\exists x.\ P\ x)$
proof
assume *eP1*: $\exists x.\ P1\ x$
then obtain *x* **where** *P1*: *P1 x* **..**
from *ePeqP1* **obtain** *z* **where** *P1eqP*: $\forall x.\ x < z \longrightarrow (P\ x = P1\ x)$ **..**
let *?w* = $x - (|x - z| + 1) * d$
from *dpos* **have** *w*: $?w < z$ **by**(*rule decr-lemma*)
have $P1\ x = P1\ ?w$ **using** *P1eqP1* **by** *blast*
also have $\ldots = P(?w)$ **using** *w P1eqP* **by** *blast*
finally have $P\ ?w$ **using** *P1* **by** *blast*
thus $\exists x.\ P\ x$ **..**
qed

lemma *cpmi*:
assumes *dp*: $0 < D$ **and** *p1*:$\exists z.\ \forall\ x< z.\ P\ x = P'\ x$
and *nb*:$\forall x.(\forall\ j\in \{1..D\}.\ \forall (b::int) \in B.\ x \neq b+j) \longrightarrow P\ (x) \longrightarrow P\ (x - D)$
and *pd*: $\forall\ x\ k.\ P'\ x = P'\ (x-k*D)$
shows $(\exists x.\ P\ x) = ((\exists j \in \{1..D\}\ .\ P'\ j) \vee (\exists j \in \{1..D\}.\ \exists\ b \in B.\ P\ (b+j)))$
(**is** $?L = (?R1 \vee ?R2)$)
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**

moreover
{assume *H*:*?R1* **hence** *?L* **using** *minusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**
moreover
{ fix *x*
assume *P*: *P x* **and** *H*: ¬ *?R2*
{fix *y* **assume** ¬ (∃ *j*∈{*1..D*}. ∃ *b*∈*B*. *P* (*b* + *j*)) **and** *P*: *P y*
hence ¬(∃ (*j*::*int*) ∈ {*1..D*}. ∃ (*b*::*int*) ∈ *B*. *y* = *b*+*j*) **by** *auto*
with *nb P* **have** *P* (*y* − *D*) **by** *auto* **}**
hence ∀ *x*. ¬(∃ (*j*::*int*) ∈ {*1..D*}. ∃ (*b*::*int*) ∈ *B*. *P*(*b*+*j*)) ⟶ *P* (*x*) ⟶ *P* (*x* − *D*) **by** *blast*
with *H P* **have** *th*: ∀ *x*. *P x* ⟶ *P* (*x* − *D*) **by** *auto*
from *p1* **obtain** *z* **where** *z*: ∀ *x*. *x* < *z* ⟶ (*P x* = *P′ x*) **by** *blast*
let *?y* = *x* − (|*x* − *z*| + *1*)∗*D*
have *zp*: *0* <= (|*x* − *z*| + *1*) **by** *arith*
from *dp* **have** *yz*: *?y* < *z* **using** *decr-lemma*[*OF dp*] **by** *simp*
from *z*[*rule-format*, *OF yz*] *decr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*] **have** *th2*: *P′ ?y* **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have *?R1* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

62.3.3 The +∞ Version

lemma *plusinfinity*:
assumes *dpos*: (*0*::*int*) < *d* **and**
P1eqP1: ∀ *x k*. *P′ x* = *P′*(*x* − *k*∗*d*) **and** *ePeqP1*: ∃ *z*. ∀ *x*>*z*. *P x* = *P′ x*
shows (∃ *x*. *P′ x*) ⟶ (∃ *x*. *P x*)
proof
assume *eP1*: ∃ *x*. *P′ x*
then obtain *x* **where** *P1*: *P′ x* **..**
from *ePeqP1* **obtain** *z* **where** *P1eqP*: ∀ *x*>*z*. *P x* = *P′ x* **..**
let *?w′* = *x* + (|*x* − *z*| + *1*) ∗ *d*
let *?w* = *x* − (− (|*x* − *z*| + *1*)) ∗ *d*
have *ww′*[*simp*]: *?w* = *?w′* **by** (*simp add*: *algebra-simps*)
from *dpos* **have** *w*: *?w* > *z* **by**(*simp only*: *ww′ incr-lemma*)
hence *P′ x* = *P′ ?w* **using** *P1eqP1* **by** *blast*
also have ... = *P*(*?w*) **using** *w P1eqP* **by** *blast*
finally have *P ?w* **using** *P1* **by** *blast*
thus ∃ *x*. *P x* **..**
qed

lemma *incr-mult-lemma*:
assumes *dpos*: (*0*::*int*) < *d* **and** *plus*: ∀ *x*::*int*. *P x* ⟶ *P*(*x* + *d*) **and** *knneg*: *0* <= *k*
shows ∀ *x*. *P x* ⟶ *P*(*x* + *k*∗*d*)
using *knneg*
proof (*induct rule*:*int-ge-induct*)

case *base* **thus** *?case* **by** *simp*
next
case (*step i*)
{fix *x*
have *P x* ⟶ *P* (*x* + *i* ∗ *d*) **using** *step.hyps* **by** *blast*
also have ... ⟶ *P*(*x* + (*i* + *1*) ∗ *d*) **using** *plus*[*THEN spec*, *of x* + *i* ∗ *d*]
by (*simp add*:*int-distrib ac-simps*)
ultimately have *P x* ⟶ *P*(*x* + (*i* + *1*) ∗ *d*) **by** *blast***}**
thus *?case* **..**
qed

lemma *cppi*:
assumes *dp*: *0* < *D* **and** *p1*:∃ *z*. ∀ *x*> *z*. *P x* = *P*′ *x*
and *nb*:∀ *x*.(∀ *j*∈ {*1*..*D*}. ∀ (*b*::*int*) ∈ *A*. *x* ≠ *b* − *j*) ⟶ *P* (*x*) ⟶ *P* (*x* + *D*)
and *pd*: ∀ *x k*. *P*′ *x*= *P*′ (*x*−*k*∗*D*)
shows (∃ *x*. *P x*) = ((∃ *j* ∈ {*1*..*D*} . *P*′ *j*) ∨ (∃ *j* ∈ {*1*..*D*}. ∃ *b*∈ *A*. *P* (*b* − *j*)))
(**is** *?L* = (*?R1* ∨ *?R2*))
proof−
{assume *?R2* **hence** *?L* **by** *blast***}**
moreover
{assume *H*:*?R1* **hence** *?L* **using** *plusinfinity*[*OF dp pd p1*] *periodic-finite-ex*[*OF dp pd*] **by** *simp***}**
moreover
{ fix *x*
assume *P*: *P x* **and** *H*: ¬ *?R2*
{fix *y* **assume** ¬ (∃ *j*∈{*1*..*D*}. ∃ *b*∈*A*. *P* (*b* − *j*)) **and** *P*: *P y*
hence ¬(∃ (*j*::*int*) ∈ {*1*..*D*}. ∃ (*b*::*int*) ∈ *A*. *y* = *b* − *j*) **by** *auto*
with *nb P* **have** *P* (*y* + *D*) **by** *auto* **}**
hence ∀ *x*. ¬(∃ (*j*::*int*) ∈ {*1*..*D*}. ∃ (*b*::*int*) ∈ *A*. *P*(*b*−*j*)) ⟶ *P* (*x*) ⟶ *P* (*x* + *D*) **by** *blast*
with *H P* **have** *th*: ∀ *x*. *P x* ⟶ *P* (*x* + *D*) **by** *auto*
from *p1* **obtain** *z* **where** *z*: ∀ *x*. *x* > *z* ⟶ (*P x* = *P*′ *x*) **by** *blast*
let *?y* = *x* + (|*x* − *z*| + *1*)∗*D*
have *zp*: *0* <= (|*x* − *z*| + *1*) **by** *arith*
from *dp* **have** *yz*: *?y* > *z* **using** *incr-lemma*[*OF dp*] **by** *simp*
from *z*[*rule-format*, *OF yz*] *incr-mult-lemma*[*OF dp th zp*, *rule-format*, *OF P*]
have *th2*: *P*′ *?y* **by** *auto*
with *periodic-finite-ex*[*OF dp pd*]
have *?R1* **by** *blast***}**
ultimately show *?thesis* **by** *blast*
qed

lemma *simp-from-to*: {*i*..*j*::*int*} = (*if j* < *i then* {} *else insert i* {*i*+*1*..*j*})
apply(*simp add*:*atLeastAtMost-def atLeast-def atMost-def*)
apply(*fastforce*)
done

theorem *unity-coeff-ex*: (∃ (*x*::′*a*::{*semiring-0*,*Rings.dvd*}). *P* (*l* ∗ *x*)) ≡ (∃ *x*. *l dvd* (*x* + *0*) ∧ *P x*)

unfolding *dvd-def* **by** (*rule eq-reflection*, *rule iffI*) *auto*

lemma *zdvd-mono*:
fixes *k m t* :: *int*
assumes $k \neq 0$
shows *m dvd t* $\equiv$ *k* $*$ *m dvd k* $*$ *t*
using *assms* **by** *simp*

lemma *uminus-dvd-conv*:
fixes *d t* :: *int*
shows *d dvd t* $\equiv -$ *d dvd t* **and** *d dvd t* $\equiv$ *d dvd* $-$ *t*
by *simp-all*

Theorems for transforming predicates on nat to predicates on *int*

lemma *zdiff-int-split*: *P* (*int* (*x* $-$ *y*)) =
($(y \leq x \longrightarrow P\ (int\ x - int\ y)) \wedge (x < y \longrightarrow P\ 0)$)
by (*cases* $y \leq x$) (*simp-all add*: *of-nat-diff*)

Specific instances of congruence rules, to prevent simplifier from looping.

theorem *imp-le-cong*:
$[\![x = x'; 0 \leq x' \Longrightarrow P = P']\!] \Longrightarrow (0 \leq (x::int) \longrightarrow P) = (0 \leq x' \longrightarrow P')$
by *simp*

theorem *conj-le-cong*:
$[\![x = x'; 0 \leq x' \Longrightarrow P = P']\!] \Longrightarrow (0 \leq (x::int) \wedge P) = (0 \leq x' \wedge P')$
by (*simp cong*: *conj-cong*)

ML-file ‹*Tools/Qelim/cooper.ML*›

method-setup *presburger* = ‹
let
fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ()
fun simple-keyword k = Scan.lift (Args.$$$ k) >> K ()
val addN = add
val delN = del
val elimN = elim
val any-keyword = keyword addN || keyword delN || simple-keyword elimN
val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm)
in
Scan.optional (simple-keyword elimN >> K false) true --
Scan.optional (keyword addN |-- thms) [] --
Scan.optional (keyword delN |-- thms) [] >>
(fn ((elim, add-ths), del-ths) => fn ctxt =>
SIMPLE-METHOD' (Cooper.tac elim add-ths del-ths ctxt))
end
› *Cooper's algorithm for Presburger arithmetic*

declare *mod-eq-0-iff-dvd* [*presburger*]

declare *mod-by-Suc-0* [*presburger*]
declare *mod-0* [*presburger*]
declare *mod-by-1* [*presburger*]
declare *mod-self* [*presburger*]
declare *div-by-0* [*presburger*]
declare *mod-by-0* [*presburger*]
declare *mod-div-trivial* [*presburger*]
declare *mult-div-mod-eq* [*presburger*]
declare *div-mult-mod-eq* [*presburger*]
declare *mod-mult-self1* [*presburger*]
declare *mod-mult-self2* [*presburger*]
declare *mod2-Suc-Suc* [*presburger*]
declare *not-mod-2-eq-0-eq-1* [*presburger*]
declare *nat-zero-less-power-iff* [*presburger*]

lemma [*presburger*, *algebra*]: $m \bmod 2 = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*, *algebra*]: $m \bmod 2 = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*, *algebra*]: $m \bmod (Suc\ (Suc\ 0)) = (1::nat) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*, *algebra*]: $m \bmod (Suc\ (Suc\ 0)) = Suc\ 0 \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*
lemma [*presburger*, *algebra*]: $m \bmod 2 = (1::int) \longleftrightarrow \neg\ 2\ dvd\ m$ **by** *presburger*

context *semiring-parity*
begin

declare *even-mult-iff* [*presburger*]

declare *even-power* [*presburger*]

lemma [*presburger*]:
$even\ (a + b) \longleftrightarrow even\ a \wedge even\ b \vee odd\ a \wedge odd\ b$
by *auto*

end

context *ring-parity*
begin

declare *even-minus* [*presburger*]

end

context *linordered-idom*
begin

declare *zero-le-power-eq* [*presburger*]

declare *zero-less-power-eq* [*presburger*]

declare *power-less-zero-eq* [*presburger*]

declare *power-le-zero-eq* [*presburger*]

end

declare *even-Suc* [*presburger*]

lemma [*presburger*]:
Suc n div Suc (Suc 0) = n div Suc (Suc 0) ⟷ even n
by *presburger*

declare *even-diff-nat* [*presburger*]

lemma [*presburger*]:
fixes *k* :: *int*
shows *(k + 1) div 2 = k div 2 ⟷ even k*
by *presburger*

lemma [*presburger*]:
fixes *k* :: *int*
shows *(k + 1) div 2 = k div 2 + 1 ⟷ odd k*
by *presburger*

lemma [*presburger*]:
even n ⟷ even (int n)
by *simp*

62.4 Nice facts about division by *4*::'*a*

lemma *even-even-mod-4-iff*:
even (n::*nat) ⟷ even (n mod 4)*
by *presburger*

lemma *odd-mod-4-div-2*:
n mod 4 = (3::*nat) ⟹ odd ((n − Suc 0) div 2)*
by *presburger*

lemma *even-mod-4-div-2*:
n mod 4 = Suc 0 ⟹ even ((n − Suc 0) div 2)
by *presburger*

end

theory *Try0-HOL*
imports *Try0 Presburger*
begin

ML ‹
```
signature TRY0-HOL =
sig
  val silence-methods : Proof.context -> Proof.context
end

structure Try0-HOL : TRY0-HOL = struct

(* Makes reconstructor tools as silent as possible. The set-visible calls suppresses
Unification
   bound exceeded warnings and the like. *)
fun silence-methods ctxt =
  ctxt
  |> Config.put Metis-Tactic.verbose false
  |> Simplifier-Trace.disable
  |> Context-Position.set-visible false
  |> Config.put Unify.unify-trace false
  |> Config.put Argo-Tactic.trace none

local

open Try0-Util

(* name * (run-if-auto-try * (all-goals * tags)) *)
val raw-named-methods =
  [(auto, (true, (true, full-attrs))),
   (blast, (true, (false, clas-attrs))),
   (metis, (true, (false, metis-attrs))),
   (argo, (true, (false, no-attrs))),
   (linarith, (true, (false, no-attrs))),
   (presburger, (true, (false, no-attrs))),
   (algebra, (true, (false, no-attrs))),
   (fast, (false, (false, clas-attrs))),
   (fastforce, (false, (false, full-attrs))),
   (force, (false, (false, full-attrs))),
   (meson, (false, (false, metis-attrs))),
   (satx, (false, (false, no-attrs))),
   (order, (true, (false, no-attrs)))]

in

val () = raw-named-methods
  |> List.app (fn (name, (run-if-auto-try, (all-goals, tags))) =>
    let
      val meth : Try0.proof-method =
        Try0-Util.apply-raw-named-method name all-goals tags silence-methods
    in
      Try0.register-proof-method name {run-if-auto-try = run-if-auto-try} meth
      handle Symtab.DUP - => ()
```

```
    end)

end

end
›

declare [[try0-schedule =
  satx metis
  order presburger linarith algebra argo
  simp auto blast fast fastforce force meson
]]

end
```

63 Bindings to Satisfiability Modulo Theories (SMT) solvers based on SMT-LIB 2

```
theory SMT
  imports Numeral-Simprocs
    ATP
  keywords
    smt-status :: diag
begin
```

63.1 A skolemization tactic and proof method

```
lemma ex-iff-push: (∃ y. P ⟷ Q y) ⟷ (P ⟶ (∃ y. Q y)) ∧ ((∀ y. Q y) ⟶
P)
  by metis

ML ‹
fun moura-tac ctxt =
  TRY o Atomize-Elim.atomize-elim-tac ctxt THEN'
  REPEAT o EqSubst.eqsubst-tac ctxt [0]
    @{thms choice-iff[symmetric] bchoice-iff[symmetric]} THEN'
  TRY o Simplifier.asm-full-simp-tac
   (Simplifier.clear-simpset ctxt |> Simplifier.add-simps @{thms all-simps ex-simps
ex-iff-push}) THEN-ALL-NEW
  Metis-Tactic.metis-tac (take 1 ATP-Proof-Reconstruct.partial-type-encs)
    ATP-Proof-Reconstruct.default-metis-lam-trans ctxt []
›

method-setup moura = ‹
  Scan.succeed (SIMPLE-METHOD' o moura-tac)
› solve skolemization goals, especially those arising from Z3 proofs

hide-fact (open) ex-iff-push
```

63.2 Triggers for quantifier instantiation

Some SMT solvers support patterns as a quantifier instantiation heuristics. Patterns may either be positive terms (tagged by "pat") triggering quantifier instantiations – when the solver finds a term matching a positive pattern, it instantiates the corresponding quantifier accordingly – or negative terms (tagged by "nopat") inhibiting quantifier instantiations. A list of patterns of the same kind is called a multipattern, and all patterns in a multipattern are considered conjunctively for quantifier instantiation. A list of multipatterns is called a trigger, and their multipatterns act disjunctively during quantifier instantiation. Each multipattern should mention at least all quantified variables of the preceding quantifier block.

typedecl *'a symb-list*

consts
Symb-Nil :: *'a symb-list*
Symb-Cons :: *'a* ⇒ *'a symb-list* ⇒ *'a symb-list*

typedecl *pattern*

consts
pat :: *'a* ⇒ *pattern*
nopat :: *'a* ⇒ *pattern*

definition *trigger* :: *pattern symb-list symb-list* ⇒ *bool* ⇒ *bool* **where**
trigger - P = *P*

63.3 Higher-order encoding

Application is made explicit for constants occurring with varying numbers of arguments. This is achieved by the introduction of the following constant.

definition *fun-app* :: *'a* ⇒ *'a* **where** *fun-app f* = *f*

Some solvers support a theory of arrays which can be used to encode higher-order functions. The following set of lemmas specifies the properties of such (extensional) arrays.

lemmas *array-rules* = *ext fun-upd-apply fun-upd-same fun-upd-other fun-upd-upd fun-app-def*

63.4 Normalization

lemma *case-bool-if*[*abs-def*]: *case-bool x y P* = (*if P then x else y*)
by *simp*

lemmas *Ex1-def-raw* = *Ex1-def*[*abs-def*]
lemmas *Ball-def-raw* = *Ball-def*[*abs-def*]
lemmas *Bex-def-raw* = *Bex-def*[*abs-def*]

lemmas *abs-if-raw* = *abs-if*[*abs-def*]
lemmas *min-def-raw* = *min-def*[*abs-def*]
lemmas *max-def-raw* = *max-def*[*abs-def*]

definition *pow-2* **where** *pow-2 y* = *power* (*2*::*int*) (*nat y*)

lemma *pow-2-raw''*: *power* (*2*::*nat*) ≡ (λ*b*. *nat* (*pow-2* (*int b*)))
unfolding *pow-2-def*
apply (*rule eq-reflection*)
apply *standard*+
using *nat-eq-iff2* **by** *force*

lemma *nat-zero-as-int*:
0 = *nat 0*
by *simp*

lemma *nat-one-as-int*:
1 = *nat 1*
by *simp*

lemma *nat-numeral-as-int*: *numeral* = (λ*i*. *nat* (*numeral i*)) **by** *simp*
lemma *nat-less-as-int*: (<) = (λ*a b*. *int a* < *int b*) **by** *simp*
lemma *nat-leq-as-int*: (≤) = (λ*a b*. *int a* ≤ *int b*) **by** *simp*
lemma *Suc-as-int*: *Suc* = (λ*a*. *nat* (*int a* + *1*)) **by** (*rule ext*) *simp*
lemma *nat-plus-as-int*: (+) = (λ*a b*. *nat* (*int a* + *int b*)) **by** (*rule ext*)+ *simp*
lemma *nat-minus-as-int*: (−) = (λ*a b*. *nat* (*int a* − *int b*)) **by** (*rule ext*)+ *simp*
lemma *nat-times-as-int*: (∗) = (λ*a b*. *nat* (*int a* ∗ *int b*)) **by** (*simp add*: *nat-mult-distrib*)
lemma *nat-div-as-int*: (*div*) = (λ*a b*. *nat* (*int a div int b*)) **by** (*simp add*: *nat-div-distrib*)
lemma *nat-mod-as-int*: (*mod*) = (λ*a b*. *nat* (*int a mod int b*)) **by** (*simp add*: *nat-mod-distrib*)

lemma *int-Suc*: *int* (*Suc n*) = *int n* + *1* **by** *simp*
lemma *int-plus*: *int* (*n* + *m*) = *int n* + *int m* **by** (*rule of-nat-add*)
lemma *int-minus*: *int* (*n* − *m*) = *int* (*nat* (*int n* − *int m*)) **by** *auto*

lemma *nat-int-comparison*:
fixes *a b* :: *nat*
shows (*a* = *b*) = (*int a* = *int b*)
and (*a* < *b*) = (*int a* < *int b*)
and (*a* ≤ *b*) = (*int a* ≤ *int b*)
by *simp-all*

named-theorems *int-ops* ‹*Used when embedding natural numbers into integers*›
named-theorems *nat-embedding-ops* ‹*Used when embedding natural numbers into integers*›

lemma [*int-ops*]:
fixes *a b* :: *nat*
shows *int 0* = *0*

and *int 1 = 1*
and *int (numeral n) = numeral n*
and *int (Suc a) = int a + 1*
and *int (a + b) = int a + int b*
and *int (a − b) = (if int a < int b then 0 else int a − int b)*
and *int (a * b) = int a * int b*
and *int (a div b) = int a div int b*
and *int (a mod b) = int a mod int b*
by (*auto intro*: *zdiv-int zmod-int*)

lemma [*int-ops*]:
int (nat (pow-2 x)) = pow-2 x
unfolding *pow-2-def*
by *simp*

lemma *int-if*:
fixes *a b* :: *nat*
shows *int (if P then a else b) = (if P then int a else int b)*
by *simp*

named-theorems *pow-2-word* ‹*power on words should be translated natively*›

63.5 Integer division and modulo for Z3

The following Z3-inspired definitions are overspecified for the case where $l = 0$. This Schönheitsfehler is corrected in the *div-as-z3div* and *mod-as-z3mod* theorems.

definition *z3div* :: *int* ⇒ *int* ⇒ *int* **where**
z3div k l = (if l ≥ 0 then k div l else − (k div − l))

definition *z3mod* :: *int* ⇒ *int* ⇒ *int* **where**
z3mod k l = k mod (if l ≥ 0 then l else − l)

lemma *div-as-z3div*:
∀ *k l. k div l = (if l = 0 then 0 else if l > 0 then z3div k l else z3div (− k) (− l))*
by (*simp add*: *z3div-def*)

lemma *mod-as-z3mod*:
∀ *k l. k mod l = (if l = 0 then k else if l > 0 then z3mod k l else − z3mod (− k) (− l))*
by (*simp add*: *z3mod-def*)

63.6 Extra theorems for Alethe reconstruction

lemma *alethe-sko-forall*[*no-atp*]: ‹(∀ *x. P x*) ⟷ *P* (*SOME x.* ¬*P x*)›
using *someI*[*of* ‹λ*x.* ¬*P x*›]
by *auto*

lemma *alethe-sko-forall'*[*no-atp*]: ‹*P* (*SOME x*. ¬*P x*) = *A* ⟹ (∀ *x*. *P x*) = *A*›
by (*subst alethe-sko-forall*)

lemma *alethe-sko-forall''*: ‹*B* = *A* ⟹ (*SOME x*. *P x*) = *A* ≡ (*SOME x*. *P x*) = *B*›
by *auto*

lemma *alethe-sko-forall-indirect*[*no-atp*]: ‹*x* = (*SOME x*. ¬*P x*) ⟹ (∀ *x*. *P x*) ⟷ *P x*›
using *someI*[*of* ‹λ*x*. ¬*P x*›]
by *auto*

lemma *alethe-sko-forall-indirect2*[*no-atp*]:
‹*x* = (*SOME x*. ¬*P x*) ⟹ (⋀*x* :: 'a. (*P x* = *P' x*)) ⟹ (∀ *x*. *P' x*) ⟷ *P x*›
using *someI*[*of* ‹λ*x*. ¬*P x*›]
by *auto*

lemma *alethe-sko-ex*[*no-atp*]: ‹(∃ *x*. *P x*) ⟷ *P* (*SOME x*. *P x*)›
using *someI*[*of* ‹λ*x*. *P x*›]
by *auto*

lemma *alethe-sko-ex'*[*no-atp*]: ‹*P* (*SOME x*. *P x*) = *A* ⟹ (∃ *x*. *P x*) = *A*›
by (*subst alethe-sko-ex*)

lemma *alethe-sko-ex-indirect*[*no-atp*]: ‹*x* = (*SOME x*. *P x*) ⟹ (∃ *x*. *P x*) ⟷ *P x*›
using *someI*[*of* ‹λ*x*. *P x*›]
by *auto*

lemma *alethe-sko-ex-indirect2*[*no-atp*]: ‹*x* = (*SOME x*. *P x*) ⟹ (⋀*x*. *P x* = *P' x*) ⟹ (∃ *x*. *P' x*) ⟷ *P x*›
using *someI*[*of* ‹λ*x*. *P x*›]
by *auto*

lemma *alethe-Pure-trans*[*no-atp*]:
‹*P* ≡ *Q* ⟹ *Q* ⟹ *P*›
by *auto*

lemma *alethe-if-cong*[*no-atp*]:
assumes ‹*b* ≡ *c*›
and ‹*c* ⟹ *x* ≡ *u*›
and ‹¬ *c* ⟹ *y* ≡ *v*›
shows ‹(*if b then x else y*) ≡ (*if c then u else v*)›
using *assms if-cong*[*of b c x u*] **by** *auto*

lemma *alethe-if-weak-cong'*[*no-atp*]:
‹*b* ≡ *c* ⟹ (*if b then x else y*) ≡ (*if c then x else y*)›
by *auto*

lemma *alethe-or-neg*[*no-atp*]:
  ‹$(A \Longrightarrow B) \Longrightarrow B \vee \neg A$›
  by *auto*

lemma *alethe-implies-pos*[*no-atp*]: ‹$\neg(A \longrightarrow B) \vee \neg A \vee B$›
  by *auto*

lemma *alethe-subst-bool*[*no-atp*]: ‹$P \Longrightarrow f\ True \Longrightarrow f\ P$›
  by *auto*

lemma *alethe-and-pos*[*no-atp*]:
  ‹$(a \Longrightarrow \neg(b \wedge c) \vee A) \Longrightarrow \neg(a \wedge b \wedge c) \vee A$›
  ‹$(a \Longrightarrow b \Longrightarrow A) \Longrightarrow \neg(a \wedge b) \vee A$›
  by *blast+*

lemma *alethe-and-pos0*[*no-atp*]:
  ‹$(\neg(b \wedge c) \vee A) \Longrightarrow \neg(a \wedge b \wedge c) \vee A$›
  ‹$(\neg b \vee A) \Longrightarrow \neg(a \wedge b) \vee A$›
  ‹$A \Longrightarrow \neg a \vee A$›
  by *blast+*

lemma *alethe-farkas*[*no-atp*]:
  ‹$(a \Longrightarrow A) \Longrightarrow \neg a \vee A$›
  ‹$(\neg a \Longrightarrow A) \Longrightarrow a \vee A$›
  by *blast+*

lemma *alethe-or-pos*[*no-atp*]:
  ‹$A \wedge A' \Longrightarrow (c \wedge A) \vee (\neg c \wedge A')$›
  ‹$A \wedge A' \Longrightarrow (\neg c \wedge A) \vee (c \wedge A')$›
  by *blast+*

lemma *alethe-distinct-elim-two-clauses*[*no-atp*]:
  $((x::bool) \neq y \wedge x \neq z \wedge y \neq z) = False$
  apply (*cases x*)
   apply (*cases y*)
  by *simp-all*

lemma *alethe-distinct-elim-0*[*no-atp*]:
  ‹$(x \Longrightarrow \neg y \Longrightarrow \neg z \Longrightarrow ((x \neq a) \wedge A) = False)$
$\Longrightarrow (\neg x \Longrightarrow y \Longrightarrow z \Longrightarrow ((x \neq a) \wedge A) = False)$
$\Longrightarrow (((x \neq y) \wedge (x \neq z) \wedge (x \neq a) \wedge A) = False)$›
  by (*cases x*; *cases y*; *cases z*; *simp*)

lemma *alethe-distinct-elim-1*[*no-atp*]:
  ‹$((x \neq b) \wedge A) = False \Longrightarrow ((x \neq a) \wedge (x \neq b) \wedge A) = False$›
  ‹$A = False \Longrightarrow ((x \neq a) \wedge A) = False$›
  ‹$(x = a) \Longrightarrow ((x \neq a) \wedge A) = False$›
  by *blast+*

lemma *alethe-distinct-elim-2*[*no-atp*]:
‹¬ $y \Longrightarrow \neg\ z \Longrightarrow y = z$›
‹$y \Longrightarrow z \Longrightarrow y = z$›
by *blast+*

lemma *alethe-shuffle-and1*[*no-atp*]:
‹ $A = B \Longrightarrow (a \wedge A) = (a \wedge B)$›
by *blast*

lemma *alethe-shuffle-and2*[*no-atp*]:
‹$(\neg a \longrightarrow \neg B) \Longrightarrow (a \Longrightarrow A = (b \wedge B)) \Longrightarrow (a \wedge A) = (b \wedge B)$›
‹$(\neg a \longrightarrow \neg(b \wedge B)) \Longrightarrow (a \Longrightarrow (b \wedge B)) \Longrightarrow a = (b \wedge B)$›
by (*cases a*) *auto*

lemma *alethe-shuffle-and3*[*no-atp*]:
‹$b = A \Longrightarrow (a \Longrightarrow b = (a \wedge A))$›
apply (*cases a*)
by *simp-all*

lemma *alethe-shuffle-and4*[*no-atp*]:
‹$A \Longrightarrow (a = (a \wedge A))$›
apply (*cases a*)
by *simp-all*

lemma *alethe-shuffle-or-split*[*no-atp*]:
$(a \longrightarrow (b \vee B)) \Longrightarrow (\neg a \Longrightarrow A = (b \vee B)) \Longrightarrow (a \vee A) = (b \vee B)$
$(a \longrightarrow (b \vee B)) \Longrightarrow (\neg a \Longrightarrow \neg(b \vee B)) \Longrightarrow a = (b \vee B)$
by *auto*

lemma *alethe-shuffle-or-resolve*[*no-atp*]:
$a \longrightarrow (a \vee A)$
$a \longrightarrow A \Longrightarrow a \longrightarrow (b \vee A)$
by *auto*

lemma *alethe-shuffle-or1a*[*no-atp*]: $(A = B) \Longrightarrow (a \vee A) = (a \vee B)$ **by** *auto*
lemma *alethe-shuffle-or1b*[*no-atp*]: $\neg B \Longrightarrow a = (a \vee B)$ **by** *auto*

lemma *alethe-shuffle-or4a*[*no-atp*]: $a \longrightarrow (a \vee B)$ **by** *auto*
lemma *alethe-shuffle-or4b*[*no-atp*]: $(a \longrightarrow B) \Longrightarrow (a \longrightarrow (b \vee B))$ **by** *auto*

lemma *alethe-shuffle-or2b*[*no-atp*]: $(b \Longrightarrow A) \Longrightarrow (\neg b \Longrightarrow A = B) \Longrightarrow A = (b \vee B)$
by *auto*

lemma *alethe-shuffle-or3*[*no-atp*]: $(a \Longrightarrow A) \Longrightarrow (\neg a \Longrightarrow A) \Longrightarrow (a \vee A)$
by *auto*

lemma *alethe-shuffle-or5*[*no-atp*]: $(a \Longrightarrow A) \Longrightarrow (\neg a \Longrightarrow \neg A) \Longrightarrow (a = A)$
by *auto*

lemma *alethe-shuffle-or6*[*no-atp*]: $\neg A \Longrightarrow (\neg a \Longrightarrow (a = A))$
by *auto*

lemma *alethe-la-generic*[*no-atp*]:
$\langle (a{::}int) \leq x \vee a = x \vee a \geq x \rangle$
by *linarith*

lemma *alethe-bfun-elim*[*no-atp*]:
$\langle (if\ b\ then\ P\ True\ else\ P\ False) = P\ b \rangle$
$\langle (\forall\, b.\ P'\ b) = (P'\ False \wedge P'\ True) \rangle$
$\langle (\exists\, b.\ P'\ b) = (P'\ False \vee P'\ True) \rangle$
by (*cases b*) (*auto simp*: *all-bool-eq ex-bool-eq*)

lemma *alethe-eq-true-simplify*[*no-atp*]:
$\langle (P = True) \equiv P \rangle$
by *auto*

lemma *alethe-and-neg*[*no-atp*]:
$\langle (a \Longrightarrow \neg b \vee A) \Longrightarrow \neg(a \wedge b) \vee A \rangle$
$\langle (a \Longrightarrow A) \Longrightarrow \neg a \vee A \rangle$
$\langle (\neg a \Longrightarrow A) \Longrightarrow a \vee A \rangle$
by *blast+*

lemma *alethe-forall-inst*[*no-atp*]:
$\langle A \longleftrightarrow B \Longrightarrow \neg A \vee B \rangle$
$\langle \neg A \longleftrightarrow B \Longrightarrow A \vee B \rangle$
$\langle A \longleftrightarrow B \Longrightarrow \neg B \vee A \rangle$
$\langle A \longleftrightarrow \neg B \Longrightarrow B \vee A \rangle$
$\langle A \longrightarrow B \Longrightarrow \neg A \vee B \rangle$
$\langle \neg A \longrightarrow B \Longrightarrow A \vee B \rangle$
by *blast+*

lemma *alethe-eq-transitive*[*no-atp*]:
$\langle A = B \Longrightarrow B = C \Longrightarrow A = C \rangle$
$\langle A = B \Longrightarrow C = B \Longrightarrow A = C \rangle$
$\langle B = A \Longrightarrow B = C \Longrightarrow A = C \rangle$
$\langle B = A \Longrightarrow C = B \Longrightarrow A = C \rangle$
by *auto*

lemma *alethe-bool-simplify*[*no-atp*]:
$\langle \neg(P \longrightarrow Q) \longleftrightarrow P \wedge \neg Q \rangle$
$\langle \neg(P \vee Q) \longleftrightarrow \neg P \wedge \neg Q \rangle$
$\langle \neg(P \wedge Q) \longleftrightarrow \neg P \vee \neg Q \rangle$
$\langle (P \longrightarrow (Q \longrightarrow R)) \longleftrightarrow ((P \wedge Q) \longrightarrow R) \rangle$
$\langle ((P \longrightarrow Q) \longrightarrow Q) \longleftrightarrow P \vee Q \rangle$
$\langle (Q \longleftrightarrow (P \vee Q)) \longleftrightarrow (P \longrightarrow Q) \rangle$ — This rule was inverted
$\langle P \wedge (P \longrightarrow Q) \longleftrightarrow P \wedge Q \rangle$
$\langle (P \longrightarrow Q) \wedge P \longleftrightarrow P \wedge Q \rangle$

unfolding *not-imp imp-conjL*
by *auto*

lemma *alethe-connective-def*[*no-atp*]:
‹$(A \neq B) = ((\neg A \wedge B) \vee (A \wedge \neg B))$› — xor case
‹$(A = B) = ((A \longrightarrow B) \wedge (B \longrightarrow A))$›
‹$(If\ A\ B\ C) = ((A \longrightarrow B) \wedge (\neg A \longrightarrow C))$›
apply (*case-tac* [!] *A*)
by *simp-all*

lemma *alethe-connective-def-forall*[*no-atp*]:
assumes $\bigwedge x::'a.\ (P\ x = (Q\ x))$
shows $(\forall x.\ P\ x) = (\forall x.\ Q\ x\ )$
using *assms*
unfolding *All-def*
by (*iprover intro*: *ext eqTrueI assms*)

lemma *alethe-connective-def-forall2*[*no-atp*]:
assumes $(P = Q)$
shows $(\forall x.\ P\ x) = (\forall x.\ Q\ x)$
unfolding *assms* **..**

lemma *alethe-ite-simplify*[*no-atp*]:
‹$(If\ True\ B\ C) = B$›
‹$(If\ False\ B\ C) = C$›
‹$(If\ A'\ B\ B) = B$›
‹$(If\ (\neg A')\ B\ C) = (If\ A'\ C\ B)$›
‹$(If\ c\ (If\ c\ A\ B)\ C) = (If\ c\ A\ C)$›
‹$(If\ c\ C\ (If\ c\ A\ B)) = (If\ c\ C\ B)$›
‹$(If\ A'\ True\ False) = A'$›
‹$(If\ A'\ False\ True) \longleftrightarrow \neg A'$›
‹$(If\ A'\ True\ B') \longleftrightarrow A' \vee B'$›
‹$(If\ A'\ B'\ False) \longleftrightarrow A' \wedge B'$›
‹$(If\ A'\ False\ B') \longleftrightarrow \neg A' \wedge B'$›
‹$(If\ A'\ B'\ True) \longleftrightarrow \neg A' \vee B'$›
‹$x \wedge True \longleftrightarrow x$›
‹$x \vee False \longleftrightarrow x$›
‹$x \vee True \longleftrightarrow True$›
‹$x \wedge False \longleftrightarrow False$›
‹$True \wedge x \longleftrightarrow x$›
‹$False \vee x \longleftrightarrow x$›
‹$True \vee x \longleftrightarrow True$›
‹$False \wedge x \longleftrightarrow False$›
for $B\ C :: 'a$ **and** $A'\ B'\ C' :: bool$
by *auto*

lemma *alethe-and-simplify1*[*no-atp*]:

‹*True* ∧ *b* ⟷ *b*› ‹*b* ∧ *True* ⟷ *b*›
‹*False* ∧ *b* ⟷ *False*› ‹*b* ∧ *False* ⟷ *False*›
‹(*c* ∧ ¬*c*) ⟷ *False*› ‹(¬*c* ∧ *c*) ⟷ *False*›
‹¬¬*a* = *a*›
by *auto*

lemmas *alethe-and-simplify* [*no-atp*] = *conj-ac de-Morgan-conj disj-not1*

lemma *alethe-or-simplify-1*[*no-atp*]:
‹*False* ∨ *b* ⟷ *b*› ‹*b* ∨ *False* ⟷ *b*›
‹*b* ∨ ¬*b*›
‹¬*b* ∨ *b*›
by *auto*

lemmas *alethe-or-simplify* [*no-atp*] = *disj-ac*

lemma *alethe-not-simplify* [*no-atp*]:
‹¬ ¬*b* ⟷ *b*› ‹¬*True* ⟷ *False*› ‹¬*False* ⟷ *True*›
by *auto*

lemma *alethe-implies-simplify*[*no-atp*]:
‹(¬*a* ⟶ ¬*b*) ⟷ (*b* ⟶ *a*)›
‹(*False* ⟶ *a*) ⟷ *True*›
‹(*a* ⟶ *True*) ⟷ *True*›
‹(*True* ⟶ *a*) ⟷ *a*›
‹(*a* ⟶ *False*) ⟷ ¬*a*›
‹(*a* ⟶ *a*) ⟷ *True*›
‹(¬*a* ⟶ *a*) ⟷ *a*›
‹(*a* ⟶ ¬*a*) ⟷ ¬*a*›
‹((*a* ⟶ *b*) ⟶ *b*) ⟷ *a* ∨ *b*›
‹(¬*True*) ⟷ *False*›
‹(¬*False*) ⟷ *True*›
by *auto*

lemma *alethe-equiv-simplify*[*no-atp*]:
‹((¬*a*) = (¬*b*)) ⟷ (*a* = *b*)›
‹(*a* = *a*) ⟷ *True*›
‹(*a* = (¬*a*)) ⟷ *False*›
‹((¬*a*) = *a*) ⟷ *False*›
‹(*True* = *a*) ⟷ *a*›
‹(*a* = *True*) ⟷ *a*›
‹(*False* = *a*) ⟷ ¬*a*›
‹(*a* = *False*) ⟷ ¬*a*›
‹¬¬*a* ⟷ *a*›
‹(¬ *False*) = *True*›
for *a b* :: *bool*
by *auto*

lemmas *alethe-eq-simplify*[*no-atp*] =

eq-refl num.simps neg-equal-zero equal-neg-zero neg-equal-iff-equal Num.rel-simps

lemma *alethe-minus-simplify*[*no-atp*]:
‹(*a* :: *'a* :: *cancel-comm-monoid-add*) − *a* = *0*›
‹(*a* :: *'a* :: *cancel-comm-monoid-add*) − *0* = *a*›
‹*0* − (*b* :: *'b* :: {*group-add*}) = −*b*›
‹− (− (*b* :: *'b* :: *group-add*)) = *b*›
by *auto*

lemma *alethe-sum-simplify*[*no-atp*]:
‹(*a* :: *'a* :: *cancel-comm-monoid-add*) + *0* = *a*›
by *auto*

lemmas *alethe-prod-simplify*[*no-atp*] =
mult-1
mult-1-right

lemmas *alethe-div-simplify*[*no-atp*] =
divide-self div-minus-minus div-by-1
divide-numeral-1 one-plus-numeral
divmod-steps less-irrefl divmod-trivial divmod-cancel
numeral-div-numeral prod.case mult.right-neutral
divmod-step-def euclidean-size-int-def comp-def of-bool-eq
nat-numeral fst-conv snd-conv if-False if-True
minus-numeral-div-numeral Parity.adjust-div-eq
order-refl
num.simps numerals eq-neg-numeral-simps eq-numeral-simps
old.nat.distinct
mult-numeral-left-semiring-numeral le-numeral-Suc pred-numeral-simps Suc-le-mono
le-zero-eq
le-num-simps less-num-simps euclidean-size-numeral le-numeral-simps numeral-plus-one
mult.left-neutral numeral-div-minus-numeral
nat-1 nat-0

lemma *alethe-comp-simplify1*[*no-atp*]:
‹(*a* :: *'a* ::*order*) < *a* ⟷ *False*›
‹*a* ≤ *a*›
‹¬(*b'* ≤ *a'*) ⟷ (*a'* :: *'b* :: *linorder*) < *b'*›
by *auto*

lemmas *alethe-comp-simplify*[*no-atp*] =
alethe-comp-simplify1
zero-less-one
zero-le-one
rel-simps

lemma *alethe-la-disequality*[*no-atp*]:
‹(*a* :: *'a* ::*linorder*) = *b* ∨ ¬*a* ≤ *b* ∨ ¬*b* ≤ *a*›
by *auto*

lemma *alethe-la-mult-pos-less*[*no-atp*]:
‹$(0 :: 'a :: linordered\text{-}idom) < m \wedge (a < b) \longrightarrow m*a < m*b$›
by *simp*

lemma *alethe-la-mult-pos*[*no-atp*]:
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge (a < b) \longrightarrow m*a < m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge (a \leq b) \longrightarrow m*a \leq m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge (a = b) \longrightarrow m*a = m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge \neg(a = b) \longrightarrow \neg(m*a = m*b)$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge (a \geq b) \longrightarrow m*a \geq m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow c < m \wedge (a > b) \longrightarrow m*a > m*b$›
by *simp-all*

lemma *alethe-la-mult-neg*[*no-atp*]:
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge (a < b) \longrightarrow m*a > m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge (a \leq b) \longrightarrow m*a \geq m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge (a = b) \longrightarrow m*a = m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge \neg(a = b) \longrightarrow \neg(m*a = m*b)$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge (a > b) \longrightarrow m*a < m*b$›
‹$c = (0 :: 'a :: linordered\text{-}idom) \Longrightarrow m < c \wedge (a \geq b) \longrightarrow m*a \leq m*b$›
by *simp-all*

context
begin

For the reconstruction, we need to keep the order of the arguments.

named-theorems *smt-arith-multiplication* ‹*Theorems to reconstruct arithmetic theorems.*›

named-theorems *smt-arith-combine* ‹*Theorems to reconstruct arithmetic theorems.*›

named-theorems *smt-arith-simplify* ‹*Theorems to combine theorems in the LA procedure*›

named-theorems *smt-rewrite-lemma-simplify* ‹*Theorems to simplify instantiate lemma during rewrite reconstruction*›

named-theorems *arith-simp-cvc5* ‹*Might be temp and integrated into smt-arith-simplify*›
named-theorems *arith-mult-poly-norm-cvc5* ‹*Extra rules for reconstruction of poly−norm.*›

lemmas [*arith-mult-poly-norm-cvc5*] =
Groups.monoid-mult-class.mult-1-right Nat.mult-Suc-right Nat.mult-0-right Nat.add-Suc-right Groups.monoid-add-class.add.right-neutral Num.numeral-2-eq-2 Nat.One-nat-def Num.numeral-2-eq-2 Nat.One-nat-def Nat.Suc-less-eq Nat.zero-less-Suc minus-nat.diff-0 Nat.diff-Suc-Suc Nat.le0

prod.case numeral-plus-one divmod-step-def order.refl le-zero-eq
le-numeral-simps less-numeral-simps mult.right-neutral divides-aux-eq
mult-nonneg-nonneg dvd-imp-mod-0 dvd-add zero-less-one mod-mult-self4 numeral-mod-numeral
divmod-trivial prod.sel mult.left-neutral div-pos-pos-trivial arith-simps div-add div-mult-self1
add-le-cancel-left add-le-same-cancel2 not-one-le-zero le-numeral-simps add-le-same-cancel1
zero-neq-one zero-le-one le-num-simps add-Suc mod-div-trivial nat.distinct mult-minus-right
add.inverse-inverse distrib-left-numeral mult-num-simps numeral-times-numeral add-num-simps
divmod-steps rel-simps if-True if-False numeral-div-numeral divmod-cancel prod.case
add-num-simps one-plus-numeral fst-conv arith-simps sub-num-simps dbl-inc-simps
dbl-simps mult-1 add-le-cancel-right left-diff-distrib-numeral add-uminus-conv-diff zero-neq-one
zero-le-one One-nat-def add-Suc mod-div-trivial nat.distinct of-int-1 numerals numeral-One
of-int-numeral add-uminus-conv-diff zle-diff1-eq add-less-same-cancel2 minus-add-distrib
add-uminus-conv-diff mult.left-neutral semiring-class.distrib-right
add-diff-cancel-left' ring-distribs mult-minus-left minus-diff-eq

lemmas [*smt-arith-simplify*] =
Groups.monoid-mult-class.mult-1-right Nat.mult-Suc-right
Nat.mult-0-right Nat.add-Suc-right Groups.monoid-add-class.add.right-neutral
Num.numeral-2-eq-2 Nat.One-nat-def
Nat.Suc-less-eq Nat.zero-less-Suc minus-nat.diff-0 Nat.diff-Suc-Suc Nat.le0
prod.case numeral-plus-one divmod-step-def order.refl le-zero-eq
mult.right-neutral divides-aux-eq
mult-nonneg-nonneg dvd-imp-mod-0 dvd-add zero-less-one mod-mult-self4 numeral-mod-numeral
divmod-trivial prod.sel mult.left-neutral div-pos-pos-trivial arith-simps div-add div-mult-self1
add-le-cancel-left add-le-same-cancel2 not-one-le-zero add-le-same-cancel1
zero-neq-one zero-le-one add-Suc mod-div-trivial nat.distinct mult-minus-right
add.inverse-inverse distrib-left-numeral
divmod-steps rel-simps if-True if-False numeral-div-numeral divmod-cancel
one-plus-numeral fst-conv
mult-1 add-le-cancel-right left-diff-distrib-numeral add-uminus-conv-diff
of-int-1 numerals numeral-One
of-int-numeral zle-diff1-eq add-less-same-cancel2 minus-add-distrib
semiring-class.distrib-right
add-diff-cancel-left' add-diff-eq ring-distribs mult-minus-left minus-diff-eq
mod-mult-self2-is-0
less-irrefl

lemmas [*arith-simp-cvc5*] =

smt-arith-simplify
ab-semigroup-mult-class.mult.commute

lemma [*arith-simp-cvc5*,*arith-mult-poly-norm-cvc5*,*smt-arith-simplify*]:
‹¬ (*a'* :: *'a* :: *linorder*) < *b'* ⟷ *b'* ≤ *a'*›
‹¬ (*a'* :: *'a* :: *linorder*) ≤ *b'* ⟷ *b'* < *a'*›
‹(*c*::*int*) *mod Numeral1* = *0*›
‹(*a*::*nat*) *mod Numeral1* = *0*›
‹(*c*::*int*) *div Numeral1* = *c*›
‹*a div Numeral1* = *a*›
‹(*c*::*int*) *mod 1* = *0*›
‹*a mod 1* = *0*›
‹(*c*::*int*) *div 1* = *c*›
‹*a div 1* = *a*›
‹¬(*a'* ≠ *b'*) ⟷ *a'* = *b'*›
by *auto*

lemma [*arith-simp-cvc5*,*arith-mult-poly-norm-cvc5*]:
‹*NO-MATCH 0* (*b*:: *int*) ⟹ *NO-MATCH 0* (*a*:: *int*) ⟹ *a* < *b* ⟷ *b* − *a* > *0*›
‹*NO-MATCH 0 b* ⟹ *NO-MATCH 0 a* ⟹ *a* ≤ *b* ⟷ *b* − *a* ≥ *0*›
‹*NO-MATCH 0 b* ⟹ *NO-MATCH 0 a* ⟹ *a* = *b* ⟷ *b* − *a* = *0*›
by *auto*

lemmas [*arith-simp-cvc5*,*arith-mult-poly-norm-cvc5*,*smt-arith-simplify*] = *divide-eq-eq-numeral1*
uminus-add-conv-diff divide-less-eq-numeral1
diff-gt-0-iff-gt times-divide-eq-left

lemma *div-mod-decomp*: *A* = (*A div n*) ∗ *n* + (*A mod n*) **for** *A* :: *nat*
by *auto*

lemma *div-less-mono*:
fixes *A B* :: *nat*
assumes *A* < *B 0* < *n* **and**
mod: *A mod n* = *0B mod n* = *0*
shows (*A div n*) < (*B div n*)
using *assms*(*1*)
by (*subst* (*asm*) *div-mod-decomp*[*of A n*], *subst* (*asm*) *div-mod-decomp*[*of B n*],
unfold mod, *use assms*(*2*,*3*) **in** ‹*auto simp*: *ac-simps*›)

lemma *alethe-le-mono-div*[*no-atp*]:
fixes *A B* :: *nat*
assumes *A* < *B 0* < *n*
shows (*A div n*) + (*if B mod n* = *0 then 1 else 0*) ≤ (*B div n*)
by (*auto simp*: *ac-simps Suc-leI assms less-mult-imp-div-less div-le-mono less-imp-le-nat*)

lemmas [*smt-arith-multiplication*] =
alethe-le-mono-div[*THEN mult-le-mono1*, *unfolded add-mult-distrib*]
div-le-mono[*THEN mult-le-mono2*, *unfolded add-mult-distrib*]

lemma *div-mod-decomp-int*: $A = (A\ div\ n) * n + (A\ mod\ n)$ **for** $A :: int$
by *auto*

lemma *zdiv-mono-strict*:
fixes $A\ B :: int$
assumes $A < B\ 0 < n$ **and**
mod: $A\ mod\ n = 0 B\ mod\ n = 0$
shows $(A\ div\ n) < (B\ div\ n)$
proof −
show *?thesis*
using *assms*(*1*)
apply (*subst* (*asm*) *div-mod-decomp-int*[*of A n*])
apply (*subst* (*asm*) *div-mod-decomp-int*[*of B n*])
unfolding *mod*
by (*use assms*(*2*,*3*) **in** ‹*auto simp*: *ac-simps*›)
qed

lemma *alethe-le-mono-div-int*[*no-atp*]:
‹$A\ div\ n + (if\ B\ mod\ n = 0\ then\ 1\ else\ 0) \leq B\ div\ n$›
if ‹$A < B$› ‹$0 < n$›
for $A\ B\ n :: int$
proof −
from ‹$A < B$› ‹$0 < n$› **have** ‹$A\ div\ n \leq B\ div\ n$›
by (*auto intro*: *zdiv-mono1*)
show *?thesis*
proof (*cases* ‹*n dvd B*›)
case *False*
with ‹$A\ div\ n \leq B\ div\ n$› **show** *?thesis*
by *auto*
next
case *True*
then obtain C **where** ‹$B = n * C$› **..**
then have ‹$B\ div\ n = C$›
using ‹$0 < n$› **by** *simp*
from ‹$0 < n$› **have** ‹$A\ mod\ n \geq 0$›
by *simp*
have ‹$A\ div\ n < C$›
proof (*rule ccontr*)
assume ‹$\neg\ A\ div\ n < C$›
then have ‹$C \leq A\ div\ n$›
by *simp*
with ‹$B\ div\ n = C$› ‹$A\ div\ n \leq B\ div\ n$›
have ‹$A\ div\ n = C$›
by *simp*
moreover from ‹$A < B$› **have** ‹$n * (A\ div\ n) + A\ mod\ n < B$›
by *simp*
ultimately have ‹$n * C + A\ mod\ n < n * C$›
using ‹$B = n * C$› **by** *simp*

```
    moreover have ‹A mod n ≥ 0›
      using ‹0 < n› by simp
    ultimately show False
      by simp
  qed
  with ‹n dvd B› ‹B div n = C› show ?thesis
    by simp
 qed
qed

lemma alethe-less-mono-div-int2[no-atp]:
 fixes A B :: int
 assumes A ≤ B 0 < −n
 shows (A div n) ≥ (B div n)
 using assms(1) assms(2) zdiv-mono1-neg by auto

lemmas [smt-arith-multiplication] =
 alethe-le-mono-div-int[THEN mult-left-mono, unfolded int-distrib]
 zdiv-mono1[THEN mult-left-mono, unfolded int-distrib]


lemma [smt-arith-multiplication]:
 (x::nat) = y ⟹ x div n * p = y div n * p
 (x'::int) = y' ⟹ x' div n' * p' = y' div n' * p'
 by simp-all

lemma [smt-arith-combine]:
 a < b ⟹ c < d ⟹ a + c + 2 ≤ b + d
 a < b ⟹ c ≤ d ⟹ a + c + 1 ≤ b + d
 a ≤ b ⟹ c < d ⟹ a + c + 1 ≤ b + d for a b c :: int
 by auto

lemma [smt-arith-combine]:
 a < b ⟹ c < d ⟹ a + c + 2 ≤ b + d
 a < b ⟹ c ≤ d ⟹ a + c + 1 ≤ b + d
 a ≤ b ⟹ c < d ⟹ a + c + 1 ≤ b + d for a b c :: nat
 by auto

lemmas [smt-arith-combine] =
 add-strict-mono
 add-less-le-mono
 add-mono
 add-le-less-mono

lemma [smt-arith-combine]:
 ‹m < n ⟹ c = d ⟹ m + c < n + d›
 ‹m ≤ n ⟹ c = d ⟹ m + c ≤ n + d›
 ‹c = d ⟹ m < n ⟹ m + c < n + d›
 ‹c = d ⟹ m ≤ n ⟹ m + c ≤ n + d›
```

for $m :: \langle 'a :: \mathit{ordered\text{-}cancel\text{-}ab\text{-}semigroup\text{-}add}\rangle$
by (*auto intro*: *ordered-cancel-ab-semigroup-add-class.add-strict-right-mono ordered-ab-semigroup-add-class.add-right-mono*)

lemma [*smt-arith-combine*]:
$c = d \Longrightarrow e = f \Longrightarrow c + e = d + f$
by *simp*

lemma *alethe-negate-coefficient*[*no-atp*]:
$\langle a \leq (b :: 'a :: \{\mathit{ordered\text{-}ab\text{-}group\text{-}add}\}) \Longrightarrow -a \geq -b\rangle$
$\langle a < b \Longrightarrow -a > -b\rangle$
$\langle a = b \Longrightarrow -a = -b\rangle$
by *auto*

lemma *alethe-invert-farkas-equation*[*no-atp*]:
$\langle a \leq (b :: 'a :: \{\mathit{ordered\text{-}ab\text{-}group\text{-}add}\}) \Longrightarrow -a \geq -b\rangle$
$\langle a < b \Longrightarrow -a > -b\rangle$
$\langle a = b \Longrightarrow b = a\rangle$
by *auto*

end

lemma *alethe-ite-intro*[*no-atp*]:
$\langle (\mathit{If}\ p\ (a' = (\mathit{If}\ p\ a'\ b'))\ (b' = (\mathit{If}\ p\ a'\ b'))) \longleftrightarrow \mathit{True}\rangle$
$\langle (\mathit{If}\ p\ (a' = (\mathit{If}\ p\ a'\ b'))\ ((\mathit{If}\ p\ a'\ b') = b')) \longleftrightarrow \mathit{True}\rangle$
$\langle (\mathit{If}\ p\ ((\mathit{If}\ p\ a'\ b') = a')\ (b' = (\mathit{If}\ p\ a'\ b'))) \longleftrightarrow \mathit{True}\rangle$
$\langle (\mathit{If}\ p\ ((\mathit{If}\ p\ a'\ b') = a')\ ((\mathit{If}\ p\ a'\ b') = b')) \longleftrightarrow \mathit{True}\rangle$
$\langle (\mathit{if}\ a\ \mathit{then}\ P\ (\mathit{if}\ a\ \mathit{then}\ a'\ \mathit{else}\ b')\ \mathit{else}\ Q) \longleftrightarrow (\mathit{if}\ a\ \mathit{then}\ P\ a'\ \mathit{else}\ Q)\rangle$
$\langle (\mathit{if}\ a\ \mathit{then}\ P'\ \mathit{else}\ Q'\ (\mathit{if}\ a\ \mathit{then}\ a'\ \mathit{else}\ b')) \longleftrightarrow (\mathit{if}\ a\ \mathit{then}\ P'\ \mathit{else}\ Q'\ b')\rangle$
$\langle A = f\ (\mathit{if}\ a\ \mathit{then}\ R\ \mathit{else}\ S) \longleftrightarrow (\mathit{if}\ a\ \mathit{then}\ A = f\ R\ \mathit{else}\ A = f\ S)\rangle$
by *auto*

lemma *alethe-ite-if-cong*[*no-atp*]:
fixes $x\ y :: \mathit{bool}$
assumes $b = c$
and $c \equiv \mathit{True} \Longrightarrow x = u$
and $c \equiv \mathit{False} \Longrightarrow y = v$
shows $(\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) \equiv (\mathit{if}\ c\ \mathit{then}\ u\ \mathit{else}\ v)$
proof −
have *H*: $(\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) = (\mathit{if}\ c\ \mathit{then}\ u\ \mathit{else}\ v)$
using *assms* **by** (*auto split*: *if-splits*)

show $(\mathit{if}\ b\ \mathit{then}\ x\ \mathit{else}\ y) \equiv (\mathit{if}\ c\ \mathit{then}\ u\ \mathit{else}\ v)$
by (*subst H*) *auto*
qed

lemma *alethe-miniscope-distribute*[*no-atp*]:
$\langle (\forall X.\ (F1\ X \wedge A\ X)) = ((\forall X.\ F1\ X) \wedge (\forall X.\ A\ X))\rangle$
$\langle (\exists X.\ (F1\ X \vee A\ X)) = ((\exists X.\ F1\ X) \vee (\exists X.\ A\ X))\rangle$

by (*simp-all only*: *all-conj-distrib ex-disj-distrib*)

lemma *alethe-miniscope-split*[*no-atp*]:
‹((∀ *x*. *F1 x*) ∨ (∀ *y*. *A y*)) = (∀ *x y*. (*F1 x* ∨ *A y*))›
‹((∀ *x*. *F1 x*) ∨ *B*) = (∀ *x*. (*F1 x* ∨ *B*))›
‹(*B* ∨ (∀ *x*. *F1 x*)) = (∀ *x*. (*B* ∨ *F1 x*))›
‹((∃ *x*. *F1 x*) ∧ (∃ *y*. *A y*)) = (∃ *x y*. (*F1 x* ∧ *A y*))›
‹((∃ *x*. *F1 x*) ∧ *B*) = (∃ *x*. (*F1 x* ∧ *B*))›
‹(*B* ∧ (∃ *x*. *F1 x*)) = (∃ *x*. (*B* ∧ *F1 x*))›
by *simp-all*

lemma *alethe-miniscope-ITE*[*no-atp*]:
‹(∀ *x Y*. (*if C then F1 x else A Y*)) = (*if C then* (∀ *x*. *F1 x*) *else* (∀ *Y*. *A Y*))›
‹(∀ *x*. (*if C then F1 x else B*)) = (*if C then* (∀ *x*. *F1 x*) *else* (∀ *Y*. *B*))›
‹(∀ *Y*. (*if C then F2 else A Y*)) = (*if C then F2 else* (∀ *Y*. *A Y*))›
by *simp-all*

named-theorems *alethe-poly-simp-rel* ‹*Theorems required to replay poly-simp-rel*›

lemma [*alethe-poly-simp-rel*]:
fixes *x1*::*int* **and** *x2 y1 y2 cx cy*
shows *cx* ≠ *0* ⟹ *cy* ≠ *0* ⟹
((*cx* ∗ (*x1* − *x2*)) = (*cy* ∗ (*y1* − *y2*))) ⟶ ((*x1* = *x2*) = (*y1* = *y2*))
by *force*

lemma [*alethe-poly-simp-rel*]:
fixes *x1*::*int* **and** *x2 y1 y2 cx cy*
shows ((*cx* > *0*) = (*cy* > *0*)) ⟹ *cx* ≠ *0* ⟹ *cy* ≠ *0* ⟹
((*cx* ∗ (*x1* − *x2*)) = (*cy* ∗ (*y1* − *y2*))) ⟶ ((*x1* < *x2*) = (*y1* < *y2*))
by (*metis less-iff-diff-less-0 mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1*::*int* **and** *x2 y1 y2 cx cy*
shows ((*cx* > *0*) = (*cy* > *0*)) ⟹ *cx* ≠ *0* ⟹ *cy* ≠ *0* ⟹
((*cx* ∗ (*x1* − *x2*)) = (*cy* ∗ (*y1* − *y2*))) ⟶ ((*x1* ≤ *x2*) = (*y1* ≤ *y2*))
by (*metis diff-gt-0-iff-gt linorder-not-le mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1*::*int* **and** *x2 y1 y2 cx cy*
shows ((*cx* > *0*) = (*cy* > *0*)) ⟹ *cx* ≠ *0* ⟹ *cy* ≠ *0* ⟹
((*cx* ∗ (*x1* − *x2*)) = (*cy* ∗ (*y1* − *y2*))) ⟶ ((*x1* > *x2*) = (*y1* > *y2*))
by (*metis le-iff-diff-le-0 linorder-not-le mult-less-0-iff nless-le zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1*::*int* **and** *x2 y1 y2 cx cy*
shows ((*cx* > *0*) = (*cy* > *0*)) ⟹ *cx* ≠ *0* ⟹ *cy* ≠ *0* ⟹
((*cx* ∗ (*x1* − *x2*)) = (*cy* ∗ (*y1* − *y2*))) ⟶ ((*x1* ≥ *x2*) = (*y1* ≥ *y2*))
by (*metis diff-ge-0-iff-ge linorder-not-le mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma *alethe-nat-embedding*[*no-atp*]:
$\exists\, x.\ x \geq 0 \wedge (y::nat) = nat\ (x::int)$
using *int-eq-iff* **by** *blast*

lemma *alethe-nat-embedding2*[*no-atp*]:
$int\ (nat\ (Num.numeral\text{-}class.numeral\ n)) \equiv numeral\ n$
by *simp*

lemma *alethe-nat-embedding-all*[*no-atp*]:
$(\forall\, (x::nat).\ P\ x) = (\forall\, (x::int).\ x \geq 0 \longrightarrow P\ (nat\ x))$
using *all-nat* **by** *simp-all*

lemma *alethe-nat-embedding-ex*[*no-atp*]:
$(\exists\, (x::nat).\ P\ x) = (\exists\, (x::int).\ x \geq 0 \wedge P\ (nat\ x))$
using *ex-nat* **by** *simp*

lemma *int-nat-embedding-preproc-all*[*no-atp*]:
$(\forall\, (x::nat)\ .\ P\ x) \equiv (\forall\, (x::int).\ x \geq 0 \longrightarrow P\ (nat\ x))$
using *all-nat* **by** *simp*

lemma *int-nat-embedding-preproc-ex*[*no-atp*]:
$(\exists\, (x::nat).\ P\ x) \equiv (\exists\, (x::int).\ x \geq 0 \wedge P\ (nat\ x))$
using *ex-nat* **by** *auto*
context
begin
qualified definition *lift-The* :: $(int \Rightarrow bool) \Rightarrow int$ **where**
$lift\text{-}The\ Q = int\ (The\ (\lambda n::nat.\ Q\ (int\ n)))$
lemma *int-nat-embedding-preproc-the*[*no-atp*]:
$(The\ (P::nat \Rightarrow bool)) \equiv nat\ (lift\text{-}The\ (\lambda z::int.\ 0 \leq z \wedge P\ (nat\ z)))$
by (*simp add*: *lift-The-def*)

lemma *alethe-nat-embedding-all-new*[*no-atp*]:
$(\forall\, x\ .\ (\exists\, x'.\ (((x::nat) = nat\ (x'::int) \wedge 0 \leq x'\ ) \wedge P\ x = P'\ x')))$
$\Longrightarrow (\forall\, x::nat.\ P\ x) = (\forall\, x'::int.\ x' \geq 0 \longrightarrow P'\ x')$
apply *simp*
apply *standard*+
subgoal using *eq-nat-nat-iff* **by** *blast*
by *blast*

lemma *alethe-nat-embedding-all2*[*no-atp*]:
$(\forall\, (x::int) \geq 0.\ int\ (nat\ x) = x)$
using *all-nat* **by** *simp*

lemma *alethe-nat-embedding-ex2*[*no-atp*]:
$(\exists\, (x::int).\ x \geq 0 \wedge int\ (nat\ x) = x)$
using *ex-nat* **by** *auto*

lemma *alethe-nat-embedding-all3*[*no-atp*]:
$P\ (int\ (nat\ x)) \Longrightarrow x \geq 0 \Longrightarrow P\ x$

using *all-nat* **by** *simp*

lemma *smt-lift-int-nat*[*no-atp*]:
int (*nat 0*) ≡ *0*
int (*nat 1*) ≡ *1*
by *simp-all*

lemma *smt-natasint-lift-nat-embedding*[*no-atp*]:
‹$x \geq 0 \Longrightarrow int\ (nat\ x) = x$›
by *simp*

lemma *smt-natasint-lift-nat-embedding'*[*no-atp*]:
‹$x \geq 0 \Longrightarrow int\ (nat\ x) \equiv x$›
by *simp*

lemma *smt-natasint-lift-nat-eq*[*no-atp*]:
fixes *a b* :: *int*
shows ‹$0 \leq a \Longrightarrow 0 \leq b \Longrightarrow nat\ a = nat\ b \equiv a = b$›
by (*simp add*: *nat-eq-iff*)

lemma *alethe-onepoint*[*no-atp*]:
‹$(\bigwedge a.\ P\ a = Q\ a) \Longrightarrow (\forall\, a.\ P\ a) = (\forall\, a.\ Q\ a)$›
‹$(\bigwedge a.\ P'\ a = Q') \Longrightarrow (\forall\, a.\ P'\ a) = Q'$›
‹$(\bigwedge a.\ P\ a = Q\ a) \Longrightarrow (\exists\, a.\ P\ a) = (\exists\, a.\ Q\ a)$›
‹$(\bigwedge a.\ P'\ a = Q') \Longrightarrow (\exists\, a.\ P'\ a) = Q'$›
by *auto*

lemma *alethe-onepoint-simp*[*no-atp*]:
‹$(\forall\, x.\ a = b \wedge P\ x \longrightarrow Q\ x) \longleftrightarrow (a = b \longrightarrow (\forall\, x.\ P\ x \longrightarrow Q\ x))$›
by *auto*

lemma *alethe-rewrite-in-imp*[*no-atp*]:
‹$((a::'a) = (b::'a) \equiv P') \Longrightarrow (P' \Longrightarrow P = Q) \Longrightarrow (a = b \longrightarrow P) \equiv (P' \longrightarrow Q)$›
by *auto*

lemma *alethe-imp-commute*[*no-atp*]: ‹$(a \longrightarrow b \longrightarrow q) = (b \longrightarrow a \longrightarrow q)$›
by *auto*

end

named-theorems *nat-normalized-input* ‹*Theorems required to replay nat operators embedded into lifted int versions*›
named-theorems *cvc5-normalized-input* ‹*Theorems required to replay our more complicated translations*›

named-theorems *cvc5-holes-simp* ‹*Simplification theorems for holes*›
named-theorems *cvc5-holes-pre* ‹*Theorems applied for holes*›

named-theorems *simplify-translation* ‹*Lemmas to normalize the assertions before*

translation. When an operator (such as a constant) appears the simplifier is called and uses this set›
lemmas [*simplify-translation*] = *arith-simps more-arith-simps*

named-theorems *alethe-poly-norm* ‹*Extra theorems for poly norm.*›
lemmas [*alethe-poly-norm*] = *mult-1-right mult-1-left*

named-theorems *alethe-aci-simp* ‹*Extra theorems for aci simp*›

lemma *alethe-onepoint2-all*[*no-atp*]:
  assumes ‹$\bigwedge x.\ x = t \Longrightarrow P\ t = Q$› ‹$\bigwedge x.\ P\ x = (x = t \longrightarrow P\ x)$›
  shows ‹$(\forall x.\ P\ x) = Q$›
  by (*use assms* **in** *metis*)+

lemma *alethe-onepoint2-ex*[*no-atp*]:
  assumes ‹$\bigwedge x.\ x = t \Longrightarrow P\ t = Q$› ‹$\bigwedge x.\ P\ x = (x = t \wedge P\ x)$›
  shows ‹$(\exists x.\ P\ x) = Q$›
  by (*use assms* **in** *metis*)+

lemmas *alethe-poly-simp-rel-simps*[*no-atp*] =
   less-divide-eq-numeral1
   less-numeral-simps alethe-div-simplify zero-less-one
   not-one-less-zero neg-0-less-iff-less
   Parity.semiring-parity-class.odd-numeral
   Parity.semiring-parity-class.odd-one

named-theorems *alethe-poly-simp-rel-bv-simps* ‹*Extra theorems for poly simp rel for bv*›

lemma *alethe-arg-cong0*[*no-atp*]: ‹$f = g \Longrightarrow f\ a = f\ a$›
  by (*rule arg-cong*)

lemma *alethe-arg-cong*[*no-atp*]: ‹$f = g \Longrightarrow a = b \Longrightarrow f\ a = g\ b$›
  by (*auto*)

context
begin

qualified definition *alethe-Box* :: ‹*bool* ⇒ *bool*› **where**
  ‹*alethe-Box P* = *P*›

qualified lemma *alethe-Box*[*no-atp*]:
  ‹$P \equiv$ *alethe-Box P*›
  unfolding *alethe-Box-def*
  by *auto*

qualified lemma *alethe-Box-def2*[*no-atp*]:
  ‹*alethe-Box P* ≡ *P*›
  unfolding *alethe-Box-def*

by *auto*

qualified definition *alethe-id* :: ‹*'a* ⇒ *'a*› **where**
‹*alethe-id x* = *x*›

end

named-theorems *rare-simplify-temp* ‹*Theorems to reconstruct bitvector theorems concerning list functions, e.g. take.*›

named-theorems *cvc-evaluate* ‹*Theorems to reconstruct evaluate steps in cvc5 proofs*›
named-theorems *cvc-evaluate-bv* ‹*Theorems to reconstruct bit−vector evaluate steps in cvc5 proofs*›

lemmas *cvc-arith-rewrite-defs* = *SMT.z3div-def linorder-not-le alethe-comp-simplify1 add1-zle-eq*

63.7 Setup

ML-file ‹*Tools/SMT/smt-util.ML*›
ML-file ‹*Tools/SMT/smt-failure.ML*›
ML-file ‹*Tools/SMT/smt-config.ML*›
ML-file ‹*Tools/SMT/smt-builtin.ML*›
ML-file ‹*Tools/SMT/smt-datatypes.ML*›
ML-file ‹*Tools/SMT/smt-normalize.ML*›
ML-file ‹*Tools/SMT/smt-translate.ML*›
ML-file ‹*Tools/SMT/smt-parser-util.ML*›
ML-file ‹*Tools/SMT/smtlib.ML*›
ML-file ‹*Tools/SMT/smtlib-interface.ML*›
ML-file ‹*Tools/SMT/smtlib-proof.ML*›
ML-file ‹*Tools/SMT/smtlib-isar.ML*›
ML-file ‹*Tools/SMT/smt-solver.ML*›

ML-file ‹*Tools/SMT/z3/z3-proof.ML*›
ML-file ‹*Tools/SMT/z3/z3-isar.ML*›

ML-file ‹*Tools/SMT/alethe/alethe-node.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-proof.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-smt-problem.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-isar.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-proof-parse.ML*›
ML-file ‹*Tools/SMT/alethe/cvc-interface.ML*›
ML-file ‹*Tools/SMT/alethe/cvc-proof-parse.ML*›

ML-file ‹*Tools/SMT/conj-disj-perm.ML*›

ML-file ‹*Tools/SMT/smt-replay-methods.ML*›
ML-file ‹*Tools/SMT/smt-replay.ML*›
ML-file ‹*Tools/SMT/smt-replay-arith.ML*›

ML-file ‹*Tools/SMT/z3/z3-interface.ML*›
ML-file ‹*Tools/SMT/z3/z3-replay-rules.ML*›
ML-file ‹*Tools/SMT/z3/z3-replay-methods.ML*›
ML-file ‹*Tools/SMT/z3/z3-replay.ML*›

ML-file ‹*Tools/SMT/alethe/alethe-replay-methods.ML*›
ML-file ‹*Tools/SMT/alethe/cvc5-replay-methods.ML*›
ML-file ‹*Tools/SMT/alethe/verit-replay-methods.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-strategies.ML*›
ML-file ‹*Tools/SMT/alethe/alethe-replay.ML*›
ML-file ‹*Tools/SMT/alethe/verit-replay.ML*›
ML-file ‹*Tools/SMT/alethe/cvc5-replay.ML*›

ML-file ‹*Tools/SMT/smt-systems.ML*›

63.8 Configuration

The current configuration can be printed by the command *smt-status*, which shows the values of most options.

63.9 General configuration options

The option *smt-solver* can be used to change the target SMT solver. The possible values can be obtained from the *smt-status* command.

declare [[*smt-solver* = *z3*]]

Since SMT solvers are potentially nonterminating, there is a timeout (given in seconds) to restrict their runtime.

declare [[*smt-timeout* = *0*]]

SMT solvers apply randomized heuristics. In case a problem is not solvable by an SMT solver, changing the following option might help.

declare [[*smt-random-seed* = *1*]]

In general, the binding to SMT solvers runs as an oracle, i.e, the SMT solvers are fully trusted without additional checks. The following option can cause the SMT solver to run in proof-producing mode, giving a checkable certificate. This is currently implemented only for veriT and Z3.

declare [[*smt-oracle* = *false*]]

Each SMT solver provides several command-line options to tweak its behaviour. They can be passed to the solver by setting the following options.

declare [[*cvc4-options* =]]
declare [[*cvc5-options* = *−−proof−alethe−define−skolems −−proof−elim−subtypes −−enum−inst*]]
declare [[*cvc5-proof-options* = *−−proof−format−mode=alethe −−proof−granularity=dsl−rewrite −−proof−mode=full−proof−strict −−enum−inst*]]
declare [[*verit-options* = *−−proof−with−sharing*]]
declare [[*z3-options* =]]

The SMT method provides an inference mechanism to detect simple triggers in quantified formulas, which might increase the number of problems solvable by SMT solvers (note: triggers guide quantifier instantiations in the SMT solver). To turn it on, set the following option.

declare [[*smt-infer-triggers* = *false*]]

Enable the following options to use built-in support for datatypes and co-datatypes.

declare [[*smt-native-datatypes* = *false*]]
declare [[*smt-native-codatatypes* = *false*]]

Enable the following option to use built-in support for div/mod, datatypes, and records in Z3. Currently, this is implemented only in oracle mode.

declare [[*z3-extensions* = *false*]]

63.10 Certificates

By setting the option *smt-certificates* to the name of a file, all following applications of an SMT solver are cached in that file. Any further application of the same SMT solver (using the very same configuration) re-uses the cached certificate instead of invoking the solver. An empty string disables caching certificates.

The filename should be given as an explicit path. It is good practice to use the name of the current theory (with ending *.certs* instead of *.thy*) as the certificates file. Certificate files should be used at most once in a certain theory context, to avoid race conditions with other concurrent accesses.

declare [[*smt-certificates* =]]

The option *smt-read-only-certificates* controls whether only stored certificates should be used or invocation of an SMT solver is allowed. When set to *true*, no SMT solver will ever be invoked and only the existing certificates found in the configured cache are used; when set to *false* and there is no cached certificate for some proposition, then the configured SMT solver is invoked.

declare [[*smt-read-only-certificates* = *false*]]

63.11 Tracing

The SMT method, when applied, traces important information. To make it entirely silent, set the following option to *false*.

declare [[*smt-verbose* = *true*]]

For tracing the generated problem file given to the SMT solver as well as the returned result of the solver, the option *smt-trace* should be set to *true*.

declare [[*smt-trace* = *false*]]

63.12 Schematic rules for Z3 proof reconstruction

Several prof rules of Z3 are not very well documented. There are two lemma groups which can turn failing Z3 proof reconstruction attempts into succeeding ones: the facts in *z3-rule* are tried prior to any implemented reconstruction procedure for all uncertain Z3 proof rules; the facts in *z3-simp* are only fed to invocations of the simplifier when reconstructing theory-specific proof steps.

lemmas [*z3-rule*] =
refl eq-commute conj-commute disj-commute simp-thms nnf-simps
ring-distribs field-simps times-divide-eq-right times-divide-eq-left
if-True if-False not-not
NO-MATCH-def

lemma [*z3-rule*]:
$(P \wedge Q) = (\neg\ (\neg\ P \vee \neg\ Q))$
$(P \wedge Q) = (\neg\ (\neg\ Q \vee \neg\ P))$
$(\neg\ P \wedge Q) = (\neg\ (P \vee \neg\ Q))$
$(\neg\ P \wedge Q) = (\neg\ (\neg\ Q \vee P))$
$(P \wedge \neg\ Q) = (\neg\ (\neg\ P \vee Q))$
$(P \wedge \neg\ Q) = (\neg\ (Q \vee \neg\ P))$
$(\neg\ P \wedge \neg\ Q) = (\neg\ (P \vee Q))$
$(\neg\ P \wedge \neg\ Q) = (\neg\ (Q \vee P))$
by *auto*

lemma [*z3-rule*]:
$(P \longrightarrow Q) = (Q \vee \neg\ P)$
$(\neg\ P \longrightarrow Q) = (P \vee Q)$
$(\neg\ P \longrightarrow Q) = (Q \vee P)$
$(\mathit{True} \longrightarrow P) = P$
$(P \longrightarrow \mathit{True}) = \mathit{True}$
$(\mathit{False} \longrightarrow P) = \mathit{True}$
$(P \longrightarrow P) = \mathit{True}$
$(\neg\ (A \longleftrightarrow \neg\ B)) \longleftrightarrow (A \longleftrightarrow B)$
by *auto*

lemma [*z3-rule*]:

$((P = Q) \longrightarrow R) = (R \vee (Q = (\neg\ P)))$
by *auto*

lemma [*z3-rule*]:
$(\neg\ True) = False$
$(\neg\ False) = True$
$(x = x) = True$
$(P = True) = P$
$(True = P) = P$
$(P = False) = (\neg\ P)$
$(False = P) = (\neg\ P)$
$((\neg\ P) = P) = False$
$(P = (\neg\ P)) = False$
$((\neg\ P) = (\neg\ Q)) = (P = Q)$
$\neg\ (P = (\neg\ Q)) = (P = Q)$
$\neg\ ((\neg\ P) = Q) = (P = Q)$
$(P \neq Q) = (Q = (\neg\ P))$
$(P = Q) = ((\neg\ P \vee Q) \wedge (P \vee \neg\ Q))$
$(P \neq Q) = ((\neg\ P \vee \neg\ Q) \wedge (P \vee Q))$
by *auto*

lemma [*z3-rule*]:
$(if\ P\ then\ P\ else\ \neg\ P) = True$
$(if\ \neg\ P\ then\ \neg\ P\ else\ P) = True$
$(if\ P\ then\ True\ else\ False) = P$
$(if\ P\ then\ False\ else\ True) = (\neg\ P)$
$(if\ P\ then\ Q\ else\ True) = ((\neg\ P) \vee Q)$
$(if\ P\ then\ Q\ else\ True) = (Q \vee (\neg\ P))$
$(if\ P\ then\ Q\ else\ \neg\ Q) = (P = Q)$
$(if\ P\ then\ Q\ else\ \neg\ Q) = (Q = P)$
$(if\ P\ then\ \neg\ Q\ else\ Q) = (P = (\neg\ Q))$
$(if\ P\ then\ \neg\ Q\ else\ Q) = ((\neg\ Q) = P)$
$(if\ \neg\ P\ then\ x\ else\ y) = (if\ P\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ x) = (if\ P \wedge (\neg\ Q)\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ x) = (if\ (\neg\ Q) \wedge P\ then\ y\ else\ x)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ y) = (if\ P \wedge Q\ then\ x\ else\ y)$
$(if\ P\ then\ (if\ Q\ then\ x\ else\ y)\ else\ y) = (if\ Q \wedge P\ then\ x\ else\ y)$
$(if\ P\ then\ x\ else\ if\ P\ then\ y\ else\ z) = (if\ P\ then\ x\ else\ z)$
$(if\ P\ then\ x\ else\ if\ Q\ then\ x\ else\ y) = (if\ P \vee Q\ then\ x\ else\ y)$
$(if\ P\ then\ x\ else\ if\ Q\ then\ x\ else\ y) = (if\ Q \vee P\ then\ x\ else\ y)$
$(if\ P\ then\ x = y\ else\ x = z) = (x = (if\ P\ then\ y\ else\ z))$
$(if\ P\ then\ x = y\ else\ y = z) = (y = (if\ P\ then\ x\ else\ z))$
$(if\ P\ then\ x = y\ else\ z = y) = (y = (if\ P\ then\ x\ else\ z))$
by *auto*

lemma [*z3-rule*]:
$0 + (x::int) = x$
$x + 0 = x$
$x + x = 2 * x$

$0 * x = 0$
$1 * x = x$
$x + y = y + x$
by *auto*

lemma [*z3-rule*]:
$P = Q \vee P \vee Q$
$P = Q \vee \neg P \vee \neg Q$
$(\neg P) = Q \vee \neg P \vee Q$
$(\neg P) = Q \vee P \vee \neg Q$
$P = (\neg Q) \vee \neg P \vee Q$
$P = (\neg Q) \vee P \vee \neg Q$
$P \neq Q \vee P \vee \neg Q$
$P \neq Q \vee \neg P \vee Q$
$P \neq (\neg Q) \vee P \vee Q$
$(\neg P) \neq Q \vee P \vee Q$
$P \vee Q \vee P \neq (\neg Q)$
$P \vee Q \vee (\neg P) \neq Q$
$P \vee \neg Q \vee P \neq Q$
$\neg P \vee Q \vee P \neq Q$
$P \vee y = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$P \vee (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = y$
$\neg P \vee x = (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y)$
$\neg P \vee (\mathit{if}\ P\ \mathit{then}\ x\ \mathit{else}\ y) = x$
$P \vee R \vee \neg (\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg P \vee Q \vee \neg (\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R)$
$\neg (\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$\neg (\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee R$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee \neg P \vee \neg Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else}\ R) \vee P \vee \neg R$
$(\mathit{if}\ P\ \mathit{then} \neg Q\ \mathit{else}\ R) \vee \neg P \vee Q$
$(\mathit{if}\ P\ \mathit{then}\ Q\ \mathit{else} \neg R) \vee P \vee R$
by *auto*

hide-type (**open**) *symb-list pattern*
hide-const (**open**) *Symb-Nil Symb-Cons trigger pat nopat fun-app z3div z3mod*

declare[[*smt-cvc-alethe* = *true*]]

end

64 Sledgehammer: Isabelle–ATP Linkup

theory *Sledgehammer*
imports
— FIXME: *HOL.Try0-HOL* has to be imported first so that *try0-schedule* gets the value assigned value there. Otherwise, the value is the one assigned in *HOL.Try0*, which is imported transitively by both *HOL.Presburger* and *HOL.SMT*. It seems that, when merging the attributes from two theories, the value assigned int the

leftmost theory has precedence.
Try0-HOL
Presburger
SMT
keywords
sledgehammer :: *diag* **and**
sledgehammer-params :: *thy-decl*
begin

syntax (**output**)
-constrain-keep :: *logic* => *type* => *logic* (- :: - [*4*, *0*] *3*)
-constrain-keep :: *prop'* => *type* => *prop'* (- :: - [*4*, *0*] *3*)

ML-file ‹*Tools/ATP/system-on-tptp.ML*›
ML-file ‹*Tools/Sledgehammer/async-manager-legacy.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-util.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-fact.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-proof-methods.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-instantiations.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar-annotate.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar-proof.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar-preplay.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar-compress.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar-minimize.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-isar.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-atp-systems.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-prover.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-prover-atp.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-prover-smt.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-prover-tactic.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-prover-minimize.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-mepo.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-mash.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-commands.ML*›
ML-file ‹*Tools/Sledgehammer/sledgehammer-tactics.ML*›

end

65 Setup for Lifting/Transfer for the set type

theory *Lifting-Set*
imports *Lifting Groups-Big*
begin

65.1 Relator and predicator properties

lemma *rel-setD1*: ⟦ *rel-set R A B*; *x* ∈ *A* ⟧ ⟹ ∃ *y* ∈ *B*. *R x y*
and *rel-setD2*: ⟦ *rel-set R A B*; *y* ∈ *B* ⟧ ⟹ ∃ *x* ∈ *A*. *R x y*

by (*simp-all add*: *rel-set-def*)

lemma *rel-set-conversep* [*simp*]: *rel-set* A^{-1-1} = (*rel-set A*)$^{-1-1}$
unfolding *rel-set-def* **by** *auto*

lemma *rel-set-eq* [*relator-eq*]: *rel-set* (=) = (=)
unfolding *rel-set-def fun-eq-iff* **by** *auto*

lemma *rel-set-mono*[*relator-mono*]:
assumes $A \leq B$
shows *rel-set A* ≤ *rel-set B*
using *assms* **unfolding** *rel-set-def* **by** *blast*

lemma *rel-set-OO*[*relator-distr*]: *rel-set R OO rel-set S* = *rel-set* (*R OO S*)
apply (*rule sym*)
apply (*intro ext*)
subgoal for *X Z*
apply (*rule iffI*)
apply (*rule relcomppI* [**where** *b*={*y*. (∃ *x*∈*X*. *R x y*) ∧ (∃ *z*∈*Z*. *S y z*)}])
apply (*simp add*: *rel-set-def*, *fast*)+
done
done

lemma *Domainp-set*[*relator-domain*]:
Domainp (*rel-set T*) = (λ*A*. *Ball A* (*Domainp T*))
unfolding *rel-set-def Domainp-iff*[*abs-def*]
apply (*intro ext*)
apply (*rule iffI*)
apply *blast*
subgoal for *A* **by** (*rule exI* [**where** *x*={*y*. ∃ *x*∈*A*. *T x y*}]) *fast*
done

lemma *left-total-rel-set*[*transfer-rule*]:
left-total A ⟹ *left-total* (*rel-set A*)
unfolding *left-total-def rel-set-def*
apply *safe*
subgoal for *X* **by** (*rule exI* [**where** *x*={*y*. ∃ *x*∈*X*. *A x y*}]) *fast*
done

lemma *left-unique-rel-set*[*transfer-rule*]:
left-unique A ⟹ *left-unique* (*rel-set A*)
unfolding *left-unique-def rel-set-def*
by *fast*

lemma *right-total-rel-set* [*transfer-rule*]:
right-total A ⟹ *right-total* (*rel-set A*)
using *left-total-rel-set*[*of* A^{-1-1}] **by** *simp*

lemma *right-unique-rel-set* [*transfer-rule*]:

right-unique A $\Longrightarrow$ *right-unique* (*rel-set A*)
unfolding *right-unique-def rel-set-def* **by** *fast*

lemma *bi-total-rel-set* [*transfer-rule*]:
bi-total A $\Longrightarrow$ *bi-total* (*rel-set A*)
by(*simp add*: *bi-total-alt-def left-total-rel-set right-total-rel-set*)

lemma *bi-unique-rel-set* [*transfer-rule*]:
bi-unique A $\Longrightarrow$ *bi-unique* (*rel-set A*)
unfolding *bi-unique-def rel-set-def* **by** *fast*

lemma *set-relator-eq-onp* [*relator-eq-onp*]:
rel-set (*eq-onp P*) = *eq-onp* (λ*A*. *Ball A P*)
unfolding *fun-eq-iff rel-set-def eq-onp-def Ball-def* **by** *fast*

lemma *bi-unique-rel-set-lemma*:
assumes *bi-unique R* **and** *rel-set R X Y*
obtains *f* **where** *Y* = *image f X* **and** *inj-on f X* **and** $\forall x{\in}X.$ *R x* (*f x*)
proof
define *f* **where** *f x* = (*THE y*. *R x y*) **for** *x*
{ fix *x* **assume** *x* $\in$ *X*
with ‹*rel-set R X Y*› ‹*bi-unique R*› **have** *R x* (*f x*)
by (*simp add*: *bi-unique-def rel-set-def f-def*) (*metis theI*)
with *assms* ‹*x* $\in$ *X*›
have *R x* (*f x*) $\forall x'{\in}X.$ *R x'* (*f x*) $\longrightarrow$ *x* = *x'* $\forall y{\in}Y.$ *R x y* $\longrightarrow$ *y* = *f x f x* $\in$ *Y*
by (*fastforce simp add*: *bi-unique-def rel-set-def*)+ **}**
note $*$ = *this*
moreover
{ fix *y* **assume** *y* $\in$ *Y*
with ‹*rel-set R X Y*› $*$(*3*) ‹*y* $\in$ *Y*› **have** $\exists x{\in}X.$ *y* = *f x*
by (*fastforce simp*: *rel-set-def*) **}**
ultimately show $\forall x{\in}X.$ *R x* (*f x*) *Y* = *image f X inj-on f X*
by (*auto simp*: *inj-on-def image-iff*)
qed

65.2 Quotient theorem for the Lifting package

lemma *Quotient-set*[*quot-map*]:
assumes *Quotient R Abs Rep T*
shows *Quotient* (*rel-set R*) (*image Abs*) (*image Rep*) (*rel-set T*)
using *assms* **unfolding** *Quotient-alt-def4*
apply (*simp add*: *rel-set-OO*[*symmetric*])
apply (*simp add*: *rel-set-def*)
apply *fast*
done

65.3 Transfer rules for the Transfer package

65.3.1 Unconditional transfer rules

context includes *lifting-syntax*
begin

lemma *empty-transfer* [*transfer-rule*]: (*rel-set A*) {} {}
  unfolding *rel-set-def* **by** *simp*

lemma *insert-transfer* [*transfer-rule*]:
  (*A* ===> *rel-set A* ===> *rel-set A*) *insert insert*
  unfolding *rel-fun-def rel-set-def* **by** *auto*

lemma *union-transfer* [*transfer-rule*]:
  (*rel-set A* ===> *rel-set A* ===> *rel-set A*) *union union*
  unfolding *rel-fun-def rel-set-def* **by** *auto*

lemma *Union-transfer* [*transfer-rule*]:
  (*rel-set* (*rel-set A*) ===> *rel-set A*) *Union Union*
  unfolding *rel-fun-def rel-set-def* **by** *simp fast*

lemma *image-transfer* [*transfer-rule*]:
  ((*A* ===> *B*) ===> *rel-set A* ===> *rel-set B*) *image image*
  unfolding *rel-fun-def rel-set-def* **by** *simp fast*

lemma *UNION-transfer* [*transfer-rule*]: — TODO deletion candidate
  (*rel-set A* ===> (*A* ===> *rel-set B*) ===> *rel-set B*) (λ*A f*. ⋃(*f* ‘ *A*)) (λ*A f*. ⋃(*f* ‘ *A*))
  by *transfer-prover*

lemma *Ball-transfer* [*transfer-rule*]:
  (*rel-set A* ===> (*A* ===> (=)) ===> (=)) *Ball Ball*
  unfolding *rel-set-def rel-fun-def* **by** *fast*

lemma *Bex-transfer* [*transfer-rule*]:
  (*rel-set A* ===> (*A* ===> (=)) ===> (=)) *Bex Bex*
  unfolding *rel-set-def rel-fun-def* **by** *fast*

lemma *Pow-transfer* [*transfer-rule*]:
  (*rel-set A* ===> *rel-set* (*rel-set A*)) *Pow Pow*
  apply (*rule rel-funI*)
  apply (*rule rel-setI*)
  subgoal for *X Y X′*
    apply (*rule rev-bexI* [**where** *x*={*y*∈*Y*. ∃ *x*∈*X′*. *A x y*}])
    apply *clarsimp*
    apply (*simp add*: *rel-set-def*)
    apply *fast*
    done
  subgoal for *X Y Y′*

apply (*rule rev-bexI* [**where** x={x∈X. ∃ y∈Y'. A x y}])
apply *clarsimp*
apply (*simp add*: *rel-set-def*)
apply *fast*
done
done

lemma *rel-set-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> (=)) ===> *rel-set A* ===> *rel-set B* ===> (=)) *rel-set rel-set*
unfolding *rel-fun-def rel-set-def* **by** *fast*

lemma *bind-transfer* [*transfer-rule*]:
(*rel-set A* ===> (*A* ===> *rel-set B*) ===> *rel-set B*) *Set.bind Set.bind*
unfolding *bind-UNION* [*abs-def*] **by** *transfer-prover*

lemma *INF-parametric* [*transfer-rule*]: — TODO deletion candidate
(*rel-set A* ===> (*A* ===> *HOL.eq*) ===> *HOL.eq*) (λ*A f*. *Inf* (*f* ‘ *A*)) (λ*A f*. *Inf* (*f* ‘ *A*))
by *transfer-prover*

lemma *SUP-parametric* [*transfer-rule*]: — TODO deletion candidate
(*rel-set R* ===> (*R* ===> *HOL.eq*) ===> *HOL.eq*) (λ*A f*. *Sup* (*f* ‘ *A*)) (λ*A f*. *Sup* (*f* ‘ *A*))
by *transfer-prover*

65.3.2 Rules requiring bi-unique, bi-total or right-total relations

lemma *member-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-set A* ===> (=)) (∈) (∈)
using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def* **by** *fast*

lemma *right-total-Collect-transfer*[*transfer-rule*]:
assumes *right-total A*
shows ((*A* ===> (=)) ===> *rel-set A*) (λ*P*. *Collect* (λ*x*. *P x* ∧ *Domainp A x*)) *Collect*
using *assms* **unfolding** *right-total-def rel-set-def rel-fun-def Domainp-iff* **by** *fast*

lemma *Collect-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows ((*A* ===> (=)) ===> *rel-set A*) *Collect Collect*
using *assms* **unfolding** *rel-fun-def rel-set-def bi-total-def* **by** *fast*

lemma *inter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> *rel-set A*) *inter inter*
using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def* **by** *fast*

lemma *Diff-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> *rel-set A*) (−) (−)
using *assms* **unfolding** *rel-fun-def rel-set-def bi-unique-def*
unfolding *Ball-def Bex-def Diff-eq*
by (*safe*, *simp*, *metis*, *simp*, *metis*)

lemma *subset-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> (=)) (⊆) (⊆)
unfolding *subset-eq* [*abs-def*] **by** *transfer-prover*

context
includes *lifting-syntax*
begin

lemma *strict-subset-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set A* ===> *rel-set A* ===> (=)) (⊂) (⊂)
unfolding *subset-not-subset-eq* **by** *transfer-prover*

end

declare *right-total-UNIV-transfer*[*transfer-rule*]

lemma *UNIV-transfer* [*transfer-rule*]:
assumes *bi-total A*
shows (*rel-set A*) *UNIV UNIV*
using *assms* **unfolding** *rel-set-def bi-total-def* **by** *simp*

lemma *right-total-Compl-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set A* ===> *rel-set A*) (λ*S*. *uminus S* ∩ *Collect* (*Domainp A*)) *uminus*
unfolding *Compl-eq* [*abs-def*]
by (*subst Collect-conj-eq*[*symmetric*]) *transfer-prover*

lemma *Compl-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set A* ===> *rel-set A*) *uminus uminus*
unfolding *Compl-eq* [*abs-def*] **by** *transfer-prover*

lemma *right-total-Inter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*S*. ⋂*S* ∩ *Collect* (*Domainp A*)) *Inter*
unfolding *Inter-eq*[*abs-def*]
by (*subst Collect-conj-eq*[*symmetric*]) *transfer-prover*

lemma *Inter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) *Inter Inter*
unfolding *Inter-eq* [*abs-def*] **by** *transfer-prover*

lemma *filter-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows ((*A* ===> (=)) ===> *rel-set A* ===> *rel-set A*) *Set.filter Set.filter*
by (*simp add*: *rel-fun-def rel-set-def*) *blast*

lemma *finite-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set A* ===> (=)) *finite finite*
by (*rule rel-funI*, *erule* (*1*) *bi-unique-rel-set-lemma*)
(*auto dest*: *finite-imageD*)

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set A* ===> (=)) *card card*
by (*rule rel-funI*, *erule* (*1*) *bi-unique-rel-set-lemma*)
(*simp add*: *card-image*)

context
includes *lifting-syntax*
begin

lemma *vimage-right-total-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique B right-total A*
shows ((*A* ===> *B*) ===> *rel-set B* ===> *rel-set A*) (λ*f X*. *f* −‘ *X* ∩ *Collect* (*Domainp A*)) *vimage*
proof −
let *?vimage* = (λ*f B*. {*x*. *f x* ∈ *B* ∧ *Domainp A x*})
have ((*A* ===> *B*) ===> *rel-set B* ===> *rel-set A*) *?vimage vimage*
unfolding *vimage-def*
by *transfer-prover*
also have *?vimage* = (λ*f X*. *f* −‘ *X* ∩ *Collect* (*Domainp A*))
by *auto*
finally show *?thesis* **.**
qed

end

lemma *vimage-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique B*
shows ((*A* ===> *B*) ===> *rel-set B* ===> *rel-set A*) *vimage vimage*
unfolding *vimage-def*[*abs-def*] **by** *transfer-prover*

lemma *Image-parametric* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-set* (*rel-prod A B*) ===> *rel-set A* ===> *rel-set B*) (‘‘) (‘‘)
by (*intro rel-funI rel-setI*)

(*force dest*: *rel-setD1 bi-uniqueDr*[*OF assms*], *force dest*: *rel-setD2 bi-uniqueDl*[*OF assms*])

lemma *inj-on-transfer*[*transfer-rule*]:
  ((*A* ===> *B*) ===> *rel-set A* ===> (=)) *inj-on inj-on*
  if [*transfer-rule*]: *bi-unique A bi-unique B*
  unfolding *inj-on-def*
  by *transfer-prover*

end

lemma (**in** *comm-monoid-set*) *F-parametric* [*transfer-rule*]:
  fixes *A* :: *'b* ⇒ *'c* ⇒ *bool*
  assumes *bi-unique A*
  shows *rel-fun* (*rel-fun A* (=)) (*rel-fun* (*rel-set A*) (=)) *F F*
proof (*rule rel-funI*)+
  fix *f* :: *'b* ⇒ *'a* **and** *g S T*
  assume *rel-fun A* (=) *f g rel-set A S T*
  with ‹*bi-unique A*› **obtain** *i* **where** *bij-betw i S T* ⋀*x*. *x* ∈ *S* ⟹ *f x* = *g* (*i x*)
    by (*auto elim*: *bi-unique-rel-set-lemma simp*: *rel-fun-def bij-betw-def*)
  then show *F f S* = *F g T*
    by (*simp add*: *reindex-bij-betw*)
qed

lemmas *sum-parametric* = *sum.F-parametric*
lemmas *prod-parametric* = *prod.F-parametric*

lemma *rel-set-UNION*:
  assumes [*transfer-rule*]: *rel-set Q A B rel-fun Q* (*rel-set R*) *f g*
  shows *rel-set R* (⋃(*f* ‘ *A*)) (⋃(*g* ‘ *B*))
  by *transfer-prover*

context
  includes *lifting-syntax*
begin

lemma *fold-graph-transfer*[*transfer-rule*]:
  assumes *bi-unique R right-total R*
  shows ((*R* ===> (=) ===> (=)) ===> (=) ===> *rel-set R* ===> (=) ===> (=)) *fold-graph fold-graph*
proof(*intro rel-funI*)
  fix *f1* :: *'a* ⇒ *'c* ⇒ *'c* **and** *f2* :: *'b* ⇒ *'c* ⇒ *'c*
  assume *rel-f*: (*R* ===> (=) ===> (=)) *f1 f2*
  fix *z1 z2* :: *'c* **assume** [*simp*]: *z1* = *z2*
  fix *A1 A2* **assume** *rel-A*: *rel-set R A1 A2*
  fix *y1 y2* :: *'c* **assume** [*simp*]: *y1* = *y2*

  from ‹*bi-unique R*› ‹*right-total R*› **have** *The-y*: ∀ *y*. ∃!*x*. *R x y*
    unfolding *bi-unique-def right-total-def* **by** *auto*

define *r* **where** *r* ≡ λ*y*. *THE x*. *R x y*

from *The-y* **have** *r-y*: *R* (*r y*) *y* **for** *y*
unfolding *r-def* **using** *the-equality* **by** *fastforce*
with *assms rel-A* **have** *inj-on r A2 A1* = *r* ‘ *A2*
unfolding *r-def rel-set-def inj-on-def bi-unique-def*
apply(*auto simp*: *image-iff*) **by** *metis+*
with ‹*bi-unique R*› *rel-f r-y* **have** (*f1 o r*) *y* = *f2 y* **for** *y*
unfolding *bi-unique-def rel-fun-def* **by** *auto*
then have (*f1 o r*) = *f2*
by *blast*
then show *fold-graph f1 z1 A1 y1* = *fold-graph f2 z2 A2 y2*
by (*fastforce simp*: *fold-graph-image*[*OF* ‹*inj-on r A2*›] ‹*A1* = *r* ‘ *A2*›)
qed

lemma *fold-transfer*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R right-total R*
shows ((*R* ===> (=) ===> (=)) ===> (=) ===> *rel-set R* ===> (=))
Finite-Set.fold Finite-Set.fold
unfolding *Finite-Set.fold-def*
by *transfer-prover*

end

end

66 The datatype of finite lists

theory *List*
imports *Sledgehammer Lifting-Set*
begin

datatype (*set*: *'a*) *list* =
Nil (‹[]›)
| *Cons* (*hd*: *'a*) (*tl*: *'a list*) (**infixr** ‹#› *65*)
for
map: *map*
rel: *list-all2*
pred: *list-all*
where
tl [] = []

bundle *list-syntax*
begin
notation *Nil* (‹[]›)
and *Cons* (**infixr** ‹#› *65*)
end

datatype-compat *list*

lemma [*case-names Nil Cons*, *cases type*: *list*]:
— for backward compatibility – names of variables differ
$(y = [] \Longrightarrow P) \Longrightarrow (\bigwedge a\ list.\ y = a \# list \Longrightarrow P) \Longrightarrow P$
by (*rule list.exhaust*)

lemma [*case-names Nil Cons*, *induct type*: *list*]:
— for backward compatibility – names of variables differ
$P\ [] \Longrightarrow (\bigwedge a\ list.\ P\ list \Longrightarrow P\ (a \# list)) \Longrightarrow P\ list$
by (*rule list.induct*)

Compatibility:

setup ‹*Sign.mandatory-path list*›

lemmas *inducts* = *list.induct*
lemmas *recs* = *list.rec*
lemmas *cases* = *list.case*

setup ‹*Sign.parent-path*›

lemmas *set-simps* = *list.set*

List enumeration

open-bundle *list-enumeration-syntax*
begin

syntax
-list :: *args* ⇒ *'a list* (‹(‹*indent=1 notation=‹mixfix list enumeration›*›[-])›)
syntax-consts
-list ⇌ *Cons*
translations
[*x*, *xs*] ⇌ *x#[xs]*
[*x*] ⇌ *x#[]*

end

66.1 Basic list processing functions

primrec (*nonexhaustive*) *last* :: *'a list* ⇒ *'a* **where**
last (*x* # *xs*) = (*if xs* = [] *then x else last xs*)

primrec *butlast* :: *'a list* ⇒ *'a list* **where**
butlast [] = [] |
butlast (*x* # *xs*) = (*if xs* = [] *then* [] *else x* # *butlast xs*)

lemma *set-rec*: *set xs* = *rec-list* {} (λ*x* -. *insert x*) *xs*
by (*induct xs*) *auto*

definition *coset* :: *'a list* ⇒ *'a set* **where**
[*simp*]: *coset xs* = − *set xs*

primrec *append* :: *'a list* ⇒ *'a list* ⇒ *'a list* (**infixr** ‹@› *65*) **where**
append-Nil: [] @ *ys* = *ys* |
append-Cons: (*x*#*xs*) @ *ys* = *x* # *xs* @ *ys*

primrec *rev* :: *'a list* ⇒ *'a list* **where**
rev [] = [] |
rev (*x* # *xs*) = *rev xs* @ [*x*]

primrec *filter*:: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list* **where**
filter P [] = [] |
filter P (*x* # *xs*) = (*if P x then x* # *filter P xs else filter P xs*)

open-bundle *filter-syntax* — Special input syntax for filter
begin

syntax (*ASCII*)
-*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list* (‹(‹*indent=1 notation=‹mixfix filter›*›[-<−-./
-])›)
syntax
-*filter* :: [*pttrn*, *'a list*, *bool*] => *'a list* (‹(‹*indent=1 notation=‹mixfix filter›*›[-←-
./ -])›)
syntax-consts
-*filter* ⇌ *filter*
translations
[*x*<−*xs* . *P*] ⇀ *CONST filter* (λ*x*. *P*) *xs*

end

primrec *fold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
fold-Nil: *fold f* [] = *id* |
fold-Cons: *fold f* (*x* # *xs*) = *fold f xs* ∘ *f x*

primrec *foldr* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'a list* ⇒ *'b* ⇒ *'b* **where**
foldr-Nil: *foldr f* [] = *id* |
foldr-Cons: *foldr f* (*x* # *xs*) = *f x* ∘ *foldr f xs*

primrec *foldl* :: (*'b* ⇒ *'a* ⇒ *'b*) ⇒ *'b* ⇒ *'a list* ⇒ *'b* **where**
foldl-Nil: *foldl f a* [] = *a* |
foldl-Cons: *foldl f a* (*x* # *xs*) = *foldl f* (*f a x*) *xs*

primrec *concat*:: *'a list list* ⇒ *'a list* **where**
concat [] = [] |
concat (*x* # *xs*) = *x* @ *concat xs*

primrec *drop*:: *nat* ⇒ *'a list* ⇒ *'a list* **where**
drop-Nil: *drop n* [] = [] |

drop-Cons: *drop n* (*x # xs*) = (*case n of 0 ⇒ x # xs | Suc m ⇒ drop m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec *take*:: *nat ⇒ 'a list ⇒ 'a list* **where**
take-Nil:*take n* [] = [] |
take-Cons: *take n* (*x # xs*) = (*case n of 0 ⇒* [] *| Suc m ⇒ x # take m xs*)
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec (*nonexhaustive*) *nth* :: *'a list => nat => 'a* (**infixl** ‹!› *100*) **where**
nth-Cons: (*x # xs*) ! *n* = (*case n of 0 ⇒ x | Suc k ⇒ xs ! k*)
— Warning: simpset does not contain this definition, but separate theorems for *n = 0* and *n = Suc k*

primrec *list-update* :: *'a list ⇒ nat ⇒ 'a ⇒ 'a list* **where**
list-update [] *i v* = [] |
list-update (*x # xs*) *i v* =
(*case i of 0 ⇒ v # xs | Suc j ⇒ x # list-update xs j v*)

nonterminal *lupdbinds* **and** *lupdbind*

open-bundle *list-update-syntax*
begin

syntax
-lupdbind:: ['*a*, '*a*] => *lupdbind* (‹(‹*indent=2 notation=‹mixfix update›*›- :=/ -)›)
:: *lupdbind => lupdbinds* (‹-›)
-lupdbinds :: [*lupdbind*, *lupdbinds*] => *lupdbinds* (‹-,/ -›)
-LUpdate :: ['*a*, *lupdbinds*] => '*a*
(‹(‹*open-block notation=‹mixfix list update›*›-/[(-)])› [*1000*,*0*] *900*)
syntax-consts
-LUpdate ⇌ *list-update*
translations
-LUpdate xs (*-lupdbinds b bs*) == *-LUpdate* (*-LUpdate xs b*) *bs*
xs[*i*:=*x*] == *CONST list-update xs i x*

end

primrec *takeWhile* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* **where**
takeWhile P [] = [] |
takeWhile P (*x # xs*) = (*if P x then x # takeWhile P xs else* [])

primrec *dropWhile* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* **where**
dropWhile P [] = [] |
dropWhile P (*x # xs*) = (*if P x then dropWhile P xs else x # xs*)

primrec *zip* :: *'a list ⇒ 'b list ⇒* ('*a* × '*b*) *list* **where**

zip xs [] = [] |
zip-Cons: *zip xs* (*y* # *ys*) =
(*case xs of* [] ⇒ [] | *z* # *zs* ⇒ (*z*, *y*) # *zip zs ys*)
— Warning: simpset does not contain this definition, but separate theorems for *xs* = [] and *xs* = *z* # *zs*

abbreviation *map2* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list* **where**
map2 f xs ys ≡ *map* (λ(*x*,*y*). *f x y*) (*zip xs ys*)

primrec *product* :: '*a list* ⇒ '*b list* ⇒ ('*a* × '*b*) *list* **where**
product [] - = [] |
product (*x*#*xs*) *ys* = *map* (*Pair x*) *ys* @ *product xs ys*

hide-const (**open**) *product*

primrec *product-lists* :: '*a list list* ⇒ '*a list list* **where**
product-lists [] = [[]] |
product-lists (*xs* # *xss*) = *concat* (*map* (λ*x*. *map* (*Cons x*) (*product-lists xss*)) *xs*)

primrec *upt* :: *nat* ⇒ *nat* ⇒ *nat list* (‹(‹*indent=1 notation=‹mixfix list interval*››[-..</-']›) **where**
upt-0: [*i*..<*0*] = [] |
upt-Suc: [*i*..<(*Suc j*)] = (*if i* ≤ *j then* [*i*..<*j*] @ [*j*] *else* [])

definition *insert* :: '*a* ⇒ '*a list* ⇒ '*a list* **where**
insert x xs = (*if x* ∈ *set xs then xs else x* # *xs*)

definition *union* :: '*a list* ⇒ '*a list* ⇒ '*a list* **where**
union = *fold insert*

hide-const (**open**) *insert union*
hide-fact (**open**) *insert-def union-def*

primrec *find* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a option* **where**
find - [] = *None* |
find P (*x*#*xs*) = (*if P x then Some x else find P xs*)

In the context of multisets, *count-list* is equivalent to *count* ∘ *mset* and it is advisable to use the latter.

primrec *count-list* :: '*a list* ⇒ '*a* ⇒ *nat* **where**
count-list [] *y* = *0* |
count-list (*x*#*xs*) *y* = (*if x*=*y then count-list xs y* + *1 else count-list xs y*)

definition
extract :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ ('*a list* ∗ '*a* ∗ '*a list*) *option*
where *extract P xs* =
(*case dropWhile* (*Not* ∘ *P*) *xs of*
[] ⇒ *None* |
y#*ys* ⇒ *Some*(*takeWhile* (*Not* ∘ *P*) *xs*, *y*, *ys*))

hide-const (**open**) *extract*

primrec *those* :: *'a option list* ⇒ *'a list option*
where
those [] = *Some* [] |
those (*x* # *xs*) = (*case x of*
  None ⇒ *None*
| *Some y* ⇒ *map-option* (*Cons y*) (*those xs*))

primrec *remove1* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
remove1 x [] = [] |
remove1 x (*y* # *xs*) = (*if x* = *y then xs else y* # *remove1 x xs*)

primrec *removeAll* :: *'a* ⇒ *'a list* ⇒ *'a list* **where**
removeAll x [] = [] |
removeAll x (*y* # *xs*) = (*if x* = *y then removeAll x xs else y* # *removeAll x xs*)

definition *minus-list-mset* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
minus-list-mset xs ys = *foldr remove1 ys xs*

definition *minus-list-set* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
minus-list-set xs ys = *foldr removeAll ys xs*

definition *inter-list-set* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
inter-list-set xs ys = *filter* (λ*x*. *x* ∈ *set ys*) *xs*

primrec *distinct* :: *'a list* ⇒ *bool* **where**
distinct [] ⟷ *True* |
distinct (*x* # *xs*) ⟷ *x* ∉ *set xs* ∧ *distinct xs*

fun *successively* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
successively P [] = *True* |
successively P [*x*] = *True* |
successively P (*x* # *y* # *xs*) = (*P x y* ∧ *successively P* (*y*#*xs*))

definition *distinct-adj* **where**
distinct-adj = *successively* (≠)

primrec *remdups* :: *'a list* ⇒ *'a list* **where**
remdups [] = [] |
remdups (*x* # *xs*) = (*if x* ∈ *set xs then remdups xs else x* # *remdups xs*)

fun *remdups-adj* :: *'a list* ⇒ *'a list* **where**
remdups-adj [] = [] |
remdups-adj [*x*] = [*x*] |
remdups-adj (*x* # *y* # *xs*) = (*if x* = *y then remdups-adj* (*x* # *xs*) *else x* #
remdups-adj (*y* # *xs*))

primrec *replicate* :: *nat* ⇒ *'a* ⇒ *'a list* **where**
replicate-0: *replicate 0 x* = [] |
replicate-Suc: *replicate* (*Suc n*) *x* = *x* # *replicate n x*

Function *size* is overloaded for all datatypes. Users may refer to the list version as *length*.

abbreviation *length* :: *'a list* ⇒ *nat* **where**
length ≡ *size*

definition *indexed-from* :: *nat* ⇒ *'a list* ⇒ (*nat* × *'a*) *list* **where**
indexed-from-eq-zip: *indexed-from n xs* = *zip* [*n*..<*n* + *length xs*] *xs*

primrec *rotate1* :: *'a list* ⇒ *'a list* **where**
rotate1 [] = [] |
rotate1 (*x* # *xs*) = *xs* @ [*x*]

definition *rotate* :: *nat* ⇒ *'a list* ⇒ *'a list* **where**
rotate n = *rotate1* ^^ *n*

definition *nths* :: *'a list* => *nat set* => *'a list* **where**
nths xs A = *map fst* (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*0*..<*size xs*]))

primrec *subseqs* :: *'a list* ⇒ *'a list list* **where**
subseqs [] = [[]] |
subseqs (*x*#*xs*) = (*let xss* = *subseqs xs in map* (*Cons x*) *xss* @ *xss*)

primrec *n-lists* :: *nat* ⇒ *'a list* ⇒ *'a list list* **where**
n-lists 0 xs = [[]] |
n-lists (*Suc n*) *xs* = *concat* (*map* (λ*ys*. *map* (λ*y*. *y* # *ys*) *xs*) (*n-lists n xs*))

hide-const (**open**) *n-lists*

function *splice* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
splice [] *ys* = *ys* |
splice (*x*#*xs*) *ys* = *x* # *splice ys xs*
by *pat-completeness auto*

termination
by(*relation measure*(λ(*xs*,*ys*). *size xs* + *size ys*)) *auto*

function *shuffles* **where**
  shuffles [] *ys* = {*ys*}
| *shuffles xs* [] = {*xs*}
| *shuffles* (*x* # *xs*) (*y* # *ys*) = (#) *x* ‘ *shuffles xs* (*y* # *ys*) ∪ (#) *y* ‘ *shuffles* (*x* # *xs*) *ys*
  by *pat-completeness simp-all*
termination by *lexicographic-order*

Use only if you cannot use *Min* instead:

fun *min-list* :: *'a::ord list* ⇒ *'a* **where**
min-list (*x* # *xs*) = (*case xs of* [] ⇒ *x* | - ⇒ *min x* (*min-list xs*))

Returns first minimum:

fun *arg-min-list* :: (*'a* ⇒ (*'b::linorder*)) ⇒ *'a list* ⇒ *'a* **where**
arg-min-list f [*x*] = *x* |
arg-min-list f (*x*#*y*#*zs*) = (*let m* = *arg-min-list f* (*y*#*zs*) *in if f x* ≤ *f m then x else m*)

Figure 1 shows characteristic examples that should give an intuitive understanding of the above functions.

The following simple sort(ed) functions are intended for proofs, not for efficient implementations.

A sorted predicate w.r.t. a relation:

fun *sorted-wrt* :: (*'a* ⇒ *'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
sorted-wrt P [] = *True* |
sorted-wrt P (*x* # *ys*) = ((∀ *y* ∈ *set ys*. *P x y*) ∧ *sorted-wrt P ys*)

A class-based sorted predicate:

context *linorder*
begin

abbreviation *sorted* :: *'a list* ⇒ *bool* **where**
sorted ≡ *sorted-wrt* (≤)

lemma *sorted-simps*: *sorted* [] = *True sorted* (*x* # *ys*) = ((∀ *y* ∈ *set ys*. *x*≤*y*) ∧ *sorted ys*)
by *auto*

lemma *strict-sorted-simps*: *sorted-wrt* (<) [] = *True sorted-wrt* (<) (*x* # *ys*) = ((∀ *y* ∈ *set ys*. *x*<*y*) ∧ *sorted-wrt* (<) *ys*)
by *auto*

primrec *insort-key* :: (*'b* ⇒ *'a*) ⇒ *'b* ⇒ *'b list* ⇒ *'b list* **where**
insort-key f x [] = [*x*] |
insort-key f x (*y*#*ys*) =
(*if f x* ≤ *f y then* (*x*#*y*#*ys*) *else y*#(*insort-key f x ys*))

definition *sort-key* :: (*'b* ⇒ *'a*) ⇒ *'b list* ⇒ *'b list* **where**
sort-key f xs = *foldr* (*insort-key f*) *xs* []

definition *insort-insert-key* :: (*'b* ⇒ *'a*) ⇒ *'b* ⇒ *'b list* ⇒ *'b list* **where**
insort-insert-key f x xs =
(*if f x* ∈ *f* ‘ *set xs then xs else insort-key f x xs*)

abbreviation *sort* ≡ *sort-key* (λ*x*. *x*)
abbreviation *insort* ≡ *insort-key* (λ*x*. *x*)

$[a, b] @ [c, d] = [a, b, c, d]$
length $[a, b, c] = 3$
set $[a, b, c] = \{a, b, c\}$
map f $[a, b, c] = [f\ a, f\ b, f\ c]$
rev $[a, b, c] = [c, b, a]$
hd $[a, b, c, d] = a$
tl $[a, b, c, d] = [b, c, d]$
last $[a, b, c, d] = d$
butlast $[a, b, c, d] = [a, b, c]$
filter $(\lambda n{::}nat.\ n{<}2)\ [0,2,1] = [0,1]$
concat $[[a, b], [c, d, e], [], [f]] = [a, b, c, d, e, f]$
fold f $[a, b, c]\ x = f\ c\ (f\ b\ (f\ a\ x))$
foldr f $[a, b, c]\ x = f\ a\ (f\ b\ (f\ c\ x))$
foldl f x $[a, b, c] = f\ (f\ (f\ x\ a)\ b)\ c$
successively $(\neq)$ $[True, False, True, False]$
zip $[a, b, c]\ [x, y, z] = [(a, x), (b, y), (c, z)]$
zip $[a, b]\ [x, y, z] = [(a, x), (b, y)]$
indexed-from 3 $[a, b, c] = [(3, a), (4, b), (5, c)]$
List.product $[a, b]\ [c, d] = [(a, c), (a, d), (b, c), (b, d)]$
product-lists $[[a, b], [c], [d, e]] = [[a, c, d], [a, c, e], [b, c, d], [b, c, e]]$
splice $[a, b, c]\ [x, y, z] = [a, x, b, y, c, z]$
splice $[a, b, c, d]\ [x, y] = [a, x, b, y, c, d]$
shuffles $[a, b]\ [c, d] = \{[a, b, c, d], [a, c, b, d], [a, c, d, b], [c, a, b, d], [c, a, d, b], [c, d, a, b]\}$
take 2 $[a, b, c, d] = [a, b]$
take 6 $[a, b, c, d] = [a, b, c, d]$
drop 2 $[a, b, c, d] = [c, d]$
drop 6 $[a, b, c, d] = []$
takeWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [1, 2]$
dropWhile $(\lambda n.\ n < 3)\ [1, 2, 3, 0] = [3, 0]$
distinct $[2, 0, 1]$
remdups $[2, 0, 2, 1, 2] = [0, 1, 2]$
remdups-adj $[2, 2, 3, 1, 1, 2, 1] = [2, 3, 1, 2, 1]$
List.insert 2 $[0, 1, 2] = [0, 1, 2]$
List.insert 3 $[0, 1, 2] = [3, 0, 1, 2]$
List.union $[2, 3, 4]\ [0, 1, 2] = [4, 3, 0, 1, 2]$
find $((<)\ 0)\ [0, 0] = None$
find $((<)\ 0)\ [0, 1, 0, 2] = Some\ 1$
count-list $[0, 1, 0, 2]\ 0 = 2$
List.extract $((<)\ 0)\ [0, 0] = None$
List.extract $((<)\ 0)\ [0, 1, 0, 2] = Some\ ([0], 1, [0, 2])$
remove1 2 $[2, 0, 2, 1, 2] = [0, 2, 1, 2]$
removeAll 2 $[2, 0, 2, 1, 2] = [0, 1]$
$[a, b, c, d]\ !\ 2 = c$
$[a, b, c, d][2 := x] = [a, b, x, d]$
nths $[a, b, c, d, e]\ \{0, 2, 3\} = [a, c, d]$
subseqs $[a, b] = [[a, b], [a], [b], []]$
List.n-lists 2 $[a, b, c] = [[a, a], [b, a], [c, a], [a, b], [b, b], [c, b], [a, c], [b, c], [c, c]]$
rotate1 $[a, b, c, d] = [b, c, d, a]$
rotate 3 $[a, b, c, d] = [d, a, b, c]$
replicate 4 a $= [a, a, a, a]$
$[2..{<}5] = [2, 3, 4]$
min-list $[3, 1, -2] = -2$
arg-min-list $(\lambda i.\ i * i)\ [3, -1, 1, -2] = -1$

abbreviation *insort-insert* ≡ *insort-insert-key* (λ*x*. *x*)

definition *stable-sort-key* :: ((′*b* ⇒ ′*a*) ⇒ ′*b list* ⇒ ′*b list*) ⇒ *bool* **where**
stable-sort-key sk =
(∀ *f xs k*. *filter* (λ*y*. *f y* = *k*) (*sk f xs*) = *filter* (λ*y*. *f y* = *k*) *xs*)

lemma *strict-sorted-iff*: *sorted-wrt* (<) *l* ⟷ *sorted l* ∧ *distinct l*
by (*induction l*) (*auto iff*: *antisym-conv1*)

lemma *strict-sorted-imp-sorted*: *sorted-wrt* (<) *xs* ⟹ *sorted xs*
by (*auto simp*: *strict-sorted-iff*)

end

66.1.1 List homomorphism

definition *hom-list* :: (′*a list* ⇒ ′*b list*) ⇒ *bool* **where**
hom-list h = (∀ *xs ys*. *h* (*xs* @ *ys*) = *h xs* @ *h ys*)

66.1.2 List comprehension

Input syntax for Haskell-like list comprehension notation. Typical example: [(*x*,*y*). *x* ← *xs*, *y* ← *ys*, *x* ≠ *y*], the list of all pairs of distinct elements from *xs* and *ys*. The syntax is as in Haskell, except that | becomes a dot (like in Isabelle's set comprehension): [*e*. *x* ← *xs*, ...] rather than `[e| x <- xs, ...]`.

The qualifiers after the dot are

generators *p* ← *xs*, where *p* is a pattern and *xs* an expression of list type,
or

guards *b*, where *b* is a boolean expression.

Just like in Haskell, list comprehension is just a shorthand. To avoid misunderstandings, the translation into desugared form is not reversed upon output. Note that the translation of [*e*. *x* ← *xs*] is optmized to *map* (λ*x*. *e*) *xs*.

It is easy to write short list comprehensions which stand for complex expressions. During proofs, they may become unreadable (and mangled). In such cases it can be advisable to introduce separate definitions for the list comprehensions in question.

nonterminal *lc-qual* **and** *lc-quals*

open-bundle *list-comprehension-syntax*
begin
syntax
-listcompr :: ′*a* ⇒ *lc-qual* ⇒ *lc-quals* ⇒ ′*a list* (‹[- . --›)

```
  -lc-gen :: 'a ⇒ 'a list ⇒ lc-qual  (‹- ← ->)
  -lc-test :: bool ⇒ lc-qual (‹->)

  -lc-end :: lc-quals (‹]›)
  -lc-quals :: lc-qual ⇒ lc-quals ⇒ lc-quals  (‹, -->)

syntax (ASCII)
  -lc-gen :: 'a ⇒ 'a list ⇒ lc-qual  (‹- <- ->)
end

parse-translation ‹
let
  val NilC = Syntax.const const-syntax‹Nil›;
  val ConsC = Syntax.const const-syntax‹Cons›;
  val mapC = Syntax.const const-syntax‹map›;
  val concatC = Syntax.const const-syntax‹concat›;
  val IfC = Syntax.const const-syntax‹If›;
  val dummyC = Syntax.const const-syntax‹Pure.dummy-pattern›

  fun single x = ConsC $ x $ NilC;

  fun pat-tr ctxt p e opti = (* %x. case x of p => e | - => [] *)
    let
      (* FIXME proper name context!? *)
      val x =
        Free (singleton (Name.variant-list (fold Term.add-free-names [p, e] [])) x,
dummyT);
      val e = if opti then single e else e;
      val case1 = Syntax.const syntax-const‹-case1› $ p $ e;
      val case2 =
        Syntax.const syntax-const‹-case1› $ dummyC $ NilC;
      val cs = Syntax.const syntax-const‹-case2› $ case1 $ case2;
    in Syntax-Trans.abs-tr [x, Case-Translation.case-tr false ctxt [x, cs]] end;

  fun pair-pat-tr (x as Free -) e = Syntax-Trans.abs-tr [x, e]
    | pair-pat-tr (- $ p1 $ p2) e =
        Syntax.const const-syntax‹case-prod› $ pair-pat-tr p1 (pair-pat-tr p2 e)
    | pair-pat-tr dummy e = Syntax-Trans.abs-tr [Syntax.const -idtdummy, e]

  fun pair-pat ctxt (Const (const-syntax‹Pair›,-) $ s $ t) =
        pair-pat ctxt s andalso pair-pat ctxt t
    | pair-pat ctxt (Free (s,-)) =
        let
          val thy = Proof-Context.theory-of ctxt;
          val s' = Proof-Context.intern-const ctxt s;
        in not (Sign.declared-const thy s') end
    | pair-pat - t = (t = dummyC);

  fun abs-tr ctxt p e opti =
```

```
    let val p = Term-Position.strip-positions p
    in if pair-pat ctxt p
       then (pair-pat-tr p e, true)
       else (pat-tr ctxt p e opti, false)
    end

  fun lc-tr ctxt [e, Const (syntax-const ‹-lc-test›, -) $ b, qs] =
    let
      val res =
        (case qs of
           Const (syntax-const ‹-lc-end›, -) => single e
         | Const (syntax-const ‹-lc-quals›, -) $ q $ qs => lc-tr ctxt [e, q, qs]);
    in IfC $ b $ res $ NilC end
  | lc-tr ctxt
      [e, Const (syntax-const ‹-lc-gen›, -) $ p $ es,
          Const(syntax-const ‹-lc-end›, -)] =
      (case abs-tr ctxt p e true of
         (f, true) => mapC $ f $ es
       | (f, false) => concatC $ (mapC $ f $ es))
  | lc-tr ctxt
      [e, Const (syntax-const ‹-lc-gen›, -) $ p $ es,
          Const (syntax-const ‹-lc-quals›, -) $ q $ qs] =
      let val e' = lc-tr ctxt [e, q, qs];
      in concatC $ (mapC $ (fst (abs-tr ctxt p e' false)) $ es) end;

in [(syntax-const ‹-listcompr›, lc-tr)] end
›

ML-val ‹
  let
    val read = Syntax.read-term context o Syntax.implode-input;
    fun check s1 s2 =
      read s1 aconv read s2 orelse
        error (Check failed: ^
          quote (#1 (Input.source-content s1)) ^ Position.here-list [Input.pos-of s1,
Input.pos-of s2]);
  in
    check ‹[(x,y,z). b]› ‹if b then [(x, y, z)] else []›;
    check ‹[(x,y,z). (x,-,y)←xs]› ‹map (λ(x,-,y). (x, y, z)) xs›;
    check ‹[e x y. (x,-)←xs, y←ys]› ‹concat (map (λ(x,-). map (λy. e x y) ys) xs)›;
    check ‹[(x,y,z). x<a, x>b]› ‹if x < a then if b < x then [(x, y, z)] else [] else
[]›;
    check ‹[(x,y,z). x←xs, x>b]› ‹concat (map (λx. if b < x then [(x, y, z)] else [])
xs)›;
    check ‹[(x,y,z). x<a, x←xs]› ‹if x < a then map (λx. (x, y, z)) xs else []›;
    check ‹[(x,y). Cons True x ← xs]›
      ‹concat (map (λxa. case xa of [] ⇒ [] | True # x ⇒ [(x, y)] | False # x ⇒ [])
xs)›;
    check ‹[(x,y,z). Cons x [] ← xs]›
```

```
    ‹concat (map (λxa. case xa of [] ⇒ [] | [x] ⇒ [(x, y, z)] | x # aa # lista ⇒
[]) xs)›;
    check ‹[(x,y,z). x<a, x>b, x=d]›
      ‹if x < a then if b < x then if x = d then [(x, y, z)] else [] else [] else []›;
    check ‹[(x,y,z). x<a, x>b, y←ys]›
      ‹if x < a then if b < x then map (λy. (x, y, z)) ys else [] else []›;
    check ‹[(x,y,z). x<a, (-,x)←xs,y>b]›
     ‹if x < a then concat (map (λ(-,x). if b < y then [(x, y, z)] else []) xs) else []›;
    check ‹[(x,y,z). x<a, x←xs, y←ys]›
      ‹if x < a then concat (map (λx. map (λy. (x, y, z)) ys) xs) else []›;
    check ‹[(x,y,z). x←xs, x>b, y<a]›
      ‹concat (map (λx. if b < x then if y < a then [(x, y, z)] else [] else []) xs)›;
    check ‹[(x,y,z). x←xs, x>b, y←ys]›
      ‹concat (map (λx. if b < x then map (λy. (x, y, z)) ys else []) xs)›;
    check ‹[(x,y,z). x←xs, (y,-)←ys,y>x]›
      ‹concat (map (λx. concat (map (λ(y,-). if x < y then [(x, y, z)] else []) ys))
xs)›;
    check ‹[(x,y,z). x←xs, y←ys,z←zs]›
      ‹concat (map (λx. concat (map (λy. map (λz. (x, y, z)) zs) ys)) xs)›
  end;
›

ML ‹
(* Simproc for rewriting list comprehensions applied to List.set to set
   comprehension. *)

signature LIST-TO-SET-COMPREHENSION =
sig
  val proc: Simplifier.proc
end

structure List-to-Set-Comprehension : LIST-TO-SET-COMPREHENSION =
struct

(* conversion *)

fun all-exists-conv cv ctxt ct =
  (case Thm.term-of ct of
    Const (const-name ‹Ex›, -) $ Abs - =>
      Conv.arg-conv (Conv.abs-conv (all-exists-conv cv o #2) ctxt) ct
  | - => cv ctxt ct)

fun all-but-last-exists-conv cv ctxt ct =
  (case Thm.term-of ct of
    Const (const-name ‹Ex›, -) $ Abs (-, -, Const (const-name ‹Ex›, -) $ -) =>
      Conv.arg-conv (Conv.abs-conv (all-but-last-exists-conv cv o #2) ctxt) ct
  | - => cv ctxt ct)

fun Collect-conv cv ctxt ct =
```

```
  (case Thm.term-of ct of
    Const (const-name‹Collect›, -) $ Abs - => Conv.arg-conv (Conv.abs-conv cv
ctxt) ct
  | - => raise CTERM (Collect-conv, [ct]))

fun rewr-conv' th = Conv.rewr-conv (mk-meta-eq th)

fun conjunct-assoc-conv ct =
  Conv.try-conv
   (rewr-conv' @{thm conj-assoc} then-conv HOLogic.conj-conv Conv.all-conv con-
junct-assoc-conv) ct

fun right-hand-set-comprehension-conv conv ctxt =
  HOLogic.Trueprop-conv (HOLogic.eq-conv Conv.all-conv
    (Collect-conv (all-exists-conv conv o #2) ctxt))


(* term abstraction of list comprehension patterns *)

datatype termlets = If | Case of typ * int

local

val set-Nil-I = @{lemma set [] = {x. False} by (simp add: empty-def [symmetric])}
val set-singleton = @{lemma set [a] = {x. x = a} by simp}
val inst-Collect-mem-eq = @{lemma set A = {x. x ∈ set A} by simp}
val del-refl-eq = @{lemma (t = t ∧ P) ≡ P by simp}

fun mk-set T = Const (const-name‹set›, HOLogic.listT T --> HOLogic.mk-setT
T)
fun dest-set (Const (const-name‹set›, -) $ xs) = xs

fun dest-singleton-list (Const (const-name‹Cons›, -) $ t $ (Const (const-name‹Nil›,
-))) = t
  | dest-singleton-list t = raise TERM (dest-singleton-list, [t])

(*We check that one case returns a singleton list and all other cases
  return [], and return the index of the one singleton list case.*)
fun possible-index-of-singleton-case cases =
  let
    fun check (i, case-t) s =
      (case strip-abs-body case-t of
        (Const (const-name‹Nil›, -)) => s
      | - => (case s of SOME NONE => SOME (SOME i) | - => NONE))
  in
    fold-index check cases (SOME NONE) |> the-default NONE
  end

(*returns condition continuing term option*)
```

```
fun dest-if (Const (const-name ‹If›, -) $ cond $ then-t $ Const (const-name ‹Nil›,
-)) =
      SOME (cond, then-t)
  | dest-if - = NONE

(*returns (case-expr type index chosen-case constr-name) option*)
fun dest-case ctxt case-term =
  let
    val (case-const, args) = strip-comb case-term
  in
    (case try dest-Const case-const of
      SOME (c, T) =>
        (case Ctr-Sugar.ctr-sugar-of-case ctxt c of
          SOME {ctrs, ...} =>
            (case possible-index-of-singleton-case (fst (split-last args)) of
              SOME i =>
                let
                  val constr-names = map dest-Const-name ctrs
                  val (Ts, -) = strip-type T
                  val T' = List.last Ts
                in SOME (List.last args, T', i, nth args i, nth constr-names i) end
            | NONE => NONE)
        | NONE => NONE)
    | NONE => NONE)
  end

fun tac ctxt [] =
      resolve-tac ctxt [set-singleton] 1 ORELSE
      resolve-tac ctxt [inst-Collect-mem-eq] 1
  | tac ctxt (If :: cont) =
      Splitter.split-tac ctxt @{thms if-split} 1
      THEN resolve-tac ctxt @{thms conjI} 1
      THEN resolve-tac ctxt @{thms impI} 1
      THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
        CONVERSION (right-hand-set-comprehension-conv (K
       (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm Eq-TrueI}))
Conv.all-conv
           then-conv
           rewr-conv' @{lemma (True ∧ P) = P by simp})) ctxt') 1) ctxt 1
      THEN tac ctxt cont
      THEN resolve-tac ctxt @{thms impI} 1
      THEN Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
          CONVERSION (right-hand-set-comprehension-conv (K
                (HOLogic.conj-conv (Conv.rewr-conv (List.last prems RS @{thm
Eq-FalseI})) Conv.all-conv
            then-conv rewr-conv' @{lemma (False ∧ P) = False by simp})) ctxt') 1)
ctxt 1
      THEN resolve-tac ctxt [set-Nil-I] 1
  | tac ctxt (Case (T, i) :: cont) =
```

```
    let
      val SOME {injects, distincts, case-thms, split, ...} =
        Ctr-Sugar.ctr-sugar-of ctxt (dest-Type-name T)
    in
      (* do case distinction *)
      Splitter.split-tac ctxt [split] 1
      THEN EVERY (map-index (fn (i', -) =>
        (if i' < length case-thms - 1 then resolve-tac ctxt @{thms conjI} 1 else
all-tac)
      THEN REPEAT-DETERM (resolve-tac ctxt @{thms allI} 1)
      THEN resolve-tac ctxt @{thms impI} 1
      THEN (if i' = i then
        (* continue recursively *)
        Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
      CONVERSION (Thm.eta-conversion then-conv right-hand-set-comprehension-conv
(K
            ((HOLogic.conj-conv
                (HOLogic.eq-conv Conv.all-conv (rewr-conv' (List.last prems))
then-conv
                  (Conv.try-conv (Conv.rewrs-conv (map mk-meta-eq injects))))
              Conv.all-conv)
              then-conv (Conv.try-conv (Conv.rewr-conv del-refl-eq))
              then-conv conjunct-assoc-conv)) ctxt'
          then-conv
            (HOLogic.Trueprop-conv
              (HOLogic.eq-conv Conv.all-conv (Collect-conv (fn (-, ctxt'') =>
                Conv.repeat-conv
                  (all-but-last-exists-conv
                    (K (rewr-conv'
                        @{lemma (∃x. x = t ∧ P x) = P t by simp})) ctxt''))
ctxt'))))) 1) ctxt 1
        THEN tac ctxt cont
      else
        Subgoal.FOCUS (fn {prems, context = ctxt', ...} =>
          CONVERSION
            (right-hand-set-comprehension-conv (K
              (HOLogic.conj-conv
                ((HOLogic.eq-conv Conv.all-conv
                  (rewr-conv' (List.last prems))) then-conv
                    (Conv.rewrs-conv (map (fn th => th RS @{thm Eq-FalseI})
distincts)))
                Conv.all-conv then-conv
                    (rewr-conv' @{lemma (False ∧ P) = False by simp}))) ctxt'
then-conv
              HOLogic.Trueprop-conv
                (HOLogic.eq-conv Conv.all-conv
                  (Collect-conv (fn (-, ctxt'') =>
                    Conv.repeat-conv
                      (Conv.bottom-conv
```

```
                                (K (rewr-conv' @{lemma (∃x. P) = P by simp})) ctxt''))
ctxt'))) 1) ctxt 1
          THEN resolve-tac ctxt [set-Nil-I] 1)) case-thms)
      end

in

fun proc ctxt redex =
  let
    fun make-inner-eqs bound-vs Tis eqs t =
      (case dest-case ctxt t of
        SOME (x, T, i, cont, constr-name) =>
          let
            val (vs, body) = strip-abs (Envir.eta-long (map snd bound-vs) cont)
            val x' = incr-boundvars (length vs) x
            val eqs' = map (incr-boundvars (length vs)) eqs
            val constr-t =
              list-comb
                 (Const (constr-name, map snd vs ---> T), map Bound (((length
vs) - 1) downto 0))
           val constr-eq = Const (const-name‹HOL.eq›, T --> T --> typ‹bool›)
$ constr-t $ x'
          in
           make-inner-eqs (rev vs @ bound-vs) (Case (T, i) :: Tis) (constr-eq :: eqs')
body
          end
      | NONE =>
          (case dest-if t of
          SOME (condition, cont) => make-inner-eqs bound-vs (If :: Tis) (condition
:: eqs) cont
          | NONE =>
            if null eqs then NONE (*no rewriting, nothing to be done*)
            else
              let
               val Type (type-name‹list›, [rT]) = fastype-of1 (map snd bound-vs, t)
                val pat-eq =
                  (case try dest-singleton-list t of
                    SOME t' =>
                      Const (const-name‹HOL.eq›, rT --> rT --> typ‹bool›) $
                        Bound (length bound-vs) $ t'
                  | NONE =>
                       Const (const-name‹Set.member›, rT --> HOLogic.mk-setT
rT --> typ‹bool›) $
                         Bound (length bound-vs) $ (mk-set rT $ t))
                val reverse-bounds = curry subst-bounds
                  ((map Bound ((length bound-vs - 1) downto 0)) @ [Bound (length
bound-vs)])
                val eqs' = map reverse-bounds eqs
                val pat-eq' = reverse-bounds pat-eq
```

```
            val inner-t =
            fold (fn (-, T) => fn t => HOLogic.exists-const T $ absdummy T t)
                (rev bound-vs) (fold (curry HOLogic.mk-conj) eqs' pat-eq')
            val lhs = Thm.term-of redex
            val rhs = HOLogic.mk-Collect (x, rT, inner-t)
          val rewrite-rule-t = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs))
            val eq-thm = Goal.norm-result ctxt (Goal.prove-internal ctxt []
              (Thm.cterm-of ctxt rewrite-rule-t) (fn - => tac ctxt (rev Tis)))
        in SOME (eq-thm RS @{thm eq-reflection}) end))
  in
    make-inner-eqs [] [] [] (dest-set (Thm.term-of redex))
  end

end

end
›
```

simproc-setup *list-to-set-comprehension* (*set xs*) =
  ‹*K List-to-Set-Comprehension.proc*›

code-datatype *set coset*
hide-const (**open**) *coset*

66.1.3 [] and (#)

lemma *not-Cons-self* [*simp*]:
  $xs \neq x \# xs$
by (*induct xs*) *auto*

lemma *not-Cons-self2* [*simp*]: $x \# xs \neq xs$
by (*rule not-Cons-self* [*symmetric*])

lemma *neq-Nil-conv*: $(xs \neq []) = (\exists y\ ys.\ xs = y \# ys)$
by (*induct xs*) *auto*

lemma *tl-Nil*: $tl\ xs = [] \longleftrightarrow xs = [] \vee (\exists x.\ xs = [x])$
by (*cases xs*) *auto*

lemmas *Nil-tl* = *tl-Nil*[*THEN eq-iff-swap*]

lemma *length-induct*:
  $(\bigwedge xs.\ \forall ys.\ length\ ys < length\ xs \longrightarrow P\ ys \Longrightarrow P\ xs) \Longrightarrow P\ xs$
by (*fact measure-induct*)

lemma *induct-list012*:
  $[\![P\ []; \bigwedge x.\ P\ [x]; \bigwedge x\ y\ zs.\ [\![\ P\ zs; P\ (y \# zs)\ ]\!] \Longrightarrow P\ (x \# y \# zs)]\!] \Longrightarrow P\ xs$
by *induction-schema* (*pat-completeness*, *lexicographic-order*)

lemma *list-nonempty-induct* [*consumes 1*, *case-names single cons*]:
⟦ *xs* ≠ []; ⋀*x*. *P* [*x*]; ⋀*x xs*. *xs* ≠ [] ⟹ *P xs* ⟹ *P* (*x* # *xs*)⟧ ⟹ *P xs*
by(*induction xs rule*: *induct-list012*) *auto*

lemma *inj-split-Cons*: *inj-on* (λ(*xs*, *n*). *n*#*xs*) *X*
by (*auto intro*!: *inj-onI*)

lemma *inj-on-Cons1* [*simp*]: *inj-on* ((#) *x*) *A*
by(*simp add*: *inj-on-def*)

66.1.4 *length*

Needs to come before @ because of theorem *append-eq-append-conv*.

lemma *length-append* [*simp*]: *length* (*xs* @ *ys*) = *length xs* + *length ys*
by (*induct xs*) *auto*

lemma *length-map* [*simp*]: *length* (*map f xs*) = *length xs*
by (*induct xs*) *auto*

lemma *length-rev* [*simp*]: *length* (*rev xs*) = *length xs*
by (*induct xs*) *auto*

lemma *length-tl* [*simp*]: *length* (*tl xs*) = *length xs* − *1*
by (*cases xs*) *auto*

lemma *length-0-conv* [*iff*]: (*length xs* = *0*) = (*xs* = [])
by (*induct xs*) *auto*

lemma *length-greater-0-conv* [*iff*]: (*0* < *length xs*) = (*xs* ≠ [])
by (*induct xs*) *auto*

lemma *length-pos-if-in-set*: *x* ∈ *set xs* ⟹ *length xs* > *0*
by *auto*

lemma *length-Suc-conv*: (*length xs* = *Suc n*) = (∃ *y ys*. *xs* = *y* # *ys* ∧ *length ys* = *n*)
by (*induct xs*) *auto*

lemmas *Suc-length-conv* = *length-Suc-conv*[*THEN eq-iff-swap*]

lemma *Suc-le-length-iff*:
(*Suc n* ≤ *length xs*) = (∃ *x ys*. *xs* = *x* # *ys* ∧ *n* ≤ *length ys*)
by (*metis Suc-le-D*[*of n*] *Suc-le-mono*[*of n*] *Suc-length-conv*[*of - xs*])

lemma *impossible-Cons*: *length xs* ≤ *length ys* ⟹ *xs* = *x* # *ys* = *False*
by (*induct xs*) *auto*

lemma *list-induct2* [*consumes 1*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *P* [] [] ⟹

(⋀*x xs y ys*. *length xs* = *length ys* ⟹ *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*))
⟹ *P xs ys*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed *simp*

lemma *list-induct3* [*consumes 2*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *length ys* = *length zs* ⟹ *P* [] [] [] ⟹
(⋀*x xs y ys z zs*. *length xs* = *length ys* ⟹ *length ys* = *length zs* ⟹ *P xs ys zs*
⟹ *P* (*x*#*xs*) (*y*#*ys*) (*z*#*zs*))
⟹ *P xs ys zs*
proof (*induct xs arbitrary*: *ys zs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys zs*) **then show** *?case* **by** (*cases ys*, *simp-all*)
(*cases zs*, *simp-all*)
qed

lemma *list-induct4* [*consumes 3*, *case-names Nil Cons*]:
length xs = *length ys* ⟹ *length ys* = *length zs* ⟹ *length zs* = *length ws* ⟹
P [] [] [] [] ⟹ (⋀*x xs y ys z zs w ws*. *length xs* = *length ys* ⟹
length ys = *length zs* ⟹ *length zs* = *length ws* ⟹ *P xs ys zs ws* ⟹
P (*x*#*xs*) (*y*#*ys*) (*z*#*zs*) (*w*#*ws*)) ⟹ *P xs ys zs ws*
proof (*induct xs arbitrary*: *ys zs ws*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs ys zs ws*) **then show** *?case* **by** ((*cases ys*, *simp-all*), (*cases zs*,*simp-all*)) (*cases ws*, *simp-all*)
qed

lemma *list-induct2′*:
⟦ *P* [] [];
⋀*x xs*. *P* (*x*#*xs*) [];
⋀*y ys*. *P* [] (*y*#*ys*);
⋀*x xs y ys*. *P xs ys* ⟹ *P* (*x*#*xs*) (*y*#*ys*) ⟧
⟹ *P xs ys*
by (*induct xs arbitrary*: *ys*) (*case-tac x*, *auto*)+

lemma *list-all2-iff*:
list-all2 P xs ys ⟷ *length xs* = *length ys* ∧ (∀(*x*, *y*) ∈ *set* (*zip xs ys*). *P x y*)
by (*induct xs ys rule*: *list-induct2′*) *auto*

lemma *neq-if-length-neq*: *length xs* ≠ *length ys* ⟹ (*xs* = *ys*) == *False*
by (*rule Eq-FalseI*) *auto*

66.1.5 @ − append

named-theorems *append-eq-iffs iffs for decomposing equations involving* (@) − *may create new vars*

global-interpretation *append*: *monoid append Nil*
proof
  fix *xs ys zs* :: *'a list*
  show (*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*)
    by (*induct xs*) *simp-all*
  show *xs* @ [] = *xs*
    by (*induct xs*) *simp-all*
qed *simp*

lemma *append-assoc* [*simp*]: (*xs* @ *ys*) @ *zs* = *xs* @ (*ys* @ *zs*)
  by (*fact append.assoc*)

lemma *append-Nil2*: *xs* @ [] = *xs*
  by (*fact append.right-neutral*)

lemma *append-is-Nil-conv* [*iff*]: (*xs* @ *ys* = []) = (*xs* = [] ∧ *ys* = [])
by (*induct xs*) *auto*

lemmas *Nil-is-append-conv* [*iff*] = *append-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *append-self-conv* [*iff*]: (*xs* @ *ys* = *xs*) = (*ys* = [])
by (*induct xs*) *auto*

lemmas *self-append-conv* [*iff*] = *append-self-conv*[*THEN eq-iff-swap*]

lemma *append-eq-append-conv* [*simp*]:
  length xs = *length ys* ∨ *length us* = *length vs*
  ⟹ (*xs*@*us* = *ys*@*vs*) = (*xs*=*ys* ∧ *us*=*vs*)
  by (*induct xs arbitrary*: *ys*; *case-tac ys*; *force*)

lemma *append-eq-append-conv2*[*append-eq-iffs*]: (*xs* @ *ys* = *zs* @ *ts*) =
  (∃ *us*. *xs* = *zs* @ *us* ∧ *us* @ *ys* = *ts* ∨ *xs* @ *us* = *zs* ∧ *ys* = *us* @ *ts*)
proof (*induct xs arbitrary*: *ys zs ts*)
  case (*Cons x xs*)
  then show *?case*
    by (*cases zs*) *auto*
qed *fastforce*

lemma *same-append-eq* [*iff*, *induct-simp*]: (*xs* @ *ys* = *xs* @ *zs*) = (*ys* = *zs*)
by *simp*

lemma *append1-eq-conv* [*iff*]: (*xs* @ [*x*] = *ys* @ [*y*]) = (*xs* = *ys* ∧ *x* = *y*)
by *simp*

lemma *append-same-eq* [*iff*, *induct-simp*]: (*ys* @ *xs* = *zs* @ *xs*) = (*ys* = *zs*)
by *simp*

lemma *append-self-conv2* [*iff*]: (*xs* @ *ys* = *ys*) = (*xs* = [])

using *append-same-eq* [*of* - - []] **by** *auto*

lemmas *self-append-conv2* [*iff*] = *append-self-conv2*[*THEN eq-iff-swap*]

lemma *hd-Cons-tl*: $xs \neq [] \Longrightarrow hd\ xs \# tl\ xs = xs$
by (*fact list.collapse*)

lemma *hd-append*: $hd\ (xs @ ys) = (if\ xs = []\ then\ hd\ ys\ else\ hd\ xs)$
by (*induct xs*) *auto*

lemma *hd-append2* [*simp*]: $xs \neq [] \Longrightarrow hd\ (xs @ ys) = hd\ xs$
by (*simp add*: *hd-append split*: *list.split*)

lemma *tl-append*: $tl\ (xs @ ys) = (case\ xs\ of\ [] \Rightarrow tl\ ys \mid z\#zs \Rightarrow zs @ ys)$
by (*simp split*: *list.split*)

lemma *tl-append2* [*simp*]: $xs \neq [] \Longrightarrow tl\ (xs @ ys) = tl\ xs @ ys$
by (*simp add*: *tl-append split*: *list.split*)

lemma *tl-append-if*: $tl\ (xs @ ys) = (if\ xs = []\ then\ tl\ ys\ else\ tl\ xs @ ys)$
by (*simp*)

lemma *Cons-eq-append-conv*[*append-eq-iffs*]: $x\#xs = ys@zs =$
$(ys = [] \wedge x\#xs = zs \vee (\exists ys'.\ x\#ys' = ys \wedge xs = ys'@zs))$
by(*cases ys*) *auto*

lemma *append-eq-Cons-conv*[*append-eq-iffs*]: $(ys@zs = x\#xs) =$
$(ys = [] \wedge zs = x\#xs \vee (\exists ys'.\ ys = x\#ys' \wedge ys'@zs = xs))$
by(*cases ys*) *auto*

lemma *longest-common-prefix*:
$\exists ps\ xs'\ ys'.\ xs = ps @ xs' \wedge ys = ps @ ys'$
$\wedge\ (xs' = [] \vee ys' = [] \vee hd\ xs' \neq hd\ ys')$
by (*induct xs ys rule*: *list-induct2'*)
(*blast*, *blast*, *blast*,
metis (*no-types*, *opaque-lifting*) *append-Cons append-Nil list.sel*(*1*))

Trivial rules for solving @-equations automatically.

lemma *eq-Nil-appendI*: $xs = ys \Longrightarrow xs = [] @ ys$
by *simp*

lemma *Cons-eq-appendI*: $[\![x \# xs1 = ys;\ xs = xs1 @ zs]\!] \Longrightarrow x \# xs = ys @ zs$
by *auto*

lemma *append-eq-appendI*: $[\![xs @ xs1 = zs;\ ys = xs1 @ us]\!] \Longrightarrow xs @ ys = zs @ us$
by *auto*

lemma *hom-list-Nil*: *hom-list h* $\Longrightarrow$ *h* [] = []
unfolding *hom-list-def* **by** (*metis self-append-conv*)

Simplification procedure for all list equalities. Currently only tries to rearrange @ to see if - both lists end in a singleton list, - or both lists end in the same list.

```
simproc-setup list-eq ((xs::'a list) = ys) = ‹
  let
    fun last (cons as Const (const-name‹Cons›, -) $ - $ xs) =
          (case xs of Const (const-name‹Nil›, -) => cons | - => last xs)
      | last (Const(const-name‹append›,-) $ - $ ys) = last ys
      | last t = t;

    fun list1 (Const(const-name‹Cons›,-) $ - $ Const(const-name‹Nil›,-)) =
true
      | list1 - = false;

    fun butlast ((cons as Const(const-name‹Cons›,-) $ x) $ xs) =
          (case xs of Const (const-name‹Nil›, -) => xs | - => cons $ butlast xs)
      | butlast ((app as Const (const-name‹append›, -) $ xs) $ ys) = app $ butlast
ys
      | butlast xs = Const(const-name‹Nil›, fastype-of xs);

    val rearr-ss =
      HOL-basic-ss
      |> Simplifier.simpset-map context
        (Simplifier.add-simps @{thms append-assoc append-Nil append-Cons});

    fun list-eq ctxt (F as (eq as Const(-,eqT)) $ lhs $ rhs) =
      let
        val lastl = last lhs and lastr = last rhs;
        fun rearr conv =
          let
            val lhs1 = butlast lhs and rhs1 = butlast rhs;
            val Type(-,listT::-) = eqT
            val appT = [listT,listT] ---> listT
            val app = Const(const-name‹append›,appT)
            val F2 = eq $ (app$lhs1$lastl) $ (app$rhs1$lastr)
            val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (F,F2));
            val thm = Goal.prove ctxt [] [] eq
              (K (simp-tac (put-simpset rearr-ss ctxt) 1));
          in SOME ((conv RS (thm RS trans)) RS eq-reflection) end;
      in
        if list1 lastl andalso list1 lastr then rearr @{thm append1-eq-conv}
        else if lastl aconv lastr then rearr @{thm append-same-eq}
        else NONE
      end;
  in K (fn ctxt => fn ct => list-eq ctxt (Thm.term-of ct)) end
›
```

66.1.6 *map*

lemma *hd-map*: $xs \neq [] \Longrightarrow hd\ (map\ f\ xs) = f\ (hd\ xs)$
by (*cases xs*) *simp-all*

lemma *map-tl*: $map\ f\ (tl\ xs) = tl\ (map\ f\ xs)$
by (*cases xs*) *simp-all*

lemma *map-ext*: $(\bigwedge x.\ x \in set\ xs \longrightarrow f\ x = g\ x) \Longrightarrow map\ f\ xs = map\ g\ xs$
by (*induct xs*) *simp-all*

lemma *map-ident* [*simp*]: $map\ (\lambda x.\ x) = (\lambda xs.\ xs)$
by (*rule ext*, *induct-tac xs*) *auto*

lemma *map-append* [*simp*]: $map\ f\ (xs\ @\ ys) = map\ f\ xs\ @\ map\ f\ ys$
by (*induct xs*) *auto*

lemma *map-map* [*simp*]: $map\ f\ (map\ g\ xs) = map\ (f \circ g)\ xs$
by (*induct xs*) *auto*

lemma *map-comp-map*[*simp*]: $((map\ f) \circ (map\ g)) = map(f \circ g)$
by (*rule ext*) *simp*

lemma *rev-map*: $rev\ (map\ f\ xs) = map\ f\ (rev\ xs)$
by (*induct xs*) *auto*

lemma *map-eq-conv*[*simp*]: $(map\ f\ xs = map\ g\ xs) = (\forall x \in set\ xs.\ f\ x = g\ x)$
by (*induct xs*) *auto*

lemma *map-cong* [*fundef-cong*]:
$xs = ys \Longrightarrow (\bigwedge x.\ x \in set\ ys \Longrightarrow f\ x = g\ x) \Longrightarrow map\ f\ xs = map\ g\ ys$
by *simp*

lemma *map-is-Nil-conv* [*iff*]: $(map\ f\ xs = []) = (xs = [])$
by (*rule list.map-disc-iff*)

lemmas *Nil-is-map-conv* [*iff*] = *map-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *map-eq-Cons-conv*:
$(map\ f\ xs = y\#ys) = (\exists z\ zs.\ xs = z\#zs \wedge f\ z = y \wedge map\ f\ zs = ys)$
by (*cases xs*) *auto*

lemma *Cons-eq-map-conv*:
$(x\#xs = map\ f\ ys) = (\exists z\ zs.\ ys = z\#zs \wedge x = f\ z \wedge xs = map\ f\ zs)$
by (*cases ys*) *auto*

lemmas *map-eq-Cons-D* = *map-eq-Cons-conv* [*THEN iffD1*]
lemmas *Cons-eq-map-D* = *Cons-eq-map-conv* [*THEN iffD1*]
declare *map-eq-Cons-D* [*dest*!] *Cons-eq-map-D* [*dest*!]

lemma *ex-map-conv*:
(∃ *xs*. *ys* = *map f xs*) = (∀ *y* ∈ *set ys*. ∃ *x*. *y* = *f x*)
by(*induct ys*, *auto simp add*: *Cons-eq-map-conv*)

lemma *map-eq-imp-length-eq*:
assumes *map f xs* = *map g ys*
shows *length xs* = *length ys*
using *assms*
proof (*induct ys arbitrary*: *xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons y ys*) **then obtain** *z zs* **where** *xs*: *xs* = *z* # *zs* **by** *auto*
from *Cons xs* **have** *map f zs* = *map g ys* **by** *simp*
with *Cons* **have** *length zs* = *length ys* **by** *blast*
with *xs* **show** *?case* **by** *simp*
qed

lemma *map-eq-append-conv*[*append-eq-iffs*]:
map f xs = *ys* @ *zs* ⟷ (∃ *us vs*. *xs* = *us* @ *vs* ∧ *ys* = *map f us* ∧ *zs* = *map f vs*)
proof(*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by**(*auto*)
next
case *IH*: (*Cons x xs*)
show *?case* (**is** *?L* = (∃ *us*. *?R us*))
proof
assume *?L*
show ∃ *us*. *?R us*
proof (*cases ys*)
case *Nil* **then show** *?thesis* **using** ‹*?L*› **by** *force*
next
case [*simp*]: (*Cons y ys′*)
with ‹*?L*› **have** [*simp*]: *y* = *f x* **and** ∗: *map f xs* = *ys′* @ *zs* **by** *simp*+
from *IH* ∗ **obtain** *us vs* **where** [*simp*]: *xs* = *us* @ *vs* ∧ *ys′* = *map f us* ∧ *zs* = *map f vs* **by** *blast*
then have *?R* (*x*#*us*) **by** *simp*
then show *?thesis* **by** *blast*
qed
next
assume ∃ *us*. *?R us*
then show *?L* **using** *IH map-eq-Cons-conv*[*of f x* # *xs f x map f xs*] **by** *force*
qed
qed

lemmas *append-eq-map-conv*[*append-eq-iffs*] = *map-eq-append-conv*[*THEN eq-iff-swap*]

lemma *map-inj-on*:
assumes *map*: *map f xs* = *map f ys* **and** *inj*: *inj-on f* (*set xs Un set ys*)

shows *xs = ys*
using *map-eq-imp-length-eq* [*OF map*] *assms*
proof (*induct rule*: *list-induct2*)
case (*Cons x xs y ys*)
then show *?case*
by (*auto intro*: *sym*)
qed *auto*

lemma *inj-on-map-eq-map*:
inj-on f (*set xs Un set ys*) ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-inj-on*)

lemma *map-injective*:
map f xs = *map f ys* ⟹ *inj f* ⟹ *xs* = *ys*
by (*induct ys arbitrary*: *xs*) (*auto dest*!:*injD*)

lemma *inj-map-eq-map*[*simp*]: *inj f* ⟹ (*map f xs* = *map f ys*) = (*xs* = *ys*)
by(*blast dest*:*map-injective*)

lemma *inj-mapI*: *inj f* ⟹ *inj* (*map f*)
by (*rule list.inj-map*)

lemma *inj-mapD*: *inj* (*map f*) ⟹ *inj f*
by (*metis* (*no-types*, *opaque-lifting*) *injI list.inject list.simps*(*9*) *the-inv-f-f*)

lemma *inj-map*[*iff*]: *inj* (*map f*) = *inj f*
by (*blast dest*: *inj-mapD intro*: *inj-mapI*)

lemma *inj-on-mapI*: *inj-on f* (⋃(*set* ' *A*)) ⟹ *inj-on* (*map f*) *A*
by (*blast intro*:*inj-onI dest*:*inj-onD map-inj-on*)

lemma *map-idI*: (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *x*) ⟹ *map f xs* = *xs*
by (*rule list.map-ident-strong*)

lemma *map-fun-upd* [*simp*]: *y* ∉ *set xs* ⟹ *map* (*f*(*y*:=*v*)) *xs* = *map f xs*
by (*induct xs*) *auto*

lemma *map-fst-zip*[*simp*]:
length xs = *length ys* ⟹ *map fst* (*zip xs ys*) = *xs*
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-snd-zip*[*simp*]:
length xs = *length ys* ⟹ *map snd* (*zip xs ys*) = *ys*
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *map-fst-zip-take*:
map fst (*zip xs ys*) = *take* (*min* (*length xs*) (*length ys*)) *xs*
by (*induct xs ys rule*: *list-induct2'*) *simp-all*

lemma *map-snd-zip-take*:
  map snd (*zip xs ys*) = *take* (*min* (*length xs*) (*length ys*)) *ys*
by (*induct xs ys rule*: *list-induct2'*) *simp-all*

lemma *map2-map-map*: *map2 h* (*map f xs*) (*map g xs*) = *map* (λ*x*. *h* (*f x*) (*g x*)) *xs*
by (*induction xs*) (*auto*)

functor *map*: *map*
by (*simp-all add*: *id-def*)

declare *map.id* [*simp*]

66.1.7 *rev*

lemma *rev-append* [*simp*]: *rev* (*xs* @ *ys*) = *rev ys* @ *rev xs*
by (*induct xs*) *auto*

lemma *rev-rev-ident* [*simp*]: *rev* (*rev xs*) = *xs*
by (*induct xs*) *auto*

lemma *rev-involution*[*simp*]: *rev* ∘ *rev* = *id*
by *auto*

lemma *rev-swap*: (*rev xs* = *ys*) = (*xs* = *rev ys*)
by *auto*

lemma *rev-is-Nil-conv* [*iff*]: (*rev xs* = []) = (*xs* = [])
by (*induct xs*) *auto*

lemmas *Nil-is-rev-conv* [*iff*] = *rev-is-Nil-conv*[*THEN eq-iff-swap*]

lemma *rev-singleton-conv* [*simp*]: (*rev xs* = [*x*]) = (*xs* = [*x*])
by (*cases xs*) *auto*

lemma *singleton-rev-conv* [*simp*]: ([*x*] = *rev xs*) = ([*x*] = *xs*)
by (*cases xs*) *auto*

lemma *rev-is-rev-conv* [*iff*]: (*rev xs* = *rev ys*) = (*xs* = *ys*)
proof (*induct xs arbitrary*: *ys*)
  case *Nil*
  then show *?case* **by** *force*
next
  case *Cons*
  then show *?case* **by** (*cases ys*) *auto*
qed

lemma *rev-eq-append-conv*: *rev xs* = *ys* @ *zs* ⟷ *xs* = *rev zs* @ *rev ys*
by (*metis rev-append rev-rev-ident*)

lemma *append-eq-rev-conv*: *ys* @ *zs* = *rev xs* ⟷ *rev zs* @ *rev ys* = *xs*
using *rev-eq-append-conv*[*THEN eq-iff-swap*] **by** *metis*

lemma *rev-eq-Cons-iff*[*iff*]: (*rev xs* = *y*#*ys*) = (*xs* = *rev ys* @ [*y*])
by (*simp add*: *rev-swap*)

lemmas *Cons-eq-rev-iff* = *rev-eq-Cons-iff*[*THEN eq-iff-swap*]

lemma *inj-on-rev*[*iff*]: *inj-on rev A*
by(*simp add*:*inj-on-def*)

lemma *rev-induct* [*case-names Nil snoc*]:
  assumes *P* [] **and** ⋀*x xs*. *P xs* ⟹ *P* (*xs* @ [*x*])
  shows *P xs*
proof −
  have *P* (*rev* (*rev xs*))
    by (*rule-tac list* = *rev xs* **in** *list.induct*, *simp-all add*: *assms*)
  then show *?thesis* **by** *simp*
qed

lemma *rev-exhaust* [*case-names Nil snoc*]:
  (*xs* = [] ⟹ *P*) ⟹(⋀*ys y*. *xs* = *ys* @ [*y*] ⟹ *P*) ⟹ *P*
by (*induct xs rule*: *rev-induct*) *auto*

lemmas *rev-cases* = *rev-exhaust*

lemma *rev-nonempty-induct* [*consumes 1*, *case-names single snoc*]:
  assumes *xs* ≠ []
  and *single*: ⋀*x*. *P* [*x*]
  and *snoc'*: ⋀*x xs*. *xs* ≠ [] ⟹ *P xs* ⟹ *P* (*xs*@[*x*])
  shows *P xs*
using ‹*xs* ≠ []› **proof** (*induct xs rule*: *rev-induct*)
  case (*snoc x xs*) **then show** *?case*
  proof (*cases xs*)
    case *Nil* **thus** *?thesis* **by** (*simp add*: *single*)
  next
    case *Cons* **with** *snoc* **show** *?thesis* **by** (*fastforce intro*!: *snoc'*)
  qed
qed *simp*

lemma *rev-induct2*:
  ⟦ *P* [] [];
  ⋀*x xs*. *P* (*xs* @ [*x*]) [];
  ⋀*y ys*. *P* [] (*ys* @ [*y*]);
  ⋀*x xs y ys*. *P xs ys* ⟹ *P* (*xs* @ [*x*]) (*ys* @ [*y*]) ⟧
 ⟹ *P xs ys*
proof (*induct xs arbitrary*: *ys rule*: *rev-induct*)
  case *Nil*

then show *?case* **using** *rev-induct*[*of P* []] **by** *presburger*
next
case (*snoc x xs*)
hence *P xs ys'* **for** *ys'* **by** *simp*
then show *?case* **by** (*simp add*: *rev-induct snoc.prems*(*2*,*4*))
qed

lemma *length-Suc-conv-rev*: (*length xs* = *Suc n*) = (∃ *y ys*. *xs* = *ys* @ [*y*] ∧ *length ys* = *n*)
by (*induct xs rule*: *rev-induct*) *auto*

66.1.8 *set*

declare *list.set*[*code-post*] — pretty output

lemma *finite-set* [*iff*]: *finite* (*set xs*)
by (*induct xs*) *auto*

lemma *set-append* [*simp*]: *set* (*xs* @ *ys*) = (*set xs* ∪ *set ys*)
by (*induct xs*) *auto*

lemma *hd-in-set*[*simp*]: *xs* ≠ [] ⟹ *hd xs* ∈ *set xs*
by(*cases xs*) *auto*

lemma *set-subset-Cons*: *set xs* ⊆ *set* (*x* # *xs*)
by *auto*

lemma *set-ConsD*: *y* ∈ *set* (*x* # *xs*) ⟹ *y*=*x* ∨ *y* ∈ *set xs*
by *auto*

lemma *set-empty* [*iff*]: (*set xs* = {}) = (*xs* = [])
by (*induct xs*) *auto*

lemmas *set-empty2*[*iff*] = *set-empty*[*THEN eq-iff-swap*]

lemma *append-eq-append-conv-if-disj*:
(*set xs* ∪ *set xs'*) ∩ (*set ys* ∪ *set ys'*) = {}
⟹ *xs* @ *ys* = *xs'* @ *ys'* ⟷ *xs* = *xs'* ∧ *ys* = *ys'*
by (*auto simp*: *all-conj-distrib disjoint-iff append-eq-append-conv2*)

lemma *set-rev* [*simp*]: *set* (*rev xs*) = *set xs*
by (*induct xs*) *auto*

lemma *set-map* [*simp*]: *set* (*map f xs*) = *f*‘(*set xs*)
by (*rule list.set-map*)

lemma *set-filter* [*simp*]: *set* (*filter P xs*) = {*x*. *x* ∈ *set xs* ∧ *P x*}
by (*induct xs*) *auto*

lemma *set-upt* [*simp*]: $set[i..<j] = \{i..<j\}$
by (*induct j*) *auto*

lemma *atMost-upto*:
‹$\{..n\} = set\ [0..<Suc\ n]$›
by *auto*

lemma *atLeast-upt*:
‹$\{..<n\} = set\ [0..<n]$›
by *auto*

lemma *greaterThanLessThan-upt*:
‹$\{n<..<m\} = set\ [Suc\ n..<m]$›
by *auto*

lemma *atLeastLessThan-upt*:
‹$\{i..<j\} = set\ [i..<j]$›
by *auto*

lemma *greaterThanAtMost-upt*:
$\{n<..m\} = set\ [Suc\ n..<Suc\ m]$
by *auto*

lemma *atLeastAtMost-upt*:
$\{n..m\} = set\ [n..<Suc\ m]$
by *auto*

lemma *split-list*: $x \in set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case* **by** (*auto intro*: *Cons-eq-appendI*)
qed

lemma *in-set-conv-decomp*: $x \in set\ xs \longleftrightarrow (\exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs)$
by (*auto elim*: *split-list*)

lemma *split-list-first*: $x \in set\ xs \Longrightarrow \exists\, ys\ zs.\ xs = ys\ @\ x\ \#\ zs \wedge x \notin set\ ys$
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
show *?case*
proof *cases*
assume $x = a$ **thus** *?case* **using** *Cons* **by** *fastforce*
next
assume $x \neq a$ **thus** *?case* **using** *Cons* **by**(*fastforce intro*!: *Cons-eq-appendI*)
qed
qed

lemma *in-set-conv-decomp-first*:
(*x* ∈ *set xs*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set ys*)
by (*auto dest*!: *split-list-first*)

lemma *split-list-last*: *x* ∈ *set xs* ⟹ ∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set zs*
proof (*induct xs rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*snoc a xs*)
show *?case*
proof *cases*
assume *x* = *a* **thus** *?case* **using** *snoc* **by** (*auto intro*!: *exI*)
next
assume *x* ≠ *a* **thus** *?case* **using** *snoc* **by** *fastforce*
qed
qed

lemma *in-set-conv-decomp-last*:
(*x* ∈ *set xs*) = (∃ *ys zs*. *xs* = *ys* @ *x* # *zs* ∧ *x* ∉ *set zs*)
by (*auto dest*!: *split-list-last*)

lemma *split-list-prop*: ∃ *x* ∈ *set xs*. *P x* ⟹ ∃ *ys x zs*. *xs* = *ys* @ *x* # *zs* ∧ *P x*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case *Cons* **thus** *?case*
by(*simp add*:*Bex-def*)(*metis append-Cons append.simps*(*1*))
qed

lemma *split-list-propE*:
assumes ∃ *x* ∈ *set xs*. *P x*
obtains *ys x zs* **where** *xs* = *ys* @ *x* # *zs* **and** *P x*
using *split-list-prop* [*OF assms*] **by** *blast*

lemma *split-list-first-prop*:
∃ *x* ∈ *set xs*. *P x* ⟹
∃ *ys x zs*. *xs* = *ys*@*x*#*zs* ∧ *P x* ∧ (∀ *y* ∈ *set ys*. ¬ *P y*)
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof *cases*
assume *P x*
hence *x* # *xs* = [] @ *x* # *xs* ∧ *P x* ∧ (∀ *y*∈*set* []. ¬ *P y*) **by** *simp*
thus *?thesis* **by** *fast*
next
assume ¬ *P x*

```
    hence ∃x∈set xs. P x using Cons(2) by simp
    thus ?thesis using ‹¬ P x› Cons(1) by (metis append-Cons set-ConsD)
  qed
qed

lemma split-list-first-propE:
  assumes ∃x ∈ set xs. P x
  obtains ys x zs where xs = ys @ x # zs and P x and ∀y ∈ set ys. ¬ P y
using split-list-first-prop [OF assms] by blast

lemma split-list-first-prop-iff:
  (∃x ∈ set xs. P x) ⟷
   (∃ys x zs. xs = ys@x#zs ∧ P x ∧ (∀y ∈ set ys. ¬ P y))
by (rule, erule split-list-first-prop) auto

lemma split-list-last-prop:
  ∃x ∈ set xs. P x ⟹
   ∃ys x zs. xs = ys@x#zs ∧ P x ∧ (∀z ∈ set zs. ¬ P z)
proof(induct xs rule:rev-induct)
  case Nil thus ?case by simp
next
  case (snoc x xs)
  show ?case
  proof cases
    assume P x thus ?thesis by (auto intro!: exI)
  next
    assume ¬ P x
    hence ∃x∈set xs. P x using snoc(2) by simp
    thus ?thesis using ‹¬ P x› snoc(1) by fastforce
  qed
qed

lemma split-list-last-propE:
  assumes ∃x ∈ set xs. P x
  obtains ys x zs where xs = ys @ x # zs and P x and ∀z ∈ set zs. ¬ P z
using split-list-last-prop [OF assms] by blast

lemma split-list-last-prop-iff:
  (∃x ∈ set xs. P x) ⟷
   (∃ys x zs. xs = ys@x#zs ∧ P x ∧ (∀z ∈ set zs. ¬ P z))
  by rule (erule split-list-last-prop, auto)


lemma finite-list: finite A ⟹ ∃xs. set xs = A
 by (erule finite-induct) (auto simp add: list.set(2)[symmetric] simp del: list.set(2))

lemma card-length: card (set xs) ≤ length xs
by (induct xs) (auto simp add: card-insert-if)
```

lemma *set-minus-filter-out*:
set xs − {y} = set (filter (λx. ¬ (x = y)) xs)
by (*induct xs*) *auto*

lemma *append-Cons-eq-iff*:
⟦ x ∉ set xs; x ∉ set ys ⟧ ⟹
xs @ x # ys = xs′ @ x # ys′ ⟷ (xs = xs′ ∧ ys = ys′)
by(*auto simp*: *append-eq-iffs*)

66.1.9 *concat*

lemma *concat-append* [*simp*]: *concat (xs @ ys) = concat xs @ concat ys*
by (*induct xs*) *auto*

lemma *concat-eq-Nil-conv* [*simp*]: *(concat xss = []) = (∀ xs ∈ set xss. xs = [])*
by (*induct xss*) *auto*

lemmas *Nil-eq-concat-conv* [*simp*] = *concat-eq-Nil-conv*[*THEN eq-iff-swap*]

lemma *set-concat* [*simp*]: *set (concat xs) = (⋃x∈set xs. set x)*
by (*induct xs*) *auto*

lemma *concat-map-singleton*[*simp*]: *concat*(*map (%x. [f x]) xs) = map f xs*
by (*induct xs*) *auto*

lemma *map-concat*: *map f (concat xs) = concat (map (map f) xs)*
by (*induct xs*) *auto*

lemma *rev-concat*: *rev (concat xs) = concat (map rev (rev xs))*
by (*induct xs*) *auto*

lemma *length-concat-rev*[*simp*]: *length (concat (rev xs)) = length (concat xs)*
by (*induction xs*) *auto*

lemma *concat-eq-concat-iff*: *∀ (x, y) ∈ set (zip xs ys). length x = length y ⟹ length xs = length ys ⟹ (concat xs = concat ys) = (xs = ys)*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*)
thus *?case* **by** (*cases ys*) *auto*
qed (*auto*)

lemma *concat-injective*: *concat xs = concat ys ⟹ length xs = length ys ⟹ ∀ (x, y) ∈ set (zip xs ys). length x = length y ⟹ xs = ys*
by (*simp add*: *concat-eq-concat-iff*)

lemma *concat-eq-appendD*:
assumes *concat xss = ys @ zs xss ≠ []*
shows *∃ xss1 xs xs′ xss2. xss = xss1 @ (xs @ xs′) # xss2 ∧ ys = concat xss1 @ xs ∧ zs = xs′ @ concat xss2*

```
  using assms
proof(induction xss arbitrary: ys)
  case (Cons xs xss)
  from Cons.prems consider
    us where xs @ us = ys concat xss = us @ zs |
    us where xs = ys @ us us @ concat xss = zs
    by(auto simp add: append-eq-append-conv2)
  then show ?case
  proof cases
    case 1
    then show ?thesis using Cons.IH[OF 1(2)]
      by(cases xss)(auto intro: exI[where x=[]], metis append.assoc append-Cons
concat.simps(2))
  qed(auto intro: exI[where x=[]])
qed simp

lemma concat-eq-append-conv:
  concat xss = ys @ zs ⟷
  (if xss = [] then ys = [] ∧ zs = []
   else ∃ xss1 xs xs' xss2. xss = xss1 @ (xs @ xs') # xss2 ∧ ys = concat xss1 @ xs
∧ zs = xs' @ concat xss2)
  by(auto dest: concat-eq-appendD)

lemma hd-concat: ⟦xs ≠ []; hd xs ≠ []⟧ ⟹ hd (concat xs) = hd (hd xs)
  by (metis concat.simps(2) hd-Cons-tl hd-append2)

lemma map-eq-concat-iff:
  map f xs = concat yss ⟷ (∃ xss. xs = concat xss ∧ yss = map (map f) xss)
proof(induction yss arbitrary: xs)
  case Nil
  then show ?case by auto
next
  case (Cons ys yss)
  show ?case (is ?L = (∃ xss. ?R xss))
  proof
    assume ?L
    then obtain xs1 xs2 where xs = xs1 @ xs2 and mapf12: map f xs1 = ys
map f xs2 = concat yss
      by (auto simp: map-eq-append-conv)
    with Cons.IH[of xs2] obtain xss where xs2 = concat xss yss = map (map f)
xss by blast
    then have ?R (xs1#xss) using ‹xs = -› ‹- = ys› by simp
    then show (∃ xss. ?R xss) ..
  next
    assume ∃ xss. ?R xss
    then show ?L by (auto simp add: map-concat)
  qed
qed
```

simproc-setup *list-neq* ((*xs*::'*a list*) = *ys*) = ‹
(*
Reduces xs=ys to False if xs and ys cannot be of the same length.
This is the case if the atomic sublists of one are a submultiset
of those of the other list and there are fewer Cons's in one than the other.
*)

let

fun len (*Const*(***const-name*** ‹*Nil*›,-)) *acc* = *acc*
| *len* (*Const*(***const-name*** ‹*Cons*›,-) $ - $ *xs*) (*ts*,*n*) = *len xs* (*ts*,*n*+*1*)
| *len* (*Const*(***const-name*** ‹*append*›,-) $ *xs* $ *ys*) *acc* = *len xs* (*len ys acc*)
| *len* (*Const*(***const-name*** ‹*rev*›,-) $ *xs*) *acc* = *len xs acc*
| *len* (*Const*(***const-name*** ‹*map*›,-) $ - $ *xs*) *acc* = *len xs acc*
| *len* (*Const*(***const-name*** ‹*concat*›,*T*) $ (*Const*(***const-name*** ‹*rev*›,-) $ *xss*)) *acc*
= *len* (*Const*(***const-name*** ‹*concat*›,*T*) $ *xss*) *acc*
| *len t* (*ts*,*n*) = (*t*::*ts*,*n*);

val ss = *Simplifier.simpset-of* ***context***;

fun list-neq ctxt ct =
let
val (*Const*(-,*eqT*) $ *lhs* $ *rhs*) = *Thm.term-of ct*;
val (*ls*,*m*) = *len lhs* ([],*0*) *and* (*rs*,*n*) = *len rhs* ([],*0*);
fun prove-neq() =
let
val Type(-,*listT*::-) = *eqT*;
val size = *HOLogic.size-const listT*;
val eq-len = *HOLogic.mk-eq* (*size* $ *lhs*, *size* $ *rhs*);
val neq-len = *HOLogic.mk-Trueprop* (*HOLogic.Not* $ *eq-len*);
val thm = *Goal.prove ctxt* [] [] *neq-len*
(*K* (*simp-tac* (*put-simpset ss ctxt*) *1*));
in SOME (*thm RS* @{*thm neq-if-length-neq*}) *end*
in
if m < *n andalso submultiset* (*op aconv*) (*ls*,*rs*) *orelse*
n < *m andalso submultiset* (*op aconv*) (*rs*,*ls*)
then prove-neq() *else NONE*
end;
in K list-neq end
›

66.1.10 *filter*

lemma *filter-append* [*simp*]: *filter P* (*xs* @ *ys*) = *filter P xs* @ *filter P ys*
by (*induct xs*) *auto*

lemma *rev-filter*: *rev* (*filter P xs*) = *filter P* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *filter-filter* [*simp*]: *filter P* (*filter Q xs*) = *filter* (λ*x*. *Q x* ∧ *P x*) *xs*
by (*induct xs*) *auto*

lemma *filter-concat*: *filter p* (*concat xs*) = *concat* (*map* (*filter p*) *xs*)
by (*induct xs*) *auto*

lemma *length-filter-le* [*simp*]: *length* (*filter P xs*) ≤ *length xs*
by (*induct xs*) (*auto simp add*: *le-SucI*)

lemma *sum-length-filter-compl*:
length(*filter P xs*) + *length*(*filter* (λ*x*. ¬*P x*) *xs*) = *length xs*
by(*induct xs*) *simp-all*

lemma *filter-True* [*simp*]: ∀ *x* ∈ *set xs*. *P x* ⟹ *filter P xs* = *xs*
by (*induct xs*) *auto*

lemma *filter-False* [*simp*]: ∀ *x* ∈ *set xs*. ¬ *P x* ⟹ *filter P xs* = []
by (*induct xs*) *auto*

lemma *filter-empty-conv*: (*filter P xs* = []) = (∀ *x*∈*set xs*. ¬ *P x*)
by (*induct xs*) *simp-all*

lemmas *empty-filter-conv* = *filter-empty-conv*[*THEN eq-iff-swap*]

lemma *filter-id-conv*: (*filter P xs* = *xs*) = (∀ *x*∈*set xs*. *P x*)
proof (*induct xs*)
case (*Cons x xs*)
then show *?case*
using *length-filter-le*
by (*simp add*: *impossible-Cons*)
qed *auto*

lemma *filter-map*: *filter P* (*map f xs*) = *map f* (*filter* (*P* ∘ *f*) *xs*)
by (*induct xs*) *simp-all*

lemma *length-filter-map*[*simp*]:
length (*filter P* (*map f xs*)) = *length*(*filter* (*P* ∘ *f*) *xs*)
by (*simp add*:*filter-map*)

lemma *filter-is-subset* [*simp*]: *set* (*filter P xs*) ≤ *set xs*
by *auto*

lemma *length-filter-less*:
⟦ *x* ∈ *set xs*; ¬ *P x* ⟧ ⟹ *length*(*filter P xs*) < *length xs*
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
using *Suc-le-eq* **by** *fastforce*

qed

lemma *length-filter-conv-card*:
length(*filter p xs*) = *card*{*i*. *i* < *length xs* ∧ *p*(*xs*!*i*)}
proof (*induct xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
let *?S* = {*i*. *i* < *length xs* ∧ *p*(*xs*!*i*)}
have *fin*: *finite ?S* **by**(*fast intro*: *bounded-nat-set-is-finite*)
show *?case* (**is** *?l* = *card ?S'*)
proof (*cases*)
assume *p x*
hence *eq*: *?S'* = *insert 0* (*Suc* ‘ *?S*)
by(*auto simp*: *image-def split*:*nat.split dest*:*gr0-implies-Suc*)
have *length* (*filter p* (*x* # *xs*)) = *Suc*(*card ?S*)
using *Cons* ‹*p x*› **by** *simp*
also have ... = *Suc*(*card*(*Suc* ‘ *?S*)) **using** *fin*
by (*simp add*: *card-image*)
also have ... = *card ?S'* **using** *eq fin*
by (*simp add*:*card-insert-if*)
finally show *?thesis* **.**
next
assume ¬ *p x*
hence *eq*: *?S'* = *Suc* ‘ *?S*
by(*auto simp add*: *image-def split*:*nat.split elim*:*lessE*)
have *length* (*filter p* (*x* # *xs*)) = *card ?S*
using *Cons* ‹¬ *p x*› **by** *simp*
also have ... = *card*(*Suc* ‘ *?S*) **using** *fin*
by (*simp add*: *card-image*)
also have ... = *card ?S'* **using** *eq fin*
by (*simp add*:*card-insert-if*)
finally show *?thesis* **.**
qed
qed

lemma *Cons-eq-filterD*:
x#*xs* = *filter P ys* ⟹
∃ *us vs*. *ys* = *us* @ *x* # *vs* ∧ (∀ *u*∈*set us*. ¬ *P u*) ∧ *P x* ∧ *xs* = *filter P vs*
(**is** - ⟹ ∃ *us vs*. *?P ys us vs*)
proof(*induct ys*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons y ys*)
show *?case* (**is** ∃ *x*. *?Q x*)
proof *cases*
assume *Py*: *P y*
show *?thesis*
proof *cases*

```
    assume x = y
    with Py Cons.prems have ?Q [] by simp
    then show ?thesis ..
   next
    assume x ≠ y
    with Py Cons.prems show ?thesis by simp
   qed
  next
   assume ¬ P y
   with Cons obtain us vs where ?P (y#ys) (y#us) vs by fastforce
   then have ?Q (y#us) by simp
   then show ?thesis ..
  qed
qed

lemma filter-eq-ConsD:
 filter P ys = x#xs ⟹
 ∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs
 by(rule Cons-eq-filterD) simp

lemma filter-eq-Cons-iff:
 (filter P ys = x#xs) =
 (∃us vs. ys = us @ x # vs ∧ (∀u∈set us. ¬ P u) ∧ P x ∧ xs = filter P vs)
 by(auto dest:filter-eq-ConsD)

lemmas Cons-eq-filter-iff = filter-eq-Cons-iff[THEN eq-iff-swap]

lemma inj-on-filter-key-eq:
 assumes inj-on f (insert y (set xs))
 shows filter (λx. f y = f x) xs = filter (HOL.eq y) xs
 using assms by (induct xs) auto

lemma filter-cong[fundef-cong]:
 xs = ys ⟹ (⋀x. x ∈ set ys ⟹ P x = Q x) ⟹ filter P xs = filter Q ys
 by (induct ys arbitrary: xs) auto
```

66.1.11 List partitioning

```
primrec partition :: ('a ⇒ bool) ⇒'a list ⇒ 'a list × 'a list where
 partition P [] = ([], []) |
 partition P (x # xs) =
 (let (yes, no) = partition P xs
  in if P x then (x # yes, no) else (yes, x # no))

lemma partition-filter1: fst (partition P xs) = filter P xs
 by (induct xs) (auto simp add: Let-def split-def)

lemma partition-filter2: snd (partition P xs) = filter (Not ∘ P) xs
 by (induct xs) (auto simp add: Let-def split-def)
```

lemma *partition-P*:
assumes *partition P xs* = (*yes*, *no*)
shows (∀ *p* ∈ *set yes*. *P p*) ∧ (∀ *p* ∈ *set no*. ¬ *P p*)
proof −
from *assms* **have** *yes* = *fst* (*partition P xs*) **and** *no* = *snd* (*partition P xs*)
by *simp-all*
then show *?thesis* **by** (*simp-all add*: *partition-filter1 partition-filter2*)
qed

lemma *partition-set*:
assumes *partition P xs* = (*yes*, *no*)
shows *set yes* ∪ *set no* = *set xs*
proof −
from *assms* **have** *yes* = *fst* (*partition P xs*) **and** *no* = *snd* (*partition P xs*)
by *simp-all*
then show *?thesis* **by** (*auto simp add*: *partition-filter1 partition-filter2*)
qed

lemma *partition-filter-conv*[*simp*]:
partition f xs = (*filter f xs*,*filter* (*Not* ∘ *f*) *xs*)
unfolding *partition-filter2*[*symmetric*]
unfolding *partition-filter1*[*symmetric*] **by** *simp*

declare *partition.simps*[*simp del*]

66.1.12 (!)

lemma *nth-Cons-0* [*simp*, *code*]: (*x* # *xs*)!*0* = *x*
by *auto*

lemma *nth-Cons-Suc* [*simp*, *code*]: (*x* # *xs*)!(*Suc n*) = *xs*!*n*
by *auto*

declare *nth.simps* [*simp del*]

lemma *nth-Cons-pos*[*simp*]: *0* < *n* ⟹ (*x*#*xs*) ! *n* = *xs* ! (*n* − *1*)
by(*auto simp*: *Nat.gr0-conv-Suc*)

lemma *nth-append*:
(*xs* @ *ys*)!*n* = (*if n* < *length xs then xs*!*n else ys*!(*n* − *length xs*))
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs*)
then show *?case*
using *less-Suc-eq-0-disj* **by** *auto*
qed *simp*

lemma *nth-append-left*: *i* < *length xs* ⟹ (*xs* @ *ys*) ! *i* = *xs* ! *i*
by (*auto simp*: *nth-append*)

lemma *nth-append-right*: $i \geq length\ xs \Longrightarrow (xs\ @\ ys)\ !\ i = ys\ !\ (i - length\ xs)$
by (*auto simp*: *nth-append*)

lemma *nth-append-length* [*simp*]: $(xs\ @\ x\ \#\ ys)\ !\ length\ xs = x$
by (*induct xs*) *auto*

lemma *nth-append-length-plus*[*simp*]: $(xs\ @\ ys)\ !\ (length\ xs + n) = ys\ !\ n$
by (*induct xs*) *auto*

lemma *nth-map* [*simp*]: $n < length\ xs \Longrightarrow (map\ f\ xs)!n = f(xs!n)$
proof (*induct xs arbitrary*: *n*)
case (*Cons x xs*)
then show *?case*
using *less-Suc-eq-0-disj* **by** *auto*
qed *simp*

lemma *nth-tl*: $n < length\ (tl\ xs) \Longrightarrow tl\ xs\ !\ n = xs\ !\ Suc\ n$
by (*induction xs*) *auto*

lemma *hd-conv-nth*: $xs \neq [] \Longrightarrow hd\ xs = xs!0$
by(*cases xs*) *simp-all*

lemma *list-eq-iff-nth-eq*:
$(xs = ys) = (length\ xs = length\ ys \wedge (\forall i{<}length\ xs.\ xs!i = ys!i))$
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*)
show *?case*
proof (*cases ys*)
case (*Cons y ys*)
with *Cons.hyps* **show** *?thesis* **by** *fastforce*
qed *simp*
qed *force*

lemma *map-equality-iff*:
$map\ f\ xs = map\ g\ ys \longleftrightarrow length\ xs = length\ ys \wedge (\forall i{<}length\ ys.\ f\ (xs!i) = g\ (ys!i))$
by (*fastforce simp*: *list-eq-iff-nth-eq*)

lemma *set-conv-nth*: $set\ xs = \{xs!i \mid i.\ i < length\ xs\}$
proof (*induct xs*)
case (*Cons x xs*)
have $insert\ x\ \{xs\ !\ i\ |i.\ i < length\ xs\} = \{(x\ \#\ xs)\ !\ i\ |i.\ i < Suc\ (length\ xs)\}$
(**is** *?L*=*?R*)
proof
show $?L \subseteq ?R$
by *force*
show $?R \subseteq ?L$
using *less-Suc-eq-0-disj* **by** *auto*

```
  qed
  with Cons show ?case
    by simp
qed simp

lemma in-set-conv-nth: (x ∈ set xs) = (∃ i < length xs. xs!i = x)
  by(auto simp:set-conv-nth)

lemma nth-equal-first-eq:
  assumes x ∉ set xs
  assumes n ≤ length xs
  shows (x # xs) ! n = x ⟷ n = 0 (is ?lhs ⟷ ?rhs)
proof
  assume ?lhs
  show ?rhs
  proof (rule ccontr)
    assume n ≠ 0
    then have n > 0 by simp
    with ‹?lhs› have xs ! (n − 1) = x by simp
    moreover from ‹n > 0› ‹n ≤ length xs› have n − 1 < length xs by simp
    ultimately have ∃ i<length xs. xs ! i = x by auto
    with ‹x ∉ set xs› in-set-conv-nth [of x xs] show False by simp
  qed
next
  assume ?rhs then show ?lhs by simp
qed

lemma nth-non-equal-first-eq:
  assumes x ≠ y
  shows (x # xs) ! n = y ⟷ xs ! (n − 1) = y ∧ n > 0 (is ?lhs ⟷ ?rhs)
proof
  assume ?lhs with assms have n > 0 by (cases n) simp-all
  with ‹?lhs› show ?rhs by simp
next
  assume ?rhs then show ?lhs by simp
qed

lemma list-ball-nth: ⟦n < length xs; ∀ x ∈ set xs. P x⟧ ⟹ P(xs!n)
  by (auto simp add: set-conv-nth)

lemma nth-mem [simp]: n < length xs ⟹ xs!n ∈ set xs
  by (auto simp add: set-conv-nth)

lemma all-nth-imp-all-set:
  ⟦∀ i < length xs. P(xs!i); x ∈ set xs⟧ ⟹ P x
  by (auto simp add: set-conv-nth)

lemma all-set-conv-all-nth:
  (∀ x ∈ set xs. P x) = (∀ i. i < length xs ⟶ P (xs ! i))
```

by (*auto simp add*: *set-conv-nth*)

lemma *rev-nth*:
n < *size xs* ⟹ *rev xs* ! *n* = *xs* ! (*length xs* − *Suc n*)
proof (*induct xs arbitrary*: *n*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*)
hence *n*: *n* < *Suc* (*length xs*) **by** *simp*
moreover
{ assume *n* < *length xs*
with *n* **obtain** *n′* **where** *n′*: *length xs* − *n* = *Suc n′*
by (*cases length xs* − *n*, *auto*)
moreover
from *n′* **have** *length xs* − *Suc n* = *n′* **by** *simp*
ultimately
have *xs* ! (*length xs* − *Suc n*) = (*x* # *xs*) ! (*length xs* − *n*) **by** *simp*
}
ultimately
show *?case* **by** (*clarsimp simp add*: *Cons nth-append*)
qed

lemma *Skolem-list-nth*:
(∀ *i*<*k*. ∃ *x*. *P i x*) = (∃ *xs*. *size xs* = *k* ∧ (∀ *i*<*k*. *P i* (*xs*!*i*)))
(**is** - = (∃ *xs*. *?P k xs*))
proof(*induct k*)
case *0* **show** *?case* **by** *simp*
next
case (*Suc k*)
show *?case* (**is** *?L* = *?R* **is** - = (∃ *xs*. *?P′ xs*))
proof
assume *?R* **thus** *?L* **using** *Suc* **by** *auto*
next
assume *?L*
with *Suc* **obtain** *x xs* **where** *?P k xs* ∧ *P k x* **by** (*metis less-Suc-eq*)
hence *?P′*(*xs*@[*x*]) **by**(*simp add*:*nth-append less-Suc-eq*)
thus *?R* **..**
qed
qed

66.1.13 *list-update*

lemma *length-list-update* [*simp*]: *length*(*xs*[*i*:=*x*]) = *length xs*
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *nth-list-update*:
i < *length xs*⟹ (*xs*[*i*:=*x*])!*j* = (*if i* = *j then x else xs*!*j*)
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *nth-list-update-eq* [*simp*]: $i < length\ xs \Longrightarrow (xs[i:=x])!i = x$
by (*simp add*: *nth-list-update*)

lemma *nth-list-update-neq* [*simp*]: $i \neq j \Longrightarrow xs[i:=x]!j = xs!j$
by (*induct xs arbitrary*: *i j*) (*auto simp add*: *nth-Cons split*: *nat.split*)

lemma *list-update-id*[*simp*]: $xs[i := xs!i] = xs$
by (*induct xs arbitrary*: *i*) (*simp-all split*:*nat.splits*)

lemma *list-update-beyond*: $length\ xs \leq i \Longrightarrow xs[i:=x] = xs$
proof (*induct xs arbitrary*: *i*)
case (*Cons x xs i*)
then show *?case*
by (*metis leD length-list-update list-eq-iff-nth-eq nth-list-update-neq*)
qed *simp*

lemma *list-update-nonempty*[*simp*]: $xs[k:=x] = [] \longleftrightarrow xs=[]$
by (*metis length-greater-0-conv length-list-update*)

lemma *list-update-same-conv*:
$i < length\ xs \Longrightarrow (xs[i := x] = xs) = (xs!i = x)$
by (*metis list-update-id nth-list-update-eq*)

lemma *list-update-append1*:
$i < size\ xs \Longrightarrow (xs\ @\ ys)[i:=x] = xs[i:=x]\ @\ ys$
by (*induct xs arbitrary*: *i*)(*auto split*:*nat.split*)

lemma *list-update-append*:
$(xs\ @\ ys)\ [n:=\ x] =$
$(if\ n < length\ xs\ then\ xs[n:=\ x]\ @\ ys\ else\ xs\ @\ (ys\ [n-length\ xs:=\ x]))$
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-length* [*simp*]:
$(xs\ @\ x\ \#\ ys)[length\ xs := y] = (xs\ @\ y\ \#\ ys)$
by (*induct xs*, *auto*)

lemma *map-update*: $map\ f\ (xs[k:=\ y]) = (map\ f\ xs)[k := f\ y]$
by(*induct xs arbitrary*: *k*)(*auto split*:*nat.splits*)

lemma *rev-update*:
$k < length\ xs \Longrightarrow rev\ (xs[k:=\ y]) = (rev\ xs)[length\ xs - k - 1 := y]$
by (*induct xs arbitrary*: *k*) (*auto simp*: *list-update-append split*:*nat.splits*)

lemma *update-zip*:
$(zip\ xs\ ys)[i:=xy] = zip\ (xs[i:=fst\ xy])\ (ys[i:=snd\ xy])$
by (*induct ys arbitrary*: *i xy xs*) (*auto*, *case-tac xs*, *auto split*: *nat.split*)

lemma *set-update-subset-insert*: $set(xs[i:=x]) \leq insert\ x\ (set\ xs)$
by (*induct xs arbitrary*: *i*) (*auto split*: *nat.split*)

lemma *set-update-subsetI*: $[\![set\ xs \subseteq A;\ x \in A]\!] \Longrightarrow set(xs[i := x]) \subseteq A$
by (*blast dest*!: *set-update-subset-insert* [*THEN subsetD*])

lemma *set-update-memI*: $n < length\ xs \Longrightarrow x \in set\ (xs[n := x])$
by (*induct xs arbitrary*: *n*) (*auto split*:*nat.splits*)

lemma *list-update-overwrite*[*simp*]:
$xs\ [i := x,\ i := y] = xs\ [i := y]$
by (*induct xs arbitrary*: *i*) (*simp-all split*: *nat.split*)

lemma *list-update-swap*:
$i \neq i' \Longrightarrow xs\ [i := x,\ i' := x'] = xs\ [i' := x',\ i := x]$
by (*induct xs arbitrary*: *i i'*) (*simp-all split*: *nat.split*)

lemma *list-update-code* [*code*]:
$[][i := y] = []$
$(x \# xs)[0 := y] = y \# xs$
$(x \# xs)[Suc\ i := y] = x \# xs[i := y]$
by *simp-all*

66.1.14 *last* and *butlast*

lemma *hd-Nil-eq-last*: $hd\ Nil = last\ Nil$
unfolding *hd-def last-def* **by** *simp*

lemma *last-snoc* [*simp*]: $last\ (xs\ @\ [x]) = x$
by (*induct xs*) *auto*

lemma *butlast-snoc* [*simp*]: $butlast\ (xs\ @\ [x]) = xs$
by (*induct xs*) *auto*

lemma *last-ConsL*: $xs = [] \Longrightarrow last(x\#xs) = x$
by *simp*

lemma *last-ConsR*: $xs \neq [] \Longrightarrow last(x\#xs) = last\ xs$
by *simp*

lemma *last-append*: $last(xs\ @\ ys) = (if\ ys = []\ then\ last\ xs\ else\ last\ ys)$
by (*induct xs*) (*auto*)

lemma *last-appendL*[*simp*]: $ys = [] \Longrightarrow last(xs\ @\ ys) = last\ xs$
by(*simp add*:*last-append*)

lemma *last-appendR*[*simp*]: $ys \neq [] \Longrightarrow last(xs\ @\ ys) = last\ ys$
by(*simp add*:*last-append*)

lemma *last-tl*: $xs = [] \vee tl\ xs \neq [] \Longrightarrow last\ (tl\ xs) = last\ xs$
by (*induct xs*) *simp-all*

lemma *butlast-tl*: *butlast* (*tl xs*) = *tl* (*butlast xs*)
by (*induct xs*) *simp-all*

lemma *hd-rev*: *hd*(*rev xs*) = *last xs*
by (*metis hd-Cons-tl hd-Nil-eq-last last-snoc rev-eq-Cons-iff rev-is-Nil-conv*)

lemma *last-rev*: *last*(*rev xs*) = *hd xs*
by (*metis hd-rev rev-swap*)

lemma *last-in-set*[*simp*]: *as* ≠ [] ⟹ *last as* ∈ *set as*
by (*induct as*) *auto*

lemma *length-butlast* [*simp*]: *length* (*butlast xs*) = *length xs* − *1*
by (*induct xs rule*: *rev-induct*) *auto*

lemma *butlast-append*:
butlast (*xs* @ *ys*) = (*if ys* = [] *then butlast xs else xs* @ *butlast ys*)
by (*induct xs arbitrary*: *ys*) *auto*

lemma *append-butlast-last-id* [*simp*]:
xs ≠ [] ⟹ *butlast xs* @ [*last xs*] = *xs*
by (*induct xs*) *auto*

lemma *in-set-butlastD*: *x* ∈ *set* (*butlast xs*) ⟹ *x* ∈ *set xs*
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *in-set-butlast-appendI*:
x ∈ *set* (*butlast xs*) ∨ *x* ∈ *set* (*butlast ys*) ⟹ *x* ∈ *set* (*butlast* (*xs* @ *ys*))
by (*auto dest*: *in-set-butlastD simp add*: *butlast-append*)

lemma *last-drop*[*simp*]: *n* < *length xs* ⟹ *last* (*drop n xs*) = *last xs*
by (*induct xs arbitrary*: *n*)(*auto split*:*nat.split*)

lemma *nth-butlast*:
assumes *n* < *length* (*butlast xs*) **shows** *butlast xs* ! *n* = *xs* ! *n*
proof (*cases xs*)
case (*Cons y ys*)
moreover from *assms* **have** *butlast xs* ! *n* = (*butlast xs* @ [*last xs*]) ! *n*
by (*simp add*: *nth-append*)
ultimately show *?thesis* **using** *append-butlast-last-id* **by** *simp*
qed *simp*

lemma *last-conv-nth*: *xs*≠[] ⟹ *last xs* = *xs*!(*length xs* − *1*)
by(*induct xs*)(*auto simp*:*neq-Nil-conv*)

lemma *butlast-conv-take*: *butlast xs* = *take* (*length xs* − *1*) *xs*
by (*induction xs rule*: *induct-list012*) *simp-all*

lemma *last-list-update*:
xs ≠ [] ⟹ *last*(*xs*[*k*:=*x*]) = (*if k* = *size xs* − *1 then x else last xs*)
by (*auto simp*: *last-conv-nth*)

lemma *butlast-list-update*:
butlast(*xs*[*k*:=*x*]) =
(*if k* = *size xs* − *1 then butlast xs else* (*butlast xs*)[*k*:=*x*])
by(*cases xs rule*:*rev-cases*)(*auto simp*: *list-update-beyond list-update-append split*: *nat.splits*)

lemma *last-map*: *xs* ≠ [] ⟹ *last* (*map f xs*) = *f* (*last xs*)
by (*cases xs rule*: *rev-cases*) *simp-all*

lemma *map-butlast*: *map f* (*butlast xs*) = *butlast* (*map f xs*)
by (*induct xs*) *simp-all*

lemma *snoc-eq-iff-butlast*:
xs @ [*x*] = *ys* ⟷ (*ys* ≠ [] ∧ *butlast ys* = *xs* ∧ *last ys* = *x*)
by *fastforce*

corollary *longest-common-suffix*:
∃ *ss xs′ ys′*. *xs* = *xs′* @ *ss* ∧ *ys* = *ys′* @ *ss*
∧ (*xs′* = [] ∨ *ys′* = [] ∨ *last xs′* ≠ *last ys′*)
using *longest-common-prefix*[*of rev xs rev ys*]
unfolding *rev-swap rev-append* **by** (*metis last-rev rev-is-Nil-conv*)

lemma *butlast-rev* [*simp*]: *butlast* (*rev xs*) = *rev* (*tl xs*)
by (*cases xs*) *simp-all*

66.1.15 *take* and *drop*

lemma *take-0*: *take 0 xs* = []
by (*induct xs*) *auto*

lemma *drop-0*: *drop 0 xs* = *xs*
by (*induct xs*) *auto*

lemma *take0*[*simp*]: *take 0* = (λ*xs*. [])
by(*rule ext*) (*rule take-0*)

lemma *drop0*[*simp*]: *drop 0* = (λ*x*. *x*)
by(*rule ext*) (*rule drop-0*)

lemma *take-Suc-Cons* [*simp*]: *take* (*Suc n*) (*x* # *xs*) = *x* # *take n xs*
by *simp*

lemma *drop-Suc-Cons* [*simp*]: *drop* (*Suc n*) (*x* # *xs*) = *drop n xs*
by *simp*

declare *take-Cons* [*simp del*] **and** *drop-Cons* [*simp del*]

lemma *take-Suc*: $xs \neq [] \Longrightarrow take\ (Suc\ n)\ xs = hd\ xs\ \#\ take\ n\ (tl\ xs)$
by(*clarsimp simp add*:*neq-Nil-conv*)

lemma *drop-Suc*: $drop\ (Suc\ n)\ xs = drop\ n\ (tl\ xs)$
by(*cases xs*, *simp-all*)

lemma *hd-take*[*simp*]: $j > 0 \Longrightarrow hd\ (take\ j\ xs) = hd\ xs$
by (*metis gr0-conv-Suc list.sel*(*1*) *take.simps*(*1*) *take-Suc*)

lemma *take-tl*: $take\ n\ (tl\ xs) = tl\ (take\ (Suc\ n)\ xs)$
by (*induct xs arbitrary*: *n*) *simp-all*

lemma *drop-tl*: $drop\ n\ (tl\ xs) = tl(drop\ n\ xs)$
by(*induct xs arbitrary*: *n*, *simp-all add*:*drop-Cons drop-Suc split*:*nat.split*)

lemma *tl-take*: $tl\ (take\ n\ xs) = take\ (n - 1)\ (tl\ xs)$
by (*cases n*, *simp*, *cases xs*, *auto*)

lemma *tl-drop*: $tl\ (drop\ n\ xs) = drop\ n\ (tl\ xs)$
by (*simp only*: *drop-tl*)

lemma *nth-via-drop*: $drop\ n\ xs = y\#ys \Longrightarrow xs!n = y$
by (*induct xs arbitrary*: *n*, *simp*)(*auto simp*: *drop-Cons nth-Cons split*: *nat.splits*)

lemma *take-Suc-conv-app-nth*:
$i < length\ xs \Longrightarrow take\ (Suc\ i)\ xs = take\ i\ xs\ @\ [xs!i]$
proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *Cons-nth-drop-Suc*:
$i < length\ xs \Longrightarrow (xs!i)\ \#\ (drop\ (Suc\ i)\ xs) = drop\ i\ xs$
proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *length-take* [*simp*]: $length\ (take\ n\ xs) = min\ (length\ xs)\ n$
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *length-drop* [*simp*]: *length* (*drop n xs*) = (*length xs* − *n*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all* [*simp*]: *length xs* ≤ *n* ⟹ *take n xs* = *xs*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-all* [*simp*]: *length xs* ≤ *n* ⟹ *drop n xs* = []
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-all-iff* [*simp*]: *take n xs* = *xs* ⟷ *length xs* ≤ *n*
by (*metis length-take min.order-iff take-all*)

lemma *take-eq-Nil*[*simp*]: (*take n xs* = []) = (*n* = *0* ∨ *xs* = [])
by(*induct xs arbitrary*: *n*)(*auto simp*: *take-Cons split*:*nat.split*)

lemmas *take-eq-Nil2*[*simp*] = *take-eq-Nil*[*THEN eq-iff-swap*]

lemma *drop-eq-Nil* [*simp*]: *drop n xs* = [] ⟷ *length xs* ≤ *n*
by (*metis diff-is-0-eq drop-all length-drop list.size*(*3*))

lemmas *drop-eq-Nil2* [*simp*] = *drop-eq-Nil*[*THEN eq-iff-swap*]

lemma *take-append* [*simp*]:
take n (*xs* @ *ys*) = (*take n xs* @ *take* (*n* − *length xs*) *ys*)
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *drop-append* [*simp*]:
drop n (*xs* @ *ys*) = *drop n xs* @ *drop* (*n* − *length xs*) *ys*
by (*induct n arbitrary*: *xs*) (*auto*, *case-tac xs*, *auto*)

lemma *take-take* [*simp*]: *take n* (*take m xs*) = *take* (*min n m*) *xs*
proof (*induct m arbitrary*: *xs n*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*; *cases n*) *simp-all*
qed

lemma *drop-drop* [*simp*]: *drop n* (*drop m xs*) = *drop* (*n* + *m*) *xs*
proof (*induct m arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *take-drop*: *take n* (*drop m xs*) = *drop m* (*take* (*n* + *m*) *xs*)
proof (*induct m arbitrary*: *xs n*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*; *cases n*) *simp-all*
qed

lemma *drop-take*: *drop n* (*take m xs*) = *take* (*m*−*n*) (*drop n xs*)
by(*induct xs arbitrary*: *m n*)(*auto simp*: *take-Cons drop-Cons split*: *nat.split*)

lemma *append-take-drop-id* [*simp*]: *take n xs* @ *drop n xs* = *xs*
proof (*induct n arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *take-cong-le*:
take n xs = *take n ys* ⟹ *m* ≤ *n* ⟹ *take m xs* = *take m ys*
by (*metis min-def take-take*)

lemma *drop-cong-le*:
drop n xs = *drop n ys* ⟹ *n* ≤ *m* ⟹ *drop m xs* = *drop m ys*
using *drop-drop*[*of m*−*n n xs*] *drop-drop*[*of m*−*n n ys*] **by** *simp*

lemma *take-map*: *take n* (*map f xs*) = *map f* (*take n xs*)
proof (*induct n arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *drop-map*: *drop n* (*map f xs*) = *map f* (*drop n xs*)
proof (*induct n arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *rev-take*: *rev* (*take i xs*) = *drop* (*length xs* − *i*) (*rev xs*)

proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *rev-drop*: *rev* (*drop i xs*) = *take* (*length xs* − *i*) (*rev xs*)
proof (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *drop-rev*: *drop n* (*rev xs*) = *rev* (*take* (*length xs* − *n*) *xs*)
by (*cases length xs* < *n*) (*auto simp*: *rev-take*)

lemma *take-rev*: *take n* (*rev xs*) = *rev* (*drop* (*length xs* − *n*) *xs*)
by (*cases length xs* < *n*) (*auto simp*: *rev-drop*)

lemma *nth-take* [*simp*]: *i* < *n* ⟹ (*take n xs*)!*i* = *xs*!*i*
proof (*induct xs arbitrary*: *i n*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases n*; *cases i*) *simp-all*
qed

lemma *nth-drop* [*simp*]:
n ≤ *length xs* ⟹ (*drop n xs*)!*i* = *xs*!(*n* + *i*)
proof (*induct n arbitrary*: *xs*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*) *simp-all*
qed

lemma *butlast-take*:
n ≤ *length xs* ⟹ *butlast* (*take n xs*) = *take* (*n* − *1*) *xs*
by (*simp add*: *butlast-conv-take*)

lemma *butlast-drop*: *butlast* (*drop n xs*) = *drop n* (*butlast xs*)
by (*simp add*: *butlast-conv-take drop-take ac-simps*)

lemma *take-butlast*: $n < length\ xs \Longrightarrow take\ n\ (butlast\ xs) = take\ n\ xs$
by (*simp add*: *butlast-conv-take*)

lemma *drop-butlast*: $drop\ n\ (butlast\ xs) = butlast\ (drop\ n\ xs)$
by (*simp add*: *butlast-conv-take drop-take ac-simps*)

lemma *butlast-power*: $(butlast\ \widehat{}\widehat{}\ n)\ xs = take\ (length\ xs - n)\ xs$
by (*induct n*) (*auto simp*: *butlast-take*)

lemma *hd-drop-conv-nth*: $n < length\ xs \Longrightarrow hd(drop\ n\ xs) = xs!n$
by(*simp add*: *hd-conv-nth*)

lemma *set-take-subset-set-take*:
$m \leq n \Longrightarrow set(take\ m\ xs) \leq set(take\ n\ xs)$
proof (*induct xs arbitrary*: *m n*)
case (*Cons x xs m n*) **then show** *?case*
by (*cases n*) (*auto simp*: *take-Cons*)
qed *simp*

lemma *set-take-subset*: $set(take\ n\ xs) \subseteq set\ xs$
by(*induct xs arbitrary*: *n*)(*auto simp*:*take-Cons split*:*nat.split*)

lemma *set-drop-subset*: $set(drop\ n\ xs) \subseteq set\ xs$
by(*induct xs arbitrary*: *n*)(*auto simp*:*drop-Cons split*:*nat.split*)

lemma *set-drop-subset-set-drop*:
$m \geq n \Longrightarrow set(drop\ m\ xs) \leq set(drop\ n\ xs)$
proof (*induct xs arbitrary*: *m n*)
case (*Cons x xs m n*)
then show *?case*
by (*clarsimp simp*: *drop-Cons split*: *nat.split*) (*metis set-drop-subset subset-iff*)
qed *simp*

lemma *in-set-takeD*: $x \in set(take\ n\ xs) \Longrightarrow x \in set\ xs$
using *set-take-subset* **by** *fast*

lemma *in-set-dropD*: $x \in set(drop\ n\ xs) \Longrightarrow x \in set\ xs$
using *set-drop-subset* **by** *fast*

lemma *append-eq-conv-conj*:
$(xs\ @\ ys = zs) = (xs = take\ (length\ xs)\ zs \wedge ys = drop\ (length\ xs)\ zs)$
proof (*induct xs arbitrary*: *zs*)
case (*Cons x xs zs*) **then show** *?case*
by (*cases zs*, *auto*)
qed *auto*

lemma *take-add*: $take\ (i+j)\ xs = take\ i\ xs\ @\ take\ j\ (drop\ i\ xs)$
proof (*induct xs arbitrary*: *i*)
case (*Cons x xs i*) **then show** *?case*

by (*cases i*, *auto*)
qed *auto*

lemma *append-eq-append-conv-if*:
(*xs*$_1$ @ *xs*$_2$ = *ys*$_1$ @ *ys*$_2$) =
(*if size xs*$_1$ ≤ *size ys*$_1$
then xs$_1$ = *take* (*size xs*$_1$) *ys*$_1$ ∧ *xs*$_2$ = *drop* (*size xs*$_1$) *ys*$_1$ @ *ys*$_2$
else take (*size ys*$_1$) *xs*$_1$ = *ys*$_1$ ∧ *drop* (*size ys*$_1$) *xs*$_1$ @ *xs*$_2$ = *ys*$_2$)
proof (*induct xs*$_1$ *arbitrary*: *ys*$_1$)
case (*Cons a xs*$_1$ *ys*$_1$) **then show** *?case*
by (*cases ys*$_1$, *auto*)
qed *auto*

lemma *take-hd-drop*:
n < *length xs* ⟹ *take n xs* @ [*hd* (*drop n xs*)] = *take* (*Suc n*) *xs*
by (*induct xs arbitrary*: *n*) (*simp-all add*:*drop-Cons split*:*nat.split*)

lemma *id-take-nth-drop*:
i < *length xs* ⟹ *xs* = *take i xs* @ *xs*!*i* # *drop* (*Suc i*) *xs*
proof −
assume *si*: *i* < *length xs*
hence *xs* = *take* (*Suc i*) *xs* @ *drop* (*Suc i*) *xs* **by** *auto*
moreover
from *si* **have** *take* (*Suc i*) *xs* = *take i xs* @ [*xs*!*i*]
using *take-Suc-conv-app-nth* **by** *blast*
ultimately show *?thesis* **by** *auto*
qed

lemma *take-update-cancel*[*simp*]: *n* ≤ *m* ⟹ *take n* (*xs*[*m* := *y*]) = *take n xs*
by(*simp add*: *list-eq-iff-nth-eq*)

lemma *drop-update-cancel*[*simp*]: *n* < *m* ⟹ *drop m* (*xs*[*n* := *x*]) = *drop m xs*
by(*simp add*: *list-eq-iff-nth-eq*)

lemma *upd-conv-take-nth-drop*:
i < *length xs* ⟹ *xs*[*i*:=*a*] = *take i xs* @ *a* # *drop* (*Suc i*) *xs*
proof −
assume *i*: *i* < *length xs*
have *xs*[*i*:=*a*] = (*take i xs* @ *xs*!*i* # *drop* (*Suc i*) *xs*)[*i*:=*a*]
by(*rule arg-cong*[*OF id-take-nth-drop*[*OF i*]])
also have ... = *take i xs* @ *a* # *drop* (*Suc i*) *xs*
using *i* **by** (*simp add*: *list-update-append*)
finally show *?thesis* **.**
qed

lemma *take-update-swap*: *take m* (*xs*[*n* := *x*]) = (*take m xs*)[*n* := *x*]
proof (*cases n* ≥ *length xs*)
case *False*
then show *?thesis*

by (*simp add*: *list-update-beyond upd-conv-take-nth-drop take-Cons drop-take min-def diff-Suc split*: *nat.split*)
qed (*auto simp*: *list-update-beyond*)

lemma *drop-update-swap*:
assumes $m \leq n$ **shows** *drop m* (*xs*[*n* := *x*]) = (*drop m xs*)[*n*−*m* := *x*]
proof (*cases n* ≥ *length xs*)
case *False*
with *assms* **show** *?thesis*
by (*simp add*: *upd-conv-take-nth-drop drop-take*)
qed (*auto simp*: *list-update-beyond*)

lemma *nth-image*: *l* ≤ *size xs* ⟹ *nth xs* ‘ {*0*..<*l*} = *set*(*take l xs*)
by (*simp add*: *set-conv-nth*) *force*

lemma *set-list-update*:
set (*xs* [*i* := *k*]) = *insert k* (*set* (*take i xs*) ∪ *set* (*drop* (*Suc i*) *xs*))
if ‹*i* < *length xs*›
using *that* **proof** (*induct xs arbitrary*: *i*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons x xs i*)
then show *?case*
by (*cases i*) (*simp-all add*: *insert-commute*)
qed

66.1.16 *takeWhile* and *dropWhile*

lemma *length-takeWhile-le*: *length* (*takeWhile P xs*) ≤ *length xs*
by (*induct xs*) *auto*

lemma *takeWhile-dropWhile-id* [*simp*]: *takeWhile P xs* @ *dropWhile P xs* = *xs*
by (*induct xs*) *auto*

lemma *takeWhile-append1* [*simp*]:
⟦*x* ∈ *set xs*; ¬*P*(*x*)⟧ ⟹ *takeWhile P* (*xs* @ *ys*) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *takeWhile-append2* [*simp*]:
(⋀*x*. *x* ∈ *set xs* ⟹ *P x*) ⟹ *takeWhile P* (*xs* @ *ys*) = *xs* @ *takeWhile P ys*
by (*induct xs*) *auto*

lemma *takeWhile-append*:
takeWhile P (*xs* @ *ys*) = (*if* ∀*x*∈*set xs*. *P x then xs* @ *takeWhile P ys else takeWhile P xs*)
using *takeWhile-append1*[*of* - *xs P ys*] *takeWhile-append2*[*of xs P ys*] **by** *auto*

lemma *takeWhile-tail*: $\neg\ P\ x \Longrightarrow takeWhile\ P\ (xs\ @\ (x\#l)) = takeWhile\ P\ xs$
by (*induct xs*) *auto*

lemma *takeWhile-eq-Nil-iff*: $takeWhile\ P\ xs = [] \longleftrightarrow xs = [] \vee \neg P\ (hd\ xs)$
by (*cases xs*) *auto*

lemma *takeWhile-nth*: $j < length\ (takeWhile\ P\ xs) \Longrightarrow takeWhile\ P\ xs\ !\ j = xs\ !\ j$
by (*metis nth-append takeWhile-dropWhile-id*)

lemma *takeWhile-takeWhile*: $takeWhile\ Q\ (takeWhile\ P\ xs) = takeWhile\ (\lambda x.\ P\ x \wedge Q\ x)\ xs$
by(*induct xs*) *simp-all*

lemma *dropWhile-nth*: $j < length\ (dropWhile\ P\ xs) \Longrightarrow$
$dropWhile\ P\ xs\ !\ j = xs\ !\ (j + length\ (takeWhile\ P\ xs))$
by (*metis add.commute nth-append-length-plus takeWhile-dropWhile-id*)

lemma *length-dropWhile-le*: $length\ (dropWhile\ P\ xs) \leq length\ xs$
by (*induct xs*) *auto*

lemma *dropWhile-append1* [*simp*]:
$[\![x \in set\ xs; \neg P(x)]\!] \Longrightarrow dropWhile\ P\ (xs\ @\ ys) = (dropWhile\ P\ xs)@ys$
by (*induct xs*) *auto*

lemma *dropWhile-append2* [*simp*]:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow P(x)) \Longrightarrow dropWhile\ P\ (xs\ @\ ys) = dropWhile\ P\ ys$
by (*induct xs*) *auto*

lemma *dropWhile-id*[*simp*]:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow \neg\ P\ x) \Longrightarrow dropWhile\ P\ xs = xs$
using *takeWhile-dropWhile-id*[*of P xs*] *takeWhile-eq-Nil-iff*[*of P xs*]
by *fastforce*

lemma *dropWhile-append3*:
$\neg\ P\ y \Longrightarrow dropWhile\ P\ (xs\ @\ y\ \#\ ys) = dropWhile\ P\ xs\ @\ y\ \#\ ys$
by (*induct xs*) *auto*

lemma *dropWhile-append*:
$dropWhile\ P\ (xs\ @\ ys) = (if\ \forall x{\in}set\ xs.\ P\ x\ then\ dropWhile\ P\ ys\ else\ dropWhile\ P\ xs\ @\ ys)$
using *dropWhile-append1*[*of - xs P ys*] *dropWhile-append2*[*of xs P ys*] **by** *auto*

lemma *dropWhile-last*:
$x \in set\ xs \Longrightarrow \neg\ P\ x \Longrightarrow last\ (dropWhile\ P\ xs) = last\ xs$
by (*auto simp add*: *dropWhile-append3 in-set-conv-decomp*)

lemma *set-dropWhileD*: $x \in set\ (dropWhile\ P\ xs) \Longrightarrow x \in set\ xs$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *set-takeWhileD*: $x \in set\ (takeWhile\ P\ xs) \Longrightarrow x \in set\ xs \wedge P\ x$
by (*induct xs*) (*auto split*: *if-split-asm*)

lemma *takeWhile-eq-all-conv*[*simp*]:
$(takeWhile\ P\ xs = xs) = (\forall x \in set\ xs.\ P\ x)$
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Nil-conv*[*simp*]:
$(dropWhile\ P\ xs = []) = (\forall x \in set\ xs.\ P\ x)$
by(*induct xs*, *auto*)

lemma *dropWhile-eq-Cons-conv*:
$(dropWhile\ P\ xs = y\#ys) = (xs = takeWhile\ P\ xs\ @\ y\ \#\ ys \wedge \neg\ P\ y)$
by(*induct xs*, *auto*)

lemma *dropWhile-eq-self-iff*: $dropWhile\ P\ xs = xs \longleftrightarrow xs = [] \vee \neg P\ (hd\ xs)$
by (*cases xs*) (*auto simp*: *dropWhile-eq-Cons-conv*)

lemma *dropWhile-dropWhile1*: $(\bigwedge x.\ Q\ x \Longrightarrow P\ x) \Longrightarrow dropWhile\ Q\ (dropWhile\ P\ xs) = dropWhile\ P\ xs$
by(*induct xs*) *simp-all*

lemma *dropWhile-dropWhile2*: $(\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow takeWhile\ P\ (takeWhile\ Q\ xs) = takeWhile\ P\ xs$
by(*induct xs*) *simp-all*

lemma *dropWhile-takeWhile*:
$(\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow dropWhile\ P\ (takeWhile\ Q\ xs) = takeWhile\ Q\ (dropWhile\ P\ xs)$
by (*induction xs*) *auto*

lemma *distinct-takeWhile*[*simp*]: $distinct\ xs \Longrightarrow distinct\ (takeWhile\ P\ xs)$
by (*induct xs*) (*auto dest*: *set-takeWhileD*)

lemma *distinct-dropWhile*[*simp*]: $distinct\ xs \Longrightarrow distinct\ (dropWhile\ P\ xs)$
by (*induct xs*) *auto*

lemma *takeWhile-map*: $takeWhile\ P\ (map\ f\ xs) = map\ f\ (takeWhile\ (P \circ f)\ xs)$
by (*induct xs*) *auto*

lemma *dropWhile-map*: $dropWhile\ P\ (map\ f\ xs) = map\ f\ (dropWhile\ (P \circ f)\ xs)$
by (*induct xs*) *auto*

lemma *takeWhile-eq-take*: $takeWhile\ P\ xs = take\ (length\ (takeWhile\ P\ xs))\ xs$
by (*induct xs*) *auto*

lemma *dropWhile-eq-drop*: $dropWhile\ P\ xs = drop\ (length\ (takeWhile\ P\ xs))\ xs$
by (*induct xs*) *auto*

lemma *hd-dropWhile*: *dropWhile P xs* ≠ [] ⟹ ¬ *P* (*hd* (*dropWhile P xs*))
by (*induct xs*) *auto*

lemma *takeWhile-eq-filter*:
assumes ⋀ *x*. *x* ∈ *set* (*dropWhile P xs*) ⟹ ¬ *P x*
shows *takeWhile P xs* = *filter P xs*
proof −
have *A*: *filter P xs* = *filter P* (*takeWhile P xs* @ *dropWhile P xs*)
by *simp*
have *B*: *filter P* (*dropWhile P xs*) = []
unfolding *filter-empty-conv* **using** *assms* **by** *blast*
have *filter P xs* = *takeWhile P xs*
unfolding *A filter-append B*
by (*auto simp add*: *filter-id-conv dest*: *set-takeWhileD*)
thus *?thesis* **..**
qed

lemma *takeWhile-eq-take-P-nth*:
⟦ ⋀ *i*. ⟦ *i* < *n* ; *i* < *length xs* ⟧ ⟹ *P* (*xs* ! *i*) ; *n* < *length xs* ⟹ ¬ *P* (*xs* ! *n*) ⟧
⟹
takeWhile P xs = *take n xs*
proof (*induct xs arbitrary*: *n*)
case *Nil*
thus *?case* **by** *simp*
next
case (*Cons x xs*)
show *?case*
proof (*cases n*)
case *0*
with *Cons* **show** *?thesis* **by** *simp*
next
case [*simp*]: (*Suc n′*)
have *P x* **using** *Cons.prems*(*1*)[*of 0*] **by** *simp*
moreover have *takeWhile P xs* = *take n′ xs*
proof (*rule Cons.hyps*)
fix *i*
assume *i* < *n′ i* < *length xs*
thus *P* (*xs* ! *i*) **using** *Cons.prems*(*1*)[*of Suc i*] **by** *simp*
next
assume *n′* < *length xs*
thus ¬ *P* (*xs* ! *n′*) **using** *Cons* **by** *auto*
qed
ultimately show *?thesis* **by** *simp*
qed
qed

lemma *nth-length-takeWhile*:
length (*takeWhile P xs*) < *length xs* ⟹ ¬ *P* (*xs* ! *length* (*takeWhile P xs*))

by (*induct xs*) *auto*

lemma *length-takeWhile-less-P-nth*:
assumes *all*: $\bigwedge$ *i*. *i* < *j* $\Longrightarrow$ *P* (*xs* ! *i*) **and** *j* ≤ *length xs*
shows *j* ≤ *length* (*takeWhile P xs*)
proof (*rule classical*)
assume ¬ *?thesis*
hence *length* (*takeWhile P xs*) < *length xs* **using** *assms* **by** *simp*
thus *?thesis* **using** *all* ‹¬ *?thesis*› *nth-length-takeWhile*[*of P xs*] **by** *auto*
qed

lemma *takeWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ $\Longrightarrow$
takeWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *rev* (*tl* (*dropWhile* (λ*y*. *y* ≠ *x*) *xs*))
by(*induct xs*) (*auto simp*: *takeWhile-tail*[**where** *l*=[]])

lemma *dropWhile-neq-rev*: ⟦*distinct xs*; *x* ∈ *set xs*⟧ $\Longrightarrow$
dropWhile (λ*y*. *y* ≠ *x*) (*rev xs*) = *x* # *rev* (*takeWhile* (λ*y*. *y* ≠ *x*) *xs*)
proof (*induct xs*)
case (*Cons a xs*)
then show *?case*
by(*auto*, *subst dropWhile-append2*, *auto*)
qed *simp*

lemma *takeWhile-not-last*:
distinct xs $\Longrightarrow$ *takeWhile* (λ*y*. *y* ≠ *last xs*) *xs* = *butlast xs*
by(*induction xs rule*: *induct-list012*) *auto*

lemma *takeWhile-cong* [*fundef-cong*]:
⟦*l* = *k*; $\bigwedge$*x*. *x* ∈ *set l* $\Longrightarrow$ *P x* = *Q x*⟧
$\Longrightarrow$ *takeWhile P l* = *takeWhile Q k*
by (*induct k arbitrary*: *l*) (*simp-all*)

lemma *dropWhile-cong* [*fundef-cong*]:
⟦*l* = *k*; $\bigwedge$*x*. *x* ∈ *set l* $\Longrightarrow$ *P x* = *Q x*⟧
$\Longrightarrow$ *dropWhile P l* = *dropWhile Q k*
by (*induct k arbitrary*: *l*, *simp-all*)

lemma *takeWhile-idem* [*simp*]:
takeWhile P (*takeWhile P xs*) = *takeWhile P xs*
by (*induct xs*) *auto*

lemma *dropWhile-idem* [*simp*]:
dropWhile P (*dropWhile P xs*) = *dropWhile P xs*
by (*induct xs*) *auto*

66.1.17 *zip*

lemma *zip-Nil* [*simp*]: *zip* [] *ys* = []
by (*induct ys*) *auto*

lemma *zip-Cons-Cons* [*simp*]: *zip* (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
by *simp*

declare *zip-Cons* [*simp del*]

lemma [*code*]:
zip [] *ys* = []
zip xs [] = []
zip (*x* # *xs*) (*y* # *ys*) = (*x*, *y*) # *zip xs ys*
by (*fact zip-Nil zip.simps*(*1*) *zip-Cons-Cons*)+

lemma *zip-Cons1*:
zip (*x*#*xs*) *ys* = (*case ys of* [] ⇒ [] | *y*#*ys* ⇒ (*x*,*y*)#*zip xs ys*)
by(*auto split*:*list.split*)

lemma *length-zip* [*simp*]:
length (*zip xs ys*) = *min* (*length xs*) (*length ys*)
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-obtain-same-length*:
assumes ⋀*zs ws n*. *length zs* = *length ws* ⟹ *n* = *min* (*length xs*) (*length ys*)
⟹ *zs* = *take n xs* ⟹ *ws* = *take n ys* ⟹ *P* (*zip zs ws*)
shows *P* (*zip xs ys*)
proof −
let *?n* = *min* (*length xs*) (*length ys*)
have *P* (*zip* (*take ?n xs*) (*take ?n ys*))
by (*rule assms*) *simp-all*
moreover have *zip xs ys* = *zip* (*take ?n xs*) (*take ?n ys*)
proof (*induct xs arbitrary*: *ys*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **then show** *?case* **by** (*cases ys*) *simp-all*
qed
ultimately show *?thesis* **by** *simp*
qed

lemma *zip-append1*:
zip (*xs* @ *ys*) *zs* =
zip xs (*take* (*length xs*) *zs*) @ *zip ys* (*drop* (*length xs*) *zs*)
by (*induct xs zs rule*:*list-induct2'*) *auto*

lemma *zip-append2*:
zip xs (*ys* @ *zs*) =
zip (*take* (*length ys*) *xs*) *ys* @ *zip* (*drop* (*length ys*) *xs*) *zs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *zip-append* [*simp*]:
⟦*length xs* = *length us*⟧ ⟹

zip (*xs*@*ys*) (*us*@*vs*) = *zip xs us* @ *zip ys vs*
by (*simp add*: *zip-append1*)

lemma *zip-rev*:
length xs = *length ys* ⟹ *zip* (*rev xs*) (*rev ys*) = *rev* (*zip xs ys*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *zip-map-map*:
zip (*map f xs*) (*map g ys*) = *map* (λ (*x*, *y*). (*f x*, *g y*)) (*zip xs ys*)
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs*) **note** *Cons-x-xs* = *Cons.hyps*
show *?case*
proof (*cases ys*)
case (*Cons y ys'*)
show *?thesis* **unfolding** *Cons* **using** *Cons-x-xs* **by** *simp*
qed *simp*
qed *simp*

lemma *zip-map1*:
zip (*map f xs*) *ys* = *map* (λ(*x*, *y*). (*f x*, *y*)) (*zip xs ys*)
using *zip-map-map*[*of f xs* λ*x*. *x ys*] **by** *simp*

lemma *zip-map2*:
zip xs (*map f ys*) = *map* (λ(*x*, *y*). (*x*, *f y*)) (*zip xs ys*)
using *zip-map-map*[*of* λ*x*. *x xs f ys*] **by** *simp*

lemma *map-zip-map*:
map f (*zip* (*map g xs*) *ys*) = *map* (%(*x*,*y*). *f*(*g x*, *y*)) (*zip xs ys*)
by (*auto simp*: *zip-map1*)

lemma *map-zip-map2*:
map f (*zip xs* (*map g ys*)) = *map* (%(*x*,*y*). *f*(*x*, *g y*)) (*zip xs ys*)
by (*auto simp*: *zip-map2*)

Courtesy of Andreas Lochbihler:

lemma *zip-same-conv-map*: *zip xs xs* = *map* (λ*x*. (*x*, *x*)) *xs*
by(*induct xs*) *auto*

lemma *nth-zip* [*simp*]:
⟦*i* < *length xs*; *i* < *length ys*⟧ ⟹ (*zip xs ys*)!*i* = (*xs*!*i*, *ys*!*i*)
proof (*induct ys arbitrary*: *i xs*)
case (*Cons y ys*)
then show *?case*
by (*cases xs*) (*simp-all add*: *nth.simps split*: *nat.split*)
qed *auto*

lemma *set-zip*:
set (*zip xs ys*) = {(*xs*!*i*, *ys*!*i*) | *i*. *i* < *min* (*length xs*) (*length ys*)}
by(*simp add*: *set-conv-nth cong*: *rev-conj-cong*)

lemma *zip-same*: $((a,b) \in set\ (zip\ xs\ xs)) = (a \in set\ xs \wedge a = b)$
by(*induct xs*) *auto*

lemma *zip-update*: $zip\ (xs[i:=x])\ (ys[i:=y]) = (zip\ xs\ ys)[i:=(x,y)]$
by (*simp add*: *update-zip*)

lemma *zip-replicate* [*simp*]:
$zip\ (replicate\ i\ x)\ (replicate\ j\ y) = replicate\ (min\ i\ j)\ (x,y)$
proof (*induct i arbitrary*: *j*)
case (*Suc i*)
then show *?case*
by (*cases j*, *auto*)
qed *auto*

lemma *zip-replicate1*: $zip\ (replicate\ n\ x)\ ys = map\ (Pair\ x)\ (take\ n\ ys)$
by(*induction ys arbitrary*: *n*)(*case-tac* [*2*] *n*, *simp-all*)

lemma *take-zip*: $take\ n\ (zip\ xs\ ys) = zip\ (take\ n\ xs)\ (take\ n\ ys)$
proof (*induct n arbitrary*: *xs ys*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*; *cases ys*) *simp-all*
qed

lemma *drop-zip*: $drop\ n\ (zip\ xs\ ys) = zip\ (drop\ n\ xs)\ (drop\ n\ ys)$
proof (*induct n arbitrary*: *xs ys*)
case *0*
then show *?case* **by** *simp*
next
case *Suc*
then show *?case* **by** (*cases xs*; *cases ys*) *simp-all*
qed

lemma *zip-takeWhile-fst*: $zip\ (takeWhile\ P\ xs)\ ys = takeWhile\ (P \circ fst)\ (zip\ xs\ ys)$
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases ys*) *auto*
qed

lemma *zip-takeWhile-snd*: $zip\ xs\ (takeWhile\ P\ ys) = takeWhile\ (P \circ snd)\ (zip\ xs\ ys)$
proof (*induct xs arbitrary*: *ys*)

case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases ys*) *auto*
qed

lemma *set-zip-leftD*: (*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *x* ∈ *set xs*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *set-zip-rightD*: (*x*,*y*)∈ *set* (*zip xs ys*) ⟹ *y* ∈ *set ys*
by (*induct xs ys rule*:*list-induct2'*) *auto*

lemma *in-set-zipE*:
(*x*,*y*) ∈ *set*(*zip xs ys*) ⟹ (⟦ *x* ∈ *set xs*; *y* ∈ *set ys* ⟧ ⟹ *R*) ⟹ *R*
by(*blast dest*: *set-zip-leftD set-zip-rightD*)

lemma *zip-map-fst-snd*: *zip* (*map fst zs*) (*map snd zs*) = *zs*
by (*induct zs*) *simp-all*

lemma *zip-eq-conv*:
length xs = *length ys* ⟹ *zip xs ys* = *zs* ⟷ *map fst zs* = *xs* ∧ *map snd zs* = *ys*
by (*auto simp add*: *zip-map-fst-snd*)

lemma *in-set-zip*:
p ∈ *set* (*zip xs ys*) ⟷ (∃ *n*. *xs* ! *n* = *fst p* ∧ *ys* ! *n* = *snd p*
∧ *n* < *length xs* ∧ *n* < *length ys*)
by (*cases p*) (*auto simp add*: *set-zip*)

lemma *in-set-impl-in-set-zip1*:
assumes *length xs* = *length ys*
assumes *x* ∈ *set xs*
obtains *y* **where** (*x*, *y*) ∈ *set* (*zip xs ys*)
proof −
from *assms* **have** *x* ∈ *set* (*map fst* (*zip xs ys*)) **by** *simp*
from *this that* **show** *?thesis* **by** *fastforce*
qed

lemma *in-set-impl-in-set-zip2*:
assumes *length xs* = *length ys*
assumes *y* ∈ *set ys*
obtains *x* **where** (*x*, *y*) ∈ *set* (*zip xs ys*)
proof −
from *assms* **have** *y* ∈ *set* (*map snd* (*zip xs ys*)) **by** *simp*
from *this that* **show** *?thesis* **by** *fastforce*
qed

lemma *zip-eq-Nil-iff*[*simp*]:
zip xs ys = [] ⟷ *xs* = [] ∨ *ys* = []

```
  by (cases xs; cases ys) simp-all

lemmas Nil-eq-zip-iff[simp] = zip-eq-Nil-iff[THEN eq-iff-swap]

lemma zip-eq-ConsE:
  assumes zip xs ys = xy # xys
  obtains x xs' y ys' where xs = x # xs'
    and ys = y # ys' and xy = (x, y)
    and xys = zip xs' ys'
proof −
  from assms have xs ≠ [] and ys ≠ []
    using zip-eq-Nil-iff [of xs ys] by simp-all
  then obtain x xs' y ys' where xs: xs = x # xs'
    and ys: ys = y # ys'
    by (cases xs; cases ys) auto
  with assms have xy = (x, y) and xys = zip xs' ys'
    by simp-all
  with xs ys show ?thesis ..
qed

lemma semilattice-map2:
  semilattice (map2 (∗)) if semilattice (∗)
    for f (infixl ‹∗› 70)
proof −
  from that interpret semilattice f .
  show ?thesis
  proof
    show map2 (∗) (map2 (∗) xs ys) zs = map2 (∗) xs (map2 (∗) ys zs)
      for xs ys zs :: 'a list
    proof (induction zip xs (zip ys zs) arbitrary: xs ys zs)
      case Nil
      from Nil [symmetric] show ?case
        by auto
    next
      case (Cons xyz xyzs)
      from Cons.hyps(2) [symmetric] show ?case
        by (rule zip-eq-ConsE) (erule zip-eq-ConsE,
          auto intro: Cons.hyps(1) simp add: ac-simps)
    qed
    show map2 (∗) xs ys = map2 (∗) ys xs
      for xs ys :: 'a list
    proof (induction zip xs ys arbitrary: xs ys)
      case Nil
      then show ?case
        by auto
    next
      case (Cons xy xys)
      from Cons.hyps(2) [symmetric] show ?case
        by (rule zip-eq-ConsE) (auto intro: Cons.hyps(1) simp add: ac-simps)
```

```
    qed
    show map2 (∗) xs xs = xs
      for xs :: 'a list
      by (induction xs) simp-all
  qed
qed

lemma pair-list-eqI:
  assumes map fst xs = map fst ys and map snd xs = map snd ys
  shows xs = ys
proof −
  from assms(1) have length xs = length ys by (rule map-eq-imp-length-eq)
  from this assms show ?thesis
    by (induct xs ys rule: list-induct2) (simp-all add: prod-eqI)
qed

lemma hd-zip:
  ‹hd (zip xs ys) = (hd xs, hd ys)› if ‹xs ≠ []› and ‹ys ≠ []›
  using that by (cases xs; cases ys) simp-all

lemma last-zip:
  ‹last (zip xs ys) = (last xs, last ys)› if ‹xs ≠ []› and ‹ys ≠ []›
    and ‹length xs = length ys›
  using that by (cases xs rule: rev-cases; cases ys rule: rev-cases) simp-all
```

66.1.18 *list-all2*

```
lemma list-all2-lengthD [intro?]:
  list-all2 P xs ys ⟹ length xs = length ys
  by (simp add: list-all2-iff)

lemma list-all2-Nil [iff, code]: list-all2 P [] ys = (ys = [])
  by (simp add: list-all2-iff)

lemma list-all2-Nil2 [iff, code]: list-all2 P xs [] = (xs = [])
  by (simp add: list-all2-iff)

lemma list-all2-Cons [iff, code]:
  list-all2 P (x # xs) (y # ys) = (P x y ∧ list-all2 P xs ys)
  by (auto simp add: list-all2-iff)

lemma list-all2-Cons1:
  list-all2 P (x # xs) ys = (∃ z zs. ys = z # zs ∧ P x z ∧ list-all2 P xs zs)
  by (cases ys) auto

lemma list-all2-Cons2:
  list-all2 P xs (y # ys) = (∃ z zs. xs = z # zs ∧ P z y ∧ list-all2 P zs ys)
  by (cases xs) auto
```

lemma *list-all2-induct*
[*consumes 1*, *case-names Nil Cons*, *induct set*: *list-all2*]:
assumes *P*: *list-all2 P xs ys*
assumes *Nil*: *R* [] []
assumes *Cons*: $\bigwedge x\ xs\ y\ ys.$
$[\![P\ x\ y;\ list\text{-}all2\ P\ xs\ ys;\ R\ xs\ ys]\!] \Longrightarrow R\ (x\ \#\ xs)\ (y\ \#\ ys)$
shows *R xs ys*
using *P*
by (*induct xs arbitrary*: *ys*) (*auto simp add*: *list-all2-Cons1 Nil Cons*)

lemma *list-all2-rev* [*iff*]:
list-all2 P (*rev xs*) (*rev ys*) = *list-all2 P xs ys*
by (*simp add*: *list-all2-iff zip-rev cong*: *conj-cong*)

lemma *list-all2-rev1*:
list-all2 P (*rev xs*) *ys* = *list-all2 P xs* (*rev ys*)
by (*subst list-all2-rev* [*symmetric*]) *simp*

lemma *list-all2-append1*:
list-all2 P (*xs* @ *ys*) *zs* =
($\exists$ *us vs*. *zs* = *us* @ *vs* $\wedge$ *length us* = *length xs* $\wedge$ *length vs* = *length ys* $\wedge$
list-all2 P xs us $\wedge$ *list-all2 P ys vs*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
apply (*rule-tac x* = *take* (*length xs*) *zs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length xs*) *zs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *list-all2-iff zip-append1*)
done
next
assume *?rhs*
then show *?lhs*
by (*auto simp add*: *list-all2-iff*)
qed

lemma *list-all2-append2*:
list-all2 P xs (*ys* @ *zs*) =
($\exists$ *us vs*. *xs* = *us* @ *vs* $\wedge$ *length us* = *length ys* $\wedge$ *length vs* = *length zs* $\wedge$
list-all2 P us ys $\wedge$ *list-all2 P vs zs*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
apply (*rule-tac x* = *take* (*length ys*) *xs* **in** *exI*)
apply (*rule-tac x* = *drop* (*length ys*) *xs* **in** *exI*)
apply (*force split*: *nat-diff-split simp add*: *list-all2-iff zip-append2*)
done
next
assume *?rhs*
then show *?lhs*

by (*auto simp add*: *list-all2-iff*)
qed

lemma *list-all2-append*:
length xs = *length ys* ⟹
list-all2 P (*xs*@*us*) (*ys*@*vs*) = (*list-all2 P xs ys* ∧ *list-all2 P us vs*)
by (*induct rule*:*list-induct2*, *simp-all*)

lemma *list-all2-appendI* [*intro?*, *trans*]:
⟦ *list-all2 P a b*; *list-all2 P c d* ⟧ ⟹ *list-all2 P* (*a*@*c*) (*b*@*d*)
by (*simp add*: *list-all2-append list-all2-lengthD*)

lemma *list-all2-conv-all-nth*:
list-all2 P xs ys =
(*length xs* = *length ys* ∧ (∀ *i* < *length xs*. *P* (*xs*!*i*) (*ys*!*i*)))
by (*force simp add*: *list-all2-iff set-zip*)

lemma *list-all2-trans*:
assumes *tr*: !!*a b c*. *P1 a b* ⟹ *P2 b c* ⟹ *P3 a c*
shows !!*bs cs*. *list-all2 P1 as bs* ⟹ *list-all2 P2 bs cs* ⟹ *list-all2 P3 as cs*
(**is** !!*bs cs*. *PROP ?Q as bs cs*)
proof (*induct as*)
fix *x xs bs* **assume** *I1*: !!*bs cs*. *PROP ?Q xs bs cs*
show !!*cs*. *PROP ?Q* (*x* # *xs*) *bs cs*
proof (*induct bs*)
fix *y ys cs* **assume** *I2*: !!*cs*. *PROP ?Q* (*x* # *xs*) *ys cs*
show *PROP ?Q* (*x* # *xs*) (*y* # *ys*) *cs*
by (*induct cs*) (*auto intro*: *tr I1 I2*)
qed *simp*
qed *simp*

lemma *list-all2-all-nthI* [*intro?*]:
length a = *length b* ⟹ (⋀*n*. *n* < *length a* ⟹ *P* (*a*!*n*) (*b*!*n*)) ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2I*:
∀ *x* ∈ *set* (*zip a b*). *case-prod P x* ⟹ *length a* = *length b* ⟹ *list-all2 P a b*
by (*simp add*: *list-all2-iff*)

lemma *list-all2-nthD*:
⟦ *list-all2 P xs ys*; *p* < *size xs* ⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-nthD2*:
⟦*list-all2 P xs ys*; *p* < *size ys*⟧ ⟹ *P* (*xs*!*p*) (*ys*!*p*)
by (*frule list-all2-lengthD*) (*auto intro*: *list-all2-nthD*)

lemma *list-all2-map1*:
list-all2 P (*map f as*) *bs* = *list-all2* (λ*x y*. *P* (*f x*) *y*) *as bs*

by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-map2*:
list-all2 P as (*map f bs*) = *list-all2* (λ*x y*. *P x* (*f y*)) *as bs*
by (*auto simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-refl* [*intro?*]:
(⋀*x*. *P x x*) ⟹ *list-all2 P xs xs*
by (*simp add*: *list-all2-conv-all-nth*)

lemma *list-all2-update-cong*:
⟦ *list-all2 P xs ys*; *P x y* ⟧ ⟹ *list-all2 P* (*xs*[*i*:=*x*]) (*ys*[*i*:=*y*])
by (*cases i* < *length ys*) (*auto simp add*: *list-all2-conv-all-nth nth-list-update*)

lemma *list-all2-takeI* [*simp*,*intro?*]:
list-all2 P xs ys ⟹ *list-all2 P* (*take n xs*) (*take n ys*)
proof (*induct xs arbitrary*: *n ys*)
case (*Cons x xs*)
then show *?case*
by (*cases n*) (*auto simp*: *list-all2-Cons1*)
qed *auto*

lemma *list-all2-dropI* [*simp*,*intro?*]:
list-all2 P xs ys ⟹ *list-all2 P* (*drop n xs*) (*drop n ys*)
proof (*induct xs arbitrary*: *n ys*)
case (*Cons x xs*)
then show *?case*
by (*cases n*) (*auto simp*: *list-all2-Cons1*)
qed *auto*

lemma *list-all2-mono* [*intro?*]:
list-all2 P xs ys ⟹ (⋀*xs ys*. *P xs ys* ⟹ *Q xs ys*) ⟹ *list-all2 Q xs ys*
by (*rule list.rel-mono-strong*)

lemma *list-all2-eq*:
xs = *ys* ⟷ *list-all2* (=) *xs ys*
by (*induct xs ys rule*: *list-induct2'*) *auto*

lemma *list-eq-iff-zip-eq*:
xs = *ys* ⟷ *length xs* = *length ys* ∧ (∀(*x*,*y*) ∈ *set* (*zip xs ys*). *x* = *y*)
by(*auto simp add*: *set-zip list-all2-eq list-all2-conv-all-nth cong*: *conj-cong*)

lemma *list-all2-same*: *list-all2 P xs xs* ⟷ (∀*x*∈*set xs*. *P x x*)
by(*auto simp add*: *list-all2-conv-all-nth set-conv-nth*)

lemma *zip-assoc*:
zip xs (*zip ys zs*) = *map* (λ((*x*, *y*), *z*). (*x*, *y*, *z*)) (*zip* (*zip xs ys*) *zs*)
by(*rule list-all2-all-nthI*[**where** *P*=(=), *unfolded list.rel-eq*]) *simp-all*

lemma *zip-commute*: *zip xs ys = map* (λ(*x*, *y*). (*y*, *x*)) (*zip ys xs*)
by(*rule list-all2-all-nthI*[**where** *P*=(=), *unfolded list.rel-eq*]) *simp-all*

lemma *zip-left-commute*:
zip xs (*zip ys zs*) = *map* (λ(*y*, (*x*, *z*)). (*x*, *y*, *z*)) (*zip ys* (*zip xs zs*))
by(*rule list-all2-all-nthI*[**where** *P*=(=), *unfolded list.rel-eq*]) *simp-all*

lemma *zip-replicate2*: *zip xs* (*replicate n y*) = *map* (λ*x*. (*x*, *y*)) (*take n xs*)
by(*subst zip-commute*)(*simp add*: *zip-replicate1*)

66.1.19 *List.product* and *product-lists*

lemma *product-concat-map*:
List.product xs ys = concat (*map* (λ*x*. *map* (λ*y*. (*x*,*y*)) *ys*) *xs*)
by(*induction xs*) (*simp*)+

lemma *set-product*[*simp*]: *set* (*List.product xs ys*) = *set xs* × *set ys*
by (*induct xs*) *auto*

lemma *length-product* [*simp*]:
length (*List.product xs ys*) = *length xs* ∗ *length ys*
by (*induct xs*) *simp-all*

lemma *product-nth*:
assumes *n* < *length xs* ∗ *length ys*
shows *List.product xs ys* ! *n* = (*xs* ! (*n div length ys*), *ys* ! (*n mod length ys*))
using *assms* **proof** (*induct xs arbitrary*: *n*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs n*)
then have *length ys* > *0* **by** *auto*
with *Cons* **show** *?case*
by (*auto simp add*: *nth-append not-less le-mod-geq le-div-geq*)
qed

lemma *in-set-product-lists-length*:
xs ∈ *set* (*product-lists xss*) ⟹ *length xs = length xss*
by (*induct xss arbitrary*: *xs*) *auto*

lemma *product-lists-set*:
set (*product-lists xss*) = {*xs*. *list-all2* (λ*x ys*. *x* ∈ *set ys*) *xs xss*} (**is** *?L = Collect ?R*)
proof (*intro equalityI subsetI*, *unfold mem-Collect-eq*)
fix *xs* **assume** *xs* ∈ *?L*
then have *length xs = length xss* **by** (*rule in-set-product-lists-length*)
from *this* ‹*xs* ∈ *?L*› **show** *?R xs* **by** (*induct xs xss rule*: *list-induct2*) *auto*
next
fix *xs* **assume** *?R xs*
then show *xs* ∈ *?L* **by** *induct auto*

qed

66.1.20 *fold* with natural argument order

lemma *fold-simps* [*code*]: — eta-expanded variant for generated code – enables tail-recursion optimisation in Scala
fold f [] *s* = *s*
fold f (*x* # *xs*) *s* = *fold f xs* (*f x s*)
by *simp-all*

lemma *fold-remove1-split*:
⟦ ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *f x* ∘ *f y* = *f y* ∘ *f x*;
x ∈ *set xs* ⟧
⟹ *fold f xs* = *fold f* (*remove1 x xs*) ∘ *f x*
by (*induct xs*) (*auto simp add*: *comp-assoc*)

lemma *fold-cong* [*fundef-cong*]:
a = *b* ⟹ *xs* = *ys* ⟹ (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *g x*)
⟹ *fold f xs a* = *fold g ys b*
by (*induct ys arbitrary*: *a b xs*) *simp-all*

lemma *fold-id*: (⋀*x*. *x* ∈ *set xs* ⟹ *f x* = *id*) ⟹ *fold f xs* = *id*
by (*induct xs*) *simp-all*

lemma *fold-commute*:
(⋀*x*. *x* ∈ *set xs* ⟹ *h* ∘ *g x* = *f x* ∘ *h*) ⟹ *h* ∘ *fold g xs* = *fold f xs* ∘ *h*
by (*induct xs*) (*simp-all add*: *fun-eq-iff*)

lemma *fold-commute-apply*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *h* ∘ *g x* = *f x* ∘ *h*
shows *h* (*fold g xs s*) = *fold f xs* (*h s*)
proof −
from *assms* **have** *h* ∘ *fold g xs* = *fold f xs* ∘ *h* **by** (*rule fold-commute*)
then show *?thesis* **by** (*simp add*: *fun-eq-iff*)
qed

lemma *fold-invariant*:
⟦ ⋀*x*. *x* ∈ *set xs* ⟹ *Q x*; *P s*; ⋀*x s*. *Q x* ⟹ *P s* ⟹ *P* (*f x s*) ⟧
⟹ *P* (*fold f xs s*)
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *fold-append* [*simp*]: *fold f* (*xs* @ *ys*) = *fold f ys* ∘ *fold f xs*
by (*induct xs*) *simp-all*

lemma *fold-map* [*code-unfold*]: *fold g* (*map f xs*) = *fold* (*g* ∘ *f*) *xs*
by (*induct xs*) *simp-all*

lemma *fold-filter*:
fold f (*filter P xs*) = *fold* (λ*x*. *if P x then f x else id*) *xs*

by (*induct xs*) *simp-all*

lemma *fold-rev*:
(⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *f y* ∘ *f x* = *f x* ∘ *f y*)
⟹ *fold f* (*rev xs*) = *fold f xs*
by (*induct xs*) (*simp-all add*: *fold-commute-apply fun-eq-iff*)

lemma *fold-Cons-rev*: *fold Cons xs* = *append* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *rev-conv-fold* [*code*]: *rev xs* = *fold Cons xs* []
by (*simp add*: *fold-Cons-rev*)

lemma *fold-append-concat-rev*: *fold append xss* = *append* (*concat* (*rev xss*))
by (*induct xss*) *simp-all*

lemma *fold-inject*:
assumes
⋀*w x y z*. *f w x* = *f y z* ⟷ *w* = *y* ∧ *x* = *z* **and**
⋀*x y*. *f x y* ≠ *a* **and**
⋀*x y*. *f x y* ≠ *b*
shows *fold f xs a* = *fold f ys b* ⟷ *xs* = *ys* ∧ *a* = *b*
by (*induction xs ys rule*: *List.rev-induct2*) (*use assms*(*2*,*3*,*1*) **in** *auto*)

Finite-Set.fold and *fold*

lemma (**in** *comp-fun-commute-on*) *fold-set-fold-remdups*:
assumes *set xs* ⊆ *S*
shows *Finite-Set.fold f y* (*set xs*) = *fold f* (*remdups xs*) *y*
by (*rule sym*, *use assms* **in** ‹*induct xs arbitrary*: *y*›)
(*simp-all add*: *insert-absorb fold-fun-left-comm*)

lemma (**in** *comp-fun-idem-on*) *fold-set-fold*:
assumes *set xs* ⊆ *S*
shows *Finite-Set.fold f y* (*set xs*) = *fold f xs y*
by (*rule sym*, *use assms* **in** ‹*induct xs arbitrary*: *y*›) (*simp-all add*: *fold-fun-left-comm*)

lemma *union-set-fold* [*code*]: *set xs* ∪ *A* = *fold Set.insert xs A*
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
show *?thesis* **by** (*simp add*: *union-fold-insert fold-set-fold*)
qed

lemma *union-coset-filter* [*code*]:
List.coset xs ∪ *A* = *List.coset* (*List.filter* (λ*x*. *x* ∉ *A*) *xs*)
by *auto*

lemma *minus-set-fold* [*code*]: *A* − *set xs* = *fold Set.remove xs A*
proof −

```
  interpret comp-fun-idem Set.remove
    by (fact comp-fun-idem-remove)
  show ?thesis
    by (simp add: minus-fold-remove [of - A] fold-set-fold)
qed

lemma minus-coset-filter [code]:
  A − List.coset xs = set (List.filter (λx. x ∈ A) xs)
  by auto

lemma inter-set-filter [code]:
  A ∩ set xs = set (List.filter (λx. x ∈ A) xs)
  by auto

lemma inter-coset-fold [code]:
  A ∩ List.coset xs = fold Set.remove xs A
  by (simp add: Diff-eq [symmetric] minus-set-fold)

definition abort-empty-set :: ‹('a set ⇒ 'a) ⇒ 'a›
  where [simp]: ‹abort-empty-set F = F {}›

declare [[code abort: abort-empty-set]]

lemma (in semilattice-set) set-empty-abort [code]:
  ‹F (set []) = abort-empty-set F›
  by simp

lemma (in semilattice-set) set-eq-fold [code]:
  ‹F (set (x # xs)) = fold f xs x›
proof −
  interpret comp-fun-idem f
    by standard (simp-all add: fun-eq-iff left-commute)
  show ?thesis by (simp add: eq-fold fold-set-fold)
qed

lemma (in complete-lattice) Inf-set-fold [code]:
  Inf (set xs) = fold inf xs top
proof −
  interpret comp-fun-idem inf :: 'a ⇒ 'a ⇒ 'a
    by (fact comp-fun-idem-inf)
  show ?thesis by (simp add: Inf-fold-inf fold-set-fold inf-commute)
qed

declare Inf-set-fold [where 'a = ‹'a set›, code]

lemma (in complete-lattice) Sup-set-fold [code]:
  Sup (set xs) = fold sup xs bot
proof −
  interpret comp-fun-idem sup :: 'a ⇒ 'a ⇒ 'a
```

by (*fact comp-fun-idem-sup*)
show *?thesis* **by** (*simp add*: *Sup-fold-sup fold-set-fold sup-commute*)
qed

declare *Sup-set-fold* [**where** *'a* = ‹*'a set*›, *code*]

lemma (**in** *complete-lattice*) *INF-set-fold*:
$\bigsqcap$(*f* ‘ *set xs*) = *fold* (*inf* ∘ *f*) *xs top*
using *Inf-set-fold* [*of map f xs*] **by** (*simp add*: *fold-map*)

lemma (**in** *complete-lattice*) *SUP-set-fold*:
$\bigsqcup$(*f* ‘ *set xs*) = *fold* (*sup* ∘ *f*) *xs bot*
using *Sup-set-fold* [*of map f xs*] **by** (*simp add*: *fold-map*)

66.1.21 Fold variants: *foldr* and *foldl*

Correspondence

lemma *foldr-conv-fold* [*code-abbrev*]: *foldr f xs* = *fold f* (*rev xs*)
by (*induct xs*) *simp-all*

lemma *foldl-conv-fold*: *foldl f s xs* = *fold* (λ*x s*. *f s x*) *xs s*
by (*induct xs arbitrary*: *s*) *simp-all*

lemma *foldr-conv-foldl*: — The "Third Duality Theorem" in Bird & Wadler:
foldr f xs a = *foldl* (λ*x y*. *f y x*) *a* (*rev xs*)
by (*simp add*: *foldr-conv-fold foldl-conv-fold*)

lemma *foldl-conv-foldr*:
foldl f a xs = *foldr* (λ*x y*. *f y x*) (*rev xs*) *a*
by (*simp add*: *foldr-conv-fold foldl-conv-fold*)

lemma *foldr-fold*:
(⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *f y* ∘ *f x* = *f x* ∘ *f y*)
⟹ *foldr f xs* = *fold f xs*
unfolding *foldr-conv-fold* **by** (*rule fold-rev*)

lemma *foldr-cong* [*fundef-cong*]:
a = *b* ⟹ *l* = *k* ⟹ (⋀*a x*. *x* ∈ *set l* ⟹ *f x a* = *g x a*) ⟹ *foldr f l a* = *foldr g k b*
by (*auto simp add*: *foldr-conv-fold intro*!: *fold-cong*)

lemma *foldl-cong* [*fundef-cong*]:
a = *b* ⟹ *l* = *k* ⟹ (⋀*a x*. *x* ∈ *set l* ⟹ *f a x* = *g a x*) ⟹ *foldl f a l* = *foldl g b k*
by (*auto simp add*: *foldl-conv-fold intro*!: *fold-cong*)

lemma *foldr-append* [*simp*]: *foldr f* (*xs* @ *ys*) *a* = *foldr f xs* (*foldr f ys a*)
by (*simp add*: *foldr-conv-fold*)

lemma *foldl-append* [*simp*]: *foldl f a* (*xs* @ *ys*) = *foldl f* (*foldl f a xs*) *ys*
by (*simp add*: *foldl-conv-fold*)

lemma *foldr-map* [*code-unfold*]: *foldr g* (*map f xs*) *a* = *foldr* (*g* ∘ *f*) *xs a*
by (*simp add*: *foldr-conv-fold fold-map rev-map*)

lemma *foldr-filter*:
foldr f (*filter P xs*) = *foldr* (λ*x*. *if P x then f x else id*) *xs*
by (*simp add*: *foldr-conv-fold rev-filter fold-filter*)

lemma *foldl-map* [*code-unfold*]:
foldl g a (*map f xs*) = *foldl* (λ*a x*. *g a* (*f x*)) *a xs*
by (*simp add*: *foldl-conv-fold fold-map comp-def*)

lemma *concat-conv-foldr* [*code*]:
concat xss = *foldr append xss* []
by (*simp add*: *fold-append-concat-rev foldr-conv-fold*)

lemma *foldl-inject*:
assumes
⋀*w x y z*. *f w x* = *f y z* ⟷ *w* = *y* ∧ *x* = *z* **and**
⋀*x y*. *f x y* ≠ *a* **and**
⋀*x y*. *f x y* ≠ *b*
shows *foldl f a xs* = *foldl f b ys* ⟷ *a* = *b* ∧ *xs* = *ys*
by (*induction xs ys rule*: *rev-induct2*) (*use assms*(*2*,*3*,*1*) **in** *auto*)

lemma *foldr-inject*:
assumes
⋀*w x y z*. *f w x* = *f y z* ⟷ *w* = *y* ∧ *x* = *z* **and**
⋀*x y*. *f x y* ≠ *a* **and**
⋀*x y*. *f x y* ≠ *b*
shows *foldr f xs a* = *foldr f ys b* ⟷ *xs* = *ys* ∧ *a* = *b*
by (*induction xs ys rule*: *list-induct2′*) (*use assms*(*2*,*3*,*1*) **in** *auto*)

66.1.22 *upt*

lemma *upt-rec*[*code*]: [*i*..<*j*] = (*if i*<*j then i*#[*Suc i*..<*j*] *else* [])
— simp does not terminate!
by (*induct j*) *auto*

lemmas *upt-rec-numeral*[*simp*] = *upt-rec*[*of numeral m numeral n*] **for** *m n*

lemma *upt-conv-Nil* [*simp*]: *j* ≤ *i* ⟹ [*i*..<*j*] = []
by (*subst upt-rec*) *simp*

lemma *upt-eq-Nil-conv*[*simp*]: ([*i*..<*j*] = []) = (*j* = *0* ∨ *j* ≤ *i*)
by(*induct j*)*simp-all*

lemma *upt-eq-Cons-conv*:

```
  ([i..<j] = x#xs) = (i < j ∧ i = x ∧ [i+1..<j] = xs)
proof (induct j arbitrary: x xs)
  case (Suc j)
  then show ?case
    by (simp add: upt-rec)
qed simp

lemma upt-Suc-append: i ≤ j ⟹ [i..<(Suc j)] = [i..<j]@[j]
  — Only needed if upt-Suc is deleted from the simpset.
  by simp

lemma upt-conv-Cons: i < j ⟹ [i..<j] = i # [Suc i..<j]
  by (simp add: upt-rec)

lemma upt-conv-Cons-Cons: — no precondition
  m # n # ns = [m..<q] ⟷ n # ns = [Suc m..<q]
proof (cases m < q)
  case False then show ?thesis by simp
next
  case True then show ?thesis by (simp add: upt-conv-Cons)
qed

lemma upt-add-eq-append: i<=j ⟹ [i..<j+k] = [i..<j]@[j..<j+k]
  — LOOPS as a simprule, since j ≤ j.
  by (induct k) auto

lemma length-upt [simp]: length [i..<j] = j - i
  by (induct j) (auto simp add: Suc-diff-le)

lemma nth-upt [simp]: i + k < j ⟹ [i..<j] ! k = i + k
  by (induct j) (auto simp add: less-Suc-eq nth-append split: nat-diff-split)

lemma hd-upt[simp]: i < j ⟹ hd[i..<j] = i
  by(simp add:upt-conv-Cons)

lemma tl-upt [simp]: tl [m..<n] = [Suc m..<n]
  by (simp add: upt-rec)

lemma last-upt[simp]: i < j ⟹ last[i..<j] = j - 1
  by(cases j)(auto simp: upt-Suc-append)

lemma take-upt [simp]: i+m ≤ n ⟹ take m [i..<n] = [i..<i+m]
proof (induct m arbitrary: i)
  case (Suc m)
  then show ?case
    by (subst take-Suc-conv-app-nth) auto
qed simp

lemma drop-upt[simp]: drop m [i..<j] = [i+m..<j]
```

by(*induct j*) *auto*

lemma *map-Suc-upt*: *map Suc* [*m*..<*n*] = [*Suc m*..<*Suc n*]
by (*induct n*) *auto*

lemma *map-add-upt*: *map* (λ*i*. *i* + *n*) [0..<*m*] = [*n*..<*m* + *n*]
by (*induct m*) *simp-all*

lemma *nth-map-upt*: *i* < *n*−*m* ⟹ (*map f* [*m*..<*n*]) ! *i* = *f*(*m*+*i*)
proof (*induct n m arbitrary*: *i rule*: *diff-induct*)
case (*3 x y*)
then show *?case*
by (*metis add.commute length-upt less-diff-conv nth-map nth-upt*)
qed *auto*

lemma *map-decr-upt*: *map* (λ*n*. *n* − *Suc 0*) [*Suc m*..<*Suc n*] = [*m*..<*n*]
by (*induct n*) *simp-all*

lemma *map-upt-Suc*: *map f* [*0* ..< *Suc n*] = *f 0* # *map* (λ*i*. *f* (*Suc i*)) [*0* ..< *n*]
by (*induct n arbitrary*: *f*) *auto*

lemma *nth-take-lemma*:
k ≤ *length xs* ⟹ *k* ≤ *length ys* ⟹
(⋀*i*. *i* < *k* ⟶ *xs*!*i* = *ys*!*i*) ⟹ *take k xs* = *take k ys*
by (*induct k arbitrary*: *xs ys*) (*simp-all add*: *take-Suc-conv-app-nth*)

lemma *nth-equalityI*:
⟦*length xs* = *length ys*; ⋀*i*. *i* < *length xs* ⟹ *xs*!*i* = *ys*!*i*⟧ ⟹ *xs* = *ys*
by (*frule nth-take-lemma* [*OF le-refl eq-imp-le*]) *simp-all*

lemma *map-nth*:
map (*nth xs*) [*0*..<*length xs*] = *xs*
by (*rule nth-equalityI*, *auto*)

lemma *map-nth-upt*: **assumes** *n* ≤ *length xs*
shows *map* (*nth xs*) [*m*..<*n*] = *take* (*n*−*m*) (*drop m xs*)
proof −
have *map* (*nth xs*) [*m*..<*n*] = *take* (*n*−*m*) (*drop m* (*map* (*nth xs*) [*0*..<*length xs*]))
using *assms take-map*[*of n* (!) *xs*] *drop-take*[*of m n map* ((!) *xs*) [*0*..<*length xs*]]
by(*simp add*: *drop-map*)
thus *?thesis* **by** (*metis map-nth*)
qed

corollary *map-nth-upt0*: *n* ≤ *length xs* ⟹ *map* (*nth xs*) [*0*..<*n*] = *take n xs*
by(*simp add*: *map-nth-upt*)

lemma *list-all2-antisym*:
⟦ (⋀*x y*. ⟦*P x y*; *Q y x*⟧ ⟹ *x* = *y*); *list-all2 P xs ys*; *list-all2 Q ys xs* ⟧

$\Longrightarrow$ *xs = ys*
by (*simp add*: *list-all2-conv-all-nth nth-equalityI*)

lemma *take-equalityI*: (∀ *i*. *take i xs* = *take i ys*) $\Longrightarrow$ *xs* = *ys*
— The famous take-lemma.
by (*metis length-take min.commute order-refl take-all*)

lemma *take-Cons'*:
take n (*x* # *xs*) = (*if n* = *0 then* [] *else x* # *take* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *drop-Cons'*:
drop n (*x* # *xs*) = (*if n* = *0 then x* # *xs else drop* (*n* − *1*) *xs*)
by (*cases n*) *simp-all*

lemma *nth-Cons'*: (*x* # *xs*)!*n* = (*if n* = *0 then x else xs*!(*n* − *1*))
by (*cases n*) *simp-all*

lemma *take-Cons-numeral* [*simp*]:
take (*numeral v*) (*x* # *xs*) = *x* # *take* (*numeral v* − *1*) *xs*
by (*simp add*: *take-Cons'*)

lemma *drop-Cons-numeral* [*simp*]:
drop (*numeral v*) (*x* # *xs*) = *drop* (*numeral v* − *1*) *xs*
by (*simp add*: *drop-Cons'*)

lemma *nth-Cons-numeral* [*simp*]:
(*x* # *xs*) ! *numeral v* = *xs* ! (*numeral v* − *1*)
by (*simp add*: *nth-Cons'*)

lemma *map-upt-eqI*:
‹*map f* [*m*..<*n*] = *xs*› **if** ‹*length xs* = *n* − *m*›
‹⋀*i*. *i* < *length xs* $\Longrightarrow$ *xs* ! *i* = *f* (*m* + *i*)›
proof (*rule nth-equalityI*)
from ‹*length xs* = *n* − *m*› **show** ‹*length* (*map f* [*m*..<*n*]) = *length xs*›
by *simp*
next
fix *i*
assume ‹*i* < *length* (*map f* [*m*..<*n*])›
then have ‹*i* < *n* − *m*›
by *simp*
with *that* **have** ‹*xs* ! *i* = *f* (*m* + *i*)›
by *simp*
with ‹*i* < *n* − *m*› **show** ‹*map f* [*m*..<*n*] ! *i* = *xs* ! *i*›
by *simp*
qed

66.1.23 *upto*: interval-list on *int*

function *upto* :: *int* ⇒ *int* ⇒ *int list* (‹(‹*indent=1 notation=‹mixfix list interval›*›[-../-])›) **where**
  upto i j = (*if i* ≤ *j then i* # [*i+1..j*] *else* [])
by *auto*
termination
by(*relation measure*(%(*i*::*int*,*j*). *nat*(*j* − *i* + *1*))) *auto*

declare *upto.simps*[*simp del*]

lemmas *upto-rec-numeral* [*simp*] =
  upto.simps[*of numeral m numeral n*]
  upto.simps[*of numeral m* − *numeral n*]
  upto.simps[*of* − *numeral m numeral n*]
  upto.simps[*of* − *numeral m* − *numeral n*] **for** *m n*

lemma *upto-empty*[*simp*]: *j* < *i* ⟹ [*i..j*] = []
by(*simp add*: *upto.simps*)

lemma *upto-single*[*simp*]: [*i..i*] = [*i*]
by(*simp add*: *upto.simps*)

lemma *upto-Nil*[*simp*]: [*i..j*] = [] ⟷ *j* < *i*
by (*simp add*: *upto.simps*)

lemmas *upto-Nil2*[*simp*] = *upto-Nil*[*THEN eq-iff-swap*]

lemma *upto-rec1*: *i* ≤ *j* ⟹ [*i..j*] = *i*#[*i+1..j*]
by(*simp add*: *upto.simps*)

lemma *upto-rec2*: *i* ≤ *j* ⟹ [*i..j*] = [*i..j* − *1*]@[*j*]
proof(*induct nat*(*j*−*i*) *arbitrary*: *i j*)
  case *0* **thus** *?case* **by**(*simp add*: *upto.simps*)
next
  case (*Suc n*)
  hence *n* = *nat* (*j* − (*i* + *1*)) *i* < *j* **by** *linarith*+
  from *this*(*2*) *Suc.hyps*(*1*)[*OF this*(*1*)] *Suc*(*2*,*3*) *upto-rec1* **show** *?case* **by** *simp*
qed

lemma *length-upto*[*simp*]: *length* [*i..j*] = *nat*(*j* − *i* + *1*)
by(*induction i j rule*: *upto.induct*) (*auto simp*: *upto.simps*)

lemma *set-upto*[*simp*]: *set*[*i..j*] = {*i..j*}
proof(*induct i j rule*:*upto.induct*)
  case (*1 i j*)
  from *this* **show** *?case*
    unfolding *upto.simps*[*of i j*] **by** *auto*
qed

lemma *nth-upto*[*simp*]: $i + int\ k \leq j \Longrightarrow [i..j]\ !\ k = i + int\ k$
proof(*induction i j arbitrary*: *k rule*: *upto.induct*)
case (*1 i j*)
then show *?case*
by (*auto simp add*: *upto-rec1* [*of i j*] *nth-Cons'*)
qed

lemma *upto-split1*:
$i \leq j \Longrightarrow j \leq k \Longrightarrow [i..k] = [i..j-1]\ @\ [j..k]$
proof (*induction j rule*: *int-ge-induct*)
case *base* **thus** *?case* **by** (*simp add*: *upto-rec1*)
next
case *step* **thus** *?case* **using** *upto-rec1 upto-rec2* **by** *simp*
qed

lemma *upto-split2*:
$i \leq j \Longrightarrow j \leq k \Longrightarrow [i..k] = [i..j]\ @\ [j+1..k]$
using *upto-rec1 upto-rec2 upto-split1* **by** *auto*

lemma *upto-split3*: $[\![\ i \leq j; j \leq k\ ]\!] \Longrightarrow [i..k] = [i..j-1]\ @\ j\ \#\ [j+1..k]$
using *upto-rec1 upto-split1* **by** *auto*

Tail recursive version for code generation:

definition *upto-aux* :: *int* ⇒ *int* ⇒ *int list* ⇒ *int list* **where**
upto-aux i j js = [*i..j*] @ *js*

lemma *upto-aux-rec* [*code*]:
upto-aux i j js = (*if j*<*i then js else upto-aux i* (*j* − *1*) (*j*#*js*))
by (*simp add*: *upto-aux-def upto-rec2*)

lemma *upto-code*[*code*]: [*i..j*] = *upto-aux i j* []
by(*simp add*: *upto-aux-def*)

66.1.24 *successively*

lemma *successively-Cons*:
successively P (*x* # *xs*) ⟷ *xs* = [] ∨ *P x* (*hd xs*) ∧ *successively P xs*
by (*cases xs*) *auto*

lemma *successively-cong* [*cong*]:
assumes ⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *P x y* ⟷ *Q x y xs* = *ys*
shows *successively P xs* ⟷ *successively Q ys*
unfolding *assms*(*2*) [*symmetric*] **using** *assms*(*1*)
by (*induction xs*) (*auto simp*: *successively-Cons*)

lemma *successively-append-iff*:
successively P (*xs* @ *ys*) ⟷
successively P xs ∧ *successively P ys* ∧

```
    (xs = [] ∨ ys = [] ∨ P (last xs) (hd ys))
by (induction xs) (auto simp: successively-Cons)

lemma successively-if-sorted-wrt: sorted-wrt P xs ⟹ successively P xs
by (induction xs rule: induct-list012) auto


lemma successively-iff-sorted-wrt-strong:
  assumes ⋀x y z. x ∈ set xs ⟹ y ∈ set xs ⟹ z ∈ set xs ⟹
                  P x y ⟹ P y z ⟹ P x z
  shows   successively P xs ⟷ sorted-wrt P xs
proof
  assume successively P xs
  from this and assms show sorted-wrt P xs
  proof (induction xs rule: induct-list012)
    case (3 x y xs)
    from 3.prems have P x y
      by auto
    have IH: sorted-wrt P (y # xs)
      using 3.prems
      by(intro 3.IH(2) list.set-intros(2))(simp, blast intro: list.set-intros(2))
    have P x z if asm: z ∈ set xs for z
    proof −
      from IH and asm have P y z
        by auto
      with ‹P x y› show P x z
        using 3.prems asm by auto
    qed
    with IH and ‹P x y› show ?case by auto
  qed auto
qed (use successively-if-sorted-wrt in blast)

lemma successively-conv-sorted-wrt:
  assumes transp P
  shows   successively P xs ⟷ sorted-wrt P xs
  using assms unfolding transp-def
  by (intro successively-iff-sorted-wrt-strong) blast

lemma successively-rev [simp]: successively P (rev xs) ⟷ successively (λx y. P y
x) xs
  by (induction xs rule: remdups-adj.induct)
     (auto simp: successively-append-iff successively-Cons)

lemma successively-map: successively P (map f xs) ⟷ successively (λx y. P (f
x) (f y)) xs
  by (induction xs rule: induct-list012) auto

lemma successively-mono:
  assumes successively P xs
```

assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
shows *successively Q xs*
using *assms* **by** (*induction Q xs rule*: *successively.induct*) *auto*

lemma *successively-altdef*:
successively = (λP. *rec-list True* ($\lambda x\ xs\ b$. *case xs of* [] ⇒ *True* | *y* # - ⇒ *P x y* ∧ *b*))
proof (*intro ext*)
fix *P* **and** *xs* :: *'a list*
show *successively P xs* = *rec-list True* ($\lambda x\ xs\ b$. *case xs of* [] ⇒ *True* | *y* # - ⇒ *P x y* ∧ *b*) *xs*
by (*induction xs*) (*auto simp*: *successively-Cons split*: *list.splits*)
qed

66.1.25 *distinct* and *remdups* and *remdups-adj*

lemma *distinct-tl*: *distinct xs* ⟹ *distinct* (*tl xs*)
by (*cases xs*) *simp-all*

lemma *distinct-append* [*simp*]:
distinct (*xs* @ *ys*) = (*distinct xs* ∧ *distinct ys* ∧ *set xs* ∩ *set ys* = {})
by (*induct xs*) *auto*

lemma *distinct-rev*[*simp*]: *distinct*(*rev xs*) = *distinct xs*
by(*induct xs*) *auto*

lemma *set-remdups* [*simp*]: *set* (*remdups xs*) = *set xs*
by (*induct xs*) (*auto simp add*: *insert-absorb*)

lemma *distinct-remdups* [*iff*]: *distinct* (*remdups xs*)
by (*induct xs*) *auto*

lemma *distinct-remdups-id*: *distinct xs* ⟹ *remdups xs* = *xs*
by (*induct xs*, *auto*)

lemma *remdups-id-iff-distinct* [*simp*]: *remdups xs* = *xs* ⟷ *distinct xs*
by (*metis distinct-remdups distinct-remdups-id*)

lemma *finite-distinct-list*: *finite A* ⟹ ∃ *xs*. *set xs* = *A* ∧ *distinct xs*
by (*metis distinct-remdups finite-list set-remdups*)

lemma *remdups-eq-nil-iff* [*simp*]: (*remdups x* = []) = (*x* = [])
by (*induct x*, *auto*)

lemmas *remdups-eq-nil-right-iff* [*simp*] = *remdups-eq-nil-iff*[*THEN eq-iff-swap*]

lemma *length-remdups-leq*[*iff*]: *length*(*remdups xs*) ≤ *length xs*
by (*induct xs*) *auto*

lemma *length-remdups-eq*[*iff*]:
  (*length* (*remdups xs*) = *length xs*) = (*remdups xs* = *xs*)
proof (*induct xs*)
  case (*Cons a xs*)
  then show *?case*
    by *simp* (*metis Suc-n-not-le-n impossible-Cons length-remdups-leq*)
qed *auto*

lemma *remdups-filter*: *remdups*(*filter P xs*) = *filter P* (*remdups xs*)
by (*induct xs*) *auto*

lemma *distinct-map*:
  distinct(*map f xs*) = (*distinct xs* ∧ *inj-on f* (*set xs*))
by (*induct xs*) *auto*

lemma *distinct-map-filter*:
  distinct (*map f xs*) ⟹ *distinct* (*map f* (*filter P xs*))
by (*induct xs*) *auto*

lemma *distinct-filter* [*simp*]: *distinct xs* ⟹ *distinct* (*filter P xs*)
by (*induct xs*) *auto*

lemma *distinct-upt*[*simp*]: *distinct*[*i*..<*j*]
by (*induct j*) *auto*

lemma *distinct-upto*[*simp*]: *distinct*[*i*..*j*]
proof (*induction i j rule*: *upto.induct*)
  case (*1 i j*)
  then show *?case*
    by (*simp add*: *upto.simps* [*of i*])
qed

lemma *distinct-take*[*simp*]: *distinct xs* ⟹ *distinct* (*take i xs*)
proof (*induct xs arbitrary*: *i*)
  case (*Cons a xs*)
  then show *?case*
    by (*metis Cons.prems append-take-drop-id distinct-append*)
qed *auto*

lemma *distinct-drop*[*simp*]: *distinct xs* ⟹ *distinct* (*drop i xs*)
proof (*induct xs arbitrary*: *i*)
  case (*Cons a xs*)
  then show *?case*
    by (*metis Cons.prems append-take-drop-id distinct-append*)
qed *auto*

lemma *distinct-list-update*:
  assumes *d*: *distinct xs* **and** *a*: *a* ∉ *set xs* − {*xs*!*i*}
  shows *distinct* (*xs*[*i*:=*a*])

proof (*cases i < length xs*)
case *True*
with *a* **have** *anot*: *a* ∉ *set* (*take i xs* @ *xs* ! *i* # *drop* (*Suc i*) *xs*) − {*xs*!*i*}
by *simp* (*metis in-set-dropD in-set-takeD*)
show *?thesis*
proof (*cases a* = *xs*!*i*)
case *True*
with *d* **show** *?thesis*
by *auto*
next
case *False*
have *set* (*take i xs*) ∩ *set* (*drop* (*Suc i*) *xs*) = {}
by (*metis True d disjoint-insert*(*1*) *distinct-append id-take-nth-drop list.set*(*2*))
then show *?thesis*
using *d False anot* ‹*i* < *length xs*› **by** (*simp add*: *upd-conv-take-nth-drop*)
qed
next
case *False* **with** *d* **show** *?thesis* **by** (*auto simp*: *list-update-beyond*)
qed

lemma *distinct-concat-rev*[*simp*]: *distinct* (*concat* (*rev xs*)) = *distinct* (*concat xs*)
by (*induction xs*) *auto*

lemma *distinct-concat*:
⟦ *distinct xs*;
⋀ *ys*. *ys* ∈ *set xs* ⟹ *distinct ys*;
⋀ *ys zs*. ⟦ *ys* ∈ *set xs* ; *zs* ∈ *set xs* ; *ys* ≠ *zs* ⟧ ⟹ *set ys* ∩ *set zs* = {}
⟧ ⟹ *distinct* (*concat xs*)
by (*induct xs*) *auto*

An iff-version of ⟦*distinct ?xs*; ⋀*ys*. *ys* ∈ *set ?xs* ⟹ *distinct ys*; ⋀*ys zs*. ⟦*ys* ∈ *set ?xs*; *zs* ∈ *set ?xs*; *ys* ≠ *zs*⟧ ⟹ *set ys* ∩ *set zs* = {}⟧ ⟹ *distinct* (*concat ?xs*) is available further down as *distinct-concat-iff*.

It is best to avoid the following indexed version of distinct, but sometimes it is useful.

lemma *distinct-conv-nth*: *distinct xs* = (∀ *i* < *size xs*. ∀ *j* < *size xs*. *i* ≠ *j* ⟶ *xs*!*i* ≠ *xs*!*j*)
proof (*induct xs*)
case (*Cons x xs*)
show *?case*
apply (*auto simp add*: *Cons nth-Cons less-Suc-eq-le split*: *nat.split-asm*)
apply (*metis Suc-leI in-set-conv-nth length-pos-if-in-set lessI less-imp-le-nat less-nat-zero-code*)
apply (*metis Suc-le-eq*)
done
qed *auto*

lemma *nth-eq-iff-index-eq*:

⟦ *distinct xs*; *i* < *length xs*; *j* < *length xs* ⟧ ⟹ (*xs*!*i* = *xs*!*j*) = (*i* = *j*)
by(*auto simp*: *distinct-conv-nth*)

lemma *distinct-Ex1*:
distinct xs ⟹ *x* ∈ *set xs* ⟹ (∃!*i*. *i* < *length xs* ∧ *xs* ! *i* = *x*)
by (*auto simp*: *in-set-conv-nth nth-eq-iff-index-eq*)

lemma *inj-on-nth*: *distinct xs* ⟹ ∀ *i* ∈ *I*. *i* < *length xs* ⟹ *inj-on* (*nth xs*) *I*
by (*rule inj-onI*) (*simp add*: *nth-eq-iff-index-eq*)

lemma *bij-betw-nth*:
assumes *distinct xs A* = {..<*length xs*} *B* = *set xs*
shows *bij-betw* ((!) *xs*) *A B*
using *assms* **unfolding** *bij-betw-def*
by (*auto intro*!: *inj-on-nth simp*: *set-conv-nth*)

lemma *set-update-distinct*: ⟦ *distinct xs*; *n* < *length xs* ⟧ ⟹
set(*xs*[*n* := *x*]) = *insert x* (*set xs* − {*xs*!*n*})
by(*auto simp*: *set-eq-iff in-set-conv-nth nth-list-update nth-eq-iff-index-eq*)

lemma *distinct-swap*[*simp*]: ⟦ *i* < *size xs*; *j* < *size xs*⟧ ⟹
distinct(*xs*[*i* := *xs*!*j*, *j* := *xs*!*i*]) = *distinct xs*
by (*smt* (*verit*, *del-insts*) *distinct-conv-nth length-list-update nth-list-update*)

lemma *set-swap*[*simp*]:
⟦ *i* < *size xs*; *j* < *size xs* ⟧ ⟹ *set*(*xs*[*i* := *xs*!*j*, *j* := *xs*!*i*]) = *set xs*
by(*simp add*: *set-conv-nth nth-list-update*) *metis*

lemma *distinct-card*: *distinct xs* ⟹ *card* (*set xs*) = *size xs*
by (*induct xs*) *auto*

lemma *card-distinct*: *card* (*set xs*) = *size xs* ⟹ *distinct xs*
proof (*induct xs*)
case (*Cons x xs*)
show *?case*
proof (*cases x* ∈ *set xs*)
case *False* **with** *Cons* **show** *?thesis* **by** *simp*
next
case *True* **with** *Cons.prems*
have *card* (*set xs*) = *Suc* (*length xs*)
by (*simp add*: *card-insert-if split*: *if-split-asm*)
moreover have *card* (*set xs*) ≤ *length xs* **by** (*rule card-length*)
ultimately have *False* **by** *simp*
thus *?thesis* **..**
qed
qed *simp*

lemma *distinct-length-filter*: *distinct xs* ⟹ *length* (*filter P xs*) = *card* ({*x*. *P x*}
Int set xs)

by (*induct xs*) (*auto*)

lemma *not-distinct-decomp*: ¬ *distinct ws* ⟹ ∃ *xs ys zs y*. *ws* = *xs*@[*y*]@*ys*@[*y*]@*zs*
proof (*induct n* == *length ws arbitrary*:*ws*)
case (*Suc n ws*)
then show *?case*
using *length-Suc-conv* [*of ws n*]
apply (*auto simp*: *eq-commute*)
apply (*metis append-Nil in-set-conv-decomp-first*)
by (*metis append-Cons*)
qed *simp*

lemma *not-distinct-conv-prefix*:
defines *dec as xs y ys* ≡ *y* ∈ *set xs* ∧ *distinct xs* ∧ *as* = *xs* @ *y* # *ys*
shows ¬*distinct as* ⟷ (∃ *xs y ys*. *dec as xs y ys*) (**is** *?L* = *?R*)
proof
assume *?L* **then show** *?R*
proof (*induct length as arbitrary*: *as rule*: *less-induct*)
case *less*
obtain *xs ys zs y* **where** *decomp*: *as* = (*xs* @ *y* # *ys*) @ *y* # *zs*
using *not-distinct-decomp*[*OF less.prems*] **by** *auto*
show *?case*
proof (*cases distinct* (*xs* @ *y* # *ys*))
case *True*
with *decomp* **have** *dec as* (*xs* @ *y* # *ys*) *y zs* **by** (*simp add*: *dec-def*)
then show *?thesis* **by** *blast*
next
case *False*
with *less decomp* **obtain** *xs′ y′ ys′* **where** *dec* (*xs* @ *y* # *ys*) *xs′ y′ ys′*
by *atomize-elim auto*
with *decomp* **have** *dec as xs′ y′* (*ys′* @ *y* # *zs*) **by** (*simp add*: *dec-def*)
then show *?thesis* **by** *blast*
qed
qed
qed (*auto simp*: *dec-def*)

lemma *distinct-product*:
distinct xs ⟹ *distinct ys* ⟹ *distinct* (*List.product xs ys*)
by (*induct xs*) (*auto intro*: *inj-onI simp add*: *distinct-map*)

lemma *distinct-product-lists*:
assumes ∀ *xs* ∈ *set xss*. *distinct xs*
shows *distinct* (*product-lists xss*)
using *assms* **proof** (*induction xss*)
case (*Cons xs xss*) **note** ∗ = *this*
then show *?case*
proof (*cases product-lists xss*)
case *Nil* **then show** *?thesis* **by** (*induct xs*) *simp-all*
next

```
    case (Cons ps pss) with ∗ show ?thesis
      by (auto intro!: inj-onI distinct-concat simp add: distinct-map)
  qed
qed simp

lemma length-remdups-concat:
  length (remdups (concat xss)) = card (⋃ xs∈set xss. set xs)
by (simp add: distinct-card [symmetric])

lemma remdups-append2:
  remdups (xs @ remdups ys) = remdups (xs @ ys)
by(induction xs) auto

lemma length-remdups-card-conv: length(remdups xs) = card(set xs)
proof −
  have xs: concat[xs] = xs by simp
  from length-remdups-concat[of [xs]] show ?thesis unfolding xs by simp
qed

lemma remdups-remdups: remdups (remdups xs) = remdups xs
by (induct xs) simp-all

lemma distinct-butlast:
  assumes distinct xs
  shows distinct (butlast xs)
proof (cases xs = [])
  case False
    from ‹xs ≠ []› obtain ys y where xs = ys @ [y] by (cases xs rule: rev-cases)
auto
    with ‹distinct xs› show ?thesis by simp
qed (auto)

lemma remdups-map-remdups:
  remdups (map f (remdups xs)) = remdups (map f xs)
by (induct xs) simp-all

lemma distinct-zipI1:
  assumes distinct xs
  shows distinct (zip xs ys)
proof (rule zip-obtain-same-length)
  fix xs′ :: 'a list and ys′ :: 'b list and n
  assume length xs′ = length ys′
  assume xs′ = take n xs
  with assms have distinct xs′ by simp
  with ‹length xs′ = length ys′› show distinct (zip xs′ ys′)
    by (induct xs′ ys′ rule: list-induct2) (auto elim: in-set-zipE)
qed

lemma distinct-zipI2:
```

assumes *distinct ys*
shows *distinct* (*zip xs ys*)
proof (*rule zip-obtain-same-length*)
fix *xs'* :: *'b list* **and** *ys'* :: *'a list* **and** *n*
assume *length xs' = length ys'*
assume *ys' = take n ys*
with *assms* **have** *distinct ys'* **by** *simp*
with ‹*length xs' = length ys'*› **show** *distinct* (*zip xs' ys'*)
by (*induct xs' ys' rule*: *list-induct2*) (*auto elim*: *in-set-zipE*)
qed

lemma *set-take-disj-set-drop-if-distinct*:
distinct vs ⟹ *i* ≤ *j* ⟹ *set* (*take i vs*) ∩ *set* (*drop j vs*) = {}
by (*auto simp*: *in-set-conv-nth distinct-conv-nth*)

lemma *distinct-singleton*: *distinct* [*x*] **by** *simp*

lemma *distinct-length-2-or-more*:
distinct (*a* # *b* # *xs*) ⟷ (*a* ≠ *b* ∧ *distinct* (*a* # *xs*) ∧ *distinct* (*b* # *xs*))
by *force*

lemma *remdups-adj-altdef*: (*remdups-adj xs = ys*) ⟷
(∃*f*::*nat* => *nat*. *mono f* ∧ *f* ‘ {*0* ..< *size xs*} = {*0* ..< *size ys*}
∧ (∀ *i* < *size xs*. *xs*!*i* = *ys*!(*f i*))
∧ (∀ *i*. *i* + *1* < *size xs* ⟶ (*xs*!*i* = *xs*!(*i*+*1*) ⟷ *f i* = *f*(*i*+*1*)))) (**is** *?L* ⟷
(∃*f*. *?p f xs ys*))
proof
assume *?L*
then show ∃*f*. *?p f xs ys*
proof (*induct xs arbitrary*: *ys rule*: *remdups-adj.induct*)
case (*1 ys*)
thus *?case* **by** (*intro exI*[*of* - *id*]) (*auto simp*: *mono-def*)
next
case (*2 x ys*)
thus *?case* **by** (*intro exI*[*of* - *id*]) (*auto simp*: *mono-def*)
next
case (*3 x1 x2 xs ys*)
let *?xs* = *x1* # *x2* # *xs*
let *?cond* = *x1* = *x2*
define *zs* **where** *zs* = *remdups-adj* (*x2* # *xs*)
from *3*(*1*−*2*)[*of zs*]
obtain *f* **where** *p*: *?p f* (*x2* # *xs*) *zs* **unfolding** *zs-def* **by** (*cases ?cond*) *auto*
then have *f0*: *f 0* = *0*
by (*intro mono-image-least*[**where** *f*=*f*]) *blast*+
from *p* **have** *mono*: *mono f* **and** *f-xs-zs*: *f* ‘ {*0*..<*length* (*x2* # *xs*)} = {*0*..<*length zs*} **by** *auto*
have *ys*: *ys* = (*if x1* = *x2 then zs else x1* # *zs*)

```
      unfolding 3(3)[symmetric] zs-def by auto
    have zs0: zs ! 0 = x2 unfolding zs-def by (induct xs) auto
    have zsne: zs ≠ [] unfolding zs-def by (induct xs) auto
    let ?Succ = if ?cond then id else Suc
    let ?x1 = if ?cond then id else Cons x1
    let ?f = λ i. if i = 0 then 0 else ?Succ (f (i − 1))
    have ys: ys = ?x1 zs unfolding ys by (cases ?cond, auto)
    have mono: mono ?f using ‹mono f› unfolding mono-def by auto
    show ?case unfolding ys
    proof (intro exI[of - ?f] conjI allI impI)
      show mono ?f by fact
    next
      fix i assume i: i < length ?xs
      with p show ?xs ! i = ?x1 zs ! (?f i) using zs0 by auto
    next
      fix i assume i: i + 1 < length ?xs
      with p show (?xs ! i = ?xs ! (i + 1)) = (?f i = ?f (i + 1))
        by (cases i) (auto simp: f0)
    next
      have id: {0 ..< length (?x1 zs)} = insert 0 (?Succ ‘ {0 ..< length zs})
        using zsne by (cases ?cond, auto)
      { fix i  assume i < Suc (length xs)
        hence Suc i ∈ {0..<Suc (Suc (length xs))} ∩ Collect ((<) 0) by auto
        from imageI[OF this, of λi. ?Succ (f (i − Suc 0))]
         have ?Succ (f i) ∈ (λi. ?Succ (f (i − Suc 0))) ‘ ({0..<Suc (Suc (length
xs))} ∩ Collect ((<) 0)) by auto
      }
      then show ?f ‘ {0 ..< length ?xs} = {0 ..< length (?x1  zs)}
        unfolding id f-xs-zs[symmetric] by auto
    qed
  qed
next
  assume ∃ f. ?p f xs ys
  then show ?L
  proof (induct xs arbitrary: ys rule: remdups-adj.induct)
    case 1 then show ?case by auto
  next
    case (2 x) then obtain f where f-img: f ‘ {0 ..< size [x]} = {0 ..< size ys}
       and f-nth: ⋀i. i < size [x] ⟹ [x]!i = ys!(f i)
      by blast

    have length ys = card (f ‘ {0 ..< size [x]})
      using f-img by auto
    then have ∗: length ys = 1 by auto
    then have f 0 = 0 using f-img by auto
    with ∗ show ?case using f-nth by (cases ys) auto
  next
    case (3 x1 x2 xs)
    from 3.prems obtain f where f-mono: mono f
```

```
    and f-img: f ` {0..<length (x1 # x2 # xs)} = {0..<length ys}
    and f-nth:
      ⋀i. i < length (x1 # x2 # xs) ⟹ (x1 # x2 # xs) ! i = ys ! f i
      ⋀i. i + 1 < length (x1 # x2 #xs) ⟹
        ((x1 # x2 # xs) ! i = (x1 # x2 # xs) ! (i + 1)) = (f i = f (i + 1))
    by blast

  show ?case
  proof cases
    assume x1 = x2

    let ?f' = f ∘ Suc

    have remdups-adj (x1 # xs) = ys
    proof (intro 3.hyps exI conjI impI allI)
      show mono ?f'
        using f-mono by (simp add: mono-iff-le-Suc)
    next
      have ?f' ` {0 ..< length (x1 # xs)} = f ` {Suc 0 ..< length (x1 # x2 #
xs)}
        using less-Suc-eq-0-disj by auto
      also have ... = f ` {0 ..< length (x1 # x2 # xs)}
      proof −
        have f 0 = f (Suc 0) using ‹x1 = x2› f-nth[of 0] by simp
        then show ?thesis
          using less-Suc-eq-0-disj by auto
      qed
      also have ... = {0 ..< length ys} by fact
      finally show ?f' ` {0 ..< length (x1 # xs)} = {0 ..< length ys} .
    qed (insert f-nth[of Suc i for i], auto simp: ‹x1 = x2›)
    then show ?thesis using ‹x1 = x2› by simp
  next
    assume x1 ≠ x2

    have two: Suc (Suc 0) ≤ length ys
    proof −
      have 2 = card {f 0, f 1} using ‹x1 ≠ x2› f-nth[of 0] by auto
      also have ... ≤ card (f ` {0..< length (x1 # x2 # xs)})
        by (rule card-mono) auto
      finally show ?thesis using f-img by simp
    qed

    have f 0 = 0 using f-mono f-img by (rule mono-image-least) simp

    have f (Suc 0) = Suc 0
    proof (rule ccontr)
      assume f (Suc 0) ≠ Suc 0
      then have Suc 0 < f (Suc 0) using f-nth[of 0] ‹x1 ≠ x2› ‹f 0 = 0› by
auto
```

```
      then have ⋀i. Suc 0 < f (Suc i)
        using f-mono
        by (meson Suc-le-mono le0 less-le-trans monoD)
      then have Suc 0 ≠ f i for i using ‹f 0 = 0›
        by (cases i) fastforce+
      then have Suc 0 ∉ f ‘ {0 ..< length (x1 # x2 # xs)} by auto
      then show False using f-img two by auto
    qed
    obtain ys' where ys = x1 # x2 # ys'
      using two f-nth[of 0] f-nth[of 1]
      by (auto simp: Suc-le-length-iff ‹f 0 = 0› ‹f (Suc 0) = Suc 0›)

    have Suc0-le-f-Suc: Suc 0 ≤ f (Suc i) for i
      by (metis Suc-le-mono ‹f (Suc 0) = Suc 0› f-mono le0 mono-def)

    define f' where f' x = f (Suc x) - 1 for x
    have f-Suc: f (Suc i) = Suc (f' i) for i
       using Suc0-le-f-Suc[of i] by (auto simp: f'-def)

    have remdups-adj (x2 # xs) = (x2 # ys')
    proof (intro 3.hyps exI conjI impI allI)
      show mono f'
       using Suc0-le-f-Suc f-mono by (auto simp: f'-def mono-iff-le-Suc le-diff-iff)
    next
      have f' ‘ {0 ..< length (x2 # xs)} = (λx. f x - 1) ‘ {0 ..< length (x1 #
x2 #xs)}
        by (auto simp: f'-def ‹f 0 = 0› ‹f (Suc 0) = Suc 0› image-def Bex-def
less-Suc-eq-0-disj)
      also have ... = (λx. x - 1) ‘ f ‘ {0 ..< length (x1 # x2 #xs)}
        by (auto simp: image-comp)
      also have ... = (λx. x - 1) ‘ {0 ..< length ys}
        by (simp only: f-img)
      also have ... = {0 ..< length (x2 # ys')}
        using ‹ys = -› by (fastforce intro: rev-image-eqI)
      finally show f' ‘ {0 ..< length (x2 # xs)} = {0 ..< length (x2 # ys')} .
    qed (insert f-nth[of Suc i for i] ‹x1 ≠ x2›, auto simp add: f-Suc ‹ys = -›)
    then show ?case using ‹ys = -› ‹x1 ≠ x2› by simp
  qed
 qed
qed

lemma hd-remdups-adj[simp]: hd (remdups-adj xs) = hd xs
 by (induction xs rule: remdups-adj.induct) simp-all

lemma remdups-adj-Cons: remdups-adj (x # xs) =
 (case remdups-adj xs of [] ⇒ [x] | y # xs ⇒ if x = y then y # xs else x # y #
xs)
 by (induct xs arbitrary: x) (auto split: list.splits)
```

lemma *remdups-adj-append-two*:
remdups-adj (*xs* @ [*x*,*y*]) = *remdups-adj* (*xs* @ [*x*]) @ (*if x* = *y then* [] *else* [*y*])
by (*induct xs rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-adjacent*:
Suc i < *length* (*remdups-adj xs*) ⟹ *remdups-adj xs* ! *i* ≠ *remdups-adj xs* ! *Suc i*
proof (*induction xs arbitrary*: *i rule*: *remdups-adj.induct*)
case (*3 x y xs i*)
thus *?case* **by** (*cases i*, *cases x* = *y*) (*simp*, *auto simp*: *hd-conv-nth*[*symmetric*])
qed *simp-all*

lemma *remdups-adj-rev*[*simp*]: *remdups-adj* (*rev xs*) = *rev* (*remdups-adj xs*)
by (*induct xs rule*: *remdups-adj.induct*, *simp-all add*: *remdups-adj-append-two*)

lemma *remdups-adj-length*[*simp*]: *length* (*remdups-adj xs*) ≤ *length xs*
by (*induct xs rule*: *remdups-adj.induct*, *auto*)

lemma *remdups-adj-length-ge1*[*simp*]: *xs* ≠ [] ⟹ *length* (*remdups-adj xs*) ≥ *Suc 0*
by (*induct xs rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-Nil-iff*[*simp*]: *remdups-adj xs* = [] ⟷ *xs* = []
by (*induct xs rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-set*[*simp*]: *set* (*remdups-adj xs*) = *set xs*
by (*induct xs rule*: *remdups-adj.induct*, *simp-all*)

lemma *last-remdups-adj* [*simp*]: *last* (*remdups-adj xs*) = *last xs*
by (*induction xs rule*: *remdups-adj.induct*) *auto*

lemma *remdups-adj-Cons-alt*[*simp*]: *x* # *tl* (*remdups-adj* (*x* # *xs*)) = *remdups-adj* (*x* # *xs*)
by (*induct xs rule*: *remdups-adj.induct*, *auto*)

lemma *remdups-adj-distinct*: *distinct xs* ⟹ *remdups-adj xs* = *xs*
by (*induct xs rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-append*:
remdups-adj (*xs*$_1$ @ *x* # *xs*$_2$) = *remdups-adj* (*xs*$_1$ @ [*x*]) @ *tl* (*remdups-adj* (*x* # *xs*$_2$))
by (*induct xs*$_1$ *rule*: *remdups-adj.induct*, *simp-all*)

lemma *remdups-adj-singleton*:
remdups-adj xs = [*x*] ⟹ *xs* = *replicate* (*length xs*) *x*
by (*induct xs rule*: *remdups-adj.induct*, *auto split*: *if-split-asm*)

lemma *remdups-adj-map-injective*:
assumes *inj f*
shows *remdups-adj* (*map f xs*) = *map f* (*remdups-adj xs*)

by (*induct xs rule*: *remdups-adj.induct*) (*auto simp add*: *injD*[*OF assms*])

lemma *remdups-adj-replicate*:
remdups-adj (*replicate n x*) = (*if n = 0 then* [] *else* [*x*])
by (*induction n*) (*auto simp*: *remdups-adj-Cons*)

lemma *remdups-upt* [*simp*]: *remdups* [*m*..<*n*] = [*m*..<*n*]
proof (*cases m* ≤ *n*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **then obtain** *q* **where** *n* = *m* + *q*
by (*auto simp add*: *le-iff-add*)
moreover have *remdups* [*m*..<*m* + *q*] = [*m*..<*m* + *q*]
by (*induct q*) *simp-all*
ultimately show *?thesis* **by** *simp*
qed

lemma *successively-remdups-adjI*:
successively P xs ⟹ *successively P* (*remdups-adj xs*)
by (*induction xs rule*: *remdups-adj.induct*) (*auto simp*: *successively-Cons*)

lemma *successively-remdups-adj-iff*:
(⋀*x*. *x* ∈ *set xs* ⟹ *P x x*) ⟹
successively P (*remdups-adj xs*) ⟷ *successively P xs*
by (*induction xs rule*: *remdups-adj.induct*)(*auto simp*: *successively-Cons*)

lemma *successively-conv-nth*:
successively P xs ⟷ (∀ *i*. *Suc i* < *length xs* ⟶ *P* (*xs* ! *i*) (*xs* ! *Suc i*))
by (*induction P xs rule*: *successively.induct*)
(*force simp*: *nth-Cons split*: *nat.splits*)+

lemma *successively-nth*: *successively P xs* ⟹ *Suc i* < *length xs* ⟹ *P* (*xs* ! *i*) (*xs* ! *Suc i*)
unfolding *successively-conv-nth* **by** *blast*

lemma *distinct-adj-conv-nth*:
distinct-adj xs ⟷ (∀ *i*. *Suc i* < *length xs* ⟶ *xs* ! *i* ≠ *xs* ! *Suc i*)
by (*simp add*: *distinct-adj-def successively-conv-nth*)

lemma *distinct-adj-nth*: *distinct-adj xs* ⟹ *Suc i* < *length xs* ⟹ *xs* ! *i* ≠ *xs* ! *Suc i*
unfolding *distinct-adj-conv-nth* **by** *blast*

lemma *remdups-adj-Cons'*:
remdups-adj (*x* # *xs*) = *x* # *remdups-adj* (*dropWhile* (λ*y*. *y* = *x*) *xs*)
by (*induction xs*) *auto*

lemma *remdups-adj-singleton-iff*:
length (*remdups-adj xs*) = *Suc 0* ⟷ *xs* ≠ [] ∧ *xs* = *replicate* (*length xs*) (*hd xs*)

proof *safe*
assume ∗: *xs* = *replicate* (*length xs*) (*hd xs*) **and** [*simp*]: *xs* ≠ []
show *length* (*remdups-adj xs*) = *Suc 0*
by (*subst* ∗) (*auto simp*: *remdups-adj-replicate*)
next
assume *length* (*remdups-adj xs*) = *Suc 0*
thus *xs* = *replicate* (*length xs*) (*hd xs*)
by (*induction xs rule*: *remdups-adj.induct*) (*auto split*: *if-splits*)
qed *auto*

lemma *tl-remdups-adj*:
ys ≠ [] ⟹ *tl* (*remdups-adj ys*) = *remdups-adj* (*dropWhile* (λ*x*. *x* = *hd ys*) (*tl ys*))
by (*cases ys*) (*simp-all add*: *remdups-adj-Cons′*)

lemma *remdups-adj-append-dropWhile*:
remdups-adj (*xs* @ *y* # *ys*) = *remdups-adj* (*xs* @ [*y*]) @ *remdups-adj* (*dropWhile* (λ*x*. *x* = *y*) *ys*)
by (*subst remdups-adj-append*) (*simp add*: *tl-remdups-adj*)

lemma *remdups-adj-append′*:
assumes *xs* = [] ∨ *ys* = [] ∨ *last xs* ≠ *hd ys*
shows *remdups-adj* (*xs* @ *ys*) = *remdups-adj xs* @ *remdups-adj ys*
proof −
have *?thesis* **if** [*simp*]: *xs* ≠ [] *ys* ≠ [] **and** *last xs* ≠ *hd ys*
proof −
obtain *x xs′* **where** *xs*: *xs* = *xs′* @ [*x*]
by (*cases xs rule*: *rev-cases*) *auto*
have *remdups-adj* (*xs′* @ *x* # *ys*) = *remdups-adj* (*xs′* @ [*x*]) @ *remdups-adj ys*
using ‹*last xs* ≠ *hd ys*› **unfolding** *xs*
by (*metis* (*full-types*) *dropWhile-eq-self-iff last-snoc remdups-adj-append-dropWhile*)
thus *?thesis* **by** (*simp add*: *xs*)
qed
thus *?thesis* **using** *assms*
by (*cases xs* = []; *cases ys* = []) *auto*
qed

lemma *remdups-adj-append′′*: *xs* ≠ []
⟹ *remdups-adj* (*xs* @ *ys*) = *remdups-adj xs* @ *remdups-adj* (*dropWhile* (λ*y*. *y* = *last xs*) *ys*)
by (*induction xs rule*: *remdups-adj.induct*) (*auto simp*: *remdups-adj-Cons′*)

lemma *remdups-filter-last*:
last [*x*←*remdups xs*. *P x*] = *last* [*x*←*xs*. *P x*]
by (*induction xs*, *auto simp*: *filter-empty-conv*)

lemma *remdups-append*:
set xs ⊆ *set ys* ⟹ *remdups* (*xs* @ *ys*) = *remdups ys*
by (*induction xs*, *simp-all*)

lemma *remdups-concat*:
remdups (*concat* (*remdups xs*)) = *remdups* (*concat xs*)
proof (*induction xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs*)
show *?case*
proof (*cases a* ∈ *set xs*)
case *True*
then have *remdups* (*concat xs*) = *remdups* (*a* @ *concat xs*)
by (*metis remdups-append concat.simps*(*2*) *insert-absorb set-simps*(*2*) *set-append set-concat sup-ge1*)
then show *?thesis*
by (*simp add*: *Cons True*)
next
case *False*
then show *?thesis*
by (*metis Cons remdups-append2 concat.simps*(*2*) *remdups.simps*(*2*))
qed
qed

66.2 *distinct-adj*

lemma *distinct-adj-Nil* [*simp*]: *distinct-adj* []
and *distinct-adj-singleton* [*simp*]: *distinct-adj* [*x*]
and *distinct-adj-Cons-Cons* [*simp*]: *distinct-adj* (*x* # *y* # *xs*) ⟷ *x* ≠ *y* ∧ *distinct-adj* (*y* # *xs*)
by (*auto simp*: *distinct-adj-def*)

lemma *distinct-adj-Cons*: *distinct-adj* (*x* # *xs*) ⟷ *xs* = [] ∨ *x* ≠ *hd xs* ∧ *distinct-adj xs*
by (*cases xs*) *auto*

lemma *distinct-adj-ConsD*: *distinct-adj* (*x* # *xs*) ⟹ *distinct-adj xs*
by (*cases xs*) *auto*

lemma *distinct-adj-remdups-adj*[*simp*]: *distinct-adj* (*remdups-adj xs*)
by (*induction xs rule*: *remdups-adj.induct*) (*auto simp*: *distinct-adj-Cons*)

lemma *distinct-adj-altdef*: *distinct-adj xs* ⟷ *remdups-adj xs* = *xs*
proof
assume *remdups-adj xs* = *xs*
with *distinct-adj-remdups-adj*[*of xs*] **show** *distinct-adj xs*
by *simp*
next
assume *distinct-adj xs*
thus *remdups-adj xs* = *xs*

by (*induction xs rule*: *induct-list012*) *auto*
qed

lemma *distinct-adj-rev* [*simp*]: *distinct-adj* (*rev xs*) ⟷ *distinct-adj xs*
by (*simp add*: *distinct-adj-def eq-commute*)

lemma *distinct-adj-append-iff*:
distinct-adj (*xs* @ *ys*) ⟷
distinct-adj xs ∧ *distinct-adj ys* ∧ (*xs* = [] ∨ *ys* = [] ∨ *last xs* ≠ *hd ys*)
by (*auto simp*: *distinct-adj-def successively-append-iff*)

lemma *distinct-adj-appendD1* [*dest*]: *distinct-adj* (*xs* @ *ys*) ⟹ *distinct-adj xs*
and *distinct-adj-appendD2* [*dest*]: *distinct-adj* (*xs* @ *ys*) ⟹ *distinct-adj ys*
by (*auto simp*: *distinct-adj-append-iff*)

lemma *distinct-adj-mapI*: *distinct-adj xs* ⟹ *inj-on f* (*set xs*) ⟹ *distinct-adj* (*map f xs*)
unfolding *distinct-adj-def successively-map*
by (*erule successively-mono*) (*auto simp*: *inj-on-def*)

lemma *distinct-adj-mapD*: *distinct-adj* (*map f xs*) ⟹ *distinct-adj xs*
unfolding *distinct-adj-def successively-map* **by** (*erule successively-mono*) *auto*

lemma *distinct-adj-map-iff*: *inj-on f* (*set xs*) ⟹ *distinct-adj* (*map f xs*) ⟷ *distinct-adj xs*
using *distinct-adj-mapD distinct-adj-mapI* **by** *blast*

lemma *distinct-adj-conv-length-remdups-adj*:
distinct-adj xs ⟷ *length* (*remdups-adj xs*) = *length xs*
proof (*induction xs rule*: *remdups-adj.induct*)
case (*3 x y xs*)
thus *?case*
using *remdups-adj-length*[*of y # xs*] **by** *auto*
qed *auto*

66.2.1 *insert*

lemma *in-set-insert* [*simp*]:
x ∈ *set xs* ⟹ *List.insert x xs* = *xs*
by (*simp add*: *List.insert-def*)

lemma *not-in-set-insert* [*simp*]:
x ∉ *set xs* ⟹ *List.insert x xs* = *x # xs*
by (*simp add*: *List.insert-def*)

lemma *insert-Nil* [*simp*]: *List.insert x* [] = [*x*]
by *simp*

lemma *set-insert* [*simp*]: *set* (*List.insert x xs*) = *insert x* (*set xs*)

by (*auto simp add*: *List.insert-def*)

lemma *distinct-insert* [*simp*]: *distinct* (*List.insert x xs*) = *distinct xs*
by (*simp add*: *List.insert-def*)

lemma *insert-remdups*:
List.insert x (*remdups xs*) = *remdups* (*List.insert x xs*)
by (*simp add*: *List.insert-def*)

66.2.2 *List.union*

This is all one should need to know about union:

lemma *set-union*[*simp*]: *set* (*List.union xs ys*) = *set xs* ∪ *set ys*
unfolding *List.union-def*
by(*induct xs arbitrary*: *ys*) *simp-all*

lemma *distinct-union*[*simp*]: *distinct*(*List.union xs ys*) = *distinct ys*
unfolding *List.union-def*
by(*induct xs arbitrary*: *ys*) *simp-all*

66.2.3 *find*

lemma *find-None-iff*: *List.find P xs* = *None* ⟷ ¬ (∃ *x*. *x* ∈ *set xs* ∧ *P x*)
proof (*induction xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case* **by** (*fastforce split*: *if-splits*)
qed

lemmas *find-None-iff2* = *find-None-iff*[*THEN eq-iff-swap*]

lemma *find-Some-iff*:
List.find P xs = *Some x* ⟷
(∃ *i*<*length xs*. *P* (*xs*!*i*) ∧ *x* = *xs*!*i* ∧ (∀ *j*<*i*. ¬ *P* (*xs*!*j*)))
proof (*induction xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons x xs*) **thus** *?case*
apply(*auto simp*: *nth-Cons'* *split*: *if-splits*)
using *diff-Suc-1 less-Suc-eq-0-disj* **by** *fastforce*
qed

lemmas *find-Some-iff2* = *find-Some-iff*[*THEN eq-iff-swap*]

lemma *find-cong*[*fundef-cong*]:
assumes *xs* = *ys* **and** ⋀*x*. *x* ∈ *set ys* ⟹ *P x* = *Q x*
shows *List.find P xs* = *List.find Q ys*
proof (*cases List.find P xs*)
case *None* **thus** *?thesis* **by** (*metis find-None-iff assms*)

next
case (*Some x*)
hence *List.find Q ys = Some x* **using** *assms*
by (*auto simp add*: *find-Some-iff*)
thus *?thesis* **using** *Some* **by** *auto*
qed

lemma *find-dropWhile*:
List.find P xs = (*case dropWhile* (*Not* ∘ *P*) *xs*
of [] ⇒ *None*
| *x* # - ⇒ *Some x*)
by (*induct xs*) *simp-all*

66.2.4 *count-list*

This library is intentionally minimal. See the remark about multisets at the point above where *count-list* is defined. However, dragging in multisets can be overkill, so we should not be too minimal here.

lemma *count-list-append*[*simp*]: *count-list* (*xs* @ *ys*) *x* = *count-list xs x* + *count-list ys x*
by (*induction xs*) *simp-all*

lemma *count-list-0-iff*: *count-list xs x* = 0 ⟷ *x* ∉ *set xs*
by (*induction xs*) *simp-all*

lemma *count-notin*[*simp*]: *x* ∉ *set xs* ⟹ *count-list xs x* = 0
by(*simp add*: *count-list-0-iff*)

lemma *count-le-length*: *count-list xs x* ≤ *length xs*
by (*induction xs*) *auto*

lemma *count-list-map-ge*: *count-list xs x* ≤ *count-list* (*map f xs*) (*f x*)
by (*induction xs*) *auto*

lemma *count-list-inj-map*:
⟦ *inj-on f* (*set xs*); *x* ∈ *set xs* ⟧ ⟹ *count-list* (*map f xs*) (*f x*) = *count-list xs x*
by (*induction xs*) (*simp*, *fastforce*)

lemma *count-list-map-conv*:
assumes *inj f* **shows** *count-list* (*map f xs*) (*f x*) = *count-list xs x*
by (*induction xs*) (*simp-all add*: *inj-eq*[*OF assms*])

lemma *count-list-rev*[*simp*]: *count-list* (*rev xs*) *x* = *count-list xs x*
by (*induction xs*) *auto*

lemma *sum-count-set*:
set xs ⊆ *X* ⟹ *finite X* ⟹ *sum* (*count-list xs*) *X* = *length xs*
proof (*induction xs arbitrary*: *X*)
case (*Cons x xs*)

then show *?case*
using *sum.remove* [*of X x count-list xs*]
by (*auto simp*: *sum.If-cases simp flip*: *diff-eq*)
qed *simp*

lemma *count-list-Suc-split-first*:
assumes *count-list xs x* = *Suc n*
shows ∃ *pref rest*. *xs* = *pref* @ *x* # *rest* ∧ *x* ∉ *set pref* ∧ *count-list rest x* = *n*
proof −
let *?pref* = *takeWhile* (λ*u*. *u* ≠ *x*) *xs*
let *?rest* = *drop* (*length ?pref*) *xs*
have *x* ∈ *set xs* **using** *assms count-notin* **by** *fastforce*
hence *rest*: *?rest* ≠ [] ∧ *hd ?rest* = *x*
by (*metis* (*mono-tags*, *lifting*) *append-Nil2 dropWhile-eq-drop hd-dropWhile takeWhile-dropWhile-id takeWhile-eq-all-conv*)
have *1*: *x* ∉ *set ?pref* **by** (*metis* (*full-types*) *set-takeWhileD*)
have *2*: *xs* = *?pref* @ *x* # *tl ?rest*
by (*metis rest append-eq-conv-conj hd-Cons-tl takeWhile-eq-take*)
have *count-list* (*tl ?rest*) *x* = *n*
using *assms rest 1 2 count-notin count-list-append*[*of ?pref x* # *tl ?rest x*] **by** *simp*
with *1 2* **show** *?thesis* **by** *blast*
qed

lemma *count-list-eq-length-filter*: *count-list xs y* = *length*(*filter* ((=) *y*) *xs*)
by (*induction xs*) *auto*

lemma *count-list-replicate*[*simp*]: *count-list* (*replicate n x*) *y* = (*if x* = *y then n else 0*)
by (*induction n*) *auto*

lemma *split-list-cycles*:
∃ *pref xss*. *xs* = *pref* @ *concat xss* ∧ *x* ∉ *set pref* ∧ (∀ *ys* ∈ *set xss*. ∃ *zs*. *ys* = *x* # *zs*)
proof (*induction count-list xs x arbitrary*: *xs*)
case *0*
show *?case* **using** *0*[*symmetric*] *concat.simps*(*1*) *count-list-0-iff* **by** *fastforce*
next
case (*Suc n*)
from *Suc.hyps*(*2*) **obtain** *pref rest* **where**
∗: *xs* = *pref* @ *x* # *rest x* ∉ *set pref count-list rest x* = *n*
by (*metis count-list-Suc-split-first*)
from *Suc.hyps*(*1*)[*OF* ∗(*3*)[*symmetric*]] **obtain** *pref1 xss* **where**
∗∗: *rest* = *pref1* @ *concat xss x* ∉ *set pref1* ∀ *ys*∈*set xss*. ∃ *zs*. *ys* = *x* # *zs*
by *blast*
let *?xss* = (*x* # *pref1*) # *xss*
have *xs* = *pref* @ *concat ?xss* ∧ *x* ∉ *set pref* ∧ (∀ *ys* ∈ *set ?xss*. ∃ *zs*. *ys* = *x* # *zs*)
using ∗(*1*,*2*) ∗∗ **by** *auto*

thus *?case* **by** *blast*
qed

66.2.5 *List.extract*

lemma *extract-None-iff*: *List.extract P xs = None ⟷ ¬ (∃ x∈set xs. P x)*
by(*auto simp*: *extract-def dropWhile-eq-Cons-conv split*: *list.splits*)
(*metis in-set-conv-decomp*)

lemma *extract-SomeE*:
List.extract P xs = Some (ys, y, zs) ⟹
xs = ys @ y # zs ∧ P y ∧ ¬ (∃ y ∈ set ys. P y)
by(*auto simp*: *extract-def dropWhile-eq-Cons-conv split*: *list.splits*)

lemma *extract-Some-iff*:
List.extract P xs = Some (ys, y, zs) ⟷
xs = ys @ y # zs ∧ P y ∧ ¬ (∃ y ∈ set ys. P y)
by(*auto simp*: *extract-def dropWhile-eq-Cons-conv dest*: *set-takeWhileD split*: *list.splits*)

lemma *extract-Nil-code* [*code*]: *List.extract P [] = None*
by(*simp add*: *extract-def*)

lemma *extract-Cons-code* [*code*]:
List.extract P (x # xs) = (if P x then Some ([], x, xs) else
(case List.extract P xs of
None ⇒ None |
Some (ys, y, zs) ⇒ Some (x#ys, y, zs)))
by(*auto simp add*: *extract-def comp-def split*: *list.splits*)
(*metis dropWhile-eq-Nil-conv list.distinct*(*1*))

66.2.6 *remove1*

lemma *count-list-remove1* [*simp*]:
count-list (*remove1 a xs*) *b = count-list xs b − (if a=b then 1 else 0*)
by(*induction xs*) *auto*

lemma *remove1-append*:
remove1 x (*xs @ ys*) =
(*if x ∈ set xs then remove1 x xs @ ys else xs @ remove1 x ys*)
by (*induct xs*) *auto*

lemma *remove1-commute*: *remove1 x* (*remove1 y zs*) = *remove1 y* (*remove1 x zs*)
by (*induct zs*) *auto*

lemma *in-set-remove1* [*simp*]:
a ≠ b ⟹ a ∈ set(*remove1 b xs*) = (*a ∈ set xs*)
by (*induct xs*) *auto*

lemma *set-remove1-subset*: *set*(*remove1 x xs*) ⊆ *set xs*
by (*induct xs*) *auto*

lemma *set-remove1-eq* [*simp*]: *distinct xs* $\Longrightarrow$ *set*(*remove1 x xs*) = *set xs* − {*x*}
by (*induct xs*) *auto*

lemma *length-remove1*:
length(*remove1 x xs*) = (*if x* ∈ *set xs then length xs* − *1 else length xs*)
by (*induct xs*) (*auto dest*!:*length-pos-if-in-set*)

lemma *remove1-filter-not*[*simp*]:
¬ *P x* $\Longrightarrow$ *remove1 x* (*filter P xs*) = *filter P xs*
by(*induct xs*) *auto*

lemma *filter-remove1*:
filter Q (*remove1 x xs*) = *remove1 x* (*filter Q xs*)
by (*induct xs*) *auto*

lemma *notin-set-remove1*[*simp*]: *x* ∉ *set xs* $\Longrightarrow$ *x* ∉ *set*(*remove1 y xs*)
by(*insert set-remove1-subset*) *fast*

lemma *distinct-remove1*[*simp*]: *distinct xs* $\Longrightarrow$ *distinct*(*remove1 x xs*)
by (*induct xs*) *simp-all*

lemma *remove1-remdups*:
distinct xs $\Longrightarrow$ *remove1 x* (*remdups xs*) = *remdups* (*remove1 x xs*)
by (*induct xs*) *simp-all*

lemma *remove1-idem*: *x* ∉ *set xs* $\Longrightarrow$ *remove1 x xs* = *xs*
by (*induct xs*) *simp-all*

lemma *remove1-split*:
a ∈ *set xs* $\Longrightarrow$ *remove1 a xs* = *ys* $\longleftrightarrow$ (∃ *ls rs*. *xs* = *ls* @ *a* # *rs* ∧ *a* ∉ *set ls* ∧ *ys* = *ls* @ *rs*)
by (*metis remove1.simps*(*2*) *remove1-append split-list-first*)

lemma *foldr-fold-remove1*[*code-unfold*]: *foldr remove1* = *fold remove1*
using *foldr-fold*[*of - remove1*] *remove1-commute* **by** *fastforce*

66.2.7 *removeAll*

lemma *removeAll-filter-not-eq*:
removeAll x = *filter* (λ*y*. *x* ≠ *y*)
proof
fix *xs*
show *removeAll x xs* = *filter* (λ*y*. *x* ≠ *y*) *xs*
by (*induct xs*) *auto*
qed

lemma *removeAll-append*[*simp*]:
removeAll x (*xs* @ *ys*) = *removeAll x xs* @ *removeAll x ys*

by (*induct xs*) *auto*

lemma *removeAll-commute*: *removeAll x* (*removeAll y zs*) = *removeAll y* (*removeAll x zs*)
by (*induct zs*) *auto*

lemma *set-removeAll*[*simp*]: *set*(*removeAll x xs*) = *set xs* − {*x*}
by (*induct xs*) *auto*

lemma *removeAll-id*[*simp*]: *x* ∉ *set xs* ⟹ *removeAll x xs* = *xs*
by (*induct xs*) *auto*

lemma *length-removeAll*:
length(*removeAll x xs*) = *length xs* − *count-list xs x*
by(*induction xs*)(*auto simp*: *Suc-diff-le count-le-length*)

lemma *removeAll-filter-not*[*simp*]:
¬ *P x* ⟹ *removeAll x* (*filter P xs*) = *filter P xs*
by(*induct xs*) *auto*

lemma *distinct-removeAll*:
distinct xs ⟹ *distinct* (*removeAll x xs*)
by (*simp add*: *removeAll-filter-not-eq*)

lemma *distinct-remove1-removeAll*:
distinct xs ⟹ *remove1 x xs* = *removeAll x xs*
by (*induct xs*) *simp-all*

lemma *map-removeAll-inj-on*: *inj-on f* (*insert x* (*set xs*)) ⟹
map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
by (*induct xs*) (*simp-all add*:*inj-on-def*)

lemma *map-removeAll-inj*: *inj f* ⟹
map f (*removeAll x xs*) = *removeAll* (*f x*) (*map f xs*)
by (*rule map-removeAll-inj-on*, *erule inj-on-subset*, *rule subset-UNIV*)

lemma *length-removeAll-less-eq* [*simp*]:
length (*removeAll x xs*) ≤ *length xs*
by (*simp add*: *removeAll-filter-not-eq*)

lemma *length-removeAll-less* [*termination-simp*]:
x ∈ *set xs* ⟹ *length* (*removeAll x xs*) < *length xs*
by (*auto dest*: *length-filter-less simp add*: *removeAll-filter-not-eq*)

lemma *distinct-concat-iff*: *distinct* (*concat xs*) ⟷
distinct (*removeAll* [] *xs*) ∧
(∀ *ys*. *ys* ∈ *set xs* ⟶ *distinct ys*) ∧
(∀ *ys zs*. *ys* ∈ *set xs* ∧ *zs* ∈ *set xs* ∧ *ys* ≠ *zs* ⟶ *set ys* ∩ *set zs* = {})
proof (*induct xs*)

case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
have ⟦*set a* ∩ ⋃ (*set ' set xs*) = {}; *a* ∈ *set xs*⟧ ⟹ *a*=[]
by (*metis Int-iff UN-I empty-iff equals0I set-empty*)
then show *?case*
by (*auto simp*: *Cons*)
qed

lemma *foldr-fold-removeAll*[*code-unfold*]: *foldr removeAll* = *fold removeAll*
using *foldr-fold*[*of - removeAll*] *removeAll-commute* **by** *fastforce*

66.2.8 *minus-list-mset*

The difference of two lists viewed as multisets. Conceptually, the result of *minus-list-mset* is only determined up to permutation, i.e. up to the multiset of elements. Thus this function comes into its own in connection with multisets where *mset*(*minus-list-mset xs ys*) = *mset xs* − *mset ys* is proved. Lemma *count-list-minus-list-mset* is the equivalent on the list level.

lemma *minus-list-mset-Nil2* [*simp*]: *minus-list-mset xs* [] = *xs*
by (*simp add*: *minus-list-mset-def*)

lemma *minus-list-mset-Cons2* [*simp*]: *minus-list-mset xs* (*y*#*ys*) = *remove1 y* (*minus-list-mset xs ys*)
by (*simp add*: *minus-list-mset-def*)

lemma *count-list-minus-list-mset*[*simp*]:
count-list (*minus-list-mset xs ys*) *a* = *count-list xs a* − *count-list ys a*
by(*induction ys arbitrary*: *xs*) *auto*

lemma *minus-list-set-subset-minus-list-mset*: *set xs* − *set ys* ⊆ *set*(*minus-list-mset xs ys*)
by(*induction ys*)(*simp*, *fastforce*)

lemma *minus-list-mset-remove1-commute*:
minus-list-mset (*remove1 x xs*) *ys* = *remove1 x* (*minus-list-mset xs ys*)
by (*induction ys*) (*auto simp*: *remove1-commute*)

lemma *minus-list-mset-append* [*simp*]:
minus-list-mset xs (*ys*@*zs*) = *minus-list-mset* (*minus-list-mset xs ys*) *zs*
by (*induction ys*) (*auto simp add*: *minus-list-mset-remove1-commute*)

lemma *minus-list-mset-Nil1* [*simp*]: *minus-list-mset* [] *xs* = []
by (*induction xs*) *auto*

lemma *minus-list-mset-Cons1*: *minus-list-mset* (*x*#*xs*) *ys* =
(*if x* ∈ *set ys then minus-list-mset xs* (*remove1 x ys*) *else x* # (*minus-list-mset xs*

ys))
proof (*induction ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a ys*)
then show *?case*
by (*metis list.set-intros*(*1*,*2*) *minus-list-mset-Cons2 minus-list-mset-remove1-commute remove1.simps*(*2*)
set-ConsD)
qed

lemma *length-minus-list-mset*: *length*(*minus-list-mset xs ys*) ≤ *length xs*
by (*induction ys*) (*auto simp*: *le-diff-conv length-remove1*)

lemma *minus-list-mset-subset*:
set (*minus-list-mset xs ys*) ⊆ *set xs*
by (*induction ys*) (*simp*, *force*)

lemma *distinct-minus-list-mset*:
assumes *distinct xs*
shows *distinct* (*minus-list-mset xs ys*)
by (*induction ys*) (*use assms* **in** *auto*)

lemma *set-minus-list-mset-distinct*:
assumes *distinct xs* **shows** *set* (*minus-list-mset xs ys*) = *set xs* − *set ys*
by (*induction ys*) (*use assms distinct-minus-list-mset*[*of xs*] **in** *auto*)

66.2.9 *minus-list-set*

The difference of two lists viewed as sets. Conceptually, the result of *minus-list-set* is only determined up to the set of elements:

lemma *set-minus-list-set*[*simp*]: *set*(*minus-list-set xs ys*) = *set xs* − *set ys*
by(*induction ys*) (*auto simp*: *minus-list-set-def*)

lemma *minus-list-set-Nil2*[*simp*]: *minus-list-set xs* [] = *xs*
by(*simp add*: *minus-list-set-def*)

lemma *minus-list-set-Cons2*[*simp*]: *minus-list-set xs* (*y*#*ys*) = *removeAll y* (*minus-list-set xs ys*)
by(*simp add*: *minus-list-set-def*)

lemma *minus-list-set-eq-filter*: *minus-list-set xs ys* = *filter* (λ*x*. *x* ∉ *set ys*) *xs*
by(*induction ys arbitrary*: *xs*) (*auto simp*: *removeAll-filter-not-eq intro*: *filter-cong*)

lemma *minus-list-set-removeAll-commute*:
minus-list-set (*removeAll x xs*) *ys* = *removeAll x* (*minus-list-set xs ys*)
by (*induction ys*) (*auto simp*: *removeAll-commute*)

lemma *minus-list-set-Nil1* [*simp*]: *minus-list-set* [] *xs* = []
by (*simp add*: *minus-list-set-eq-filter*)

lemma *minus-list-set-Cons1*: *minus-list-set* (*x*#*xs*) *ys* =
(*if x* ∈ *set ys then minus-list-set xs ys else x* # (*minus-list-set xs ys*))
by(*simp add*:*minus-list-set-eq-filter*)

lemma *minus-list-set-append2*[*simp*]:
minus-list-set xs (*ys* @ *zs*) = *minus-list-set* (*minus-list-set xs ys*) *zs*
by (*induction ys*) (*auto simp*: *minus-list-set-removeAll-commute*)

lemma *length-minus-list-set*: *length*(*minus-list-set xs ys*) ≤ *length xs*
by (*simp add*: *minus-list-set-eq-filter*)

lemma *distinct-minus-list-set*: *distinct xs* ⟹ *distinct* (*minus-list-set xs ys*)
by (*simp add*: *minus-list-set-eq-filter*)

66.2.10 *inter-list-set*

The intersection of two lists viewed as sets. Conceptually, the result of *inter-list-set* is only determined up to the set of elements:

lemma *set-inter-list-set*[*simp*]: *set*(*inter-list-set xs ys*) = *set xs* ∩ *set ys*
by(*auto simp add*: *inter-list-set-def*)

lemma *inter-list-set-Nil*[*simp*]: *inter-list-set* [] *xs* = []
by (*simp add*: *inter-list-set-def*)

lemma *inter-list-set-Cons*[*simp*]: *inter-list-set* (*x*#*xs*) *ys* =
(*if x* ∈ *set ys then x* # *inter-list-set xs ys else inter-list-set xs ys*)
by(*simp add*:*inter-list-set-def*)

lemma *inter-list-set-Nil2*[*simp*]: *inter-list-set xs* [] = []
by(*simp add*: *inter-list-set-def*)

lemma *distinct-inter-list-set*[*simp*]: *distinct xs* ⟹ *distinct* (*inter-list-set xs ys*)
by (*simp add*: *inter-list-set-def*)

lemma *inter-list-set-append*[*simp*]:
inter-list-set (*xs* @ *ys*) *zs* = *inter-list-set xs zs* @ *inter-list-set ys zs*
by (*simp add*: *inter-list-set-def*)

lemma *length-inter-list-set*: *length*(*inter-list-set xs ys*) ≤ *length xs*
by (*simp add*: *inter-list-set-def*)

66.2.11 *replicate*

lemma *length-replicate* [*simp*]: *length* (*replicate n x*) = *n*
by (*induct n*) *auto*

lemma *replicate-eqI*:
assumes *length xs* = *n* **and** ⋀*y*. *y* ∈ *set xs* ⟹ *y* = *x*
shows *xs* = *replicate n x*
using *assms*
proof (*induct xs arbitrary*: *n*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **then show** *?case* **by** (*cases n*) *simp-all*
qed

lemma *Ex-list-of-length*: ∃ *xs*. *length xs* = *n*
by (*rule exI*[*of* - *replicate n undefined*]) *simp*

lemma *map-replicate* [*simp*]: *map f* (*replicate n x*) = *replicate n* (*f x*)
by (*induct n*) *auto*

lemma *map-replicate-const*:
map (λ *x*. *k*) *lst* = *replicate* (*length lst*) *k*
by (*induct lst*) *auto*

lemma *replicate-app-Cons-same*:
(*replicate n x*) @ (*x* # *xs*) = *x* # *replicate n x* @ *xs*
by (*induct n*) *auto*

lemma *rev-replicate* [*simp*]: *rev* (*replicate n x*) = *replicate n x*
by (*metis length-rev map-replicate map-replicate-const rev-map*)

lemma *replicate-add*: *replicate* (*n* + *m*) *x* = *replicate n x* @ *replicate m x*
by (*induct n*) *auto*

Courtesy of Matthias Daum:

lemma *append-replicate-commute*:
replicate n x @ *replicate k x* = *replicate k x* @ *replicate n x*
by (*metis add.commute replicate-add*)

Courtesy of Andreas Lochbihler:

lemma *filter-replicate*:
filter P (*replicate n x*) = (*if P x then replicate n x else* [])
by(*induct n*) *auto*

lemma *hd-replicate* [*simp*]: *n* ≠ *0* ⟹ *hd* (*replicate n x*) = *x*
by (*induct n*) *auto*

lemma *tl-replicate* [*simp*]: *tl* (*replicate n x*) = *replicate* (*n* − *1*) *x*
by (*induct n*) *auto*

lemma *last-replicate* [*simp*]: *n* ≠ *0* ⟹ *last* (*replicate n x*) = *x*
by (*atomize* (*full*), *induct n*) *auto*

lemma *nth-replicate*[*simp*]: $i < n \Longrightarrow (replicate\ n\ x)!i = x$
by (*induct n arbitrary*: *i*)(*auto simp*: *nth-Cons split*: *nat.split*)

Courtesy of Matthias Daum (2 lemmas):

lemma *take-replicate*[*simp*]: *take i* (*replicate k x*) = *replicate* (*min i k*) *x*
proof (*cases* $k \leq i$)
  case *True*
  then show *?thesis*
    by (*simp add*: *min-def*)
next
  case *False*
  then have *replicate k x* = *replicate i x* @ *replicate* ($k - i$) *x*
    by (*simp add*: *replicate-add* [*symmetric*])
  then show *?thesis*
    by (*simp add*: *min-def*)
qed

lemma *drop-replicate*[*simp*]: *drop i* (*replicate k x*) = *replicate* ($k-i$) *x*
proof (*induct k arbitrary*: *i*)
  case (*Suc k*)
  then show *?case*
    by (*simp add*: *drop-Cons'*)
qed *simp*

lemma *set-replicate-Suc*: *set* (*replicate* (*Suc n*) *x*) = $\{x\}$
by (*induct n*) *auto*

lemma *set-replicate* [*simp*]: $n \neq 0 \Longrightarrow set\ (replicate\ n\ x) = \{x\}$
by (*fast dest*!: *not0-implies-Suc intro*!: *set-replicate-Suc*)

lemma *set-replicate-conv-if*: *set* (*replicate n x*) = (*if n* = *0 then* {} *else* $\{x\}$)
by *auto*

lemma *in-set-replicate*[*simp*]: ($x \in set\ (replicate\ n\ y)$) = ($x = y \wedge n \neq 0$)
by (*simp add*: *set-replicate-conv-if*)

lemma *card-set-1-iff-replicate*:
  card(*set xs*) = *Suc 0* $\longleftrightarrow$ $xs \neq [] \wedge (\exists x.\ xs = replicate\ (length\ xs)\ x)$
by (*metis card-1-singleton-iff empty-iff insert-iff replicate-eqI set-empty2 set-replicate*)

lemma *Ball-set-replicate*[*simp*]:
  ($\forall x \in set(replicate\ n\ a).\ P\ x$) = ($P\ a \vee n{=}0$)
by(*simp add*: *set-replicate-conv-if*)

lemma *Bex-set-replicate*[*simp*]:
  ($\exists x \in set(replicate\ n\ a).\ P\ x$) = ($P\ a \wedge n{\neq}0$)
by(*simp add*: *set-replicate-conv-if*)

lemma *replicate-append-same*:
replicate i x @ [x] = x # replicate i x
by (*induct i*) *simp-all*

lemma *map-replicate-trivial*:
map (λ*i*. *x*) [*0*..<*i*] = *replicate i x*
by (*induct i*) (*simp-all add*: *replicate-append-same*)

lemma *concat-replicate-trivial*[*simp*]:
concat (*replicate i* []) = []
by (*induct i*) (*auto simp add*: *map-replicate-const*)

lemma *concat-replicate-single*[*simp*]: *concat* (*replicate m* [*a*]) = *replicate m a*
by(*induction m*) *auto*

lemma *replicate-empty*[*simp*]: (*replicate n x* = []) ⟷ *n=0*
by (*induct n*) *auto*

lemmas *empty-replicate*[*simp*] = *replicate-empty*[*THEN eq-iff-swap*]

lemma *replicate-eq-replicate*[*simp*]:
(*replicate m x* = *replicate n y*) ⟷ (*m=n* ∧ (*m≠0* ⟶ *x=y*))
proof (*induct m arbitrary*: *n*)
case (*Suc m n*)
then show *?case*
by (*induct n*) *auto*
qed *simp*

lemma *takeWhile-replicate*[*simp*]:
takeWhile P (*replicate n x*) = (*if P x then replicate n x else* [])
using *takeWhile-eq-Nil-iff* **by** *fastforce*

lemma *dropWhile-replicate*[*simp*]:
dropWhile P (*replicate n x*) = (*if P x then* [] *else replicate n x*)
using *dropWhile-eq-self-iff* **by** *fastforce*

lemma *replicate-length-filter*:
replicate (*length* (*filter* (λ*y*. *x* = *y*) *xs*)) *x* = *filter* (λ*y*. *x* = *y*) *xs*
by (*induct xs*) *auto*

lemma *comm-append-are-replicate*:
xs @ ys = *ys @ xs* ⟹ ∃ *m n zs*. *concat* (*replicate m zs*) = *xs* ∧ *concat* (*replicate n zs*) = *ys*
proof (*induction length* (*xs @ ys*) + *length xs arbitrary*: *xs ys rule*: *less-induct*)
case *less*
consider (*1*) *length ys* < *length xs* | (*2*) *xs* = [] | (*3*) *length xs* ≤ *length ys* ∧ *xs* ≠ []
by *linarith*
then show *?case*

```
  proof (cases)
    case 1
    then show ?thesis
      using less.hyps[OF - less.prems[symmetric]] nat-add-left-cancel-less by auto
  next
    case 2
    then have concat (replicate 0 ys) = xs ∧ concat (replicate 1 ys) = ys
      by simp
    then show ?thesis
      by blast
  next
    case 3
    then have length xs ≤ length ys and xs ≠ []
      by blast+
    from ‹length xs ≤ length ys› and  ‹xs @ ys = ys @ xs›
    obtain ws where ys = xs @ ws
      by (auto simp: append-eq-append-conv2)
    from this and ‹xs ≠ []›
    have length ws < length ys
      by simp
    from ‹xs @ ys = ys @ xs›[unfolded ‹ys = xs @ ws›]
    have xs @ ws = ws @ xs
      by simp
    from less.hyps[OF - this] ‹length ws < length ys›
    obtain m n' zs where concat (replicate m zs) = xs and  concat (replicate n'
zs) = ws
      by auto
    then have concat (replicate (m+n') zs) = ys
      using ‹ys = xs @ ws›
      by (simp add: replicate-add)
    then show ?thesis
      using ‹concat (replicate m zs) = xs› by blast
  qed
qed

lemma comm-append-is-replicate:
  fixes xs ys :: 'a list
  assumes xs ≠ [] ys ≠ []
  assumes xs @ ys = ys @ xs
  shows ∃ n zs. n > 1 ∧ concat (replicate n zs) = xs @ ys
proof −
  obtain m n zs where concat (replicate m zs) = xs
    and concat (replicate n zs) = ys
    using comm-append-are-replicate[OF assms(3)] by blast
  then have m + n > 1 and concat (replicate (m+n) zs) = xs @ ys
    using ‹xs ≠ []› and ‹ys ≠ []›
    by (auto simp: replicate-add)
  then show ?thesis by blast
qed
```

lemma *Cons-replicate-eq*:
x # xs = replicate n y ⟷ x = y ∧ n > 0 ∧ xs = replicate (n − 1) x
by (*induct n*) *auto*

lemma *replicate-length-same*:
(∀ y∈set xs. y = x) ⟹ replicate (length xs) x = xs
by (*induct xs*) *simp-all*

lemma *foldr-replicate* [*simp*]:
foldr f (replicate n x) = f x ^^ n
by (*induct n*) (*simp-all*)

lemma *fold-replicate* [*simp*]:
fold f (replicate n x) = f x ^^ n
by (*subst foldr-fold* [*symmetric*]) *simp-all*

66.2.12 *indexed-from*

lemma *indexed-from-simps* [*simp*, *code*]:
indexed-from n [] = []
indexed-from n (x # xs) = (n, x) # indexed-from (Suc n) xs
by (*simp-all add*: *indexed-from-eq-zip upt-rec*)

lemma *length-indexed-from* [*simp*]:
length (indexed-from n xs) = length xs
by (*simp add*: *indexed-from-eq-zip*)

lemma *map-fst-indexed-from* [*simp*]:
map fst (indexed-from n xs) = [n..<n + length xs]
by (*simp add*: *indexed-from-eq-zip*)

lemma *map-snd-indexed-from* [*simp*]:
map snd (indexed-from n xs) = xs
by (*simp add*: *indexed-from-eq-zip*)

lemma *in-set-indexed-from-eq*:
p ∈ set (indexed-from n xs) ⟷ n ≤ fst p ∧ fst p < length xs + n ∧ nth xs (fst p − n) = snd p
proof −
have *∃ q. [n..<n + length xs] ! q = m ∧ xs ! q = xs ! (m − n) ∧ q < length xs*
if *n ≤ m m < length xs + n* **for** *m*
proof −
from *that* **have** *[n..<n + length xs] ! (m − n) = m ∧*
xs ! (m − n) = xs ! (m − n) ∧ m − n < length xs **by** *auto*
then show *?thesis*
by *auto*
qed
then show *?thesis*

by (*cases p*) (*auto simp add*: *indexed-from-eq-zip in-set-zip*)
qed

lemma *nth-indexed-from-eq*: $m < length\ xs \Longrightarrow indexed\text{-}from\ n\ xs\ !\ m = (n + m, xs\ !\ m)$
by (*simp add*: *indexed-from-eq-zip*)

lemma *indexed-from-replicate-eq*:
indexed-from n (*replicate m a*) = *map* ($\lambda q.\ (q,\ a)$) [*n*..<*n* + *m*]
by (*rule pair-list-eqI*)
(*simp-all add*: *indexed-from-eq-zip comp-def map-replicate-const*)

lemma *indexed-from-Suc-eq*:
indexed-from (*Suc n*) *xs* = *map* (*apfst Suc*) (*indexed-from n xs*)
by (*rule pair-list-eqI*)
(*simp-all add*: *not-le*, *simp del*: *map-map add*: *map-Suc-upt map-map* [*symmetric*])

lemma *distinct-indexed-from* [*simp*]:
distinct (*indexed-from n xs*)
by (*simp add*: *indexed-from-eq-zip distinct-zipI1*)

lemma *indexed-from-append-eq*:
indexed-from n (*xs* @ *ys*) = *indexed-from n xs* @ *indexed-from* (*n* + *length xs*) *ys*
by (*simp add*: *indexed-from-eq-zip add.assoc zip-append2*)

lemma *indexed-from-map-upt*:
indexed-from n (*map f* [*n*..<*m*]) = *map* ($\lambda k.\ (k,\ f\ k)$) [*n*..<*m*]
by (*cases n* ≤ *m*) (*simp-all add*: *zip-map2 zip-same-conv-map indexed-from-eq-zip*)

66.2.13 *rotate1* and *rotate*

lemma *rotate0*[*simp*]: *rotate 0* = *id*
by(*simp add*:*rotate-def*)

lemma *rotate-Suc*[*simp*]: *rotate* (*Suc n*) *xs* = *rotate1*(*rotate n xs*)
by(*simp add*:*rotate-def*)

lemma *rotate-add*:
rotate (*m*+*n*) = *rotate m* ∘ *rotate n*
by(*simp add*:*rotate-def funpow-add*)

lemma *rotate-rotate*: *rotate m* (*rotate n xs*) = *rotate* (*m*+*n*) *xs*
by(*simp add*:*rotate-add*)

lemma *rotate1-map*: *rotate1* (*map f xs*) = *map f* (*rotate1 xs*)
by(*cases xs*) *simp-all*

lemma *rotate1-rotate-swap*: *rotate1* (*rotate n xs*) = *rotate n* (*rotate1 xs*)
by(*simp add*:*rotate-def funpow-swap1*)

lemma *rotate1-length01*[*simp*]: *length xs* ≤ *1* ⟹ *rotate1 xs* = *xs*
by(*cases xs*) *simp-all*

lemma *rotate-length01*[*simp*]: *length xs* ≤ *1* ⟹ *rotate n xs* = *xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate1-hd-tl*: *xs* ≠ [] ⟹ *rotate1 xs* = *tl xs* @ [*hd xs*]
by (*cases xs*) *simp-all*

lemma *rotate-drop-take*:
rotate n xs = *drop* (*n mod length xs*) *xs* @ *take* (*n mod length xs*) *xs*
proof (*induct n*)
case (*Suc n*)
show *?case*
proof (*cases xs* = [])
case *False*
then show *?thesis*
proof (*cases n mod length xs* = *0*)
case *True*
then show *?thesis*
by (*auto simp add*: *mod-Suc False Suc.hyps drop-Suc rotate1-hd-tl take-Suc Suc-length-conv*)
next
case *False*
with ‹*xs* ≠ []› *Suc*
show *?thesis*
by (*simp add*: *rotate-def mod-Suc rotate1-hd-tl drop-Suc*[*symmetric*] *drop-tl*[*symmetric*] *take-hd-drop linorder-not-le*)
qed
qed *simp*
qed *simp*

lemma *rotate-conv-mod*: *rotate n xs* = *rotate* (*n mod length xs*) *xs*
by(*simp add*:*rotate-drop-take*)

lemma *rotate-id*[*simp*]: *n mod length xs* = *0* ⟹ *rotate n xs* = *xs*
by(*simp add*:*rotate-drop-take*)

lemma *length-rotate1*[*simp*]: *length*(*rotate1 xs*) = *length xs*
by (*cases xs*) *simp-all*

lemma *length-rotate*[*simp*]: *length*(*rotate n xs*) = *length xs*
by (*induct n arbitrary*: *xs*) (*simp-all add*:*rotate-def*)

lemma *distinct1-rotate*[*simp*]: *distinct*(*rotate1 xs*) = *distinct xs*
by (*cases xs*) *auto*

lemma *distinct-rotate*[*simp*]: *distinct*(*rotate n xs*) = *distinct xs*

by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-map*: *rotate n* (*map f xs*) = *map f* (*rotate n xs*)
by(*simp add*:*rotate-drop-take take-map drop-map*)

lemma *set-rotate1*[*simp*]: *set*(*rotate1 xs*) = *set xs*
by (*cases xs*) *auto*

lemma *set-rotate*[*simp*]: *set*(*rotate n xs*) = *set xs*
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate1-replicate*[*simp*]: *rotate1* (*replicate n a*) = *replicate n a*
by (*cases n*) (*simp-all add*: *replicate-append-same*)

lemma *rotate1-is-Nil-conv*[*simp*]: (*rotate1 xs* = []) = (*xs* = [])
by (*cases xs*) *auto*

lemma *rotate-is-Nil-conv*[*simp*]: (*rotate n xs* = []) = (*xs* = [])
by (*induct n*) (*simp-all add*:*rotate-def*)

lemma *rotate-rev*:
rotate n (*rev xs*) = *rev*(*rotate* (*length xs* − (*n mod length xs*)) *xs*)
proof (*cases length xs* = *0* ∨ *n mod length xs* = *0*)
case *False*
then show *?thesis*
by(*simp add*:*rotate-drop-take rev-drop rev-take*)
qed *force*

lemma *hd-rotate-conv-nth*:
assumes *xs* ≠ [] **shows** *hd*(*rotate n xs*) = *xs*!(*n mod length xs*)
proof −
have *n mod length xs* < *length xs*
using *assms* **by** *simp*
then show *?thesis*
by (*metis drop-eq-Nil hd-append2 hd-drop-conv-nth leD rotate-drop-take*)
qed

lemma *rotate-append*: *rotate* (*length l*) (*l* @ *q*) = *q* @ *l*
by (*induct l arbitrary*: *q*) (*auto simp add*: *rotate1-rotate-swap*)

lemma *nth-rotate*:
‹*rotate m xs* ! *n* = *xs* ! ((*m* + *n*) *mod length xs*)› **if** ‹*n* < *length xs*›
by (*smt* (*verit*) *add.commute hd-rotate-conv-nth length-rotate not-less0 list.size*(*3*)
mod-less rotate-rotate that)

lemma *nth-rotate1*:
‹*rotate1 xs* ! *n* = *xs* ! (*Suc n mod length xs*)› **if** ‹*n* < *length xs*›
using *that nth-rotate* [*of n xs 1*] **by** *simp*

lemma *inj-rotate1*: *inj rotate1*
proof
  fix *xs ys* :: *'a list* **show** *rotate1 xs = rotate1 ys ⟹ xs = ys*
    by (*cases xs*; *cases ys*; *simp*)
qed

lemma *surj-rotate1*: *surj rotate1*
proof (*safe*, *simp-all*)
  fix *xs* :: *'a list* **show** *xs ∈ range rotate1*
  proof (*cases xs rule*: *rev-exhaust*)
    case *Nil*
    hence *xs = rotate1 []* **by** *auto*
    thus *?thesis* **by** *fast*
  next
    case (*snoc as a*)
    hence *xs = rotate1 (a#as)* **by** *force*
    thus *?thesis* **by** *fast*
  qed
qed

lemma *bij-rotate1*: *bij* (*rotate1* :: *'a list ⇒ 'a list*)
using *bijI inj-rotate1 surj-rotate1* **by** *blast*

lemma *rotate1-fixpoint-card*: *rotate1 xs = xs ⟹ xs = [] ∨ card*(*set xs*) = *1*
by(*induction xs*) (*auto simp*: *card-insert-if*[*OF finite-set*] *append-eq-Cons-conv*)

66.2.14 *nths* — a generalization of (!) to sets

lemma *nths-empty* [*simp*]: *nths xs {} = []*
  by (*auto simp add*: *nths-def*)

lemma *nths-nil* [*simp*]: *nths [] A = []*
  by (*auto simp add*: *nths-def*)

lemma *nths-all*: *∀ i < length xs. i ∈ I ⟹ nths xs I = xs*
  unfolding *nths-def*
  by (*metis add-0 diff-zero filter-True in-set-zip length-upt nth-upt zip-eq-conv*)

lemma *length-nths*:
  length (*nths xs I*) = *card*{*i*. *i < length xs ∧ i ∈ I*}
  by(*simp add*: *nths-def length-filter-conv-card cong*:*conj-cong*)

lemma *nths-shift-lemma-Suc*:
  map fst (*filter* (*λp*. *P*(*Suc*(*snd p*))) (*zip xs is*)) =
   map fst (*filter* (*λp*. *P*(*snd p*)) (*zip xs* (*map Suc is*)))
proof (*induct xs arbitrary*: *is*)
  case (*Cons x xs is*)
  show *?case*
    by (*cases is*) (*auto simp add*: *Cons.hyps*)

qed *simp*

lemma *nths-shift-lemma*:
map fst (*filter* (λ*p*. *snd p* ∈ *A*) (*zip xs* [*i*..<*i* + *length xs*])) =
map fst (*filter* (λ*p*. *snd p* + *i* ∈ *A*) (*zip xs* [0..<*length xs*]))
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *add.commute*)

lemma *nths-append*:
nths (*l* @ *l'*) *A* = *nths l A* @ *nths l'* {*j*. *j* + *length l* ∈ *A*}
unfolding *nths-def*
proof (*induct l' rule*: *rev-induct*)
case (*snoc x xs*)
then show *?case*
by (*simp add*: *upt-add-eq-append*[*of 0*] *nths-shift-lemma add.commute*)
qed *auto*

lemma *nths-Cons*:
nths (*x* # *l*) *A* = (*if* 0 ∈ *A then* [*x*] *else* []) @ *nths l* {*j*. *Suc j* ∈ *A*}
proof (*induct l rule*: *rev-induct*)
case (*snoc x xs*)
then show *?case*
by (*simp flip*: *append-Cons add*: *nths-append*)
qed (*auto simp*: *nths-def*)

lemma *nths-map*: *nths* (*map f xs*) *I* = *map f* (*nths xs I*)
by(*induction xs arbitrary*: *I*) (*simp-all add*: *nths-Cons*)

lemma *set-nths*: *set*(*nths xs I*) = {*xs*!*i*|*i*. *i*<*size xs* ∧ *i* ∈ *I*}
by (*induct xs arbitrary*: *I*) (*auto simp*: *nths-Cons nth-Cons split*:*nat.split dest*!:
gr0-implies-Suc)

lemma *set-nths-subset*: *set*(*nths xs I*) ⊆ *set xs*
by(*auto simp add*:*set-nths*)

lemma *notin-set-nthsI*[*simp*]: *x* ∉ *set xs* ⟹ *x* ∉ *set*(*nths xs I*)
by(*auto simp add*:*set-nths*)

lemma *in-set-nthsD*: *x* ∈ *set*(*nths xs I*) ⟹ *x* ∈ *set xs*
by(*auto simp add*:*set-nths*)

lemma *nths-singleton* [*simp*]: *nths* [*x*] *A* = (*if* 0 ∈ *A then* [*x*] *else* [])
by (*simp add*: *nths-Cons*)

lemma *distinct-nthsI*[*simp*]: *distinct xs* ⟹ *distinct* (*nths xs I*)
by (*induct xs arbitrary*: *I*) (*auto simp*: *nths-Cons*)

lemma *nths-upt-eq-take* [*simp*]: *nths l* {..<*n*} = *take n l*
by (*induct l rule*: *rev-induct*) (*simp-all split*: *nat-diff-split add*: *nths-append*)

lemma *nths-nths*: *nths* (*nths xs A*) *B* = *nths xs* {*i* ∈ *A*. ∃ *j* ∈ *B*. *card* {*i'* ∈ *A*. *i'* < *i*} = *j*}
by (*induction xs arbitrary*: *A B*) (*auto simp add*: *nths-Cons card-less-Suc card-less-Suc2*)

lemma *drop-eq-nths*: *drop n xs* = *nths xs* {*i*. *i* ≥ *n*}
by (*induction xs arbitrary*: *n*) (*auto simp add*: *nths-Cons nths-all drop-Cons'* *intro*: *arg-cong2*[**where** *f*=*nths*, *OF refl*])

lemma *nths-drop*: *nths* (*drop n xs*) *I* = *nths xs* ((+) *n* ' *I*)
by(*force simp*: *drop-eq-nths nths-nths simp flip*: *atLeastLessThan-iff*
intro: *arg-cong2*[**where** *f*=*nths*, *OF refl*])

lemma *filter-eq-nths*: *filter P xs* = *nths xs* {*i*. *i*<*length xs* ∧ *P*(*xs*!*i*)}
by(*induction xs*) (*auto simp*: *nths-Cons*)

lemma *filter-in-nths*:
distinct xs ⟹ *filter* (%*x*. *x* ∈ *set* (*nths xs s*)) *xs* = *nths xs s*
proof (*induct xs arbitrary*: *s*)
case *Nil* **thus** *?case* **by** *simp*
next
case (*Cons a xs*)
then have ∀ *x*. *x* ∈ *set xs* ⟶ *x* ≠ *a* **by** *auto*
with *Cons* **show** *?case* **by**(*simp add*: *nths-Cons cong*:*filter-cong*)
qed

66.2.15 *subseqs* and *List.n-lists*

lemma *length-subseqs*: *length* (*subseqs xs*) = 2 ^ *length xs*
by (*induct xs*) (*simp-all add*: *Let-def*)

lemma *subseqs-powset*: *set* ' *set* (*subseqs xs*) = *Pow* (*set xs*)
proof −
have *aux*: ⋀*x A*. *set* ' *Cons x* ' *A* = *insert x* ' *set* ' *A*
by (*auto simp add*: *image-def*)
have *set* (*map set* (*subseqs xs*)) = *Pow* (*set xs*)
by (*induct xs*) (*simp-all add*: *aux Let-def Pow-insert Un-commute comp-def del*: *map-map*)
then show *?thesis* **by** *simp*
qed

lemma *distinct-set-subseqs*:
assumes *distinct xs*
shows *distinct* (*map set* (*subseqs xs*))
by (*simp add*: *assms card-Pow card-distinct distinct-card length-subseqs subseqs-powset*)

lemma *n-lists-Nil* [*simp*]: *List.n-lists n* [] = (*if n* = *0 then* [[]] *else* [])
by (*induct n*) *simp-all*

lemma *length-n-lists-elem*: *ys* ∈ *set* (*List.n-lists n xs*) ⟹ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*

lemma *set-n-lists*: *set* (*List.n-lists n xs*) = {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}
proof (*rule set-eqI*)
fix *ys* :: *'a list*
show *ys* ∈ *set* (*List.n-lists n xs*) ⟷ *ys* ∈ {*ys*. *length ys* = *n* ∧ *set ys* ⊆ *set xs*}
proof −
have *ys* ∈ *set* (*List.n-lists n xs*) ⟹ *length ys* = *n*
by (*induct n arbitrary*: *ys*) *auto*
moreover have ⋀*x*. *ys* ∈ *set* (*List.n-lists n xs*) ⟹ *x* ∈ *set ys* ⟹ *x* ∈ *set xs*
by (*induct n arbitrary*: *ys*) *auto*
moreover have *set ys* ⊆ *set xs* ⟹ *ys* ∈ *set* (*List.n-lists* (*length ys*) *xs*)
by (*induct ys*) *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *subseqs-refl*: *xs* ∈ *set* (*subseqs xs*)
by (*induct xs*) (*simp-all add*: *Let-def*)

lemma *subset-subseqs*: *X* ⊆ *set xs* ⟹ *X* ∈ *set* ‘ *set* (*subseqs xs*)
unfolding *subseqs-powset* **by** *simp*

lemma *Cons-in-subseqsD*: *y* # *ys* ∈ *set* (*subseqs xs*) ⟹ *ys* ∈ *set* (*subseqs xs*)
by (*induct xs*) (*auto simp*: *Let-def*)

lemma *subseqs-distinctD*: ⟦ *ys* ∈ *set* (*subseqs xs*); *distinct xs* ⟧ ⟹ *distinct ys*
proof (*induct xs arbitrary*: *ys*)
case (*Cons x xs ys*)
then show *?case*
by (*auto simp*: *Let-def*) (*metis Pow-iff contra-subsetD image-eqI subseqs-powset*)
qed *simp*

66.2.16 *splice*

lemma *splice-Nil2* [*simp*]: *splice xs* [] = *xs*
by (*cases xs*) *simp-all*

lemma *length-splice*[*simp*]: *length*(*splice xs ys*) = *length xs* + *length ys*
by (*induct xs ys rule*: *splice.induct*) *auto*

lemma *split-Nil-iff*[*simp*]: *splice xs ys* = [] ⟷ *xs* = [] ∧ *ys* = []
by (*induct xs ys rule*: *splice.induct*) *auto*

lemma *splice-replicate*[*simp*]: *splice* (*replicate m x*) (*replicate n x*) = *replicate* (*m*+*n*) *x*
proof (*induction replicate m x replicate n x arbitrary*: *m n rule*: *splice.induct*)
case (*2 x xs*)

then show *?case*
by (*auto simp add*: *Cons-replicate-eq dest*: *gr0-implies-Suc*)
qed *auto*

66.2.17 *shuffles*

lemma *shuffles-commutes*: *shuffles xs ys* = *shuffles ys xs*
by (*induction xs ys rule*: *shuffles.induct*) (*simp-all add*: *Un-commute*)

lemma *Nil-in-shuffles*[*simp*]: [] ∈ *shuffles xs ys* ⟷ *xs* = [] ∧ *ys* = []
by (*induct xs ys rule*: *shuffles.induct*) *auto*

lemma *shufflesE*:
zs ∈ *shuffles xs ys* ⟹
(*zs* = *xs* ⟹ *ys* = [] ⟹ *P*) ⟹
(*zs* = *ys* ⟹ *xs* = [] ⟹ *P*) ⟹
(⋀*x xs' z zs'*. *xs* = *x* # *xs'* ⟹ *zs* = *z* # *zs'* ⟹ *x* = *z* ⟹ *zs'* ∈ *shuffles xs' ys* ⟹ *P*) ⟹
(⋀*y ys' z zs'*. *ys* = *y* # *ys'* ⟹ *zs* = *z* # *zs'* ⟹ *y* = *z* ⟹ *zs'* ∈ *shuffles xs ys'* ⟹ *P*) ⟹ *P*
by (*induct xs ys rule*: *shuffles.induct*) *auto*

lemma *Cons-in-shuffles-iff*:
z # *zs* ∈ *shuffles xs ys* ⟷
(*xs* ≠ [] ∧ *hd xs* = *z* ∧ *zs* ∈ *shuffles* (*tl xs*) *ys* ∨
ys ≠ [] ∧ *hd ys* = *z* ∧ *zs* ∈ *shuffles xs* (*tl ys*))
by (*induct xs ys rule*: *shuffles.induct*) *auto*

lemma *splice-in-shuffles* [*simp*, *intro*]: *splice xs ys* ∈ *shuffles xs ys*
by (*induction xs ys rule*: *splice.induct*) (*simp-all add*: *Cons-in-shuffles-iff shuffles-commutes*)

lemma *Nil-in-shufflesI*: *xs* = [] ⟹ *ys* = [] ⟹ [] ∈ *shuffles xs ys*
by *simp*

lemma *Cons-in-shuffles-leftI*: *zs* ∈ *shuffles xs ys* ⟹ *z* # *zs* ∈ *shuffles* (*z* # *xs*) *ys*
by (*cases ys*) *auto*

lemma *Cons-in-shuffles-rightI*: *zs* ∈ *shuffles xs ys* ⟹ *z* # *zs* ∈ *shuffles xs* (*z* # *ys*)
by (*cases xs*) *auto*

lemma *finite-shuffles* [*simp*, *intro*]: *finite* (*shuffles xs ys*)
by (*induction xs ys rule*: *shuffles.induct*) *simp-all*

lemma *length-shuffles*: *zs* ∈ *shuffles xs ys* ⟹ *length zs* = *length xs* + *length ys*
by (*induction xs ys arbitrary*: *zs rule*: *shuffles.induct*) *auto*

lemma *set-shuffles*: *zs* ∈ *shuffles xs ys* ⟹ *set zs* = *set xs* ∪ *set ys*
by (*induction xs ys arbitrary*: *zs rule*: *shuffles.induct*) *auto*

lemma *distinct-disjoint-shuffles*:
assumes *distinct xs distinct ys set xs* ∩ *set ys* = {} *zs* ∈ *shuffles xs ys*
shows *distinct zs*
using *assms*
proof (*induction xs ys arbitrary*: *zs rule*: *shuffles.induct*)
case (*3 x xs y ys*)
show *?case*
proof (*cases zs*)
case (*Cons z zs'*)
with *3.prems* **and** *3.IH*[*of zs'*] **show** *?thesis* **by** (*force dest*: *set-shuffles*)
qed *simp-all*
qed *simp-all*

lemma *Cons-shuffles-subset1*: (#) *x* ‘ *shuffles xs ys* ⊆ *shuffles* (*x* # *xs*) *ys*
by (*cases ys*) *auto*

lemma *Cons-shuffles-subset2*: (#) *y* ‘ *shuffles xs ys* ⊆ *shuffles xs* (*y* # *ys*)
by (*cases xs*) *auto*

lemma *filter-shuffles*:
filter P ‘ *shuffles xs ys* = *shuffles* (*filter P xs*) (*filter P ys*)
proof −
have ∗: *filter P* ‘ (#) *x* ‘ *A* = (*if P x then* (#) *x* ‘ *filter P* ‘ *A else filter P* ‘ *A*)
for *x A*
by (*auto simp*: *image-image*)
show *?thesis*
by (*induction xs ys rule*: *shuffles.induct*)
(*simp-all split*: *if-splits add*: *image-Un* ∗ *Un-absorb1 Un-absorb2*
Cons-shuffles-subset1 Cons-shuffles-subset2)
qed

lemma *filter-shuffles-disjoint1*:
assumes *set xs* ∩ *set ys* = {} *zs* ∈ *shuffles xs ys*
shows *filter* (λ*x*. *x* ∈ *set xs*) *zs* = *xs* (**is** *filter ?P - = -*)
and *filter* (λ*x*. *x* ∉ *set xs*) *zs* = *ys* (**is** *filter ?Q - = -*)
using *assms*
proof −
from *assms* **have** *filter ?P zs* ∈ *filter ?P* ‘ *shuffles xs ys* **by** *blast*
also have *filter ?P* ‘ *shuffles xs ys* = *shuffles* (*filter ?P xs*) (*filter ?P ys*)
by (*rule filter-shuffles*)
also have *filter ?P xs* = *xs* **by** (*rule filter-True*) *simp-all*
also have *filter ?P ys* = [] **by** (*rule filter-False*) (*insert assms*(*1*), *auto*)
also have *shuffles xs* [] = {*xs*} **by** *simp*
finally show *filter ?P zs* = *xs* **by** *simp*
next
from *assms* **have** *filter ?Q zs* ∈ *filter ?Q* ‘ *shuffles xs ys* **by** *blast*

also have *filter ?Q ‘ shuffles xs ys = shuffles (filter ?Q xs) (filter ?Q ys)*
by (*rule filter-shuffles*)
also have *filter ?Q ys = ys* **by** (*rule filter-True*) (*insert assms(1), auto*)
also have *filter ?Q xs = []* **by** (*rule filter-False*) (*insert assms(1), auto*)
also have *shuffles [] ys = {ys}* **by** *simp*
finally show *filter ?Q zs = ys* **by** *simp*
qed

lemma *filter-shuffles-disjoint2*:
assumes *set xs ∩ set ys = {}* *zs ∈ shuffles xs ys*
shows *filter (λx. x ∈ set ys) zs = ys filter (λx. x ∉ set ys) zs = xs*
using *filter-shuffles-disjoint1*[*of ys xs zs*] *assms*
by (*simp-all add*: *shuffles-commutes Int-commute*)

lemma *partition-in-shuffles*:
xs ∈ shuffles (filter P xs) (filter (λx. ¬P x) xs)
proof (*induction xs*)
case (*Cons x xs*)
show *?case*
proof (*cases P x*)
case *True*
hence *x # xs ∈ (#) x ‘ shuffles (filter P xs) (filter (λx. ¬P x) xs)*
by (*intro imageI Cons.IH*)
also have *... ⊆ shuffles (filter P (x # xs)) (filter (λx. ¬P x) (x # xs))*
by (*simp add*: *True Cons-shuffles-subset1*)
finally show *?thesis* **.**
next
case *False*
hence *x # xs ∈ (#) x ‘ shuffles (filter P xs) (filter (λx. ¬P x) xs)*
by (*intro imageI Cons.IH*)
also have *... ⊆ shuffles (filter P (x # xs)) (filter (λx. ¬P x) (x # xs))*
by (*simp add*: *False Cons-shuffles-subset2*)
finally show *?thesis* **.**
qed
qed *auto*

lemma *inv-image-partition*:
assumes *⋀x. x ∈ set xs ⟹ P x ⋀y. y ∈ set ys ⟹ ¬P y*
shows *partition P −‘ {(xs, ys)} = shuffles xs ys*
proof (*intro equalityI subsetI*)
fix *zs* **assume** *zs*: *zs ∈ shuffles xs ys*
hence [*simp*]: *set zs = set xs ∪ set ys* **by** (*rule set-shuffles*)
from *assms* **have** *filter P zs = filter (λx. x ∈ set xs) zs*
filter (λx. ¬P x) zs = filter (λx. x ∈ set ys) zs
by (*intro filter-cong refl*; *force*)+
moreover from *assms* **have** *set xs ∩ set ys = {}* **by** *auto*
ultimately show *zs ∈ partition P −‘ {(xs, ys)}* **using** *zs*
by (*simp add*: *o-def filter-shuffles-disjoint1 filter-shuffles-disjoint2*)
next

fix *zs* **assume** *zs* ∈ *partition P* −‘ {(*xs*, *ys*)}
thus *zs* ∈ *shuffles xs ys* **using** *partition-in-shuffles*[*of zs*] **by** (*auto simp*: *o-def*)
qed

66.2.18 Transpose

function *transpose* **where**
transpose [] = [] |
transpose ([] # *xss*) = *transpose xss* |
transpose ((*x*#*xs*) # *xss*) =
(*x* # [*h*. (*h*#*t*) ← *xss*]) # *transpose* (*xs* # [*t*. (*h*#*t*) ← *xss*])
by *pat-completeness auto*

lemma *transpose-aux-filter-head*:
concat (*map* (*case-list* [] (λ*h t*. [*h*])) *xss*) =
map (λ*xs*. *hd xs*) (*filter* (λ*ys*. *ys* ≠ []) *xss*)
by (*induct xss*) (*auto split*: *list.split*)

lemma *transpose-aux-filter-tail*:
concat (*map* (*case-list* [] (λ*h t*. [*t*])) *xss*) =
map (λ*xs*. *tl xs*) (*filter* (λ*ys*. *ys* ≠ []) *xss*)
by (*induct xss*) (*auto split*: *list.split*)

lemma *transpose-aux-max*:
max (*Suc* (*length xs*)) (*foldr* (λ*xs*. *max* (*length xs*)) *xss* 0) =
Suc (*max* (*length xs*) (*foldr* (λ*x*. *max* (*length x* − *Suc* 0)) (*filter* (λ*ys*. *ys* ≠ []) *xss*) 0))
(**is** *max* - *?foldB* = *Suc* (*max* - *?foldA*))
proof (*cases* (*filter* (λ*ys*. *ys* ≠ []) *xss*) = [])
case *True*
hence *foldr* (λ*xs*. *max* (*length xs*)) *xss* 0 = 0
proof (*induct xss*)
case (*Cons x xs*)
then have *x* = [] **by** (*cases x*) *auto*
with *Cons* **show** *?case* **by** *auto*
qed *simp*
thus *?thesis* **using** *True* **by** *simp*
next
case *False*

have *foldA*: *?foldA* = *foldr* (λ*x*. *max* (*length x*)) (*filter* (λ*ys*. *ys* ≠ []) *xss*) 0 − 1
by (*induct xss*) *auto*
have *foldB*: *?foldB* = *foldr* (λ*x*. *max* (*length x*)) (*filter* (λ*ys*. *ys* ≠ []) *xss*) 0
by (*induct xss*) *auto*

have 0 < *?foldB*
proof −
from *False*
obtain *z zs* **where** *zs*: (*filter* (λ*ys*. *ys* ≠ []) *xss*) = *z*#*zs* **by** (*auto simp*:

```
neq-Nil-conv)
    hence z ∈ set (filter (λys. ys ≠ []) xss) by auto
    hence z ≠ [] by auto
    thus ?thesis
      unfolding foldB zs
      by (auto simp: max-def intro: less-le-trans)
  qed
  thus ?thesis
    unfolding foldA foldB max-Suc-Suc[symmetric]
    by simp
qed

termination transpose
  by (relation measure (λxs. foldr (λxs. max (length xs)) xs 0 + length xs))
      (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
less-Suc-eq-le)

lemma transpose-empty: (transpose xs = []) ⟷ (∀ x ∈ set xs. x = [])
  by (induct rule: transpose.induct) simp-all

lemma length-transpose:
  fixes xs :: 'a list list
  shows length (transpose xs) = foldr (λxs. max (length xs)) xs 0
  by (induct rule: transpose.induct)
    (auto simp: transpose-aux-filter-tail foldr-map comp-def transpose-aux-max
              max-Suc-Suc[symmetric] simp del: max-Suc-Suc)

lemma nth-transpose:
  fixes xs :: 'a list list
  assumes i < length (transpose xs)
  shows transpose xs ! i = map (λxs. xs ! i) (filter (λys. i < length ys) xs)
using assms proof (induct arbitrary: i rule: transpose.induct)
  case (3 x xs xss)
  define XS where XS = (x # xs) # xss
  hence [simp]: XS ≠ [] by auto
  thus ?case
  proof (cases i)
    case 0
    thus ?thesis by (simp add: transpose-aux-filter-head hd-conv-nth)
  next
    case (Suc j)
    have ∗: ⋀xss. xs # map tl xss = map tl ((x#xs)#xss) by simp
      have ∗∗: ⋀xss. (x#xs) # filter (λys. ys ≠ []) xss = filter (λys. ys ≠ [])
((x#xs)#xss) by simp
    { fix xs :: ‹'a list› have Suc j < length xs ⟷ xs ≠ [] ∧ j < length xs − Suc 0
      by (cases xs) simp-all
    } note ∗∗∗ = this

     have j-less: j < length (transpose (xs # concat (map (case-list [] (λh t. [t]))
```

xss)))
using *3.prems* **by** (*simp add*: *transpose-aux-filter-tail length-transpose Suc*)

show *?thesis*
unfolding *transpose.simps* ‹*i* = *Suc j*› *nth-Cons-Suc 3.hyps*[*OF j-less*]
by (*auto simp*: *nth-tl transpose-aux-filter-tail filter-map comp-def length-transpose* * ** *** *XS-def*[*symmetric*])
qed
qed *simp-all*

lemma *transpose-map-map*:
transpose (*map* (*map f*) *xs*) = *map* (*map f*) (*transpose xs*)
proof (*rule nth-equalityI*)
have [*simp*]: *length* (*transpose* (*map* (*map f*) *xs*)) = *length* (*transpose xs*)
by (*simp add*: *length-transpose foldr-map comp-def*)
show *length* (*transpose* (*map* (*map f*) *xs*)) = *length* (*map* (*map f*) (*transpose xs*))
by *simp*

fix *i* **assume** *i* < *length* (*transpose* (*map* (*map f*) *xs*))
thus *transpose* (*map* (*map f*) *xs*) ! *i* = *map* (*map f*) (*transpose xs*) ! *i*
by (*simp add*: *nth-transpose filter-map comp-def*)
qed

66.2.19 *min* and *arg-min*

lemma *min-list-Min*: *xs* ≠ [] ⟹ *min-list xs* = *Min* (*set xs*)
by (*induction xs rule*: *induct-list012*)(*auto*)

lemma *f-arg-min-list-f*: *xs* ≠ [] ⟹ *f* (*arg-min-list f xs*) = *Min* (*f* ‘ (*set xs*))
by(*induction f xs rule*: *arg-min-list.induct*) (*auto simp*: *min-def intro*!: *antisym*)

lemma *arg-min-list-in*: *xs* ≠ [] ⟹ *arg-min-list f xs* ∈ *set xs*
by(*induction xs rule*: *induct-list012*) (*auto simp*: *Let-def*)

66.2.20 (In)finiteness

lemma *finite-list-length*: *finite* {*xs*::(*'a*::*finite*) *list*. *length xs* = *n*}
proof(*induction n*)
case (*Suc n*)
have {*xs*::*'a list*. *length xs* = *Suc n*} = (⋃ *x*. (#) *x* ‘ {*xs*. *length xs* = *n*})
by (*auto simp*: *length-Suc-conv*)
then show *?case* **using** *Suc* **by** *simp*
qed *simp*

lemma *finite-maxlen*:
finite (*M*::*'a list set*) ⟹ ∃ *n*. ∀ *s*∈*M*. *size s* < *n*
proof (*induct rule*: *finite.induct*)
case *emptyI* **show** *?case* **by** *simp*
next
case (*insertI M xs*)

then obtain *n* **where** $\forall s{\in}M.\ length\ s < n$ **by** *blast*
hence $\forall s{\in}insert\ xs\ M.\ size\ s < max\ n\ (size\ xs) + 1$ **by** *auto*
thus *?case* **..**
qed

lemma *lists-length-Suc-eq*:
$\{xs.\ set\ xs \subseteq A \wedge length\ xs = Suc\ n\} =$
$(\lambda(xs,\ n).\ n\#xs)\ `\ (\{xs.\ set\ xs \subseteq A \wedge length\ xs = n\} \times A)$
by (*auto simp*: *length-Suc-conv*)

lemma
assumes *finite A*
shows *finite-lists-length-eq*: *finite* $\{xs.\ set\ xs \subseteq A \wedge length\ xs = n\}$
and *card-lists-length-eq*: *card* $\{xs.\ set\ xs \subseteq A \wedge length\ xs = n\} = (card\ A)\hat{}n$
using ‹*finite A*›
by (*induct n*)
(*auto simp*: *card-image inj-split-Cons lists-length-Suc-eq cong*: *conj-cong*)

lemma *finite-lists-length-le*:
assumes *finite A* **shows** *finite* $\{xs.\ set\ xs \subseteq A \wedge length\ xs \leq n\}$
(**is** *finite ?S*)
proof−
have $?S = (\bigcup n{\in}\{0..n\}.\ \{xs.\ set\ xs \subseteq A \wedge length\ xs = n\})$ **by** *auto*
thus *?thesis* **by** (*auto intro*!: *finite-lists-length-eq*[*OF* ‹*finite A*›] *simp only*:)
qed

lemma *card-lists-length-le*:
assumes *finite A* **shows** *card* $\{xs.\ set\ xs \subseteq A \wedge length\ xs \leq n\} = (\sum i{\leq}n.\ card\ A\hat{}i)$
proof −
have $(\sum i{\leq}n.\ card\ A\hat{}i) = card\ (\bigcup i{\leq}n.\ \{xs.\ set\ xs \subseteq A \wedge length\ xs = i\})$
using ‹*finite A*›
by (*subst card-UN-disjoint*)
(*auto simp add*: *card-lists-length-eq finite-lists-length-eq*)
also have $(\bigcup i{\leq}n.\ \{xs.\ set\ xs \subseteq A \wedge length\ xs = i\}) = \{xs.\ set\ xs \subseteq A \wedge length\ xs \leq n\}$
by *auto*
finally show *?thesis* **by** *simp*
qed

lemma *finite-subset-distinct*:
assumes *finite A*
shows *finite* $\{xs.\ set\ xs \subseteq A \wedge distinct\ xs\}$ (**is** *finite ?S*)
proof (*rule finite-subset*)
from *assms* **show** $?S \subseteq \{xs.\ set\ xs \subseteq A \wedge length\ xs \leq card\ A\}$
by *clarsimp* (*metis distinct-card card-mono*)
from *assms* **show** *finite ...* **by** (*rule finite-lists-length-le*)
qed

lemma *card-lists-distinct-length-eq*:
assumes *finite A* $k \leq$ *card A*
shows *card* $\{xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq A\} = \prod \{card\ A - k + 1\ ..\ card\ A\}$
using *assms*
proof (*induct k*)
case *0*
then have $\{xs.\ length\ xs = 0 \wedge distinct\ xs \wedge set\ xs \subseteq A\} = \{[]\}$ **by** *auto*
then show *?case* **by** *simp*
next
case (*Suc k*)
let *?k-list* = $\lambda k\ xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq A$
have *inj-Cons*: $\bigwedge A.$ *inj-on* $(\lambda(xs,\ n).\ n\ \#\ xs)\ A$ **by** (*rule inj-onI*) *auto*

from *Suc* **have** $k \leq$ *card A* **by** *simp*
moreover note ‹*finite A*›
moreover have *finite* $\{xs.\ ?k\text{-}list\ k\ xs\}$
by (*rule finite-subset*) (*use finite-lists-length-eq*[*OF* ‹*finite A*›, *of k*] **in** *auto*)
moreover have $\bigwedge i\ j.\ i \neq j \longrightarrow \{i\} \times (A - set\ i) \cap \{j\} \times (A - set\ j) = \{\}$
by *auto*
moreover have $\bigwedge i.\ i \in \{xs.\ ?k\text{-}list\ k\ xs\} \Longrightarrow card\ (A - set\ i) = card\ A - k$
by (*simp add*: *card-Diff-subset distinct-card*)
moreover have $\{xs.\ ?k\text{-}list\ (Suc\ k)\ xs\} =$
$(\lambda(xs,\ n).\ n\#xs)\ `\ \bigcup((\lambda xs.\ \{xs\} \times (A - set\ xs))\ `\ \{xs.\ ?k\text{-}list\ k\ xs\})$
by (*auto simp*: *length-Suc-conv*)
moreover have *Suc* (*card A* − *Suc k*) = *card A* − *k* **using** *Suc.prems* **by** *simp*
then have (*card A* − *k*) ∗ $\prod \{Suc\ (card\ A - k)..card\ A\} = \prod \{Suc\ (card\ A - Suc\ k)..card\ A\}$
by (*subst prod.insert*[*symmetric*]) (*simp add*: *atLeastAtMost-insertL*)+
ultimately show *?case*
by (*simp add*: *card-image inj-Cons card-UN-disjoint Suc.hyps algebra-simps*)
qed

lemma *card-lists-distinct-length-eq′*:
assumes *k* < *card A*
shows *card* $\{xs.\ length\ xs = k \wedge distinct\ xs \wedge set\ xs \subseteq A\} = \prod \{card\ A - k + 1\ ..\ card\ A\}$
proof −
from ‹*k* < *card A*› **have** *finite A* **and** *k* ≤ *card A* **using** *card.infinite* **by** *force*+
from *this* **show** *?thesis* **by** (*rule card-lists-distinct-length-eq*)
qed

lemma *infinite-UNIV-listI*: ¬ *finite*(*UNIV*::′*a list set*)
by (*metis UNIV-I finite-maxlen length-replicate less-irrefl*)

lemma *same-length-different*:
assumes *xs* ≠ *ys* **and** *length xs* = *length ys*
shows $\exists pre\ x\ xs'\ y\ ys'.\ x{\neq}y \wedge xs = pre\ @\ [x]\ @\ xs' \wedge ys = pre\ @\ [y]\ @\ ys'$
using *assms*

proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons x xs*)
then obtain *z zs* **where** *ys*: *ys* = *Cons z zs*
by (*metis length-Suc-conv*)
show *?case*
proof (*cases x=z*)
case *True*
then have *xs* ≠ *zs length xs* = *length zs*
using *Cons.prems ys* **by** *auto*
then obtain *pre u xs′ v ys′* **where** *u≠v* **and** *xs*: *xs* = *pre* @ [*u*] @ *xs′* **and** *zs*: *zs* = *pre* @ [*v*] @*ys′*
using *Cons.IH* **by** *meson*
then have *x* # *xs* = (*z*#*pre*) @ [*u*] @ *xs′* ∧ *ys* = (*z*#*pre*) @ [*v*] @ *ys′*
by (*simp add*: *True ys*)
with ‹*u≠v*› **show** *?thesis*
by *blast*
next
case *False*
then have *x* # *xs* = [] @ [*x*] @ *xs* ∧ *ys* = [] @ [*z*] @ *zs*
by (*simp add*: *ys*)
then show *?thesis*
using *False* **by** *blast*
qed
qed

66.3 Sorting

66.3.1 *sorted-wrt*

Sometimes the second equation in the definition of *sorted-wrt* is too aggressive because it relates each list element to *all* its successors. Then this equation should be removed and *sorted-wrt2-simps* should be added instead.

lemma *sorted-wrt1*: *sorted-wrt P* [*x*] = *True*
by(*simp*)

lemma *sorted-wrt2*: *transp P* ⟹ *sorted-wrt P* (*x* # *y* # *zs*) = (*P x y* ∧ *sorted-wrt P* (*y* # *zs*))
proof (*induction zs arbitrary*: *x y*)
case (*Cons z zs*)
then show *?case*
by *simp* (*meson transpD*)+
qed *auto*

lemmas *sorted-wrt2-simps* = *sorted-wrt1 sorted-wrt2*

lemma *sorted-wrt-true* [*simp*]:

sorted-wrt (λ- -. *True*) *xs*
by (*induction xs*) *simp-all*

lemma *sorted-wrt-append*:
sorted-wrt P (*xs* @ *ys*) ⟷
sorted-wrt P xs ∧ *sorted-wrt P ys* ∧ (∀ *x*∈*set xs*. ∀ *y*∈*set ys*. *P x y*)
by (*induction xs*) *auto*

lemma *sorted-wrt-map*:
sorted-wrt R (*map f xs*) = *sorted-wrt* (λ*x y*. *R* (*f x*) (*f y*)) *xs*
by (*induction xs*) *simp-all*

lemma
assumes *sorted-wrt f xs*
shows *sorted-wrt-take*[*simp*]: *sorted-wrt f* (*take n xs*)
and *sorted-wrt-drop*[*simp*]: *sorted-wrt f* (*drop n xs*)
proof −
from *assms* **have** *sorted-wrt f* (*take n xs* @ *drop n xs*) **by** *simp*
thus *sorted-wrt f* (*take n xs*) **and** *sorted-wrt f* (*drop n xs*)
unfolding *sorted-wrt-append* **by** *simp-all*
qed

lemma *sorted-wrt-dropWhile*[*simp*]: *sorted-wrt R xs* ⟹ *sorted-wrt R* (*dropWhile P xs*)
by (*auto dest*: *sorted-wrt-drop simp*: *dropWhile-eq-drop*)

lemma *sorted-wrt-takeWhile*[*simp*]: *sorted-wrt R xs* ⟹ *sorted-wrt R* (*takeWhile P xs*)
by (*subst takeWhile-eq-take*) (*auto dest*: *sorted-wrt-take*)

lemma *sorted-wrt-filter*:
sorted-wrt f xs ⟹ *sorted-wrt f* (*filter P xs*)
by (*induction xs*) *auto*

lemma *sorted-wrt-rev*:
sorted-wrt P (*rev xs*) = *sorted-wrt* (λ*x y*. *P y x*) *xs*
by (*induction xs*) (*auto simp add*: *sorted-wrt-append*)

lemma *sorted-wrt-mono-rel*:
(⋀*x y*. ⟦ *x* ∈ *set xs*; *y* ∈ *set xs*; *P x y* ⟧ ⟹ *Q x y*) ⟹ *sorted-wrt P xs* ⟹ *sorted-wrt Q xs*
by(*induction xs*)(*auto*)

lemma *sorted-wrt01*: *length xs* ≤ *1* ⟹ *sorted-wrt P xs*
by(*auto simp*: *le-Suc-eq length-Suc-conv*)

lemma *sorted-wrt-iff-nth-less*:
sorted-wrt P xs = (∀ *i j*. *i* < *j* ⟶ *j* < *length xs* ⟶ *P* (*xs* ! *i*) (*xs* ! *j*))
by (*induction xs*) (*auto simp add*: *in-set-conv-nth Ball-def nth-Cons split*: *nat.split*)

lemma *sorted-wrt-nth-less*:
⟦ *sorted-wrt P xs*; *i* < *j*; *j* < *length xs* ⟧ ⟹ *P* (*xs* ! *i*) (*xs* ! *j*)
by(*auto simp*: *sorted-wrt-iff-nth-less*)

lemma *sorted-wrt-iff-nth-Suc-transp*: **assumes** *transp P*
shows *sorted-wrt P xs* ⟷ (∀ *i*. *Suc i* < *length xs* ⟶ *P* (*xs*!*i*) (*xs*!(*Suc i*))) (**is** *?L* = *?R*)
proof
assume *?L*
thus *?R*
by (*simp add*: *sorted-wrt-iff-nth-less*)
next
assume *?R*
have *i* < *j* ⟹ *j* < *length xs* ⟹ *P* (*xs* ! *i*) (*xs* ! *j*) **for** *i j*
by(*induct i j rule*: *less-Suc-induct*)(*simp add*: ‹*?R*›, *meson assms transpE transp-on-less*)
thus *?L*
by (*simp add*: *sorted-wrt-iff-nth-less*)
qed

lemma *sorted-wrt-upt*[*simp*]: *sorted-wrt* (<) [*m*..<*n*]
by(*induction n*) (*auto simp*: *sorted-wrt-append*)

lemma *sorted-wrt-upto*[*simp*]: *sorted-wrt* (<) [*i*..*j*]
proof(*induct i j rule*:*upto.induct*)
case (*1 i j*)
from *this* **show** *?case*
unfolding *upto.simps*[*of i j*] **by** *auto*
qed

Each element is greater or equal to its index:

lemma *sorted-wrt-less-idx*:
sorted-wrt (<) *ns* ⟹ *i* < *length ns* ⟹ *i* ≤ *ns*!*i*
proof (*induction ns arbitrary*: *i rule*: *rev-induct*)
case *Nil* **thus** *?case* **by** *simp*
next
case *snoc*
thus *?case*
by (*simp add*: *nth-append sorted-wrt-append*)
(*metis less-antisym not-less nth-mem*)
qed

66.3.2 *sorted*

context *linorder*
begin

Sometimes the second equation in the definition of *sorted* is too aggressive because it relates each list element to *all* its successors. Then this equation

should be removed and *sorted2-simps* should be added instead. Executable code is one such use case.

lemma *sorted0*: *sorted* [] = *True*
by *simp*

lemma *sorted1*: *sorted* [*x*] = *True*
by *simp*

lemma *sorted2*: *sorted* (*x* # *y* # *zs*) = (*x* ≤ *y* ∧ *sorted* (*y* # *zs*))
by *auto*

lemmas *sorted2-simps* = *sorted1 sorted2*

lemma *sorted-append*:
sorted (*xs*@*ys*) = (*sorted xs* ∧ *sorted ys* ∧ (∀ *x* ∈ *set xs*. ∀ *y* ∈ *set ys*. *x*≤*y*))
by (*simp add*: *sorted-wrt-append*)

lemma *sorted-map*:
sorted (*map f xs*) = *sorted-wrt* (λ*x y*. *f x* ≤ *f y*) *xs*
by (*simp add*: *sorted-wrt-map*)

lemma *sorted01*: *length xs* ≤ *1* ⟹ *sorted xs*
by (*simp add*: *sorted-wrt01*)

lemma *sorted-tl*:
sorted xs ⟹ *sorted* (*tl xs*)
by (*cases xs*) (*simp-all*)

lemma *sorted-iff-nth-mono-less*:
sorted xs = (∀ *i j*. *i* < *j* ⟶ *j* < *length xs* ⟶ *xs* ! *i* ≤ *xs* ! *j*)
by (*simp add*: *sorted-wrt-iff-nth-less*)

lemma *sorted-iff-nth-mono*:
sorted xs = (∀ *i j*. *i* ≤ *j* ⟶ *j* < *length xs* ⟶ *xs* ! *i* ≤ *xs* ! *j*)
by (*auto simp*: *sorted-iff-nth-mono-less nat-less-le*)

lemma *sorted-nth-mono*:
sorted xs ⟹ *i* ≤ *j* ⟹ *j* < *length xs* ⟹ *xs*!*i* ≤ *xs*!*j*
by (*auto simp*: *sorted-iff-nth-mono*)

lemma *sorted-iff-nth-Suc*:
sorted xs ⟷ (∀ *i*. *Suc i* < *length xs* ⟶ *xs*!*i* ≤ *xs*!(*Suc i*))
by(*simp add*: *sorted-wrt-iff-nth-Suc-transp*)

lemma *sorted-rev-nth-mono*:
sorted (*rev xs*) ⟹ *i* ≤ *j* ⟹ *j* < *length xs* ⟹ *xs*!*j* ≤ *xs*!*i*
by (*metis local.nle-le order-class.antisym-conv1 sorted-wrt-iff-nth-less sorted-wrt-rev*)

lemma *sorted-rev-iff-nth-mono*:

sorted (*rev xs*) ⟷ (∀ *i j*. *i* ≤ *j* ⟶ *j* < *length xs* ⟶ *xs*!*j* ≤ *xs*!*i*) (**is** *?L* = *?R*)
proof
assume *?L* **thus** *?R*
by (*blast intro*: *sorted-rev-nth-mono*)
next
assume *?R*
have *rev xs* ! *k* ≤ *rev xs* ! *l* **if** *asms*: *k* ≤ *l l* < *length*(*rev xs*) **for** *k l*
proof −
have *k* < *length xs* *l* < *length xs*
length xs − *Suc l* ≤ *length xs* − *Suc k length xs* − *Suc k* < *length xs*
using *asms* **by** *auto*
thus *rev xs* ! *k* ≤ *rev xs* ! *l*
by (*simp add*: ‹*?R*› *rev-nth*)
qed
thus *?L* **by** (*simp add*: *sorted-iff-nth-mono*)
qed

lemma *sorted-rev-iff-nth-Suc*:
sorted (*rev xs*) ⟷ (∀ *i*. *Suc i* < *length xs* ⟶ *xs*!(*Suc i*) ≤ *xs*!*i*)
proof−
interpret *dual*: *linorder* (λ*x y*. *y* ≤ *x*) (λ*x y*. *y* < *x*)
using *dual-linorder* **.**
show *?thesis*
using *dual-linorder dual.sorted-iff-nth-Suc dual.sorted-iff-nth-mono*
unfolding *sorted-rev-iff-nth-mono* **by** *simp*
qed

lemma *sorted-map-remove1*:
sorted (*map f xs*) ⟹ *sorted* (*map f* (*remove1 x xs*))
by (*induct xs*) (*auto*)

lemma *sorted-remove1*: *sorted xs* ⟹ *sorted* (*remove1 a xs*)
using *sorted-map-remove1* [*of* λ*x*. *x*] **by** *simp*

lemma *sorted-butlast*:
assumes *sorted xs*
shows *sorted* (*butlast xs*)
by (*simp add*: *assms butlast-conv-take*)

lemma *sorted-replicate* [*simp*]: *sorted*(*replicate n x*)
by(*induction n*) (*auto*)

lemma *sorted-remdups*[*simp*]:
sorted xs ⟹ *sorted* (*remdups xs*)
by (*induct xs*) (*auto*)

lemma *sorted-remdups-adj*[*simp*]:
sorted xs ⟹ *sorted* (*remdups-adj xs*)
by (*induct xs rule*: *remdups-adj.induct*, *simp-all split*: *if-split-asm*)

lemma *sorted-nths*: *sorted xs* $\Longrightarrow$ *sorted* (*nths xs I*)
by(*induction xs arbitrary*: *I*)(*auto simp*: *nths-Cons*)

lemma *sorted-distinct-set-unique*:
assumes *sorted xs distinct xs sorted ys distinct ys set xs* = *set ys*
shows *xs* = *ys*
proof −
  from *assms* **have** *1*: *length xs* = *length ys* **by** (*auto dest*!: *distinct-card*)
  from *assms* **show** *?thesis*
  proof(*induct rule*:*list-induct2*[*OF 1*])
    case *1* **show** *?case* **by** *simp*
  next
    case (*2 x xs y ys*)
    then show *?case*
      by (*cases* ‹*x* = *y*›) (*auto simp add*: *insert-eq-iff*)
  qed
qed

lemma *map-sorted-distinct-set-unique*:
  assumes *inj-on f* (*set xs* ∪ *set ys*)
  assumes *sorted* (*map f xs*) *distinct* (*map f xs*)
    sorted (*map f ys*) *distinct* (*map f ys*)
  assumes *set xs* = *set ys*
  shows *xs* = *ys*
  using *assms map-inj-on sorted-distinct-set-unique* **by** *fastforce*

lemma *sorted-dropWhile*: *sorted xs* $\Longrightarrow$ *sorted* (*dropWhile P xs*)
  by (*auto dest*: *sorted-wrt-drop simp add*: *dropWhile-eq-drop*)

lemma *sorted-takeWhile*: *sorted xs* $\Longrightarrow$ *sorted* (*takeWhile P xs*)
  by (*subst takeWhile-eq-take*) (*auto dest*: *sorted-wrt-take*)

lemma *sorted-filter*:
  sorted (*map f xs*) $\Longrightarrow$ *sorted* (*map f* (*filter P xs*))
  by (*induct xs*) *simp-all*

lemma *foldr-max-sorted*:
  assumes *sorted* (*rev xs*)
  shows *foldr max xs y* = (*if xs* = [] *then y else max* (*xs* ! *0*) *y*)
  using *assms*
proof (*induct xs*)
  case (*Cons x xs*)
  then have *sorted* (*rev xs*) **using** *sorted-append* **by** *auto*
  with *Cons* **show** *?case*
    by (*cases xs*) (*auto simp add*: *sorted-append max-def*)
qed *simp*

lemma *filter-equals-takeWhile-sorted-rev*:

assumes *sorted*: *sorted* (*rev* (*map f xs*))
shows *filter* (λ*x*. *t* < *f x*) *xs* = *takeWhile* (λ *x*. *t* < *f x*) *xs*
(**is** *filter ?P xs* = *?tW*)
proof (*rule takeWhile-eq-filter*[*symmetric*])
let *?dW* = *dropWhile ?P xs*
fix *x* **assume** *x*: *x* ∈ *set ?dW*
then obtain *i* **where** *i*: *i* < *length ?dW* **and** *nth-i*: *x* = *?dW* ! *i*
unfolding *in-set-conv-nth* **by** *auto*
hence *length ?tW* + *i* < *length* (*?tW* @ *?dW*)
unfolding *length-append* **by** *simp*
hence *i'*: *length* (*map f ?tW*) + *i* < *length* (*map f xs*) **by** *simp*
have (*map f ?tW* @ *map f ?dW*) ! (*length* (*map f ?tW*) + *i*) ≤
(*map f ?tW* @ *map f ?dW*) ! (*length* (*map f ?tW*) + *0*)
using *sorted-rev-nth-mono*[*OF sorted - i'*, *of length ?tW*]
unfolding *map-append*[*symmetric*] **by** *simp*
hence *f x* ≤ *f* (*?dW* ! *0*)
unfolding *nth-append-length-plus nth-i*
using *i preorder-class.le-less-trans*[*OF le0 i*] **by** *simp*
also have ... ≤ *t*
by (*metis hd-conv-nth hd-dropWhile length-greater-0-conv length-pos-if-in-set local.leI x*)
finally show ¬ *t* < *f x* **by** *simp*
qed

lemma *sorted-map-same*:
sorted (*map f* (*filter* (λ*x*. *f x* = *g xs*) *xs*))
proof (*induct xs arbitrary*: *g*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*)
then have *sorted* (*map f* (*filter* (λ*y*. *f y* = (λ*xs*. *f x*) *xs*) *xs*)) **.**
moreover from *Cons* **have** *sorted* (*map f* (*filter* (λ*y*. *f y* = (*g* ∘ *Cons x*) *xs*) *xs*)) **.**
ultimately show *?case* **by** *simp-all*
qed

lemma *sorted-same*:
sorted (*filter* (λ*x*. *x* = *g xs*) *xs*)
using *sorted-map-same* [*of* λ*x*. *x*] **by** *simp*

end

lemma *sorted-upt*[*simp*]: *sorted* [*m*..<*n*]
by(*simp add*: *sorted-wrt-mono-rel*[*OF - sorted-wrt-upt*])

lemma *sorted-upto*[*simp*]: *sorted* [*m*..*n*]
by(*simp add*: *sorted-wrt-mono-rel*[*OF - sorted-wrt-upto*])

66.3.3 Sorting functions

Currently it is not shown that *sort* returns a permutation of its input because the nicest proof is via multisets, which are not part of Main. Alternatively one could define a function that counts the number of occurrences of an element in a list and use that instead of multisets to state the correctness property.

context *linorder*
begin

lemma *set-insort-key*:
  set (*insort-key f x xs*) = *insert x* (*set xs*)
by (*induct xs*) *auto*

lemma *length-insort* [*simp*]:
  length (*insort-key f x xs*) = *Suc* (*length xs*)
by (*induct xs*) *simp-all*

lemma *insort-key-left-comm*:
  assumes *f x* ≠ *f y*
  shows *insort-key f y* (*insort-key f x xs*) = *insort-key f x* (*insort-key f y xs*)
by (*induct xs*) (*auto simp add*: *assms dest*: *order.antisym*)

lemma *insort-left-comm*:
  insort x (*insort y xs*) = *insort y* (*insort x xs*)
by (*cases x* = *y*) (*auto intro*: *insort-key-left-comm*)

lemma *comp-fun-commute-insort*: *comp-fun-commute insort*
proof
qed (*simp add*: *insort-left-comm fun-eq-iff*)

lemma *sort-key-simps* [*simp*]:
  sort-key f [] = []
  sort-key f (*x*#*xs*) = *insort-key f x* (*sort-key f xs*)
by (*simp-all add*: *sort-key-def*)

lemma *sort-key-conv-fold*:
  assumes *inj-on f* (*set xs*)
  shows *sort-key f xs* = *fold* (*insort-key f*) *xs* []
proof −
  have *fold* (*insort-key f*) (*rev xs*) = *fold* (*insort-key f*) *xs*
  proof (*rule fold-rev*, *rule ext*)
    fix *zs*
    fix *x y*
    assume *x* ∈ *set xs y* ∈ *set xs*
    with *assms* **have** ∗: *f y* = *f x* ⟹ *y* = *x* **by** (*auto dest*: *inj-onD*)
    have ∗∗: *x* = *y* ⟷ *y* = *x* **by** *auto*
    show (*insort-key f y* ∘ *insort-key f x*) *zs* = (*insort-key f x* ∘ *insort-key f y*) *zs*

by (*induct zs*) (*auto intro*: ∗ *simp add*: ∗∗)
qed
then show *?thesis* **by** (*simp add*: *sort-key-def foldr-conv-fold*)
qed

lemma *sort-conv-fold*:
sort xs = *fold insort xs* []
by (*rule sort-key-conv-fold*) *simp*

lemma *length-sort*[*simp*]: *length* (*sort-key f xs*) = *length xs*
by (*induct xs*, *auto*)

lemma *set-sort*[*simp*]: *set*(*sort-key f xs*) = *set xs*
by (*induct xs*) (*simp-all add*: *set-insort-key*)

lemma *distinct-insort*: *distinct* (*insort-key f x xs*) = (*x* ∉ *set xs* ∧ *distinct xs*)
by(*induct xs*)(*auto simp*: *set-insort-key*)

lemma *distinct-insort-key*:
distinct (*map f* (*insort-key f x xs*)) = (*f x* ∉ *f* ' *set xs* ∧ (*distinct* (*map f xs*)))
by (*induct xs*) (*auto simp*: *set-insort-key*)

lemma *distinct-sort*[*simp*]: *distinct* (*sort-key f xs*) = *distinct xs*
by (*induct xs*) (*simp-all add*: *distinct-insort*)

lemma *sorted-insort-key*: *sorted* (*map f* (*insort-key f x xs*)) = *sorted* (*map f xs*)
by (*induct xs*) (*auto simp*: *set-insort-key*)

lemma *sorted-insort*: *sorted* (*insort x xs*) = *sorted xs*
using *sorted-insort-key* [**where** *f*=λ*x*. *x*] **by** *simp*

theorem *sorted-sort-key* [*simp*]: *sorted* (*map f* (*sort-key f xs*))
by (*induct xs*) (*auto simp*:*sorted-insort-key*)

theorem *sorted-sort* [*simp*]: *sorted* (*sort xs*)
using *sorted-sort-key* [**where** *f*=λ*x*. *x*] **by** *simp*

lemma *insort-not-Nil* [*simp*]:
insort-key f a xs ≠ []
by (*induction xs*) *simp-all*

lemma *insort-is-Cons*: ∀ *x*∈*set xs*. *f a* ≤ *f x* ⟹ *insort-key f a xs* = *a* # *xs*
by (*cases xs*) *auto*

lemma *sort-key-id-if-sorted*: *sorted* (*map f xs*) ⟹ *sort-key f xs* = *xs*
by (*induction xs*) (*auto simp add*: *insort-is-Cons*)

Subsumed by *sorted* (*map ?f ?xs*) ⟹ *sort-key ?f ?xs* = *?xs* but easier to find:

lemma *sorted-sort-id*: *sorted xs* $\Longrightarrow$ *sort xs* = *xs*
by (*simp add*: *sort-key-id-if-sorted*)

lemma *sort-replicate* [*simp*]: *sort* (*replicate n x*) = *replicate n x*
using *sorted-replicate sorted-sort-id*
by *presburger*

lemma *insort-key-remove1*:
assumes *a* ∈ *set xs* **and** *sorted* (*map f xs*) **and** *hd* (*filter* (λ*x*. *f a* = *f x*) *xs*) = *a*
shows *insort-key f a* (*remove1 a xs*) = *xs*
using *assms* **proof** (*induct xs*)
case (*Cons x xs*)
then show *?case*
proof (*cases x* = *a*)
case *False*
then have *f x* ≠ *f a* **using** *Cons.prems* **by** *auto*
then have *f x* < *f a* **using** *Cons.prems* **by** *auto*
with ‹*f x* ≠ *f a*› **show** *?thesis* **using** *Cons* **by** (*auto simp*: *insort-is-Cons*)
qed (*auto simp*: *insort-is-Cons*)
qed *simp*

lemma *insort-remove1*:
assumes *a* ∈ *set xs* **and** *sorted xs*
shows *insort a* (*remove1 a xs*) = *xs*
proof (*rule insort-key-remove1*)
define *n* **where** *n* = *length* (*filter* ((=) *a*) *xs*) − *1*
from ‹*a* ∈ *set xs*› **show** *a* ∈ *set xs* **.**
from ‹*sorted xs*› **show** *sorted* (*map* (λ*x*. *x*) *xs*) **by** *simp*
from ‹*a* ∈ *set xs*› **have** *a* ∈ *set* (*filter* ((=) *a*) *xs*) **by** *auto*
then have *set* (*filter* ((=) *a*) *xs*) ≠ {} **by** *auto*
then have *filter* ((=) *a*) *xs* ≠ [] **by** (*auto simp only*: *set-empty*)
then have *length* (*filter* ((=) *a*) *xs*) > *0* **by** *simp*
then have *n*: *Suc n* = *length* (*filter* ((=) *a*) *xs*) **by** (*simp add*: *n-def*)
moreover have *replicate* (*Suc n*) *a* = *a* # *replicate n a*
by *simp*
ultimately show *hd* (*filter* ((=) *a*) *xs*) = *a* **by** (*simp add*: *replicate-length-filter*)
qed

lemma *finite-sorted-distinct-unique*:
assumes *finite A* **shows** ∃!*xs*. *set xs* = *A* ∧ *sorted xs* ∧ *distinct xs*
proof −
obtain *xs* **where** *distinct xs A* = *set xs*
using *finite-distinct-list* [*OF assms*] **by** *metis*
then show *?thesis*
by (*rule-tac a*=*sort xs* **in** *ex1I*) (*auto simp*: *sorted-distinct-set-unique*)
qed

lemma *insort-insert-key-triv*:
f x ∈ *f* ‘ *set xs* $\Longrightarrow$ *insort-insert-key f x xs* = *xs*

by (*simp add*: *insort-insert-key-def*)

lemma *insort-insert-triv*:
x ∈ *set xs* ⟹ *insort-insert x xs* = *xs*
using *insort-insert-key-triv* [*of* λ*x*. *x*] **by** *simp*

lemma *insort-insert-insort-key*:
f x ∉ *f* ' *set xs* ⟹ *insort-insert-key f x xs* = *insort-key f x xs*
by (*simp add*: *insort-insert-key-def*)

lemma *insort-insert-insort*:
x ∉ *set xs* ⟹ *insort-insert x xs* = *insort x xs*
using *insort-insert-insort-key* [*of* λ*x*. *x*] **by** *simp*

lemma *set-insort-insert*:
set (*insort-insert x xs*) = *insert x* (*set xs*)
by (*auto simp add*: *insort-insert-key-def set-insort-key*)

lemma *distinct-insort-insert*:
assumes *distinct xs*
shows *distinct* (*insort-insert-key f x xs*)
using *assms* **by** (*induct xs*) (*auto simp add*: *insort-insert-key-def set-insort-key*)

lemma *sorted-insort-insert-key*:
assumes *sorted* (*map f xs*)
shows *sorted* (*map f* (*insort-insert-key f x xs*))
using *assms* **by** (*simp add*: *insort-insert-key-def sorted-insort-key*)

lemma *sorted-insort-insert*:
assumes *sorted xs*
shows *sorted* (*insort-insert x xs*)
using *assms sorted-insort-insert-key* [*of* λ*x*. *x*] **by** *simp*

lemma *filter-insort-triv*:
¬ *P x* ⟹ *filter P* (*insort-key f x xs*) = *filter P xs*
by (*induct xs*) *simp-all*

lemma *filter-insort*:
sorted (*map f xs*) ⟹ *P x* ⟹ *filter P* (*insort-key f x xs*) = *insort-key f x* (*filter P xs*)
by (*induct xs*) (*auto*, *subst insort-is-Cons*, *auto*)

lemma *filter-sort*:
filter P (*sort-key f xs*) = *sort-key f* (*filter P xs*)
by (*induct xs*) (*simp-all add*: *filter-insort-triv filter-insort*)

lemma *remove1-insort-key* [*simp*]:
remove1 x (*insort-key f x xs*) = *xs*
by (*induct xs*) *simp-all*

end

lemma *sort-upt* [*simp*]: *sort* [*m*..<*n*] = [*m*..<*n*]
by (*rule sort-key-id-if-sorted*) *simp*

lemma *sort-upto* [*simp*]: *sort* [*i*..*j*] = [*i*..*j*]
by (*rule sort-key-id-if-sorted*) *simp*

lemma *sorted-find-Min*:
sorted xs ⟹ ∃ *x* ∈ *set xs*. *P x* ⟹ *List.find P xs* = *Some* (*Min* {*x*∈*set xs*. *P x*})
proof (*induct xs*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* **proof** (*cases P x*)
case *True*
with *Cons* **show** *?thesis* **by** (*auto intro*: *Min-eqI* [*symmetric*])
next
case *False* **then have** {*y*. (*y* = *x* ∨ *y* ∈ *set xs*) ∧ *P y*} = {*y* ∈ *set xs*. *P y*}
by *auto*
with *Cons False* **show** *?thesis* **by** (*simp-all*)
qed
qed

lemma *sorted-indexed-from* [*simp*]: *sorted* (*map fst* (*indexed-from n xs*))
by (*simp add*: *indexed-from-eq-zip*)

lemma *sorted-insort-is-snoc*: *sorted xs* ⟹ ∀ *x* ∈ *set xs*. *a* ≥ *x* ⟹ *insort a xs* = *xs* @ [*a*]
by (*induct xs*) (*auto dest*!: *insort-is-Cons*)

Stability of *sort-key*:

lemma *sort-key-stable*: *filter* (λ*y*. *f y* = *k*) (*sort-key f xs*) = *filter* (λ*y*. *f y* = *k*) *xs*
by (*induction xs*) (*auto simp*: *filter-insort insort-is-Cons filter-insort-triv*)

corollary *stable-sort-key-sort-key*: *stable-sort-key sort-key*
by(*simp add*: *stable-sort-key-def sort-key-stable*)

lemma *sort-key-const*: *sort-key* (λ*x*. *c*) *xs* = *xs*
by (*metis* (*mono-tags*) *filter-True sort-key-stable*)

66.3.4 *transpose* on sorted lists

lemma *sorted-transpose*[*simp*]: *sorted* (*rev* (*map length* (*transpose xs*)))
by (*auto simp*: *sorted-iff-nth-mono rev-nth nth-transpose*
length-filter-conv-card intro: *card-mono*)

lemma *transpose-max-length*:
foldr (λ*xs*. *max* (*length xs*)) (*transpose xs*) *0* = *length* (*filter* (λ*x*. *x* ≠ []) *xs*)

```
  (is ?L = ?R)
proof (cases transpose xs = [])
  case False
  have ?L = foldr max (map length (transpose xs)) 0
    by (simp add: foldr-map comp-def)
  also have ... = length (transpose xs ! 0)
    using False sorted-transpose by (simp add: foldr-max-sorted)
  finally show ?thesis
    using False by (simp add: nth-transpose)
next
  case True
  hence filter (λx. x ≠ []) xs = []
    by (auto intro!: filter-False simp: transpose-empty)
  thus ?thesis by (simp add: transpose-empty True)
qed

lemma length-transpose-sorted:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows length (transpose xs) = (if xs = [] then 0 else length (xs ! 0))
proof (cases xs = [])
  case False
  thus ?thesis
    using foldr-max-sorted[OF sorted] False
    unfolding length-transpose foldr-map comp-def
    by simp
qed simp

lemma nth-nth-transpose-sorted[simp]:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  and i: i < length (transpose xs)
  and j: j < length (filter (λys. i < length ys) xs)
  shows transpose xs ! i ! j = xs ! j ! i
using j filter-equals-takeWhile-sorted-rev[OF sorted, of i]
    nth-transpose[OF i] nth-map[OF j]
by (simp add: takeWhile-nth)

lemma transpose-column-length:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs)) and i < length xs
  shows length (filter (λys. i < length ys) (transpose xs)) = length (xs ! i)
proof −
  have xs ≠ [] using ‹i < length xs› by auto
  note filter-equals-takeWhile-sorted-rev[OF sorted, simp]
  { fix j assume j ≤ i
    note sorted-rev-nth-mono[OF sorted, of j i, simplified, OF this ‹i < length xs›]
  } note sortedE = this[consumes 1]
```

```
  have {j. j < length (transpose xs) ∧ i < length (transpose xs ! j)}
    = {..< length (xs ! i)}
  proof safe
    fix j
    assume j < length (transpose xs) and i < length (transpose xs ! j)
    with this(2) nth-transpose[OF this(1)]
    have i < length (takeWhile (λys. j < length ys) xs) by simp
    from nth-mem[OF this] takeWhile-nth[OF this]
    show j < length (xs ! i) by (auto dest: set-takeWhileD)
  next
    fix j assume j < length (xs ! i)
    thus j < length (transpose xs)
      using foldr-max-sorted[OF sorted] ‹xs ≠ []› sortedE[OF le0]
      by (auto simp: length-transpose comp-def foldr-map)

    have Suc i ≤ length (takeWhile (λys. j < length ys) xs)
      using ‹i < length xs› ‹j < length (xs ! i)› less-Suc-eq-le
      by (auto intro!: length-takeWhile-less-P-nth dest!: sortedE)
    with nth-transpose[OF ‹j < length (transpose xs)›]
    show i < length (transpose xs ! j) by simp
  qed
  thus ?thesis by (simp add: length-filter-conv-card)
qed

lemma transpose-column:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs)) and i < length xs
  shows map (λys. ys ! i) (filter (λys. i < length ys) (transpose xs))
    = xs ! i (is ?R = -)
proof (rule nth-equalityI)
  show length: length ?R = length (xs ! i)
    using transpose-column-length[OF assms] by simp

  fix j assume j: j < length ?R
  note * = less-le-trans[OF this, unfolded length-map, OF length-filter-le]
  from j have j-less: j < length (xs ! i) using length by simp
  have i-less-tW: Suc i ≤ length (takeWhile (λys. Suc j ≤ length ys) xs)
  proof (rule length-takeWhile-less-P-nth)
    show Suc i ≤ length xs using ‹i < length xs› by simp
    fix k assume k < Suc i
    hence k ≤ i by auto
    with sorted-rev-nth-mono[OF sorted this] ‹i < length xs›
    have length (xs ! i) ≤ length (xs ! k) by simp
    thus Suc j ≤ length (xs ! k) using j-less by simp
  qed
  have i-less-filter: i < length (filter (λys. j < length ys) xs)
    unfolding filter-equals-takeWhile-sorted-rev[OF sorted, of j]
    using i-less-tW by (simp-all add: Suc-le-eq)
  from j show ?R ! j = xs ! i ! j
```

```
    unfolding filter-equals-takeWhile-sorted-rev[OF sorted-transpose, of i]
   by (simp add: takeWhile-nth nth-nth-transpose-sorted[OF sorted ∗ i-less-filter])
qed

lemma transpose-transpose:
  fixes xs :: 'a list list
  assumes sorted: sorted (rev (map length xs))
  shows transpose (transpose xs) = takeWhile (λx. x ≠ []) xs (is ?L = ?R)
proof −
  have len: length ?L = length ?R
    unfolding length-transpose transpose-max-length
    using filter-equals-takeWhile-sorted-rev[OF sorted, of 0]
    by simp

  { fix i assume i < length ?R
    with less-le-trans[OF - length-takeWhile-le[of - xs]]
    have i < length xs by simp
  } note ∗ = this
  show ?thesis
    by (rule nth-equalityI)
     (simp-all add: len nth-transpose transpose-column[OF sorted] ∗ takeWhile-nth)
qed

theorem transpose-rectangle:
  assumes xs = [] ⟹ n = 0
  assumes rect: ⋀ i. i < length xs ⟹ length (xs ! i) = n
  shows transpose xs = map (λ i. map (λ j. xs ! j ! i) [0..<length xs]) [0..<n]
    (is ?trans = ?map)
proof (rule nth-equalityI)
  have sorted (rev (map length xs))
    by (auto simp: rev-nth rect sorted-iff-nth-mono)
  from foldr-max-sorted[OF this] assms
  show len: length ?trans = length ?map
    by (simp-all add: length-transpose foldr-map comp-def)
  moreover
  { fix i assume i < n hence filter (λys. i < length ys) xs = xs
      using rect by (auto simp: in-set-conv-nth intro!: filter-True) }
  ultimately show ⋀i. i < length (transpose xs) ⟹ ?trans ! i = ?map ! i
    by (auto simp: nth-transpose intro: nth-equalityI)
qed
```

66.3.5 *sorted-key-list-of-set*

This function maps (finite) linearly ordered sets to sorted lists. The linear order is obtained by a key function that maps the elements of the set to a type that is linearly ordered. Warning: in most cases it is not a good idea to convert from sets to lists but one should convert in the other direction (via *set*).

Note: this is a generalisation of the older *sorted-list-of-set* that is obtained by setting the key function to the identity. Consequently, new theorems should be added to the locale below. They should also be aliased to more convenient names for use with *sorted-list-of-set* as seen further below.

definition (**in** *linorder*) *sorted-key-list-of-set* :: $('b \Rightarrow 'a) \Rightarrow 'b\ set \Rightarrow 'b\ list$
  where *sorted-key-list-of-set f* $\equiv$ *folding-on.F* (*insort-key f*) []

locale *folding-insort-key* = *lo?*: *linorder less-eq* :: $'a \Rightarrow 'a \Rightarrow bool$ *less*
  for *less-eq* (**infix** ‹$\preceq$› *50*) **and** *less* (**infix** ‹$\prec$› *50*) +
  fixes *S*
  fixes *f* :: $'b \Rightarrow 'a$
  assumes *inj-on*: *inj-on f S*
begin

lemma *insort-key-commute*:
  $x \in S \Longrightarrow y \in S \Longrightarrow$ *insort-key f y o insort-key f x* = *insort-key f x o insort-key f y*
proof(*rule ext*, *goal-cases*)
  case (*1 xs*)
  with *inj-on* **show** *?case* **by** (*induction xs*) (*auto simp*: *inj-onD*)
qed

sublocale *fold-insort-key*: *folding-on S insort-key f* []
  rewrites *folding-on.F* (*insort-key f*) [] = *sorted-key-list-of-set f*
proof −
  show *folding-on S* (*insort-key f*)
    by *standard* (*simp add*: *insort-key-commute*)
qed (*simp add*: *sorted-key-list-of-set-def*)

lemma *idem-if-sorted-distinct*:
  assumes *set xs* $\subseteq$ *S* **and** *sorted* (*map f xs*) *distinct xs*
  shows *sorted-key-list-of-set f* (*set xs*) = *xs*
proof(*cases S* = {})
  case *True*
  then show *?thesis* **using** ‹*set xs* $\subseteq$ *S*› **by** *auto*
next
  case *False*
  with *assms* **show** *?thesis*
  proof(*induction xs*)
    case (*Cons a xs*)
    with *Cons* **show** *?case* **by** (*cases xs*) *auto*
  qed *simp*
qed

lemma *sorted-key-list-of-set-empty*:
  sorted-key-list-of-set f {} = []
  by (*fact fold-insort-key.empty*)

lemma *sorted-key-list-of-set-insert*:

assumes *insert x A* ⊆ *S* **and** *finite A x* ∉ *A*
shows *sorted-key-list-of-set f* (*insert x A*)
= *insort-key f x* (*sorted-key-list-of-set f A*)
using *assms* **by** (*fact fold-insort-key.insert*)

lemma *sorted-key-list-of-set-insert-remove* [*simp*]:
assumes *insert x A* ⊆ *S* **and** *finite A*
shows *sorted-key-list-of-set f* (*insert x A*)
= *insort-key f x* (*sorted-key-list-of-set f* (*A* − {*x*}))
using *assms* **by** (*fact fold-insort-key.insert-remove*)

lemma *sorted-key-list-of-set-eq-Nil-iff* [*simp*]:
assumes *A* ⊆ *S* **and** *finite A*
shows *sorted-key-list-of-set f A* = [] ⟷ *A* = {}
using *assms* **by** (*auto simp*: *fold-insort-key.remove*)

lemma *set-sorted-key-list-of-set* [*simp*]:
assumes *A* ⊆ *S* **and** *finite A*
shows *set* (*sorted-key-list-of-set f A*) = *A*
using *assms*(*2*,*1*)
by (*induct A rule*: *finite-induct*) (*simp-all add*: *set-insort-key*)

lemma *sorted-sorted-key-list-of-set* [*simp*]:
assumes *A* ⊆ *S*
shows *sorted* (*map f* (*sorted-key-list-of-set f A*))
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** ‹*A* ⊆ *S*›
by (*induction A*) (*simp-all add*: *sorted-insort-key*)
next
case *False* **thus** *?thesis* **by** *simp*
qed

lemma *distinct-if-distinct-map*: *distinct* (*map f xs*) ⟹ *distinct xs*
using *inj-on* **by** (*simp add*: *distinct-map*)

lemma *distinct-sorted-key-list-of-set* [*simp*]:
assumes *A* ⊆ *S*
shows *distinct* (*map f* (*sorted-key-list-of-set f A*))
proof (*cases finite A*)
case *True* **thus** *?thesis* **using** ‹*A* ⊆ *S*› *inj-on*
by (*induction A*) (*force simp*: *distinct-insort-key dest*: *inj-onD*)+
next
case *False* **thus** *?thesis* **by** *simp*
qed

lemma *length-sorted-key-list-of-set* [*simp*]:
assumes *A* ⊆ *S*
shows *length* (*sorted-key-list-of-set f A*) = *card A*
proof (*cases finite A*)

case *True*
with *assms inj-on* **show** *?thesis*
using *distinct-card*[*symmetric*, *OF distinct-sorted-key-list-of-set*]
by (*auto simp*: *inj-on-subset intro*!: *card-image*)
qed *auto*

lemmas *sorted-key-list-of-set* =
set-sorted-key-list-of-set sorted-sorted-key-list-of-set distinct-sorted-key-list-of-set

lemma *sorted-key-list-of-set-remove*:
assumes *insert x A* ⊆ *S* **and** *finite A*
shows *sorted-key-list-of-set f* (*A* − {*x*}) = *remove1 x* (*sorted-key-list-of-set f A*)
proof (*cases x* ∈ *A*)
case *False* **with** *assms* **have** *x* ∉ *set* (*sorted-key-list-of-set f A*) **by** *simp*
with *False* **show** *?thesis* **by** (*simp add*: *remove1-idem*)
next
case *True* **then obtain** *B* **where** *A*: *A* = *insert x B* **by** (*rule Set.set-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *strict-sorted-key-list-of-set* [*simp*]:
A ⊆ *S* ⟹ *sorted-wrt* (≺) (*map f* (*sorted-key-list-of-set f A*))
by (*cases finite A*) (*auto simp*: *strict-sorted-iff inj-on-subset*[*OF inj-on*])

lemma *finite-set-strict-sorted*:
assumes *A* ⊆ *S* **and** *finite A*
obtains *l* **where** *sorted-wrt* (≺) (*map f l*) *set l* = *A length l* = *card A*
using *assms*
by (*meson length-sorted-key-list-of-set set-sorted-key-list-of-set strict-sorted-key-list-of-set*)

lemma (**in** *linorder*) *strict-sorted-equal*:
assumes *sorted-wrt* (<) *xs*
and *sorted-wrt* (<) *ys*
and *set ys* = *set xs*
shows *ys* = *xs*
using *assms*
proof (*induction xs arbitrary*: *ys*)
case (*Cons x xs*)
show *?case*
proof (*cases ys*)
case *Nil*
then show *?thesis*
using *Cons.prems* **by** *auto*
next
case (*Cons y ys*′)
then have *xs* = *ys*′
by (*metis Cons.prems list.inject sorted-distinct-set-unique strict-sorted-iff*)
moreover have *x* = *y*
using *Cons.prems* ‹*xs* = *ys*′› *local.Cons* **by** *fastforce*

ultimately show *?thesis*
using *local.Cons* **by** *blast*
qed
qed *auto*

lemma (**in** *linorder*) *strict-sorted-equal-Uniq*: $\exists_{\leq 1} xs.\ sorted\text{-}wrt\ (<)\ xs \wedge set\ xs = A$
by (*simp add*: *Uniq-def strict-sorted-equal*)

lemma *sorted-key-list-of-set-inject*:
assumes $A \subseteq S$ $B \subseteq S$
assumes *sorted-key-list-of-set f A* = *sorted-key-list-of-set f B finite A finite B*
shows $A = B$
using *assms set-sorted-key-list-of-set* **by** *metis*

lemma *sorted-key-list-of-set-unique*:
assumes $A \subseteq S$ **and** *finite A*
shows *sorted-wrt* $(\prec)$ (*map f l*) $\wedge$ *set l* = *A* $\wedge$ *length l* = *card A*
$\longleftrightarrow$ *sorted-key-list-of-set f A* = *l*
using *assms*
by (*auto simp*: *strict-sorted-iff card-distinct idem-if-sorted-distinct*)

end

context *linorder*
begin

definition *sorted-list-of-set* $\equiv$ *sorted-key-list-of-set* $(\lambda x{::}'a.\ x)$

We abuse the *rewrites* functionality of locales to remove trivial assumptions that result from instantiating the key function to the identity.

sublocale *sorted-list-of-set*: *folding-insort-key* $(\leq)$ $(<)$ *UNIV* $(\lambda x.\ x)$
rewrites *sorted-key-list-of-set* $(\lambda x.\ x)$ = *sorted-list-of-set*
and $\bigwedge xs.\ map\ (\lambda x.\ x)\ xs \equiv xs$
and $\bigwedge X.\ (X \subseteq UNIV) \equiv True$
and $\bigwedge x.\ x \in UNIV \equiv True$
and $\bigwedge P.\ (True \Longrightarrow P) \equiv Trueprop\ P$
and $\bigwedge P\ Q.\ (True \Longrightarrow PROP\ P \Longrightarrow PROP\ Q) \equiv (PROP\ P \Longrightarrow True \Longrightarrow PROP\ Q)$
proof −
show *folding-insort-key* $(\leq)$ $(<)$ *UNIV* $(\lambda x.\ x)$
by *standard simp*
qed (*simp-all add*: *sorted-list-of-set-def*)

lemma *ex1-sorted-list-for-set-if-finite*:
finite X $\Longrightarrow$ $\exists!xs.\ sorted\text{-}wrt\ (<)\ xs \wedge set\ xs = X$
by (*metis sorted-list-of-set.finite-set-strict-sorted strict-sorted-equal*)

Alias theorems for backwards compatibility and ease of use.

lemmas *sorted-list-of-set = sorted-list-of-set.sorted-key-list-of-set* **and**
sorted-list-of-set-empty = sorted-list-of-set.sorted-key-list-of-set-empty **and**
sorted-list-of-set-insert = sorted-list-of-set.sorted-key-list-of-set-insert **and**
sorted-list-of-set-insert-remove = sorted-list-of-set.sorted-key-list-of-set-insert-remove
and
sorted-list-of-set-eq-Nil-iff = sorted-list-of-set.sorted-key-list-of-set-eq-Nil-iff
and
set-sorted-list-of-set = sorted-list-of-set.set-sorted-key-list-of-set **and**
sorted-sorted-list-of-set = sorted-list-of-set.sorted-sorted-key-list-of-set **and**
distinct-sorted-list-of-set = sorted-list-of-set.distinct-sorted-key-list-of-set **and**
length-sorted-list-of-set = sorted-list-of-set.length-sorted-key-list-of-set **and**
sorted-list-of-set-remove = sorted-list-of-set.sorted-key-list-of-set-remove **and**
strict-sorted-list-of-set = sorted-list-of-set.strict-sorted-key-list-of-set **and**
sorted-list-of-set-inject = sorted-list-of-set.sorted-key-list-of-set-inject **and**
sorted-list-of-set-unique = sorted-list-of-set.sorted-key-list-of-set-unique **and**
finite-set-strict-sorted = sorted-list-of-set.finite-set-strict-sorted

lemma *sorted-list-of-set-sort-remdups* [*code*]:
sorted-list-of-set (*set xs*) = *sort* (*remdups xs*)
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis*
by (*simp add*: *sorted-list-of-set.fold-insort-key.eq-fold sort-conv-fold fold-set-fold-remdups*)
qed

end

lemma *sorted-list-of-set-range* [*simp*]:
sorted-list-of-set {*m*..<*n*} = [*m*..<*n*]
by (*rule sorted-distinct-set-unique*) *simp-all*

lemma *sorted-list-of-set-lessThan-Suc* [*simp*]:
sorted-list-of-set {..<*Suc k*} = *sorted-list-of-set* {..<*k*} @ [*k*]
using *le0 lessThan-atLeast0 sorted-list-of-set-range upt-Suc-append* **by** *presburger*

lemma *sorted-list-of-set-atMost-Suc* [*simp*]:
sorted-list-of-set {..*Suc k*} = *sorted-list-of-set* {..*k*} @ [*Suc k*]
using *lessThan-Suc-atMost sorted-list-of-set-lessThan-Suc* **by** *fastforce*

lemma *sorted-lift-of-set-eq-upto* [*simp*]:
‹*sorted-list-of-set* {*k*..*l*} = [*k*..*l*]›
by (*rule sorted-distinct-set-unique*) *simp-all*

lemma *sorted-list-of-set-nonempty*:
assumes *finite A A* ≠ {}
shows *sorted-list-of-set A = Min A # sorted-list-of-set* (*A* − {*Min A*})
using *assms*
by (*auto simp*: *less-le simp flip*: *sorted-list-of-set.sorted-key-list-of-set-unique in-*

tro: *Min-in*)

lemma *sorted-list-of-set-greaterThanLessThan*:
assumes *Suc i < j*
shows *sorted-list-of-set {i<..<j} = Suc i # sorted-list-of-set {Suc i<..<j}*
proof −
have *{i<..<j} = insert (Suc i) {Suc i<..<j}*
using *assms* **by** *auto*
then show *?thesis*
by (*metis assms atLeastSucLessThan-greaterThanLessThan sorted-list-of-set-range upt-conv-Cons*)
qed

lemma *sorted-list-of-set-greaterThanAtMost*:
assumes *Suc i ≤ j*
shows *sorted-list-of-set {i<..j} = Suc i # sorted-list-of-set {Suc i<..j}*
using *sorted-list-of-set-greaterThanLessThan* [*of i Suc j*]
by (*metis assms greaterThanAtMost-def greaterThanLessThan-eq le-imp-less-Suc lessThan-Suc-atMost*)

lemma *nth-sorted-list-of-set-greaterThanLessThan*:
n < j − Suc i ⟹ sorted-list-of-set {i<..<j} ! n = Suc (i+n)
by (*induction n arbitrary*: *i*) (*auto simp*: *sorted-list-of-set-greaterThanLessThan*)

lemma *nth-sorted-list-of-set-greaterThanAtMost*:
n < j − i ⟹ sorted-list-of-set {i<..j} ! n = Suc (i+n)
using *nth-sorted-list-of-set-greaterThanLessThan* [*of n Suc j i*]
by (*simp add*: *greaterThanAtMost-def greaterThanLessThan-eq lessThan-Suc-atMost*)

lemma *sorted-wrt-induct* [*consumes 1*, *case-names Nil Cons*]:
assumes *sorted-wrt R xs*
assumes *P []*
⋀x xs. (⋀y. y ∈ set xs ⟹ R x y) ⟹ P xs ⟹ P (x # xs)
shows *P xs*
using *assms*(*1*) **by** (*induction xs*) (*auto intro*: *assms*)

lemma *sorted-wrt-trans-induct* [*consumes 2*, *case-names Nil single Cons*]:
assumes *sorted-wrt R xs transp R*
assumes *P []* *⋀x. P [x]*
⋀x y xs. R x y ⟹ P (y # xs) ⟹ P (x # y # xs)
shows *P xs*
using *assms*(*1*)
by (*induction xs rule*: *induct-list012*)
(*auto intro*: *assms simp*: *sorted-wrt2*[*OF assms*(*2*)])

lemmas *sorted-induct* [*consumes 1*, *case-names Nil single Cons*] =
sorted-wrt-trans-induct[*OF - preorder-class.transp-on-le*]

lemma *sorted-wrt-map-mono*:
assumes *sorted-wrt R xs*
assumes $\bigwedge x\ y.\ x \in set\ xs \Longrightarrow y \in set\ xs \Longrightarrow R\ x\ y \Longrightarrow R'\ (f\ x)\ (f\ y)$
shows *sorted-wrt R' (map f xs)*
using *assms* **by** (*induction rule*: *sorted-wrt-induct*) *auto*

lemma *sorted-map-mono*:
assumes *sorted xs* **and** *mono-on* (*set xs*) *f*
shows *sorted* (*map f xs*)
using *assms*(*1*)
by (*rule sorted-wrt-map-mono*) (*use assms* **in** ‹*auto simp*: *mono-on-def*›)

66.3.6 *lists*: the list-forming operator over sets

inductive-set
lists :: *'a set* => *'a list set*
for *A* :: *'a set*
where
Nil [*intro*!, *simp*]: [] ∈ *lists A*
| *Cons* [*intro*!, *simp*]: ⟦*a* ∈ *A*; *l* ∈ *lists A*⟧ ⟹ *a*#*l* ∈ *lists A*

inductive-cases *listsE* [*elim*!]: *x*#*l* ∈ *lists A*
inductive-cases *listspE* [*elim*!]: *listsp A* (*x* # *l*)

inductive-simps *listsp-simps*[*code*]:
listsp A []
listsp A (*x* # *xs*)

lemma *listsp-mono* [*mono*]: *A* ≤ *B* ⟹ *listsp A* ≤ *listsp B*
by (*rule predicate1I*, *erule listsp.induct*, *blast*+)

lemmas *lists-mono* = *listsp-mono* [*to-set*]

lemma *listsp-infI*:
assumes *l*: *listsp A l* **shows** *listsp B l* ⟹ *listsp* (*inf A B*) *l* **using** *l*
by *induct blast*+

lemmas *lists-IntI* = *listsp-infI* [*to-set*]

lemma *listsp-inf-eq* [*simp*]: *listsp* (*inf A B*) = *inf* (*listsp A*) (*listsp B*)
proof (*rule mono-inf* [**where** *f*=*listsp*, *THEN order-antisym*])
show *mono listsp* **by** (*simp add*: *mono-def listsp-mono*)
show *inf* (*listsp A*) (*listsp B*) ≤ *listsp* (*inf A B*) **by** (*blast intro*!: *listsp-infI*)
qed

lemmas *listsp-conj-eq* [*simp*] = *listsp-inf-eq* [*simplified inf-fun-def inf-bool-def*]

lemmas *lists-Int-eq* [*simp*] = *listsp-inf-eq* [*to-set*]

lemma *Cons-in-lists-iff*[*simp*]: $x\#xs \in lists\ A \longleftrightarrow x \in A \wedge xs \in lists\ A$
by *auto*

lemma *append-in-listsp-conv* [*iff*]: $(listsp\ A\ (xs\ @\ ys)) = (listsp\ A\ xs \wedge listsp\ A\ ys)$
by (*induct xs*) *auto*

lemmas *append-in-lists-conv* [*iff*] = *append-in-listsp-conv* [*to-set*]

lemma *in-listsp-conv-set*: $(listsp\ A\ xs) = (\forall x \in set\ xs.\ A\ x)$
— eliminate *listsp* in favour of *set*
by (*induct xs*) *auto*

lemmas *in-lists-conv-set* [*code-unfold*] = *in-listsp-conv-set* [*to-set*]

lemma *in-listspD* [*dest*!]: $listsp\ A\ xs \Longrightarrow \forall x{\in}set\ xs.\ A\ x$
by (*rule in-listsp-conv-set* [*THEN iffD1*])

lemmas *in-listsD* [*dest*!] = *in-listspD* [*to-set*]

lemma *in-listspI* [*intro*!]: $\forall x{\in}set\ xs.\ A\ x \Longrightarrow listsp\ A\ xs$
by (*rule in-listsp-conv-set* [*THEN iffD2*])

lemmas *in-listsI* [*intro*!] = *in-listspI* [*to-set*]

lemma *mono-lists*: *mono lists*
unfolding *mono-def* **by** *auto*

lemma *lists-eq-set*: $lists\ A = \{xs.\ set\ xs \leq A\}$
by *auto*

lemma *lists-empty* [*simp*]: $lists\ \{\} = \{[]\}$
by *auto*

lemma *lists-UNIV* [*simp*]: $lists\ UNIV = UNIV$
by *auto*

lemma *lists-image*: $lists\ (f`A) = map\ f\ `\ lists\ A$
proof −
  { fix *xs* **have** $\forall x{\in}set\ xs.\ x \in f\ `\ A \Longrightarrow xs \in map\ f\ `\ lists\ A$
    by (*induct xs*) (*auto simp del*: *list.map simp add*: *list.map*[*symmetric*] *intro*!: *imageI*) **}**
  then show *?thesis* **by** *auto*
qed

lemma *inj-on-map-lists*: **assumes** *inj-on f A*
  shows *inj-on* (*map f*) (*lists A*)
proof
  fix *xs ys*

assume *xs* ∈ *lists A* **and** *ys* ∈ *lists A* **and** *map f xs* = *map f ys*
have *x* = *y* **if** *x* ∈ *set xs* **and** *y* ∈ *set ys* **and** *f x* = *f y* **for** *x y*
using *in-listsD*[*OF* ‹*xs* ∈ *lists A*›, *rule-format*, *OF* ‹*x* ∈ *set xs*›]
in-listsD[*OF* ‹*ys* ∈ *lists A*›, *rule-format*, *OF* ‹*y* ∈ *set ys*›]
‹*inj-on f A*›[*unfolded inj-on-def*, *rule-format*, *OF* - - ‹*f x* = *f y*›] **by** *blast*
from *list.inj-map-strong*[*OF this* ‹*map f xs* = *map f ys*›]
show *xs* = *ys*.
qed

lemma *bij-lists*: *bij-betw f X Y* ⟹ *bij-betw* (*map f*) (*lists X*) (*lists Y*)
unfolding *bij-betw-def* **using** *inj-on-map-lists lists-image* **by** *metis*

lemma *replicate-in-lists*: *a* ∈ *A* ⟹ *replicate k a* ∈ *lists A*
by (*induction k*) *auto*

66.3.7 Inductive definition for membership

inductive *ListMem* :: *'a* ⇒ *'a list* ⇒ *bool*
where
elem: *ListMem x* (*x* # *xs*)
| *insert*: *ListMem x xs* ⟹ *ListMem x* (*y* # *xs*)

lemma *ListMem-iff*: (*ListMem x xs*) = (*x* ∈ *set xs*)
proof
show *ListMem x xs* ⟹ *x* ∈ *set xs*
by (*induct set*: *ListMem*) *auto*
show *x* ∈ *set xs* ⟹ *ListMem x xs*
by (*induct xs*) (*auto intro*: *ListMem.intros*)
qed

66.3.8 Lists as Cartesian products

set-Cons A Xs: the set of lists with head drawn from *A* and tail drawn from *Xs*.

definition *set-Cons* :: *'a set* ⇒ *'a list set* ⇒ *'a list set* **where**
set-Cons A XS = {*z*. ∃ *x xs*. *z* = *x* # *xs* ∧ *x* ∈ *A* ∧ *xs* ∈ *XS*}

lemma *set-Cons-sing-Nil* [*simp*]: *set-Cons A* {[]} = (%*x*. [*x*])‘*A*
by (*auto simp add*: *set-Cons-def*)

Yields the set of lists, all of the same length as the argument and with elements drawn from the corresponding element of the argument.

primrec *listset* :: *'a set list* ⇒ *'a list set* **where**
listset [] = {[]} |
listset (*A* # *As*) = *set-Cons A* (*listset As*)

66.3.9 Transitive Closure on Lists

Use $^+$ on binary relations if possible. Transitive closure on lists is useful for executable definitions on the list level. Is not efficient, naive closure computation.

definition *trans-list-step ps* = [(*a*,*c*). (*a*,*b*) ← *ps*, (*b'*,*c*) ← *ps*, *b*=*b'*]

lemma *set-trans-list-step-subset-trancl*: *set* (*trans-list-step ps*) ⊆ (*set ps*)^+
unfolding *trans-list-step-def* **by** *auto*

function *trancl-list* :: ('*a* ∗ '*a*) *list* ⇒ ('*a* ∗ '*a*) *list* **where**
trancl-list ps =
(*let ps'* = *trans-list-step ps*
in if set ps' ⊆ *set ps then ps else trancl-list* (*List.union ps' ps*))
by *pat-completeness auto*

termination
proof
let *?r* = λ*ps*::('*a* ∗ '*a*) *list*. *card* ((*set ps*)^+ − *set*(*ps*))

show *wf* (*measure ?r*) **by** *blast*

fix *ps ps'* :: ('*a* ∗ '*a*) *list*
assume *asms*: *ps'* = *trans-list-step ps* ¬ *set ps'* ⊆ *set ps*
let *?P* = *set ps* **let** *?P'* = *set*(*trans-list-step ps*)
have (*?P'* ∪ *?P*)^+ − (*?P'* ∪ *?P*) = *?P*^+ − (*?P'* ∪ *?P*)
using *trancl-absorb-subset-trancl*[*OF set-trans-list-step-subset-trancl*] **by** (*metis Un-commute*)
also have *?P*^+ − (*?P'* ∪ *?P*) < *?P*^+ − *?P*
using *asms*(*1*,*2*) *set-trans-list-step-subset-trancl* **by** *fastforce*
finally have *card*((*?P'* ∪ *?P*)^+ − (*?P'* ∪ *?P*)) < *card* (*?P*^+ − *?P*)
by (*meson List.finite-set finite-Diff finite-trancl psubset-card-mono*)
with *asms* **show** (*List.union ps' ps*, *ps*) ∈ *measure ?r* **by**(*simp*)
qed

declare *trancl-list.simps*[*code*, *simp del*]

lemma *set-trancl-list*: *set*(*trancl-list ps*) = (*set ps*)^+
proof (*induction ps rule*: *trancl-list.induct*)
case (*1 ps*)
let *?P* = *set ps* **let** *?P'* = *set*(*trans-list-step ps*)
show *?case*
proof (*cases ?P'* ⊆ *?P*)
case *True*
then have (*a*,*b*) ∈ *set ps* ⟹ (*b*,*c*) ∈ *set ps* ⟹ (*a*,*c*) ∈ *set ps* **for** *a b c*
unfolding *trans-list-step-def* **by** *fastforce*
then show *?thesis* **using** *True trancl-id*[*OF transI*, *of ?P*]
using [[*simp-depth-limit*=*3*]] **by**(*simp add*: *Let-def trancl-list.simps*[*of ps*])
next

```
    case False
    from 1[OF refl False] False
   show ?thesis using trancl-absorb-subset-trancl[OF set-trans-list-step-subset-trancl]
      by(auto simp add: Un-commute Let-def trancl-list.simps[of ps])
  qed
qed
```

66.4 Relations on Lists

66.4.1 Length Lexicographic Ordering

These orderings preserve well-foundedness: shorter lists precede longer lists. These ordering are not used in dictionaries.

```
primrec — The lexicographic ordering for lists of the specified length
  lexn :: ('a × 'a) set ⇒ nat ⇒ ('a list × 'a list) set where
lexn r 0 = {} |
lexn r (Suc n) =
  (map-prod (%(x, xs). x#xs) (%(x, xs). x#xs) ' (r <*lex*> lexn r n)) Int
  {(xs, ys). length xs = Suc n ∧ length ys = Suc n}

definition lex :: ('a × 'a) set ⇒ ('a list × 'a list) set where
lex r = (⋃ n. lexn r n) — Holds only between lists of the same length

definition lenlex :: ('a × 'a) set => ('a list × 'a list) set where
lenlex r = inv-image (less-than <*lex*> lex r) (λxs. (length xs, xs))
      — Compares lists by their length and then lexicographically

lemma wf-lexn: assumes wf r shows wf (lexn r n)
proof (induct n)
  case (Suc n)
  have inj: inj (λ(x, xs). x # xs)
    using assms by (auto simp: inj-on-def)
  have wf: wf (map-prod (λ(x, xs). x # xs) (λ(x, xs). x # xs) ' (r <*lex*> lexn
r n))
    by (simp add: Suc.hyps assms wf-lex-prod wf-map-prod-image [OF - inj])
  then show ?case
    by (rule wf-subset) auto
qed auto

lemma lexn-length:
  (xs, ys) ∈ lexn r n ⟹ length xs = n ∧ length ys = n
  by (induct n arbitrary: xs ys) auto

lemma wf-lex [intro!]:
  assumes wf r shows wf (lex r)
  unfolding lex-def
proof (rule wf-UN)
  show wf (lexn r i) for i
    by (simp add: assms wf-lexn)
```

```
  show ⋀i j. lexn r i ≠ lexn r j ⟹ Domain (lexn r i) ∩ Range (lexn r j) = {}
    by (metis DomainE Int-emptyI RangeE lexn-length)
qed

lemma lexn-conv:
  lexn r n =
    {(xs,ys). length xs = n ∧ length ys = n ∧
    (∃ xys x y xs' ys'. xs = xys @ x#xs' ∧ ys = xys @ y # ys' ∧ (x, y) ∈ r)}
  (is ?L n = ?R n is - = {(xs,ys). ?len n xs ∧ ?len n ys ∧ (∃ xys. ?P xs ys xys)})
proof (induction n)
  case (Suc n)

  have (xs,ys) ∈ ?L (Suc n) if r: (xs,ys) ∈ ?R (Suc n) for xs ys
  proof −
    from r obtain xys where r': ?len (Suc n) xs ?len (Suc n) ys ?P xs ys xys by
auto
    then show ?thesis
      using r' Suc
      by (cases xys; fastforce simp: image-Collect lex-prod-def)
  qed
  moreover have (xs,ys) ∈ ?L (Suc n) ⟹ (xs,ys) ∈ ?R (Suc n) for xs ys
   using Suc by (auto simp add: image-Collect lex-prod-def)(blast, meson Cons-eq-appendI)
  ultimately show ?case by (meson pred-equals-eq2)
qed auto
```

By Mathias Fleury:

```
proposition lexn-transI:
  assumes trans r shows trans (lexn r n)
unfolding trans-def
proof (intro allI impI)
  fix as bs cs
  assume asbs: (as, bs) ∈ lexn r n and bscs: (bs, cs) ∈ lexn r n
  obtain abs a b as' bs' where
    n: length as = n and length bs = n and
    as: as = abs @ a # as' and
    bs: bs = abs @ b # bs' and
    abr: (a, b) ∈ r
    using asbs unfolding lexn-conv by blast
  obtain bcs b' c' cs' bs' where
    n': length cs = n and length bs = n and
    bs': bs = bcs @ b' # bs' and
    cs: cs = bcs @ c' # cs' and
    b'c'r: (b', c') ∈ r
    using bscs unfolding lexn-conv by blast
  consider (le) length bcs < length abs
    | (eq) length bcs = length abs
    | (ge) length bcs > length abs by linarith
  thus (as, cs) ∈ lexn r n
  proof cases
```

```
    let ?k = length bcs
    case le
    hence as ! ?k = bs ! ?k unfolding as bs by (simp add: nth-append)
    hence (as ! ?k, cs ! ?k) ∈ r using b'c'r unfolding bs' cs by auto
    moreover
    have length bcs < length as using le unfolding as by simp
    from id-take-nth-drop[OF this]
    have as = take ?k as @ as ! ?k # drop (Suc ?k) as .
    moreover
    have length bcs < length cs unfolding cs by simp
    from id-take-nth-drop[OF this]
    have cs = take ?k cs @ cs ! ?k # drop (Suc ?k) cs .
    moreover have take ?k as = take ?k cs
      using le arg-cong[OF bs, of take (length bcs)]
      unfolding cs as bs' by auto
    ultimately show ?thesis using n n' unfolding lexn-conv by auto
  next
    let ?k = length abs
    case ge
    hence bs ! ?k = cs ! ?k unfolding bs' cs by (simp add: nth-append)
    hence (as ! ?k, cs ! ?k) ∈ r using abr unfolding as bs by auto
    moreover
    have length abs < length as using ge unfolding as by simp
    from id-take-nth-drop[OF this]
    have as = take ?k as @ as ! ?k # drop (Suc ?k) as .
    moreover have length abs < length cs using n n' unfolding as by simp
    from id-take-nth-drop[OF this]
    have cs = take ?k cs @ cs ! ?k # drop (Suc ?k) cs .
    moreover have take ?k as = take ?k cs
      using ge arg-cong[OF bs', of take (length abs)]
      unfolding cs as bs by auto
    ultimately show ?thesis using n n' unfolding lexn-conv by auto
  next
    let ?k = length abs
    case eq
    hence *: abs = bcs b = b' using bs bs' by auto
    hence (a, c') ∈ r
      using abr b'c'r assms unfolding trans-def by blast
    with * show ?thesis using n n' unfolding lexn-conv as bs cs by auto
  qed
qed

corollary lex-transI:
    assumes trans r shows trans (lex r)
  using lexn-transI [OF assms]
  by (clarsimp simp add: lex-def trans-def) (metis lexn-length)

lemma lex-conv:
  lex r =
```

```
    {(xs,ys). length xs = length ys ∧
    (∃ xys x y xs' ys'. xs = xys @ x # xs' ∧ ys = xys @ y # ys' ∧ (x, y) ∈ r)}
by (force simp add: lex-def lexn-conv)

lemma wf-lenlex [intro!]: wf r ⟹ wf (lenlex r)
by (unfold lenlex-def) blast

lemma lenlex-conv:
    lenlex r = {(xs,ys). length xs < length ys ∨
                length xs = length ys ∧ (xs, ys) ∈ lex r}
  by (auto simp add: lenlex-def Id-on-def lex-prod-def inv-image-def)

lemma total-lenlex:
  assumes total r
  shows total (lenlex r)
proof −
  have (xs,ys) ∈ lexn r (length xs) ∨ (ys,xs) ∈ lexn r (length xs)
    if xs ≠ ys and len: length xs = length ys for xs ys
  proof −
    obtain pre x xs' y ys' where x≠y and xs: xs = pre @ [x] @ xs' and ys: ys =
pre @ [y] @ys'
      by (meson len ‹xs ≠ ys› same-length-different)
    then consider (x,y) ∈ r | (y,x) ∈ r
      by (meson UNIV-I assms total-on-def)
    then show ?thesis
    by cases (use len in ‹(force simp add: lexn-conv xs ys)+›)
qed
  then show ?thesis
    by (fastforce simp: lenlex-def total-on-def lex-def)
qed

lemma lenlex-transI [intro]: trans r ⟹ trans (lenlex r)
  unfolding lenlex-def
  by (meson lex-transI trans-inv-image trans-less-than trans-lex-prod)

lemma Nil-notin-lex [iff]: ([], ys) ∉ lex r
  by (simp add: lex-conv)

lemma Nil2-notin-lex [iff]: (xs, []) ∉ lex r
  by (simp add:lex-conv)

lemma Cons-in-lex [simp]:
  (x # xs, y # ys) ∈ lex r ⟷ (x, y) ∈ r ∧ length xs = length ys ∨ x = y ∧ (xs,
ys) ∈ lex r
 (is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs
    by (simp add: lex-conv) (metis hd-append list.sel(1) list.sel(3) tl-append2)
next
```

assume *?rhs* **then show** *?lhs*
by (*simp add*: *lex-conv*) (*blast intro*: *Cons-eq-appendI*)
qed

lemma *Nil-lenlex-iff1* [*simp*]: ([], *ns*) ∈ *lenlex r* ⟷ *ns* ≠ []
and *Nil-lenlex-iff2* [*simp*]: (*ns*,[]) ∉ *lenlex r*
by (*auto simp*: *lenlex-def*)

lemma *Cons-lenlex-iff*:
((*m* # *ms*, *n* # *ns*) ∈ *lenlex r*) ⟷
length ms < *length ns*
∨ *length ms* = *length ns* ∧ (*m*,*n*) ∈ *r*
∨ (*m* = *n* ∧ (*ms*,*ns*) ∈ *lenlex r*)
by (*auto simp*: *lenlex-def*)

lemma *lenlex-irreflexive*: (⋀*x*. (*x*,*x*) ∉ *r*) ⟹ (*xs*,*xs*) ∉ *lenlex r*
by (*induction xs*) (*auto simp add*: *Cons-lenlex-iff*)

lemma *lenlex-trans*:
⟦(*x*,*y*) ∈ *lenlex r*; (*y*,*z*) ∈ *lenlex r*; *trans r*⟧ ⟹ (*x*,*z*) ∈ *lenlex r*
by (*meson lenlex-transI transD*)

lemma *lenlex-length*: (*ms*, *ns*) ∈ *lenlex r* ⟹ *length ms* ≤ *length ns*
by (*auto simp*: *lenlex-def*)

lemma *lex-append-rightI*:
(*xs*, *ys*) ∈ *lex r* ⟹ *length vs* = *length us* ⟹ (*xs* @ *us*, *ys* @ *vs*) ∈ *lex r*
by (*fastforce simp*: *lex-def lexn-conv*)

lemma *lex-append-leftI*:
(*ys*, *zs*) ∈ *lex r* ⟹ (*xs* @ *ys*, *xs* @ *zs*) ∈ *lex r*
by (*induct xs*) *auto*

lemma *lex-append-leftD*:
∀ *x*. (*x*,*x*) ∉ *r* ⟹ (*xs* @ *ys*, *xs* @ *zs*) ∈ *lex r* ⟹ (*ys*, *zs*) ∈ *lex r*
by (*induct xs*) *auto*

lemma *lex-append-left-iff*:
∀ *x*. (*x*,*x*) ∉ *r* ⟹ (*xs* @ *ys*, *xs* @ *zs*) ∈ *lex r* ⟷ (*ys*, *zs*) ∈ *lex r*
by(*metis lex-append-leftD lex-append-leftI*)

lemma *lex-take-index*:
assumes (*xs*, *ys*) ∈ *lex r*
obtains *i* **where** *i* < *length xs* **and** *i* < *length ys* **and** *take i xs* = *take i ys*
and (*xs* ! *i*, *ys* ! *i*) ∈ *r*
proof −
obtain *n us x xs′ y ys′* **where** (*xs*, *ys*) ∈ *lexn r n* **and** *length xs* = *n* **and** *length ys* = *n*
and *xs* = *us* @ *x* # *xs′* **and** *ys* = *us* @ *y* # *ys′* **and** (*x*, *y*) ∈ *r*

using *assms* **by** (*fastforce simp*: *lex-def lexn-conv*)
then show *?thesis* **by** (*intro that* [*of length us*]) *auto*
qed

lemma *irrefl-lex*: *irrefl r* $\Longrightarrow$ *irrefl* (*lex r*)
by (*meson irrefl-def lex-take-index*)

lemma *lexl-not-refl* [*simp*]: *irrefl r* $\Longrightarrow$ (*x*,*x*) $\notin$ *lex r*
by (*meson irrefl-def lex-take-index*)

66.4.2 Lexicographic Ordering

Classical lexicographic ordering on lists, ie. "a" < "ab" < "b". This ordering does *not* preserve well-foundedness. Author: N. Voelker, March 2005.

definition *lexord* :: ('*a* × '*a*) *set* ⇒ ('*a list* × '*a list*) *set* **where**
lexord r = {(*x*,*y*). ∃ *a v*. *y* = *x* @ *a* # *v* ∨
(∃ *u a b v w*. (*a*,*b*) ∈ *r* ∧ *x* = *u* @ (*a* # *v*) ∧ *y* = *u* @ (*b* # *w*))}

lemma *lexord-Nil-left*[*simp*]: ([],*y*) ∈ *lexord r* = (∃ *a x*. *y* = *a* # *x*)
by (*unfold lexord-def*, *induct-tac y*, *auto*)

lemma *lexord-Nil-right*[*simp*]: (*x*,[]) ∉ *lexord r*
by (*unfold lexord-def*, *induct-tac x*, *auto*)

lemma *lexord-cons-cons*[*simp*]:
(*a* # *x*, *b* # *y*) ∈ *lexord r* ⟷ (*a*,*b*)∈ *r* ∨ (*a* = *b* ∧ (*x*,*y*)∈ *lexord r*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
apply (*simp add*: *lexord-def*)
apply (*metis hd-append list.sel*(*1*) *list.sel*(*3*) *tl-append2*)
done
qed (*auto simp add*: *lexord-def*; (*blast* | *meson Cons-eq-appendI*))

lemmas *lexord-simps* = *lexord-Nil-left lexord-Nil-right lexord-cons-cons*

lemma *lexord-same-pref-iff*:
(*xs* @ *ys*, *xs* @ *zs*) ∈ *lexord r* ⟷ (∃ *x* ∈ *set xs*. (*x*,*x*) ∈ *r*) ∨ (*ys*, *zs*) ∈ *lexord r*
by(*induction xs*) *auto*

lemma *lexord-same-pref-if-irrefl*[*simp*]:
irrefl r $\Longrightarrow$ (*xs* @ *ys*, *xs* @ *zs*) ∈ *lexord r* ⟷ (*ys*, *zs*) ∈ *lexord r*
by (*simp add*: *irrefl-def lexord-same-pref-iff*)

lemma *lexord-append-rightI*: ∃ *b z*. *y* = *b* # *z* $\Longrightarrow$ (*x*, *x* @ *y*) ∈ *lexord r*
by (*metis append-Nil2 lexord-Nil-left lexord-same-pref-iff*)

lemma *lexord-append-left-rightI*:

$(a,b) \in r \Longrightarrow (u \mathbin{@} a \# x,\ u \mathbin{@} b \# y) \in lexord\ r$
by (*simp add*: *lexord-same-pref-iff*)

lemma *lexord-append-leftI*: $(u,v) \in lexord\ r \Longrightarrow (x \mathbin{@} u,\ x \mathbin{@} v) \in lexord\ r$
by (*simp add*: *lexord-same-pref-iff*)

lemma *lexord-append-leftD*:
$[\![(x \mathbin{@} u,\ x \mathbin{@} v) \in lexord\ r;\ (\forall a.\ (a,a) \notin r)\ ]\!] \Longrightarrow (u,v) \in lexord\ r$
by (*simp add*: *lexord-same-pref-iff*)

lemma *lexord-take-index-conv*:
$((x,y) \in lexord\ r) =$
$((length\ x < length\ y \wedge take\ (length\ x)\ y = x) \vee$
$(\exists i.\ i < min(length\ x)(length\ y) \wedge take\ i\ x = take\ i\ y \wedge (x!i,y!i) \in r))$
proof −
have $(\exists a\ v.\ y = x \mathbin{@} a \# v) = (length\ x < length\ y \wedge take\ (length\ x)\ y = x)$
by (*metis Cons-nth-drop-Suc append-eq-conv-conj drop-all list.simps*(3) *not-le*)
moreover
have $(\exists u\ a\ b.\ (a,\ b) \in r \wedge (\exists v.\ x = u \mathbin{@} a \# v) \wedge (\exists w.\ y = u \mathbin{@} b \# w)) =$
$(\exists i{<}length\ x.\ i < length\ y \wedge take\ i\ x = take\ i\ y \wedge (x\ !\ i,\ y\ !\ i) \in r)$
(**is** *?L=?R*)
proof
show *?L⟹?R*
by (*metis append-eq-conv-conj drop-all leI list.simps*(3) *nth-append-length*)
show *?R⟹?L*
by (*metis id-take-nth-drop*)
qed
ultimately show *?thesis*
by (*auto simp*: *lexord-def Let-def*)
qed

— lexord is extension of partial ordering List.lex
lemma *lexord-lex*: $(x,y) \in lex\ r = ((x,y) \in lexord\ r \wedge length\ x = length\ y)$
proof (*induction x arbitrary*: *y*)
case (*Cons a x y*) **then show** *?case*
by (*cases y*) (*force+*)
qed *auto*

lemma *lexord-sufI*:
assumes $(u,w) \in lexord\ r\ length\ w \leq length\ u$
shows $(u@v,w@z) \in lexord\ r$
proof−
from *leD*[*OF assms*(2)] *assms*(1)[*unfolded lexord-take-index-conv*[*of u w r*] *min-absorb2*[*OF assms*(2)]]
obtain *i* **where** $take\ i\ u = take\ i\ w$ **and** $(u!i,w!i) \in r$ **and** $i < length\ w$
by *blast*
hence $((u@v)!i,\ (w@z)!i) \in r$
unfolding *nth-append* **using** *less-le-trans*[*OF ‹i < length w› assms*(2)] ‹$(u!i,w!i) \in r$›

by *presburger*
moreover have $i < min\ (length\ (u@v))\ (length\ (w@z))$
using *assms(2)* ‹$i < length\ w$› **by** *simp*
moreover have $take\ i\ (u@v) = take\ i\ (w@z)$
using *assms(2)* ‹$i < length\ w$› ‹$take\ i\ u = take\ i\ w$› **by** *simp*
ultimately show *?thesis*
using *lexord-take-index-conv* **by** *blast*
qed

lemma *lexord-sufE*:
assumes $(xs@zs,ys@qs) \in lexord\ r\ xs \neq ys\ length\ xs = length\ ys\ length\ zs = length\ qs$
shows $(xs,ys) \in lexord\ r$
proof−
obtain i **where** $i < length\ (xs@zs)$ **and** $i < length\ (ys@qs)$ **and** $take\ i\ (xs@zs) = take\ i\ (ys@qs)$
and $((xs@zs)\ !\ i, (ys@qs)\ !\ i) \in r$
using *assms(1) lex-take-index*[*unfolded lexord-lex*,*of xs @ zs ys @ qs r*]
length-append[*of xs zs*, *unfolded assms(3,4)*, *folded length-append*[*of ys qs*]]
by *blast*
have $length\ (take\ i\ xs) = length\ (take\ i\ ys)$
by (*simp add*: *assms(3)*)
have $i < length\ xs$
using *assms(2,3) le-less-linear take-all*[*of xs i*] *take-all*[*of ys i*]
‹$take\ i\ (xs\ @\ zs) = take\ i\ (ys\ @\ qs)$› *append-eq-append-conv take-append*
by *metis*
hence $(xs\ !\ i, ys\ !\ i) \in r$
using ‹$((xs\ @\ zs)\ !\ i, (ys\ @\ qs)\ !\ i) \in r$› *assms(3)* **by** (*simp add*: *nth-append*)
moreover have $take\ i\ xs = take\ i\ ys$
using *assms(3)* ‹$take\ i\ (xs\ @\ zs) = take\ i\ (ys\ @\ qs)$› **by** *auto*
ultimately show *?thesis*
unfolding *lexord-take-index-conv* **using** ‹$i < length\ xs$› *assms(3)* **by** *fastforce*
qed

lemma *lexord-irreflexive*: $\forall x.\ (x,x) \notin r \Longrightarrow (xs,xs) \notin lexord\ r$
by (*induct xs*) *auto*

By René Thiemann:

lemma *lexord-partial-trans*:
$(\bigwedge x\ y\ z.\ x \in set\ xs \Longrightarrow (x,y) \in r \Longrightarrow (y,z) \in r \Longrightarrow (x,z) \in r)$
$\Longrightarrow\ (xs,ys) \in lexord\ r\ \Longrightarrow\ (ys,zs) \in lexord\ r \Longrightarrow\ (xs,zs) \in lexord\ r$
proof (*induct xs arbitrary*: *ys zs*)
case *Nil*
from *Nil(3)* **show** *?case* **unfolding** *lexord-def* **by** (*cases zs*, *auto*)
next
case (*Cons x xs yys zzs*)
from *Cons(3)* **obtain** *y ys* **where** *yys*: $yys = y\ \#\ ys$ **unfolding** *lexord-def*
by (*cases yys*, *auto*)
note *Cons* = *Cons*[*unfolded yys*]

```
  from Cons(3) have one: (x,y) ∈ r ∨ x = y ∧ (xs,ys) ∈ lexord r by auto
  from Cons(4) obtain z zs where zzs: zzs = z # zs unfolding lexord-def
    by (cases zzs, auto)
  note Cons = Cons[unfolded zzs]
  from Cons(4) have two: (y,z) ∈ r ∨ y = z ∧ (ys,zs) ∈ lexord r by auto
  {
    assume (xs,ys) ∈ lexord r and (ys,zs) ∈ lexord r
    from Cons(1)[OF - this] Cons(2)
    have (xs,zs) ∈ lexord r by auto
  } note ind1 = this
  {
    assume (x,y) ∈ r and (y,z) ∈ r
    from Cons(2)[OF - this] have (x,z) ∈ r by auto
  } note ind2 = this
  from one two ind1 ind2
  have (x,z) ∈ r ∨ x = z ∧ (xs,zs) ∈ lexord r by blast
  thus ?case unfolding zzs by auto
qed

lemma lexord-trans:
  ⟦ (x, y) ∈ lexord r; (y, z) ∈ lexord r; trans r ⟧ ⟹ (x, z) ∈ lexord r
  by(auto simp: trans-def intro:lexord-partial-trans)

lemma lexord-transI:  trans r ⟹ trans (lexord r)
  by (meson lexord-trans transI)

lemma total-lexord: total r ⟹ total (lexord r)
  unfolding total-on-def
proof clarsimp
  fix x y
  assume ∀ x y. x ≠ y ⟶ (x, y) ∈ r ∨ (y, x) ∈ r
    and (x::'a list) ≠ y
    and (y, x) ∉ lexord r
  then
  show (x, y) ∈ lexord r
  proof (induction x arbitrary: y)
    case Nil
    then show ?case
      by (metis lexord-Nil-left list.exhaust)
  next
    case (Cons a x y) then show ?case
      by (cases y) (force+)
  qed
qed

corollary lexord-linear: (∀ a b. (a,b) ∈ r ∨ a = b ∨ (b,a) ∈ r) ⟹ (x,y) ∈ lexord
r ∨ x = y ∨ (y,x) ∈ lexord r
  using total-lexord by (metis UNIV-I total-on-def)
```

lemma *lexord-irrefl*:
  irrefl R $\Longrightarrow$ *irrefl* (*lexord R*)
by (*simp add*: *irrefl-def lexord-irreflexive*)

lemma *lexord-asym*:
  assumes *asym R*
  shows *asym* (*lexord R*)
proof
  fix *xs ys*
  assume (*xs*, *ys*) ∈ *lexord R*
  then show (*ys*, *xs*) ∉ *lexord R*
  proof (*induct xs arbitrary*: *ys*)
    case *Nil*
    then show *?case* **by** *simp*
  next
    case (*Cons x xs*)
    then obtain *z zs* **where** *ys*: *ys* = *z* # *zs* **by** (*cases ys*) *auto*
    with *assms Cons* **show** *?case* **by** (*auto dest*: *asymD*)
  qed
qed

lemma *lexord-asymmetric*:
  assumes *asym R*
  assumes *hyp*: (*a*, *b*) ∈ *lexord R*
  shows (*b*, *a*) ∉ *lexord R*
proof −
  from ‹*asym R*› **have** *asym* (*lexord R*) **by** (*rule lexord-asym*)
  then show *?thesis* **by** (*auto simp*: *hyp dest*: *asymD*)
qed

lemma *asym-lex*: *asym R* $\Longrightarrow$ *asym* (*lex R*)
  by (*meson asymI asymD irrefl-lex lexord-asym lexord-lex*)

lemma *asym-lenlex*: *asym R* $\Longrightarrow$ *asym* (*lenlex R*)
  by (*simp add*: *lenlex-def asym-inv-image asym-less-than asym-lex*)

lemma *lenlex-append1*:
  assumes *len*: (*us*,*xs*) ∈ *lenlex R* **and** *eq*: *length vs* = *length ys*
  shows (*us* @ *vs*, *xs* @ *ys*) ∈ *lenlex R*
  using *len*
proof (*induction us*)
  case *Nil*
  then show *?case*
    by (*simp add*: *lenlex-def eq*)
next
  case (*Cons u us*)
  with *lex-append-rightI* **show** *?case*
    by (*fastforce simp add*: *lenlex-def eq*)
qed

lemma *lenlex-append2* [*simp*]:
  assumes *irrefl R*
  shows (*us* @ *xs*, *us* @ *ys*) ∈ *lenlex R* ⟷ (*xs*, *ys*) ∈ *lenlex R*
proof (*induction us*)
  case *Nil*
  then show *?case*
    by (*simp add*: *lenlex-def*)
next
  case (*Cons u us*)
  with *assms* **show** *?case*
    by (*auto simp*: *lenlex-def irrefl-def*)
qed

Predicate version of lexicographic order integrated with Isabelle's order type classes. Author: Andreas Lochbihler

context *ord*
begin

context
  notes [[*inductive-internals*]]
begin

inductive *lexordp* :: *'a list* ⇒ *'a list* ⇒ *bool*
where
  Nil: *lexordp* [] (*y* # *ys*)
| *Cons*: *x* < *y* ⟹ *lexordp* (*x* # *xs*) (*y* # *ys*)
| *Cons-eq*:
  ⟦ ¬ *x* < *y*; ¬ *y* < *x*; *lexordp xs ys* ⟧ ⟹ *lexordp* (*x* # *xs*) (*y* # *ys*)

end

lemma *lexordp-simps* [*simp*, *code*]:
  lexordp [] *ys* ⟷ *ys* ≠ []
  lexordp xs [] ⟷ *False*
  lexordp (*x* # *xs*) (*y* # *ys*) ⟷ *x* < *y* ∨ ¬ *y* < *x* ∧ *lexordp xs ys*
by(*subst lexordp.simps*, *fastforce simp add*: *neq-Nil-conv*)+

inductive *lexordp-eq* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
  Nil: *lexordp-eq* [] *ys*
| *Cons*: *x* < *y* ⟹ *lexordp-eq* (*x* # *xs*) (*y* # *ys*)
| *Cons-eq*: ⟦ ¬ *x* < *y*; ¬ *y* < *x*; *lexordp-eq xs ys* ⟧ ⟹ *lexordp-eq* (*x* # *xs*) (*y* # *ys*)

lemma *lexordp-eq-simps* [*simp*, *code*]:
  lexordp-eq [] *ys* ⟷ *True*
  lexordp-eq xs [] ⟷ *xs* = []
  lexordp-eq (*x* # *xs*) (*y* # *ys*) ⟷ *x* < *y* ∨ ¬ *y* < *x* ∧ *lexordp-eq xs ys*
by(*subst lexordp-eq.simps*, *fastforce*)+

lemma *lexordp-append-rightI*: $ys \neq Nil \Longrightarrow lexordp\ xs\ (xs\ @\ ys)$
by(*induct xs*)(*auto simp add*: *neq-Nil-conv*)

lemma *lexordp-append-left-rightI*: $x < y \Longrightarrow lexordp\ (us\ @\ x\ \#\ xs)\ (us\ @\ y\ \#\ ys)$
by(*induct us*) *auto*

lemma *lexordp-eq-refl*: *lexordp-eq xs xs*
by(*induct xs*) *simp-all*

lemma *lexordp-append-leftI*: *lexordp us vs* $\Longrightarrow$ *lexordp* (*xs* @ *us*) (*xs* @ *vs*)
by(*induct xs*) *auto*

lemma *lexordp-append-leftD*: $[\![\ lexordp\ (xs\ @\ us)\ (xs\ @\ vs);\ \forall a.\ \neg\ a < a\ ]\!] \Longrightarrow$ *lexordp us vs*
by(*induct xs*) *auto*

lemma *lexordp-irreflexive*:
assumes *irrefl*: $\forall x.\ \neg\ x < x$
shows $\neg$ *lexordp xs xs*
proof
assume *lexordp xs xs*
thus *False* **by**(*induct xs ys*≡*xs*)(*simp-all add*: *irrefl*)
qed

lemma *lexordp-into-lexordp-eq*:
lexordp xs ys $\Longrightarrow$ *lexordp-eq xs ys*
by (*induction rule*: *lexordp.induct*) *simp-all*

lemma *lexordp-eq-pref*: *lexordp-eq u* (*u* @ *v*)
by (*metis append-Nil2 lexordp-append-rightI lexordp-eq-refl lexordp-into-lexordp-eq*)

end

declare *ord.lexordp-simps* [*simp*, *code*]
declare *ord.lexordp-eq-simps* [*simp*, *code*]

context *order*
begin

lemma *lexordp-antisym*:
assumes *lexordp xs ys lexordp ys xs*
shows *False*
using *assms* **by** *induct auto*

lemma *lexordp-irreflexive'*: $\neg$ *lexordp xs xs*
by(*rule lexordp-irreflexive*) *simp*

end

context *linorder* **begin**

lemma *lexordp-cases* [*consumes 1*, *case-names Nil Cons Cons-eq*, *cases pred*: *lexordp*]:
  assumes *lexordp xs ys*
  obtains (*Nil*) *y ys'* **where** *xs* = [] *ys* = *y* # *ys'*
  | (*Cons*) *x xs' y ys'* **where** *xs* = *x* # *xs' ys* = *y* # *ys' x* < *y*
  | (*Cons-eq*) *x xs' ys'* **where** *xs* = *x* # *xs' ys* = *x* # *ys' lexordp xs' ys'*
using *assms* **by** *cases* (*fastforce simp add*: *not-less-iff-gr-or-eq*)+

lemma *lexordp-induct* [*consumes 1*, *case-names Nil Cons Cons-eq*, *induct pred*: *lexordp*]:
  assumes *major*: *lexordp xs ys*
  and *Nil*: ⋀*y ys*. *P* [] (*y* # *ys*)
  and *Cons*: ⋀*x xs y ys*. *x* < *y* ⟹ *P* (*x* # *xs*) (*y* # *ys*)
  and *Cons-eq*: ⋀*x xs ys*. ⟦ *lexordp xs ys*; *P xs ys* ⟧ ⟹ *P* (*x* # *xs*) (*x* # *ys*)
  shows *P xs ys*
using *major* **by** *induct* (*simp-all add*: *Nil Cons not-less-iff-gr-or-eq Cons-eq*)

lemma *lexordp-iff*:
  lexordp xs ys ⟷ (∃*x vs*. *ys* = *xs* @ *x* # *vs*) ∨ (∃*us a b vs ws*. *a* < *b* ∧ *xs* = *us* @ *a* # *vs* ∧ *ys* = *us* @ *b* # *ws*)
  (**is** *?lhs* = *?rhs*)
proof
  assume *?lhs* **thus** *?rhs*
  proof *induct*
    case *Cons-eq* **thus** *?case* **by** *simp* (*metis append.simps*(*2*))
  qed(*fastforce intro*: *disjI2 del*: *disjCI intro*: *exI*[**where** *x*=[]])+
next
  assume *?rhs* **thus** *?lhs*
   by(*auto intro*: *lexordp-append-leftI*[**where** *us*=[], *simplified*] *lexordp-append-leftI*)
qed

lemma *lexordp-conv-lexord*:
  lexordp xs ys ⟷ (*xs*, *ys*) ∈ *lexord* {(*x*, *y*). *x* < *y*}
by(*simp add*: *lexordp-iff lexord-def*)

lemma *lexordp-eq-antisym*:
  assumes *lexordp-eq xs ys lexordp-eq ys xs*
  shows *xs* = *ys*
using *assms* **by** *induct simp-all*

lemma *lexordp-eq-trans*:
  assumes *lexordp-eq xs ys* **and** *lexordp-eq ys zs*
  shows *lexordp-eq xs zs*
  using *assms*
  by (*induct arbitrary*: *zs*) (*case-tac zs*; *auto*)+

lemma *lexordp-trans*:

assumes *lexordp xs ys lexordp ys zs*
shows *lexordp xs zs*
using *assms*
by (*induct arbitrary*: *zs*) (*case-tac zs*; *auto*)+

lemma *lexordp-linear*: *lexordp xs ys* ∨ *xs* = *ys* ∨ *lexordp ys xs*
by(*induct xs arbitrary*: *ys*; *case-tac ys*; *fastforce*)

lemma *lexordp-conv-lexordp-eq*: *lexordp xs ys* ⟷ *lexordp-eq xs ys* ∧ ¬ *lexordp-eq ys xs*
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs*
hence ¬ *lexordp-eq ys xs* **by** *induct simp-all*
with ‹*?lhs*› **show** *?rhs* **by** (*simp add*: *lexordp-into-lexordp-eq*)
next
assume *?rhs*
hence *lexordp-eq xs ys* ¬ *lexordp-eq ys xs* **by** *simp-all*
thus *?lhs* **by** *induct simp-all*
qed

lemma *lexordp-eq-conv-lexord*: *lexordp-eq xs ys* ⟷ *xs* = *ys* ∨ *lexordp xs ys*
by(*auto simp add*: *lexordp-conv-lexordp-eq lexordp-eq-refl dest*: *lexordp-eq-antisym*)

lemma *lexordp-eq-linear*: *lexordp-eq xs ys* ∨ *lexordp-eq ys xs*
by (*induct xs arbitrary*: *ys*) (*case-tac ys*; *auto*)+

lemma *lexordp-linorder*: *class.linorder lexordp-eq lexordp*
by *unfold-locales*
(*auto simp add*: *lexordp-conv-lexordp-eq lexordp-eq-refl lexordp-eq-antisym intro*: *lexordp-eq-trans del*: *disjCI intro*: *lexordp-eq-linear*)

end

66.4.3 Lexicographic combination of measure functions

These are useful for termination proofs

definition *measures fs* = *inv-image* (*lex less-than*) (%*a*. *map* (%*f*. *f a*) *fs*)

lemma *wf-measures*[*simp*]: *wf* (*measures fs*)
unfolding *measures-def*
by *blast*

lemma *in-measures*[*simp*]:
(*x*, *y*) ∈ *measures* [] = *False*
(*x*, *y*) ∈ *measures* (*f* # *fs*)
= (*f x* < *f y* ∨ (*f x* = *f y* ∧ (*x*, *y*) ∈ *measures fs*))
unfolding *measures-def*
by *auto*

lemma *measures-less*: $f\ x < f\ y \Longrightarrow (x,\ y) \in measures\ (f\#fs)$
by *simp*

lemma *measures-lesseq*: $f\ x \leq f\ y \Longrightarrow (x,\ y) \in measures\ fs \Longrightarrow (x,\ y) \in measures\ (f\#fs)$
by *auto*

66.4.4 Lifting Relations to Lists: one element

definition *listrel1* :: $('a \times 'a)\ set \Rightarrow ('a\ list \times 'a\ list)\ set$ **where**
$listrel1\ r = \{(xs,ys).$
$\exists us\ z\ z'\ vs.\ xs = us\ @\ z\ \#\ vs \wedge (z,z') \in r \wedge ys = us\ @\ z'\ \#\ vs\}$

lemma *listrel1I*:
$[\![(x,\ y) \in r;\ \ xs = us\ @\ x\ \#\ vs;\ \ ys = us\ @\ y\ \#\ vs\ ]\!] \Longrightarrow$
$(xs,\ ys) \in listrel1\ r$
unfolding *listrel1-def* **by** *auto*

lemma *listrel1E*:
$[\![(xs,\ ys) \in listrel1\ r;$
$!!x\ y\ us\ vs.\ [\![(x,\ y) \in r;\ \ xs = us\ @\ x\ \#\ vs;\ \ ys = us\ @\ y\ \#\ vs\ ]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
unfolding *listrel1-def* **by** *auto*

lemma *not-Nil-listrel1* [*iff*]: $([],\ xs) \notin listrel1\ r$
unfolding *listrel1-def* **by** *blast*

lemma *not-listrel1-Nil* [*iff*]: $(xs,\ []) \notin listrel1\ r$
unfolding *listrel1-def* **by** *blast*

lemma *Cons-listrel1-Cons* [*iff*]:
$(x\ \#\ xs,\ y\ \#\ ys) \in listrel1\ r \longleftrightarrow$
$(x,y) \in r \wedge xs = ys \vee x = y \wedge (xs,\ ys) \in listrel1\ r$
by (*simp add*: *listrel1-def Cons-eq-append-conv*) (*blast*)

lemma *listrel1I1*: $(x,y) \in r \Longrightarrow (x\ \#\ xs,\ y\ \#\ xs) \in listrel1\ r$
by *fast*

lemma *listrel1I2*: $(xs,\ ys) \in listrel1\ r \Longrightarrow (x\ \#\ xs,\ x\ \#\ ys) \in listrel1\ r$
by *fast*

lemma *append-listrel1I*:
$(xs,\ ys) \in listrel1\ r \wedge us = vs \vee xs = ys \wedge (us,\ vs) \in listrel1\ r$
$\Longrightarrow (xs\ @\ us,\ ys\ @\ vs) \in listrel1\ r$
unfolding *listrel1-def*
by *auto* (*blast intro*: *append-eq-appendI*)+

lemma *Cons-listrel1E1* [*elim*!]:

assumes $(x \# xs,\ ys) \in$ *listrel1 r*
and $\bigwedge y.\ ys = y \# xs \Longrightarrow (x,\ y) \in r \Longrightarrow R$
and $\bigwedge zs.\ ys = x \# zs \Longrightarrow (xs,\ zs) \in$ *listrel1 r* $\Longrightarrow R$
shows *R*
using *assms* **by** (*cases ys*) *blast+*

lemma *Cons-listrel1E2*[*elim*!]:
assumes $(xs,\ y \# ys) \in$ *listrel1 r*
and $\bigwedge x.\ xs = x \# ys \Longrightarrow (x,\ y) \in r \Longrightarrow R$
and $\bigwedge zs.\ xs = y \# zs \Longrightarrow (zs,\ ys) \in$ *listrel1 r* $\Longrightarrow R$
shows *R*
using *assms* **by** (*cases xs*) *blast+*

lemma *snoc-listrel1-snoc-iff*:
$(xs\ @\ [x],\ ys\ @\ [y]) \in$ *listrel1 r*
$\longleftrightarrow (xs,\ ys) \in$ *listrel1 r* $\wedge\ x = y \vee xs = ys \wedge (x,y) \in r$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L* **thus** *?R*
by (*fastforce simp*: *listrel1-def snoc-eq-iff-butlast butlast-append*)
next
assume *?R* **then show** *?L* **unfolding** *listrel1-def* **by** *force*
qed

lemma *listrel1-eq-len*: $(xs,ys) \in$ *listrel1 r* $\Longrightarrow$ *length xs* = *length ys*
unfolding *listrel1-def* **by** *auto*

lemma *listrel1-mono*:
$r \subseteq s \Longrightarrow$ *listrel1 r* $\subseteq$ *listrel1 s*
unfolding *listrel1-def* **by** *blast*

lemma *listrel1-converse*: *listrel1* $(r^{-1}) = ($*listrel1 r*$)^{-1}$
unfolding *listrel1-def* **by** *blast*

lemma *in-listrel1-converse*:
$(x,y) \in$ *listrel1* $(r^{-1}) \longleftrightarrow (x,y) \in ($*listrel1 r*$)^{-1}$
unfolding *listrel1-def* **by** *blast*

lemma *listrel1-iff-update*:
$(xs,ys) \in ($*listrel1 r*$)$
$\longleftrightarrow (\exists y\ n.\ (xs\ !\ n,\ y) \in r \wedge n <$ *length xs* $\wedge\ ys = xs[n:=y])$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
assume *?L*
then obtain *x y u v* **where** $xs = u\ @\ x \# v\ \ ys = u\ @\ y \# v\ \ (x,y) \in r$
unfolding *listrel1-def* **by** *auto*
then have $ys = xs[$*length u* $:=\ y]$ **and** *length u* $<$ *length xs*
and $(xs\ !$ *length u*$,\ y) \in r$ **by** *auto*
then show *?R* **by** *auto*
next

assume *?R*
then obtain *x y n* **where** $(xs!n,\ y) \in r\ n < size\ xs\ ys = xs[n:=y]\ x = xs!n$
by *auto*
then obtain *u v* **where** $xs = u\ @\ x\ \#\ v$ **and** $ys = u\ @\ y\ \#\ v$ **and** $(x,\ y) \in r$
by (*auto intro*: *upd-conv-take-nth-drop id-take-nth-drop*)
then show *?L* **by** (*auto simp*: *listrel1-def*)
qed

Accessible part and wellfoundedness:

lemma *Cons-acc-listrel1I* [*intro*!]:
$x \in Wellfounded.acc\ r \Longrightarrow xs \in Wellfounded.acc\ (listrel1\ r) \Longrightarrow (x\ \#\ xs) \in Wellfounded.acc\ (listrel1\ r)$
proof (*induction arbitrary*: *xs set*: *Wellfounded.acc*)
case *outer*: (*1 u*)
show *?case*
proof (*induct xs rule*: *acc-induct*)
case *1*
show $xs \in Wellfounded.acc\ (listrel1\ r)$
by (*simp add*: *outer.prems*)
qed (*metis* (*no-types*, *lifting*) *Cons-listrel1E2 acc.simps outer.IH*)
qed

lemma *lists-accD*: $xs \in lists\ (Wellfounded.acc\ r) \Longrightarrow xs \in Wellfounded.acc\ (listrel1\ r)$
proof (*induct set*: *lists*)
case *Nil*
then show *?case*
by (*meson acc.intros not-listrel1-Nil*)
next
case (*Cons a l*)
then show *?case*
by *blast*
qed

lemma *lists-accI*: $xs \in Wellfounded.acc\ (listrel1\ r) \Longrightarrow xs \in lists\ (Wellfounded.acc\ r)$
proof (*induction set*: *Wellfounded.acc*)
case (*1 x*)
then have $\bigwedge u\ v.\ [\![u \in set\ x;\ (v,\ u) \in r]\!] \Longrightarrow v \in Wellfounded.acc\ r$
by (*metis in-lists-conv-set in-set-conv-decomp listrel1I*)
then show *?case*
by (*meson acc.intros in-listsI*)
qed

lemma *wf-listrel1-iff*[*simp*]: $wf(listrel1\ r) = wf\ r$
by (*auto simp*: *wf-iff-acc*
intro: *lists-accD lists-accI*[*THEN Cons-in-lists-iff*[*THEN iffD1*, *THEN conjunct1*]])

66.4.5 Lifting Relations to Lists: all elements

inductive-set
  listrel :: $('a \times 'b)$ *set* $\Rightarrow$ $('a$ *list* $\times$ $'b$ *list*) *set*
  for r :: $('a \times 'b)$ *set*
where
    Nil: ([],[]) $\in$ *listrel r*
  | *Cons*: $[\![(x,y) \in r;\ (xs,ys) \in$ *listrel* $r]\!] \Longrightarrow (x\#xs,\ y\#ys) \in$ *listrel r*

inductive-cases *listrel-Nil1* [*elim*!]: ([],*xs*) $\in$ *listrel r*
inductive-cases *listrel-Nil2* [*elim*!]: (*xs*,[]) $\in$ *listrel r*
inductive-cases *listrel-Cons1* [*elim*!]: (*y*#*ys*,*xs*) $\in$ *listrel r*
inductive-cases *listrel-Cons2* [*elim*!]: (*xs*,*y*#*ys*) $\in$ *listrel r*

lemma *listrel-eq-len*: $(xs,\ ys) \in$ *listrel* $r \Longrightarrow$ *length xs* = *length ys*
  by(*induct rule*: *listrel.induct*) *auto*

lemma *listrel-iff-zip* [*code-unfold*]: (*xs*,*ys*) $\in$ *listrel r* $\longleftrightarrow$
  length xs = *length ys* $\wedge$ $(\forall (x,y) \in$ *set*(*zip xs ys*). $(x,y) \in r)$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof
  assume *?L* **thus** *?R* **by** *induct* (*auto intro*: *listrel-eq-len*)
next
  assume *?R* **thus** *?L*
    apply (*clarify*)
    by (*induct rule*: *list-induct2*) (*auto intro*: *listrel.intros*)
qed

lemma *listrel-iff-nth*: (*xs*,*ys*) $\in$ *listrel r* $\longleftrightarrow$
  length xs = *length ys* $\wedge$ $(\forall\, n <$ *length xs*. $(xs!n,\ ys!n) \in r)$ (**is** *?L* $\longleftrightarrow$ *?R*)
  by (*auto simp add*: *all-set-conv-all-nth listrel-iff-zip*)

lemma *listrel-mono*: $r \subseteq s \Longrightarrow$ *listrel* $r \subseteq$ *listrel s*
  by (*meson listrel-iff-nth subrelI subset-eq*)

lemma *listrel-subset*:
  assumes $r \subseteq A \times A$ **shows** *listrel* $r \subseteq$ *lists* $A \times$ *lists A*
proof *clarify*
  show $a \in$ *lists* $A \wedge b \in$ *lists A* **if** $(a,\ b) \in$ *listrel r* **for** *a b*
    using *that assms* **by** (*induction rule*: *listrel.induct*, *auto*)
qed

lemma *listrel-refl-on*:
  assumes *refl-on A r* **shows** *refl-on* (*lists A*) (*listrel r*)
proof −
  have $l \in$ *lists* $A \Longrightarrow (l,\ l) \in$ *listrel r* **for** *l*
    using *assms* **unfolding** *refl-on-def*
    by (*induction l*, *auto intro*: *listrel.intros*)
  then show *?thesis*
    by (*meson assms listrel-subset refl-on-def*)

qed

lemma *listrel-sym*: *sym r* $\Longrightarrow$ *sym* (*listrel r*)
by (*simp add*: *listrel-iff-nth sym-def*)

lemma *listrel-trans*:
assumes *trans r* **shows** *trans* (*listrel r*)
proof −
have (*x*, *z*) ∈ *listrel r* **if** (*x*, *y*) ∈ *listrel r* (*y*, *z*) ∈ *listrel r* **for** *x y z*
using *that*
proof *induction*
case (*Cons x y xs ys*)
then show *?case*
by *clarsimp* (*metis assms listrel.Cons listrel-iff-nth transD*)
qed *auto*
then show *?thesis*
using *transI* **by** *blast*
qed

theorem *equiv-listrel*: *equiv A r* $\Longrightarrow$ *equiv* (*lists A*) (*listrel r*)
by (*simp add*: *equiv-def listrel-subset listrel-refl-on listrel-sym listrel-trans*)

lemma *listrel-rtrancl-refl*[*iff*]: (*xs*,*xs*) ∈ *listrel*(*r**)
using *listrel-refl-on*[*of UNIV*, *OF refl-rtrancl*]
by(*auto simp*: *refl-on-def*)

lemma *listrel-rtrancl-trans*:
⟦(*xs*,*ys*) ∈ *listrel*(*r**); (*ys*,*zs*) ∈ *listrel*(*r**)⟧ $\Longrightarrow$ (*xs*,*zs*) ∈ *listrel*(*r**)
by (*metis listrel-trans trans-def trans-rtrancl*)

lemma *listrel-Nil* [*simp*]: *listrel r* `` {[]} = {[]}
by (*blast intro*: *listrel.intros*)

lemma *listrel-Cons*:
listrel r `` {*x*#*xs*} = *set-Cons* (*r*``{*x*}) (*listrel r* `` {*xs*})
by (*auto simp add*: *set-Cons-def intro*: *listrel.intros*)

Relating *listrel1*, *listrel* and closures:

lemma *listrel1-rtrancl-subset-rtrancl-listrel1*: *listrel1* (*r**) ⊆ (*listrel1 r*)*
proof (*rule subrelI*)
fix *xs ys* **assume** *1*: (*xs*,*ys*) ∈ *listrel1* (*r**)
{ fix *x y us vs*
have (*x*,*y*) ∈ *r** $\Longrightarrow$ (*us* @ *x* # *vs*, *us* @ *y* # *vs*) ∈ (*listrel1 r*)*
proof(*induct rule*: *rtrancl.induct*)
case *rtrancl-refl* **show** *?case* **by** *simp*
next
case *rtrancl-into-rtrancl* **thus** *?case*
by (*metis listrel1I rtrancl.rtrancl-into-rtrancl*)
qed **}**

thus $(xs,ys) \in (listrel1\ r)^*$ **using** *1* **by**(*blast elim*: *listrel1E*)
qed

lemma *rtrancl-listrel1-eq-len*: $(x,y) \in (listrel1\ r)^* \Longrightarrow length\ x = length\ y$
by (*induct rule*: *rtrancl.induct*) (*auto intro*: *listrel1-eq-len*)

lemma *rtrancl-listrel1-ConsI1*:
$(xs,ys) \in (listrel1\ r)^* \Longrightarrow (x\#xs,x\#ys) \in (listrel1\ r)^*$
proof (*induction rule*: *rtrancl.induct*)
case (*rtrancl-into-rtrancl a b c*)
then show *?case*
by (*metis listrel1I2 rtrancl.rtrancl-into-rtrancl*)
qed *auto*

lemma *rtrancl-listrel1-ConsI2*:
$(x,y) \in r^* \Longrightarrow (xs,\ ys) \in (listrel1\ r)^* \Longrightarrow (x\ \#\ xs,\ y\ \#\ ys) \in (listrel1\ r)^*$
by (*meson in-mono listrel1I1 listrel1-rtrancl-subset-rtrancl-listrel1 rtrancl-listrel1-ConsI1 rtrancl-trans*)

lemma *listrel1-subset-listrel*:
$r \subseteq r' \Longrightarrow refl\ r' \Longrightarrow listrel1\ r \subseteq listrel(r')$
by(*auto elim*!: *listrel1E simp add*: *listrel-iff-zip set-zip refl-on-def*)

lemma *listrel-reflcl-if-listrel1*:
$(xs,ys) \in listrel1\ r \Longrightarrow (xs,ys) \in listrel(r^*)$
by(*erule listrel1E*)(*auto simp add*: *listrel-iff-zip set-zip*)

lemma *listrel-rtrancl-eq-rtrancl-listrel1*: $listrel\ (r^*) = (listrel1\ r)^*$
proof
{ fix *x y* **assume** $(x,y) \in listrel\ (r^*)$
then have $(x,y) \in (listrel1\ r)^*$
by *induct* (*auto intro*: *rtrancl-listrel1-ConsI2*) **}**
then show $listrel\ (r^*) \subseteq (listrel1\ r)^*$
by (*rule subrelI*)
next
show $listrel\ (r^*) \supseteq (listrel1\ r)^*$
proof(*rule subrelI*)
fix *xs ys* **assume** $(xs,ys) \in (listrel1\ r)^*$
then show $(xs,ys) \in listrel\ (r^*)$
proof *induct*
case *base* **show** *?case* **by**(*auto simp add*: *listrel-iff-zip set-zip*)
next
case (*step ys zs*)
thus *?case* **by** (*metis listrel-reflcl-if-listrel1 listrel-rtrancl-trans*)
qed
qed
qed

lemma *rtrancl-listrel1-if-listrel*:

$(xs,ys) \in listrel\ r \Longrightarrow (xs,ys) \in (listrel1\ r)^*$
by(*metis listrel-rtrancl-eq-rtrancl-listrel1 subsetD*[*OF listrel-mono*] *r-into-rtrancl subsetI*)

lemma *listrel-subset-rtrancl-listrel1*: $listrel\ r \subseteq (listrel1\ r)^*$
by(*fast intro*:*rtrancl-listrel1-if-listrel*)

66.5 Size function

lemma [*measure-function*]: $is\text{-}measure\ f \Longrightarrow is\text{-}measure\ (size\text{-}list\ f)$
by (*rule is-measure-trivial*)

lemma [*measure-function*]: $is\text{-}measure\ f \Longrightarrow is\text{-}measure\ (size\text{-}option\ f)$
by (*rule is-measure-trivial*)

lemma *size-list-estimation*[*termination-simp*]:
$x \in set\ xs \Longrightarrow y < f\ x \Longrightarrow y < size\text{-}list\ f\ xs$
by (*induct xs*) *auto*

lemma *size-list-estimation'*[*termination-simp*]:
$x \in set\ xs \Longrightarrow y \leq f\ x \Longrightarrow y \leq size\text{-}list\ f\ xs$
by (*induct xs*) *auto*

lemma *size-list-map*[*simp*]: $size\text{-}list\ f\ (map\ g\ xs) = size\text{-}list\ (f \circ g)\ xs$
by (*induct xs*) *auto*

lemma *size-list-append*[*simp*]: $size\text{-}list\ f\ (xs\ @\ ys) = size\text{-}list\ f\ xs + size\text{-}list\ f\ ys$
by (*induct xs*, *auto*)

lemma *size-list-pointwise*[*termination-simp*]:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow size\text{-}list\ f\ xs \leq size\text{-}list\ g\ xs$
by (*induct xs*) *force*+

66.6 Monad operation

definition *bind* :: $'a\ list \Rightarrow ('a \Rightarrow 'b\ list) \Rightarrow 'b\ list$ **where**
$bind\ xs\ f = concat\ (map\ f\ xs)$

hide-const (**open**) *bind*

lemma *bind-simps* [*simp*]:
$List.bind\ []\ f = []$
$List.bind\ (x\ \#\ xs)\ f = f\ x\ @\ List.bind\ xs\ f$
by (*simp-all add*: *bind-def*)

lemma *list-bind-cong* [*fundef-cong*]:
assumes $xs = ys$ $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x = g\ x)$
shows $List.bind\ xs\ f = List.bind\ ys\ g$
proof −
from *assms*(*2*) **have** $List.bind\ xs\ f = List.bind\ xs\ g$

by (*induction xs*) *simp-all*
with *assms*(*1*) **show** *?thesis* **by** *simp*
qed

lemma *set-list-bind*: *set* (*List.bind xs f*) = (⋃*x*∈*set xs*. *set* (*f x*))
by (*induction xs*) *simp-all*

66.7 Code generation

66.7.1 Counterparts for set-related operations

context
begin

qualified definition *member* :: ‹*'a list* ⇒ *'a* ⇒ *bool*› — only for code generation
where *member-iff* [*code-abbrev*, *simp*]: ‹*member xs x* ⟷ *x* ∈ *set xs*›

Use *member* only for generating executable code. Otherwise use $x \in set\ xs$ instead — it is much easier to reason about.

qualified lemma *member-code* [*code*, *no-atp*]:
‹*member* [] *y* ⟷ *False*›
‹*member* (*x* # *xs*) *y* ⟷ *x* = *y* ∨ *member xs y*›
by *auto*

qualified lemma *Collect-member* [*code-unfold*, *no-atp*]: — make preprocessor setup confluent
‹{*x*. *List.member xs x* ∧ *P x*} = *Set.filter P* (*set xs*)›
by *simp*

qualified lemma *Collect-pair-member* [*code-unfold*, *no-atp*]: — make preprocessor setup confluent
‹{(*x*, *y*). *List.member xs* (*x*, *y*) ∧ *P x y*} = *Set.filter* (λ(*x*, *y*). *P x y*) (*set xs*)›
by *auto*

qualified lemma *Collect-triple-member* [*code-unfold*, *no-atp*]: — make preprocessor setup confluent
‹{(*x*, *y*, *z*). *List.member xs* (*x*, *y*, *z*) ∧ *P x y z*} = *Set.filter* (λ(*x*, *y*, *z*). *P x y z*) (*set xs*)›
by *auto*

end

lemma *list-all-iff* [*code-abbrev*]:
‹*list-all P xs* ⟷ *Ball* (*set xs*) *P*›
by (*simp add*: *list.pred-set*)

definition *list-ex* :: ‹(*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool*›
where *list-ex-iff* [*code-abbrev*]: ‹*list-ex P xs* ⟷ *Bex* (*set xs*) *P*›

definition *list-ex1* :: ‹(*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool*›

where *list-ex1-iff* [*code-abbrev*]: ‹*list-ex1 P xs* ⟷ *Set.can-select P* (*set xs*)›

Usually you should prefer ∀ *x*∈*set xs*, ∃ *x*∈*set xs* and ∃ !*x*. *x*∈*set xs* ∧ - over *list-all*, *list-ex* and *list-ex1* in specifications.

lemma *list-all-Nil-iff* [*code*, *no-atp*]:
‹*list-all P* [] ⟷ *True*›
by (*simp add*: *list-all-iff*)

lemma *list-all-Cons-iff* [*code*, *no-atp*]:
‹*list-all P* (*x* # *xs*) ⟷ *P x* ∧ *list-all P xs*›
by (*simp add*: *list-all-iff*)

lemma *list-ex-Nil-iff* [*simp*, *code*, *no-atp*]:
‹*list-ex P* [] ⟷ *False*›
by (*simp add*: *list-ex-iff*)

lemma *list-ex-Cons-iff* [*simp*, *code*, *no-atp*]:
‹*list-ex P* (*x* # *xs*) ⟷ *P x* ∨ *list-ex P xs*›
by (*simp add*: *list-ex-iff*)

lemma *list-ex1-Nil-iff* [*simp*, *code*, *no-atp*]:
‹*list-ex1 P* [] ⟷ *False*›
by (*auto simp add*: *list-ex1-iff*)

lemma *list-ex1-Cons-iff* [*simp*, *code*, *no-atp*]:
‹*list-ex1 P* (*x* # *xs*) ⟷ (*if P x then list-all* (λ*y*. ¬ *P y* ∨ *x* = *y*) *xs else list-ex1 P xs*)›
by (*auto simp add*: *list-ex1-iff list-all-iff*)

lemma *list-all-append* [*simp*]:
‹*list-all P* (*xs* @ *ys*) ⟷ *list-all P xs* ∧ *list-all P ys*›
by (*auto simp add*: *list-all-iff*)

lemma *list-ex-append* [*simp*]:
‹*list-ex P* (*xs* @ *ys*) ⟷ *list-ex P xs* ∨ *list-ex P ys*›
by (*auto simp add*: *list-ex-iff*)

lemma *list-all-rev* [*simp*]:
‹*list-all P* (*rev xs*) ⟷ *list-all P xs*›
by (*simp add*: *list-all-iff*)

lemma *list-ex-rev* [*simp*]:
‹*list-ex P* (*rev xs*) ⟷ *list-ex P xs*›
by (*simp add*: *list-ex-iff*)

lemma *list-all-length*:
‹*list-all P xs* ⟷ (∀ *n* < *length xs*. *P* (*xs* ! *n*))›
by (*auto simp add*: *list-all-iff set-conv-nth*)

lemma *list-ex-length*:
‹*list-ex P xs* ⟷ (∃ *n* < *length xs*. *P* (*xs* ! *n*))›
by (*auto simp add*: *list-ex-iff set-conv-nth*)

lemma *list-all-cong* [*fundef-cong*]:
‹*list-all f xs* = *list-all g ys*›
if ‹*xs* = *ys*› ‹(⋀*x*. *x* ∈ *set ys* ⟹ *f x* = *g x*)›
using *that* **by** (*rule list.pred-cong*)

lemma *list-ex-cong* [*fundef-cong*]:
‹*list-ex f xs* = *list-ex g ys*›
if ‹*xs* = *ys*› ‹(⋀*x*. *x* ∈ *set ys* ⟹ *f x* = *g x*)›
using *that* **by** (*simp add*: *list-ex-iff*)

context
begin

qualified definition *superset* :: ‹'*a list* ⇒ '*a list* ⇒ *bool*›
where *superset-iff* [*code-abbrev*, *simp*]: ‹*superset ys xs* ⟷ *set xs* ⊆ *set ys*›

lemma [*code*, *no-atp*]:
‹*superset xs* = *list-all* (λ*x*. *x* ∈ *set xs*)›
by (*auto simp*: *fun-eq-iff list-all-iff*)

end

Executable checks for relations on sets

definition *listrel1p* :: ‹('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool*› — only for code generation
where ‹*listrel1p r xs ys* ⟷ (*xs*, *ys*) ∈ *listrel1* {(*x*, *y*). *r x y*}›

lemma [*code-unfold*]:
‹(*xs*, *ys*) ∈ *listrel1 r* ⟷ *listrel1p* (λ*x y*. (*x*, *y*) ∈ *r*) *xs ys*›
by (*simp add*: *listrel1p-def*)

lemma [*code*]:
‹*listrel1p r* [] *xs* ⟷ *False*›
‹*listrel1p r xs* [] ⟷ *False*›
‹*listrel1p r* (*x* # *xs*) (*y* # *ys*) ⟷
r x y ∧ *xs* = *ys* ∨ *x* = *y* ∧ *listrel1p r xs ys*›
by (*simp-all add*: *listrel1p-def*)

definition *lexordp* :: ‹('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool*› — only for code generation
where ‹*lexordp r xs ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*x*, *y*). *r x y*}›

lemma [*code-unfold*]:
‹(*xs*, *ys*) ∈ *lexord r* = *lexordp* (λ*x y*. (*x*, *y*) ∈ *r*) *xs ys*›
by (*simp add*: *lexordp-def*)

lemma [*code*]:
‹*lexordp r xs* [] ⟷ *False*›
‹*lexordp r* [] (*y* # *ys*) ⟷ *True*›
‹*lexordp r* (*x* # *xs*) (*y* # *ys*) ⟷
r x y ∨ (*x* = *y* ∧ *lexordp r xs ys*)›
by (*simp-all add*: *add*: *lexordp-def*)

Executable intervals

context *preorder*
begin

lemma *forall-less-eq-iff* [*code-unfold*]:
‹(∀ *n*≤*b*. *P n*) ⟷ (∀ *n*∈{..*b*}. *P n*)›
by *auto*

lemma *exists-less-eq-iff* [*code-unfold*]:
‹(∃ *n*≤*b*. *P n*) ⟷ (∃ *n*∈{..*b*}. *P n*)›
by *auto*

lemma *forall-less-iff* [*code-unfold*]:
‹(∀ *n*<*b*. *P n*) ⟷ (∀ *n*∈{..<*b*}. *P n*)›
by *auto*

lemma *exists-less-iff* [*code-unfold*]:
‹(∃ *n*<*b*. *P n*) ⟷ (∃ *n*∈{..<*b*}. *P n*)›
by *auto*

lemma *forall-greater-eq-iff* [*code-unfold*]:
‹(∀ *n*≥*a*. *P n*) ⟷ (∀ *n*∈{*a*..}. *P n*)›
by *auto*

lemma *exists-greater-eq-iff* [*code-unfold*]:
‹(∃ *n*≥*a*. *P n*) ⟷ (∃ *n*∈{*a*..}. *P n*)›
by *auto*

lemma *forall-greater-iff* [*code-unfold*]:
‹(∀ *n*>*a*. *P n*) ⟷ (∀ *n*∈{*a*<..}. *P n*)›
by *auto*

lemma *exists-greater-iff* [*code-unfold*]:
‹(∃ *n*>*a*. *P n*) ⟷ (∃ *n*∈{*a*<..}. *P n*)›
by *auto*

end

class *interval* = *linorder* + *comm-semiring-1-cancel* +
assumes *finite-atLeastAtMost*: ‹*finite* {*a*..*b*}›
assumes *dec-less-imp-less-eq*: ‹*a* − *1* < *b* ⟹ *a* ≤ *b*›

assumes *less-inc-imp-less-eq*: ‹$a < b + 1 \Longrightarrow a \leq b$›
assumes *dec-greater-eq-self-imp-bot*: ‹$a \leq a - 1 \Longrightarrow a \leq c$›
assumes *inc-less-eq-self-imp-top*: ‹$b + 1 \leq b \Longrightarrow d \leq b$›
begin

context
begin

qualified lemma *less-imp-less-eq-dec*:
‹$c < b \Longrightarrow a < b \Longrightarrow a \leq b - 1$›
using *local.dec-less-imp-less-eq local.not-less* **by** *blast*

qualified lemma *less-imp-in-less-eq*:
‹$a < c \Longrightarrow a < b \Longrightarrow a + 1 \leq b$›
using *local.less-inc-imp-less-eq local.not-less* **by** *blast*

qualified lemma *less-eq-dec-imp-less*:
‹$c < b \Longrightarrow a \leq b - 1 \Longrightarrow a < b$›
using *local.dec-greater-eq-self-imp-bot local.dual-order.trans local.not-le* **by** *blast*

qualified lemma *inc-less-eq-imp-less*:
‹$a < c \Longrightarrow a + 1 \leq b \Longrightarrow a < b$›
using *local.inc-less-eq-self-imp-top local.not-le local.order.strict-trans2* **by** *blast*

qualified definition *interval* :: ‹$'a \Rightarrow 'a \Rightarrow 'a$ *list*› — only for code generation
where *interval-eq*: ‹*interval a b* = *sorted-list-of-set* $\{a..b\}$›

qualified lemma *set-interval-eq* [*simp*]:
‹*set* (*interval a b*) = $\{a..b\}$›
using *finite-atLeastAtMost* [*of a b*] **by** (*simp add*: *interval-eq*)

qualified lemma *distinct-interval* [*simp*]:
‹*distinct* (*interval a b*)›
by (*simp add*: *interval-eq*)

qualified lemma *interval-code* [*code*]:
‹*interval a b* = (*if a* < *b then a* # *interval* (*a* + *1*) *b else if a* = *b then* [*a*] *else* [])›
proof −
consider (*less*) ‹$a < b$› | (*eq*) ‹$a = b$› | (*greater*) ‹$a > b$›
using *less-linear* **by** *blast*
then show *?thesis* **proof** *cases*
case *less*
then have ‹$\{a..b\}$ = *insert a* $\{a + 1..b\}$›
by (*auto simp add*: *not-le dest*: *less-imp-le local.inc-less-eq-imp-less dest*!: *less-inc-imp-less-eq*)
moreover have ‹$\{a + 1..b\} - \{a\} = \{a + 1..b\}$›
using *less* **by** (*auto dest*: *local.inc-less-eq-imp-less*)
moreover have ‹*insort a* (*sorted-list-of-set* $\{a + 1..b\}$) = *a* # *sorted-list-of-set*

```
{a + 1..b}›
      using finite-atLeastAtMost [of ‹a + 1› b] less
      by (auto intro!: insort-is-Cons dest: local.inc-less-eq-imp-less less-imp-le)
    ultimately show ?thesis
      using less finite-atLeastAtMost [of ‹a + 1› b]
      by (simp add: interval-eq)
  next
    case eq
    then show ?thesis
      by (simp add: interval-eq)
  next
    case greater
    then show ?thesis
      by (auto simp add: interval-eq)
  qed
qed

qualified lemma atLeastAtMost-eq-interval [code]:
  ‹{a..b} = set (interval a b)›
  by simp

qualified lemma atLeastLessThan-eq-interval [code]:
  ‹{a..<b} = (let d = b - 1 in if d < b then set (interval a d) else {})›
 by (auto simp add: Let-def not-less local.less-imp-less-eq-dec intro: dec-greater-eq-self-imp-bot)

qualified lemma greaterThanAtMost-eq-interval [code]:
  ‹{a<..b} = (let c = a + 1 in if a < c then set (interval c b) else {})›
 by (auto simp add: Let-def not-less dec-less-imp-less-eq intro: inc-less-eq-self-imp-top)

qualified lemma greaterThanLessThan-eq-interval [code]:
  ‹{a<..<b} = (let c = a + 1; d = b - 1 in if a < c ∧ d < b then set (interval c
d) else {})›
  by (auto simp add: Let-def not-less dec-less-imp-less-eq
   dest: local.less-imp-less-eq-dec local.inc-less-eq-imp-less local.less-eq-dec-imp-less)

qualified definition all-interval :: ‹('a ⇒ bool) ⇒ 'a ⇒ 'a ⇒ bool› — only for
code generation
  where all-interval-iff [code-post, simp]: ‹all-interval P a b ⟷ (∀n∈{a..b}. P
n)›

qualified lemma all-interval-code [code]:
  ‹all-interval P a b ⟷ ((a < b ⟶ P a ∧ all-interval P (a + 1) b) ∧ (a = b
⟶ P a))›
  by (simp only: all-interval-iff interval-code [of a b] flip: set-interval-eq) auto

qualified lemma forall-atLeastAtMost-iff [code-unfold]:
  ‹(∀n∈{a..b}. P n) ⟷ all-interval P a b›
  by simp
```

qualified lemma *exists-atLeastAtMost-iff* [*code-unfold*]:
‹(∃ n∈{a..b}. P n) ⟷ ¬ all-interval (Not ∘ P) a b›
using *forall-atLeastAtMost-iff* [*of a b ‹Not ∘ P›*] **by** *simp*

qualified lemma *forall-atLeastLessThan-iff* [*code-unfold*]:
‹(∀ n∈{a..<b}. P n) ⟷ (let d = b − 1 in d < b ⟶ all-interval P a d)›
by (*auto simp add*: *not-less Let-def intro*: *local.less-eq-dec-imp-less local.less-imp-less-eq-dec elim*!: *bspec*)

qualified lemma *exists-atLeastLessThan-iff* [*code-unfold*]:
‹(∃ n∈{a..<b}. P n) ⟷ (let d = b − 1 in d < b ∧ ¬ all-interval (Not ∘ P) a d)›
using *forall-atLeastLessThan-iff* [*of a b ‹Not ∘ P›*] **by** (*auto simp add*: *Let-def*)

qualified lemma *forall-greaterThanAtMost-iff* [*code-unfold*]:
‹(∀ n∈{a<..b}. P n) ⟷ (let c = a + 1 in a < c ⟶ all-interval P c b)›
by (*auto simp add*: *Let-def not-less intro*: *local.less-imp-in-less-eq local.inc-less-eq-imp-less elim*!: *bspec*)

qualified lemma *exists-greaterThanAtMost-iff* [*code-unfold*]:
‹(∃ n∈{a<..b}. P n) ⟷ (let c = a + 1 in a < c ∧ ¬ all-interval (Not ∘ P) c b)›
using *forall-greaterThanAtMost-iff* [*of a b ‹Not ∘ P›*] **by** (*auto simp add*: *Let-def*)

qualified lemma *forall-greaterThanLessThan-iff* [*code-unfold*]:
‹(∀ n∈{a<..<b}. P n) ⟷ (let c = a + 1; d = b − 1 in a < c ⟶ d < b ⟶ all-interval P c d)›
by (*auto simp add*: *Let-def not-less local.less-imp-in-less-eq local.less-imp-less-eq-dec intro*: *local.inc-less-eq-imp-less local.less-eq-dec-imp-less elim*!: *bspec*)

qualified lemma *exists-greaterThanLessThan-iff* [*code-unfold*]:
‹(∃ n∈{a<..<b}. P n) ⟷ (let c = a + 1; d = b − 1 in a < c ∧ d < b ∧ ¬ all-interval (Not ∘ P) c d)›
using *forall-greaterThanLessThan-iff* [*of a b ‹Not ∘ P›*] **by** (*auto simp add*: *Let-def*)

end

end

class *interval-top* = *interval* + *order-top*
begin

lemma *atLeast-eq-atLeastAtMost-top* [*code*, *code-unfold*]:
‹{a..} = {a..top}›
by *auto*

lemma *greaterThan-eq-greaterThanAtMost-top* [*code*, *code-unfold*]:
‹{a<..} = {a<..top}›

by *auto*

end

class *interval-bot* = *interval* + *order-bot*
begin

lemma *atMost-eq-atLeastAtMost-bot* [*code*, *code-unfold*]:
‹{..*b*} = {*bot*..*b*}›
by *auto*

lemma *lessThan-eq-atLeastLessThan-bot* [*code*, *code-unfold*]:
‹{..<*b*} = {*bot*..<*b*}›
by *auto*

end

instance *nat* :: *interval-bot*
by *standard simp-all*

instance *int* :: *interval*
by *standard simp-all*

context
begin

qualified lemma *interval-eq-upt* [*simp*]:
‹*List.interval m n* = [*m*..<*Suc n*]›
by (*simp add*: *List.interval-eq flip*: *atLeastLessThanSuc-atLeastAtMost*)

qualified lemma *interval-eq-upto* [*simp*]:
‹*List.interval i k* = [*i*..*k*]›
by (*simp add*: *List.interval-eq*)

end

66.7.2 Special implementations

context
begin

qualified definition *map-tailrec-rev* :: ‹('*a* ⇒ '*b*) ⇒ '*a list* ⇒ '*b list* ⇒ '*b list*› — only for code generation
where *map-tailrec-rev* [*simp*]: ‹*map-tailrec-rev f as bs* = *rev* (*map f as*) @ *bs*›

Optional tail recursive version of *map*. Can avoid stack overflow in some target languages. Do not use for proving.

qualified lemma *map-tailrec-rev-code* [*code*, *no-atp*]:
‹*map-tailrec-rev f* [] *bs* = *bs*›

‹*map-tailrec-rev f* (*a* # *as*) *bs* = *map-tailrec-rev f as* (*f a* # *bs*)›
by *simp-all*

qualified definition *map-tailrec* :: ‹('*a* ⇒ '*b*) ⇒ '*a list* ⇒ '*b list*› — only for code generation
where *map-tailrec-eq* [*simp*]: ‹*map-tailrec* = *map*›

qualified lemma *map-tailrec-code* [*code*, *no-atp*]:
‹*map-tailrec f as* = *rev* (*map-tailrec-rev f as* []›
by *simp*

Potential code equation:

qualified lemma *map-eq-map-tailrec*:
‹*map* = *map-tailrec*›
by *simp*

end

definition *map-filter* :: ‹('*a* ⇒ '*b option*) ⇒ '*a list* ⇒ '*b list*›
where [*code-post*]: *map-filter f xs* = *map* (*the* ∘ *f*) (*filter* (λ*x*. *f x* ≠ *None*) *xs*)

Operation *map-filter* avoids intermediate lists on execution – do not use for proving.

lemma *map-filter-simps* [*simp*, *code*, *no-atp*]:
‹*map-filter f* [] = []›
‹*map-filter f* (*x* # *xs*) = (*case f x of None* ⇒ *map-filter f xs* | *Some y* ⇒ *y* # *map-filter f xs*)›
by (*simp-all add*: *map-filter-def split*: *option.split*)

lemma *map-filter-map-filter* [*code-unfold*]:
‹*map f* (*filter P xs*) = *map-filter* (λ*x*. *if P x then Some* (*f x*) *else None*) *xs*›
by (*simp add*: *map-filter-def*)

lemma *map-map-filter*: *map f* (*map-filter g xs*) = *map-filter* (*map-option f* ∘ *g*) *xs*
by (*induction xs*) (*auto simp*: *option.case-eq-if*)

hide-const (**open**) *map-filter*

66.7.3 Operations for optimization and efficiency

context
begin

qualified definition *null* :: ‹'*a list* ⇒ *bool*› — only for code generation
where *null-iff* [*code-abbrev*, *simp*]: ‹*null xs* ⟷ *xs* = []›

qualified lemma *null-code* [*code*, *no-atp*]:
‹*null* [] ⟷ *True*›

‹*null* (*x* # *xs*) ⟷ *False*›
by *simp-all*

qualified lemma *equal-Nil-null* [*code-unfold*, *no-atp*]:
‹*HOL.equal xs* [] ⟷ *null xs*›
‹*HOL.equal* [] = *null*›
by (*auto simp add*: *equal*)

qualified definition *length-tailrec* :: ‹'*a list* ⇒ *nat* ⇒ *nat*› — only for code generation
where *length-tailrec-eq* [*simp*]: ‹*length-tailrec xs* = (+) (*length xs*)›

optimized code (tail-recursive) for *length*

qualified lemma *length-tailrec-code* [*code*, *no-atp*]:
‹*length-tailrec* [] *n* = *n*›
‹*length-tailrec* (*x* # *xs*) *n* = *length-tailrec xs* (*Suc n*)›
by *simp-all*

qualified lemma *length-code* [*code*, *no-atp*]:
‹*length xs* = *length-tailrec xs 0*›
by *simp*

qualified definition *maps* :: ‹('*a* ⇒ '*b list*) ⇒ '*a list* ⇒ '*b list*› — only for code generation
where *maps-eq* [*code-abbrev*, *simp*]: ‹*maps f xs* = *concat* (*map f xs*)›

Operation *maps* avoids intermediate lists on execution – do not use for proving.

qualified lemma *maps-code* [*code*, *no-atp*]:
‹*maps f* [] = []›
‹*maps f* (*x* # *xs*) = *f x* @ *maps f xs*›
by *simp-all*

end

66.7.4 Implementation of sets by lists

lemma *is-empty-set* [*code*]:
Set.is-empty (*set xs*) ⟷ *List.null xs*
by *simp*

lemma *empty-set* [*code*]:
{} = *set* []
by *simp*

lemma *UNIV-coset* [*code*]:
UNIV = *List.coset* []
by *simp*

lemma *compl-set* [*code*]:
  − *set xs* = *List.coset xs*
  by *simp*

lemma *compl-coset* [*code*]:
  − *List.coset xs* = *set xs*
  by *simp*

lemma [*code*]:
  x ∈ *set xs* ⟷ *List.member xs x*
  x ∈ *List.coset xs* ⟷ ¬ *List.member xs x*
  by *simp-all*

lemma *insert-code* [*code*]:
  insert x (*set xs*) = *set* (*List.insert x xs*)
  insert x (*List.coset xs*) = *List.coset* (*removeAll x xs*)
  by *simp-all*

lemma *remove-code* [*code*]:
  Set.remove x (*set xs*) = *set* (*removeAll x xs*)
  Set.remove x (*List.coset xs*) = *List.coset* (*List.insert x xs*)
  by (*simp-all add*: *set-eq-iff ac-simps*)

lemma *filter-set* [*code*]:
  Set.filter P (*set xs*) = *set* (*filter P xs*)
  by *simp*

lemma *image-set* [*code*]:
  image f (*set xs*) = *set* (*map f xs*)
  by *simp*

lemma *subset-code* [*code*]:
  set xs ⊆ *B* ⟷ (∀ *x*∈*set xs*. *x* ∈ *B*)
  A ⊆ *List.coset ys* ⟷ (∀ *y*∈*set ys*. *y* ∉ *A*)
  List.coset [] ⊆ *set* [] ⟷ *False*
  by *auto*

lemma *Ball-set* [*code*]:
  Ball (*set xs*) *P* ⟷ *list-all P xs*
  by (*simp add*: *list-all-iff*)

lemma *Bex-set* [*code*]:
  Bex (*set xs*) *P* ⟷ *list-ex P xs*
  by (*simp add*: *list-ex-iff*)

lemma *card-set* [*code*]:
  card (*set xs*) = *length* (*remdups xs*)
  by (*simp add*: *length-remdups-card-conv*)

lemma *the-elem-set* [*code*]:
the-elem (*set* [*x*]) = *x*
by *simp*

lemma *Pow-set* [*code*]:
Pow (*set* []) = {{}}
Pow (*set* (*x* # *xs*)) = (*let A* = *Pow* (*set xs*) *in A* ∪ *insert x* ‘ *A*)
by (*simp-all add*: *Pow-insert Let-def*)

lemma *these-set-code* [*code*]:
‹*Option.these* (*set xs*) = *set* (*List.map-filter* (λ*x*. *x*) *xs*)›
by (*simp add*: *Option.these-eq Option.is-none-def set-eq-iff map-filter-def*)

lemma *image-filter-set-eq* [*code*]:
‹*Option.image-filter f* (*set xs*) = *set* (*List.map-filter f xs*)›
apply (*simp add*: *Option.image-filter-eq these-set-code set-eq-iff flip*: *set-map*)
apply (*auto simp add*: *map-filter-def image-iff*)
done

lemma *can-select-set-list-ex1* [*code*]:
Set.can-select P (*set A*) = *list-ex1 P A*
by (*simp add*: *list-ex1-iff*)

lemma *product-code* [*code*]:
Product-Type.product (*set xs*) (*set ys*) = *set* [(*x*, *y*). *x* ← *xs*, *y* ← *ys*]
by (*auto simp add*: *Product-Type.product-def*)

lemma *Id-on-set* [*code*]:
Id-on (*set xs*) = *set* [(*x*, *x*). *x* ← *xs*]
by (*auto simp add*: *Id-on-def*)

lemma *Image-code* [*code*]:
R ‘‘ *S* = *Option.image-filter* (λ(*x*, *y*). *if x* ∈ *S then Some y else None*) *R*
apply (*simp add*: *Option.image-filter-eq case-prod-unfold Option.these-eq*)
apply *force*
done

lemma *trancl-set-ntrancl* [*code*]:
trancl (*set xs*) = *ntrancl* (*card* (*set xs*) − *1*) (*set xs*)
by (*simp add*: *finite-trancl-ntranl*)

lemma *set-relcomp* [*code*]:
set xys O set yzs = *set* ([(*fst xy*, *snd yz*). *xy* ← *xys*, *yz* ← *yzs*, *snd xy* = *fst yz*])
by *simp* (*auto simp add*: *Bex-def image-def*)

lemma *wf-set*:
wf (*set xs*) = *acyclic* (*set xs*)
by (*simp add*: *wf-iff-acyclic-if-finite*)

lemma *wf-code-set* [*code*]:
wf-code (*set xs*) = *acyclic* (*set xs*)
unfolding *wf-code-def* **using** *wf-set* **.**

66.7.5 Pretty lists

ML ‹
(∗ *Code generation for list literals.* ∗)

signature LIST-CODE =
sig
val add-literal-list: *string* −> *theory* −> *theory*
end;

structure List-Code : *LIST-CODE* =
struct

open Basic-Code-Thingol;

fun implode-list t =
let
fun dest-cons (*IConst* { *sym* = *Code-Symbol.Constant* ***const-name*** ‹*Cons*›, ...
} $ *t1* $ *t2*) = *SOME* (*t1*, *t2*)
| *dest-cons* - = *NONE*;
val (*ts*, *t′*) = *Code-Thingol.unfoldr dest-cons t*;
in case t′
of IConst { *sym* = *Code-Symbol.Constant* ***const-name*** ‹*Nil*›, ... } => *SOME*
ts
| - => *NONE*
end;

fun print-list (*target-fxy*, *target-cons*) *pr fxy t1 t2* =
Code-Printer.brackify-infix (*target-fxy*, *Code-Printer.R*) *fxy* (
pr (*Code-Printer.INFX* (*target-fxy*, *Code-Printer.X*)) *t1*,
Pretty.str target-cons,
pr (*Code-Printer.INFX* (*target-fxy*, *Code-Printer.R*)) *t2*
);

fun add-literal-list target =
let
fun pretty literals pr - *vars fxy* [(*t1*, -), (*t2*, -)] =
case Option.map (*cons t1*) (*implode-list t2*)
of SOME ts =>
Code-Printer.literal-list literals (*map* (*pr vars Code-Printer.NOBR*) *ts*)
| *NONE* =>
print-list (*Code-Printer.infix-cons literals*) (*pr vars*) *fxy t1 t2*;
in
Code-Target.set-printings (*Code-Symbol.Constant* (***const-name*** ‹*Cons*›,
[(*target*, *SOME* (*Code-Printer.complex-const-syntax* (2, *pretty*)))]))

```
  end

end;
›
```

```
code-printing
  type-constructor list ⇀
    (SML) - list
    and (OCaml) - list
    and (Haskell) ![(-)]
    and (Scala) List[(-)]
| constant Nil ⇀
    (SML) []
    and (OCaml) []
    and (Haskell) []
    and (Scala) !Nil
| class-instance list :: equal ⇀
    (Haskell) –
| constant HOL.equal :: 'a list ⇒ 'a list ⇒ bool ⇀
    (Haskell) infix 4 ==
```

```
setup ‹fold (List-Code.add-literal-list) [SML, OCaml, Haskell, Scala]›
```

```
code-reserved
  (SML) list
  and (OCaml) list
```

66.7.6 Use convenient predefined operations

```
code-printing
  constant (@) ⇀
    (SML) infixr 7 @
    and (OCaml) infixr 6 @
    and (Haskell) infixr 5 ++
    and (Scala) infixl 7 ++
| constant map ⇀
    (Haskell) map
| constant filter ⇀
    (Haskell) filter
| constant concat ⇀
    (Haskell) concat
| constant List.maps ⇀
    (Haskell) concatMap
| constant rev ⇀
    (Haskell) reverse
| constant zip ⇀
    (Haskell) zip
| constant List.null ⇀
    (Haskell) null
```

| **constant** *takeWhile* ⇀
(*Haskell*) *takeWhile*
| **constant** *dropWhile* ⇀
(*Haskell*) *dropWhile*
| **constant** *list-all* ⇀
(*Haskell*) *all*
| **constant** *list-ex* ⇀
(*Haskell*) *any*

66.8 Setup for Lifting/Transfer

66.8.1 Transfer rules for the Transfer package

context includes *lifting-syntax*
begin

lemma *tl-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *tl tl*
unfolding *tl-def*[*abs-def*] **by** *transfer-prover*

lemma *butlast-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *butlast butlast*
by (*rule rel-funI*, *erule list-all2-induct*, *auto*)

lemma *append-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A* ===> *list-all2 A*) *append append*
unfolding *List.append-def* **by** *transfer-prover*

lemma *rev-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *rev rev*
unfolding *List.rev-def* **by** *transfer-prover*

lemma *filter-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> *list-all2 A*) *filter filter*
unfolding *List.filter-def* **by** *transfer-prover*

lemma *fold-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *B*) ===> *list-all2 A* ===> *B* ===> *B*) *fold fold*
unfolding *List.fold-def* **by** *transfer-prover*

lemma *foldr-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> *B*) ===> *list-all2 A* ===> *B* ===> *B*) *foldr foldr*
unfolding *List.foldr-def* **by** *transfer-prover*

lemma *foldl-transfer* [*transfer-rule*]:
((*B* ===> *A* ===> *B*) ===> *B* ===> *list-all2 A* ===> *B*) *foldl foldl*
unfolding *List.foldl-def* **by** *transfer-prover*

lemma *concat-transfer* [*transfer-rule*]:
(*list-all2* (*list-all2 A*) ===> *list-all2 A*) *concat concat*

unfolding *List.concat-def* **by** *transfer-prover*

lemma *drop-transfer* [*transfer-rule*]:
(*(=) ===> list-all2 A ===> list-all2 A) drop drop*
unfolding *List.drop-def* **by** *transfer-prover*

lemma *take-transfer* [*transfer-rule*]:
(*(=) ===> list-all2 A ===> list-all2 A) take take*
unfolding *List.take-def* **by** *transfer-prover*

lemma *list-update-transfer* [*transfer-rule*]:
(*list-all2 A ===> (=) ===> A ===> list-all2 A) list-update list-update*
unfolding *list-update-def* **by** *transfer-prover*

lemma *takeWhile-transfer* [*transfer-rule*]:
(*(A ===> (=)) ===> list-all2 A ===> list-all2 A) takeWhile takeWhile*
unfolding *takeWhile-def* **by** *transfer-prover*

lemma *dropWhile-transfer* [*transfer-rule*]:
(*(A ===> (=)) ===> list-all2 A ===> list-all2 A) dropWhile dropWhile*
unfolding *dropWhile-def* **by** *transfer-prover*

lemma *zip-transfer* [*transfer-rule*]:
(*list-all2 A ===> list-all2 B ===> list-all2 (rel-prod A B)) zip zip*
unfolding *zip-def* **by** *transfer-prover*

lemma *product-transfer* [*transfer-rule*]:
(*list-all2 A ===> list-all2 B ===> list-all2 (rel-prod A B)) List.product List.product*
unfolding *List.product-def* **by** *transfer-prover*

lemma *product-lists-transfer* [*transfer-rule*]:
(*list-all2 (list-all2 A) ===> list-all2 (list-all2 A)) product-lists product-lists*
unfolding *product-lists-def* **by** *transfer-prover*

lemma *insert-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A ===> list-all2 A ===> list-all2 A) List.insert List.insert*
unfolding *List.insert-def* [*abs-def*] **by** *transfer-prover*

lemma *find-transfer* [*transfer-rule*]:
(*(A ===> (=)) ===> list-all2 A ===> rel-option A) List.find List.find*
unfolding *List.find-def* **by** *transfer-prover*

lemma *those-transfer* [*transfer-rule*]:
(*list-all2 (rel-option P) ===> rel-option (list-all2 P)) those those*
unfolding *List.those-def* **by** *transfer-prover*

lemma *remove1-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*

shows (*A* ===> *list-all2 A* ===> *list-all2 A*) *remove1 remove1*
unfolding *remove1-def* **by** *transfer-prover*

lemma *removeAll-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *list-all2 A* ===> *list-all2 A*) *removeAll removeAll*
unfolding *removeAll-def* **by** *transfer-prover*

lemma *successively-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> (=)) ===> *list-all2 A* ===> (=)) *successively successively*
unfolding *successively-altdef* **by** *transfer-prover*

lemma *distinct-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> (=)) *distinct distinct*
unfolding *distinct-def* **by** *transfer-prover*

lemma *distinct-adj-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*list-all2 A* ===> (=)) *distinct-adj distinct-adj*
unfolding *rel-fun-def*
proof (*intro allI impI*)
fix *xs ys* **assume** *list-all2 A xs ys*
thus *distinct-adj xs* ⟷ *distinct-adj ys*
proof (*induction rule*: *list-all2-induct*)
case (*Cons x xs y ys*)
show *?case*
by (*metis Cons assms bi-unique-def distinct-adj-Cons list.rel-sel*)
qed *auto*
qed

lemma *remdups-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A*) *remdups remdups*
unfolding *remdups-def* **by** *transfer-prover*

lemma *remdups-adj-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A*) *remdups-adj remdups-adj*
proof (*rule rel-funI*, *erule list-all2-induct*)
qed (*auto simp*: *remdups-adj-Cons assms*[*unfolded bi-unique-def*] *split*: *list.splits*)

lemma *replicate-transfer* [*transfer-rule*]:
((=) ===> *A* ===> *list-all2 A*) *replicate replicate*
unfolding *replicate-def* **by** *transfer-prover*

lemma *length-transfer* [*transfer-rule*]:
(*list-all2 A* ===> (=)) *length length*
unfolding *size-list-overloaded-def size-list-def* **by** *transfer-prover*

lemma *rotate1-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A*) *rotate1 rotate1*
unfolding *rotate1-def* **by** *transfer-prover*

lemma *rotate-transfer* [*transfer-rule*]:
((=) ===> *list-all2 A* ===> *list-all2 A*) *rotate rotate*
unfolding *rotate-def* [*abs-def*] **by** *transfer-prover*

lemma *nths-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *rel-set* (=) ===> *list-all2 A*) *nths nths*
unfolding *nths-def* [*abs-def*] **by** *transfer-prover*

lemma *subseqs-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2* (*list-all2 A*)) *subseqs subseqs*
unfolding *subseqs-def* [*abs-def*] **by** *transfer-prover*

lemma *partition-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> *rel-prod* (*list-all2 A*) (*list-all2 A*))
partition partition
unfolding *partition-def* **by** *transfer-prover*

lemma *lists-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set* (*list-all2 A*)) *lists lists*
proof (*rule rel-funI*, *rule rel-setI*)
show ⟦*l* ∈ *lists X*; *rel-set A X Y*⟧ ⟹ ∃ *y*∈*lists Y*. *list-all2 A l y* **for** *X Y l*
proof (*induction l rule*: *lists.induct*)
case (*Cons a l*)
then show *?case*
by (*simp only*: *rel-set-def list-all2-Cons1*, *metis lists.Cons*)
qed *auto*
show ⟦*l* ∈ *lists Y*; *rel-set A X Y*⟧ ⟹ ∃ *x*∈*lists X*. *list-all2 A x l* **for** *X Y l*
proof (*induction l rule*: *lists.induct*)
case (*Cons a l*)
then show *?case*
by (*simp only*: *rel-set-def list-all2-Cons2*, *metis lists.Cons*)
qed *auto*
qed

lemma *set-Cons-transfer* [*transfer-rule*]:
(*rel-set A* ===> *rel-set* (*list-all2 A*) ===> *rel-set* (*list-all2 A*))
set-Cons set-Cons
unfolding *rel-fun-def rel-set-def set-Cons-def*
by (*fastforce simp add*: *list-all2-Cons1 list-all2-Cons2*)

lemma *listset-transfer* [*transfer-rule*]:
(*list-all2* (*rel-set A*) ===> *rel-set* (*list-all2 A*)) *listset listset*
unfolding *listset-def* **by** *transfer-prover*

lemma *null-transfer* [*transfer-rule*]:
(*list-all2 A* ===> (=)) *List.null List.null*
unfolding *rel-fun-def* **by** *auto*

lemma *list-all-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> (=)) *list-all list-all*
using *list.pred-transfer* **by** *blast*

lemma *list-ex-transfer* [*transfer-rule*]:
((*A* ===> (=)) ===> *list-all2 A* ===> (=)) *list-ex list-ex*
unfolding *list-ex-iff* [*abs-def*] **by** *transfer-prover*

lemma *splice-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A* ===> *list-all2 A*) *splice splice*
apply (*rule rel-funI*, *erule list-all2-induct*, *simp add*: *rel-fun-def*, *simp*)
apply (*rule rel-funI*)
apply (*erule-tac xs*=*x* **in** *list-all2-induct*, *simp*, *simp add*: *rel-fun-def*)
done

lemma *shuffles-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *list-all2 A* ===> *rel-set* (*list-all2 A*)) *shuffles shuffles*
proof (*intro rel-funI*, *goal-cases*)
case (*1 xs xs' ys ys'*)
thus *?case*
proof (*induction xs ys arbitrary*: *xs' ys' rule*: *shuffles.induct*)
case (*3 x xs y ys xs' ys'*)
from *3.prems* **obtain** *x' xs''* **where** *xs'*: *xs'* = *x'* # *xs''* **by** (*cases xs'*) *auto*
from *3.prems* **obtain** *y' ys''* **where** *ys'*: *ys'* = *y'* # *ys''* **by** (*cases ys'*) *auto*
have [*transfer-rule*]: *A x x' A y y' list-all2 A xs xs'' list-all2 A ys ys''*
using *3.prems* **by** (*simp-all add*: *xs' ys'*)
have [*transfer-rule*]: *rel-set* (*list-all2 A*) (*shuffles xs* (*y* # *ys*)) (*shuffles xs'' ys'*)
and
[*transfer-rule*]: *rel-set* (*list-all2 A*) (*shuffles* (*x* # *xs*) *ys*) (*shuffles xs' ys''*)
using *3.prems* **by** (*auto intro*!: *3.IH simp*: *xs' ys'*)
have *rel-set* (*list-all2 A*) ((#) *x* ' *shuffles xs* (*y* # *ys*) ∪ (#) *y* ' *shuffles* (*x* # *xs*) *ys*)
((#) *x'* ' *shuffles xs'' ys'* ∪ (#) *y'* ' *shuffles xs' ys''*) **by** *transfer-prover*
thus *?case* **by** (*simp add*: *xs' ys'*)
qed (*auto simp*: *rel-set-def*)
qed

lemma *rtrancl-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A bi-total A*
shows (*rel-set* (*rel-prod A A*) ===> *rel-set* (*rel-prod A A*)) *rtrancl rtrancl*
unfolding *rtrancl-def* **by** *transfer-prover*

lemma *monotone-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *A* ===> (=)) ===> (*B* ===> *B* ===> (=)) ===> (*A*

===> *B*) ===> (=)) *monotone monotone*
unfolding *monotone-def*[*abs-def*] **by** *transfer-prover*

lemma *fun-ord-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total C*
shows ((*A* ===> *B* ===> (=)) ===> (*C* ===> *A*) ===> (*C* ===> *B*) ===> (=)) *fun-ord fun-ord*
unfolding *fun-ord-def*[*abs-def*] **by** *transfer-prover*

lemma *fun-lub-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A bi-unique A*
shows ((*rel-set A* ===> *B*) ===> *rel-set* (*C* ===> *A*) ===> *C* ===> *B*) *fun-lub fun-lub*
unfolding *fun-lub-def*[*abs-def*] **by** *transfer-prover*

end

66.9 Misc

lemma *Ball-set-list-all*:
Ball (*set xs*) *P* ⟷ *list-all P xs*
by (*fact Ball-set*)

lemma *Bex-set-list-ex*:
Bex (*set xs*) *P* ⟷ *list-ex P xs*
by (*fact Bex-set*)

end

67 Sum and product over lists

theory *Groups-List*
imports *List*
begin

locale *monoid-list* = *monoid*
begin

definition *F* :: *'a list* ⇒ *'a*
where
eq-foldr [*code*]: *F xs* = *foldr f xs* **1**

lemma *Nil* [*simp*]:
F [] = **1**
by (*simp add*: *eq-foldr*)

lemma *Cons* [*simp*]:
F (*x* # *xs*) = *x* * *F xs*
by (*simp add*: *eq-foldr*)

lemma *append* [*simp*]:
F (*xs* @ *ys*) = *F xs* ∗ *F ys*
by (*induct xs*) (*simp-all add*: *assoc*)

end

locale *comm-monoid-list* = *comm-monoid* + *monoid-list*
begin

lemma *rev* [*simp*]:
F (*rev xs*) = *F xs*
by (*simp add*: *eq-foldr foldr-fold fold-rev fun-eq-iff assoc left-commute*)

end

locale *comm-monoid-list-set* = *list*: *comm-monoid-list* + *set*: *comm-monoid-set*
begin

lemma *distinct-set-conv-list*:
distinct xs ⟹ *set.F g* (*set xs*) = *list.F* (*map g xs*)
by (*induct xs*) *simp-all*

lemma *set-conv-list* [*code*]:
set.F g (*set xs*) = *list.F* (*map g* (*remdups xs*))
by (*simp add*: *distinct-set-conv-list* [*symmetric*])

lemma *list-conv-set-nth*:
list.F xs = *set.F* (λ*i*. *xs* ! *i*) {*0*..<*length xs*}
proof −
have *xs* = *map* (λ*i*. *xs* ! *i*) [*0*..<*length xs*]
by (*simp add*: *map-nth*)
also have *list.F* ... = *set.F* (λ*i*. *xs* ! *i*) {*0*..<*length xs*}
by (*subst distinct-set-conv-list* [*symmetric*]) *auto*
finally show *?thesis* **.**
qed

lemma *atLeastAtMost-conv-list* [*code-unfold*]:
‹*set.F g* {*a*..*b*} = *list.F* (*map g* (*List.interval a b*))›
by (*simp flip*: *List.set-interval-eq add*: *distinct-set-conv-list*)

lemma *atLeastLessThan-conv-list* [*code-unfold*]:
‹*set.F g* {*a*..<*b*} = (*let d* = *b* − *1 in if d* < *b*
then list.F (*map g* (*List.interval a d*))
else **1**)›
using *List.atLeastLessThan-eq-interval* [*of a b*]
by (*simp flip*: *List.set-interval-eq add*: *distinct-set-conv-list Let-def*)

lemma *greaterThanAtMost-conv-list* [*code-unfold*]:

‹*set.F g* {*a*<..*b*} = (*let c* = *a* + *1 in if a* < *c*
then list.F (*map g* (*List.interval c b*))
else **1**)›
using *List.greaterThanAtMost-eq-interval* [*of a b*]
by (*simp flip*: *List.set-interval-eq add*: *distinct-set-conv-list Let-def*)

lemma *greaterThanLessThan-conv-list* [*code-unfold*]:
‹*set.F g* {*a*<..<*b*} = (*let c* = *a* + *1*; *d* = *b* − *1 in if a* < *c* ∧ *d* < *b*
then list.F (*map g* (*List.interval* (*a* + *1*) (*b* − *1*)))
else **1**)›
using *List.greaterThanLessThan-eq-interval* [*of a b*]
by (*simp flip*: *List.set-interval-eq add*: *distinct-set-conv-list Let-def*)

end

67.1 List summation

context *monoid-add*
begin

sublocale *sum-list*: *monoid-list plus 0*
defines
sum-list = *sum-list.F* **..**

end

lemma *sum-list-bound*: ∀ *n* ∈ *set ns*. *n* ≤ *k* ⟹ *sum-list ns* ≤ *k* ∗ *length ns*
by (*induction ns*) *auto*

context *comm-monoid-add*
begin

sublocale *sum-list*: *comm-monoid-list plus 0*
rewrites
monoid-list.F plus 0 = *sum-list*
proof −
show *comm-monoid-list plus 0* **..**
then interpret *sum-list*: *comm-monoid-list plus 0* **.**
from *sum-list-def* **show** *monoid-list.F plus 0* = *sum-list* **by** *simp*
qed

sublocale *sum*: *comm-monoid-list-set plus 0*
rewrites
monoid-list.F plus 0 = *sum-list*
and *comm-monoid-set.F plus 0* = *sum*
proof −
show *comm-monoid-list-set plus 0* **..**
then interpret *sum*: *comm-monoid-list-set plus 0* **.**
from *sum-list-def* **show** *monoid-list.F plus 0* = *sum-list* **by** *simp*

from *sum-def* **show** *comm-monoid-set.F plus 0 = sum* **by** (*auto intro*: *sym*)
qed

end

Some syntactic sugar for summing a function over a list:

open-bundle *sum-list-syntax*
begin

syntax (*ASCII*)
-sum-list :: *pttrn* => *'a list* => *'b* => *'b* (‹(‹*indent=3 notation=‹binder SUM››SUM -<−-. -)›* [*0*, *51*, *10*] *10*)
syntax
-sum-list :: *pttrn* => *'a list* => *'b* => *'b* (‹(‹*indent=3 notation=‹binder* $\sum$*››*$\sum$ *-←-. -)›* [*0*, *51*, *10*] *10*)
syntax-consts
-sum-list == *sum-list*
translations — Beware of argument permutation!
$\sum x{\leftarrow}xs.\ b$ == *CONST sum-list* (*CONST map* ($\lambda x.\ b$) *xs*)

end

context
includes *lifting-syntax*
begin

lemma *sum-list-transfer* [*transfer-rule*]:
(*list-all2 A* ===> *A*) *sum-list sum-list*
if [*transfer-rule*]: *A 0 0* (*A* ===> *A* ===> *A*) (+) (+)
unfolding *sum-list.eq-foldr* [*abs-def*]
by *transfer-prover*

end

TODO duplicates

lemmas *sum-list-simps* = *sum-list.Nil sum-list.Cons*
lemmas *sum-list-append* = *sum-list.append*
lemmas *sum-list-rev* = *sum-list.rev*

lemma (**in** *monoid-add*) *fold-plus-sum-list-rev*:
fold plus xs = *plus* (*sum-list* (*rev xs*))
proof
fix *x*
have *fold plus xs x* = *sum-list* (*rev xs* @ [*x*])
by (*simp add*: *foldr-conv-fold sum-list.eq-foldr*)
also have ... = *sum-list* (*rev xs*) + *x*
by *simp*
finally show *fold plus xs x* = *sum-list* (*rev xs*) + *x*
.

qed

lemma *sum-list-of-nat*: *sum-list* (*map of-nat xs*) = *of-nat* (*sum-list xs*)
by (*induction xs*) *auto*

lemma *sum-list-of-int*: *sum-list* (*map of-int xs*) = *of-int* (*sum-list xs*)
by (*induction xs*) *auto*

lemma *count-list-concat*: *count-list* (*concat xss*) *x* = *sum-list* (*map* (λ*xs*. *count-list xs x*) *xss*)
by(*induction xss*) *auto*

lemma (**in** *comm-monoid-add*) *sum-list-map-remove1*:
x ∈ *set xs* ⟹ *sum-list* (*map f xs*) = *f x* + *sum-list* (*map f* (*remove1 x xs*))
by (*induct xs*) (*auto simp add*: *ac-simps*)

lemma (**in** *monoid-add*) *size-list-conv-sum-list*:
size-list f xs = *sum-list* (*map f xs*) + *size xs*
by (*induct xs*) *auto*

lemma (**in** *monoid-add*) *length-concat*:
length (*concat xss*) = *sum-list* (*map length xss*)
by (*induct xss*) *simp-all*

lemma (**in** *monoid-add*) *length-product-lists*:
length (*product-lists xss*) = *foldr* (∗) (*map length xss*) *1*
proof (*induct xss*)
case (*Cons xs xss*) **then show** *?case* **by** (*induct xs*) (*auto simp*: *length-concat o-def*)
qed *simp*

lemma (**in** *monoid-add*) *sum-list-map-filter*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ ¬ *P x* ⟹ *f x* = *0*
shows *sum-list* (*map f* (*filter P xs*)) = *sum-list* (*map f xs*)
using *assms* **by** (*induct xs*) *auto*

lemma *sum-list-filter-le-nat*:
fixes *f* :: *'a* ⇒ *nat*
shows *sum-list* (*map f* (*filter P xs*)) ≤ *sum-list* (*map f xs*)
by(*induction xs*; *simp*)

lemma (**in** *comm-monoid-add*) *distinct-sum-list-conv-Sum*:
distinct xs ⟹ *sum-list xs* = *Sum* (*set xs*)
by (*metis local.sum.set-conv-list local.sum-list-def map-ident remdups-id-iff-distinct*)

lemma *sum-list-upt*[*simp*]:
m ≤ *n* ⟹ *sum-list* [*m*..<*n*] = ∑ {*m*..<*n*}
by(*simp add*: *distinct-sum-list-conv-Sum*)

context *ordered-comm-monoid-add*
begin

lemma *sum-list-nonneg*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow 0 \leq x) \Longrightarrow 0 \leq sum\text{-}list\ xs$
by (*induction xs*) *auto*

lemma *sum-list-nonpos*: $(\bigwedge x.\ x \in set\ xs \Longrightarrow x \leq 0) \Longrightarrow sum\text{-}list\ xs \leq 0$
by (*induction xs*) (*auto simp*: *add-nonpos-nonpos*)

lemma *sum-list-nonneg-eq-0-iff*:
$(\bigwedge x.\ x \in set\ xs \Longrightarrow 0 \leq x) \Longrightarrow sum\text{-}list\ xs = 0 \longleftrightarrow (\forall x{\in}\ set\ xs.\ x = 0)$
by (*induction xs*) (*simp-all add*: *add-nonneg-eq-0-iff sum-list-nonneg*)

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-list-eq-0-iff* [*simp*]:
$sum\text{-}list\ ns = 0 \longleftrightarrow (\forall n \in set\ ns.\ n = 0)$
by (*simp add*: *sum-list-nonneg-eq-0-iff*)

lemma *member-le-sum-list*:
$x \in set\ xs \Longrightarrow x \leq sum\text{-}list\ xs$
by (*induction xs*) (*auto simp*: *add-increasing add-increasing2*)

lemma *elem-le-sum-list*:
$k < size\ ns \Longrightarrow ns\ !\ k \leq sum\text{-}list\ (ns)$
by (*simp add*: *member-le-sum-list*)

end

lemma (**in** *ordered-cancel-comm-monoid-diff*) *sum-list-update*:
$k < size\ xs \Longrightarrow sum\text{-}list\ (xs[k := x]) = sum\text{-}list\ xs + x - xs\ !\ k$
proof (*induction xs arbitrary*:*k*)
case *Nil*
then show *?case* **by** *auto*
next
case (*Cons a xs*)
then show *?case*
apply (*simp add*: *add-ac split*: *nat.split*)
using *add-increasing diff-add-assoc elem-le-sum-list zero-le* **by** *force*
qed

lemma (**in** *monoid-add*) *sum-list-triv*:
$(\sum x{\leftarrow}xs.\ r) = of\text{-}nat\ (length\ xs) * r$
by (*induct xs*) (*simp-all add*: *distrib-right*)

lemma (**in** *monoid-add*) *sum-list-0* [*simp*]:

$(\sum x{\leftarrow}xs.\ 0) = 0$
by (*induct xs*) (*simp-all add*: *distrib-right*)

For non-Abelian groups *xs* needs to be reversed on one side:

lemma (**in** *ab-group-add*) *uminus-sum-list-map*:
$-$ *sum-list* (*map f xs*) = *sum-list* (*map* (*uminus* $\circ$ *f*) *xs*)
by (*induct xs*) *simp-all*

lemma (**in** *comm-monoid-add*) *sum-list-addf*:
$(\sum x{\leftarrow}xs.\ f\ x + g\ x)$ = *sum-list* (*map f xs*) + *sum-list* (*map g xs*)
by (*induct xs*) (*simp-all add*: *algebra-simps*)

lemma (**in** *ab-group-add*) *sum-list-subtractf*:
$(\sum x{\leftarrow}xs.\ f\ x - g\ x)$ = *sum-list* (*map f xs*) $-$ *sum-list* (*map g xs*)
by (*induct xs*) (*simp-all add*: *algebra-simps*)

lemma (**in** *semiring-0*) *sum-list-const-mult*:
$(\sum x{\leftarrow}xs.\ c * f\ x) = c * (\sum x{\leftarrow}xs.\ f\ x)$
by (*induct xs*) (*simp-all add*: *algebra-simps*)

lemma (**in** *semiring-0*) *sum-list-mult-const*:
$(\sum x{\leftarrow}xs.\ f\ x * c) = (\sum x{\leftarrow}xs.\ f\ x) * c$
by (*induct xs*) (*simp-all add*: *algebra-simps*)

lemma (**in** *ordered-ab-group-add-abs*) *sum-list-abs*:
$|$*sum-list xs*$| \leq$ *sum-list* (*map abs xs*)
by (*induct xs*) (*simp-all add*: *order-trans* [*OF abs-triangle-ineq*])

lemma *sum-list-mono*:
fixes *f g* :: *'a* $\Rightarrow$ *'b*::{*monoid-add*, *ordered-ab-semigroup-add*}
shows $(\bigwedge x.\ x \in set\ xs \Longrightarrow f\ x \leq g\ x) \Longrightarrow (\sum x{\leftarrow}xs.\ f\ x) \leq (\sum x{\leftarrow}xs.\ g\ x)$
by (*induct xs*) (*simp*, *simp add*: *add-mono*)

lemma *sum-list-strict-mono*:
fixes *f g* :: *'a* $\Rightarrow$ *'b*::{*monoid-add*, *strict-ordered-ab-semigroup-add*}
shows $[\![\ xs \neq [];\ \bigwedge x.\ x \in set\ xs \Longrightarrow f\ x < g\ x\ ]\!]$
$\Longrightarrow$ *sum-list* (*map f xs*) $<$ *sum-list* (*map g xs*)
proof (*induction xs*)
case *Nil* **thus** *?case* **by** *simp*
next
case *C*: (*Cons - xs*)
show *?case*
proof (*cases xs*)
case *Nil* **thus** *?thesis* **using** *C.prems* **by** *simp*
next
case *Cons* **thus** *?thesis* **using** *C* **by**(*simp add*: *add-strict-mono*)
qed
qed

A much more general version of this monotonicity lemma can be formulated

with multisets and the multiset order

lemma *sum-list-mono2*: **fixes** *xs* :: *'a* ::*ordered-comm-monoid-add list*
shows ⟦ *length xs* = *length ys*; ⋀*i*. *i* < *length xs* ⟶ *xs*!*i* ≤ *ys*!*i* ⟧
⟹ *sum-list xs* ≤ *sum-list ys*
by (*induction xs ys rule*: *list-induct2*) (*auto simp*: *nth-Cons' less-Suc-eq-0-disj imp-ex add-mono*)

lemma (**in** *monoid-add*) *sum-list-distinct-conv-sum-set*:
distinct xs ⟹ *sum-list* (*map f xs*) = *sum f* (*set xs*)
by (*induct xs*) *simp-all*

lemma (**in** *monoid-add*) *interv-sum-list-conv-sum-set-nat*:
sum-list (*map f* [*m*..<*n*]) = *sum f* (*set* [*m*..<*n*])
by (*simp add*: *sum-list-distinct-conv-sum-set*)

lemma (**in** *monoid-add*) *interv-sum-list-conv-sum-set-int*:
sum-list (*map f* [*k*..*l*]) = *sum f* (*set* [*k*..*l*])
by (*simp add*: *sum-list-distinct-conv-sum-set*)

General equivalence between *sum-list* and *sum*

lemma (**in** *monoid-add*) *sum-list-sum-nth*:
sum-list xs = (∑ *i* = *0* ..< *length xs*. *xs* ! *i*)
using *interv-sum-list-conv-sum-set-nat* [*of* (!) *xs 0 length xs*] **by** (*simp add*: *map-nth*)

lemma *sum-list-map-eq-sum-count*:
sum-list (*map f xs*) = *sum* (λ*x*. *count-list xs x* ∗ *f x*) (*set xs*)
proof(*induction xs*)
case (*Cons x xs*)
show *?case* (**is** *?l* = *?r*)
proof *cases*
assume *x* ∈ *set xs*
have *?l* = *f x* + (∑*x*∈*set xs*. *count-list xs x* ∗ *f x*) **by** (*simp add*: *Cons.IH*)
also have *set xs* = *insert x* (*set xs* − {*x*}) **using** ‹*x* ∈ *set xs*›**by** *blast*
also have *f x* + (∑*x*∈*insert x* (*set xs* − {*x*}). *count-list xs x* ∗ *f x*) = *?r*
by (*simp add*: *sum.insert-remove eq-commute*)
finally show *?thesis* **.**
next
assume *x* ∉ *set xs*
hence ⋀*xa*. *xa* ∈ *set xs* ⟹ *x* ≠ *xa* **by** *blast*
thus *?thesis* **by** (*simp add*: *Cons.IH* ‹*x* ∉ *set xs*›)
qed
qed *simp*

lemma *sum-list-map-eq-sum-count2*:
assumes *set xs* ⊆ *X finite X*
shows *sum-list* (*map f xs*) = *sum* (λ*x*. *count-list xs x* ∗ *f x*) *X*
proof−
let *?F* = λ*x*. *count-list xs x* ∗ *f x*

have *sum ?F X = sum ?F* (*set xs* ∪ (*X* − *set xs*))
using *Un-absorb1*[*OF assms*(*1*)] **by**(*simp*)
also have ... = *sum ?F* (*set xs*)
using *assms*(*2*)
by(*simp add*: *sum.union-disjoint*[*OF - - Diff-disjoint*] *del*: *Un-Diff-cancel*)
finally show *?thesis* **by**(*simp add*:*sum-list-map-eq-sum-count*)
qed

lemma *sum-list-replicate*: *sum-list* (*replicate n c*) = *of-nat n* ∗ *c*
by(*induction n*)(*auto simp add*: *distrib-right*)

lemma *sum-list-nonneg*:
(⋀*x*. *x* ∈ *set xs* ⟹ (*x* :: *'a* :: *ordered-comm-monoid-add*) ≥ *0*) ⟹ *sum-list xs* ≥ *0*
by (*induction xs*) *simp-all*

lemma *sum-list-Suc*:
sum-list (*map* (λ*x*. *Suc*(*f x*)) *xs*) = *sum-list* (*map f xs*) + *length xs*
by(*induction xs*; *simp*)

lemma (**in** *monoid-add*) *sum-list-map-filter'*:
sum-list (*map f* (*filter P xs*)) = *sum-list* (*map* (λ*x*. *if P x then f x else 0*) *xs*)
by (*induction xs*) *simp-all*

Summation of a strictly ascending sequence with length n can be upper-bounded by summation over $\{0..<n\}$.

lemma *sorted-wrt-less-sum-mono-lowerbound*:
fixes *f* :: *nat* ⇒ (*'b*::*ordered-comm-monoid-add*)
assumes *mono*: ⋀*x y*. *x*≤*y* ⟹ *f x* ≤ *f y*
shows *sorted-wrt* (<) *ns* ⟹
(∑ *i*∈{*0*..<*length ns*}. *f i*) ≤ (∑ *i*←*ns*. *f i*)
proof (*induction ns rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *simp*
next
case (*snoc n ns*)
have *sum f* {*0*..<*length* (*ns* @ [*n*])}
= *sum f* {*0*..<*length ns*} + *f* (*length ns*)
by *simp*
also have *sum f* {*0*..<*length ns*} ≤ *sum-list* (*map f ns*)
using *snoc* **by** (*auto simp*: *sorted-wrt-append*)
also have *length ns* ≤ *n*
using *sorted-wrt-less-idx*[*OF snoc.prems*(*1*), *of length ns*] **by** *auto*
finally have *sum f* {*0*..<*length* (*ns* @ [*n*])} ≤ *sum-list* (*map f ns*) + *f n*
using *mono add-mono* **by** *blast*
thus *?case* **by** *simp*
qed

lemma *member-le-sum-list*:
fixes $x :: 'a :: ordered\text{-}comm\text{-}monoid\text{-}add$
assumes $x \in set\ xs\ \bigwedge x.\ x \in set\ xs \Longrightarrow x \geq 0$
shows $x \leq sum\text{-}list\ xs$
using *assms*
proof (*induction xs*)
case (*Cons y xs*)
show *?case*
proof (*cases* $y = x$)
case *True*
have $x + 0 \leq x + sum\text{-}list\ xs$
by (*intro add-mono order.refl sum-list-nonneg*) (*use Cons* **in** *auto*)
thus *?thesis*
using *True* **by** *auto*
next
case *False*
have $0 + x \leq y + sum\text{-}list\ xs$
by (*intro add-mono Cons.IH Cons.prems*) (*use Cons.prems False* **in** *auto*)
thus *?thesis*
by *auto*
qed
qed *auto*

67.2 Horner sums

context *comm-semiring-0*
begin

definition $horner\text{-}sum :: \langle('b \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'b\ list \Rightarrow 'a\rangle$
where *horner-sum-foldr*: $\langle horner\text{-}sum\ f\ a\ xs = foldr\ (\lambda x\ b.\ f\ x + a * b)\ xs\ 0\rangle$

lemma *horner-sum-simps* [*simp*]:
$\langle horner\text{-}sum\ f\ a\ [] = 0\rangle$
$\langle horner\text{-}sum\ f\ a\ (x \# xs) = f\ x + a * horner\text{-}sum\ f\ a\ xs\rangle$
by (*simp-all add*: *horner-sum-foldr*)

lemma *horner-sum-eq-sum-funpow*:
$\langle horner\text{-}sum\ f\ a\ xs = (\sum n = 0..<length\ xs.\ ((*)\ a\ \widehat{\ }\widehat{\ }\ n)\ (f\ (xs\ !\ n)))\rangle$
proof (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons x xs*)
then show *?case*
by (*simp add*: *sum.atLeast0-lessThan-Suc-shift sum-distrib-left del*: *sum.op-ivl-Suc*)
qed

end

context
includes *lifting-syntax*
begin

lemma *horner-sum-transfer* [*transfer-rule*]:
‹((*B* ===> *A*) ===> *A* ===> *list-all2 B* ===> *A*) *horner-sum horner-sum*›
if [*transfer-rule*]: ‹*A 0 0*›
and [*transfer-rule*]: ‹(*A* ===> *A* ===> *A*) (+) (+)›
and [*transfer-rule*]: ‹(*A* ===> *A* ===> *A*) (∗) (∗)›
by (*unfold horner-sum-foldr*) *transfer-prover*

end

context *comm-semiring-1*
begin

lemma *horner-sum-eq-sum*:
‹*horner-sum f a xs* = (∑ *n* = *0*..<*length xs*. *f* (*xs* ! *n*) ∗ *a* ^ *n*)›
proof −
have ‹(∗) *a* ^^ *n* = (∗) (*a* ^ *n*)› **for** *n*
by (*induction n*) (*simp-all add*: *ac-simps*)
then show *?thesis*
by (*simp add*: *horner-sum-eq-sum-funpow ac-simps*)
qed

lemma *horner-sum-append*:
‹*horner-sum f a* (*xs* @ *ys*) = *horner-sum f a xs* + *a* ^ *length xs* ∗ *horner-sum f a ys*›
using *sum.atLeastLessThan-shift-bounds* [*of - 0* ‹*length xs*› ‹*length ys*›]
atLeastLessThan-add-Un [*of 0* ‹*length xs*› ‹*length ys*›]
by (*simp add*: *horner-sum-eq-sum sum-distrib-left sum.union-disjoint ac-simps nth-append power-add*)

end

context *linordered-semidom*
begin

lemma *horner-sum-nonnegative*:
‹*0* ≤ *horner-sum of-bool 2 bs*›
by (*induction bs*) *simp-all*

end

context *discrete-linordered-semidom*
begin

lemma *horner-sum-bound*:
‹*horner-sum of-bool 2 bs* < *2* ^ *length bs*›
proof (*induction bs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons b bs*)
moreover define *a* **where** ‹*a* = *2* ^ *length bs* − *horner-sum of-bool 2 bs*›
ultimately have ∗: ‹*2* ^ *length bs* = *horner-sum of-bool 2 bs* + *a*›
by *simp*
have ‹*0* < *a*›
using *Cons* ∗ **by** *simp*
moreover have ‹*1* ≤ *a*›
using ‹*0* < *a*› **by** (*simp add*: *less-eq-iff-succ-less*)
ultimately have ‹*0* + *1* < *a* + *a*›
by (*rule add-less-le-mono*)
then have ‹*1* < *a* ∗ *2*›
by (*simp add*: *mult-2-right*)
with *Cons* **show** *?case*
by (*simp add*: ∗ *algebra-simps*)
qed

lemma *horner-sum-of-bool-2-less*:
‹(*horner-sum of-bool 2 bs*) < *2* ^ *length bs*›
by (*fact horner-sum-bound*)

end

lemma *nat-horner-sum* [*simp*]:
‹*nat* (*horner-sum of-bool 2 bs*) = *horner-sum of-bool 2 bs*›
by (*induction bs*) (*auto simp add*: *nat-add-distrib horner-sum-nonnegative*)

context *discrete-linordered-semidom*
begin

lemma *horner-sum-less-eq-iff-lexordp-eq*:
‹*horner-sum of-bool 2 bs* ≤ *horner-sum of-bool 2 cs* ⟷ *lexordp-eq* (*rev bs*) (*rev cs*)›
if ‹*length bs* = *length cs*›
proof −
have ‹*horner-sum of-bool 2* (*rev bs*) ≤ *horner-sum of-bool 2* (*rev cs*) ⟷ *lexordp-eq bs cs*›
if ‹*length bs* = *length cs*› **for** *bs cs*
using *that* **proof** (*induction bs cs rule*: *list-induct2*)
case *Nil*
then show *?case*
by *simp*
next

```
    case (Cons b bs c cs)
   with horner-sum-nonnegative [of ‹rev bs›] horner-sum-nonnegative [of ‹rev cs›]
     horner-sum-bound [of ‹rev bs›] horner-sum-bound [of ‹rev cs›]
    show ?case
    by (auto simp add: horner-sum-append not-le Cons intro: add-strict-increasing2
add-increasing)
  qed
  from that this [of ‹rev bs› ‹rev cs›] show ?thesis
    by simp
qed

lemma horner-sum-less-iff-lexordp:
  ‹horner-sum of-bool 2 bs < horner-sum of-bool 2 cs ⟷ ord-class.lexordp (rev bs)
(rev cs)›
  if ‹length bs = length cs›
proof −
  have ‹horner-sum of-bool 2 (rev bs) < horner-sum of-bool 2 (rev cs) ⟷ ord-class.lexordp
bs cs›
    if ‹length bs = length cs› for bs cs
  using that proof (induction bs cs rule: list-induct2)
    case Nil
    then show ?case
      by simp
  next
    case (Cons b bs c cs)
   with horner-sum-nonnegative [of ‹rev bs›] horner-sum-nonnegative [of ‹rev cs›]
     horner-sum-bound [of ‹rev bs›] horner-sum-bound [of ‹rev cs›]
    show ?case
    by (auto simp add: horner-sum-append not-less Cons intro: add-strict-increasing2
add-increasing)
  qed
  from that this [of ‹rev bs› ‹rev cs›] show ?thesis
    by simp
qed

end
```

67.3 Further facts about *List.n-lists*

```
lemma length-n-lists: length (List.n-lists n xs) = length xs ^ n
  by (induct n) (auto simp add: comp-def length-concat sum-list-triv)

lemma distinct-n-lists:
  assumes distinct xs
  shows distinct (List.n-lists n xs)
proof (rule card-distinct)
  from assms have card-length: card (set xs) = length xs by (rule distinct-card)
  have card (set (List.n-lists n xs)) = card (set xs) ^ n
  proof (induct n)
```

```
    case 0 then show ?case by simp
  next
    case (Suc n)
    moreover have card (⋃ys∈set (List.n-lists n xs). (λy. y # ys) ` set xs)
      = (∑ ys∈set (List.n-lists n xs). card ((λy. y # ys) ` set xs))
      by (rule card-UN-disjoint) auto
    moreover have ⋀ys. card ((λy. y # ys) ` set xs) = card (set xs)
      by (rule card-image) (simp add: inj-on-def)
    ultimately show ?case by auto
  qed
  also have ... = length xs ^ n by (simp add: card-length)
  finally show card (set (List.n-lists n xs)) = length (List.n-lists n xs)
    by (simp add: length-n-lists)
qed
```

67.4 Tools setup

```
lemmas sum-code = sum.set-conv-list

lemma sum-set-upto-conv-sum-list-int:
  sum f (set [i..j::int]) = sum-list (map f [i..j])
  by (simp add: interv-sum-list-conv-sum-set-int)

lemma sum-set-upt-conv-sum-list-nat:
  sum f (set [m..<n]) = sum-list (map f [m..<n])
  by (simp add: interv-sum-list-conv-sum-set-nat)
```

67.5 List product

```
context monoid-mult
begin

sublocale prod-list: monoid-list times 1
defines
  prod-list = prod-list.F ..

end

context comm-monoid-mult
begin

sublocale prod-list: comm-monoid-list times 1
rewrites
  monoid-list.F times 1 = prod-list
proof −
  show comm-monoid-list times 1 ..
  then interpret prod-list: comm-monoid-list times 1 .
  from prod-list-def show monoid-list.F times 1 = prod-list by simp
qed
```

sublocale *prod*: *comm-monoid-list-set times 1*
rewrites
monoid-list.F times 1 = prod-list
and *comm-monoid-set.F times 1 = prod*
proof −
show *comm-monoid-list-set times 1* **..**
then interpret *prod*: *comm-monoid-list-set times 1* **.**
from *prod-list-def* **show** *monoid-list.F times 1 = prod-list* **by** *simp*
from *prod-def* **show** *comm-monoid-set.F times 1 = prod* **by** (*auto intro*: *sym*)
qed

end

Some syntactic sugar:

open-bundle *prod-list-syntax*
begin

syntax (*ASCII*)
-prod-list :: *pttrn => 'a list => 'b => 'b* (‹(‹*indent=3 notation=‹binder PROD››PROD -<−-. -)*› [*0*, *51*, *10*] *10*)
syntax
-prod-list :: *pttrn => 'a list => 'b => 'b* (‹(‹*indent=3 notation=‹binder* ∏ *››*∏ *-←-. -)*› [*0*, *51*, *10*] *10*)
syntax-consts
-prod-list ⇌ *prod-list*
translations — Beware of argument permutation!
∏ *x←xs. b* ⇌ *CONST prod-list* (*CONST map* (λ*x. b*) *xs*)

end

context
includes *lifting-syntax*
begin

lemma *prod-list-transfer* [*transfer-rule*]:
(*list-all2 A ===> A*) *prod-list prod-list*
if [*transfer-rule*]: *A 1 1* (*A ===> A ===> A*) (∗) (∗)
unfolding *prod-list.eq-foldr* [*abs-def*]
by *transfer-prover*

end

lemma *prod-list-zero-iff*:
prod-list xs = 0 ⟷ (*0* :: *'a* :: {*semiring-no-zero-divisors*, *semiring-1*}) ∈ *set xs*
by (*induction xs*) *simp-all*

lemma *prod-list-nonneg*: (⋀ *x*. (*x* :: *'a* :: *ordered-semiring-1*) ∈ *set xs* ⟹ *x* ≥ *0*)
⟹ *prod-list xs* ≥ *0*
by (*induct xs*) *auto*

lemma *prod-list-replicate*[*simp*]: *prod-list* (*replicate n a*) = *a* ^ *n*
by (*induct n*) *auto*

lemma *prod-list-power*:
fixes *xs* :: *'a* :: *comm-monoid-mult list*
shows *prod-list xs* ^ *n* = (∏ *x*←*xs*. *x* ^ *n*)
by (*induct xs*, *auto simp*: *power-mult-distrib*)

lemma *prod-list-dvd*:
assumes (*x* :: *'a* :: *comm-monoid-mult*) ∈ *set xs*
shows *x dvd prod-list xs*
by (*metis assms dvd-mult dvd-triv-left in-set-conv-decomp prod-list.Cons prod-list.append*)

end

68 Bit operations in suitable algebraic structures

theory *Bit-Operations*
imports *Presburger Groups-List*
begin

68.1 Abstract bit structures

class *semiring-bits* = *semiring-parity* + *semiring-modulo-trivial* +
assumes *bit-induct* [*case-names stable rec*]:
‹(⋀*a*. *a div 2* = *a* ⟹ *P a*)
⟹ (⋀*a b*. *P a* ⟹ (*of-bool b* + *2* ∗ *a*) *div 2* = *a* ⟹ *P* (*of-bool b* + *2* ∗ *a*))
⟹ *P a*›
assumes *bits-mod-div-trivial* [*simp*]: ‹*a mod b div b* = *0*›
and *half-div-exp-eq*: ‹*a div 2 div 2* ^ *n* = *a div 2* ^ *Suc n*›
and *even-double-div-exp-iff*: ‹*2* ^ *Suc n* ≠ *0* ⟹ *even* (*2* ∗ *a div 2* ^ *Suc n*)
⟷ *even* (*a div 2* ^ *n*)›
fixes *bit* :: ‹*'a* ⇒ *nat* ⇒ *bool*›
assumes *bit-iff-odd*: ‹*bit a n* ⟷ *odd* (*a div 2* ^ *n*)›
begin

Having *bit* as definitional class operation takes into account that specific instances can be implemented differently wrt. code generation.

lemma *half-1* [*simp*]:
‹*1 div 2* = *0*›
using *even-half-succ-eq* [*of 0*] **by** *simp*

lemma *div-exp-eq-funpow-half*:
‹*a div 2* ^ *n* = ((λ*a*. *a div 2*) ^^ *n*) *a*›
proof −
have ‹((λ*a*. *a div 2*) ^^ *n*) = (λ*a*. *a div 2* ^ *n*)›
by (*induction n*)
(*simp-all del*: *funpow.simps power.simps add*: *power-0 funpow-Suc-right half-div-exp-eq*)

then show *?thesis*
by *simp*
qed

lemma *div-exp-eq*:
‹*a div 2 ^ m div 2 ^ n = a div 2 ^ (m + n)*›
by (*simp add*: *div-exp-eq-funpow-half Groups.add.commute* [*of m*] *funpow-add*)

lemma *bit-0*:
‹*bit a 0 ⟷ odd a*›
by (*simp add*: *bit-iff-odd*)

lemma *bit-Suc*:
‹*bit a* (*Suc n*) ⟷ *bit* (*a div 2*) *n*›
using *div-exp-eq* [*of a 1 n*] **by** (*simp add*: *bit-iff-odd*)

lemma *bit-rec*:
‹*bit a n ⟷* (*if n = 0 then odd a else bit* (*a div 2*) (*n − 1*))›
by (*cases n*) (*simp-all add*: *bit-Suc bit-0*)

context
fixes *a*
assumes *stable*: ‹*a div 2 = a*›
begin

lemma *bits-stable-imp-add-self*:
‹*a + a mod 2 = 0*›
proof −
have ‹*a div 2 * 2 + a mod 2 = a*›
by (*fact div-mult-mod-eq*)
then have ‹*a * 2 + a mod 2 = a*›
by (*simp add*: *stable*)
then show *?thesis*
by (*simp add*: *mult-2-right ac-simps*)
qed

lemma *stable-imp-bit-iff-odd*:
‹*bit a n ⟷ odd a*›
by (*induction n*) (*simp-all add*: *stable bit-Suc bit-0*)

end

lemma *bit-iff-odd-imp-stable*:
‹*a div 2 = a*› **if** ‹⋀*n*. *bit a n ⟷ odd a*›
using *that* **proof** (*induction a rule*: *bit-induct*)
case (*stable a*)
then show *?case*
by *simp*
next

```
  case (rec a b)
  from rec.prems [of 1] have [simp]: ‹b = odd a›
    by (simp add: rec.hyps bit-Suc bit-0)
  from rec.hyps have hyp: ‹(of-bool (odd a) + 2 * a) div 2 = a›
    by simp
  have ‹bit a n ⟷ odd a› for n
    using rec.prems [of ‹Suc n›] by (simp add: hyp bit-Suc)
  then have ‹a div 2 = a›
    by (rule rec.IH)
  then have ‹of-bool (odd a) + 2 * a = 2 * (a div 2) + of-bool (odd a)›
    by (simp add: ac-simps)
  also have ‹... = a›
    using mult-div-mod-eq [of 2 a]
    by (simp add: of-bool-odd-eq-mod-2)
  finally show ?case
    using ‹a div 2 = a› by (simp add: hyp)
qed

lemma even-succ-div-exp [simp]:
  ‹(1 + a) div 2 ^ n = a div 2 ^ n› if ‹even a› and ‹n > 0›
proof (cases n)
  case 0
  with that show ?thesis
    by simp
next
  case (Suc n)
  with ‹even a› have ‹(1 + a) div 2 ^ Suc n = a div 2 ^ Suc n›
  proof (induction n)
    case 0
    then show ?case
      by simp
  next
    case (Suc n)
    then show ?case
      using div-exp-eq [of - 1 ‹Suc n›, symmetric]
      by simp
  qed
  with Suc show ?thesis
    by simp
qed

lemma even-succ-mod-exp [simp]:
  ‹(1 + a) mod 2 ^ n = 1 + (a mod 2 ^ n)› if ‹even a› and ‹n > 0›
  using div-mult-mod-eq [of ‹1 + a› ‹2 ^ n›] div-mult-mod-eq [of a ‹2 ^ n›] that
  by simp (metis (full-types) add.left-commute add-left-imp-eq)

lemma half-numeral-Bit1-eq [simp]:
  ‹numeral (num.Bit1 m) div 2 = numeral (num.Bit0 m) div 2›
  using even-half-succ-eq [of ‹2 * numeral m›]
```

by *simp*

lemma *double-half-numeral-Bit-0-eq* [*simp*]:
‹*2 * (numeral (num.Bit0 m) div 2) = numeral (num.Bit0 m)*›
‹*(numeral (num.Bit0 m) div 2) * 2 = numeral (num.Bit0 m)*›
using *mod-mult-div-eq* [*of* ‹*numeral (Num.Bit0 m)*› *2*]
by (*simp-all add*: *mod2-eq-if ac-simps*)

named-theorems *bit-simps* ‹*Simplification rules for* ***const*** ‹*bit*››

definition *possible-bit* :: ‹*'a itself ⇒ nat ⇒ bool*›
where ‹*possible-bit TYPE('a) n ⟷ 2 ^ n ≠ 0*›
— This auxiliary avoids non-termination with extensionality.

lemma *possible-bit-0* [*simp*]:
‹*possible-bit TYPE('a) 0*›
by (*simp add*: *possible-bit-def*)

lemma *fold-possible-bit*:
‹*2 ^ n = 0 ⟷ ¬ possible-bit TYPE('a) n*›
by (*simp add*: *possible-bit-def*)

lemma *bit-imp-possible-bit*:
‹*possible-bit TYPE('a) n*› **if** ‹*bit a n*›
by (*rule ccontr*) (*use that* **in** ‹*auto simp*: *bit-iff-odd possible-bit-def*›)

lemma *impossible-bit*:
‹*¬ bit a n*› **if** ‹*¬ possible-bit TYPE('a) n*›
using *that* **by** (*blast dest*: *bit-imp-possible-bit*)

lemma *possible-bit-less-imp*:
‹*possible-bit TYPE('a) j*› **if** ‹*possible-bit TYPE('a) i*› ‹*j ≤ i*›
using *power-add* [*of 2 j* ‹*i − j*›] *that mult-not-zero* [*of* ‹*2 ^ j*› ‹*2 ^ (i − j)*›]
by (*simp add*: *possible-bit-def*)

lemma *possible-bit-min* [*simp*]:
‹*possible-bit TYPE('a) (min i j) ⟷ possible-bit TYPE('a) i ∨ possible-bit TYPE('a) j*›
by (*auto simp*: *min-def elim*: *possible-bit-less-imp*)

lemma *bit-eqI*:
‹*a = b*› **if** ‹*⋀n. possible-bit TYPE('a) n ⟹ bit a n ⟷ bit b n*›
proof −
have ‹*bit a n ⟷ bit b n*› **for** *n*
proof (*cases* ‹*possible-bit TYPE('a) n*›)
case *False*
then show *?thesis*
by (*simp add*: *impossible-bit*)
next

```
    case True
    then show ?thesis
      by (rule that)
  qed
  then show ?thesis proof (induction a arbitrary: b rule: bit-induct)
    case (stable a)
    from stable(2) [of 0] have **: ‹even b ⟷ even a›
      by (simp add: bit-0)
    have ‹b div 2 = b›
    proof (rule bit-iff-odd-imp-stable)
      fix n
      from stable have *: ‹bit b n ⟷ bit a n›
        by simp
      also have ‹bit a n ⟷ odd a›
        using stable by (simp add: stable-imp-bit-iff-odd)
      finally show ‹bit b n ⟷ odd b›
        by (simp add: **)
    qed
    from ** have ‹a mod 2 = b mod 2›
      by (simp add: mod2-eq-if)
    then have ‹a mod 2 + (a + b) = b mod 2 + (a + b)›
      by simp
    then have ‹a + a mod 2 + b = b + b mod 2 + a›
      by (simp add: ac-simps)
    with ‹a div 2 = a› ‹b div 2 = b› show ?case
      by (simp add: bits-stable-imp-add-self)
  next
    case (rec a p)
    from rec.prems [of 0] have [simp]: ‹p = odd b›
      by (simp add: bit-0)
    from rec.hyps have ‹bit a n ⟷ bit (b div 2) n› for n
      using rec.prems [of ‹Suc n›] by (simp add: bit-Suc)
    then have ‹a = b div 2›
      by (rule rec.IH)
    then have ‹2 * a = 2 * (b div 2)›
      by simp
    then have ‹b mod 2 + 2 * a = b mod 2 + 2 * (b div 2)›
      by simp
    also have ‹... = b›
      by (fact mod-mult-div-eq)
    finally show ?case
      by (auto simp: mod2-eq-if)
  qed
qed

lemma bit-eq-rec:
  ‹a = b ⟷ (even a ⟷ even b) ∧ a div 2 = b div 2› (is ‹?P = ?Q›)
proof
  assume ?P
```

```
  then show ?Q
    by simp
next
  assume ?Q
  then have ‹even a ⟷ even b› and ‹a div 2 = b div 2›
    by simp-all
  show ?P
  proof (rule bit-eqI)
    fix n
    show ‹bit a n ⟷ bit b n›
    proof (cases n)
      case 0
      with ‹even a ⟷ even b› show ?thesis
        by (simp add: bit-0)
    next
      case (Suc n)
      moreover from ‹a div 2 = b div 2› have ‹bit (a div 2) n = bit (b div 2) n›
        by simp
      ultimately show ?thesis
        by (simp add: bit-Suc)
    qed
  qed
qed

lemma bit-eq-iff:
  ‹a = b ⟷ (∀ n. possible-bit TYPE('a) n ⟶ bit a n ⟷ bit b n)›
  by (auto intro: bit-eqI simp add: possible-bit-def)

lemma bit-0-eq [simp]:
  ‹bit 0 = ⊥›
proof −
  have ‹0 div 2 ^ n = 0› for n
    unfolding div-exp-eq-funpow-half by (induction n) simp-all
  then show ?thesis
    by (simp add: fun-eq-iff bit-iff-odd)
qed

lemma bit-double-Suc-iff:
  ‹bit (2 * a) (Suc n) ⟷ possible-bit TYPE('a) (Suc n) ∧ bit a n›
  using even-double-div-exp-iff [of n a]
  by (cases ‹possible-bit TYPE('a) (Suc n)›)
    (auto simp: bit-iff-odd possible-bit-def)

lemma bit-double-iff [bit-simps]:
  ‹bit (2 * a) n ⟷ possible-bit TYPE('a) n ∧ n ≠ 0 ∧ bit a (n − 1)›
  by (cases n) (simp-all add: bit-0 bit-double-Suc-iff)

lemma even-bit-succ-iff:
  ‹bit (1 + a) n ⟷ bit a n ∨ n = 0› if ‹even a›
```

```
  using that by (cases ‹n = 0›) (simp-all add: bit-iff-odd)

lemma odd-bit-iff-bit-pred:
  ‹bit a n ⟷ bit (a − 1) n ∨ n = 0› if ‹odd a›
proof −
  from ‹odd a› obtain b where ‹a = 2 * b + 1› ..
  moreover have ‹bit (2 * b) n ∨ n = 0 ⟷ bit (1 + 2 * b) n›
    using even-bit-succ-iff by simp
  ultimately show ?thesis by (simp add: ac-simps)
qed

lemma bit-exp-iff [bit-simps]:
  ‹bit (2 ^ m) n ⟷ possible-bit TYPE('a) n ∧ n = m›
proof (cases ‹possible-bit TYPE('a) n›)
  case False
  then show ?thesis
    by (simp add: impossible-bit)
next
  case True
  then show ?thesis
  proof (induction n arbitrary: m)
    case 0
    show ?case
      by (simp add: bit-0)
  next
    case (Suc n)
    then have ‹possible-bit TYPE('a) n›
      by (simp add: possible-bit-less-imp)
    show ?case
    proof (cases m)
      case 0
      then show ?thesis
        by (simp add: bit-Suc)
    next
      case (Suc m)
      with Suc.IH [of m] ‹possible-bit TYPE('a) n› show ?thesis
        by (simp add: bit-double-Suc-iff)
    qed
  qed
qed

lemma bit-1-iff [bit-simps]:
  ‹bit 1 n ⟷ n = 0›
  using bit-exp-iff [of 0 n] by auto

lemma bit-2-iff [bit-simps]:
  ‹bit 2 n ⟷ possible-bit TYPE('a) 1 ∧ n = 1›
  using bit-exp-iff [of 1 n] by auto
```

```
lemma bit-of-bool-iff [bit-simps]:
  ‹bit (of-bool b) n ⟷ n = 0 ∧ b›
  by (simp add: bit-1-iff)

lemma bit-mod-2-iff [simp]:
  ‹bit (a mod 2) n ⟷ n = 0 ∧ odd a›
  by (simp add: mod-2-eq-odd bit-simps)

lemma stable-index:
  obtains m where ‹possible-bit TYPE('a) m›
    ‹⋀n. possible-bit TYPE('a) n ⟹ n ≥ m ⟹ bit a n ⟷ bit a m›
proof −
  have ‹∃m. possible-bit TYPE('a) m ∧ (∀n≥m. possible-bit TYPE('a) n ⟶ bit
a n ⟷ bit a m)›
  proof (induction a rule: bit-induct)
    case (stable a)
    show ?case
      by (rule exI [of - ‹0::nat›]) (simp add: stable-imp-bit-iff-odd stable)
  next
    case (rec a b)
    then obtain m where ‹possible-bit TYPE('a) m›
      and hyp: ‹⋀n. possible-bit TYPE('a) n ⟹ n ≥ m ⟹ bit a n ⟷ bit a m›
      by blast
    show ?case
    proof (cases ‹possible-bit TYPE('a) (Suc m)›)
      case True
      moreover have ‹bit (of-bool b + 2 * a) n ⟷ bit (of-bool b + 2 * a) (Suc
m)›
        if ‹possible-bit TYPE('a) n› ‹Suc m ≤ n› for n
        using hyp [of ‹n − 1›] possible-bit-less-imp [of n ‹n − 1›] rec.hyps that
        by (cases n) (simp-all add: bit-Suc)
      ultimately show ?thesis
        by blast
    next
      case False
      have ‹bit (of-bool b + 2 * a) n ⟷ bit (of-bool b + 2 * a) m›
        if ‹possible-bit TYPE('a) n› ‹m ≤ n› for n
      proof (cases ‹m = n›)
        case True
        then show ?thesis
          by simp
      next
        case False
        with ‹m ≤ n› have ‹m < n›
          by simp
        with ‹¬ possible-bit TYPE('a) (Suc m)›
        have ‹¬ possible-bit TYPE('a) n› using possible-bit-less-imp [of n ‹Suc m›]
          by auto
        with ‹possible-bit TYPE('a) n›
```

```
      show ?thesis
        by simp
    qed
    with ‹possible-bit TYPE('a) m› show ?thesis
      by blast
  qed
 qed
 with that show thesis
  by blast
qed


end

lemma nat-bit-induct [case-names zero even odd]:
 ‹P n› if zero: ‹P 0›
  and even: ‹⋀n. P n ⟹ n > 0 ⟹ P (2 * n)›
  and odd: ‹⋀n. P n ⟹ P (Suc (2 * n))›
proof (induction n rule: less-induct)
 case (less n)
 show ‹P n›
 proof (cases ‹n = 0›)
  case True with zero show ?thesis by simp
 next
  case False
  with less have hyp: ‹P (n div 2)› by simp
  show ?thesis
  proof (cases ‹even n›)
   case True
   then have ‹n ≠ 1›
    by auto
   with ‹n ≠ 0› have ‹n div 2 > 0›
    by simp
   with ‹even n› hyp even [of ‹n div 2›] show ?thesis
    by simp
  next
   case False
   with hyp odd [of ‹n div 2›] show ?thesis
    by simp
  qed
 qed
qed

instantiation nat :: semiring-bits
begin

definition bit-nat :: ‹nat ⇒ nat ⇒ bool›
 where ‹bit-nat m n ⟷ odd (m div 2 ^ n)›
```

instance
proof
  show ‹*P n*› **if** *stable*: ‹⋀*n*. *n div 2* = *n* ⟹ *P n*›
    and *rec*: ‹⋀*n b*. *P n* ⟹ (*of-bool b* + *2* ∗ *n*) *div 2* = *n* ⟹ *P* (*of-bool b* + *2* ∗ *n*)›
    for *P* **and** *n* :: *nat*
  proof (*induction n rule*: *nat-bit-induct*)
    case *zero*
    from *stable* [*of 0*] **show** *?case*
      by *simp*
  next
    case (*even n*)
    with *rec* [*of n False*] **show** *?case*
      by *simp*
  next
    case (*odd n*)
    with *rec* [*of n True*] **show** *?case*
      by *simp*
  qed
qed (*auto simp*: *div-mult2-eq bit-nat-def*)

end

lemma *possible-bit-nat* [*simp*]:
  ‹*possible-bit TYPE*(*nat*) *n*›
  by (*simp add*: *possible-bit-def*)

lemma *bit-Suc-0-iff* [*bit-simps*]:
  ‹*bit* (*Suc 0*) *n* ⟷ *n* = *0*›
  using *bit-1-iff* [*of n*, **where** *?'a* = *nat*] **by** *simp*

lemma *not-bit-Suc-0-Suc* [*simp*]:
  ‹¬ *bit* (*Suc 0*) (*Suc n*)›
  by (*simp add*: *bit-Suc*)

lemma *not-bit-Suc-0-numeral* [*simp*]:
  ‹¬ *bit* (*Suc 0*) (*numeral n*)›
  by (*simp add*: *numeral-eq-Suc*)

context *semiring-bits*
begin

lemma *bit-of-nat-iff* [*bit-simps*]:
  ‹*bit* (*of-nat m*) *n* ⟷ *possible-bit TYPE*(*'a*) *n* ∧ *bit m n*›
proof (*cases* ‹*possible-bit TYPE*(*'a*) *n*›)
  case *False*
  then show *?thesis*
    by (*simp add*: *impossible-bit*)
next

```
  case True
  then have ‹bit (of-nat m) n ⟷ bit m n›
  proof (induction m arbitrary: n rule: nat-bit-induct)
    case zero
    then show ?case
      by simp
  next
    case (even m)
    then show ?case
      by (cases n)
        (auto simp: bit-double-iff Bit-Operations.bit-double-iff possible-bit-def bit-0
dest: mult-not-zero)
  next
    case (odd m)
    then show ?case
      by (cases n)
       (auto simp: bit-double-iff even-bit-succ-iff possible-bit-def
         Bit-Operations.bit-Suc Bit-Operations.bit-0 dest: mult-not-zero)
  qed
  with True show ?thesis
    by simp
qed

end

lemma int-bit-induct [case-names zero minus even odd]:
  ‹P k› if zero-int: ‹P 0›
    and minus-int: ‹P (- 1)›
    and even-int: ‹⋀k. P k ⟹ k ≠ 0 ⟹ P (k * 2)›
    and odd-int: ‹⋀k. P k ⟹ k ≠ - 1 ⟹ P (1 + (k * 2))› for k :: int
proof (cases ‹k ≥ 0›)
  case True
  define n where ‹n = nat k›
  with True have ‹k = int n›
    by simp
  then show ‹P k›
  proof (induction n arbitrary: k rule: nat-bit-induct)
    case zero
    then show ?case
      by (simp add: zero-int)
  next
    case (even n)
    have ‹P (int n * 2)›
      by (rule even-int) (use even in simp-all)
    with even show ?case
      by (simp add: ac-simps)
  next
    case (odd n)
    have ‹P (1 + (int n * 2))›
```

```
    by (rule odd-int) (use odd in simp-all)
  with odd show ?case
    by (simp add: ac-simps)
 qed
next
 case False
 define n where ‹n = nat (− k − 1)›
 with False have ‹k = − int n − 1›
  by simp
 then show ‹P k›
 proof (induction n arbitrary: k rule: nat-bit-induct)
  case zero
  then show ?case
   by (simp add: minus-int)
 next
  case (even n)
  have ‹P (1 + (− int (Suc n) * 2))›
   by (rule odd-int) (use even in ‹simp-all add: algebra-simps›)
  also have ‹... = − int (2 * n) − 1›
   by (simp add: algebra-simps)
  finally show ?case
   using even.prems by simp
 next
  case (odd n)
  have ‹P (− int (Suc n) * 2)›
   by (rule even-int) (use odd in ‹simp-all add: algebra-simps›)
  also have ‹... = − int (Suc (2 * n)) − 1›
   by (simp add: algebra-simps)
  finally show ?case
   using odd.prems by simp
 qed
qed

instantiation int :: semiring-bits
begin

definition bit-int :: ‹int ⇒ nat ⇒ bool›
 where ‹bit-int k n ⟷ odd (k div 2 ^ n)›

instance
proof
 show ‹P k› if stable: ‹⋀k. k div 2 = k ⟹ P k›
  and rec: ‹⋀k b. P k ⟹ (of-bool b + 2 * k) div 2 = k ⟹ P (of-bool b + 2 *
k)›
  for P and k :: int
 proof (induction k rule: int-bit-induct)
  case zero
  from stable [of 0] show ?case
   by simp
```

```
  next
    case minus
    from stable [of ‹− 1›] show ?case
      by simp
  next
    case (even k)
    with rec [of k False] show ?case
      by (simp add: ac-simps)
  next
    case (odd k)
    with rec [of k True] show ?case
      by (simp add: ac-simps)
  qed
qed (auto simp: zdiv-zmult2-eq bit-int-def)

end

lemma possible-bit-int [simp]:
  ‹possible-bit TYPE(int) n›
  by (simp add: possible-bit-def)

lemma bit-nat-iff [bit-simps]:
  ‹bit (nat k) n ⟷ k ≥ 0 ∧ bit k n›
proof (cases ‹k ≥ 0›)
  case True
  moreover define m where ‹m = nat k›
  ultimately have ‹k = int m›
    by simp
  then show ?thesis
    by (simp add: bit-simps)
next
  case False
  then show ?thesis
    by simp
qed
```

68.2 Bit operations

```
class semiring-bit-operations = semiring-bits +
  fixes and :: ‹'a ⇒ 'a ⇒ 'a›  (infixr ‹AND› 64)
    and or :: ‹'a ⇒ 'a ⇒ 'a›  (infixr ‹OR› 59)
    and xor :: ‹'a ⇒ 'a ⇒ 'a›  (infixr ‹XOR› 59)
    and mask :: ‹nat ⇒ 'a›
    and set-bit :: ‹nat ⇒ 'a ⇒ 'a›
    and unset-bit :: ‹nat ⇒ 'a ⇒ 'a›
    and flip-bit :: ‹nat ⇒ 'a ⇒ 'a›
    and push-bit :: ‹nat ⇒ 'a ⇒ 'a›
    and drop-bit :: ‹nat ⇒ 'a ⇒ 'a›
    and take-bit :: ‹nat ⇒ 'a ⇒ 'a›
```

assumes *and-rec*: ‹*a AND b = of-bool (odd a ∧ odd b) + 2 * ((a div 2) AND (b div 2))*›
and *or-rec*: ‹*a OR b = of-bool (odd a ∨ odd b) + 2 * ((a div 2) OR (b div 2))*›
and *xor-rec*: ‹*a XOR b = of-bool (odd a ≠ odd b) + 2 * ((a div 2) XOR (b div 2))*›
and *mask-eq-exp-minus-1*: ‹*mask n = 2 ^ n - 1*›
and *set-bit-eq-or*: ‹*set-bit n a = a OR push-bit n 1*›
and *unset-bit-eq-or-xor*: ‹*unset-bit n a = (a OR push-bit n 1) XOR push-bit n 1*›
and *flip-bit-eq-xor*: ‹*flip-bit n a = a XOR push-bit n 1*›
and *push-bit-eq-mult*: ‹*push-bit n a = a * 2 ^ n*›
and *drop-bit-eq-div*: ‹*drop-bit n a = a div 2 ^ n*›
and *take-bit-eq-mod*: ‹*take-bit n a = a mod 2 ^ n*›
begin

We want the bitwise operations to bind slightly weaker than + and −.

Logically, *push-bit*, *drop-bit* and *take-bit* are just aliases; having them as separate operations makes proofs easier, otherwise proof automation would fiddle with concrete expressions $(2::'a)^n$ in a way obfuscating the basic algebraic relationships between those operations.

For the sake of code generation operations are specified as definitional class operations, taking into account that specific instances of these can be implemented differently wrt. code generation.

lemma *bit-iff-odd-drop-bit*:
‹*bit a n ⟷ odd (drop-bit n a)*›
by (*simp add*: *bit-iff-odd drop-bit-eq-div*)

lemma *even-drop-bit-iff-not-bit*:
‹*even (drop-bit n a) ⟷ ¬ bit a n*›
by (*simp add*: *bit-iff-odd-drop-bit*)

lemma *bit-and-iff* [*bit-simps*]:
‹*bit (a AND b) n ⟷ bit a n ∧ bit b n*›
proof (*induction n arbitrary*: *a b*)
case *0*
show *?case*
by (*simp add*: *bit-0 and-rec* [*of a b*] *even-bit-succ-iff*)
next
case (*Suc n*)
from *Suc* [*of* ‹*a div 2*› ‹*b div 2*›]
show *?case*
by (*simp add*: *and-rec* [*of a b*] *bit-Suc*)
(*auto simp flip*: *bit-Suc simp add*: *bit-double-iff dest*: *bit-imp-possible-bit*)
qed

lemma *bit-or-iff* [*bit-simps*]:
‹*bit (a OR b) n ⟷ bit a n ∨ bit b n*›
proof (*induction n arbitrary*: *a b*)

```
  case 0
  show ?case
    by (simp add: bit-0 or-rec [of a b] even-bit-succ-iff)
next
  case (Suc n)
  from Suc [of ‹a div 2› ‹b div 2›]
  show ?case
    by (simp add: or-rec [of a b] bit-Suc)
      (auto simp flip: bit-Suc simp add: bit-double-iff dest: bit-imp-possible-bit)
qed

lemma bit-xor-iff [bit-simps]:
  ‹bit (a XOR b) n ⟷ bit a n ≠ bit b n›
proof (induction n arbitrary: a b)
  case 0
  show ?case
    by (simp add: bit-0 xor-rec [of a b] even-bit-succ-iff)
next
  case (Suc n)
  from Suc [of ‹a div 2› ‹b div 2›]
  show ?case
    by (simp add: xor-rec [of a b] bit-Suc)
      (auto simp flip: bit-Suc simp add: bit-double-iff dest: bit-imp-possible-bit)
qed

sublocale and: semilattice ‹(AND)›
  by standard (auto simp: bit-eq-iff bit-and-iff)

sublocale or: semilattice-neutr ‹(OR)› 0
  by standard (auto simp: bit-eq-iff bit-or-iff)

sublocale xor: comm-monoid ‹(XOR)› 0
  by standard (auto simp: bit-eq-iff bit-xor-iff)

lemma even-and-iff:
  ‹even (a AND b) ⟷ even a ∨ even b›
  using bit-and-iff [of a b 0] by (auto simp: bit-0)

lemma even-or-iff:
  ‹even (a OR b) ⟷ even a ∧ even b›
  using bit-or-iff [of a b 0] by (auto simp: bit-0)

lemma even-xor-iff:
  ‹even (a XOR b) ⟷ (even a ⟷ even b)›
  using bit-xor-iff [of a b 0] by (auto simp: bit-0)

lemma zero-and-eq [simp]:
  ‹0 AND a = 0›
  by (simp add: bit-eq-iff bit-and-iff)
```

lemma *and-zero-eq* [*simp*]:
  ‹*a AND 0 = 0*›
  by (*simp add*: *bit-eq-iff bit-and-iff*)

lemma *one-and-eq*:
  ‹*1 AND a = a mod 2*›
  by (*simp add*: *bit-eq-iff bit-and-iff*) (*auto simp*: *bit-1-iff bit-0*)

lemma *and-one-eq*:
  ‹*a AND 1 = a mod 2*›
  using *one-and-eq* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *one-or-eq*:
  ‹*1 OR a = a + of-bool* (*even a*)›
  by (*simp add*: *bit-eq-iff bit-or-iff add.commute* [*of - 1*] *even-bit-succ-iff*)
    (*auto simp*: *bit-1-iff bit-0*)

lemma *or-one-eq*:
  ‹*a OR 1 = a + of-bool* (*even a*)›
  using *one-or-eq* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *one-xor-eq*:
  ‹*1 XOR a = a + of-bool* (*even a*) − *of-bool* (*odd a*)›
  by (*simp add*: *bit-eq-iff bit-xor-iff add.commute* [*of - 1*] *even-bit-succ-iff*)
    (*auto simp*: *bit-1-iff odd-bit-iff-bit-pred bit-0 elim*: *oddE*)

lemma *xor-one-eq*:
  ‹*a XOR 1 = a + of-bool* (*even a*) − *of-bool* (*odd a*)›
  using *one-xor-eq* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *xor-self-eq* [*simp*]:
  ‹*a XOR a = 0*›
  by (*rule bit-eqI*) (*simp add*: *bit-simps*)

lemma *mask-0* [*simp*]:
  ‹*mask 0 = 0*›
  by (*simp add*: *mask-eq-exp-minus-1*)

lemma *inc-mask-eq-exp*:
  ‹*mask n + 1 = 2 ^ n*›
proof (*induction n*)
  case *0*
  then show *?case*
    by *simp*
next
  case (*Suc n*)
  from *Suc.IH* [*symmetric*] **have** ‹*2 ^ Suc n = 2 * mask n + 2*›
    by (*simp add*: *algebra-simps*)

```
  also have ‹... = 2 * mask n + 1 + 1›
    by (simp add: add.assoc)
  finally have *: ‹2 ^ Suc n = 2 * mask n + 1 + 1› .
  then show ?case
    by (simp add: mask-eq-exp-minus-1)
qed

lemma mask-eq-iff-eq-exp:
  ‹mask n = a ⟷ a + 1 = 2 ^ n›
  by (auto simp flip: inc-mask-eq-exp)

lemma eq-mask-iff-eq-exp:
  ‹a = mask n ⟷ a + 1 = 2 ^ n›
  by (auto simp flip: inc-mask-eq-exp)

lemma mask-Suc-double:
  ‹mask (Suc n) = 1 OR 2 * mask n›
proof -
  have ‹mask (Suc n) + 1 = (mask n + 1) + (mask n + 1)›
    by (simp add: inc-mask-eq-exp mult-2)
  also have ‹... = (1 OR 2 * mask n) + 1›
    by (simp add: one-or-eq mult-2-right algebra-simps)
  finally show ?thesis
    by simp
qed

lemma bit-mask-iff [bit-simps]:
  ‹bit (mask m) n ⟷ possible-bit TYPE('a) n ∧ n < m›
proof (cases ‹possible-bit TYPE('a) n›)
  case False
  then show ?thesis
    by (simp add: impossible-bit)
next
  case True
  then have ‹bit (mask m) n ⟷ n < m›
  proof (induction m arbitrary: n)
    case 0
    then show ?case
      by (simp add: bit-iff-odd)
  next
    case (Suc m)
    show ?case
    proof (cases n)
      case 0
      then show ?thesis
        by (simp add: bit-0 mask-Suc-double even-or-iff)
    next
      case (Suc n)
      with Suc.prems have ‹possible-bit TYPE('a) n›
```

```
      using possible-bit-less-imp by auto
    with Suc.IH [of n] have ‹bit (mask m) n ⟷ n < m› .
    with Suc.prems show ?thesis
      by (simp add: Suc mask-Suc-double bit-simps)
  qed
 qed
 with True show ?thesis
  by simp
qed

lemma even-mask-iff:
 ‹even (mask n) ⟷ n = 0›
 using bit-mask-iff [of n 0] by (auto simp: bit-0)

lemma mask-Suc-0 [simp]:
 ‹mask (Suc 0) = 1›
 by (simp add: mask-Suc-double)

lemma mask-Suc-exp:
 ‹mask (Suc n) = 2 ^ n OR mask n›
 by (auto simp: bit-eq-iff bit-simps)

lemma mask-numeral:
 ‹mask (numeral n) = 1 + 2 * mask (pred-numeral n)›
 by (simp add: numeral-eq-Suc mask-Suc-double one-or-eq ac-simps)

lemma push-bit-0-id [simp]:
 ‹push-bit 0 = id›
 by (simp add: fun-eq-iff push-bit-eq-mult)

lemma push-bit-Suc [simp]:
 ‹push-bit (Suc n) a = push-bit n (a * 2)›
 by (simp add: push-bit-eq-mult ac-simps)

lemma push-bit-double:
 ‹push-bit n (a * 2) = push-bit n a * 2›
 by (simp add: push-bit-eq-mult ac-simps)

lemma bit-push-bit-iff [bit-simps]:
 ‹bit (push-bit m a) n ⟷ m ≤ n ∧ possible-bit TYPE('a) n ∧ bit a (n - m)›
proof (induction n arbitrary: m)
 case 0
 then show ?case
  by (auto simp: bit-0 push-bit-eq-mult)
next
 case (Suc n)
 show ?case
 proof (cases m)
  case 0
```

```
    then show ?thesis
      by (auto simp: bit-imp-possible-bit)
  next
    case (Suc m')
    with Suc.prems Suc.IH [of m'] show ?thesis
      apply (simp add: push-bit-double)
      apply (auto simp: possible-bit-less-imp bit-simps mult.commute [of - 2])
      done
  qed
qed

lemma funpow-double-eq-push-bit:
  ‹times 2 ^^ n = push-bit n›
  by (induction n) (simp-all add: fun-eq-iff push-bit-double ac-simps)

lemma div-push-bit-of-1-eq-drop-bit:
  ‹a div push-bit n 1 = drop-bit n a›
  by (simp add: push-bit-eq-mult drop-bit-eq-div)

lemma bits-ident:
  ‹push-bit n (drop-bit n a) + take-bit n a = a›
 using div-mult-mod-eq by (simp add: push-bit-eq-mult take-bit-eq-mod drop-bit-eq-div)

lemma push-bit-push-bit [simp]:
  ‹push-bit m (push-bit n a) = push-bit (m + n) a›
  by (simp add: push-bit-eq-mult power-add ac-simps)

lemma push-bit-of-0 [simp]:
  ‹push-bit n 0 = 0›
  by (simp add: push-bit-eq-mult)

lemma push-bit-of-1 [simp]:
  ‹push-bit n 1 = 2 ^ n›
  by (simp add: push-bit-eq-mult)

lemma push-bit-add:
  ‹push-bit n (a + b) = push-bit n a + push-bit n b›
  by (simp add: push-bit-eq-mult algebra-simps)

lemma push-bit-numeral [simp]:
  ‹push-bit (numeral l) (numeral k) = push-bit (pred-numeral l) (numeral (Num.Bit0
k))›
  by (simp add: numeral-eq-Suc mult-2-right) (simp add: numeral-Bit0)

lemma bit-drop-bit-eq [bit-simps]:
  ‹bit (drop-bit n a) = bit a ∘ (+) n›
  by rule (simp add: drop-bit-eq-div bit-iff-odd div-exp-eq)

lemma disjunctive-xor-eq-or:
```

```
  ‹a XOR b = a OR b› if ‹a AND b = 0›
  using that by (auto simp: bit-eq-iff bit-simps)

lemma disjunctive-add-eq-or:
  ‹a + b = a OR b› if ‹a AND b = 0›
proof (rule bit-eqI)
  fix n
  assume ‹possible-bit TYPE('a) n›
  moreover from that have ‹⋀n. ¬ bit (a AND b) n›
    by simp
  then have ‹⋀n. ¬ bit a n ∨ ¬ bit b n›
    by (simp add: bit-simps)
  ultimately show ‹bit (a + b) n ⟷ bit (a OR b) n›
  proof (induction n arbitrary: a b)
    case 0
    from 0(2)[of 0] show ?case
      by (auto simp: even-or-iff bit-0)
  next
    case (Suc n)
    from Suc.prems(2) [of 0] have even: ‹even a ∨ even b›
      by (auto simp: bit-0)
    have bit: ‹¬ bit (a div 2) n ∨ ¬ bit (b div 2) n› for n
      using Suc.prems(2) [of ‹Suc n›] by (simp add: bit-Suc)
    from Suc.prems have ‹possible-bit TYPE('a) n›
      using possible-bit-less-imp by force
    with ‹⋀n. ¬ bit (a div 2) n ∨ ¬ bit (b div 2) n› Suc.IH [of ‹a div 2› ‹b div
2›]
    have IH: ‹bit (a div 2 + b div 2) n ⟷ bit (a div 2 OR b div 2) n›
      by (simp add: bit-Suc)
    have ‹a + b = (a div 2 * 2 + a mod 2) + (b div 2 * 2 + b mod 2)›
      using div-mult-mod-eq [of a 2] div-mult-mod-eq [of b 2] by simp
    also have ‹... = of-bool (odd a ∨ odd b) + 2 * (a div 2 + b div 2)›
      using even by (auto simp: algebra-simps mod2-eq-if)
    finally have ‹bit ((a + b) div 2) n ⟷ bit (a div 2 + b div 2) n›
      using ‹possible-bit TYPE('a) (Suc n)› by simp (simp-all flip: bit-Suc add:
bit-double-iff possible-bit-def)
    also have ‹... ⟷ bit (a div 2 OR b div 2) n›
      by (rule IH)
    finally show ?case
      by (simp add: bit-simps flip: bit-Suc)
  qed
qed

lemma disjunctive-add-eq-xor:
  ‹a + b = a XOR b› if ‹a AND b = 0›
  using that by (simp add: disjunctive-add-eq-or disjunctive-xor-eq-or)

lemma take-bit-0 [simp]:
  take-bit 0 a = 0
```

by (*simp add*: *take-bit-eq-mod*)

lemma *bit-take-bit-iff* [*bit-simps*]:
‹*bit* (*take-bit m a*) *n* ⟷ *n* < *m* ∧ *bit a n*›
proof −
have ‹*push-bit m* (*drop-bit m a*) *AND take-bit m a* = *0*› (**is** ‹*?lhs* = -›)
proof (*rule bit-eqI*)
fix *n*
show ‹*bit ?lhs n* ⟷ *bit 0 n*›
proof (*cases* ‹*m* ≤ *n*›)
case *False*
then show *?thesis*
by (*simp add*: *bit-simps*)
next
case *True*
moreover define *q* **where** ‹*q* = *n* − *m*›
ultimately have ‹*n* = *m* + *q*› **by** *simp*
moreover have ‹¬ *bit* (*take-bit m a*) (*m* + *q*)›
by (*simp add*: *take-bit-eq-mod bit-iff-odd flip*: *div-exp-eq*)
ultimately show *?thesis*
by (*simp add*: *bit-simps*)
qed
qed
then have ‹*push-bit m* (*drop-bit m a*) *XOR take-bit m a* = *push-bit m* (*drop-bit m a*) + *take-bit m a*›
by (*simp add*: *disjunctive-add-eq-xor*)
also have ‹... = *a*›
by (*simp add*: *bits-ident*)
finally have ‹*bit* (*push-bit m* (*drop-bit m a*) *XOR take-bit m a*) *n* ⟷ *bit a n*›
by *simp*
also have ‹... ⟷ (*m* ≤ *n* ∨ *n* < *m*) ∧ *bit a n*›
by *auto*
also have ‹... ⟷ *m* ≤ *n* ∧ *bit a n* ∨ *n* < *m* ∧ *bit a n*›
by *auto*
also have ‹*m* ≤ *n* ∧ *bit a n* ⟷ *bit* (*push-bit m* (*drop-bit m a*)) *n*›
by (*auto simp*: *bit-simps bit-imp-possible-bit*)
finally show *?thesis*
by (*auto simp*: *bit-simps*)
qed

lemma *take-bit-Suc*:
‹*take-bit* (*Suc n*) *a* = *take-bit n* (*a div 2*) ∗ *2* + *a mod 2*› (**is** ‹*?lhs* = *?rhs*›)
proof (*rule bit-eqI*)
fix *m*
assume ‹*possible-bit TYPE*('*a*) *m*›
then show ‹*bit ?lhs m* ⟷ *bit ?rhs m*›
apply (*cases a rule*: *parity-cases*; *cases m*)
apply (*simp-all add*: *bit-simps even-bit-succ-iff mult.commute* [*of - 2*] *add.commute* [*of - 1*] *flip*: *bit-Suc*)

```
    apply (simp-all add: bit-0)
   done
qed

lemma take-bit-rec:
  ‹take-bit n a = (if n = 0 then 0 else take-bit (n − 1) (a div 2) ∗ 2 + a mod 2)›
  by (cases n) (simp-all add: take-bit-Suc)

lemma take-bit-Suc-0 [simp]:
  ‹take-bit (Suc 0) a = a mod 2›
  by (simp add: take-bit-eq-mod)

lemma take-bit-of-0 [simp]:
  ‹take-bit n 0 = 0›
  by (rule bit-eqI) (simp add: bit-simps)

lemma take-bit-of-1 [simp]:
  ‹take-bit n 1 = of-bool (n > 0)›
  by (cases n) (simp-all add: take-bit-Suc)

lemma drop-bit-of-0 [simp]:
  ‹drop-bit n 0 = 0›
  by (rule bit-eqI) (simp add: bit-simps)

lemma drop-bit-of-1 [simp]:
  ‹drop-bit n 1 = of-bool (n = 0)›
  by (rule bit-eqI) (simp add: bit-simps ac-simps)

lemma drop-bit-0 [simp]:
  ‹drop-bit 0 = id›
  by (simp add: fun-eq-iff drop-bit-eq-div)

lemma drop-bit-Suc:
  ‹drop-bit (Suc n) a = drop-bit n (a div 2)›
  using div-exp-eq [of a 1] by (simp add: drop-bit-eq-div)

lemma drop-bit-rec:
  ‹drop-bit n a = (if n = 0 then a else drop-bit (n − 1) (a div 2))›
  by (cases n) (simp-all add: drop-bit-Suc)

lemma drop-bit-half:
  ‹drop-bit n (a div 2) = drop-bit n a div 2›
  by (induction n arbitrary: a) (simp-all add: drop-bit-Suc)

lemma drop-bit-of-bool [simp]:
  ‹drop-bit n (of-bool b) = of-bool (n = 0 ∧ b)›
  by (cases n) simp-all

lemma even-take-bit-eq [simp]:
```

```
  ‹even (take-bit n a) ⟷ n = 0 ∨ even a›
  by (simp add: take-bit-rec [of n a])

lemma take-bit-take-bit [simp]:
  ‹take-bit m (take-bit n a) = take-bit (min m n) a›
  by (rule bit-eqI) (simp add: bit-simps)

lemma drop-bit-drop-bit [simp]:
  ‹drop-bit m (drop-bit n a) = drop-bit (m + n) a›
  by (simp add: drop-bit-eq-div power-add div-exp-eq ac-simps)

lemma push-bit-take-bit:
  ‹push-bit m (take-bit n a) = take-bit (m + n) (push-bit m a)›
  by (rule bit-eqI) (auto simp: bit-simps)

lemma take-bit-push-bit:
  ‹take-bit m (push-bit n a) = push-bit n (take-bit (m - n) a)›
  by (rule bit-eqI) (auto simp: bit-simps)

lemma take-bit-drop-bit:
  ‹take-bit m (drop-bit n a) = drop-bit n (take-bit (m + n) a)›
  by (rule bit-eqI) (auto simp: bit-simps)

lemma drop-bit-take-bit:
  ‹drop-bit m (take-bit n a) = take-bit (n - m) (drop-bit m a)›
  by (rule bit-eqI) (auto simp: bit-simps)

lemma even-push-bit-iff [simp]:
  ‹even (push-bit n a) ⟷ n ≠ 0 ∨ even a›
  by (simp add: push-bit-eq-mult) auto

lemma stable-imp-drop-bit-eq:
  ‹drop-bit n a = a›
  if ‹a div 2 = a›
  by (induction n) (simp-all add: that drop-bit-Suc)

lemma stable-imp-take-bit-eq:
  ‹take-bit n a = (if even a then 0 else mask n)›
    if ‹a div 2 = a›
  by (rule bit-eqI) (use that in ‹simp add: bit-simps stable-imp-bit-iff-odd›)

lemma exp-dvdE:
  assumes ‹2 ^ n dvd a›
  obtains b where ‹a = push-bit n b›
proof -
  from assms obtain b where ‹a = 2 ^ n * b› ..
  then have ‹a = push-bit n b›
    by (simp add: push-bit-eq-mult ac-simps)
  with that show thesis .
```

qed

lemma *take-bit-eq-0-iff*:
‹*take-bit n a = 0 ⟷ 2 ^ n dvd a*› (**is** ‹*?P ⟷ ?Q*›)
proof
assume *?P*
then show *?Q*
by (*simp add*: *take-bit-eq-mod mod-0-imp-dvd*)
next
assume *?Q*
then obtain *b* **where** ‹*a = push-bit n b*›
by (*rule exp-dvdE*)
then show *?P*
by (*simp add*: *take-bit-push-bit*)
qed

lemma *take-bit-tightened*:
‹*take-bit m a = take-bit m b*› **if** ‹*take-bit n a = take-bit n b*› **and** ‹*m ≤ n*›
proof −
from *that* **have** ‹*take-bit m (take-bit n a) = take-bit m (take-bit n b)*›
by *simp*
then have ‹*take-bit (min m n) a = take-bit (min m n) b*›
by *simp*
with *that* **show** *?thesis*
by (*simp add*: *min-def*)
qed

lemma *take-bit-eq-self-iff-drop-bit-eq-0*:
‹*take-bit n a = a ⟷ drop-bit n a = 0*› (**is** ‹*?P ⟷ ?Q*›)
proof
assume *?P*
show *?Q*
proof (*rule bit-eqI*)
fix *m*
from ‹*?P*› **have** ‹*a = take-bit n a*› **..**
also have ‹¬ *bit (take-bit n a) (n + m)*›
unfolding *bit-simps*
by (*simp add*: *bit-simps*)
finally show ‹*bit (drop-bit n a) m ⟷ bit 0 m*›
by (*simp add*: *bit-simps*)
qed
next
assume *?Q*
show *?P*
proof (*rule bit-eqI*)
fix *m*
from ‹*?Q*› **have** ‹¬ *bit (drop-bit n a) (m − n)*›
by *simp*
then have ‹ ¬ *bit a (n + (m − n))*›

```
    by (simp add: bit-simps)
  then show ‹bit (take-bit n a) m ⟷ bit a m›
    by (cases ‹m < n›) (auto simp: bit-simps)
 qed
qed

lemma impossible-bit-imp-take-bit-eq-self:
 ‹take-bit n a = a› if ‹¬ possible-bit TYPE('a) n›
proof −
 have ‹drop-bit n a = 0›
 proof (rule bit-eqI)
  fix m
  show ‹bit (drop-bit n a) m ⟷ bit 0 m›
   using possible-bit-less-imp [of ‹n + m› n] that
   by (auto simp add: bit-simps dest: bit-imp-possible-bit)
 qed
 then show ?thesis
  by (simp add: take-bit-eq-self-iff-drop-bit-eq-0)
qed

lemma drop-bit-exp-eq:
 ‹drop-bit m (2 ^ n) = of-bool (m ≤ n ∧ possible-bit TYPE('a) n) * 2 ^ (n − m)›
 by (auto simp: bit-eq-iff bit-simps)

lemma take-bit-and [simp]:
 ‹take-bit n (a AND b) = take-bit n a AND take-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma take-bit-or [simp]:
 ‹take-bit n (a OR b) = take-bit n a OR take-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma take-bit-xor [simp]:
 ‹take-bit n (a XOR b) = take-bit n a XOR take-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma push-bit-and [simp]:
 ‹push-bit n (a AND b) = push-bit n a AND push-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma push-bit-or [simp]:
 ‹push-bit n (a OR b) = push-bit n a OR push-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma push-bit-xor [simp]:
 ‹push-bit n (a XOR b) = push-bit n a XOR push-bit n b›
 by (auto simp: bit-eq-iff bit-simps)

lemma drop-bit-and [simp]:
```

‹*drop-bit n* (*a AND b*) = *drop-bit n a AND drop-bit n b*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *drop-bit-or* [*simp*]:
‹*drop-bit n* (*a OR b*) = *drop-bit n a OR drop-bit n b*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *drop-bit-xor* [*simp*]:
‹*drop-bit n* (*a XOR b*) = *drop-bit n a XOR drop-bit n b*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *take-bit-of-mask* [*simp*]:
‹*take-bit m* (*mask n*) = *mask* (*min m n*)›
by (*rule bit-eqI*) (*simp add*: *bit-simps*)

lemma *take-bit-eq-mask*:
‹*take-bit n a* = *a AND mask n*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *or-eq-0-iff*:
‹*a OR b* = *0* ⟷ *a* = *0* ∧ *b* = *0*›
by (*auto simp*: *bit-eq-iff bit-or-iff*)

lemma *bit-iff-and-drop-bit-eq-1*:
‹*bit a n* ⟷ *drop-bit n a AND 1* = *1*›
by (*simp add*: *bit-iff-odd-drop-bit and-one-eq odd-iff-mod-2-eq-one*)

lemma *bit-iff-and-push-bit-not-eq-0*:
‹*bit a n* ⟷ *a AND push-bit n 1* ≠ *0*›
by (*cases* ‹*possible-bit TYPE*('*a*) *n*›) (*simp-all add*: *bit-eq-iff bit-simps impossible-bit*)

lemma *bit-set-bit-iff* [*bit-simps*]:
‹*bit* (*set-bit m a*) *n* ⟷ *bit a n* ∨ (*m* = *n* ∧ *possible-bit TYPE*('*a*) *n*)›
by (*auto simp*: *set-bit-eq-or bit-or-iff bit-exp-iff*)

lemma *even-set-bit-iff*:
‹*even* (*set-bit m a*) ⟷ *even a* ∧ *m* ≠ *0*›
using *bit-set-bit-iff* [*of m a 0*] **by** (*auto simp*: *bit-0*)

lemma *bit-unset-bit-iff* [*bit-simps*]:
‹*bit* (*unset-bit m a*) *n* ⟷ *bit a n* ∧ *m* ≠ *n*›
by (*auto simp*: *unset-bit-eq-or-xor bit-simps dest*: *bit-imp-possible-bit*)

lemma *even-unset-bit-iff*:
‹*even* (*unset-bit m a*) ⟷ *even a* ∨ *m* = *0*›
using *bit-unset-bit-iff* [*of m a 0*] **by** (*auto simp*: *bit-0*)

lemma *bit-flip-bit-iff* [*bit-simps*]:

‹*bit* (*flip-bit m a*) *n* ⟷ (*m* = *n* ⟷ ¬ *bit a n*) ∧ *possible-bit TYPE*('*a*) *n*›
by (*auto simp*: *bit-eq-iff bit-simps flip-bit-eq-xor bit-imp-possible-bit*)

lemma *even-flip-bit-iff*:
‹*even* (*flip-bit m a*) ⟷ ¬ (*even a* ⟷ *m* = *0*)›
using *bit-flip-bit-iff* [*of m a 0*] **by** (*auto simp*: *possible-bit-def bit-0*)

lemma *and-exp-eq-0-iff-not-bit*:
‹*a AND 2* ^ *n* = *0* ⟷ ¬ *bit a n*› (**is** ‹*?P* ⟷ *?Q*›)
using *bit-imp-possible-bit*[*of a n*]
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *bit-sum-mult-2-cases*:
assumes *a*: ‹∀ *j*. ¬ *bit a* (*Suc j*)›
shows ‹*bit* (*a* + *2* ∗ *b*) *n* = (*if n* = *0 then odd a else bit* (*2* ∗ *b*) *n*)›
proof −
from *a* **have** ‹*n* = *0*› **if** ‹*bit a n*› **for** *n* **using** *that*
by (*cases n*) *simp-all*
then have ‹*a* = *0* ∨ *a* = *1*›
by (*auto simp*: *bit-eq-iff bit-1-iff*)
then show *?thesis*
by (*cases n*) (*auto simp*: *bit-0 bit-double-iff even-bit-succ-iff*)
qed

lemma *set-bit-0*:
‹*set-bit 0 a* = *1* + *2* ∗ (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-simps even-bit-succ-iff simp flip*: *bit-Suc*)

lemma *set-bit-Suc*:
‹*set-bit* (*Suc n*) *a* = *a mod 2* + *2* ∗ *set-bit n* (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip*: *bit-Suc*
elim: *possible-bit-less-imp*)

lemma *unset-bit-0*:
‹*unset-bit 0 a* = *2* ∗ (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-simps simp flip*: *bit-Suc*)

lemma *unset-bit-Suc*:
‹*unset-bit* (*Suc n*) *a* = *a mod 2* + *2* ∗ *unset-bit n* (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip*: *bit-Suc*)

lemma *flip-bit-0*:
‹*flip-bit 0 a* = *of-bool* (*even a*) + *2* ∗ (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-simps even-bit-succ-iff bit-0 simp flip*: *bit-Suc*)

lemma *flip-bit-Suc*:
‹*flip-bit* (*Suc n*) *a* = *a mod 2* + *2* ∗ *flip-bit n* (*a div 2*)›
by (*auto simp*: *bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 simp flip*: *bit-Suc*
elim: *possible-bit-less-imp*)

lemma *flip-bit-eq-if*:
‹*flip-bit n a* = (*if bit a n then unset-bit else set-bit*) *n a*›
by (*rule bit-eqI*) (*auto simp*: *bit-set-bit-iff bit-unset-bit-iff bit-flip-bit-iff*)

lemma *take-bit-set-bit-eq*:
‹*take-bit n* (*set-bit m a*) = (*if n* ≤ *m then take-bit n a else set-bit m* (*take-bit n a*))›
by (*rule bit-eqI*) (*auto simp*: *bit-take-bit-iff bit-set-bit-iff*)

lemma *take-bit-unset-bit-eq*:
‹*take-bit n* (*unset-bit m a*) = (*if n* ≤ *m then take-bit n a else unset-bit m* (*take-bit n a*))›
by (*rule bit-eqI*) (*auto simp*: *bit-take-bit-iff bit-unset-bit-iff*)

lemma *take-bit-flip-bit-eq*:
‹*take-bit n* (*flip-bit m a*) = (*if n* ≤ *m then take-bit n a else flip-bit m* (*take-bit n a*))›
by (*rule bit-eqI*) (*auto simp*: *bit-take-bit-iff bit-flip-bit-iff*)

lemma *push-bit-Suc-numeral* [*simp*]:
‹*push-bit* (*Suc n*) (*numeral k*) = *push-bit n* (*numeral* (*Num.Bit0 k*))›
by (*simp add*: *numeral-eq-Suc mult-2-right*) (*simp add*: *numeral-Bit0*)

lemma *mask-eq-0-iff* [*simp*]:
‹*mask n* = *0* ⟷ *n* = *0*›
by (*cases n*) (*simp-all add*: *mask-Suc-double or-eq-0-iff*)

lemma *bit-horner-sum-bit-iff* [*bit-simps*]:
‹*bit* (*horner-sum of-bool 2 bs*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ *n* < *length bs* ∧ *bs* ! *n*›
proof (*induction bs arbitrary*: *n*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons b bs*)
show *?case*
proof (*cases n*)
case *0*
then show *?thesis*
by (*simp add*: *bit-0*)
next
case (*Suc m*)
with *bit-rec* [*of* - *n*] *Cons.prems Cons.IH* [*of m*]
show *?thesis*
by (*simp add*: *bit-simps*)
(*auto simp*: *possible-bit-less-imp bit-simps simp flip*: *bit-Suc*)
qed

qed

lemma *horner-sum-bit-eq-take-bit*:
‹*horner-sum of-bool 2* (*map* (*bit a*) [*0*..<*n*]) = *take-bit n a*›
by (*rule bit-eqI*) (*auto simp*: *bit-simps*)

lemma *take-bit-horner-sum-bit-eq*:
‹*take-bit n* (*horner-sum of-bool 2 bs*) = *horner-sum of-bool 2* (*take n bs*)›
by (*auto simp*: *bit-eq-iff bit-take-bit-iff bit-horner-sum-bit-iff*)

lemma *take-bit-sum*:
‹*take-bit n a* = (∑ *k* = *0*..<*n*. *push-bit k* (*of-bool* (*bit a k*)))›
by (*simp flip*: *horner-sum-bit-eq-take-bit add*: *horner-sum-eq-sum push-bit-eq-mult*)

lemma *set-bit-eq*:
‹*set-bit n a* = *a* + *of-bool* (¬ *bit a n*) * *2* ^ *n*›
proof −
have ‹*a AND of-bool* (¬ *bit a n*) * *2* ^ *n* = *0*›
by (*auto simp*: *bit-eq-iff bit-simps*)
then show *?thesis*
by (*auto simp*: *bit-eq-iff bit-simps disjunctive-add-eq-or*)
qed

end

class *ring-bit-operations* = *semiring-bit-operations* + *ring-parity* +
fixes *not* :: ‹*'a* ⇒ *'a*› (‹*NOT*›)
assumes *not-eq-complement*: ‹*NOT a* = − *a* − *1*›
begin

For the sake of code generation *NOT* is specified as definitional class operation. Note that *NOT* has no sensible definition for unlimited but only positive bit strings (type *nat*).

lemma *bits-minus-1-mod-2-eq* [*simp*]:
‹(− *1*) *mod 2* = *1*›
by (*simp add*: *mod-2-eq-odd*)

lemma *minus-eq-not-plus-1*:
‹− *a* = *NOT a* + *1*›
using *not-eq-complement* [*of a*] **by** *simp*

lemma *minus-eq-not-minus-1*:
‹− *a* = *NOT* (*a* − *1*)›
using *not-eq-complement* [*of* ‹*a* − *1*›] **by** *simp* (*simp add*: *algebra-simps*)

lemma *not-rec*:
‹*NOT a* = *of-bool* (*even a*) + *2* * *NOT* (*a div 2*)›
by (*simp add*: *not-eq-complement algebra-simps mod-2-eq-odd flip*: *minus-mod-eq-mult-div*)

lemma *decr-eq-not-minus*:
‹$a - 1 = NOT\ (-\ a)$›
using *not-eq-complement* [*of* ‹$-\ a$›] **by** *simp*

lemma *even-not-iff* [*simp*]:
‹$even\ (NOT\ a) \longleftrightarrow odd\ a$›
by (*simp add*: *not-eq-complement*)

lemma *bit-not-iff* [*bit-simps*]:
‹$bit\ (NOT\ a)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge \neg\ bit\ a\ n$›
proof (*cases* ‹$possible\text{-}bit\ TYPE('a)\ n$›)
case *False*
then show *?thesis*
by (*auto dest*: *bit-imp-possible-bit*)
next
case *True*
moreover have ‹$bit\ (NOT\ a)\ n \longleftrightarrow \neg\ bit\ a\ n$›
using ‹$possible\text{-}bit\ TYPE('a)\ n$› **proof** (*induction n arbitrary*: *a*)
case *0*
then show *?case*
by (*simp add*: *bit-0*)
next
case (*Suc n*)
from *Suc.prems Suc.IH* [*of* ‹$a\ div\ 2$›]
show *?case*
by (*simp add*: *impossible-bit possible-bit-less-imp not-rec* [*of a*] *even-bit-succ-iff bit-double-iff flip*: *bit-Suc*)
qed
ultimately show *?thesis*
by *simp*
qed

lemma *bit-not-exp-iff* [*bit-simps*]:
‹$bit\ (NOT\ (2\ \hat{}\ m))\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge n \neq m$›
by (*auto simp*: *bit-not-iff bit-exp-iff*)

lemma *bit-minus-iff* [*bit-simps*]:
‹$bit\ (-\ a)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge \neg\ bit\ (a - 1)\ n$›
by (*simp add*: *minus-eq-not-minus-1 bit-not-iff*)

lemma *bit-minus-1-iff* [*simp*]:
‹$bit\ (-\ 1)\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n$›
by (*simp add*: *bit-minus-iff*)

lemma *bit-minus-exp-iff* [*bit-simps*]:
‹$bit\ (-\ (2\ \hat{}\ m))\ n \longleftrightarrow possible\text{-}bit\ TYPE('a)\ n \wedge n \geq m$›
by (*auto simp*: *bit-simps simp flip*: *mask-eq-exp-minus-1*)

lemma *bit-minus-2-iff* [*simp*]:

‹*bit* (− *2*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ *n* > *0*›
by (*simp add*: *bit-minus-iff bit-1-iff*)

lemma *bit-decr-iff*:
‹*bit* (*a* − *1*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ ¬ *bit* (− *a*) *n*›
by (*simp add*: *decr-eq-not-minus bit-not-iff*)

lemma *bit-not-iff-eq*:
‹*bit* (*NOT a*) *n* ⟷ *2* ^ *n* ≠ *0* ∧ ¬ *bit a n*›
by (*simp add*: *bit-simps possible-bit-def*)

lemma *not-one-eq* [*simp*]:
‹*NOT 1* = − *2*›
by (*rule bit-eqI*, *simp add*: *bit-simps*)

sublocale *and*: *semilattice-neutr* ‹(*AND*)› ‹− *1*›
by *standard* (*rule bit-eqI*, *simp add*: *bit-and-iff*)

sublocale *bit*: *abstract-boolean-algebra* ‹(*AND*)› ‹(*OR*)› *NOT 0* ‹− *1*›
by *standard* (*auto simp*: *bit-and-iff bit-or-iff bit-not-iff intro*: *bit-eqI*)

sublocale *bit*: *abstract-boolean-algebra-sym-diff* ‹(*AND*)› ‹(*OR*)› *NOT 0* ‹− *1*› ‹(*XOR*)›
proof
show ‹⋀*x y*. *x XOR y* = *x AND NOT y OR NOT x AND y*›
by (*intro bit-eqI*) (*auto simp*: *bit-simps*)
qed

lemma *and-eq-not-not-or*:
‹*a AND b* = *NOT* (*NOT a OR NOT b*)›
by *simp*

lemma *or-eq-not-not-and*:
‹*a OR b* = *NOT* (*NOT a AND NOT b*)›
by *simp*

lemma *not-add-distrib*:
‹*NOT* (*a* + *b*) = *NOT a* − *b*›
by (*simp add*: *not-eq-complement algebra-simps*)

lemma *not-diff-distrib*:
‹*NOT* (*a* − *b*) = *NOT a* + *b*›
using *not-add-distrib* [*of a* ‹− *b*›] **by** *simp*

lemma *and-eq-minus-1-iff*:
‹*a AND b* = − *1* ⟷ *a* = − *1* ∧ *b* = − *1*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *disjunctive-and-not-eq-xor*:

‹*a AND NOT b = a XOR b*› **if** ‹*NOT a AND b = 0*›
using *that* **by** (*auto simp*: *bit-eq-iff bit-simps*)

lemma *disjunctive-diff-eq-and-not*:
‹*a − b = a AND NOT b*› **if** ‹*NOT a AND b = 0*›
proof −
from *that* **have** ‹*NOT a + b = NOT a OR b*›
by (*rule disjunctive-add-eq-or*)
then have ‹*NOT* (*NOT a + b*) = *NOT* (*NOT a OR b*)›
by *simp*
then show *?thesis*
by (*simp add*: *not-add-distrib*)
qed

lemma *disjunctive-diff-eq-xor*:
‹*a AND NOT b = a XOR b*› **if** ‹*NOT a AND b = 0*›
using *that* **by** (*simp add*: *disjunctive-and-not-eq-xor disjunctive-diff-eq-and-not*)

lemma *push-bit-minus*:
‹*push-bit n* (− *a*) = − *push-bit n a*›
by (*simp add*: *push-bit-eq-mult*)

lemma *take-bit-not-take-bit*:
‹*take-bit n* (*NOT* (*take-bit n a*)) = *take-bit n* (*NOT a*)›
by (*auto simp*: *bit-eq-iff bit-take-bit-iff bit-not-iff*)

lemma *take-bit-not-iff*:
‹*take-bit n* (*NOT a*) = *take-bit n* (*NOT b*) ⟷ *take-bit n a* = *take-bit n b*›
by (*auto simp*: *bit-eq-iff bit-simps*)

lemma *take-bit-not-eq-mask-diff*:
‹*take-bit n* (*NOT a*) = *mask n* − *take-bit n a*›
proof −
have ‹*NOT* (*mask n*) *AND take-bit n a = 0*›
by (*simp add*: *bit-eq-iff bit-simps*)
moreover have ‹*take-bit n* (*NOT a*) = *mask n AND NOT* (*take-bit n a*)›
by (*auto simp*: *bit-eq-iff bit-simps*)
ultimately show *?thesis*
by (*simp add*: *disjunctive-diff-eq-and-not*)
qed

lemma *mask-eq-take-bit-minus-one*:
‹*mask n = take-bit n* (− *1*)›
by (*simp add*: *bit-eq-iff bit-mask-iff bit-take-bit-iff conj-commute*)

lemma *take-bit-minus-one-eq-mask* [*simp*]:
‹*take-bit n* (− *1*) = *mask n*›
by (*simp add*: *mask-eq-take-bit-minus-one*)

lemma *minus-exp-eq-not-mask*:
‹*− (2 ^ n) = NOT (mask n)*›
by (*rule bit-eqI*) (*simp add*: *bit-minus-iff bit-not-iff flip*: *mask-eq-exp-minus-1*)

lemma *push-bit-minus-one-eq-not-mask* [*simp*]:
‹*push-bit n (− 1) = NOT (mask n)*›
by (*simp add*: *push-bit-eq-mult minus-exp-eq-not-mask*)

lemma *take-bit-not-mask-eq-0*:
‹*take-bit m (NOT (mask n)) = 0*› **if** ‹*n ≥ m*›
by (*rule bit-eqI*) (*use that* **in** ‹*simp add*: *bit-take-bit-iff bit-not-iff bit-mask-iff*›)

lemma *mask-eq-minus-one-if-not-possible-bit*:
‹*mask n = − 1*› **if** ‹¬ *possible-bit TYPE('a) n*›
using *that mask-eq-take-bit-minus-one* [*of n*] *impossible-bit-imp-take-bit-eq-self* [*of n ‹− 1›*]
by *simp*

lemma *unset-bit-eq-and-not*:
‹*unset-bit n a = a AND NOT (push-bit n 1)*›
by (*rule bit-eqI*) (*auto simp*: *bit-simps*)

lemma *push-bit-Suc-minus-numeral* [*simp*]:
‹*push-bit (Suc n) (− numeral k) = push-bit n (− numeral (Num.Bit0 k))*›
using *local.push-bit-Suc-numeral push-bit-minus* **by** *presburger*

lemma *push-bit-minus-numeral* [*simp*]:
‹*push-bit (numeral l) (− numeral k) = push-bit (pred-numeral l) (− numeral (Num.Bit0 k))*›
by (*simp only*: *numeral-eq-Suc push-bit-Suc-minus-numeral*)

lemma *take-bit-Suc-minus-1-eq*:
‹*take-bit (Suc n) (− 1) = 2 ^ Suc n − 1*›
by (*simp add*: *mask-eq-exp-minus-1*)

lemma *take-bit-numeral-minus-1-eq*:
‹*take-bit (numeral k) (− 1) = 2 ^ numeral k − 1*›
by (*simp add*: *mask-eq-exp-minus-1*)

lemma *push-bit-mask-eq*:
‹*push-bit m (mask n) = mask (n + m) AND NOT (mask m)*›
by (*rule bit-eqI*) (*auto simp*: *bit-simps not-less possible-bit-less-imp*)

lemma *slice-eq-mask*:
‹*push-bit n (take-bit m (drop-bit n a)) = a AND mask (m + n) AND NOT (mask n)*›
by (*rule bit-eqI*) (*auto simp*: *bit-simps*)

lemma *push-bit-numeral-minus-1* [*simp*]:

‹*push-bit* (*numeral n*) (− *1*) = − (*2* ^ *numeral n*)›
by (*simp add*: *push-bit-eq-mult*)

lemma *unset-bit-eq*:
‹*unset-bit n a* = *a* − *of-bool* (*bit a n*) ∗ *2* ^ *n*›
proof −
have ‹*NOT a AND of-bool* (*bit a n*) ∗ *2* ^ *n* = *0*›
by (*auto simp*: *bit-eq-iff bit-simps*)
moreover have ‹*unset-bit n a* = *a AND NOT* (*of-bool* (*bit a n*) ∗ *2* ^ *n*)›
by (*auto simp*: *bit-eq-iff bit-simps*)
ultimately show *?thesis*
by (*simp add*: *disjunctive-diff-eq-and-not*)
qed

end

68.3 Common algebraic structure

class *linordered-euclidean-semiring-bit-operations* =
linordered-euclidean-semiring + *semiring-bit-operations*
begin

lemma *possible-bit* [*simp*]:
‹*possible-bit TYPE*('*a*) *n*›
by (*simp add*: *possible-bit-def*)

lemma *take-bit-of-exp* [*simp*]:
‹*take-bit m* (*2* ^ *n*) = *of-bool* (*n* < *m*) ∗ *2* ^ *n*›
by (*simp add*: *take-bit-eq-mod exp-mod-exp*)

lemma *take-bit-of-2* [*simp*]:
‹*take-bit n 2* = *of-bool* (*2* ≤ *n*) ∗ *2*›
using *take-bit-of-exp* [*of n 1*] **by** *simp*

lemma *push-bit-eq-0-iff* [*simp*]:
‹*push-bit n a* = *0* ⟷ *a* = *0*›
by (*simp add*: *push-bit-eq-mult*)

lemma *take-bit-add*:
‹*take-bit n* (*take-bit n a* + *take-bit n b*) = *take-bit n* (*a* + *b*)›
by (*simp add*: *take-bit-eq-mod mod-simps*)

lemma *take-bit-of-1-eq-0-iff* [*simp*]:
‹*take-bit n 1* = *0* ⟷ *n* = *0*›
by (*simp add*: *take-bit-eq-mod*)

lemma *drop-bit-Suc-bit0* [*simp*]:
‹*drop-bit* (*Suc n*) (*numeral* (*Num.Bit0 k*)) = *drop-bit n* (*numeral k*)›
by (*simp add*: *drop-bit-Suc numeral-Bit0-div-2*)

lemma *drop-bit-Suc-bit1* [*simp*]:
‹*drop-bit* (*Suc n*) (*numeral* (*Num.Bit1 k*)) = *drop-bit n* (*numeral k*)›
by (*simp add*: *drop-bit-Suc numeral-Bit0-div-2*)

lemma *drop-bit-numeral-bit0* [*simp*]:
‹*drop-bit* (*numeral l*) (*numeral* (*Num.Bit0 k*)) = *drop-bit* (*pred-numeral l*) (*numeral k*)›
by (*simp add*: *drop-bit-rec numeral-Bit0-div-2*)

lemma *drop-bit-numeral-bit1* [*simp*]:
‹*drop-bit* (*numeral l*) (*numeral* (*Num.Bit1 k*)) = *drop-bit* (*pred-numeral l*) (*numeral k*)›
by (*simp add*: *drop-bit-rec numeral-Bit0-div-2*)

lemma *take-bit-Suc-1* [*simp*]:
‹*take-bit* (*Suc n*) *1* = *1*›
by (*simp add*: *take-bit-Suc*)

lemma *take-bit-Suc-bit0*:
‹*take-bit* (*Suc n*) (*numeral* (*Num.Bit0 k*)) = *take-bit n* (*numeral k*) * *2*›
by (*simp add*: *take-bit-Suc numeral-Bit0-div-2*)

lemma *take-bit-Suc-bit1*:
‹*take-bit* (*Suc n*) (*numeral* (*Num.Bit1 k*)) = *take-bit n* (*numeral k*) * *2* + *1*›
by (*simp add*: *take-bit-Suc numeral-Bit0-div-2 mod-2-eq-odd*)

lemma *take-bit-numeral-1* [*simp*]:
‹*take-bit* (*numeral l*) *1* = *1*›
by (*simp add*: *take-bit-rec* [*of* ‹*numeral l*› *1*])

lemma *take-bit-numeral-bit0*:
‹*take-bit* (*numeral l*) (*numeral* (*Num.Bit0 k*)) = *take-bit* (*pred-numeral l*) (*numeral k*) * *2*›
by (*simp add*: *take-bit-rec numeral-Bit0-div-2*)

lemma *take-bit-numeral-bit1*:
‹*take-bit* (*numeral l*) (*numeral* (*Num.Bit1 k*)) = *take-bit* (*pred-numeral l*) (*numeral k*) * *2* + *1*›
by (*simp add*: *take-bit-rec numeral-Bit0-div-2 mod-2-eq-odd*)

lemma *bit-of-nat-iff-bit* [*bit-simps*]:
‹*bit* (*of-nat m*) *n* ⟷ *bit m n*›
proof −
have ‹*even* (*m div* 2 ^ *n*) ⟷ *even* (*of-nat* (*m div* 2 ^ *n*))›
by *simp*
also have ‹*of-nat* (*m div* 2 ^ *n*) = *of-nat m div of-nat* (2 ^ *n*)›
by (*simp add*: *of-nat-div*)
finally show *?thesis*

```
    by (simp add: bit-iff-odd semiring-bits-class.bit-iff-odd)
qed

lemma drop-bit-mask-eq:
  ‹drop-bit m (mask n) = mask (n − m)›
  by (rule bit-eqI) (auto simp: bit-simps possible-bit-def)

lemma bit-push-bit-iff':
  ‹bit (push-bit m a) n ⟷ m ≤ n ∧ bit a (n − m)›
  by (simp add: bit-simps)

lemma mask-half:
  ‹mask n div 2 = mask (n − 1)›
  by (cases n) (simp-all add: mask-Suc-double one-or-eq)

lemma take-bit-Suc-from-most:
  ‹take-bit (Suc n) a = 2 ^ n * of-bool (bit a n) + take-bit n a›
  using mod-mult2-eq' [of a ‹2 ^ n› 2]
  by (simp only: take-bit-eq-mod power-Suc2)
    (simp-all add: bit-iff-odd odd-iff-mod-2-eq-one)

lemma take-bit-nonnegative [simp]:
  ‹0 ≤ take-bit n a›
  using horner-sum-nonnegative by (simp flip: horner-sum-bit-eq-take-bit)

lemma not-take-bit-negative [simp]:
  ‹¬ take-bit n a < 0›
  by (simp add: not-less)

lemma bit-imp-take-bit-positive:
  ‹0 < take-bit m a› if ‹n < m› and ‹bit a n›
proof (rule ccontr)
  assume ‹¬ 0 < take-bit m a›
  then have ‹take-bit m a = 0›
    by (auto simp: not-less intro: order-antisym)
  then have ‹bit (take-bit m a) n = bit 0 n›
    by simp
  with that show False
    by (simp add: bit-take-bit-iff)
qed

lemma take-bit-mult:
  ‹take-bit n (take-bit n a * take-bit n b) = take-bit n (a * b)›
  by (simp add: take-bit-eq-mod mod-mult-eq)

lemma drop-bit-push-bit:
  ‹drop-bit m (push-bit n a) = drop-bit (m − n) (push-bit (n − m) a)›
  by (cases ‹m ≤ n›)
     (auto simp: mult.left-commute [of - ‹2 ^ n›] mult.commute [of - ‹2 ^ n›]
```

mult.assoc
mult.commute [*of a*] *drop-bit-eq-div push-bit-eq-mult not-le power-add Orderings.not-le dest*!: *le-Suc-ex less-imp-Suc-add*)

end

68.4 Instance *int*

locale *fold2-bit-int* =
fixes *f* :: ‹*bool* ⇒ *bool* ⇒ *bool*›
begin

context
begin

function *F* :: ‹*int* ⇒ *int* ⇒ *int*›
where ‹*F k l* = (*if k* ∈ {*0*, − *1*} ∧ *l* ∈ {*0*, − *1*}
then − *of-bool* (*f* (*odd k*) (*odd l*))
else of-bool (*f* (*odd k*) (*odd l*)) + *2* ∗ (*F* (*k div 2*) (*l div 2*)))›
by *auto*

private termination proof (*relation* ‹*measure* (λ(*k*, *l*). *nat* (|*k*| + |*l*|))›)
have *less-eq*: ‹|*k div 2*| ≤ |*k*|› **for** *k* :: *int*
by (*cases k*) (*simp-all add*: *divide-int-def nat-add-distrib*)
then have *less*: ‹|*k div 2*| < |*k*|› **if** ‹*k* ∉ {*0*, − *1*}› **for** *k* :: *int*
using *that* **by** (*auto simp*: *less-le* [*of k*])
show ‹*wf* (*measure* (λ(*k*, *l*). *nat* (|*k*| + |*l*|)))›
by *simp*
show ‹((*k div 2*, *l div 2*), *k*, *l*) ∈ *measure* (λ(*k*, *l*). *nat* (|*k*| + |*l*|))›
if ‹¬ (*k* ∈ {*0*, − *1*} ∧ *l* ∈ {*0*, − *1*})› **for** *k l*
proof −
from *that* **have** ∗: ‹*k* ∉ {*0*, − *1*} ∨ *l* ∉ {*0*, − *1*}›
by *simp*
then have ‹*0* < |*k*| + |*l*|›
by *auto*
moreover from ∗ **have** ‹|*k div 2*| + |*l div 2*| < |*k*| + |*l*|›
proof
assume ‹*k* ∉ {*0*, − *1*}›
then have ‹|*k div 2*| < |*k*|›
by (*rule less*)
with *less-eq* [*of l*] **show** *?thesis*
by *linarith*
next
assume ‹*l* ∉ {*0*, − *1*}›
then have ‹|*l div 2*| < |*l*|›
by (*rule less*)
with *less-eq* [*of k*] **show** *?thesis*
by *linarith*
qed

```
    ultimately show ?thesis
      by (simp only: in-measure split-def fst-conv snd-conv nat-mono-iff)
  qed
qed

declare F.simps [simp del]

lemma rec:
  ‹F k l = of-bool (f (odd k) (odd l)) + 2 * (F (k div 2) (l div 2))›
    for k l :: int
proof (cases ‹k ∈ {0, - 1} ∧ l ∈ {0, - 1}›)
  case True
  then show ?thesis
    by (auto simp: F.simps [of 0] F.simps [of ‹- 1›])
next
  case False
  then show ?thesis
    by (auto simp: ac-simps F.simps [of k l])
qed

lemma bit-iff:
  ‹bit (F k l) n ⟷ f (bit k n) (bit l n)› for k l :: int
proof (induction n arbitrary: k l)
  case 0
  then show ?case
    by (simp add: rec [of k l] bit-0)
next
  case (Suc n)
  then show ?case
    by (simp add: rec [of k l] bit-Suc)
qed

end

end

instantiation int :: ring-bit-operations
begin

definition not-int :: ‹int ⇒ int›
  where ‹not-int k = - k - 1›

global-interpretation and-int: fold2-bit-int ‹(∧)›
  defines and-int = and-int.F .

global-interpretation or-int: fold2-bit-int ‹(∨)›
  defines or-int = or-int.F .

global-interpretation xor-int: fold2-bit-int ‹(≠)›
```

defines *xor-int = xor-int.F* **.**

definition *mask-int* :: ‹*nat ⇒ int*›
where ‹*mask n = (2 :: int) ^ n − 1*›

definition *push-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*push-bit-int n k = k * 2 ^ n*›

definition *drop-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*drop-bit-int n k = k div 2 ^ n*›

definition *take-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*take-bit-int n k = k mod 2 ^ n*›

definition *set-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*set-bit n k = k OR push-bit n 1*› **for** *k* :: *int*

definition *unset-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*unset-bit n k = k AND NOT (push-bit n 1)*› **for** *k* :: *int*

definition *flip-bit-int* :: ‹*nat ⇒ int ⇒ int*›
where ‹*flip-bit n k = k XOR push-bit n 1*› **for** *k* :: *int*

lemma *not-int-div-2*:
‹*NOT k div 2 = NOT (k div 2)*› **for** *k* :: *int*
by (*simp add*: *not-int-def*)

lemma *bit-not-int-iff*:
‹*bit (NOT k) n ⟷ ¬ bit k n*› **for** *k* :: *int*
proof (*rule sym*, *induction n arbitrary*: *k*)
case *0*
then show *?case*
by (*simp add*: *bit-0 not-int-def*)
next
case (*Suc n*)
then show *?case*
by (*simp add*: *bit-Suc not-int-div-2*)
qed

instance proof
fix *k l* :: *int* **and** *m n* :: *nat*
show ‹*unset-bit n k = (k OR push-bit n 1) XOR push-bit n 1*›
by (*rule bit-eqI*)
(*auto simp*: *unset-bit-int-def*
and-int.bit-iff or-int.bit-iff xor-int.bit-iff bit-not-int-iff push-bit-int-def bit-simps)
qed (*fact and-int.rec or-int.rec xor-int.rec mask-int-def set-bit-int-def flip-bit-int-def*
push-bit-int-def drop-bit-int-def take-bit-int-def not-int-def)+

end

instance *int* :: *linordered-euclidean-semiring-bit-operations* **..**

context *ring-bit-operations*
begin

lemma *even-of-int-iff*:
‹*even* (*of-int k*) ⟷ *even k*›
by (*induction k rule*: *int-bit-induct*) *simp-all*

lemma *bit-of-int-iff* [*bit-simps*]:
‹*bit* (*of-int k*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ *bit k n*›
proof (*cases* ‹*possible-bit TYPE*('*a*) *n*›)
case *False*
then show *?thesis*
by (*simp add*: *impossible-bit*)
next
case *True*
then have ‹*bit* (*of-int k*) *n* ⟷ *bit k n*›
proof (*induction k arbitrary*: *n rule*: *int-bit-induct*)
case *zero*
then show *?case*
by *simp*
next
case *minus*
then show *?case*
by *simp*
next
case (*even k*)
then show *?case*
using *bit-double-iff* [*of* ‹*of-int k*› *n*] *Bit-Operations.bit-double-iff* [*of k n*]
by (*cases n*) (*auto simp*: *ac-simps possible-bit-def dest*: *mult-not-zero*)
next
case (*odd k*)
then show *?case*
using *bit-double-iff* [*of* ‹*of-int k*› *n*]
by (*cases n*)
(*auto simp*: *ac-simps bit-double-iff even-bit-succ-iff Bit-Operations.bit-0*
Bit-Operations.bit-Suc
possible-bit-def dest: *mult-not-zero*)
qed
with *True* **show** *?thesis*
by *simp*
qed

lemma *push-bit-of-int*:
‹*push-bit n* (*of-int k*) = *of-int* (*push-bit n k*)›
by (*simp add*: *push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *of-int-push-bit*:
‹*of-int* (*push-bit n k*) = *push-bit n* (*of-int k*)›
by (*simp add*: *push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *take-bit-of-int*:
‹*take-bit n* (*of-int k*) = *of-int* (*take-bit n k*)›
by (*rule bit-eqI*) (*simp add*: *bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff*)

lemma *of-int-take-bit*:
‹*of-int* (*take-bit n k*) = *take-bit n* (*of-int k*)›
by (*rule bit-eqI*) (*simp add*: *bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-int-iff*)

lemma *of-int-not-eq*:
‹*of-int* (*NOT k*) = *NOT* (*of-int k*)›
by (*rule bit-eqI*) (*simp add*: *bit-not-iff Bit-Operations.bit-not-iff bit-of-int-iff*)

lemma *of-int-not-numeral*:
‹*of-int* (*NOT* (*numeral k*)) = *NOT* (*numeral k*)›
by (*simp add*: *local.of-int-not-eq*)

lemma *of-int-and-eq*:
‹*of-int* (*k AND l*) = *of-int k AND of-int l*›
by (*rule bit-eqI*) (*simp add*: *bit-of-int-iff bit-and-iff Bit-Operations.bit-and-iff*)

lemma *of-int-or-eq*:
‹*of-int* (*k OR l*) = *of-int k OR of-int l*›
by (*rule bit-eqI*) (*simp add*: *bit-of-int-iff bit-or-iff Bit-Operations.bit-or-iff*)

lemma *of-int-xor-eq*:
‹*of-int* (*k XOR l*) = *of-int k XOR of-int l*›
by (*rule bit-eqI*) (*simp add*: *bit-of-int-iff bit-xor-iff Bit-Operations.bit-xor-iff*)

lemma *of-int-mask-eq*:
‹*of-int* (*mask n*) = *mask n*›
by (*induction n*) (*simp-all add*: *mask-Suc-double Bit-Operations.mask-Suc-double of-int-or-eq*)

end

lemma *take-bit-int-less-exp* [*simp*]:
‹*take-bit n k* < *2* ^ *n*› **for** *k* :: *int*
by (*simp add*: *take-bit-eq-mod*)

lemma *take-bit-int-eq-self-iff*:
‹*take-bit n k* = *k* ⟷ *0* ≤ *k* ∧ *k* < *2* ^ *n*› (**is** ‹*?P* ⟷ *?Q*›)
for *k* :: *int*
proof
assume *?P*
moreover note *take-bit-int-less-exp* [*of n k*] *take-bit-nonnegative* [*of n k*]

ultimately show *?Q*
by *simp*
next
assume *?Q*
then show *?P*
by (*simp add*: *take-bit-eq-mod*)
qed

lemma *take-bit-int-eq-self*:
‹*take-bit n k* = *k*› **if** ‹0 ≤ *k*› ‹*k* < 2 ^ *n*› **for** *k* :: *int*
using *that* **by** (*simp add*: *take-bit-int-eq-self-iff*)

lemma *mask-nonnegative-int* [*simp*]:
‹*mask n* ≥ (0::*int*)›
by (*simp add*: *mask-eq-exp-minus-1 add-le-imp-le-diff*)

lemma *not-mask-negative-int* [*simp*]:
‹¬ *mask n* < (0::*int*)›
by (*simp add*: *not-less*)

lemma *not-nonnegative-int-iff* [*simp*]:
‹*NOT k* ≥ 0 ⟷ *k* < 0› **for** *k* :: *int*
by (*simp add*: *not-int-def*)

lemma *not-negative-int-iff* [*simp*]:
‹*NOT k* < 0 ⟷ *k* ≥ 0› **for** *k* :: *int*
by (*subst Not-eq-iff* [*symmetric*]) (*simp add*: *not-less not-le*)

lemma *and-nonnegative-int-iff* [*simp*]:
‹*k AND l* ≥ 0 ⟷ *k* ≥ 0 ∨ *l* ≥ 0› **for** *k l* :: *int*
proof (*induction k arbitrary*: *l rule*: *int-bit-induct*)
case *zero*
then show *?case*
by *simp*
next
case *minus*
then show *?case*
by *simp*
next
case (*even k*)
then show *?case*
using *and-int.rec* [*of* ‹*k* * 2› *l*]
by (*simp add*: *pos-imp-zdiv-nonneg-iff zero-le-mult-iff*)
next
case (*odd k*)
from *odd* **have** ‹0 ≤ *k AND l div* 2 ⟷ 0 ≤ *k* ∨ 0 ≤ *l div* 2›
by *simp*
then have ‹0 ≤ (1 + *k* * 2) *div* 2 *AND l div* 2 ⟷ 0 ≤ (1 + *k* * 2) *div* 2 ∨
0 ≤ *l div* 2›

```
    by simp
  with and-int.rec [of ‹1 + k * 2› l]
  show ?case
    by (auto simp: zero-le-mult-iff not-le)
qed

lemma and-negative-int-iff [simp]:
  ‹k AND l < 0 ⟷ k < 0 ∧ l < 0› for k l :: int
  by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma and-less-eq:
  ‹k AND l ≤ k› if ‹l < 0› for k l :: int
using that proof (induction k arbitrary: l rule: int-bit-induct)
  case zero
  then show ?case
    by simp
next
  case minus
  then show ?case
    by simp
next
  case (even k)
  from even.IH [of ‹l div 2›] even.hyps even.prems
  show ?case
    by (simp add: and-int.rec [of - l])
next
  case (odd k)
  from odd.IH [of ‹l div 2›] odd.hyps odd.prems
  show ?case
    by (simp add: and-int.rec [of - l])
qed

lemma or-nonnegative-int-iff [simp]:
  ‹k OR l ≥ 0 ⟷ k ≥ 0 ∧ l ≥ 0› for k l :: int
  by (simp only: or-eq-not-not-and not-nonnegative-int-iff) simp

lemma or-negative-int-iff [simp]:
  ‹k OR l < 0 ⟷ k < 0 ∨ l < 0› for k l :: int
  by (subst Not-eq-iff [symmetric]) (simp add: not-less)

lemma or-greater-eq:
  ‹k OR l ≥ k› if ‹l ≥ 0› for k l :: int
using that proof (induction k arbitrary: l rule: int-bit-induct)
  case zero
  then show ?case
    by simp
next
  case minus
  then show ?case
```

```
    by simp
next
  case (even k)
  from even.IH [of ‹l div 2›] even.hyps even.prems
  show ?case
    by (simp add: or-int.rec [of - l])
next
  case (odd k)
  from odd.IH [of ‹l div 2›] odd.hyps odd.prems
  show ?case
    by (simp add: or-int.rec [of - l])
qed

lemma xor-nonnegative-int-iff [simp]:
  ‹k XOR l ≥ 0 ⟷ (k ≥ 0 ⟷ l ≥ 0)› for k l :: int
  by (simp only: bit.xor-def or-nonnegative-int-iff) auto

lemma xor-negative-int-iff [simp]:
  ‹k XOR l < 0 ⟷ (k < 0) ≠ (l < 0)› for k l :: int
  by (subst Not-eq-iff [symmetric]) (auto simp: not-less)

lemma OR-upper:
  ‹x OR y < 2 ^ n› if ‹0 ≤ x› ‹x < 2 ^ n› ‹y < 2 ^ n› for x y :: int
using that proof (induction x arbitrary: y n rule: int-bit-induct)
  case zero
  then show ?case
    by simp
next
  case minus
  then show ?case
    by simp
next
  case (even x)
  from even.IH [of ‹n - 1› ‹y div 2›] even.prems even.hyps
  show ?case
    by (cases n) (auto simp: or-int.rec [of ‹- * 2›] elim: oddE)
next
  case (odd x)
  from odd.IH [of ‹n - 1› ‹y div 2›] odd.prems odd.hyps
  show ?case
    by (cases n) (auto simp: or-int.rec [of ‹1 + - * 2›], linarith)
qed

lemma XOR-upper:
  ‹x XOR y < 2 ^ n› if ‹0 ≤ x› ‹x < 2 ^ n› ‹y < 2 ^ n› for x y :: int
using that proof (induction x arbitrary: y n rule: int-bit-induct)
  case zero
  then show ?case
    by simp
```

```
next
  case minus
  then show ?case
    by simp
next
  case (even x)
  from even.IH [of ‹n − 1› ‹y div 2›] even.prems even.hyps
  show ?case
    by (cases n) (auto simp: xor-int.rec [of ‹- ∗ 2›] elim: oddE)
next
  case (odd x)
  from odd.IH [of ‹n − 1› ‹y div 2›] odd.prems odd.hyps
  show ?case
    by (cases n) (auto simp: xor-int.rec [of ‹1 + - ∗ 2›])
qed

lemma AND-lower [simp]:
  ‹0 ≤ x AND y› if ‹0 ≤ x› for x y :: int
  using that by simp

lemma OR-lower [simp]:
  ‹0 ≤ x OR y› if ‹0 ≤ x› ‹0 ≤ y› for x y :: int
  using that by simp

lemma XOR-lower [simp]:
  ‹0 ≤ x XOR y› if ‹0 ≤ x› ‹0 ≤ y› for x y :: int
  using that by simp

lemma AND-upper1 [simp]:
  ‹x AND y ≤ x› if ‹0 ≤ x› for x y :: int
using that proof (induction x arbitrary: y rule: int-bit-induct)
  case (odd k)
  then have ‹k AND y div 2 ≤ k›
    by simp
  then show ?case
    by (simp add: and-int.rec [of ‹1 + - ∗ 2›])
qed (simp-all add: and-int.rec [of ‹- ∗ 2›])

lemma AND-upper1' [simp]:
  ‹y AND x ≤ z› if ‹0 ≤ y› ‹y ≤ z› for x y z :: int
  using - ‹y ≤ z› by (rule order-trans) (use ‹0 ≤ y› in simp)

lemma AND-upper1'' [simp]:
  ‹y AND x < z› if ‹0 ≤ y› ‹y < z› for x y z :: int
  using - ‹y < z› by (rule order-le-less-trans) (use ‹0 ≤ y› in simp)

lemma AND-upper2 [simp]:
  ‹x AND y ≤ y› if ‹0 ≤ y› for x y :: int
  using that AND-upper1 [of y x] by (simp add: ac-simps)
```

lemma *AND-upper2'* [*simp*]:
  ‹*x AND y* ≤ *z*› **if** ‹*0* ≤ *y*› ‹*y* ≤ *z*› **for** *x y* :: *int*
  using *that AND-upper1'* [*of y z x*] **by** (*simp add*: *ac-simps*)

lemma *AND-upper2''* [*simp*]:
  ‹*x AND y* < *z*› **if** ‹*0* ≤ *y*› ‹*y* < *z*› **for** *x y* :: *int*
  using *that AND-upper1''* [*of y z x*] **by** (*simp add*: *ac-simps*)

lemma *plus-and-or*:
  ‹(*x AND y*) + (*x OR y*) = *x* + *y*› **for** *x y* :: *int*
proof (*induction x arbitrary*: *y rule*: *int-bit-induct*)
  case *zero*
  then show *?case*
    by *simp*
next
  case *minus*
  then show *?case*
    by *simp*
next
  case (*even x*)
  from *even.IH* [*of* ‹*y div 2*›]
  show *?case*
    by (*auto simp*: *and-int.rec* [*of - y*] *or-int.rec* [*of - y*] *elim*: *oddE*)
next
  case (*odd x*)
  from *odd.IH* [*of* ‹*y div 2*›]
  show *?case*
    by (*auto simp*: *and-int.rec* [*of - y*] *or-int.rec* [*of - y*] *elim*: *oddE*)
qed

lemma *push-bit-minus-one*:
  ‹*push-bit n* (− *1* :: *int*) = − (*2* ^ *n*)›
  by (*simp add*: *push-bit-eq-mult*)

lemma *minus-1-div-exp-eq-int*:
  ‹− *1 div* (*2* :: *int*) ^ *n* = − *1*›
  by (*induction n*) (*use div-exp-eq* [*symmetric*, *of* ‹− *1* :: *int*› *1*] **in** ‹*simp-all add*: *ac-simps*›)

lemma *drop-bit-minus-one* [*simp*]:
  ‹*drop-bit n* (− *1* :: *int*) = − *1*›
  by (*simp add*: *drop-bit-eq-div minus-1-div-exp-eq-int*)

lemma *take-bit-minus*:
  ‹*take-bit n* (− *take-bit n k*) = *take-bit n* (− *k*)›
    for *k* :: *int*
  by (*simp add*: *take-bit-eq-mod mod-minus-eq*)

lemma *take-bit-diff*:
  ‹*take-bit n* (*take-bit n k* − *take-bit n l*) = *take-bit n* (*k* − *l*)›
    for *k l* :: *int*
  by (*simp add*: *take-bit-eq-mod mod-diff-eq*)

lemma (**in** *ring-1*) *of-nat-nat-take-bit-eq* [*simp*]:
  ‹*of-nat* (*nat* (*take-bit n k*)) = *of-int* (*take-bit n k*)›
  by *simp*

lemma *take-bit-minus-small-eq*:
  ‹*take-bit n* (− *k*) = 2 ^ *n* − *k*› **if** ‹0 < *k*› ‹*k* ≤ 2 ^ *n*› **for** *k* :: *int*
proof −
  define *m* **where** ‹*m* = *nat k*›
  with *that* **have** ‹*k* = *int m*› **and** ‹0 < *m*› **and** ‹*m* ≤ 2 ^ *n*›
    by *simp-all*
  have ‹(2 ^ *n* − *m*) *mod* 2 ^ *n* = 2 ^ *n* − *m*›
    using ‹0 < *m*› **by** *simp*
  then have ‹*int* ((2 ^ *n* − *m*) *mod* 2 ^ *n*) = *int* (2 ^ *n* − *m*)›
    by *simp*
  then have ‹(2 ^ *n* − *int m*) *mod* 2 ^ *n* = 2 ^ *n* − *int m*›
    using ‹*m* ≤ 2 ^ *n*› **by** (*simp only*: *of-nat-mod of-nat-diff*) *simp*
  with ‹*k* = *int m*› **have** ‹(2 ^ *n* − *k*) *mod* 2 ^ *n* = 2 ^ *n* − *k*›
    by *simp*
  then show *?thesis*
    by (*simp add*: *take-bit-eq-mod*)
qed

lemma *push-bit-nonnegative-int-iff* [*simp*]:
  ‹*push-bit n k* ≥ 0 ⟷ *k* ≥ 0› **for** *k* :: *int*
  by (*simp add*: *push-bit-eq-mult zero-le-mult-iff power-le-zero-eq*)

lemma *push-bit-negative-int-iff* [*simp*]:
  ‹*push-bit n k* < 0 ⟷ *k* < 0› **for** *k* :: *int*
  by (*subst Not-eq-iff* [*symmetric*]) (*simp add*: *not-less*)

lemma *drop-bit-nonnegative-int-iff* [*simp*]:
  ‹*drop-bit n k* ≥ 0 ⟷ *k* ≥ 0› **for** *k* :: *int*
  by (*induction n*) (*auto simp*: *drop-bit-Suc drop-bit-half*)

lemma *drop-bit-negative-int-iff* [*simp*]:
  ‹*drop-bit n k* < 0 ⟷ *k* < 0› **for** *k* :: *int*
  by (*subst Not-eq-iff* [*symmetric*]) (*simp add*: *not-less*)

lemma *set-bit-nonnegative-int-iff* [*simp*]:
  ‹*set-bit n k* ≥ 0 ⟷ *k* ≥ 0› **for** *k* :: *int*
  by (*simp add*: *set-bit-eq-or*)

lemma *set-bit-negative-int-iff* [*simp*]:
  ‹*set-bit n k* < 0 ⟷ *k* < 0› **for** *k* :: *int*

by (*simp add*: *set-bit-eq-or*)

lemma *unset-bit-nonnegative-int-iff* [*simp*]:
‹*unset-bit n k* ≥ *0* ⟷ *k* ≥ *0*› **for** *k* :: *int*
by (*simp add*: *unset-bit-eq-and-not*)

lemma *unset-bit-negative-int-iff* [*simp*]:
‹*unset-bit n k* < *0* ⟷ *k* < *0*› **for** *k* :: *int*
by (*simp add*: *unset-bit-eq-and-not*)

lemma *flip-bit-nonnegative-int-iff* [*simp*]:
‹*flip-bit n k* ≥ *0* ⟷ *k* ≥ *0*› **for** *k* :: *int*
by (*simp add*: *flip-bit-eq-xor*)

lemma *flip-bit-negative-int-iff* [*simp*]:
‹*flip-bit n k* < *0* ⟷ *k* < *0*› **for** *k* :: *int*
by (*simp add*: *flip-bit-eq-xor*)

lemma *set-bit-greater-eq*:
‹*set-bit n k* ≥ *k*› **for** *k* :: *int*
by (*simp add*: *set-bit-eq-or or-greater-eq*)

lemma *unset-bit-less-eq*:
‹*unset-bit n k* ≤ *k*› **for** *k* :: *int*
by (*simp add*: *unset-bit-eq-and-not and-less-eq*)

lemma *and-int-unfold*:
‹*k AND l* = (*if k* = *0* ∨ *l* = *0 then 0 else if k* = − *1 then l else if l* = − *1 then k*
else (*k mod 2*) ∗ (*l mod 2*) + *2* ∗ ((*k div 2*) *AND* (*l div 2*)))› **for** *k l* :: *int*
by (*auto simp*: *and-int.rec* [*of k l*] *zmult-eq-1-iff elim*: *oddE*)

lemma *or-int-unfold*:
‹*k OR l* = (*if k* = − *1* ∨ *l* = − *1 then* − *1 else if k* = *0 then l else if l* = *0 then k*
else max (*k mod 2*) (*l mod 2*) + *2* ∗ ((*k div 2*) *OR* (*l div 2*)))› **for** *k l* :: *int*
by (*auto simp*: *or-int.rec* [*of k l*] *elim*: *oddE*)

lemma *xor-int-unfold*:
‹*k XOR l* = (*if k* = − *1 then NOT l else if l* = − *1 then NOT k else if k* = *0 then l else if l* = *0 then k*
else |*k mod 2* − *l mod 2*| + *2* ∗ ((*k div 2*) *XOR* (*l div 2*)))› **for** *k l* :: *int*
by (*auto simp*: *xor-int.rec* [*of k l*] *not-int-def elim*!: *oddE*)

lemma *bit-minus-int-iff*:
‹*bit* (− *k*) *n* ⟷ *bit* (*NOT* (*k* − *1*)) *n*› **for** *k* :: *int*
by (*simp add*: *bit-simps*)

lemma *take-bit-incr-eq*:
‹*take-bit n* (*k* + *1*) = *1* + *take-bit n k*› **if** ‹*take-bit n k* ≠ *2* ^ *n* − *1*› **for** *k* :: *int*

proof −
from *that* **have** ‹*2 ^ n ≠ k mod 2 ^ n + 1*›
by (*simp add*: *take-bit-eq-mod*)
moreover have ‹*k mod 2 ^ n < 2 ^ n*›
by *simp*
ultimately have ∗: ‹*k mod 2 ^ n + 1 < 2 ^ n*›
by *linarith*
have ‹*(k + 1) mod 2 ^ n = (k mod 2 ^ n + 1) mod 2 ^ n*›
by (*simp add*: *mod-simps*)
also have ‹*... = k mod 2 ^ n + 1*›
using ∗ **by** (*simp add*: *zmod-trivial-iff*)
finally have ‹*(k + 1) mod 2 ^ n = k mod 2 ^ n + 1*› **.**
then show *?thesis*
by (*simp add*: *take-bit-eq-mod*)
qed

lemma *take-bit-decr-eq*:
‹*take-bit n (k − 1) = take-bit n k − 1*› **if** ‹*take-bit n k ≠ 0*› **for** *k* :: *int*
proof −
from *that* **have** ‹*k mod 2 ^ n ≠ 0*›
by (*simp add*: *take-bit-eq-mod*)
moreover have ‹*k mod 2 ^ n ≥ 0*› ‹*k mod 2 ^ n < 2 ^ n*›
by *simp-all*
ultimately have ∗: ‹*k mod 2 ^ n > 0*›
by *linarith*
have ‹*(k − 1) mod 2 ^ n = (k mod 2 ^ n − 1) mod 2 ^ n*›
by (*simp add*: *mod-simps*)
also have ‹*... = k mod 2 ^ n − 1*›
by (*simp add*: *zmod-trivial-iff*)
(*use* ‹*k mod 2 ^ n < 2 ^ n*› ∗ **in** *linarith*)
finally have ‹*(k − 1) mod 2 ^ n = k mod 2 ^ n − 1*› **.**
then show *?thesis*
by (*simp add*: *take-bit-eq-mod*)
qed

lemma *take-bit-int-greater-eq*:
‹*k + 2 ^ n ≤ take-bit n k*› **if** ‹*k < 0*› **for** *k* :: *int*
proof −
have ‹*k + 2 ^ n ≤ take-bit n (k + 2 ^ n)*›
proof (*cases* ‹*k > − (2 ^ n)*›)
case *False*
then have ‹*k + 2 ^ n ≤ 0*›
by *simp*
also note *take-bit-nonnegative*
finally show *?thesis* **.**
next
case *True*
with *that* **have** ‹*0 ≤ k + 2 ^ n*› **and** ‹*k + 2 ^ n < 2 ^ n*›
by *simp-all*

```
    then show ?thesis
      by (simp only: take-bit-eq-mod mod-pos-pos-trivial)
  qed
  then show ?thesis
    by (simp add: take-bit-eq-mod)
qed

lemma take-bit-int-less-eq:
  ‹take-bit n k ≤ k − 2 ^ n› if ‹2 ^ n ≤ k› and ‹n > 0› for k :: int
  using that zmod-le-nonneg-dividend [of ‹k − 2 ^ n› ‹2 ^ n›]
  by (simp add: take-bit-eq-mod)

lemma take-bit-int-less-eq-self-iff:
  ‹take-bit n k ≤ k ⟷ 0 ≤ k› (is ‹?P ⟷ ?Q›) for k :: int
proof
  assume ?P
  show ?Q
  proof (rule ccontr)
    assume ‹¬ 0 ≤ k›
    then have ‹k < 0›
      by simp
    with ‹?P›
    have ‹take-bit n k < 0›
      by (rule le-less-trans)
    then show False
      by simp
  qed
next
  assume ?Q
  then show ?P
    by (simp add: take-bit-eq-mod zmod-le-nonneg-dividend)
qed

lemma take-bit-int-less-self-iff:
  ‹take-bit n k < k ⟷ 2 ^ n ≤ k› for k :: int
  by (auto simp: less-le take-bit-int-less-eq-self-iff take-bit-int-eq-self-iff
    intro: order-trans [of 0 ‹2 ^ n› k])

lemma take-bit-int-less-eq-mask:
  ‹take-bit n k ≤ mask n› for k :: int
  using take-bit-int-less-exp [of n k] by (simp add: mask-eq-exp-minus-1)

lemma take-bit-int-greater-self-iff:
  ‹k < take-bit n k ⟷ k < 0› for k :: int
  using take-bit-int-less-eq-self-iff [of n k] by auto

lemma take-bit-int-greater-eq-self-iff:
  ‹k ≤ take-bit n k ⟷ k < 2 ^ n› for k :: int
  by (auto simp: le-less take-bit-int-greater-self-iff take-bit-int-eq-self-iff
```

```
    dest: sym not-sym intro: less-trans [of k 0 ‹2 ^ n›])

lemma take-bit-tightened-less-eq-int:
  ‹take-bit m k ≤ take-bit n k› if ‹m ≤ n› for k :: int
proof −
  have ‹take-bit m (take-bit n k) ≤ take-bit n k›
    by (simp only: take-bit-int-less-eq-self-iff take-bit-nonnegative)
  with that show ?thesis
    by simp
qed

lemma not-exp-less-eq-0-int [simp]:
  ‹¬ 2 ^ n ≤ (0::int)›
  by (simp add: power-le-zero-eq)

lemma int-bit-bound:
  fixes k :: int
  obtains n where ‹⋀m. n ≤ m ⟹ bit k m ⟷ bit k n›
    and ‹n > 0 ⟹ bit k (n − 1) ≠ bit k n›
proof −
  obtain q where ∗: ‹⋀m. q ≤ m ⟹ bit k m ⟷ bit k q›
  proof (cases ‹k ≥ 0›)
    case True
    moreover from power-gt-expt [of 2 ‹nat k›]
    have ‹nat k < 2 ^ nat k›
      by simp
    then have ‹int (nat k) < int (2 ^ nat k)›
      by (simp only: of-nat-less-iff)
    ultimately have ∗: ‹k div 2 ^ nat k = 0›
      by simp
    show thesis
    proof (rule that [of ‹nat k›])
      fix m
      assume ‹nat k ≤ m›
      then show ‹bit k m ⟷ bit k (nat k)›
        by (auto simp: ∗ bit-iff-odd power-add zdiv-zmult2-eq dest!: le-Suc-ex)
    qed
  next
    case False
    moreover from power-gt-expt [of 2 ‹nat (− k)›]
    have ‹nat (− k) < 2 ^ nat (− k)›
      by simp
    then have ‹int (nat (− k)) < int (2 ^ nat (− k))›
      by (simp only: of-nat-less-iff)
    ultimately have ‹− k div − (2 ^ nat (− k)) = − 1›
      by (subst div-pos-neg-trivial) simp-all
    then have ∗: ‹k div 2 ^ nat (− k) = − 1›
      by simp
    show thesis
```

proof (*rule that* [*of* ‹*nat* (− *k*)›])
fix *m*
assume ‹*nat* (− *k*) ≤ *m*›
then show ‹*bit k m* ⟷ *bit k* (*nat* (− *k*))›
by (*auto simp*: ∗ *bit-iff-odd power-add zdiv-zmult2-eq minus-1-div-exp-eq-int dest*!: *le-Suc-ex*)
qed
qed
show *thesis*
proof (*cases* ‹∀ *m*. *bit k m* ⟷ *bit k q*›)
case *True*
then have ‹*bit k 0* ⟷ *bit k q*›
by *blast*
with *True that* [*of 0*] **show** *thesis*
by *simp*
next
case *False*
then obtain *r* **where** ∗∗: ‹*bit k r* ≠ *bit k q*›
by *blast*
have ‹*r* < *q*›
by (*rule ccontr*) (*use* ∗ [*of r*] ∗∗ **in** *simp*)
define *N* **where** ‹*N* = {*n*. *n* < *q* ∧ *bit k n* ≠ *bit k q*}›
moreover have ‹*finite N*› ‹*r* ∈ *N*›
using ∗∗ *N-def* ‹*r* < *q*› **by** *auto*
moreover define *n* **where** ‹*n* = *Suc* (*Max N*)›
ultimately have †: ‹⋀*m*. *n* ≤ *m* ⟹ *bit k m* ⟷ *bit k n*›
by (*smt* (*verit*) ∗ *Max-ge Suc-n-not-le-n linorder-not-less mem-Collect-eq not-less-eq-eq*)
have ‹*bit k* (*Max N*) ≠ *bit k n*›
by (*metis* (*mono-tags*, *lifting*) ∗ *Max-in N-def* ‹⋀*m*. *n* ≤ *m* ⟹ *bit k m* = *bit k n*› ‹*finite N*› ‹*r* ∈ *N*› *empty-iff le-cases mem-Collect-eq*)
with † *n-def that* [*of n*] **show** *thesis*
by *fastforce*
qed
qed

68.5 Instance *nat*

instantiation *nat* :: *semiring-bit-operations*
begin

definition *and-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*m AND n* = *nat* (*int m AND int n*)› **for** *m n* :: *nat*

definition *or-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*m OR n* = *nat* (*int m OR int n*)› **for** *m n* :: *nat*

definition *xor-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*m XOR n* = *nat* (*int m XOR int n*)› **for** *m n* :: *nat*

definition *mask-nat* :: ‹*nat* ⇒ *nat*›
where ‹*mask n* = (*2* :: *nat*) ^ *n* − *1*›

definition *push-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*push-bit-nat n m* = *m* ∗ *2* ^ *n*›

definition *drop-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*drop-bit-nat n m* = *m div 2* ^ *n*›

definition *take-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*take-bit-nat n m* = *m mod 2* ^ *n*›

definition *set-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*set-bit m n* = *n OR push-bit m 1*› **for** *m n* :: *nat*

definition *unset-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*unset-bit m n* = (*n OR push-bit m 1*) *XOR push-bit m 1*› **for** *m n* :: *nat*

definition *flip-bit-nat* :: ‹*nat* ⇒ *nat* ⇒ *nat*›
where ‹*flip-bit m n* = *n XOR push-bit m 1*› **for** *m n* :: *nat*

instance proof
fix *m n* :: *nat*
show ‹*m AND n* = *of-bool* (*odd m* ∧ *odd n*) + *2* ∗ (*m div 2 AND n div 2*)›
by (*simp add*: *and-nat-def and-rec* [*of* ‹*int m*› ‹*int n*›] *nat-add-distrib of-nat-div*)
show ‹*m OR n* = *of-bool* (*odd m* ∨ *odd n*) + *2* ∗ (*m div 2 OR n div 2*)›
by (*simp add*: *or-nat-def or-rec* [*of* ‹*int m*› ‹*int n*›] *nat-add-distrib of-nat-div*)
show ‹*m XOR n* = *of-bool* (*odd m* ≠ *odd n*) + *2* ∗ (*m div 2 XOR n div 2*)›
by (*simp add*: *xor-nat-def xor-rec* [*of* ‹*int m*› ‹*int n*›] *nat-add-distrib of-nat-div*)
qed (*simp-all add*: *mask-nat-def set-bit-nat-def unset-bit-nat-def flip-bit-nat-def push-bit-nat-def drop-bit-nat-def take-bit-nat-def*)

end

instance *nat* :: *linordered-euclidean-semiring-bit-operations* **..**

context *semiring-bit-operations*
begin

lemma *push-bit-of-nat*:
‹*push-bit n* (*of-nat m*) = *of-nat* (*push-bit n m*)›
by (*simp add*: *push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *of-nat-push-bit*:
‹*of-nat* (*push-bit m n*) = *push-bit m* (*of-nat n*)›
by (*simp add*: *push-bit-eq-mult Bit-Operations.push-bit-eq-mult*)

lemma *take-bit-of-nat*:

‹*take-bit n* (*of-nat m*) = *of-nat* (*take-bit n m*)›
by (*rule bit-eqI*) (*simp add*: *bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff*)

lemma *of-nat-take-bit*:
‹*of-nat* (*take-bit n m*) = *take-bit n* (*of-nat m*)›
by (*rule bit-eqI*) (*simp add*: *bit-take-bit-iff Bit-Operations.bit-take-bit-iff bit-of-nat-iff*)

lemma *of-nat-and-eq*:
‹*of-nat* (*m AND n*) = *of-nat m AND of-nat n*›
by (*rule bit-eqI*) (*simp add*: *bit-of-nat-iff bit-and-iff Bit-Operations.bit-and-iff*)

lemma *of-nat-or-eq*:
‹*of-nat* (*m OR n*) = *of-nat m OR of-nat n*›
by (*rule bit-eqI*) (*simp add*: *bit-of-nat-iff bit-or-iff Bit-Operations.bit-or-iff*)

lemma *of-nat-xor-eq*:
‹*of-nat* (*m XOR n*) = *of-nat m XOR of-nat n*›
by (*rule bit-eqI*) (*simp add*: *bit-of-nat-iff bit-xor-iff Bit-Operations.bit-xor-iff*)

lemma *of-nat-mask-eq*:
‹*of-nat* (*mask n*) = *mask n*›
by (*induction n*) (*simp-all add*: *mask-Suc-double Bit-Operations.mask-Suc-double of-nat-or-eq*)

lemma *of-nat-set-bit-eq*:
‹*of-nat* (*set-bit n m*) = *set-bit n* (*of-nat m*)›
by (*simp add*: *set-bit-eq-or Bit-Operations.set-bit-eq-or of-nat-or-eq Bit-Operations.push-bit-eq-mult*)

lemma *of-nat-unset-bit-eq*:
‹*of-nat* (*unset-bit n m*) = *unset-bit n* (*of-nat m*)›
by (*simp add*: *unset-bit-eq-or-xor Bit-Operations.unset-bit-eq-or-xor of-nat-or-eq of-nat-xor-eq Bit-Operations.push-bit-eq-mult*)

lemma *of-nat-flip-bit-eq*:
‹*of-nat* (*flip-bit n m*) = *flip-bit n* (*of-nat m*)›
by (*simp add*: *flip-bit-eq-xor Bit-Operations.flip-bit-eq-xor of-nat-xor-eq Bit-Operations.push-bit-eq-mult*)

end

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *drop-bit-of-nat*:
drop-bit n (*of-nat m*) = *of-nat* (*drop-bit n m*)
by (*simp add*: *drop-bit-eq-div Bit-Operations.drop-bit-eq-div of-nat-div* [*of m 2* ^ *n*])

lemma *of-nat-drop-bit*:
‹*of-nat* (*drop-bit m n*) = *drop-bit m* (*of-nat n*)›

by (*simp add*: *drop-bit-eq-div Bit-Operations.drop-bit-eq-div of-nat-div*)

end

lemma *take-bit-nat-less-exp* [*simp*]:
‹*take-bit n m* < *2* ^ *n*› **for** *n m* :: *nat*
by (*simp add*: *take-bit-eq-mod*)

lemma *take-bit-nat-eq-self-iff*:
‹*take-bit n m* = *m* ⟷ *m* < *2* ^ *n*› (**is** ‹*?P* ⟷ *?Q*›) **for** *n m* :: *nat*
proof
assume *?P*
moreover note *take-bit-nat-less-exp* [*of n m*]
ultimately show *?Q*
by *simp*
next
assume *?Q*
then show *?P*
by (*simp add*: *take-bit-eq-mod*)
qed

lemma *take-bit-nat-eq-self*:
‹*take-bit n m* = *m*› **if** ‹*m* < *2* ^ *n*› **for** *m n* :: *nat*
using *that* **by** (*simp add*: *take-bit-nat-eq-self-iff*)

lemma *take-bit-nat-less-eq-self* [*simp*]:
‹*take-bit n m* ≤ *m*› **for** *n m* :: *nat*
by (*simp add*: *take-bit-eq-mod*)

lemma *take-bit-nat-less-self-iff*:
‹*take-bit n m* < *m* ⟷ *2* ^ *n* ≤ *m*› (**is** ‹*?P* ⟷ *?Q*›) **for** *m n* :: *nat*
proof
assume *?P*
then have ‹*take-bit n m* ≠ *m*›
by *simp*
then show ‹*?Q*›
by (*simp add*: *take-bit-nat-eq-self-iff*)
next
have ‹*take-bit n m* < *2* ^ *n*›
by (*fact take-bit-nat-less-exp*)
also assume *?Q*
finally show *?P* **.**
qed

lemma *Suc-0-and-eq* [*simp*]:
‹*Suc 0 AND n* = *n mod 2*›
using *one-and-eq* [*of n*] **by** *simp*

lemma *and-Suc-0-eq* [*simp*]:

‹*n AND Suc 0 = n mod 2*›
using *and-one-eq* [*of n*] **by** *simp*

lemma *Suc-0-or-eq*:
‹*Suc 0 OR n = n + of-bool (even n)*›
using *one-or-eq* [*of n*] **by** *simp*

lemma *or-Suc-0-eq*:
‹*n OR Suc 0 = n + of-bool (even n)*›
using *or-one-eq* [*of n*] **by** *simp*

lemma *Suc-0-xor-eq*:
‹*Suc 0 XOR n = n + of-bool (even n) − of-bool (odd n)*›
using *one-xor-eq* [*of n*] **by** *simp*

lemma *xor-Suc-0-eq*:
‹*n XOR Suc 0 = n + of-bool (even n) − of-bool (odd n)*›
using *xor-one-eq* [*of n*] **by** *simp*

lemma *and-nat-unfold* [*code*]:
‹*m AND n = (if m = 0 ∨ n = 0 then 0 else (m mod 2) ∗ (n mod 2) + 2 ∗ ((m div 2) AND (n div 2)))*›
for *m n* :: *nat*
by (*auto simp*: *and-rec* [*of m n*] *elim*: *oddE*)

lemma *or-nat-unfold* [*code*]:
‹*m OR n = (if m = 0 then n else if n = 0 then m*
else max (m mod 2) (n mod 2) + 2 ∗ ((m div 2) OR (n div 2)))› **for** *m n* :: *nat*
by (*auto simp*: *or-rec* [*of m n*] *elim*: *oddE*)

lemma *xor-nat-unfold* [*code*]:
‹*m XOR n = (if m = 0 then n else if n = 0 then m*
else (m mod 2 + n mod 2) mod 2 + 2 ∗ ((m div 2) XOR (n div 2)))› **for** *m n* :: *nat*
by (*auto simp*: *xor-rec* [*of m n*] *elim*!: *oddE*)

lemma [*code*]:
‹*unset-bit 0 m = 2 ∗ (m div 2)*›
‹*unset-bit (Suc n) m = m mod 2 + 2 ∗ unset-bit n (m div 2)*› **for** *m n* :: *nat*
by (*simp-all add*: *unset-bit-0 unset-bit-Suc*)

lemma *push-bit-of-Suc-0* [*simp*]:
‹*push-bit n (Suc 0) = 2 ^ n*›
using *push-bit-of-1* [**where** *?'a = nat*] **by** *simp*

lemma *take-bit-of-Suc-0* [*simp*]:
‹*take-bit n (Suc 0) = of-bool (0 < n)*›
using *take-bit-of-1* [**where** *?'a = nat*] **by** *simp*

lemma *drop-bit-of-Suc-0* [*simp*]:
  ‹*drop-bit n* (*Suc 0*) = *of-bool* (*n* = *0*)›
  using *drop-bit-of-1* [**where** *?'a* = *nat*] **by** *simp*

lemma *Suc-mask-eq-exp*:
  ‹*Suc* (*mask n*) = *2* ^ *n*›
  by (*simp add*: *mask-eq-exp-minus-1*)

lemma *less-eq-mask*:
  ‹*n* ≤ *mask n*›
  by (*simp add*: *mask-eq-exp-minus-1 le-diff-conv2*)
    (*metis Suc-mask-eq-exp diff-Suc-1 diff-le-diff-pow diff-zero le-refl not-less-eq-eq power-0*)

lemma *less-mask*:
  ‹*n* < *mask n*› **if** ‹*Suc 0* < *n*›
proof −
  define *m* **where** ‹*m* = *n* − *2*›
  with *that* **have** ∗: ‹*n* = *m* + *2*›
    by *simp*
  have ‹*Suc* (*Suc* (*Suc m*)) < *4* ∗ *2* ^ *m*›
    by (*induction m*) *simp-all*
  then have ‹*Suc* (*m* + *2*) < *Suc* (*mask* (*m* + *2*))›
    by (*simp add*: *Suc-mask-eq-exp*)
  then have ‹*m* + *2* < *mask* (*m* + *2*)›
    by (*simp add*: *less-le*)
  with ∗ **show** *?thesis*
    by *simp*
qed

lemma *mask-nat-less-exp* [*simp*]:
  ‹(*mask n* :: *nat*) < *2* ^ *n*›
  by (*simp add*: *mask-eq-exp-minus-1*)

lemma *mask-nat-positive-iff* [*simp*]:
  ‹(*0*::*nat*) < *mask n* ⟷ *0* < *n*›
proof (*cases* ‹*n* = *0*›)
  case *True*
  then show *?thesis*
    by *simp*
next
  case *False*
  then have ‹*0* < *n*›
    by *simp*
  then have ‹(*0*::*nat*) < *mask n*›
    using *less-eq-mask* [*of n*] **by** (*rule order-less-le-trans*)
  with ‹*0* < *n*› **show** *?thesis*
    by *simp*

qed

lemma *take-bit-nat-less-eq-mask*:
‹*take-bit n m* ≤ *mask n*› **for** *m* :: *nat*
using *take-bit-nat-less-exp* [*of n m*]
by (*simp flip*: *Suc-mask-eq-exp*)

lemma *take-bit-tightened-less-eq-nat*:
‹*take-bit m q* ≤ *take-bit n q*› **if** ‹*m* ≤ *n*› **for** *q* :: *nat*
proof −
have ‹*take-bit m* (*take-bit n q*) ≤ *take-bit n q*›
by (*rule take-bit-nat-less-eq-self*)
with *that* **show** *?thesis*
by *simp*
qed

lemma *push-bit-nat-eq*:
‹*push-bit n* (*nat k*) = *nat* (*push-bit n k*)›
by (*cases* ‹*k* ≥ *0*›) (*simp-all add*: *push-bit-eq-mult nat-mult-distrib not-le mult-nonneg-nonpos2*)

lemma *drop-bit-nat-eq*:
‹*drop-bit n* (*nat k*) = *nat* (*drop-bit n k*)›
proof (*cases* ‹*k* ≥ *0*›)
case *True*
then show *?thesis*
by (*metis drop-bit-of-nat int-nat-eq nat-int*)
qed (*simp add*: *nat-eq-iff2*)

lemma *take-bit-nat-eq*:
‹*take-bit n* (*nat k*) = *nat* (*take-bit n k*)› **if** ‹*k* ≥ *0*›
using *that* **by** (*simp add*: *take-bit-eq-mod nat-mod-distrib nat-power-eq*)

lemma *nat-take-bit-eq*:
‹*nat* (*take-bit n k*) = *take-bit n* (*nat k*)›
if ‹*k* ≥ *0*›
using *that* **by** (*simp add*: *take-bit-eq-mod nat-mod-distrib nat-power-eq*)

lemma *nat-mask-eq*:
‹*nat* (*mask n*) = *mask n*›
by (*simp add*: *nat-eq-iff of-nat-mask-eq*)

68.6 Symbolic computations on numeral expressions

context *semiring-bits*
begin

lemma *bit-1-0* [*simp*]:
‹*bit 1 0*›
by (*simp add*: *bit-0*)

lemma *not-bit-1-Suc* [*simp*]:
‹¬ *bit 1* (*Suc n*)›
by (*simp add*: *bit-Suc*)

lemma *not-bit-1-numeral* [*simp*]:
‹¬ *bit 1* (*numeral m*)›
by (*simp add*: *numeral-eq-Suc*)

lemma *not-bit-numeral-Bit0-0* [*simp*]:
‹¬ *bit* (*numeral* (*Num.Bit0 m*)) *0*›
by (*simp add*: *bit-0*)

lemma *bit-numeral-Bit1-0* [*simp*]:
‹*bit* (*numeral* (*Num.Bit1 m*)) *0*›
by (*simp add*: *bit-0*)

lemma *bit-numeral-Bit0-iff*:
‹*bit* (*numeral* (*num.Bit0 m*)) *n*
⟷ *possible-bit TYPE*('*a*) *n* ∧ *n* > *0* ∧ *bit* (*numeral m*) (*n* − *1*)›
by (*simp only*: *numeral-Bit0-eq-double* [*of m*] *bit-simps*) *simp*

lemma *bit-numeral-Bit1-Suc-iff*:
‹*bit* (*numeral* (*num.Bit1 m*)) (*Suc n*)
⟷ *possible-bit TYPE*('*a*) (*Suc n*) ∧ *bit* (*numeral m*) *n*›
using *even-bit-succ-iff* [*of* ‹*2* ∗ *numeral m*› ‹*Suc n*›]
by (*simp only*: *numeral-Bit1-eq-inc-double* [*of m*] *bit-simps*) *simp*

end

context *ring-bit-operations*
begin

lemma *not-bit-minus-numeral-Bit0-0* [*simp*]:
‹¬ *bit* (− *numeral* (*Num.Bit0 m*)) *0*›
by (*simp add*: *bit-0*)

lemma *bit-minus-numeral-Bit1-0* [*simp*]:
‹*bit* (− *numeral* (*Num.Bit1 m*)) *0*›
by (*simp add*: *bit-0*)

lemma *bit-minus-numeral-Bit0-Suc-iff*:
‹*bit* (− *numeral* (*num.Bit0 m*)) (*Suc n*)
⟷ *possible-bit TYPE*('*a*) (*Suc n*) ∧ *bit* (− *numeral m*) *n*›
by (*simp only*: *numeral-Bit0-eq-double* [*of m*] *minus-mult-right bit-simps*) *auto*

lemma *bit-minus-numeral-Bit1-Suc-iff*:
‹*bit* (− *numeral* (*num.Bit1 m*)) (*Suc n*)
⟷ *possible-bit TYPE*('*a*) (*Suc n*) ∧ ¬ *bit* (*numeral m*) *n*›

by (*simp only*: *numeral-Bit1-eq-inc-double* [*of m*] *minus-add-distrib minus-mult-right add-uminus-conv-diff*
bit-decr-iff bit-double-iff)
auto

lemma *bit-numeral-BitM-0* [*simp*]:
‹*bit* (*numeral* (*Num.BitM m*)) *0*›
by (*simp only*: *numeral-BitM bit-decr-iff not-bit-minus-numeral-Bit0-0*) *simp*

lemma *bit-numeral-BitM-Suc-iff*:
‹*bit* (*numeral* (*Num.BitM m*)) (*Suc n*) ⟷ *possible-bit TYPE*('*a*) (*Suc n*) ∧ ¬ *bit* (− *numeral m*) *n*›
by (*simp-all only*: *numeral-BitM bit-decr-iff bit-minus-numeral-Bit0-Suc-iff*) *auto*

end

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *bit-numeral-iff*:
‹*bit* (*numeral m*) *n* ⟷ *bit* (*numeral m* :: *nat*) *n*›
using *bit-of-nat-iff-bit* [*of* ‹*numeral m*› *n*] **by** *simp*

lemma *bit-numeral-Bit0-Suc-iff* [*simp*]:
‹*bit* (*numeral* (*Num.Bit0 m*)) (*Suc n*) ⟷ *bit* (*numeral m*) *n*›
by (*simp add*: *bit-Suc numeral-Bit0-div-2*)

lemma *bit-numeral-Bit1-Suc-iff* [*simp*]:
‹*bit* (*numeral* (*Num.Bit1 m*)) (*Suc n*) ⟷ *bit* (*numeral m*) *n*›
by (*simp add*: *bit-Suc numeral-Bit0-div-2*)

lemma *bit-numeral-rec*:
‹*bit* (*numeral* (*Num.Bit0 w*)) *n* ⟷ (*case n of 0* ⇒ *False* | *Suc m* ⇒ *bit* (*numeral w*) *m*)›
‹*bit* (*numeral* (*Num.Bit1 w*)) *n* ⟷ (*case n of 0* ⇒ *True* | *Suc m* ⇒ *bit* (*numeral w*) *m*)›
by (*cases n*; *simp add*: *bit-0*)+

lemma *bit-numeral-simps* [*simp*]:
‹*bit* (*numeral* (*Num.Bit0 w*)) (*numeral n*) ⟷ *bit* (*numeral w*) (*pred-numeral n*)›
‹*bit* (*numeral* (*Num.Bit1 w*)) (*numeral n*) ⟷ *bit* (*numeral w*) (*pred-numeral n*)›
by (*simp-all add*: *bit-1-iff numeral-eq-Suc*)

lemma *and-numerals* [*simp*]:
‹*1 AND numeral* (*Num.Bit0 y*) = *0*›
‹*1 AND numeral* (*Num.Bit1 y*) = *1*›
‹*numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit0 y*) = *2* ∗ (*numeral x AND numeral y*)›
‹*numeral* (*Num.Bit0 x*) *AND numeral* (*Num.Bit1 y*) = *2* ∗ (*numeral x AND*

numeral y)›
‹*numeral (Num.Bit0 x) AND 1 = 0*›
‹*numeral (Num.Bit1 x) AND numeral (Num.Bit0 y) = 2 * (numeral x AND numeral y)*›
‹*numeral (Num.Bit1 x) AND numeral (Num.Bit1 y) = 1 + 2 * (numeral x AND numeral y)*›
‹*numeral (Num.Bit1 x) AND 1 = 1*›
by (*simp-all add*: *bit-eq-iff*) (*simp-all add*: *bit-0 bit-simps bit-Suc bit-numeral-rec split*: *nat.splits*)

lemma *or-numerals* [*simp*]:
‹*1 OR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)*›
‹*1 OR numeral (Num.Bit1 y) = numeral (Num.Bit1 y)*›
‹*numeral (Num.Bit0 x) OR numeral (Num.Bit0 y) = 2 * (numeral x OR numeral y)*›
‹*numeral (Num.Bit0 x) OR numeral (Num.Bit1 y) = 1 + 2 * (numeral x OR numeral y)*›
‹*numeral (Num.Bit0 x) OR 1 = numeral (Num.Bit1 x)*›
‹*numeral (Num.Bit1 x) OR numeral (Num.Bit0 y) = 1 + 2 * (numeral x OR numeral y)*›
‹*numeral (Num.Bit1 x) OR numeral (Num.Bit1 y) = 1 + 2 * (numeral x OR numeral y)*›
‹*numeral (Num.Bit1 x) OR 1 = numeral (Num.Bit1 x)*›
by (*simp-all add*: *bit-eq-iff*) (*simp-all add*: *bit-0 bit-simps bit-Suc bit-numeral-rec split*: *nat.splits*)

lemma *xor-numerals* [*simp*]:
‹*1 XOR numeral (Num.Bit0 y) = numeral (Num.Bit1 y)*›
‹*1 XOR numeral (Num.Bit1 y) = numeral (Num.Bit0 y)*›
‹*numeral (Num.Bit0 x) XOR numeral (Num.Bit0 y) = 2 * (numeral x XOR numeral y)*›
‹*numeral (Num.Bit0 x) XOR numeral (Num.Bit1 y) = 1 + 2 * (numeral x XOR numeral y)*›
‹*numeral (Num.Bit0 x) XOR 1 = numeral (Num.Bit1 x)*›
‹*numeral (Num.Bit1 x) XOR numeral (Num.Bit0 y) = 1 + 2 * (numeral x XOR numeral y)*›
‹*numeral (Num.Bit1 x) XOR numeral (Num.Bit1 y) = 2 * (numeral x XOR numeral y)*›
‹*numeral (Num.Bit1 x) XOR 1 = numeral (Num.Bit0 x)*›
by (*simp-all add*: *bit-eq-iff*) (*simp-all add*: *bit-0 bit-simps bit-Suc bit-numeral-rec split*: *nat.splits*)

end

lemma *drop-bit-Suc-minus-bit0* [*simp*]:
‹*drop-bit (Suc n) (− numeral (Num.Bit0 k)) = drop-bit n (− numeral k :: int)*›
by (*simp add*: *drop-bit-Suc numeral-Bit0-div-2*)

lemma *drop-bit-Suc-minus-bit1* [*simp*]:

‹*drop-bit* (*Suc n*) (− *numeral* (*Num.Bit1 k*)) = *drop-bit n* (− *numeral* (*Num.inc k*) :: *int*)›
by (*simp add*: *drop-bit-Suc numeral-Bit1-div-2 add-One*)

lemma *drop-bit-numeral-minus-bit0* [*simp*]:
‹*drop-bit* (*numeral l*) (− *numeral* (*Num.Bit0 k*)) = *drop-bit* (*pred-numeral l*) (− *numeral k* :: *int*)›
by (*simp add*: *numeral-eq-Suc numeral-Bit0-div-2*)

lemma *drop-bit-numeral-minus-bit1* [*simp*]:
‹*drop-bit* (*numeral l*) (− *numeral* (*Num.Bit1 k*)) = *drop-bit* (*pred-numeral l*) (− *numeral* (*Num.inc k*) :: *int*)›
by (*simp add*: *numeral-eq-Suc numeral-Bit1-div-2*)

lemma *take-bit-Suc-minus-bit0*:
‹*take-bit* (*Suc n*) (− *numeral* (*Num.Bit0 k*)) = *take-bit n* (− *numeral k*) ∗ (*2* :: *int*)›
by (*simp add*: *take-bit-Suc numeral-Bit0-div-2*)

lemma *take-bit-Suc-minus-bit1*:
‹*take-bit* (*Suc n*) (− *numeral* (*Num.Bit1 k*)) = *take-bit n* (− *numeral* (*Num.inc k*)) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *take-bit-Suc numeral-Bit1-div-2 add-One*)

lemma *take-bit-numeral-minus-bit0*:
‹*take-bit* (*numeral l*) (− *numeral* (*Num.Bit0 k*)) = *take-bit* (*pred-numeral l*) (− *numeral k*) ∗ (*2* :: *int*)›
by (*simp add*: *numeral-eq-Suc numeral-Bit0-div-2 take-bit-Suc-minus-bit0*)

lemma *take-bit-numeral-minus-bit1*:
‹*take-bit* (*numeral l*) (− *numeral* (*Num.Bit1 k*)) = *take-bit* (*pred-numeral l*) (− *numeral* (*Num.inc k*)) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *numeral-eq-Suc numeral-Bit1-div-2 take-bit-Suc-minus-bit1*)

lemma *and-nat-numerals* [*simp*]:
‹*Suc 0 AND numeral* (*Num.Bit0 y*) = *0*›
‹*Suc 0 AND numeral* (*Num.Bit1 y*) = *1*›
‹*numeral* (*Num.Bit0 x*) *AND Suc 0* = *0*›
‹*numeral* (*Num.Bit1 x*) *AND Suc 0* = *1*›
by (*simp-all only*: *and-numerals flip*: *One-nat-def*)

lemma *or-nat-numerals* [*simp*]:
‹*Suc 0 OR numeral* (*Num.Bit0 y*) = *numeral* (*Num.Bit1 y*)›
‹*Suc 0 OR numeral* (*Num.Bit1 y*) = *numeral* (*Num.Bit1 y*)›
‹*numeral* (*Num.Bit0 x*) *OR Suc 0* = *numeral* (*Num.Bit1 x*)›
‹*numeral* (*Num.Bit1 x*) *OR Suc 0* = *numeral* (*Num.Bit1 x*)›
by (*simp-all only*: *or-numerals flip*: *One-nat-def*)

lemma *xor-nat-numerals* [*simp*]:

‹*Suc 0 XOR numeral* (*Num.Bit0 y*) = *numeral* (*Num.Bit1 y*)›
‹*Suc 0 XOR numeral* (*Num.Bit1 y*) = *numeral* (*Num.Bit0 y*)›
‹*numeral* (*Num.Bit0 x*) *XOR Suc 0* = *numeral* (*Num.Bit1 x*)›
‹*numeral* (*Num.Bit1 x*) *XOR Suc 0* = *numeral* (*Num.Bit0 x*)›
by (*simp only*: *xor-numerals flip*: *One-nat-def*)

context *ring-bit-operations*
begin

lemma *minus-numeral-inc-eq*:
‹− *numeral* (*Num.inc n*) = *NOT* (*numeral n*)›
by (*simp add*: *not-eq-complement sub-inc-One-eq add-One*)

lemma *sub-one-eq-not-neg*:
‹*Num.sub n num.One* = *NOT* (− *numeral n*)›
by (*simp add*: *not-eq-complement*)

lemma *minus-numeral-eq-not-sub-one*:
‹− *numeral n* = *NOT* (*Num.sub n num.One*)›
by (*simp add*: *not-eq-complement*)

lemma *not-numeral-eq* [*simp*]:
‹*NOT* (*numeral n*) = − *numeral* (*Num.inc n*)›
by (*simp add*: *minus-numeral-inc-eq*)

lemma *not-minus-numeral-eq* [*simp*]:
‹*NOT* (− *numeral n*) = *Num.sub n num.One*›
by (*simp add*: *sub-one-eq-not-neg*)

lemma *minus-not-numeral-eq* [*simp*]:
‹− (*NOT* (*numeral n*)) = *numeral* (*Num.inc n*)›
by *simp*

lemma *not-numeral-BitM-eq*:
‹*NOT* (*numeral* (*Num.BitM n*)) = − *numeral* (*num.Bit0 n*)›
by (*simp add*: *inc-BitM-eq*)

lemma *not-numeral-Bit0-eq*:
‹*NOT* (*numeral* (*Num.Bit0 n*)) = − *numeral* (*num.Bit1 n*)›
by *simp*

end

lemma *bit-minus-numeral-int* [*simp*]:
‹*bit* (− *numeral* (*num.Bit0 w*) :: *int*) (*numeral n*) ⟷ *bit* (− *numeral w* :: *int*) (*pred-numeral n*)›
‹*bit* (− *numeral* (*num.Bit1 w*) :: *int*) (*numeral n*) ⟷ ¬ *bit* (*numeral w* :: *int*) (*pred-numeral n*)›
by (*simp-all add*: *bit-minus-iff bit-not-iff numeral-eq-Suc bit-Suc add-One sub-inc-One-eq*)

lemma *bit-minus-numeral-Bit0-Suc-iff* [*simp*]:
‹*bit* (− *numeral* (*num.Bit0 w*) :: *int*) (*Suc n*) ⟷ *bit* (− *numeral w* :: *int*) *n*›
by (*simp add*: *bit-Suc*)

lemma *bit-minus-numeral-Bit1-Suc-iff* [*simp*]:
‹*bit* (− *numeral* (*num.Bit1 w*) :: *int*) (*Suc n*) ⟷ ¬ *bit* (*numeral w* :: *int*) *n*›
by (*simp add*: *bit-Suc add-One flip*: *bit-not-int-iff*)

lemma *and-not-numerals*:
‹*1 AND NOT 1* = (*0* :: *int*)›
‹*1 AND NOT* (*numeral* (*Num.Bit0 n*)) = (*1* :: *int*)›
‹*1 AND NOT* (*numeral* (*Num.Bit1 n*)) = (*0* :: *int*)›
‹*numeral* (*Num.Bit0 m*) *AND NOT* (*1* :: *int*) = *numeral* (*Num.Bit0 m*)›
‹*numeral* (*Num.Bit0 m*) *AND NOT* (*numeral* (*Num.Bit0 n*)) = (*2* :: *int*) ∗ (*numeral m AND NOT* (*numeral n*))›
‹*numeral* (*Num.Bit0 m*) *AND NOT* (*numeral* (*Num.Bit1 n*)) = (*2* :: *int*) ∗ (*numeral m AND NOT* (*numeral n*))›
‹*numeral* (*Num.Bit1 m*) *AND NOT* (*1* :: *int*) = *numeral* (*Num.Bit0 m*)›
‹*numeral* (*Num.Bit1 m*) *AND NOT* (*numeral* (*Num.Bit0 n*)) = *1* + (*2* :: *int*) ∗ (*numeral m AND NOT* (*numeral n*))›
‹*numeral* (*Num.Bit1 m*) *AND NOT* (*numeral* (*Num.Bit1 n*)) = (*2* :: *int*) ∗ (*numeral m AND NOT* (*numeral n*))›
by (*simp-all add*: *bit-eq-iff*)
(*auto simp*: *bit-0 bit-simps bit-Suc bit-numeral-rec BitM-inc-eq sub-inc-One-eq split*: *nat.split*)

fun *and-not-num* :: ‹*num* ⇒ *num* ⇒ *num option*›
where
‹*and-not-num num.One num.One* = *None*›
| ‹*and-not-num num.One* (*num.Bit0 n*) = *Some num.One*›
| ‹*and-not-num num.One* (*num.Bit1 n*) = *None*›
| ‹*and-not-num* (*num.Bit0 m*) *num.One* = *Some* (*num.Bit0 m*)›
| ‹*and-not-num* (*num.Bit0 m*) (*num.Bit0 n*) = *map-option num.Bit0* (*and-not-num m n*)›
| ‹*and-not-num* (*num.Bit0 m*) (*num.Bit1 n*) = *map-option num.Bit0* (*and-not-num m n*)›
| ‹*and-not-num* (*num.Bit1 m*) *num.One* = *Some* (*num.Bit0 m*)›
| ‹*and-not-num* (*num.Bit1 m*) (*num.Bit0 n*) = (*case and-not-num m n of None* ⇒ *Some num.One* | *Some n′* ⇒ *Some* (*num.Bit1 n′*))›
| ‹*and-not-num* (*num.Bit1 m*) (*num.Bit1 n*) = *map-option num.Bit0* (*and-not-num m n*)›

lemma *int-numeral-and-not-num*:
‹*numeral m AND NOT* (*numeral n*) = (*case and-not-num m n of None* ⇒ *0* :: *int* | *Some n′* ⇒ *numeral n′*)›
by (*induction m n rule*: *and-not-num.induct*) (*simp-all del*: *not-numeral-eq not-one-eq add*: *and-not-numerals split*: *option.splits*)

lemma *int-numeral-not-and-num*:
‹*NOT* (*numeral m*) *AND numeral n* = (*case and-not-num n m of None* ⇒ *0* :: *int* | *Some n'* ⇒ *numeral n'*)›
using *int-numeral-and-not-num* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *and-not-num-eq-None-iff*:
‹*and-not-num m n* = *None* ⟷ *numeral m AND NOT* (*numeral n*) = (*0* :: *int*)›
by (*simp del*: *not-numeral-eq add*: *int-numeral-and-not-num split*: *option.split*)

lemma *and-not-num-eq-Some-iff*:
‹*and-not-num m n* = *Some q* ⟷ *numeral m AND NOT* (*numeral n*) = (*numeral q* :: *int*)›
by (*simp del*: *not-numeral-eq add*: *int-numeral-and-not-num split*: *option.split*)

lemma *and-minus-numerals* [*simp*]:
‹*1 AND* − (*numeral* (*num.Bit0 n*)) = (*0*::*int*)›
‹*1 AND* − (*numeral* (*num.Bit1 n*)) = (*1*::*int*)›
‹*numeral m AND* − (*numeral* (*num.Bit0 n*)) = (*case and-not-num m* (*Num.BitM n*) *of None* ⇒ *0* :: *int* | *Some n'* ⇒ *numeral n'*)›
‹*numeral m AND* − (*numeral* (*num.Bit1 n*)) = (*case and-not-num m* (*Num.Bit0 n*) *of None* ⇒ *0* :: *int* | *Some n'* ⇒ *numeral n'*)›
‹− (*numeral* (*num.Bit0 n*)) *AND 1* = (*0*::*int*)›
‹− (*numeral* (*num.Bit1 n*)) *AND 1* = (*1*::*int*)›
‹− (*numeral* (*num.Bit0 n*)) *AND numeral m* = (*case and-not-num m* (*Num.BitM n*) *of None* ⇒ *0* :: *int* | *Some n'* ⇒ *numeral n'*)›
‹− (*numeral* (*num.Bit1 n*)) *AND numeral m* = (*case and-not-num m* (*Num.Bit0 n*) *of None* ⇒ *0* :: *int* | *Some n'* ⇒ *numeral n'*)›
by (*simp-all del*: *not-numeral-eq add*: *ac-simps*
and-not-numerals one-and-eq not-numeral-BitM-eq not-numeral-Bit0-eq and-not-num-eq-None-iff and-not-num-eq-Some-iff split: *option.split*)

lemma *and-minus-minus-numerals* [*simp*]:
‹− (*numeral m* :: *int*) *AND* − (*numeral n* :: *int*) = *NOT* ((*numeral m* − *1*) *OR* (*numeral n* − *1*))›
by (*simp add*: *minus-numeral-eq-not-sub-one*)

lemma *or-not-numerals*:
‹*1 OR NOT 1* = *NOT* (*0* :: *int*)›
‹*1 OR NOT* (*numeral* (*Num.Bit0 n*)) = *NOT* (*numeral* (*Num.Bit0 n*) :: *int*)›
‹*1 OR NOT* (*numeral* (*Num.Bit1 n*)) = *NOT* (*numeral* (*Num.Bit0 n*) :: *int*)›
‹*numeral* (*Num.Bit0 m*) *OR NOT* (*1* :: *int*) = *NOT* (*1* :: *int*)›
‹*numeral* (*Num.Bit0 m*) *OR NOT* (*numeral* (*Num.Bit0 n*)) = *1* + (*2* :: *int*) ∗ (*numeral m OR NOT* (*numeral n*))›
‹*numeral* (*Num.Bit0 m*) *OR NOT* (*numeral* (*Num.Bit1 n*)) = (*2* :: *int*) ∗ (*numeral m OR NOT* (*numeral n*))›
‹*numeral* (*Num.Bit1 m*) *OR NOT* (*1* :: *int*) = *NOT* (*0* :: *int*)›
‹*numeral* (*Num.Bit1 m*) *OR NOT* (*numeral* (*Num.Bit0 n*)) = *1* + (*2* :: *int*) ∗ (*numeral m OR NOT* (*numeral n*))›
‹*numeral* (*Num.Bit1 m*) *OR NOT* (*numeral* (*Num.Bit1 n*)) = *1* + (*2* :: *int*) ∗

(*numeral m OR NOT (numeral n)*)›
by (*simp-all add*: *bit-eq-iff*)
(*auto simp*: *bit-0 bit-simps bit-Suc bit-numeral-rec sub-inc-One-eq split*: *nat.split*)

fun *or-not-num-neg* :: ‹*num* ⇒ *num* ⇒ *num*›
where
‹*or-not-num-neg num.One num.One = num.One*›
| ‹*or-not-num-neg num.One* (*num.Bit0 m*) = *num.Bit1 m*›
| ‹*or-not-num-neg num.One* (*num.Bit1 m*) = *num.Bit1 m*›
| ‹*or-not-num-neg* (*num.Bit0 n*) *num.One* = *num.Bit0 num.One*›
| ‹*or-not-num-neg* (*num.Bit0 n*) (*num.Bit0 m*) = *Num.BitM* (*or-not-num-neg n m*)›
| ‹*or-not-num-neg* (*num.Bit0 n*) (*num.Bit1 m*) = *num.Bit0* (*or-not-num-neg n m*)›
| ‹*or-not-num-neg* (*num.Bit1 n*) *num.One* = *num.One*›
| ‹*or-not-num-neg* (*num.Bit1 n*) (*num.Bit0 m*) = *Num.BitM* (*or-not-num-neg n m*)›
| ‹*or-not-num-neg* (*num.Bit1 n*) (*num.Bit1 m*) = *Num.BitM* (*or-not-num-neg n m*)›

lemma *int-numeral-or-not-num-neg*:
‹*numeral m OR NOT* (*numeral n* :: *int*) = − *numeral* (*or-not-num-neg m n*)›
by (*induction m n rule*: *or-not-num-neg.induct*) (*simp-all del*: *not-numeral-eq not-one-eq add*: *or-not-numerals*, *simp-all*)

lemma *int-numeral-not-or-num-neg*:
‹*NOT* (*numeral m*) *OR* (*numeral n* :: *int*) = − *numeral* (*or-not-num-neg n m*)›
using *int-numeral-or-not-num-neg* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *numeral-or-not-num-eq*:
‹*numeral* (*or-not-num-neg m n*) = − (*numeral m OR NOT* (*numeral n* :: *int*))›
using *int-numeral-or-not-num-neg* [*of m n*] **by** *simp*

lemma *or-minus-numerals* [*simp*]:
‹*1 OR* − (*numeral* (*num.Bit0 n*)) = − (*numeral* (*or-not-num-neg num.One* (*Num.BitM n*)) :: *int*)›
‹*1 OR* − (*numeral* (*num.Bit1 n*)) = − (*numeral* (*num.Bit1 n*) :: *int*)›
‹*numeral m OR* − (*numeral* (*num.Bit0 n*)) = − (*numeral* (*or-not-num-neg m* (*Num.BitM n*)) :: *int*)›
‹*numeral m OR* − (*numeral* (*num.Bit1 n*)) = − (*numeral* (*or-not-num-neg m* (*Num.Bit0 n*)) :: *int*)›
‹− (*numeral* (*num.Bit0 n*)) *OR 1* = − (*numeral* (*or-not-num-neg num.One* (*Num.BitM n*)) :: *int*)›
‹− (*numeral* (*num.Bit1 n*)) *OR 1* = − (*numeral* (*num.Bit1 n*) :: *int*)›
‹− (*numeral* (*num.Bit0 n*)) *OR numeral m* = − (*numeral* (*or-not-num-neg m* (*Num.BitM n*)) :: *int*)›
‹− (*numeral* (*num.Bit1 n*)) *OR numeral m* = − (*numeral* (*or-not-num-neg m* (*Num.Bit0 n*)) :: *int*)›
by (*simp-all only*: *or.commute* [*of - 1*] *or.commute* [*of -* ‹*numeral m*›]
minus-numeral-eq-not-sub-one or-not-numerals

numeral-or-not-num-eq arith-simps minus-minus numeral-One)

lemma *or-minus-minus-numerals* [*simp*]:
‹− (*numeral m* :: *int*) *OR* − (*numeral n* :: *int*) = *NOT* ((*numeral m* − *1*) *AND* (*numeral n* − *1*))›
by (*simp add*: *minus-numeral-eq-not-sub-one*)

lemma *xor-minus-numerals* [*simp*]:
‹− *numeral n XOR k* = *NOT* (*neg-numeral-class.sub n num.One XOR k*)›
‹*k XOR* − *numeral n* = *NOT* (*k XOR* (*neg-numeral-class.sub n num.One*))› **for** *k* :: *int*
by (*simp-all add*: *minus-numeral-eq-not-sub-one*)

definition *take-bit-num* :: ‹*nat* ⇒ *num* ⇒ *num option*›
where ‹*take-bit-num n m* =
(*if take-bit n* (*numeral m* :: *nat*) = *0 then None else Some* (*num-of-nat* (*take-bit n* (*numeral m* :: *nat*))))›

lemma *take-bit-num-simps*:
‹*take-bit-num 0 m* = *None*›
‹*take-bit-num* (*Suc n*) *Num.One* =
Some Num.One›
‹*take-bit-num* (*Suc n*) (*Num.Bit0 m*) =
(*case take-bit-num n m of None* ⇒ *None* | *Some q* ⇒ *Some* (*Num.Bit0 q*))›
‹*take-bit-num* (*Suc n*) (*Num.Bit1 m*) =
Some (*case take-bit-num n m of None* ⇒ *Num.One* | *Some q* ⇒ *Num.Bit1 q*)›
‹*take-bit-num* (*numeral r*) *Num.One* =
Some Num.One›
‹*take-bit-num* (*numeral r*) (*Num.Bit0 m*) =
(*case take-bit-num* (*pred-numeral r*) *m of None* ⇒ *None* | *Some q* ⇒ *Some* (*Num.Bit0 q*))›
‹*take-bit-num* (*numeral r*) (*Num.Bit1 m*) =
Some (*case take-bit-num* (*pred-numeral r*) *m of None* ⇒ *Num.One* | *Some q* ⇒ *Num.Bit1 q*)›
by (*auto simp*: *take-bit-num-def ac-simps mult-2 num-of-nat-double*
take-bit-Suc-bit0 take-bit-Suc-bit1 take-bit-numeral-bit0 take-bit-numeral-bit1)

lemma *take-bit-num-code* [*code*]:
— Ocaml-style pattern matching is more robust wrt. different representations of *nat*
‹*take-bit-num n m* = (*case* (*n*, *m*)
of (*0*, -) ⇒ *None*
| (*Suc n*, *Num.One*) ⇒ *Some Num.One*
| (*Suc n*, *Num.Bit0 m*) ⇒ (*case take-bit-num n m of None* ⇒ *None* | *Some q* ⇒ *Some* (*Num.Bit0 q*))
| (*Suc n*, *Num.Bit1 m*) ⇒ *Some* (*case take-bit-num n m of None* ⇒ *Num.One* | *Some q* ⇒ *Num.Bit1 q*))›
by (*cases n*; *cases m*) (*simp-all add*: *take-bit-num-simps*)

context *semiring-bit-operations*
begin

lemma *take-bit-num-eq-None-imp*:
  ‹*take-bit m* (*numeral n*) = *0*› **if** ‹*take-bit-num m n* = *None*›
proof −
  from *that* **have** ‹*take-bit m* (*numeral n* :: *nat*) = *0*›
    by (*simp add*: *take-bit-num-def split*: *if-splits*)
  then have ‹*of-nat* (*take-bit m* (*numeral n*)) = *of-nat 0*›
    by *simp*
  then show *?thesis*
    by (*simp add*: *of-nat-take-bit*)
qed

lemma *take-bit-num-eq-Some-imp*:
  ‹*take-bit m* (*numeral n*) = *numeral q*› **if** ‹*take-bit-num m n* = *Some q*›
proof −
  from *that* **have** ‹*take-bit m* (*numeral n* :: *nat*) = *numeral q*›
   by (*auto simp*: *take-bit-num-def Num.numeral-num-of-nat-unfold split*: *if-splits*)
  then have ‹*of-nat* (*take-bit m* (*numeral n*)) = *of-nat* (*numeral q*)›
    by *simp*
  then show *?thesis*
    by (*simp add*: *of-nat-take-bit*)
qed

lemma *take-bit-numeral-numeral*:
  ‹*take-bit* (*numeral m*) (*numeral n*) =
    (*case take-bit-num* (*numeral m*) *n of None* ⇒ *0* | *Some q* ⇒ *numeral q*)›
 by (*auto split*: *option.split dest*: *take-bit-num-eq-None-imp take-bit-num-eq-Some-imp*)

end

lemma *take-bit-numeral-minus-numeral-int*:
  ‹*take-bit* (*numeral m*) (− *numeral n* :: *int*) =
    (*case take-bit-num* (*numeral m*) *n of None* ⇒ *0* | *Some q* ⇒ *take-bit* (*numeral*
m) (*2* ^ *numeral m* − *numeral q*))› (**is** ‹*?lhs* = *?rhs*›)
proof (*cases* ‹*take-bit-num* (*numeral m*) *n*›)
  case *None*
  then show *?thesis*
   by (*auto dest*: *take-bit-num-eq-None-imp* [**where** *?'a* = *int*] *simp add*: *take-bit-eq-0-iff*)
next
  case (*Some q*)
  then have *q*: ‹*take-bit* (*numeral m*) (*numeral n* :: *int*) = *numeral q*›
    by (*auto dest*: *take-bit-num-eq-Some-imp*)
  let *?T* = ‹*take-bit* (*numeral m*) :: *int* ⇒ *int*›
  have ∗: ‹*?T* (*2* ^ *numeral m*) = *?T* (*?T 0*)›
    by (*simp add*: *take-bit-eq-0-iff*)
  have ‹*?lhs* = *?T* (*0* − *numeral n*)›
    by *simp*

also have ‹... = *?T* (*?T* (*?T* *0*) − *?T* (*?T* (*numeral n*)))›
by (*simp only*: *take-bit-diff*)
also have ‹... = *?T* (*2* ^ *numeral m* − *?T* (*numeral n*))›
by (*simp only*: *take-bit-diff flip*: ∗)
also have ‹... = *?rhs*›
by (*simp add*: *q Some*)
finally show *?thesis* **.**
qed

declare *take-bit-num-simps* [*simp*]
take-bit-numeral-numeral [*simp*]
take-bit-numeral-minus-numeral-int [*simp*]

68.7 Symbolic computations for code generation

lemma *bit-int-code* [*code*]:
‹*bit* (*0*::*int*) *n* ⟷ *False*›
‹*bit* (*Int.Neg num.One*) *n* ⟷ *True*›
‹*bit* (*Int.Pos num.One*) *0* ⟷ *True*›
‹*bit* (*Int.Pos* (*num.Bit0 m*)) *0* ⟷ *False*›
‹*bit* (*Int.Pos* (*num.Bit1 m*)) *0* ⟷ *True*›
‹*bit* (*Int.Neg* (*num.Bit0 m*)) *0* ⟷ *False*›
‹*bit* (*Int.Neg* (*num.Bit1 m*)) *0* ⟷ *True*›
‹*bit* (*Int.Pos num.One*) (*Suc n*) ⟷ *False*›
‹*bit* (*Int.Pos* (*num.Bit0 m*)) (*Suc n*) ⟷ *bit* (*Int.Pos m*) *n*›
‹*bit* (*Int.Pos* (*num.Bit1 m*)) (*Suc n*) ⟷ *bit* (*Int.Pos m*) *n*›
‹*bit* (*Int.Neg* (*num.Bit0 m*)) (*Suc n*) ⟷ *bit* (*Int.Neg m*) *n*›
‹*bit* (*Int.Neg* (*num.Bit1 m*)) (*Suc n*) ⟷ *bit* (*Int.Neg* (*Num.inc m*)) *n*›
by (*simp-all add*: *Num.add-One bit-0 bit-Suc*)

lemma *not-int-code* [*code*]:
‹*NOT* (*0* :: *int*) = − *1*›
‹*NOT* (*Int.Pos n*) = *Int.Neg* (*Num.inc n*)›
‹*NOT* (*Int.Neg n*) = *Num.sub n num.One*›
by (*simp-all add*: *Num.add-One not-int-def*)

fun *and-num* :: ‹*num* ⇒ *num* ⇒ *num option*›
where
‹*and-num num.One num.One* = *Some num.One*›
| ‹*and-num num.One* (*num.Bit0 n*) = *None*›
| ‹*and-num num.One* (*num.Bit1 n*) = *Some num.One*›
| ‹*and-num* (*num.Bit0 m*) *num.One* = *None*›
| ‹*and-num* (*num.Bit0 m*) (*num.Bit0 n*) = *map-option num.Bit0* (*and-num m n*)›
| ‹*and-num* (*num.Bit0 m*) (*num.Bit1 n*) = *map-option num.Bit0* (*and-num m n*)›
| ‹*and-num* (*num.Bit1 m*) *num.One* = *Some num.One*›
| ‹*and-num* (*num.Bit1 m*) (*num.Bit0 n*) = *map-option num.Bit0* (*and-num m n*)›
| ‹*and-num* (*num.Bit1 m*) (*num.Bit1 n*) = (*case and-num m n of None* ⇒ *Some num.One* | *Some n′* ⇒ *Some* (*num.Bit1 n′*))›

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *numeral-and-num*:
‹*numeral m AND numeral n = (case and-num m n of None ⇒ 0 | Some n′ ⇒ numeral n′)*›
by (*induction m n rule*: *and-num.induct*) (*simp-all add*: *split*: *option.split*)

lemma *and-num-eq-None-iff*:
‹*and-num m n = None ⟷ numeral m AND numeral n = 0*›
by (*simp add*: *numeral-and-num split*: *option.split*)

lemma *and-num-eq-Some-iff*:
‹*and-num m n = Some q ⟷ numeral m AND numeral n = numeral q*›
by (*simp add*: *numeral-and-num split*: *option.split*)

end

lemma *and-int-code* [*code*]:
fixes *i j* :: *int* **shows**
‹*0 AND j = 0*›
‹*i AND 0 = 0*›
‹*Int.Pos n AND Int.Pos m = (case and-num n m of None ⇒ 0 | Some n′ ⇒ Int.Pos n′)*›
‹*Int.Neg n AND Int.Neg m = NOT (Num.sub n num.One OR Num.sub m num.One)*›
‹*Int.Pos n AND Int.Neg num.One = Int.Pos n*›
‹*Int.Pos n AND Int.Neg (num.Bit0 m) = Num.sub (or-not-num-neg (Num.BitM m) n) num.One*›
‹*Int.Pos n AND Int.Neg (num.Bit1 m) = Num.sub (or-not-num-neg (num.Bit0 m) n) num.One*›
‹*Int.Neg num.One AND Int.Pos m = Int.Pos m*›
‹*Int.Neg (num.Bit0 n) AND Int.Pos m = Num.sub (or-not-num-neg (Num.BitM n) m) num.One*›
‹*Int.Neg (num.Bit1 n) AND Int.Pos m = Num.sub (or-not-num-neg (num.Bit0 n) m) num.One*›
apply (*auto simp*: *and-num-eq-None-iff* [**where** *?′a = int*] *and-num-eq-Some-iff* [**where** *?′a = int*]
split: *option.split*)
apply (*simp-all only*: *sub-one-eq-not-neg numeral-or-not-num-eq minus-minus and-not-numerals*
bit.de-Morgan-disj bit.double-compl and-not-num-eq-None-iff and-not-num-eq-Some-iff ac-simps)
done

context *linordered-euclidean-semiring-bit-operations*
begin

fun *or-num* :: ‹*num ⇒ num ⇒ num*›

where
  ‹*or-num num.One num.One = num.One*›
| ‹*or-num num.One (num.Bit0 n) = num.Bit1 n*›
| ‹*or-num num.One (num.Bit1 n) = num.Bit1 n*›
| ‹*or-num (num.Bit0 m) num.One = num.Bit1 m*›
| ‹*or-num (num.Bit0 m) (num.Bit0 n) = num.Bit0 (or-num m n)*›
| ‹*or-num (num.Bit0 m) (num.Bit1 n) = num.Bit1 (or-num m n)*›
| ‹*or-num (num.Bit1 m) num.One = num.Bit1 m*›
| ‹*or-num (num.Bit1 m) (num.Bit0 n) = num.Bit1 (or-num m n)*›
| ‹*or-num (num.Bit1 m) (num.Bit1 n) = num.Bit1 (or-num m n)*›

lemma *numeral-or-num*:
  ‹*numeral m OR numeral n = numeral (or-num m n)*›
  by (*induction m n rule*: *or-num.induct*) *simp-all*

lemma *numeral-or-num-eq*:
  ‹*numeral (or-num m n) = numeral m OR numeral n*›
  by (*simp add*: *numeral-or-num*)

end

lemma *or-int-code* [*code*]:
  fixes *i j* :: *int* **shows**
  ‹*0 OR j = j*›
  ‹*i OR 0 = i*›
  ‹*Int.Pos n OR Int.Pos m = Int.Pos (or-num n m)*›
  ‹*Int.Neg n OR Int.Neg m = NOT (Num.sub n num.One AND Num.sub m num.One)*›
  ‹*Int.Pos n OR Int.Neg num.One = Int.Neg num.One*›
  ‹*Int.Pos n OR Int.Neg (num.Bit0 m) = (case and-not-num (Num.BitM m) n of None ⇒ −1 | Some n′ ⇒ Int.Neg (Num.inc n′))*›
  ‹*Int.Pos n OR Int.Neg (num.Bit1 m) = (case and-not-num (num.Bit0 m) n of None ⇒ −1 | Some n′ ⇒ Int.Neg (Num.inc n′))*›
  ‹*Int.Neg num.One OR Int.Pos m = Int.Neg num.One*›
  ‹*Int.Neg (num.Bit0 n) OR Int.Pos m = (case and-not-num (Num.BitM n) m of None ⇒ −1 | Some n′ ⇒ Int.Neg (Num.inc n′))*›
  ‹*Int.Neg (num.Bit1 n) OR Int.Pos m = (case and-not-num (num.Bit0 n) m of None ⇒ −1 | Some n′ ⇒ Int.Neg (Num.inc n′))*›
  apply (*auto simp*: *numeral-or-num-eq split*: *option.splits*)
       apply (*simp-all only*: *and-not-num-eq-None-iff and-not-num-eq-Some-iff and-not-numerals*
        numeral-or-not-num-eq or-eq-not-not-and bit.double-compl ac-simps flip: *numeral-eq-iff* [**where** *?′a = int*])
      apply *simp-all*
  done

fun *xor-num* :: ‹*num ⇒ num ⇒ num option*›
where
  ‹*xor-num num.One num.One = None*›

| ‹*xor-num num.One* (*num.Bit0 n*) = *Some* (*num.Bit1 n*)›
| ‹*xor-num num.One* (*num.Bit1 n*) = *Some* (*num.Bit0 n*)›
| ‹*xor-num* (*num.Bit0 m*) *num.One* = *Some* (*num.Bit1 m*)›
| ‹*xor-num* (*num.Bit0 m*) (*num.Bit0 n*) = *map-option num.Bit0* (*xor-num m n*)›
| ‹*xor-num* (*num.Bit0 m*) (*num.Bit1 n*) = *Some* (*case xor-num m n of None* ⇒ *num.One* | *Some n′* ⇒ *num.Bit1 n′*)›
| ‹*xor-num* (*num.Bit1 m*) *num.One* = *Some* (*num.Bit0 m*)›
| ‹*xor-num* (*num.Bit1 m*) (*num.Bit0 n*) = *Some* (*case xor-num m n of None* ⇒ *num.One* | *Some n′* ⇒ *num.Bit1 n′*)›
| ‹*xor-num* (*num.Bit1 m*) (*num.Bit1 n*) = *map-option num.Bit0* (*xor-num m n*)›

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *numeral-xor-num*:
‹*numeral m XOR numeral n* = (*case xor-num m n of None* ⇒ *0* | *Some n′* ⇒ *numeral n′*)›
by (*induction m n rule*: *xor-num.induct*) (*simp-all split*: *option.split*)

lemma *xor-num-eq-None-iff*:
‹*xor-num m n* = *None* ⟷ *numeral m XOR numeral n* = *0*›
by (*simp add*: *numeral-xor-num split*: *option.split*)

lemma *xor-num-eq-Some-iff*:
‹*xor-num m n* = *Some q* ⟷ *numeral m XOR numeral n* = *numeral q*›
by (*simp add*: *numeral-xor-num split*: *option.split*)

end

context *semiring-bit-operations*
begin

lemma *push-bit-eq-pow*:
‹*push-bit* (*numeral n*) *1* = *numeral* (*Num.pow* (*Num.Bit0 Num.One*) *n*)›
by *simp*

lemma *set-bit-of-0* [*simp*]:
‹*set-bit n 0* = *2* ^ *n*›
by (*simp add*: *set-bit-eq-or*)

lemma *unset-bit-of-0* [*simp*]:
‹*unset-bit n 0* = *0*›
by (*simp add*: *unset-bit-eq-or-xor*)

lemma *flip-bit-of-0* [*simp*]:
‹*flip-bit n 0* = *2* ^ *n*›
by (*simp add*: *flip-bit-eq-xor*)

lemma *set-bit-0-numeral-eq* [*simp*]:

‹*set-bit 0 (numeral Num.One) = 1*›
‹*set-bit 0 (numeral (Num.Bit0 m)) = numeral (Num.Bit1 m)*›
‹*set-bit 0 (numeral (Num.Bit1 m)) = numeral (Num.Bit1 m)*›
by (*simp-all add*: *set-bit-0*)

lemma *set-bit-numeral-eq-or* [*simp*]:
‹*set-bit (numeral n) (numeral m) = numeral m OR push-bit (numeral n) 1*›
by (*fact set-bit-eq-or*)

lemma *unset-bit-0-numeral-eq-and-not'* [*simp*]:
‹*unset-bit 0 (numeral Num.One) = 0*›
‹*unset-bit 0 (numeral (Num.Bit0 m)) = numeral (Num.Bit0 m)*›
‹*unset-bit 0 (numeral (Num.Bit1 m)) = numeral (Num.Bit0 m)*›
by (*simp-all add*: *unset-bit-0*)

lemma *unset-bit-numeral-eq-or* [*simp*]:
‹*unset-bit (numeral n) (numeral m) =*
(case and-not-num m (Num.pow (Num.Bit0 Num.One) n)
of None ⇒ 0
| Some q ⇒ numeral q)› (**is** ‹*?lhs = -*›)
proof −
have ‹*?lhs = of-nat (unset-bit (numeral n) (numeral m))*›
by (*simp add*: *of-nat-unset-bit-eq*)
also have ‹*unset-bit (numeral n) (numeral m) = nat (unset-bit (numeral n) (numeral m))*›
by (*simp flip*: *int-int-eq add*: *Bit-Operations.of-nat-unset-bit-eq*)
finally have ∗: ‹*?lhs = of-nat (nat (unset-bit (numeral n) (numeral m)))*› **.**
show *?thesis*
by (*simp only*: ∗ *unset-bit-eq-and-not Bit-Operations.push-bit-eq-pow int-numeral-and-not-num*)
(*auto split*: *option.splits*)
qed

lemma *flip-bit-0-numeral-eq-or* [*simp*]:
‹*flip-bit 0 (numeral Num.One) = 0*›
‹*flip-bit 0 (numeral (Num.Bit0 m)) = numeral (Num.Bit1 m)*›
‹*flip-bit 0 (numeral (Num.Bit1 m)) = numeral (Num.Bit0 m)*›
by (*simp-all add*: *flip-bit-0*)

lemma *flip-bit-numeral-eq-xor* [*simp*]:
‹*flip-bit (numeral n) (numeral m) = numeral m XOR push-bit (numeral n) 1*›
by (*fact flip-bit-eq-xor*)

end

context *ring-bit-operations*
begin

lemma *set-bit-minus-numeral-eq-or* [*simp*]:
‹*set-bit (numeral n) (− numeral m) = − numeral m OR push-bit (numeral n) 1*›

by (*fact set-bit-eq-or*)

lemma *unset-bit-minus-numeral-eq-and-not* [*simp*]:
‹*unset-bit* (*numeral n*) (− *numeral m*) = − *numeral m AND NOT* (*push-bit* (*numeral n*) *1*)›
by (*fact unset-bit-eq-and-not*)

lemma *flip-bit-minus-numeral-eq-xor* [*simp*]:
‹*flip-bit* (*numeral n*) (− *numeral m*) = − *numeral m XOR push-bit* (*numeral n*) *1*›
by (*fact flip-bit-eq-xor*)

end

lemma *xor-int-code* [*code*]:
fixes *i j* :: *int* **shows**
‹*0 XOR j* = *j*›
‹*i XOR 0* = *i*›
‹*Int.Pos n XOR Int.Pos m* = (*case xor-num n m of None* ⇒ *0* | *Some n′* ⇒ *Int.Pos n′*)›
‹*Int.Neg n XOR Int.Neg m* = *Num.sub n num.One XOR Num.sub m num.One*›
‹*Int.Neg n XOR Int.Pos m* = *NOT* (*Num.sub n num.One XOR Int.Pos m*)›
‹*Int.Pos n XOR Int.Neg m* = *NOT* (*Int.Pos n XOR Num.sub m num.One*)›
by (*simp-all add*: *xor-num-eq-None-iff* [**where** *?′a* = *int*] *xor-num-eq-Some-iff* [**where** *?′a* = *int*] *split*: *option.split*)

lemma *push-bit-int-code* [*code*]:
‹*push-bit 0 i* = *i*›
‹*push-bit* (*Suc n*) *i* = *push-bit n* (*Int.dup i*)›
by (*simp-all add*: *ac-simps*)

lemma *drop-bit-int-code* [*code*]:
fixes *i* :: *int* **shows**
‹*drop-bit 0 i* = *i*›
‹*drop-bit* (*Suc n*) *0* = (*0* :: *int*)›
‹*drop-bit* (*Suc n*) (*Int.Pos num.One*) = *0*›
‹*drop-bit* (*Suc n*) (*Int.Pos* (*num.Bit0 m*)) = *drop-bit n* (*Int.Pos m*)›
‹*drop-bit* (*Suc n*) (*Int.Pos* (*num.Bit1 m*)) = *drop-bit n* (*Int.Pos m*)›
‹*drop-bit* (*Suc n*) (*Int.Neg num.One*) = − *1*›
‹*drop-bit* (*Suc n*) (*Int.Neg* (*num.Bit0 m*)) = *drop-bit n* (*Int.Neg m*)›
‹*drop-bit* (*Suc n*) (*Int.Neg* (*num.Bit1 m*)) = *drop-bit n* (*Int.Neg* (*Num.inc m*))›
by (*simp-all add*: *drop-bit-Suc add-One*)

68.8 More properties

lemma *take-bit-eq-mask-iff*:
‹*take-bit n k* = *mask n* ⟷ *take-bit n* (*k* + *1*) = *0*› (**is** ‹*?P* ⟷ *?Q*›)
for *k* :: *int*
proof

assume *?P*
then have ‹*take-bit n* (*take-bit n k* + *take-bit n 1*) = *0*›
by (*simp add*: *mask-eq-exp-minus-1 take-bit-eq-0-iff*)
then show *?Q*
by (*simp only*: *take-bit-add*)
next
assume *?Q*
then have ‹*take-bit n* (*k* + *1*) − *1* = − *1*›
by *simp*
then have ‹*take-bit n* (*take-bit n* (*k* + *1*) − *1*) = *take-bit n* (− *1*)›
by *simp*
moreover have ‹*take-bit n* (*take-bit n* (*k* + *1*) − *1*) = *take-bit n k*›
by (*simp add*: *take-bit-eq-mod mod-simps*)
ultimately show *?P*
by *simp*
qed

lemma *take-bit-eq-mask-iff-exp-dvd*:
‹*take-bit n k* = *mask n* ⟷ *2* ^ *n dvd k* + *1*›
for *k* :: *int*
by (*simp add*: *take-bit-eq-mask-iff flip*: *take-bit-eq-0-iff*)

68.9 Bit concatenation

definition *concat-bit* :: ‹*nat* ⇒ *int* ⇒ *int* ⇒ *int*›
where ‹*concat-bit n k l* = *take-bit n k OR push-bit n l*›

lemma *bit-concat-bit-iff* [*bit-simps*]:
‹*bit* (*concat-bit m k l*) *n* ⟷ *n* < *m* ∧ *bit k n* ∨ *m* ≤ *n* ∧ *bit l* (*n* − *m*)›
by (*simp add*: *concat-bit-def bit-or-iff bit-and-iff bit-take-bit-iff bit-push-bit-iff ac-simps*)

lemma *concat-bit-eq*:
‹*concat-bit n k l* = *take-bit n k* + *push-bit n l*›
proof −
have ‹*take-bit n k AND push-bit n l* = *0*›
by (*simp add*: *bit-eq-iff bit-simps*)
then show *?thesis*
by (*simp add*: *bit-eq-iff bit-simps disjunctive-add-eq-or*)
qed

lemma *concat-bit-0* [*simp*]:
‹*concat-bit 0 k l* = *l*›
by (*simp add*: *concat-bit-def*)

lemma *concat-bit-Suc*:
‹*concat-bit* (*Suc n*) *k l* = *k mod 2* + *2* ∗ *concat-bit n* (*k div 2*) *l*›
by (*simp add*: *concat-bit-eq take-bit-Suc push-bit-double*)

lemma *concat-bit-of-zero-1* [*simp*]:
‹*concat-bit n 0 l = push-bit n l*›
by (*simp add*: *concat-bit-def*)

lemma *concat-bit-of-zero-2* [*simp*]:
‹*concat-bit n k 0 = take-bit n k*›
by (*simp add*: *concat-bit-def take-bit-eq-mask*)

lemma *concat-bit-nonnegative-iff* [*simp*]:
‹*concat-bit n k l ≥ 0 ⟷ l ≥ 0*›
by (*simp add*: *concat-bit-def*)

lemma *concat-bit-negative-iff* [*simp*]:
‹*concat-bit n k l < 0 ⟷ l < 0*›
by (*simp add*: *concat-bit-def*)

lemma *concat-bit-assoc*:
‹*concat-bit n k* (*concat-bit m l r*) = *concat-bit* (*m + n*) (*concat-bit n k l*) *r*›
by (*rule bit-eqI*) (*auto simp*: *bit-concat-bit-iff ac-simps*)

lemma *concat-bit-assoc-sym*:
‹*concat-bit m* (*concat-bit n k l*) *r* = *concat-bit* (*min m n*) *k* (*concat-bit* (*m − n*) *l r*)›
by (*rule bit-eqI*) (*auto simp*: *bit-concat-bit-iff ac-simps min-def*)

lemma *concat-bit-eq-iff*:
‹*concat-bit n k l = concat-bit n r s*
⟷ *take-bit n k = take-bit n r ∧ l = s*› (**is** ‹*?P ⟷ ?Q*›)
proof
assume *?Q*
then show *?P*
by (*simp add*: *concat-bit-def*)
next
assume *?P*
then have ∗: ‹*bit* (*concat-bit n k l*) *m* = *bit* (*concat-bit n r s*) *m*› **for** *m*
by (*simp add*: *bit-eq-iff*)
have ‹*take-bit n k = take-bit n r*›
proof (*rule bit-eqI*)
fix *m*
from ∗ [*of m*]
show ‹*bit* (*take-bit n k*) *m* ⟷ *bit* (*take-bit n r*) *m*›
by (*auto simp*: *bit-take-bit-iff bit-concat-bit-iff*)
qed
moreover have ‹*push-bit n l = push-bit n s*›
proof (*rule bit-eqI*)
fix *m*
from ∗ [*of m*]
show ‹*bit* (*push-bit n l*) *m* ⟷ *bit* (*push-bit n s*) *m*›
by (*auto simp*: *bit-push-bit-iff bit-concat-bit-iff*)

qed
then have ‹*l* = *s*›
by (*simp add*: *push-bit-eq-mult*)
ultimately show *?Q*
by (*simp add*: *concat-bit-def*)
qed

lemma *take-bit-concat-bit-eq*:
‹*take-bit m* (*concat-bit n k l*) = *concat-bit* (*min m n*) *k* (*take-bit* (*m* − *n*) *l*)›
by (*rule bit-eqI*)
(*auto simp*: *bit-take-bit-iff bit-concat-bit-iff min-def*)

lemma *concat-bit-take-bit-eq*:
‹*concat-bit n* (*take-bit n b*) = *concat-bit n b*›
by (*simp add*: *concat-bit-def* [*abs-def*])

68.10 Taking bits with sign propagation

context *ring-bit-operations*
begin

definition *signed-take-bit* :: ‹*nat* ⇒ *'a* ⇒ *'a*›
where ‹*signed-take-bit n a* = *take-bit n a OR* (*of-bool* (*bit a n*) ∗ *NOT* (*mask n*))›

lemma *signed-take-bit-eq-if-positive*:
‹*signed-take-bit n a* = *take-bit n a*› **if** ‹¬ *bit a n*›
using *that* **by** (*simp add*: *signed-take-bit-def*)

lemma *signed-take-bit-eq-if-negative*:
‹*signed-take-bit n a* = *take-bit n a OR NOT* (*mask n*)› **if** ‹*bit a n*›
using *that* **by** (*simp add*: *signed-take-bit-def*)

lemma *even-signed-take-bit-iff*:
‹*even* (*signed-take-bit m a*) ⟷ *even a*›
by (*auto simp*: *bit-0 signed-take-bit-def even-or-iff even-mask-iff bit-double-iff*)

lemma *bit-signed-take-bit-iff* [*bit-simps*]:
‹*bit* (*signed-take-bit m a*) *n* ⟷ *possible-bit TYPE*(*'a*) *n* ∧ *bit a* (*min m n*)›
by (*simp add*: *signed-take-bit-def bit-take-bit-iff bit-or-iff bit-not-iff bit-mask-iff min-def not-le*)
(*blast dest*: *bit-imp-possible-bit*)

lemma *signed-take-bit-0* [*simp*]:
‹*signed-take-bit 0 a* = − (*a mod 2*)›
by (*simp add*: *bit-0 signed-take-bit-def odd-iff-mod-2-eq-one*)

lemma *signed-take-bit-Suc*:
‹*signed-take-bit* (*Suc n*) *a* = *a mod 2* + *2* ∗ *signed-take-bit n* (*a div 2*)›

by (*simp add*: *bit-eq-iff bit-sum-mult-2-cases bit-simps bit-0 possible-bit-less-imp flip*: *bit-Suc min-Suc-Suc*)

lemma *signed-take-bit-of-0* [*simp*]:
‹*signed-take-bit n 0 = 0*›
by (*simp add*: *signed-take-bit-def*)

lemma *signed-take-bit-of-minus-1* [*simp*]:
‹*signed-take-bit n (− 1) = − 1*›
by (*simp add*: *signed-take-bit-def mask-eq-exp-minus-1 possible-bit-def*)

lemma *signed-take-bit-Suc-1* [*simp*]:
‹*signed-take-bit (Suc n) 1 = 1*›
by (*simp add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-numeral-of-1* [*simp*]:
‹*signed-take-bit (numeral k) 1 = 1*›
by (*simp add*: *bit-1-iff signed-take-bit-eq-if-positive*)

lemma *signed-take-bit-rec*:
‹*signed-take-bit n a = (if n = 0 then − (a mod 2) else a mod 2 + 2 ∗ signed-take-bit (n − 1) (a div 2))*›
by (*cases n*) (*simp-all add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-eq-iff-take-bit-eq*:
‹*signed-take-bit n a = signed-take-bit n b ⟷ take-bit (Suc n) a = take-bit (Suc n) b*›
proof −
have ‹*bit (signed-take-bit n a) = bit (signed-take-bit n b) ⟷ bit (take-bit (Suc n) a) = bit (take-bit (Suc n) b)*›
by (*simp add*: *fun-eq-iff bit-signed-take-bit-iff bit-take-bit-iff not-le less-Suc-eq-le min-def*)
(*use bit-imp-possible-bit* **in** *fastforce*)
then show *?thesis*
by (*auto simp*: *fun-eq-iff intro*: *bit-eqI*)
qed

lemma *signed-take-bit-signed-take-bit* [*simp*]:
‹*signed-take-bit m (signed-take-bit n a) = signed-take-bit (min m n) a*›
by (*auto simp*: *bit-eq-iff bit-simps ac-simps*)

lemma *signed-take-bit-take-bit*:
‹*signed-take-bit m (take-bit n a) = (if n ≤ m then take-bit n else signed-take-bit m) a*›
by (*rule bit-eqI*) (*auto simp*: *bit-signed-take-bit-iff min-def bit-take-bit-iff*)

lemma *take-bit-signed-take-bit*:
‹*take-bit m (signed-take-bit n a) = take-bit m a*› **if** ‹*m ≤ Suc n*›
using *that* **by** (*rule le-SucE*; *intro bit-eqI*)

(*auto simp*: *bit-take-bit-iff bit-signed-take-bit-iff min-def less-Suc-eq*)

lemma *signed-take-bit-eq-take-bit-add*:
‹*signed-take-bit n k = take-bit (Suc n) k + NOT (mask (Suc n)) * of-bool (bit k n)*›
proof (*cases* ‹*bit k n*›)
case *False*
show *?thesis*
by (*rule bit-eqI*) (*simp add*: *False bit-simps min-def less-Suc-eq*)
next
case *True*
have ‹*signed-take-bit n k = take-bit (Suc n) k OR NOT (mask (Suc n))*›
by (*rule bit-eqI*) (*auto simp*: *bit-signed-take-bit-iff min-def bit-take-bit-iff bit-or-iff bit-not-iff bit-mask-iff less-Suc-eq True*)
also have ‹*... = take-bit (Suc n) k + NOT (mask (Suc n))*›
by (*simp add*: *disjunctive-add-eq-or bit-eq-iff bit-simps*)
finally show *?thesis*
by (*simp add*: *True*)
qed

lemma *signed-take-bit-eq-take-bit-minus*:
‹*signed-take-bit n k = take-bit (Suc n) k - 2 ^ Suc n * of-bool (bit k n)*›
by (*simp add*: *signed-take-bit-eq-take-bit-add flip*: *minus-exp-eq-not-mask*)

end

Modulus centered around 0

lemma *signed-take-bit-eq-concat-bit*:
‹*signed-take-bit n k = concat-bit n k (- of-bool (bit k n))*›
by (*simp add*: *concat-bit-def signed-take-bit-def*)

lemma *signed-take-bit-add*:
‹*signed-take-bit n (signed-take-bit n k + signed-take-bit n l) = signed-take-bit n (k + l)*›
for *k l* :: *int*
proof −
have ‹*take-bit (Suc n)*
(*take-bit (Suc n) (signed-take-bit n k) +*
take-bit (Suc n) (signed-take-bit n l)) =
take-bit (Suc n) (k + l)›
by (*simp add*: *take-bit-signed-take-bit take-bit-add*)
then show *?thesis*
by (*simp only*: *signed-take-bit-eq-iff-take-bit-eq take-bit-add*)
qed

lemma *signed-take-bit-diff*:
‹*signed-take-bit n (signed-take-bit n k - signed-take-bit n l) = signed-take-bit n (k - l)*›
for *k l* :: *int*

proof −
have ‹*take-bit* (*Suc n*)
(*take-bit* (*Suc n*) (*signed-take-bit n k*) −
take-bit (*Suc n*) (*signed-take-bit n l*)) =
take-bit (*Suc n*) (*k* − *l*)›
by (*simp add*: *take-bit-signed-take-bit take-bit-diff*)
then show *?thesis*
by (*simp only*: *signed-take-bit-eq-iff-take-bit-eq take-bit-diff*)
qed

lemma *signed-take-bit-minus*:
‹*signed-take-bit n* (− *signed-take-bit n k*) = *signed-take-bit n* (− *k*)›
for *k* :: *int*
proof −
have ‹*take-bit* (*Suc n*)
(− *take-bit* (*Suc n*) (*signed-take-bit n k*)) =
take-bit (*Suc n*) (− *k*)›
by (*simp add*: *take-bit-signed-take-bit take-bit-minus*)
then show *?thesis*
by (*simp only*: *signed-take-bit-eq-iff-take-bit-eq take-bit-minus*)
qed

lemma *signed-take-bit-mult*:
‹*signed-take-bit n* (*signed-take-bit n k* ∗ *signed-take-bit n l*) = *signed-take-bit n* (*k* ∗ *l*)›
for *k l* :: *int*
proof −
have ‹*take-bit* (*Suc n*)
(*take-bit* (*Suc n*) (*signed-take-bit n k*) ∗
take-bit (*Suc n*) (*signed-take-bit n l*)) =
take-bit (*Suc n*) (*k* ∗ *l*)›
by (*simp add*: *take-bit-signed-take-bit take-bit-mult*)
then show *?thesis*
by (*simp only*: *signed-take-bit-eq-iff-take-bit-eq take-bit-mult*)
qed

lemma *signed-take-bit-eq-take-bit-shift*:
‹*signed-take-bit n k* = *take-bit* (*Suc n*) (*k* + 2 ^ *n*) − 2 ^ *n*› (**is** ‹*?lhs* = *?rhs*›)
for *k* :: *int*
proof −
have ‹*take-bit n k AND* 2 ^ *n* = 0›
by (*rule bit-eqI*) (*simp add*: *bit-simps*)
then have ∗: ‹*take-bit n k OR* 2 ^ *n* = *take-bit n k* + 2 ^ *n*›
by (*simp add*: *disjunctive-add-eq-or*)
have ‹*take-bit n k* − 2 ^ *n* = *take-bit n k* + *NOT* (*mask n*)›
by (*simp add*: *minus-exp-eq-not-mask*)
also have ‹. . . = *take-bit n k OR NOT* (*mask n*)›
by (*rule disjunctive-add-eq-or*) (*simp add*: *bit-eq-iff bit-simps*)
finally have ∗∗: ‹*take-bit n k* − 2 ^ *n* = *take-bit n k OR NOT* (*mask n*)› **.**

have ‹*take-bit* (*Suc n*) (*k* + *2* ^ *n*) = *take-bit* (*Suc n*) (*take-bit* (*Suc n*) *k* + *take-bit* (*Suc n*) (*2* ^ *n*))›
by (*simp only*: *take-bit-add*)
also have ‹*take-bit* (*Suc n*) *k* = *2* ^ *n* * *of-bool* (*bit k n*) + *take-bit n k*›
by (*simp add*: *take-bit-Suc-from-most*)
finally have ‹*take-bit* (*Suc n*) (*k* + *2* ^ *n*) = *take-bit* (*Suc n*) (*2* ^ (*n* + *of-bool* (*bit k n*)) + *take-bit n k*)›
by (*simp add*: *ac-simps*)
also have ‹*2* ^ (*n* + *of-bool* (*bit k n*)) + *take-bit n k* = *2* ^ (*n* + *of-bool* (*bit k n*)) *OR take-bit n k*›
by (*rule disjunctive-add-eq-or*, *rule bit-eqI*) (*simp add*: *bit-simps*)
finally show *?thesis*
using * ** **by** (*simp add*: *signed-take-bit-def concat-bit-Suc min-def ac-simps*)
qed

lemma *signed-take-bit-nonnegative-iff* [*simp*]:
‹*0* ≤ *signed-take-bit n k* ⟷ ¬ *bit k n*›
for *k* :: *int*
by (*simp add*: *signed-take-bit-def not-less concat-bit-def*)

lemma *signed-take-bit-negative-iff* [*simp*]:
‹*signed-take-bit n k* < *0* ⟷ *bit k n*›
for *k* :: *int*
by (*simp add*: *signed-take-bit-def not-less concat-bit-def*)

lemma *signed-take-bit-int-greater-eq-minus-exp* [*simp*]:
‹− (*2* ^ *n*) ≤ *signed-take-bit n k*›
for *k* :: *int*
by (*simp add*: *signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-int-less-exp* [*simp*]:
‹*signed-take-bit n k* < *2* ^ *n*›
for *k* :: *int*
using *take-bit-int-less-exp* [*of* ‹*Suc n*›]
by (*simp add*: *signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-int-eq-self-iff*:
‹*signed-take-bit n k* = *k* ⟷ − (*2* ^ *n*) ≤ *k* ∧ *k* < *2* ^ *n*›
for *k* :: *int*
by (*auto simp*: *signed-take-bit-eq-take-bit-shift take-bit-int-eq-self-iff algebra-simps*)

lemma *signed-take-bit-int-eq-self*:
‹*signed-take-bit n k* = *k*› **if** ‹− (*2* ^ *n*) ≤ *k*› ‹*k* < *2* ^ *n*›
for *k* :: *int*
using *that* **by** (*simp add*: *signed-take-bit-int-eq-self-iff*)

lemma *signed-take-bit-int-less-eq-self-iff*:
‹*signed-take-bit n k* ≤ *k* ⟷ − (*2* ^ *n*) ≤ *k*›
for *k* :: *int*

by (*simp add*: *signed-take-bit-eq-take-bit-shift take-bit-int-less-eq-self-iff algebra-simps*)
linarith

lemma *signed-take-bit-int-less-self-iff*:
‹*signed-take-bit n k* < *k* ⟷ *2* ^ *n* ≤ *k*›
for *k* :: *int*
by (*simp add*: *signed-take-bit-eq-take-bit-shift take-bit-int-less-self-iff algebra-simps*)

lemma *signed-take-bit-int-greater-self-iff*:
‹*k* < *signed-take-bit n k* ⟷ *k* < − (*2* ^ *n*)›
for *k* :: *int*
by (*simp add*: *signed-take-bit-eq-take-bit-shift take-bit-int-greater-self-iff algebra-simps*)
linarith

lemma *signed-take-bit-int-greater-eq-self-iff*:
‹*k* ≤ *signed-take-bit n k* ⟷ *k* < *2* ^ *n*›
for *k* :: *int*
by (*simp add*: *signed-take-bit-eq-take-bit-shift take-bit-int-greater-eq-self-iff algebra-simps*)

lemma *signed-take-bit-int-greater-eq*:
‹*k* + *2* ^ *Suc n* ≤ *signed-take-bit n k*› **if** ‹*k* < − (*2* ^ *n*)›
for *k* :: *int*
using *that take-bit-int-greater-eq* [*of* ‹*k* + *2* ^ *n*› ‹*Suc n*›]
by (*simp add*: *signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-int-less-eq*:
‹*signed-take-bit n k* ≤ *k* − *2* ^ *Suc n*› **if** ‹*k* ≥ *2* ^ *n*›
for *k* :: *int*
using *that take-bit-int-less-eq* [*of* ‹*Suc n*› ‹*k* + *2* ^ *n*›]
by (*simp add*: *signed-take-bit-eq-take-bit-shift*)

lemma *signed-take-bit-Suc-sgn-eq* [*simp*]:
‹*signed-take-bit* (*Suc n*) (*sgn k*) = *sgn k*› **for** *k* :: *int*
by (*simp add*: *sgn-if*)

lemma *signed-take-bit-Suc-bit0* [*simp*]:
‹*signed-take-bit* (*Suc n*) (*numeral* (*Num.Bit0 k*)) = *signed-take-bit n* (*numeral k*) ∗ (*2* :: *int*)›
by (*simp add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-bit1* [*simp*]:
‹*signed-take-bit* (*Suc n*) (*numeral* (*Num.Bit1 k*)) = *signed-take-bit n* (*numeral k*) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-minus-bit0* [*simp*]:
‹*signed-take-bit* (*Suc n*) (− *numeral* (*Num.Bit0 k*)) = *signed-take-bit n* (− *numeral k*) ∗ (*2* :: *int*)›

by (*simp add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-Suc-minus-bit1* [*simp*]:
‹*signed-take-bit* (*Suc n*) (− *numeral* (*Num.Bit1 k*)) = *signed-take-bit n* (− *numeral k* − *1*) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *signed-take-bit-Suc*)

lemma *signed-take-bit-numeral-bit0* [*simp*]:
‹*signed-take-bit* (*numeral l*) (*numeral* (*Num.Bit0 k*)) = *signed-take-bit* (*pred-numeral l*) (*numeral k*) ∗ (*2* :: *int*)›
by (*simp add*: *signed-take-bit-rec*)

lemma *signed-take-bit-numeral-bit1* [*simp*]:
‹*signed-take-bit* (*numeral l*) (*numeral* (*Num.Bit1 k*)) = *signed-take-bit* (*pred-numeral l*) (*numeral k*) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *signed-take-bit-rec*)

lemma *signed-take-bit-numeral-minus-bit0* [*simp*]:
‹*signed-take-bit* (*numeral l*) (− *numeral* (*Num.Bit0 k*)) = *signed-take-bit* (*pred-numeral l*) (− *numeral k*) ∗ (*2* :: *int*)›
by (*simp add*: *signed-take-bit-rec*)

lemma *signed-take-bit-numeral-minus-bit1* [*simp*]:
‹*signed-take-bit* (*numeral l*) (− *numeral* (*Num.Bit1 k*)) = *signed-take-bit* (*pred-numeral l*) (− *numeral k* − *1*) ∗ *2* + (*1* :: *int*)›
by (*simp add*: *signed-take-bit-rec*)

lemma *signed-take-bit-code* [*code*]:
‹*signed-take-bit n a* =
(*let l* = *take-bit* (*Suc n*) *a*
in if bit l n then l + *push-bit* (*Suc n*) (− *1*) *else l*)›
by (*simp add*: *signed-take-bit-eq-take-bit-add bit-simps*)

68.11 Key ideas of bit operations

When formalizing bit operations, it is tempting to represent bit values as explicit lists over a binary type. This however is a bad idea, mainly due to the inherent ambiguities in representation concerning repeating leading bits.

Hence this approach avoids such explicit lists altogether following an algebraic path:

- Bit values are represented by numeric types: idealized unbounded bit values can be represented by type *int*, bounded bit values by quotient types over *int*.
- (A special case are idealized unbounded bit values ending in *0* which can be represented by type *nat* but only support a restricted set of

operations).

- From this idea follows that
  - multiplication by *2* is a bit shift to the left and
  - division by *2* is a bit shift to the right.
- Concerning bounded bit values, iterated shifts to the left may result in eliminating all bits by shifting them all beyond the boundary. The property $2^n \neq 0$ represents that n is *not* beyond that boundary.
- The projection on a single bit is then $bit\ a\ n = odd\ (a\ div\ 2^n)$.
- This leads to the most fundamental properties of bit values:
  - Equality rule: $(\bigwedge n.\ possible\text{-}bit\ TYPE(int)\ n \Longrightarrow bit\ a\ n = bit\ b\ n) \Longrightarrow a = b$
  - Induction rule: $[\![\bigwedge a.\ a\ div\ 2 = a \Longrightarrow P\ a;\ \bigwedge a\ b.\ [\![P\ a;\ (of\text{-}bool\ b + 2 * a)\ div\ 2 = a]\!] \Longrightarrow P\ (of\text{-}bool\ b + 2 * a)]\!] \Longrightarrow P\ a$
- Typical operations are characterized as follows:
  - Singleton nth bit: 2^n
  - Bit mask upto bit n: $mask\ n = 2^n - 1$
  - Left shift: $push\text{-}bit\ n\ a = a * 2^n$
  - Right shift: $drop\text{-}bit\ n\ a = a\ div\ 2^n$
  - Truncation: $take\text{-}bit\ n\ a = a\ mod\ 2^n$
  - Negation: $bit\ (NOT\ a)\ n = (possible\text{-}bit\ TYPE(int)\ n \wedge \neg\ bit\ a\ n)$
  - And: $bit\ (a\ AND\ b)\ n = (bit\ a\ n \wedge bit\ b\ n)$
  - Or: $bit\ (a\ OR\ b)\ n = (bit\ a\ n \vee bit\ b\ n)$
  - Xor: $bit\ (a\ XOR\ b)\ n = (bit\ a\ n \neq bit\ b\ n)$
  - Set a single bit: $set\text{-}bit\ n\ a = a\ OR\ push\text{-}bit\ n\ 1$
  - Unset a single bit: $unset\text{-}bit\ n\ a = a\ AND\ NOT\ (push\text{-}bit\ n\ 1)$
  - Flip a single bit: $flip\text{-}bit\ n\ a = a\ XOR\ push\text{-}bit\ n\ 1$
  - Signed truncation, or modulus centered around *0*: $signed\text{-}take\text{-}bit\ n\ a = take\text{-}bit\ n\ a\ OR\ of\text{-}bool\ (bit\ a\ n) * NOT\ (mask\ n)$
  - Bit concatenation: $concat\text{-}bit\ n\ k\ l = take\text{-}bit\ n\ k\ OR\ push\text{-}bit\ n\ l$
  - (Bounded) conversion from and to a list of bits: $horner\text{-}sum\ of\text{-}bool\ 2\ (map\ (bit\ a)\ [0..<n]) = take\text{-}bit\ n\ a$

68.12 Lemma duplicates and other

context *semiring-bits*
begin

lemma *exp-div-exp-eq*:
‹2 ^ m div 2 ^ n = of-bool (2 ^ m ≠ 0 ∧ m ≥ n) * 2 ^ (m − n)›
using *bit-exp-iff div-exp-eq*
by (*intro bit-eqI*) (*auto simp*: *bit-iff-odd possible-bit-def*)

lemma *bits-1-div-2*:
‹1 div 2 = 0›
by (*fact half-1*)

lemma *bits-1-div-exp*:
‹1 div 2 ^ n = of-bool (n = 0)›
using *div-exp-eq* [*of 1 1*] **by** (*cases n*) *simp-all*

lemma *exp-add-not-zero-imp*:
‹2 ^ m ≠ 0› **and** ‹2 ^ n ≠ 0› **if** ‹2 ^ (m + n) ≠ 0›
proof −
have ‹¬ (2 ^ m = 0 ∨ 2 ^ n = 0)›
proof (*rule notI*)
assume ‹2 ^ m = 0 ∨ 2 ^ n = 0›
then have ‹2 ^ (m + n) = 0›
by (*rule disjE*) (*simp-all add*: *power-add*)
with *that* **show** *False* **..**
qed
then show ‹2 ^ m ≠ 0› **and** ‹2 ^ n ≠ 0›
by *simp-all*
qed

lemma
exp-add-not-zero-imp-left: ‹2 ^ m ≠ 0›
and *exp-add-not-zero-imp-right*: ‹2 ^ n ≠ 0›
if ‹2 ^ (m + n) ≠ 0›
proof −
have ‹¬ (2 ^ m = 0 ∨ 2 ^ n = 0)›
proof (*rule notI*)
assume ‹2 ^ m = 0 ∨ 2 ^ n = 0›
then have ‹2 ^ (m + n) = 0›
by (*rule disjE*) (*simp-all add*: *power-add*)
with *that* **show** *False* **..**
qed
then show ‹2 ^ m ≠ 0› **and** ‹2 ^ n ≠ 0›
by *simp-all*
qed

lemma *exp-not-zero-imp-exp-diff-not-zero*:
‹2 ^ (n − m) ≠ 0› **if** ‹2 ^ n ≠ 0›

proof (*cases* ‹$m \leq n$›)
case *True*
moreover define q **where** ‹$q = n - m$›
ultimately have ‹$n = m + q$›
by *simp*
with *that* **show** *?thesis*
by (*simp add*: *exp-add-not-zero-imp-right*)
next
case *False*
with *that* **show** *?thesis*
by *simp*
qed

lemma *exp-eq-0-imp-not-bit*:
‹$\neg$ *bit a n*› **if** ‹$2 \mathbin{\hat{}} n = 0$›
using *that* **by** (*simp add*: *bit-iff-odd*)

lemma *bit-disjunctive-add-iff*:
‹*bit* ($a + b$) $n \longleftrightarrow$ *bit a n* $\vee$ *bit b n*›
if ‹$\bigwedge n.$ $\neg$ *bit a n* $\vee$ $\neg$ *bit b n*›
proof (*cases* ‹*possible-bit TYPE*('*a*) *n*›)
case *False*
then show *?thesis*
by (*auto dest*: *impossible-bit*)
next
case *True*
with *that* **show** *?thesis* **proof** (*induction n arbitrary*: *a b*)
case *0*
from *0.prems*(*1*) [*of 0*] **show** *?case*
by (*auto simp*: *bit-0*)
next
case (*Suc n*)
from *Suc.prems*(*1*) [*of 0*] **have** *even*: ‹*even a* $\vee$ *even b*›
by (*auto simp*: *bit-0*)
have *bit*: ‹$\neg$ *bit* (*a div 2*) *n* $\vee$ $\neg$ *bit* (*b div 2*) *n*› **for** *n*
using *Suc.prems*(*1*) [*of* ‹*Suc n*›] **by** (*simp add*: *bit-Suc*)
from *Suc.prems*(*2*) **have** ‹*possible-bit TYPE*('*a*) (*Suc n*)› ‹*possible-bit TYPE*('*a*) *n*›
by (*simp-all add*: *possible-bit-less-imp*)
have ‹$a + b = (a \mathbin{div} 2 * 2 + a \bmod 2) + (b \mathbin{div} 2 * 2 + b \bmod 2)$›
using *div-mult-mod-eq* [*of a 2*] *div-mult-mod-eq* [*of b 2*] **by** *simp*
also have ‹$\ldots$ = *of-bool* (*odd a* $\vee$ *odd b*) + $2 * (a \mathbin{div} 2 + b \mathbin{div} 2)$›
using *even* **by** (*auto simp*: *algebra-simps mod2-eq-if*)
finally have ‹*bit* ($(a + b) \mathbin{div} 2$) *n* $\longleftrightarrow$ *bit* ($a \mathbin{div} 2 + b \mathbin{div} 2$) *n*›
using ‹*possible-bit TYPE*('*a*) (*Suc n*)› **by** *simp* (*simp-all flip*: *bit-Suc add*: *bit-double-iff possible-bit-def*)
also have ‹$\ldots$ $\longleftrightarrow$ *bit* (*a div 2*) *n* $\vee$ *bit* (*b div 2*) *n*›
using *bit* ‹*possible-bit TYPE*('*a*) *n*› **by** (*rule Suc.IH*)
finally show *?case*

by (*simp add*: *bit-Suc*)
qed
qed

end

context *semiring-bit-operations*
begin

lemma *even-mask-div-iff*:
‹*even* ((2 ^ *m* − *1*) *div* 2 ^ *n*) ⟷ 2 ^ *n* = *0* ∨ *m* ≤ *n*›
using *bit-mask-iff* [*of m n*] **by** (*auto simp*: *mask-eq-exp-minus-1 bit-iff-odd possible-bit-def*)

lemma *mod-exp-eq*:
‹*a mod* 2 ^ *m mod* 2 ^ *n* = *a mod* 2 ^ *min m n*›
by (*simp flip*: *take-bit-eq-mod add*: *ac-simps*)

lemma *mult-exp-mod-exp-eq*:
‹*m* ≤ *n* ⟹ (*a* ∗ 2 ^ *m*) *mod* (2 ^ *n*) = (*a mod* 2 ^ (*n* − *m*)) ∗ 2 ^ *m*›
by (*simp flip*: *push-bit-eq-mult take-bit-eq-mod add*: *push-bit-take-bit*)

lemma *div-exp-mod-exp-eq*:
‹*a div* 2 ^ *n mod* 2 ^ *m* = *a mod* (2 ^ (*n* + *m*)) *div* 2 ^ *n*›
by (*simp flip*: *drop-bit-eq-div take-bit-eq-mod add*: *drop-bit-take-bit*)

lemma *even-mult-exp-div-exp-iff*:
‹*even* (*a* ∗ 2 ^ *m div* 2 ^ *n*) ⟷ *m* > *n* ∨ 2 ^ *n* = *0* ∨ (*m* ≤ *n* ∧ *even* (*a div* 2 ^ (*n* − *m*)))›
by (*simp flip*: *push-bit-eq-mult drop-bit-eq-div add*: *even-drop-bit-iff-not-bit bit-simps possible-bit-def*) *auto*

lemma *mod-exp-div-exp-eq-0*:
‹*a mod* 2 ^ *n div* 2 ^ *n* = *0*›
by (*simp flip*: *take-bit-eq-mod drop-bit-eq-div add*: *drop-bit-take-bit*)

lemmas *bits-one-mod-two-eq-one* = *one-mod-two-eq-one*

lemmas *set-bit-def* = *set-bit-eq-or*

lemmas *unset-bit-def* = *unset-bit-eq-and-not*

lemmas *flip-bit-def* = *flip-bit-eq-xor*

lemma *disjunctive-add*:
‹*a* + *b* = *a OR b*› **if** ‹⋀*n*. ¬ *bit a n* ∨ ¬ *bit b n*›
by (*rule disjunctive-add-eq-or*) (*use that* **in** ‹*simp add*: *bit-eq-iff bit-simps*›)

lemma *even-mod-exp-div-exp-iff*:

‹*even* (*a mod 2* ^ *m div 2* ^ *n*) ⟷ *m* ≤ *n* ∨ *even* (*a div 2* ^ *n*)›
by (*auto simp*: *even-drop-bit-iff-not-bit bit-simps simp flip*: *drop-bit-eq-div take-bit-eq-mod*)

end

context *ring-bit-operations*
begin

lemma *disjunctive-diff*:
‹*a* − *b* = *a AND NOT b*› **if** ‹⋀*n*. *bit b n* ⟹ *bit a n*›
proof −
have ‹*NOT a* + *b* = *NOT a OR b*›
by (*rule disjunctive-add*) (*auto simp*: *bit-not-iff dest*: *that*)
then have ‹*NOT* (*NOT a* + *b*) = *NOT* (*NOT a OR b*)›
by *simp*
then show *?thesis*
by (*simp add*: *not-add-distrib*)
qed

end

lemma *and-nat-rec*:
‹*m AND n* = *of-bool* (*odd m* ∧ *odd n*) + *2* ∗ ((*m div 2*) *AND* (*n div 2*))› **for** *m n* :: *nat*
by (*fact and-rec*)

lemma *or-nat-rec*:
‹*m OR n* = *of-bool* (*odd m* ∨ *odd n*) + *2* ∗ ((*m div 2*) *OR* (*n div 2*))› **for** *m n* :: *nat*
by (*fact or-rec*)

lemma *xor-nat-rec*:
‹*m XOR n* = *of-bool* (*odd m* ≠ *odd n*) + *2* ∗ ((*m div 2*) *XOR* (*n div 2*))› **for** *m n* :: *nat*
by (*fact xor-rec*)

lemma *bit-push-bit-iff-nat*:
‹*bit* (*push-bit m q*) *n* ⟷ *m* ≤ *n* ∧ *bit q* (*n* − *m*)› **for** *q* :: *nat*
by (*fact bit-push-bit-iff'*)

lemma *mask-half-int*:
‹*mask n div 2* = (*mask* (*n* − *1*) :: *int*)›
by (*fact mask-half*)

lemma *not-int-rec*:
‹*NOT k* = *of-bool* (*even k*) + *2* ∗ *NOT* (*k div 2*)› **for** *k* :: *int*
by (*fact not-rec*)

lemma *even-not-iff-int*:

‹*even* (*NOT k*) ⟷ *odd k*› **for** *k* :: *int*
by (*fact even-not-iff*)

lemma *bit-not-int-iff'*:
‹*bit* (− *k* − *1*) *n* ⟷ ¬ *bit k n*› **for** *k* :: *int*
by (*simp flip*: *not-eq-complement add*: *bit-simps*)

lemmas *and-int-rec* = *and-int.rec*

lemma *even-and-iff-int*:
‹*even* (*k AND l*) ⟷ *even k* ∨ *even l*› **for** *k l* :: *int*
by (*fact even-and-iff*)

lemmas *bit-and-int-iff* = *and-int.bit-iff*

lemmas *or-int-rec* = *or-int.rec*

lemmas *bit-or-int-iff* = *or-int.bit-iff*

lemmas *xor-int-rec* = *xor-int.rec*

lemmas *bit-xor-int-iff* = *xor-int.bit-iff*

lemma *drop-bit-push-bit-int*:
‹*drop-bit m* (*push-bit n k*) = *drop-bit* (*m* − *n*) (*push-bit* (*n* − *m*) *k*)› **for** *k* :: *int*
by (*fact drop-bit-push-bit*)

lemma *bit-push-bit-iff-int*:
‹*bit* (*push-bit m k*) *n* ⟷ *m* ≤ *n* ∧ *bit k* (*n* − *m*)› **for** *k* :: *int*
by (*fact bit-push-bit-iff'*)

bundle *bit-operations-syntax*
begin
notation
not (‹*NOT*›)
and *and* (**infixr** ‹*AND*› *64*)
and *or* (**infixr** ‹*OR*› *59*)
and *xor* (**infixr** ‹*XOR*› *59*)
end

unbundle *no bit-operations-syntax*

end

69 Numeric types for code generation onto target language numerals only

theory *Code-Numeral*

imports *Lifting Bit-Operations*
begin

69.1 Type of target language integers

typedef *integer* = *UNIV* :: *int set*
  morphisms *int-of-integer integer-of-int* **..**

setup-lifting *type-definition-integer*

lemma *integer-eq-iff*:
  $k = l \longleftrightarrow$ *int-of-integer* $k =$ *int-of-integer* l
  by *transfer rule*

lemma *integer-eqI*:
  int-of-integer $k =$ *int-of-integer* $l \Longrightarrow k = l$
  using *integer-eq-iff* [*of k l*] **by** *simp*

lemma *int-of-integer-integer-of-int* [*simp*]:
  int-of-integer (*integer-of-int* k) $= k$
  by *transfer rule*

lemma *integer-of-int-int-of-integer* [*simp*]:
  integer-of-int (*int-of-integer* k) $= k$
  by *transfer rule*

instantiation *integer* :: *ring-1*
begin

lift-definition *zero-integer* :: *integer*
  is *0* :: *int*
  .

declare *zero-integer.rep-eq* [*simp*]

lift-definition *one-integer* :: *integer*
  is *1* :: *int*
  .

declare *one-integer.rep-eq* [*simp*]

lift-definition *plus-integer* :: *integer* ⇒ *integer* ⇒ *integer*
  is *plus* :: *int* ⇒ *int* ⇒ *int*
  .

declare *plus-integer.rep-eq* [*simp*]

lift-definition *uminus-integer* :: *integer* ⇒ *integer*
  is *uminus* :: *int* ⇒ *int*

.

declare *uminus-integer.rep-eq* [*simp*]

lift-definition *minus-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *minus* :: *int* ⇒ *int* ⇒ *int*
.

declare *minus-integer.rep-eq* [*simp*]

lift-definition *times-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *times* :: *int* ⇒ *int* ⇒ *int*
.

declare *times-integer.rep-eq* [*simp*]

instance proof
qed (*transfer*, *simp add*: *algebra-simps*)+

end

instance *integer* :: *Rings.dvd* **..**

context
includes *lifting-syntax*
notes *transfer-rule-numeral* [*transfer-rule*]
begin

lemma [*transfer-rule*]:
(*pcr-integer* ===> *pcr-integer* ===> (⟷)) (*dvd*) (*dvd*)
by (*unfold dvd-def*) *transfer-prover*

lemma [*transfer-rule*]:
((⟷) ===> *pcr-integer*) *of-bool of-bool*
by (*unfold of-bool-def*) *transfer-prover*

lemma [*transfer-rule*]:
((=) ===> *pcr-integer*) *int of-nat*
by (*rule transfer-rule-of-nat*) *transfer-prover*+

lemma [*transfer-rule*]:
((=) ===> *pcr-integer*) (λ*k*. *k*) *of-int*
proof −
have ((=) ===> *pcr-integer*) *of-int of-int*
by (*rule transfer-rule-of-int*) *transfer-prover*+
then show *?thesis* **by** (*simp add*: *id-def*)
qed

lemma [*transfer-rule*]:

((=) ===> *pcr-integer*) *numeral numeral*
by *transfer-prover*

lemma [*transfer-rule*]:
((=) ===> (=) ===> *pcr-integer*) *Num.sub Num.sub*
by (*unfold Num.sub-def*) *transfer-prover*

lemma [*transfer-rule*]:
(*pcr-integer* ===> (=) ===> *pcr-integer*) (^) (^)
by (*unfold power-def*) *transfer-prover*

end

lemma *int-of-integer-of-nat* [*simp*]:
int-of-integer (*of-nat n*) = *of-nat n*
by *transfer rule*

lift-definition *integer-of-nat* :: *nat* ⇒ *integer*
is *of-nat* :: *nat* ⇒ *int*
.

lemma *integer-of-nat-eq-of-nat* [*code*]:
integer-of-nat = *of-nat*
by *transfer rule*

lemma *int-of-integer-integer-of-nat* [*simp*]:
int-of-integer (*integer-of-nat n*) = *of-nat n*
by *transfer rule*

lift-definition *nat-of-integer* :: *integer* ⇒ *nat*
is *Int.nat*
.

lemma *nat-of-integer-0* [*simp*]:
‹*nat-of-integer 0* = *0*›
by *transfer simp*

lemma *nat-of-integer-1* [*simp*]:
‹*nat-of-integer 1* = *1*›
by *transfer simp*

lemma *nat-of-integer-numeral* [*simp*]:
‹*nat-of-integer* (*numeral n*) = *numeral n*›
by *transfer simp*

lemma *nat-of-integer-of-nat* [*simp*]:
nat-of-integer (*of-nat n*) = *n*
by *transfer simp*

lemma *int-of-integer-of-int* [*simp*]:
int-of-integer (*of-int k*) = *k*
by *transfer simp*

lemma *nat-of-integer-integer-of-nat* [*simp*]:
nat-of-integer (*integer-of-nat n*) = *n*
by *transfer simp*

lemma *integer-of-int-eq-of-int* [*simp*, *code-abbrev*]:
integer-of-int = *of-int*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *of-int-integer-of* [*simp*]:
of-int (*int-of-integer k*) = (*k* :: *integer*)
by *transfer rule*

lemma *int-of-integer-numeral* [*simp*]:
int-of-integer (*numeral k*) = *numeral k*
by *transfer rule*

lemma *int-of-integer-sub* [*simp*]:
int-of-integer (*Num.sub k l*) = *Num.sub k l*
by *transfer rule*

definition *integer-of-num* :: *num* ⇒ *integer*
where [*simp*]: *integer-of-num* = *numeral*

lemma *integer-of-num* [*code*]:
integer-of-num Num.One = *1*
integer-of-num (*Num.Bit0 n*) = (*let k* = *integer-of-num n in k* + *k*)
integer-of-num (*Num.Bit1 n*) = (*let k* = *integer-of-num n in k* + *k* + *1*)
by (*simp-all only*: *integer-of-num-def numeral.simps Let-def*)

lemma *integer-of-num-triv*:
integer-of-num Num.One = *1*
integer-of-num (*Num.Bit0 Num.One*) = *2*
by *simp-all*

instantiation *integer* :: *equal*
begin

lift-definition *equal-integer* :: ‹*integer* ⇒ *integer* ⇒ *bool*›
is ‹*HOL.equal* :: *int* ⇒ *int* ⇒ *bool*›
.

instance
by (*standard*; *transfer*) (*fact equal-eq*)

end

instantiation *integer* :: *linordered-idom*
begin

lift-definition *abs-integer* :: ‹*integer* ⇒ *integer*›
  is ‹*abs* :: *int* ⇒ *int*›
  .

declare *abs-integer.rep-eq* [*simp*]

lift-definition *sgn-integer* :: ‹*integer* ⇒ *integer*›
  is ‹*sgn* :: *int* ⇒ *int*›
  .

declare *sgn-integer.rep-eq* [*simp*]

lift-definition *less-eq-integer* :: ‹*integer* ⇒ *integer* ⇒ *bool*›
  is ‹*less-eq* :: *int* ⇒ *int* ⇒ *bool*›
  .

lemma *integer-less-eq-iff*:
  ‹*k* ≤ *l* ⟷ *int-of-integer k* ≤ *int-of-integer l*›
  by (*fact less-eq-integer.rep-eq*)

lift-definition *less-integer* :: ‹*integer* ⇒ *integer* ⇒ *bool*›
  is ‹*less* :: *int* ⇒ *int* ⇒ *bool*›
  .

lemma *integer-less-iff*:
  ‹*k* < *l* ⟷ *int-of-integer k* < *int-of-integer l*›
  by (*fact less-integer.rep-eq*)

instance
  by (*standard*; *transfer*)
    (*simp-all add*: *algebra-simps less-le-not-le* [*symmetric*] *mult-strict-right-mono linear*)

end

instance *integer* :: *discrete-linordered-semidom*
  by (*standard*; *transfer*)
    (*fact less-iff-succ-less-eq*)

context
  includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
  ‹(*pcr-integer* ===> *pcr-integer* ===> *pcr-integer*) *min min*›

by (*unfold min-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹(*pcr-integer* ===> *pcr-integer* ===> *pcr-integer*) *max max*›
by (*unfold max-def*) *transfer-prover*

end

lemma *int-of-integer-min* [*simp*]:
int-of-integer (*min k l*) = *min* (*int-of-integer k*) (*int-of-integer l*)
by *transfer rule*

lemma *int-of-integer-max* [*simp*]:
int-of-integer (*max k l*) = *max* (*int-of-integer k*) (*int-of-integer l*)
by *transfer rule*

lemma *nat-of-integer-non-positive* [*simp*]:
$k \leq 0 \Longrightarrow$ *nat-of-integer k* = *0*
by *transfer simp*

lemma *of-nat-of-integer* [*simp*]:
of-nat (*nat-of-integer k*) = *max 0 k*
by *transfer auto*

instantiation *integer* :: *unique-euclidean-ring*
begin

lift-definition *divide-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *divide* :: *int* ⇒ *int* ⇒ *int*
.

declare *divide-integer.rep-eq* [*simp*]

lift-definition *modulo-integer* :: *integer* ⇒ *integer* ⇒ *integer*
is *modulo* :: *int* ⇒ *int* ⇒ *int*
.

declare *modulo-integer.rep-eq* [*simp*]

lift-definition *euclidean-size-integer* :: *integer* ⇒ *nat*
is *euclidean-size* :: *int* ⇒ *nat*
.

declare *euclidean-size-integer.rep-eq* [*simp*]

lift-definition *division-segment-integer* :: *integer* ⇒ *integer*
is *division-segment* :: *int* ⇒ *int*
.

declare *division-segment-integer.rep-eq* [*simp*]

instance
apply (*standard*; *transfer*)
apply (*use mult-le-mono2* [*of 1*] **in** ‹*auto simp add*: *sgn-mult-abs abs-mult sgn-mult abs-mod-less sgn-mod nat-mult-distrib*
division-segment-mult division-segment-mod›)
apply (*simp add*: *division-segment-int-def split*: *if-splits*)
done

end

lemma [*code*]:
euclidean-size = *nat-of-integer* ∘ *abs*
by (*simp add*: *fun-eq-iff nat-of-integer.rep-eq*)

lemma [*code*]:
division-segment (*k* :: *integer*) = (*if k* ≥ *0 then 1 else* − *1*)
by *transfer* (*simp add*: *division-segment-int-def*)

instance *integer* :: *linordered-euclidean-semiring*
by (*standard*; *transfer*) (*simp-all add*: *of-nat-div division-segment-int-def*)

instantiation *integer* :: *ring-bit-operations*
begin

lift-definition *bit-integer* :: ‹*integer* ⇒ *nat* ⇒ *bool*›
is *bit* .

lift-definition *not-integer* :: ‹*integer* ⇒ *integer*›
is *not* .

lift-definition *and-integer* :: ‹*integer* ⇒ *integer* ⇒ *integer*›
is ‹*and*› .

lift-definition *or-integer* :: ‹*integer* ⇒ *integer* ⇒ *integer*›
is *or* .

lift-definition *xor-integer* :: ‹*integer* ⇒ *integer* ⇒ *integer*›
is *xor* .

lift-definition *mask-integer* :: ‹*nat* ⇒ *integer*›
is *mask* .

lift-definition *set-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *set-bit* .

lift-definition *unset-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *unset-bit* .

lift-definition *flip-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *flip-bit* .

lift-definition *push-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *push-bit* .

lift-definition *drop-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *drop-bit* .

lift-definition *take-bit-integer* :: ‹*nat* ⇒ *integer* ⇒ *integer*›
is *take-bit* .

instance by (*standard*; *transfer*)
(*fact bit-induct div-by-0 div-by-1 div-0 even-half-succ-eq*
half-div-exp-eq even-double-div-exp-iff bits-mod-div-trivial
bit-iff-odd push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod
and-rec or-rec xor-rec mask-eq-exp-minus-1
set-bit-eq-or unset-bit-eq-or-xor flip-bit-eq-xor not-eq-complement)+

end

instance *integer* :: *linordered-euclidean-semiring-bit-operations* **..**

instantiation *integer* :: *linordered-euclidean-semiring-division*
begin

definition *divmod-integer* :: *num* ⇒ *num* ⇒ *integer* × *integer*
where
divmod-integer'-def: *divmod-integer m n* = (*numeral m div numeral n*, *numeral m mod numeral n*)

definition *divmod-step-integer* :: *integer* ⇒ *integer* × *integer* ⇒ *integer* × *integer*
where
divmod-step-integer l qr = (*let* (*q*, *r*) = *qr*
in if |*l*| ≤ |*r*| *then* (2 ∗ *q* + 1, *r* − *l*)
else (2 ∗ *q*, *r*))

instance by *standard*
(*auto simp add*: *divmod-integer'-def divmod-step-integer-def integer-less-eq-iff*)

end

lemma *integer-of-nat-0*: *integer-of-nat 0* = *0*
by *transfer simp*

lemma *integer-of-nat-1*: *integer-of-nat 1* = *1*

by *transfer simp*

lemma *integer-of-nat-numeral*:
integer-of-nat (*numeral n*) = *numeral n*
by *transfer simp*

69.2 Code theorems for target language integers

Constructors

definition *Pos* :: *num* ⇒ *integer*
where
[*simp*, *code-post*]: *Pos* = *numeral*

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
‹((=) ===> *pcr-integer*) *numeral Pos›*
by *simp transfer-prover*

end

lemma *Pos-fold* [*code-unfold*]:
numeral Num.One = *Pos Num.One*
numeral (*Num.Bit0 k*) = *Pos* (*Num.Bit0 k*)
numeral (*Num.Bit1 k*) = *Pos* (*Num.Bit1 k*)
by *simp-all*

definition *Neg* :: *num* ⇒ *integer*
where
[*simp*, *code-abbrev*]: *Neg n* = − *Pos n*

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
‹((=) ===> *pcr-integer*) (λ*n*. − *numeral n*) *Neg›*
by (*unfold Neg-def*) *transfer-prover*

end

code-datatype *0*::*integer Pos Neg*

A further pair of constructors for generated computations

context
begin

qualified definition *positive* :: *num* ⇒ *integer*
where [*simp*]: *positive* = *numeral*

qualified definition *negative* :: *num* ⇒ *integer*
where [*simp*]: *negative* = *uminus* ∘ *numeral*

lemma [*code-computation-unfold*]:
numeral = *positive*
Pos = *positive*
Neg = *negative*
by (*simp-all add*: *fun-eq-iff*)

end

Auxiliary operations

lift-definition *dup* :: *integer* ⇒ *integer*
is λ*k*::*int*. *k* + *k*
.

lemma *dup-code* [*code*]:
dup 0 = 0
dup (*Pos n*) = *Pos* (*Num.Bit0 n*)
dup (*Neg n*) = *Neg* (*Num.Bit0 n*)
by (*transfer*; *simp only*: *numeral-Bit0 minus-add-distrib*)+

lift-definition *sub* :: *num* ⇒ *num* ⇒ *integer*
is λ*m n*. *numeral m* − *numeral n* :: *int*
.

lemma *sub-code* [*code*]:
sub Num.One Num.One = 0
sub (*Num.Bit0 m*) *Num.One* = *Pos* (*Num.BitM m*)
sub (*Num.Bit1 m*) *Num.One* = *Pos* (*Num.Bit0 m*)
sub Num.One (*Num.Bit0 n*) = *Neg* (*Num.BitM n*)
sub Num.One (*Num.Bit1 n*) = *Neg* (*Num.Bit0 n*)
sub (*Num.Bit0 m*) (*Num.Bit0 n*) = *dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit1 n*) = *dup* (*sub m n*)
sub (*Num.Bit1 m*) (*Num.Bit0 n*) = *dup* (*sub m n*) + 1
sub (*Num.Bit0 m*) (*Num.Bit1 n*) = *dup* (*sub m n*) − 1
by (*transfer*; *simp add*: *dbl-def dbl-inc-def dbl-dec-def*)+

Implementations

lemma *one-integer-code* [*code*, *code-unfold*]:
1 = *Pos Num.One*
by *simp*

lemma *plus-integer-code* [*code*]:
k + 0 = (*k*::*integer*)
0 + *l* = (*l*::*integer*)

Pos m + Pos n = Pos (m + n)
Pos m + Neg n = sub m n
Neg m + Pos n = sub n m
Neg m + Neg n = Neg (m + n)
by (*transfer*, *simp*)+

lemma *uminus-integer-code* [*code*]:
uminus 0 = (0::integer)
uminus (Pos m) = Neg m
uminus (Neg m) = Pos m
by *simp-all*

lemma *minus-integer-code* [*code*]:
k − 0 = (k::integer)
0 − l = uminus (l::integer)
Pos m − Pos n = sub m n
Pos m − Neg n = Pos (m + n)
Neg m − Pos n = Neg (m + n)
Neg m − Neg n = sub n m
by (*transfer*, *simp*)+

lemma *abs-integer-code* [*code*]:
|k| = (if (k::integer) < 0 then − k else k)
by *simp*

lemma *sgn-integer-code* [*code*]:
sgn k = (if k = 0 then 0 else if (k::integer) < 0 then − 1 else 1)
by *simp*

lemma *times-integer-code* [*code*]:
*k * 0 = (0::integer)*
*0 * l = (0::integer)*
*Pos m * Pos n = Pos (m * n)*
*Pos m * Neg n = Neg (m * n)*
*Neg m * Pos n = Neg (m * n)*
*Neg m * Neg n = Pos (m * n)*
by *simp-all*

definition *divmod-integer* :: *integer* ⇒ *integer* ⇒ *integer* × *integer*
where
divmod-integer k l = (k div l, k mod l)

lemma *fst-divmod-integer* [*simp*]:
fst (divmod-integer k l) = k div l
by (*simp add*: *divmod-integer-def*)

lemma *snd-divmod-integer* [*simp*]:
snd (divmod-integer k l) = k mod l
by (*simp add*: *divmod-integer-def*)

definition *divmod-abs* :: *integer* ⇒ *integer* ⇒ *integer* × *integer*
where
divmod-abs k l = (|*k*| *div* |*l*|, |*k*| *mod* |*l*|)

lemma *fst-divmod-abs* [*simp*]:
fst (*divmod-abs k l*) = |*k*| *div* |*l*|
by (*simp add*: *divmod-abs-def*)

lemma *snd-divmod-abs* [*simp*]:
snd (*divmod-abs k l*) = |*k*| *mod* |*l*|
by (*simp add*: *divmod-abs-def*)

declare *divmod-algorithm-code* [**where** *?'a* = *integer*,
folded integer-of-num-def, *unfolded integer-of-num-triv*,
code]

lemma *divmod-abs-code* [*code*]:
divmod-abs 0 j = (*0*, *0*)
divmod-abs j 0 = (*0*, |*j*|)
divmod-abs (*Pos k*) (*Pos l*) = *divmod k l*
divmod-abs (*Pos k*) (*Neg l*) = *divmod k l*
divmod-abs (*Neg k*) (*Pos l*) = *divmod k l*
divmod-abs (*Neg k*) (*Neg l*) = *divmod k l*
by (*simp-all add*: *prod-eq-iff*)

lemma *divmod-integer-eq-cases*:
divmod-integer k l =
(*if k* = *0 then* (*0*, *0*) *else if l* = *0 then* (*0*, *k*) *else*
(*apsnd* ∘ *times* ∘ *sgn*) *l* (*if sgn k* = *sgn l*
then divmod-abs k l
else (*let* (*r*, *s*) = *divmod-abs k l in*
if s = *0 then* (− *r*, *0*) *else* (− *r* − *1*, |*l*| − *s*))))
proof −
have ∗: *sgn k* = *sgn l* ⟷ *k* = *0* ∧ *l* = *0* ∨ *0* < *l* ∧ *0* < *k* ∨ *l* < *0* ∧ *k* < *0*
for *k l* :: *int*
by (*auto simp add*: *sgn-if*)
have ∗∗: − *k* = *l* ∗ *q* ⟷ *k* = − (*l* ∗ *q*) **for** *k l q* :: *int*
by *auto*
show *?thesis*
by (*simp add*: *divmod-integer-def divmod-abs-def*)
(*transfer*, *auto simp add*: ∗ ∗∗ *not-less zdiv-zminus1-eq-if zmod-zminus1-eq-if*
div-minus-right mod-minus-right)
qed

lemma *divmod-integer-code* [*code*]:
divmod-integer k l =
(*if k* = *0 then* (*0*, *0*)
else if l > *0 then*

```
        (if k > 0 then divmod-abs k l
         else case divmod-abs k l of (r, s) ⇒
              if s = 0 then (− r, 0) else (− r − 1, l − s))
  else if l = 0 then (0, k)
  else apsnd uminus
        (if k < 0 then divmod-abs k l
         else case divmod-abs k l of (r, s) ⇒
              if s = 0 then (− r, 0) else (− r − 1, − l − s)))
 by (cases l 0 :: integer rule: linorder-cases)
  (auto split: prod.splits simp add: divmod-integer-eq-cases)

lemma div-integer-code [code]:
 k div l = fst (divmod-integer k l)
 by simp

lemma mod-integer-code [code]:
 k mod l = snd (divmod-integer k l)
 by simp

context
 includes bit-operations-syntax
begin

lemma and-integer-code [code]:
 ‹0 AND k = 0›
 ‹k AND 0 = 0›
 ‹Neg Num.One AND k = k›
 ‹k AND Neg Num.One = k›
 ‹Pos Num.One AND Pos Num.One = Pos Num.One›
 ‹Pos Num.One AND Pos (Num.Bit0 n) = 0›
 ‹Pos (Num.Bit0 m) AND Pos Num.One = 0›
 ‹Pos Num.One AND Pos (Num.Bit1 n) = Pos Num.One›
 ‹Pos (Num.Bit1 m) AND Pos Num.One = Pos Num.One›
 ‹Pos (Num.Bit0 m) AND Pos (Num.Bit0 n) = dup (Pos m AND Pos n)›
 ‹Pos (Num.Bit0 m) AND Pos (Num.Bit1 n) = dup (Pos m AND Pos n)›
 ‹Pos (Num.Bit1 m) AND Pos (Num.Bit0 n) = dup (Pos m AND Pos n)›
 ‹Pos (Num.Bit1 m) AND Pos (Num.Bit1 n) = Pos Num.One + dup (Pos m
AND Pos n)›
 ‹Pos m AND Neg (num.Bit0 n) = (case and-not-num m (Num.BitM n) of None
⇒ 0 | Some n′ ⇒ Pos n′)›
 ‹Neg (num.Bit0 m) AND Pos n = (case and-not-num n (Num.BitM m) of None
⇒ 0 | Some n′ ⇒ Pos n′)›
 ‹Pos m AND Neg (num.Bit1 n) = (case and-not-num m (Num.Bit0 n) of None
⇒ 0 | Some n′ ⇒ Pos n′)›
 ‹Neg (num.Bit1 m) AND Pos n = (case and-not-num n (Num.Bit0 m) of None
⇒ 0 | Some n′ ⇒ Pos n′)›
 ‹Neg m AND Neg n = NOT (sub m Num.One OR sub n Num.One)›
  for k :: integer
 by (transfer; simp)+
```

lemma *or-integer-code* [*code*]:
‹*0 OR k = k*›
‹*k OR 0 = k*›
‹*Neg Num.One OR k = Neg Num.One*›
‹*k OR Neg Num.One = Neg Num.One*›
‹*Pos Num.One OR Pos Num.One = Pos Num.One*›
‹*Pos Num.One OR Pos (Num.Bit0 n) = Pos (Num.Bit1 n)*›
‹*Pos (Num.Bit0 m) OR Pos Num.One = Pos (Num.Bit1 m)*›
‹*Pos Num.One OR Pos (Num.Bit1 n) = Pos (Num.Bit1 n)*›
‹*Pos (Num.Bit1 m) OR Pos Num.One = Pos (Num.Bit1 m)*›
‹*Pos (Num.Bit0 m) OR Pos (Num.Bit0 n) = dup (Pos m OR Pos n)*›
‹*Pos (Num.Bit0 m) OR Pos (Num.Bit1 n) = Pos Num.One + dup (Pos m OR Pos n)*›
‹*Pos (Num.Bit1 m) OR Pos (Num.Bit0 n) = Pos Num.One + dup (Pos m OR Pos n)*›
‹*Pos (Num.Bit1 m) OR Pos (Num.Bit1 n) = Pos Num.One + dup (Pos m OR Pos n)*›
‹*Pos m OR Neg (num.Bit0 n) = Neg (or-not-num-neg m (Num.BitM n))*›
‹*Neg (num.Bit0 m) OR Pos n = Neg (or-not-num-neg n (Num.BitM m))*›
‹*Pos m OR Neg (num.Bit1 n) = Neg (or-not-num-neg m (Num.Bit0 n))*›
‹*Neg (num.Bit1 m) OR Pos n = Neg (or-not-num-neg n (Num.Bit0 m))*›
‹*Neg m OR Neg n = NOT (sub m Num.One AND sub n Num.One)*›
for *k* :: *integer*
by (*transfer*; *simp*)+

lemma *xor-integer-code* [*code*]:
‹*0 XOR k = k*›
‹*k XOR 0 = k*›
‹*Neg Num.One XOR k = NOT k*›
‹*k XOR Neg Num.One = NOT k*›
‹*Neg m XOR k = NOT (sub m num.One XOR k)*›
‹*k XOR Neg n = NOT (k XOR (sub n num.One))*›
‹*Pos Num.One XOR Pos Num.One = 0*›
‹*Pos Num.One XOR Pos (Num.Bit0 n) = Pos (Num.Bit1 n)*›
‹*Pos (Num.Bit0 m) XOR Pos Num.One = Pos (Num.Bit1 m)*›
‹*Pos Num.One XOR Pos (Num.Bit1 n) = Pos (Num.Bit0 n)*›
‹*Pos (Num.Bit1 m) XOR Pos Num.One = Pos (Num.Bit0 m)*›
‹*Pos (Num.Bit0 m) XOR Pos (Num.Bit0 n) = dup (Pos m XOR Pos n)*›
‹*Pos (Num.Bit0 m) XOR Pos (Num.Bit1 n) = Pos Num.One + dup (Pos m XOR Pos n)*›
‹*Pos (Num.Bit1 m) XOR Pos (Num.Bit0 n) = Pos Num.One + dup (Pos m XOR Pos n)*›
‹*Pos (Num.Bit1 m) XOR Pos (Num.Bit1 n) = dup (Pos m XOR Pos n)*›
for *k* :: *integer*
by (*transfer*; *simp*)+

lemma [*code*]:
‹*NOT k = - k - 1*› **for** *k* :: *integer*

by (*fact not-eq-complement*)

lemma [*code*]:
‹*bit k n* ⟷ *k AND push-bit n 1* ≠ (*0* :: *integer*)›
by (*simp add*: *and-exp-eq-0-iff-not-bit*)

lemma [*code*]:
‹*mask n* = *push-bit n 1* − (*1* :: *integer*)›
by (*simp add*: *mask-eq-exp-minus-1*)

lemma [*code*]:
‹*set-bit n k* = *k OR push-bit n 1*› **for** *k* :: *integer*
by (*fact set-bit-def*)

lemma [*code*]:
‹*unset-bit n k* = *k AND NOT* (*push-bit n 1*)› **for** *k* :: *integer*
by (*fact unset-bit-def*)

lemma [*code*]:
‹*flip-bit n k* = *k XOR push-bit n 1*› **for** *k* :: *integer*
by (*fact flip-bit-def*)

lemma [*code*]:
‹*take-bit n k* = *k AND mask n*› **for** *k* :: *integer*
by (*fact take-bit-eq-mask*)

end

definition *bit-cut-integer* :: *integer* ⇒ *integer* × *bool*
where *bit-cut-integer k* = (*k div 2*, *odd k*)

lemma *bit-cut-integer-code* [*code*]:
bit-cut-integer k = (*if k* = *0 then* (*0*, *False*)
else let (*r*, *s*) = *Code-Numeral.divmod-abs k 2*
in (*if k* > *0 then r else* − *r* − *s*, *s* = *1*))
proof −
have *bit-cut-integer k* = (*let* (*r*, *s*) = *divmod-integer k 2 in* (*r*, *s* = *1*))
by (*simp add*: *divmod-integer-def bit-cut-integer-def odd-iff-mod-2-eq-one*)
then show *?thesis*
by (*simp add*: *divmod-integer-code*) (*auto simp add*: *split-def*)
qed

lemma *equal-integer-code* [*code*]:
HOL.equal 0 (*0*::*integer*) ⟷ *True*
HOL.equal 0 (*Pos l*) ⟷ *False*
HOL.equal 0 (*Neg l*) ⟷ *False*
HOL.equal (*Pos k*) *0* ⟷ *False*
HOL.equal (*Pos k*) (*Pos l*) ⟷ *HOL.equal k l*
HOL.equal (*Pos k*) (*Neg l*) ⟷ *False*

HOL.equal (*Neg k*) *0* ⟷ *False*
HOL.equal (*Neg k*) (*Pos l*) ⟷ *False*
HOL.equal (*Neg k*) (*Neg l*) ⟷ *HOL.equal k l*
by (*simp-all add*: *equal*)

lemma *equal-integer-refl* [*code nbe*]:
HOL.equal (*k*::*integer*) *k* ⟷ *True*
by (*fact equal-refl*)

lemma *less-eq-integer-code* [*code*]:
0 ≤ (*0*::*integer*) ⟷ *True*
0 ≤ *Pos l* ⟷ *True*
0 ≤ *Neg l* ⟷ *False*
Pos k ≤ *0* ⟷ *False*
Pos k ≤ *Pos l* ⟷ *k* ≤ *l*
Pos k ≤ *Neg l* ⟷ *False*
Neg k ≤ *0* ⟷ *True*
Neg k ≤ *Pos l* ⟷ *True*
Neg k ≤ *Neg l* ⟷ *l* ≤ *k*
by *simp-all*

lemma *less-integer-code* [*code*]:
0 < (*0*::*integer*) ⟷ *False*
0 < *Pos l* ⟷ *True*
0 < *Neg l* ⟷ *False*
Pos k < *0* ⟷ *False*
Pos k < *Pos l* ⟷ *k* < *l*
Pos k < *Neg l* ⟷ *False*
Neg k < *0* ⟷ *True*
Neg k < *Pos l* ⟷ *True*
Neg k < *Neg l* ⟷ *l* < *k*
by *simp-all*

lift-definition *num-of-integer* :: *integer* ⇒ *num*
is *num-of-nat* ∘ *nat*
.

lemma *num-of-integer-code* [*code*]:
num-of-integer k = (*if k* ≤ *1 then Num.One*
else let
(*l*, *j*) = *divmod-integer k 2*;
l' = *num-of-integer l*;
l'' = *l'* + *l'*
in if j = *0 then l''* *else l''* + *Num.One*)
proof −
{
assume *int-of-integer k mod 2* = *1*
then have *nat* (*int-of-integer k mod 2*) = *nat 1* **by** *simp*
moreover assume ∗: *1* < *int-of-integer k*

```
  ultimately have ∗∗: nat (int-of-integer k) mod 2 = 1 by (simp add: nat-mod-distrib)
   have num-of-nat (nat (int-of-integer k)) =
     num-of-nat (2 ∗ (nat (int-of-integer k) div 2) + nat (int-of-integer k) mod 2)
     by simp
   then have num-of-nat (nat (int-of-integer k)) =
     num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + nat
(int-of-integer k) mod 2)
     by (simp add: mult-2)
   with ∗∗ have num-of-nat (nat (int-of-integer k)) =
     num-of-nat (nat (int-of-integer k) div 2 + nat (int-of-integer k) div 2 + 1)
     by simp
 }
 note aux = this
 show ?thesis
   by (auto simp add: num-of-integer-def nat-of-integer-def Let-def case-prod-beta
     not-le integer-eq-iff less-eq-integer-def
     nat-mult-distrib nat-div-distrib num-of-nat-One num-of-nat-plus-distrib
      mult-2 [where 'a=nat] aux add-One)
qed

lemma nat-of-integer-code [code]:
 nat-of-integer k = (if k ≤ 0 then 0
    else let
      (l, j) = divmod-integer k 2;
      l' = nat-of-integer l;
      l'' = l' + l'
    in if j = 0 then l'' else l'' + 1)
proof −
 obtain j where k: k = integer-of-int j
 proof
   show k = integer-of-int (int-of-integer k) by simp
 qed
 have ∗: nat j mod 2 = nat-of-integer (of-int j mod 2) if j ≥ 0
   using that by transfer (simp add: nat-mod-distrib)
 from k show ?thesis
    by (auto simp add: split-def Let-def nat-of-integer-def nat-div-distrib mult-2
[symmetric]
     minus-mod-eq-mult-div [symmetric] ∗)
qed

lemma int-of-integer-code-nbe [code nbe]:
 int-of-integer 0 = 0
 int-of-integer (Pos n) = Int.Pos n
 int-of-integer (Neg n) = Int.Neg n
 by simp-all

lemma int-of-integer-code [code]:
 ‹int-of-integer k = (
   if k = 0 then 0
```

```
    else if k = - 1 then - 1
    else
      let
        (l, j) = divmod-integer k 2;
        l' = 2 * int-of-integer l
      in if j = 0 then l' else l' + 1)›
  by (auto simp add: case-prod-unfold Let-def integer-eq-iff simp flip: minus-mod-eq-mult-div)

lemma integer-of-int-code-nbe [code nbe]:
  integer-of-int 0 = 0
  integer-of-int (Int.Pos n) = Pos n
  integer-of-int (Int.Neg n) = Neg n
  by simp-all

lemma integer-of-int-code [code]:
  ‹integer-of-int k = (
    if k = 0 then 0
    else if k = - 1 then - 1
    else
      let
        l = 2 * integer-of-int (k div 2);
        j = k mod 2
      in if j = 0 then l else l + 1)›
  by (simp add: integer-eq-iff Let-def flip: minus-mod-eq-mult-div)

hide-const (open) Pos Neg sub dup divmod-abs

context
begin

qualified definition push-bit :: ‹integer ⇒ integer ⇒ integer›
  where ‹push-bit i k = Bit-Operations.push-bit (nat-of-integer |i|) k›

qualified lemma push-bit-code [code]:
  ‹push-bit i k = k * 2 ^ nat-of-integer |i|›
  by (simp add: push-bit-def push-bit-eq-mult)

lemma push-bit-integer-code [code]:
  ‹Bit-Operations.push-bit n k = push-bit (of-nat n) k›
  by (simp add: push-bit-def)

qualified definition drop-bit :: ‹integer ⇒ integer ⇒ integer›
  where ‹drop-bit i k = Bit-Operations.drop-bit (nat-of-integer |i|) k›

qualified lemma drop-bit-code [code]:
  ‹drop-bit i k = k div 2 ^ nat-of-integer |i|›
  by (simp add: drop-bit-def drop-bit-eq-div)

lemma drop-bit-integer-code [code]:
```

‹*Bit-Operations.drop-bit n k = drop-bit* (*of-nat n*) *k*›
by (*simp add*: *drop-bit-def*)

end

69.3 Serializer setup for target language integers

code-printing
type-constructor *integer* ⇀
(*SML*) *IntInf.int*
and (*OCaml*) *Z.t*
and (*Haskell*) *Integer*
and (*Scala*) *BigInt*
and (*Eval*) *int*
| **class-instance** *integer* :: *equal* ⇀
(*Haskell*) –

code-reserved
(*Eval*) *int Integer*

code-printing
constant *0*::*integer* ⇀
(*SML*) !(*0* / :/ *IntInf.int*)
and (*OCaml*) *Z.zero*
and (*Haskell*) !(*0* / ::/ *Integer*)
and (*Scala*) *BigInt*(*0*)

setup ‹
fold (*fn target* =>
Numeral.add-code ***const-name*** ‹*Code-Numeral.Pos*› *I Code-Printer.literal-numeral target*
#> *Numeral.add-code* ***const-name*** ‹*Code-Numeral.Neg*› (~) *Code-Printer.literal-numeral target*)
[*SML*, *OCaml*, *Haskell*, *Scala*]
›

code-printing
constant *plus* :: *integer* ⇒ - ⇒ - ⇀
(*SML*) *IntInf.+* ((-), (-))
and (*OCaml*) *Z.add*
and (*Haskell*) **infixl** *6* +
and (*Scala*) **infixl** *7* +
and (*Eval*) **infixl** *8* +
| **constant** *uminus* :: *integer* ⇒ - ⇀
(*SML*) *IntInf.~*
and (*OCaml*) *Z.neg*
and (*Haskell*) *negate*
and (*Scala*) !(– -)
and (*Eval*) ~/ -

```
| constant minus :: integer ⇒ - ⇀
    (SML) IntInf.~ ((-), (-))
    and (OCaml) Z.sub
    and (Haskell) infixl 6 -
    and (Scala) infixl 7 -
    and (Eval) infixl 8 -
| constant Code-Numeral.dup ⇀
    (SML) IntInf.*/ (2,/ (-))
    and (OCaml) Z.shift'-left/ -/ 1
    and (Haskell) !(2 * -)
    and (Scala) !(2 * -)
    and (Eval) !(2 * -)
| constant Code-Numeral.sub ⇀
    (SML) !(raise/ Fail/ sub)
    and (OCaml) failwith/ sub
    and (Haskell) error/ sub
    and (Scala) !sys.error(sub)
| constant times :: integer ⇒ - ⇒ - ⇀
    (SML) IntInf.* ((-), (-))
    and (OCaml) Z.mul
    and (Haskell) infixl 7 *
    and (Scala) infixl 8 *
    and (Eval) infixl 9 *
| constant Code-Numeral.divmod-abs ⇀
    (SML) IntInf.divMod/ (IntInf.abs -,/ IntInf.abs -)
  and (OCaml) !(fun k l ->/ if Z.equal Z.zero l then/ (Z.zero, l) else/ Z.div'-rem/
(Z.abs k)/ (Z.abs l))
    and (Haskell) divMod/ (abs -)/ (abs -)
    and (Scala) !((k: BigInt) => (l: BigInt) =>/ l == 0 match { case true =>
(BigInt(0), k) case false => (k.abs '/% l.abs) })
    and (Eval) Integer.div'-mod/ (abs -)/ (abs -)
| constant HOL.equal :: integer ⇒ - ⇒ bool ⇀
    (SML) !((- : IntInf.int) = -)
    and (OCaml) Z.equal
    and (Haskell) infix 4 ==
    and (Scala) infixl 5 ==
    and (Eval) infixl 6 =
| constant less-eq :: integer ⇒ - ⇒ bool ⇀
    (SML) IntInf.<= ((-), (-))
    and (OCaml) Z.leq
    and (Haskell) infix 4 <=
    and (Scala) infixl 4 <=
    and (Eval) infixl 6 <=
| constant less :: integer ⇒ - ⇒ bool ⇀
    (SML) IntInf.< ((-), (-))
    and (OCaml) Z.lt
    and (Haskell) infix 4 <
    and (Scala) infixl 4 <
    and (Eval) infixl 6 <
```

```
| constant abs :: integer ⇒ - ⇀
    (SML) IntInf.abs
    and (OCaml) Z.abs
    and (Haskell) Prelude.abs
    and (Scala) -.abs
    and (Eval) abs
| constant Bit-Operations.and :: integer ⇒ integer ⇒ integer ⇀
    (SML) IntInf.andb ((-),/ (-))
    and (OCaml) Z.logand
    and (Haskell) infixl 7 .&.
    and (Scala) infixl 3 &
| constant Bit-Operations.or :: integer ⇒ integer ⇒ integer ⇀
    (SML) IntInf.orb ((-),/ (-))
    and (OCaml) Z.logor
    and (Haskell) infixl 5 .|.
    and (Scala) infixl 1 |
| constant Bit-Operations.xor :: integer ⇒ integer ⇒ integer ⇀
    (SML) IntInf.xorb ((-),/ (-))
    and (OCaml) Z.logxor
    and (Haskell) infixl 6 .^.
    and (Scala) infixl 2 ^
| constant Bit-Operations.not :: integer ⇒ integer ⇀
    (SML) IntInf.notb
    and (OCaml) Z.lognot
    and (Haskell) Data.Bits.complement
    and (Scala) -.unary'-~

code-reserved
  (Eval) abs

code-printing code-module Bit-Shifts ⇀
    (SML) ‹
structure Bit-Shifts : sig
  type int = IntInf.int
  val push : int -> int -> int
  val drop : int -> int -> int
end = struct

open IntInf;

fun fold - [] y = y
  | fold f (x :: xs) y = fold f xs (f x y);

fun replicate n x = (if n <= 0 then [] else x :: replicate (n - 1) x);

val max-index = pow (fromInt 2, Word.wordSize) - fromInt 1; (*largest possible
word*)

val word-of-int = Word.fromLargeInt o toLarge;
```

```
val word-max-index = word-of-int max-index;

fun words-of-int k = case divMod (k, max-index)
  of (b, s) => word-of-int s :: replicate b word-max-index;

fun push' i k = << (k, i);

fun drop' i k = ~>> (k, i);

(* The implementations are formally total, though indices >~ max-index will pro-
duce heavy computation load *)

fun push i = fold push' (words-of-int (abs i));

fun drop i = fold drop' (words-of-int (abs i));

end;› for constant Code-Numeral.push-bit Code-Numeral.drop-bit
    and (OCaml) ‹
module Bit-Shifts : sig
  val push : Z.t -> Z.t -> Z.t
  val drop : Z.t -> Z.t -> Z.t
end = struct

let rec fold f xs y = match xs with
  [] -> y
  | (x :: xs) -> fold f xs (f x y);;

let rec replicate n x = (if Z.leq n Z.zero then [] else x :: replicate (Z.pred n) x);;

let max-index = Z.of-int max-int;;

let splitIndex i = let (b, s) = Z.div-rem i max-index
  in Z.to-int s :: replicate b max-int;;

let push' i k = Z.shift-left k i;;

let drop' i k = Z.shift-right k i;;

(* The implementations are formally total, though indices >~ max-index will pro-
duce heavy computation load *)

let push i = fold push' (splitIndex (Z.abs i));;

let drop i = fold drop' (splitIndex (Z.abs i));;

end;;
› for constant Code-Numeral.push-bit Code-Numeral.drop-bit
    and (Haskell) ‹
```

```
module Bit-Shifts (push, drop, push', drop') where

import Prelude (Int, Integer, toInteger, fromInteger, maxBound, divMod, (-), (<=),
abs, flip)
import GHC.Bits (Bits)
import Data.Bits (shiftL, shiftR)

fold :: (a -> b -> b) -> [a] -> b -> b
fold - [] y = y
fold f (x : xs) y = fold f xs (f x y)

replicate :: Integer -> a -> [a]
replicate k x = if k <= 0 then [] else x : replicate (k - 1) x

maxIndex :: Integer
maxIndex = toInteger (maxBound :: Int)

splitIndex :: Integer -> [Int]
splitIndex i = fromInteger s : replicate (fromInteger b) maxBound
  where (b, s) = i `divMod` maxIndex

{- The implementations are formally total, though indices >~ maxIndex will pro-
duce heavy computation load -}

push :: Integer -> Integer -> Integer
push i = fold (flip shiftL) (splitIndex (abs i))

drop :: Integer -> Integer -> Integer
drop i = fold (flip shiftR) (splitIndex (abs i))

push' :: Int -> Int -> Int
push' i = flip shiftL (abs i)

drop' :: Int -> Int -> Int
drop' i = flip shiftR (abs i)
› for constant Code-Numeral.push-bit Code-Numeral.drop-bit
    and (Scala) ‹
object Bit-Shifts {

private val maxIndex : BigInt = BigInt(Int.MaxValue);

private def replicate[A](i : BigInt, x : A) : List[A] =
  i <= 0 match {
    case true => Nil
    case false => x :: replicate[A](i - 1, x)
  }

private def splitIndex(i : BigInt) : List[Int] = {
  val (b, s) = i /% maxIndex
```

return s.intValue :: replicate(b, Int.MaxValue)
}

/ The implementations are formally total, though indices >~ maxIndex will produce heavy computation load */*

def push(i: BigInt, k: BigInt) : BigInt =
splitIndex(i).foldLeft(k) { (l, j) => l << j }

def drop(i: BigInt, k: BigInt) : BigInt =
splitIndex(i).foldLeft(k) { (l, j) => l >> j }

}
› **for constant** *Code-Numeral.push-bit Code-Numeral.drop-bit*

code-reserved
(*SML*) *Bit-Shifts*
and (*Haskell*) *Bit-Shifts*
and (*Scala*) *Bit-Shifts*

code-printing
constant *Code-Numeral.push-bit* ⇀
(*SML*) *Bit'-Shifts.push*
and (*OCaml*) *Bit'-Shifts.push*
and (*Haskell*) *Bit'-Shifts.push*
and (*Scala*) *Bit'-Shifts.push*
| **constant** *Code-Numeral.drop-bit* ⇀
(*SML*) *Bit'-Shifts.drop*
and (*OCaml*) *Bit'-Shifts.drop*
and (*Haskell*) *Bit'-Shifts.drop*
and (*Scala*) *Bit'-Shifts.drop*

code-identifier
code-module *Code-Numeral* ⇀ (*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

69.4 Type of target language naturals

typedef *natural* = *UNIV* :: *nat set*
morphisms *nat-of-natural natural-of-nat* **..**

setup-lifting *type-definition-natural*

lemma *natural-eq-iff* [*termination-simp*]:
m = n ⟷ *nat-of-natural m = nat-of-natural n*
by *transfer rule*

lemma *natural-eqI*:
nat-of-natural m = nat-of-natural n ⟹ *m = n*

using *natural-eq-iff* [*of m n*] **by** *simp*

lemma *nat-of-natural-of-nat-inverse* [*simp*]:
nat-of-natural (*natural-of-nat n*) = *n*
by *transfer rule*

lemma *natural-of-nat-of-natural-inverse* [*simp*]:
natural-of-nat (*nat-of-natural n*) = *n*
by *transfer rule*

instantiation *natural* :: {*comm-monoid-diff*, *semiring-1*}
begin

lift-definition *zero-natural* :: *natural*
is *0* :: *nat*
.

declare *zero-natural.rep-eq* [*simp*]

lift-definition *one-natural* :: *natural*
is *1* :: *nat*
.

declare *one-natural.rep-eq* [*simp*]

lift-definition *plus-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *plus* :: *nat* ⇒ *nat* ⇒ *nat*
.

declare *plus-natural.rep-eq* [*simp*]

lift-definition *minus-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *minus* :: *nat* ⇒ *nat* ⇒ *nat*
.

declare *minus-natural.rep-eq* [*simp*]

lift-definition *times-natural* :: *natural* ⇒ *natural* ⇒ *natural*
is *times* :: *nat* ⇒ *nat* ⇒ *nat*
.

declare *times-natural.rep-eq* [*simp*]

instance proof
qed (*transfer*, *simp add*: *algebra-simps*)+

end

instance *natural* :: *Rings.dvd* **..**

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
‹(*pcr-natural* ===> *pcr-natural* ===> (⟷)) (*dvd*) (*dvd*)›
by (*unfold dvd-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹((⟷) ===> *pcr-natural*) *of-bool of-bool*›
by (*unfold of-bool-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-natural*) (λ*n*. *n*) *of-nat*›
proof −
have *rel-fun HOL.eq pcr-natural* (*of-nat* :: *nat* ⇒ *nat*) (*of-nat* :: *nat* ⇒ *natural*)
by (*unfold of-nat-def*) *transfer-prover*
then show *?thesis* **by** (*simp add*: *id-def*)
qed

lemma [*transfer-rule*]:
‹((=) ===> *pcr-natural*) *numeral numeral*›
proof −
have ‹((=) ===> *pcr-natural*) *numeral* (λ*n*. *of-nat* (*numeral n*))›
by *transfer-prover*
then show *?thesis* **by** *simp*
qed

lemma [*transfer-rule*]:
‹(*pcr-natural* ===> (=) ===> *pcr-natural*) (^) (^)›
by (*unfold power-def*) *transfer-prover*

end

lemma *nat-of-natural-of-nat* [*simp*]:
nat-of-natural (*of-nat n*) = *n*
by *transfer rule*

lemma *natural-of-nat-of-nat* [*simp*, *code-abbrev*]:
natural-of-nat = *of-nat*
by *transfer rule*

lemma *of-nat-of-natural* [*simp*]:
of-nat (*nat-of-natural n*) = *n*
by *transfer rule*

lemma *nat-of-natural-numeral* [*simp*]:
nat-of-natural (*numeral k*) = *numeral k*

by *transfer rule*

instantiation *natural* :: {*linordered-semiring*, *equal*}
begin

lift-definition *less-eq-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *less-eq* :: *nat* ⇒ *nat* ⇒ *bool*
.

declare *less-eq-natural.rep-eq* [*termination-simp*]

lift-definition *less-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *less* :: *nat* ⇒ *nat* ⇒ *bool*
.

declare *less-natural.rep-eq* [*termination-simp*]

lift-definition *equal-natural* :: *natural* ⇒ *natural* ⇒ *bool*
is *HOL.equal* :: *nat* ⇒ *nat* ⇒ *bool*
.

instance proof
qed (*transfer*, *simp add*: *algebra-simps equal less-le-not-le* [*symmetric*] *linear*)+

end

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
‹(*pcr-natural* ===> *pcr-natural* ===> *pcr-natural*) *min min*›
by (*unfold min-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹(*pcr-natural* ===> *pcr-natural* ===> *pcr-natural*) *max max*›
by (*unfold max-def*) *transfer-prover*

end

lemma *nat-of-natural-min* [*simp*]:
nat-of-natural (*min k l*) = *min* (*nat-of-natural k*) (*nat-of-natural l*)
by *transfer rule*

lemma *nat-of-natural-max* [*simp*]:
nat-of-natural (*max k l*) = *max* (*nat-of-natural k*) (*nat-of-natural l*)
by *transfer rule*

instantiation *natural* :: *unique-euclidean-semiring*

begin

lift-definition *divide-natural* :: *natural* ⇒ *natural* ⇒ *natural*
  is *divide* :: *nat* ⇒ *nat* ⇒ *nat*
  .

declare *divide-natural.rep-eq* [*simp*]

lift-definition *modulo-natural* :: *natural* ⇒ *natural* ⇒ *natural*
  is *modulo* :: *nat* ⇒ *nat* ⇒ *nat*
  .

declare *modulo-natural.rep-eq* [*simp*]

lift-definition *euclidean-size-natural* :: *natural* ⇒ *nat*
  is *euclidean-size* :: *nat* ⇒ *nat*
  .

declare *euclidean-size-natural.rep-eq* [*simp*]

lift-definition *division-segment-natural* :: *natural* ⇒ *natural*
  is *division-segment* :: *nat* ⇒ *nat*
  .

declare *division-segment-natural.rep-eq* [*simp*]

instance
  by (*standard*; *transfer*)
    (*auto simp add*: *algebra-simps unit-factor-nat-def gr0-conv-Suc*)

end

lemma [*code*]:
  euclidean-size = *nat-of-natural*
  by (*simp add*: *fun-eq-iff*)

lemma [*code*]:
  division-segment (*n*::*natural*) = *1*
  by (*simp add*: *natural-eq-iff*)

instance *natural* :: *discrete-linordered-semidom*
  by (*standard*; *transfer*) (*simp-all add*: *Suc-le-eq*)

instance *natural* :: *linordered-euclidean-semiring*
  by (*standard*; *transfer*) *simp-all*

instantiation *natural* :: *semiring-bit-operations*
begin

lift-definition *bit-natural* :: ‹*natural* ⇒ *nat* ⇒ *bool*›
is *bit* **.**

lift-definition *and-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
is ‹*and*› **.**

lift-definition *or-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
is *or* **.**

lift-definition *xor-natural* :: ‹*natural* ⇒ *natural* ⇒ *natural*›
is *xor* **.**

lift-definition *mask-natural* :: ‹*nat* ⇒ *natural*›
is *mask* **.**

lift-definition *set-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *set-bit* **.**

lift-definition *unset-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *unset-bit* **.**

lift-definition *flip-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *flip-bit* **.**

lift-definition *push-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *push-bit* **.**

lift-definition *drop-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *drop-bit* **.**

lift-definition *take-bit-natural* :: ‹*nat* ⇒ *natural* ⇒ *natural*›
is *take-bit* **.**

instance by (*standard*; *transfer*)
(*fact bit-induct div-by-0 div-by-1 div-0 even-half-succ-eq*
half-div-exp-eq even-double-div-exp-iff bits-mod-div-trivial
bit-iff-odd push-bit-eq-mult drop-bit-eq-div take-bit-eq-mod
and-rec or-rec xor-rec mask-eq-exp-minus-1
set-bit-eq-or unset-bit-eq-or-xor flip-bit-eq-xor not-eq-complement)+

end

instance *natural* :: *linordered-euclidean-semiring-bit-operations* **..**

lift-definition *natural-of-integer* :: *integer* ⇒ *natural*
is *nat* :: *int* ⇒ *nat*
.

lift-definition *integer-of-natural* :: *natural* ⇒ *integer*

is *of-nat* :: *nat* ⇒ *int*
.

lemma *natural-of-integer-of-natural* [*simp*]:
natural-of-integer (*integer-of-natural n*) = *n*
by *transfer simp*

lemma *integer-of-natural-of-integer* [*simp*]:
integer-of-natural (*natural-of-integer k*) = *max 0 k*
by *transfer auto*

lemma *int-of-integer-of-natural* [*simp*]:
int-of-integer (*integer-of-natural n*) = *of-nat* (*nat-of-natural n*)
by *transfer rule*

lemma *integer-of-natural-of-nat* [*simp*]:
integer-of-natural (*of-nat n*) = *of-nat n*
by *transfer rule*

lemma [*measure-function*]:
is-measure nat-of-natural
by (*rule is-measure-trivial*)

69.5 Inductive representation of target language naturals

lift-definition *Suc* :: *natural* ⇒ *natural*
is *Nat.Suc*
.

declare *Suc.rep-eq* [*simp*]

old-rep-datatype *0*::*natural Suc*
by (*transfer*, *fact nat.induct nat.inject nat.distinct*)+

lemma *natural-cases* [*case-names nat*, *cases type*: *natural*]:
fixes *m* :: *natural*
assumes ⋀*n*. *m* = *of-nat n* ⟹ *P*
shows *P*
using *assms* **by** *transfer blast*

instantiation *natural* :: *size*
begin

definition *size-nat* **where** [*simp*, *code*]: *size-nat* = *nat-of-natural*

instance ..

end

lemma *natural-decr* [*termination-simp*]:
n ≠ *0* ⟹ *nat-of-natural n* − *Nat.Suc 0* < *nat-of-natural n*
by *transfer simp*

lemma *natural-zero-minus-one*: (*0*::*natural*) − *1* = *0*
by (*rule zero-diff*)

lemma *Suc-natural-minus-one*: *Suc n* − *1* = *n*
by *transfer simp*

hide-const (**open**) *Suc*

69.6 Code refinement for target language naturals

lift-definition *Nat* :: *integer* ⇒ *natural*
is *nat*
.

lemma [*code-post*]:
Nat 0 = *0*
Nat 1 = *1*
Nat (*numeral k*) = *numeral k*
by (*transfer*, *simp*)+

lemma [*code abstype*]:
Nat (*integer-of-natural n*) = *n*
by *transfer simp*

lemma [*code*]:
natural-of-nat n = *natural-of-integer* (*integer-of-nat n*)
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural (*natural-of-integer k*) = *max 0 k*
by *simp*

lemma [*code*]:
‹*integer-of-natural* (*mask n*) = *mask n*›
by *transfer* (*simp add*: *mask-eq-exp-minus-1*)

lemma [*code-abbrev*]:
natural-of-integer (*Code-Numeral.Pos k*) = *numeral k*
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural 0 = *0*
by *transfer simp*

lemma [*code abstract*]:

integer-of-natural 1 = 1
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural (*Code-Numeral.Suc n*) = *integer-of-natural n + 1*
by *transfer simp*

lemma [*code*]:
nat-of-natural = nat-of-integer ∘ integer-of-natural
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code*, *code-unfold*]:
case-natural f g n = (*if n = 0 then f else g* (*n − 1*))
by (*cases n rule*: *natural.exhaust*) (*simp-all*, *simp add*: *Suc-def*)

declare [[*code drop*: *rec-natural*]]

lemma [*code abstract*]:
integer-of-natural (*m + n*) = *integer-of-natural m + integer-of-natural n*
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural (*m − n*) = *max 0* (*integer-of-natural m − integer-of-natural n*)
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural (*m ∗ n*) = *integer-of-natural m ∗ integer-of-natural n*
by *transfer simp*

lemma [*code abstract*]:
integer-of-natural (*m div n*) = *integer-of-natural m div integer-of-natural n*
by *transfer* (*simp add*: *zdiv-int*)

lemma [*code abstract*]:
integer-of-natural (*m mod n*) = *integer-of-natural m mod integer-of-natural n*
by *transfer* (*simp add*: *zmod-int*)

lemma [*code nbe*]: *HOL.equal n* (*n*::*natural*) ⟷ *True*
by (*rule equal-class.equal-refl*)

lemma [*code*]:
HOL.equal m n ⟷ *HOL.equal* (*integer-of-natural m*) (*integer-of-natural n*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]: *m ≤ n* ⟷ *integer-of-natural m ≤ integer-of-natural n*
by *transfer simp*

lemma [*code*]: *m < n* ⟷ *integer-of-natural m < integer-of-natural n*

by *transfer simp*

context
includes *bit-operations-syntax*
begin

lemma [*code*]:
‹*bit m n* ⟷ *bit* (*integer-of-natural m*) *n*›
by *transfer* (*simp add*: *bit-simps*)

lemma [*code abstract*]:
‹*integer-of-natural* (*m AND n*) = *integer-of-natural m AND integer-of-natural n*›
by *transfer* (*simp add*: *of-nat-and-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*m OR n*) = *integer-of-natural m OR integer-of-natural n*›
by *transfer* (*simp add*: *of-nat-or-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*m XOR n*) = *integer-of-natural m XOR integer-of-natural n*›
by *transfer* (*simp add*: *of-nat-xor-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*mask n*) = *mask n*›
by *transfer* (*simp add*: *of-nat-mask-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*set-bit n m*) = *set-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-set-bit-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*unset-bit n m*) = *unset-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-unset-bit-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*flip-bit n m*) = *flip-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-flip-bit-eq*)

lemma [*code abstract*]:
‹*integer-of-natural* (*push-bit n m*) = *push-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-push-bit*)

lemma [*code abstract*]:
‹*integer-of-natural* (*drop-bit n m*) = *drop-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-drop-bit*)

lemma [*code abstract*]:
‹*integer-of-natural* (*take-bit n m*) = *take-bit n* (*integer-of-natural m*)›
by *transfer* (*simp add*: *of-nat-take-bit*)

end

hide-const (**open**) *Nat*

code-reflect *Code-Numeral*
  datatypes *natural*
  functions *Code-Numeral.Suc 0* :: *natural 1* :: *natural*
    plus :: *natural* ⇒ *natural* ⇒ *natural* *minus* :: *natural* ⇒ *natural* ⇒ *natural*
    times :: *natural* ⇒ *natural* ⇒ *natural* *divide* :: *natural* ⇒ *natural* ⇒ *natural*
    modulo :: *natural* ⇒ *natural* ⇒ *natural*
    integer-of-natural natural-of-integer

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

lifting-update *natural.lifting*
lifting-forget *natural.lifting*

end

70 A HOL random engine

theory *Random*
imports *List Groups-List Code-Numeral*
begin

70.1 Auxiliary functions

fun *log* :: *natural* ⇒ *natural* ⇒ *natural* **where**
  log b i = (*if b* ≤ *1* ∨ *i* < *b then 1 else 1* + *log b* (*i div b*))

definition *inc-shift* :: *natural* ⇒ *natural* ⇒ *natural* **where**
  inc-shift v k = (*if v* = *k then 1 else k* + *1*)

definition *minus-shift* :: *natural* ⇒ *natural* ⇒ *natural* ⇒ *natural* **where**
  minus-shift r k l = (*if k* < *l then r* + *k* − *l else k* − *l*)

70.2 Random seeds

type-synonym *seed* = *natural* × *natural*

primrec *next* :: *seed* ⇒ *natural* × *seed* **where**
  next (*v*, *w*) = (*let*
    k = *v div 53668*;
    v' = *minus-shift 2147483563* ((*v mod 53668*) ∗ *40014*) (*k* ∗ *12211*);
    l = *w div 52774*;
    w' = *minus-shift 2147483399* ((*w mod 52774*) ∗ *40692*) (*l* ∗ *3791*);
    z = *minus-shift 2147483562 v'* (*w'* + *1*) + *1*

```
  in (z, (v', w')))

definition split-seed :: seed ⇒ seed × seed where
  split-seed s = (let
    (v, w) = s;
    (v', w') = snd (next s);
    v'' = inc-shift 2147483562 v;
    w'' = inc-shift 2147483398 w
   in ((v'', w'), (v', w'')))
```

70.3 Base selectors

```
context
  includes state-combinator-syntax
begin

fun iterate :: natural ⇒ ('b ⇒ 'a ⇒ 'b × 'a) ⇒ 'b ⇒ 'a ⇒ 'b × 'a where
  iterate k f x = (if k = 0 then Pair x else f x ∘→ iterate (k − 1) f)

definition range :: natural ⇒ seed ⇒ natural × seed where
  range k = iterate (log 2147483561 k)
      (λl. next ∘→ (λv. Pair (v + l * 2147483561))) 1
    ∘→ (λv. Pair (v mod k))

lemma range:
  k > 0 ⟹ fst (range k s) < k
  by (simp add: range-def split-def less-natural-def del: log.simps iterate.simps)

definition select :: 'a list ⇒ seed ⇒ 'a × seed where
  select xs = range (natural-of-nat (length xs))
    ∘→ (λk. Pair (nth xs (nat-of-natural k)))

lemma select:
  assumes xs ≠ []
  shows fst (select xs s) ∈ set xs
proof −
  from assms have natural-of-nat (length xs) > 0 by (simp add: less-natural-def)
  with range have
    fst (range (natural-of-nat (length xs)) s) < natural-of-nat (length xs) by best
  then have
    nat-of-natural (fst (range (natural-of-nat (length xs)) s)) < length xs by (simp
add: less-natural-def)
  then show ?thesis
    by (simp add: split-beta select-def)
qed

primrec pick :: (natural × 'a) list ⇒ natural ⇒ 'a where
  pick (x # xs) i = (if i < fst x then snd x else pick xs (i − fst x))
```

lemma *pick-member*:
i < *sum-list* (*map fst xs*) ⟹ *pick xs i* ∈ *set* (*map snd xs*)
by (*induct xs arbitrary*: *i*) (*simp-all add*: *less-natural-def*)

lemma *pick-drop-zero*:
pick (*filter* (λ(*k*, -). *k* > *0*) *xs*) = *pick xs*
apply (*induct xs*)
apply (*auto simp*: *fun-eq-iff less-natural.rep-eq split*: *prod.split*)
by (*metis diff-zero of-nat-0 of-nat-of-natural*)

lemma *pick-same*:
l < *length xs* ⟹ *Random.pick* (*map* (*Pair 1*) *xs*) (*natural-of-nat l*) = *nth xs l*
proof (*induct xs arbitrary*: *l*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **then show** *?case* **by** (*cases l*) (*simp-all add*: *less-natural-def*)
qed

definition *select-weight* :: (*natural* × *'a*) *list* ⇒ *seed* ⇒ *'a* × *seed* **where**
select-weight xs = *range* (*sum-list* (*map fst xs*))
∘→ (λ*k*. *Pair* (*pick xs k*))

lemma *select-weight-member*:
assumes *0* < *sum-list* (*map fst xs*)
shows *fst* (*select-weight xs s*) ∈ *set* (*map snd xs*)
proof −
from *range assms*
have *fst* (*range* (*sum-list* (*map fst xs*)) *s*) < *sum-list* (*map fst xs*) **.**
with *pick-member*
have *pick xs* (*fst* (*range* (*sum-list* (*map fst xs*)) *s*)) ∈ *set* (*map snd xs*) **.**
then show *?thesis* **by** (*simp add*: *select-weight-def scomp-def split-def*)
qed

lemma *select-weight-cons-zero*:
select-weight ((*0*, *x*) # *xs*) = *select-weight xs*
by (*simp add*: *select-weight-def less-natural-def*)

lemma *select-weight-drop-zero*:
select-weight (*filter* (λ(*k*, -). *k* > *0*) *xs*) = *select-weight xs*
proof −
have *sum-list* (*map fst* [(*k*, -)←*xs* . *0* < *k*]) = *sum-list* (*map fst xs*)
by (*induct xs*) (*auto simp add*: *less-natural-def natural-eq-iff*)
then show *?thesis* **by** (*simp only*: *select-weight-def pick-drop-zero*)
qed

lemma *select-weight-select*:
assumes *xs* ≠ []
shows *select-weight* (*map* (*Pair 1*) *xs*) = *select xs*
proof −

have *less*: $\bigwedge s$. *fst* (*range* (*natural-of-nat* (*length xs*)) *s*) < *natural-of-nat* (*length xs*)
using *assms* **by** (*intro range*) (*simp add*: *less-natural-def*)
moreover have *sum-list* (*map fst* (*map* (*Pair 1*) *xs*)) = *natural-of-nat* (*length xs*)
by (*induct xs*) *simp-all*
ultimately show *?thesis*
by (*auto simp add*: *select-weight-def select-def scomp-def split-def*
fun-eq-iff pick-same [*symmetric*] *less-natural-def*)
qed

end

70.4 *ML* interface

code-reflect *Random-Engine*
functions *range select select-weight*

ML ‹
```
structure Random-Engine =
struct

open Random-Engine;

type seed = Code-Numeral.natural * Code-Numeral.natural;

local

val seed = Unsynchronized.ref
  (let
    val now = Time.toMilliseconds (Time.now ());
    val (q, s1) = IntInf.divMod (now, 2147483562);
    val s2 = q mod 2147483398;
  in apply2 Code-Numeral.natural-of-integer (s1 + 1, s2 + 1) end);

in

fun next-seed () =
  let
    val (seed1, seed') = @{code split-seed} (! seed)
    val - = seed := seed'
  in
    seed1
  end

fun run f =
  let
    val (x, seed') = f (! seed);
    val - = seed := seed'
```

in x end;

end;

end;
›

hide-type (**open**) *seed*
hide-const (**open**) *inc-shift minus-shift log next split-seed*
iterate range select pick select-weight
hide-fact (**open**) *range-def*

end

71 Maps

theory *Map*
imports *List*
abbrevs (= = $\subseteq_m$
begin

type-synonym ('*a*, '*b*) *map* = '*a* ⇒ '*b option* (**infixr** ‹⇀› *0*)

abbreviation (*input*)
empty :: '*a* ⇀ '*b* **where**
empty ≡ λ*x*. *None*

definition
map-comp :: ('*b* ⇀ '*c*) ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*c*) (**infixl** ‹∘$_m$› *55*) **where**
f ∘$_m$ *g* = (λ*k*. *case g k of None* ⇒ *None* | *Some v* ⇒ *f v*)

definition
map-add :: ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*) ⇒ ('*a* ⇀ '*b*) (**infixl** ‹++› *100*) **where**
m1 ++ *m2* = (λ*x*. *case m2 x of None* ⇒ *m1 x* | *Some y* ⇒ *Some y*)

definition
restrict-map :: ('*a* ⇀ '*b*) ⇒ '*a set* ⇒ ('*a* ⇀ '*b*) (**infixl** ‹|`› *110*) **where**
m|`*A* = (λ*x*. *if x* ∈ *A then m x else None*)

notation (*latex* **output**)
restrict-map (‹-|`-› [*111*,*110*] *110*)

definition
dom :: ('*a* ⇀ '*b*) ⇒ '*a set* **where**
dom m = {*a*. *m a* ≠ *None*}

definition
ran :: ('*a* ⇀ '*b*) ⇒ '*b set* **where**
ran m = {*b*. ∃ *a*. *m a* = *Some b*}

definition
graph :: $('a \rightharpoonup 'b) \Rightarrow ('a \times 'b)\ set$ **where**
graph $m = \{(a, b) \mid a\ b.\ m\ a = Some\ b\}$

definition
map-le :: $('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow bool$ (**infix** ‹$\subseteq_m$› *50*) **where**
$(m_1 \subseteq_m m_2) \longleftrightarrow (\forall a \in dom\ m_1.\ m_1\ a = m_2\ a)$

Function update syntax $f(x := y, \ldots)$ is extended with $x \mapsto y$, which is short for $x := Some\ y$. := and $\mapsto$ can be mixed freely. The syntax $[x \mapsto y, \ldots]$ is short for *Map.empty*$(x \mapsto y, \ldots)$ but must only contain $\mapsto$, not :=, because $[x{:=}y]$ clashes with the list update syntax $xs[i{:=}x]$.

nonterminal *maplet* **and** *maplets*

open-bundle *maplet-syntax*
begin

syntax
-maplet :: ['a, 'a] ⇒ *maplet* (‹(‹*open-block notation=‹mixfix maplet›*›- /↦/ -)›)
:: *maplet* ⇒ *updbind* (‹-›)
:: *maplet* ⇒ *maplets* (‹-›)
-Maplets :: [*maplet*, *maplets*] ⇒ *maplets* (‹-,/ -›)
-Map :: *maplets* ⇒ 'a ⇀ 'b (‹(‹*indent=1 notation=‹mixfix map›*›[-])›)

syntax (*ASCII*)
-maplet :: ['a, 'a] ⇒ *maplet* (‹(‹*open-block notation=‹mixfix maplet›*›- /|->/ -)›)

syntax-consts
-maplet -Maplets -Map ⇌ *fun-upd*

translations
-Update f (*-maplet x y*) ⇌ $f(x := CONST\ Some\ y)$
-Maplets m ms ⇀ *-updbinds m ms*
-Map ms ⇀ *-Update* (*CONST empty*) *ms*

-Map (*-maplet x y*) ↽ *-Update* (λ*u. CONST None*) (*-maplet x y*)
-Map (*-updbinds m* (*-maplet x y*)) ↽ *-Update* (*-Map m*) (*-maplet x y*)

end

Updating with lists:

primrec *map-of* :: $('a \times 'b)\ list \Rightarrow 'a \rightharpoonup 'b$ **where**
map-of [] = *empty*
| *map-of* (*p* # *ps*) = (*map-of ps*)(*fst p* ↦ *snd p*)

lemma *map-of-Cons-code* [*code*]:
map-of [] *k* = *None*
map-of ((*l*, *v*) # *ps*) *k* = (*if l* = *k then Some v else map-of ps k*)
by *simp-all*

definition *map-upds* :: ('*a* ⇀ '*b*) ⇒ '*a list* ⇒ '*b list* ⇒ '*a* ⇀ '*b* **where**
map-upds m xs ys = *m* ++ *map-of* (*rev* (*zip xs ys*))

There is also the more specialized update syntax *xs* [↦] *ys* for lists *xs* and *ys*.

open-bundle *list-maplet-syntax*
begin

syntax
-maplets :: ['*a*, '*a*] ⇒ *maplet* (‹(‹*open-block notation*=‹*mixfix maplet*››- /[↦]/ -)›)

syntax (*ASCII*)
-maplets :: ['*a*, '*a*] ⇒ *maplet* (‹(‹*open-block notation*=‹*mixfix maplet*››- /[|->]/ -)›)

syntax-consts
-maplets ⇌ *map-upds*

translations
-Update m (*-maplets xs ys*) ⇌ *CONST map-upds m xs ys*

-Map (*-maplets xs ys*) ↽ *-Update* (λ*u*. *CONST None*) (*-maplets xs ys*)
-Map (*-updbinds m* (*-maplets xs ys*)) ↽ *-Update* (*-Map m*) (*-maplets xs ys*)

end

71.1 *empty*

lemma *empty-upd-none* [*simp*]: *empty*(*x* := *None*) = *empty*
by (*rule ext*) *simp*

71.2 *map-upd*

lemma *map-upd-triv*: *t k* = *Some x* ⟹ *t*(*k*↦*x*) = *t*
by (*rule ext*) *simp*

lemma *map-upd-nonempty* [*simp*]: *t*(*k*↦*x*) ≠ *empty*
proof
assume *t*(*k* ↦ *x*) = *empty*
then have (*t*(*k* ↦ *x*)) *k* = *None* **by** *simp*
then show *False* **by** *simp*
qed

lemma *map-upd-eqD1*:
assumes $m(a\mapsto x) = n(a\mapsto y)$
shows $x = y$
proof −
from *assms* **have** $(m(a\mapsto x))\ a = (n(a\mapsto y))\ a$ **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *map-upd-Some-unfold*:
$((m(a\mapsto b))\ x = Some\ y) = (x = a \wedge b = y \vee x \neq a \wedge m\ x = Some\ y)$
by *auto*

lemma *image-map-upd* [*simp*]: $x \notin A \Longrightarrow m(x \mapsto y)\ `\ A = m\ `\ A$
by *auto*

lemma *finite-range-updI*:
assumes *finite* (*range f*) **shows** *finite* (*range* $(f(a\mapsto b))$)
proof −
have *range* $(f(a\mapsto b)) \subseteq$ *insert* (*Some b*) (*range f*)
by *auto*
then show *?thesis*
by (*rule finite-subset*) (*use assms* **in** *auto*)
qed

71.3 *map-of*

lemma *map-of-eq-empty-iff* [*simp*]:
map-of xys = *empty* $\longleftrightarrow$ *xys* = []
proof
show *map-of xys* = *empty* $\Longrightarrow$ *xys* = []
by (*induction xys*) *simp-all*
qed *simp*

lemma *empty-eq-map-of-iff* [*simp*]:
empty = *map-of xys* $\longleftrightarrow$ *xys* = []
by(*subst eq-commute*) *simp*

lemma *map-of-eq-None-iff*:
(*map-of xys x* = *None*) = ($x \notin$ *fst* ` (*set xys*))
by (*induct xys*) *simp-all*

lemma *map-of-eq-Some-iff* [*simp*]:
distinct(*map fst xys*) $\Longrightarrow$ (*map-of xys x* = *Some y*) = ((*x*,*y*) $\in$ *set xys*)
proof (*induct xys*)
case (*Cons xy xys*)
then show *?case*
by (*cases xy*) (*auto simp flip*: *map-of-eq-None-iff*)
qed *auto*

lemma *Some-eq-map-of-iff* [*simp*]:
  distinct(*map fst xys*) $\Longrightarrow$ (*Some y* = *map-of xys x*) = ((*x*,*y*) $\in$ *set xys*)
by (*auto simp del*: *map-of-eq-Some-iff simp*: *map-of-eq-Some-iff* [*symmetric*])

lemma *map-of-is-SomeI* [*simp*]:
  ⟦*distinct*(*map fst xys*); (*x*,*y*) $\in$ *set xys*⟧ $\Longrightarrow$ *map-of xys x* = *Some y*
  by *simp*

lemma *map-of-zip-is-None* [*simp*]:
  length xs = *length ys* $\Longrightarrow$ (*map-of* (*zip xs ys*) *x* = *None*) = (*x* $\notin$ *set xs*)
by (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-is-Some*:
  assumes *length xs* = *length ys*
  shows *x* $\in$ *set xs* $\longleftrightarrow$ ($\exists$ *y*. *map-of* (*zip xs ys*) *x* = *Some y*)
using *assms* **by** (*induct rule*: *list-induct2*) *simp-all*

lemma *map-of-zip-upd*:
  fixes *x* :: *'a* **and** *xs* :: *'a list* **and** *ys zs* :: *'b list*
  assumes *length ys* = *length xs*
    and *length zs* = *length xs*
    and *x* $\notin$ *set xs*
    and (*map-of* (*zip xs ys*))(*x* $\mapsto$ *y*) = (*map-of* (*zip xs zs*))(*x* $\mapsto$ *z*)
  shows *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
proof
  fix *x'* :: *'a*
  show *map-of* (*zip xs ys*) *x'* = *map-of* (*zip xs zs*) *x'*
  proof (*cases x* = *x'*)
    case *True*
    from *assms True map-of-zip-is-None* [*of xs ys x'*]
      have *map-of* (*zip xs ys*) *x'* = *None* **by** *simp*
    moreover from *assms True map-of-zip-is-None* [*of xs zs x'*]
      have *map-of* (*zip xs zs*) *x'* = *None* **by** *simp*
    ultimately show *?thesis* **by** *simp*
  next
    case *False* **from** *assms*
     have ((*map-of* (*zip xs ys*))(*x* $\mapsto$ *y*)) *x'* = ((*map-of* (*zip xs zs*))(*x* $\mapsto$ *z*)) *x'* **by** *auto*
    with *False* **show** *?thesis* **by** *simp*
  qed
qed

lemma *map-of-zip-inject*:
  assumes *length ys* = *length xs*
    and *length zs* = *length xs*
    and *dist*: *distinct xs*
    and *map-of*: *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*)
  shows *ys* = *zs*
  using *assms*(*1*) *assms*(*2*)[*symmetric*]

using *dist map-of*
proof (*induct ys xs zs rule*: *list-induct3*)
case *Nil* **show** *?case* **by** *simp*
next
case (*Cons y ys x xs z zs*)
from ‹*map-of* (*zip* (*x*#*xs*) (*y*#*ys*)) = *map-of* (*zip* (*x*#*xs*) (*z*#*zs*))›
have *map-of*: (*map-of* (*zip xs ys*))(*x* ↦ *y*) = (*map-of* (*zip xs zs*))(*x* ↦ *z*) **by** *simp*
from *Cons* **have** *length ys* = *length xs* **and** *length zs* = *length xs*
and *x* ∉ *set xs* **by** *simp-all*
then have *map-of* (*zip xs ys*) = *map-of* (*zip xs zs*) **using** *map-of* **by** (*rule map-of-zip-upd*)
with *Cons.hyps* ‹*distinct* (*x* # *xs*)› **have** *ys* = *zs* **by** *simp*
moreover from *map-of* **have** *y* = *z* **by** (*rule map-upd-eqD1*)
ultimately show *?case* **by** *simp*
qed

lemma *map-of-zip-nth*:
assumes *length xs* = *length ys*
assumes *distinct xs*
assumes *i* < *length ys*
shows *map-of* (*zip xs ys*) (*xs* ! *i*) = *Some* (*ys* ! *i*)
using *assms* **proof** (*induct arbitrary*: *i rule*: *list-induct2*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs y ys*)
then show *?case*
using *less-Suc-eq-0-disj* **by** *auto*
qed

lemma *map-of-zip-map*:
map-of (*zip xs* (*map f xs*)) = (λ*x*. *if x* ∈ *set xs then Some* (*f x*) *else None*)
by (*induct xs*) (*simp-all add*: *fun-eq-iff*)

lemma *finite-range-map-of*: *finite* (*range* (*map-of xys*))
proof (*induct xys*)
case (*Cons a xys*)
then show *?case*
using *finite-range-updI* **by** *fastforce*
qed *auto*

lemma *map-of-SomeD*: *map-of xs k* = *Some y* ⟹ (*k*, *y*) ∈ *set xs*
by (*induct xs*) (*auto split*: *if-splits*)

lemma *map-of-mapk-SomeI*:
inj f ⟹ *map-of t k* = *Some x* ⟹
map-of (*map* (*case-prod* (λ*k*. *Pair* (*f k*))) *t*) (*f k*) = *Some x*
by (*induct t*) (*auto simp*: *inj-eq*)

lemma *weak-map-of-SomeI*: $(k, x) \in set\ l \Longrightarrow \exists x.\ map\text{-}of\ l\ k = Some\ x$
by (*induct l*) *auto*

lemma *map-of-filter-in*:
$map\text{-}of\ xs\ k = Some\ z \Longrightarrow P\ k\ z \Longrightarrow map\text{-}of\ (filter\ (case\text{-}prod\ P)\ xs)\ k = Some\ z$
by (*induct xs*) *auto*

lemma *map-of-map*:
$map\text{-}of\ (map\ (\lambda(k, v).\ (k, f\ v))\ xs) = map\text{-}option\ f \circ map\text{-}of\ xs$
by (*induct xs*) (*auto simp*: *fun-eq-iff*)

lemma *map-of-filter*:
$map\text{-}of\ (filter\ (\lambda x.\ P\ (fst\ x))\ xs) = map\text{-}of\ xs\ |`\ Collect\ P$
by (*induct xs*) (*simp-all add*: *fun-eq-iff restrict-map-def*)

lemma *dom-map-option*:
$dom\ (\lambda k.\ map\text{-}option\ (f\ k)\ (m\ k)) = dom\ m$
by (*simp add*: *dom-def*)

lemma *dom-map-option-comp* [*simp*]:
$dom\ (map\text{-}option\ g \circ m) = dom\ m$
using *dom-map-option* [*of* λ-. *g m*] **by** (*simp add*: *comp-def*)

71.4 *map-option* related

lemma *map-option-o-empty* [*simp*]: $map\text{-}option\ f \circ empty = empty$
by (*rule ext*) *simp*

lemma *map-option-o-map-upd* [*simp*]:
$map\text{-}option\ f \circ m(a \mapsto b) = (map\text{-}option\ f \circ m)(a \mapsto f\ b)$
by (*rule ext*) *simp*

71.5 *map-comp* related

lemma *map-comp-empty* [*simp*]:
$m \circ_m empty = empty$
$empty \circ_m m = empty$
by (*auto simp*: *map-comp-def split*: *option.splits*)

lemma *map-comp-simps* [*simp*]:
$m2\ k = None \Longrightarrow (m1 \circ_m m2)\ k = None$
$m2\ k = Some\ k' \Longrightarrow (m1 \circ_m m2)\ k = m1\ k'$
by (*auto simp*: *map-comp-def*)

lemma *map-comp-Some-iff*:
$((m1 \circ_m m2)\ k = Some\ v) = (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = Some\ v)$
by (*auto simp*: *map-comp-def split*: *option.splits*)

lemma *map-comp-None-iff*:
$((m1 \circ_m m2)\ k = None) = (m2\ k = None \vee (\exists k'.\ m2\ k = Some\ k' \wedge m1\ k' = None))$
by (*auto simp*: *map-comp-def split*: *option.splits*)

71.6 ++

lemma *map-add-empty*[*simp*]: $m ++ empty = m$
by(*simp add*: *map-add-def*)

lemma *empty-map-add*[*simp*]: $empty ++ m = m$
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-assoc*[*simp*]: $m1 ++ (m2 ++ m3) = (m1 ++ m2) ++ m3$
by (*rule ext*) (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-Some-iff*:
$((m ++ n)\ k = Some\ x) = (n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x)$
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-SomeD* [*dest*!]:
$(m ++ n)\ k = Some\ x \Longrightarrow n\ k = Some\ x \vee n\ k = None \wedge m\ k = Some\ x$
by (*rule map-add-Some-iff* [*THEN iffD1*])

lemma *map-add-find-right* [*simp*]: $n\ k = Some\ xx \Longrightarrow (m ++ n)\ k = Some\ xx$
by (*subst map-add-Some-iff*) *fast*

lemma *map-add-None* [*iff*]: $((m ++ n)\ k = None) = (n\ k = None \wedge m\ k = None)$
by (*simp add*: *map-add-def split*: *option.split*)

lemma *map-add-upd*[*simp*]: $f ++ g(x \mapsto y) = (f ++ g)(x \mapsto y)$
by (*rule ext*) (*simp add*: *map-add-def*)

lemma *map-add-upds*[*simp*]: $m1 ++ (m2(xs[\mapsto])ys)) = (m1 ++ m2)(xs[\mapsto])ys)$
by (*simp add*: *map-upds-def*)

lemma *map-add-upd-left*: $m \notin dom\ e2 \Longrightarrow e1(m \mapsto u1) ++ e2 = (e1 ++ e2)(m \mapsto u1)$
by (*rule ext*) (*auto simp*: *map-add-def dom-def split*: *option.split*)

lemma *map-of-append*[*simp*]: $map\text{-}of\ (xs\ @\ ys) = map\text{-}of\ ys ++ map\text{-}of\ xs$
unfolding *map-add-def*
proof (*induct xs*)
case (*Cons a xs*)
then show *?case*
by (*force split*: *option.split*)
qed *auto*

lemma *finite-range-map-of-map-add*:

finite (*range f*) ⟹ *finite* (*range* (*f* ++ *map-of l*))
proof (*induct l*)
case (*Cons a l*)
then show *?case*
by (*metis finite-range-updI map-add-upd map-of.simps*(*2*))
qed *auto*

lemma *inj-on-map-add-dom* [*iff*]:
inj-on (*m* ++ *m'*) (*dom m'*) = *inj-on m'* (*dom m'*)
by (*fastforce simp*: *map-add-def dom-def inj-on-def split*: *option.splits*)

lemma *map-upds-fold-map-upd*:
m(*ks*[↦]*vs*) = *foldl* (λ*m* (*k*, *v*). *m*(*k* ↦ *v*)) *m* (*zip ks vs*)
unfolding *map-upds-def* **proof** (*rule sym*, *rule zip-obtain-same-length*)
fix *ks* :: *'a list* **and** *vs* :: *'b list*
assume *length ks* = *length vs*
then show *foldl* (λ*m* (*k*, *v*). *m*(*k*↦*v*)) *m* (*zip ks vs*) = *m* ++ *map-of* (*rev* (*zip ks vs*))
by(*induct arbitrary*: *m rule*: *list-induct2*) *simp-all*
qed

lemma *map-add-map-of-foldr*:
m ++ *map-of ps* = *foldr* (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) *ps m*
by (*induct ps*) (*auto simp*: *fun-eq-iff map-add-def*)

71.7 *restrict-map*

lemma *restrict-map-to-empty* [*simp*]: *m*|‘{} = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-map-insert*: *f* |‘ (*insert a A*) = (*f* |‘ *A*)(*a* := *f a*)
by (*auto simp*: *restrict-map-def*)

lemma *restrict-map-empty* [*simp*]: *empty*|‘*D* = *empty*
by (*simp add*: *restrict-map-def*)

lemma *restrict-in* [*simp*]: *x* ∈ *A* ⟹ (*m*|‘*A*) *x* = *m x*
by (*simp add*: *restrict-map-def*)

lemma *restrict-out* [*simp*]: *x* ∉ *A* ⟹ (*m*|‘*A*) *x* = *None*
by (*simp add*: *restrict-map-def*)

lemma *ran-restrictD*: *y* ∈ *ran* (*m*|‘*A*) ⟹ ∃ *x*∈*A*. *m x* = *Some y*
by (*auto simp*: *restrict-map-def ran-def split*: *if-split-asm*)

lemma *dom-restrict* [*simp*]: *dom* (*m*|‘*A*) = *dom m* ∩ *A*
by (*auto simp*: *restrict-map-def dom-def split*: *if-split-asm*)

lemma *restrict-upd-same* [*simp*]: *m*(*x*↦*y*)|‘(−{*x*}) = *m*|‘(−{*x*})

by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-restrict* [*simp*]: $m|`A|`B = m|`(A \cap B)$
by (*rule ext*) (*auto simp*: *restrict-map-def*)

lemma *restrict-fun-upd* [*simp*]:
$m(x := y)|`D = (\textit{if } x \in D \textit{ then } (m|`(D - \{x\}))(x := y) \textit{ else } m|`D)$
by (*simp add*: *restrict-map-def fun-eq-iff*)

lemma *fun-upd-None-restrict* [*simp*]:
$(m|`D)(x := None) = (\textit{if } x \in D \textit{ then } m|`(D - \{x\}) \textit{ else } m|`D)$
by (*simp add*: *restrict-map-def fun-eq-iff*)

lemma *fun-upd-restrict*: $(m|`D)(x := y) = (m|`(D - \{x\}))(x := y)$
by (*simp add*: *restrict-map-def fun-eq-iff*)

lemma *fun-upd-restrict-conv* [*simp*]:
$x \in D \Longrightarrow (m|`D)(x := y) = (m|`(D - \{x\}))(x := y)$
by (*rule fun-upd-restrict*)

lemma *map-of-map-restrict*:
map-of (*map* ($\lambda k.$ $(k, f\ k)$) *ks*) = (*Some* $\circ$ *f*) |` *set ks*
by (*induct ks*) (*simp-all add*: *fun-eq-iff restrict-map-insert*)

lemma *restrict-complement-singleton-eq*:
f |` (− {*x*}) = *f*(*x* := *None*)
by *auto*

71.8 *map-upds*

lemma *map-upds-Nil1* [*simp*]: $m([]\ [\mapsto]\ bs) = m$
by (*simp add*: *map-upds-def*)

lemma *map-upds-Nil2* [*simp*]: $m(as\ [\mapsto]\ []) = m$
by (*simp add*:*map-upds-def*)

lemma *map-upds-Cons* [*simp*]: $m(a\#as\ [\mapsto]\ b\#bs) = (m(a \mapsto b))(as[\mapsto]bs)$
by (*simp add*:*map-upds-def*)

lemma *map-upds-append1* [*simp*]:
size xs < *size ys* $\Longrightarrow$ $m(xs@[x]\ [\mapsto]\ ys) = m(xs\ [\mapsto]\ ys,\ x \mapsto ys!size\ xs)$
proof (*induct xs arbitrary*: *ys m*)
case *Nil*
then show *?case*
by (*auto simp*: *neq-Nil-conv*)
next
case (*Cons a xs*)
then show *?case*
by (*cases ys*) *auto*

qed

lemma *map-upds-list-update2-drop* [*simp*]:
size xs ≤ *i* ⟹ *m*(*xs*[↦]*ys*[*i*:=*y*]) = *m*(*xs*[↦]*ys*)
proof (*induct xs arbitrary*: *m ys i*)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons a xs*)
then show *?case*
by (*cases ys*) (*use Cons* **in** ‹*auto split*: *nat.split*›)
qed

Something weirdly sensitive about this proof, which needs only four lines in apply style

lemma *map-upd-upds-conv-if*:
(*f*(*x*↦*y*))(*xs* [↦] *ys*) =
(*if x* ∈ *set*(*take* (*length ys*) *xs*) *then f*(*xs* [↦] *ys*)
else (*f*(*xs* [↦] *ys*))(*x*↦*y*))
proof (*induct xs arbitrary*: *x y ys f*)
case (*Cons a xs*)
show *?case*
proof (*cases ys*)
case (*Cons z zs*)
then show *?thesis*
using *Cons.hyps*
apply (*auto split*: *if-split simp*: *fun-upd-twist*)
using *Cons.hyps* **apply** *fastforce*+
done
qed *auto*
qed *auto*

lemma *map-upds-twist* [*simp*]:
a ∉ *set as* ⟹ *m*(*a*↦*b*, *as*[↦]*bs*) = *m*(*as*[↦]*bs*, *a*↦*b*)
using *set-take-subset* **by** (*fastforce simp add*: *map-upd-upds-conv-if*)

lemma *map-upds-apply-nontin* [*simp*]:
x ∉ *set xs* ⟹ (*f*(*xs*[↦]*ys*)) *x* = *f x*
proof (*induct xs arbitrary*: *ys*)
case (*Cons a xs*)
then show *?case*
by (*cases ys*) (*auto simp*: *map-upd-upds-conv-if*)
qed *auto*

lemma *fun-upds-append-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*@*zs*[↦]*ys*) = *m*(*xs*[↦]*ys*)
proof (*induct xs arbitrary*: *ys*)

case (*Cons a xs*)
then show *?case*
by (*cases ys*) (*auto simp*: *map-upd-upds-conv-if*)
qed *auto*

lemma *fun-upds-append2-drop* [*simp*]:
size xs = *size ys* ⟹ *m*(*xs*[↦]*ys*@*zs*) = *m*(*xs*[↦]*ys*)
proof (*induct xs arbitrary*: *ys*)
case (*Cons a xs*)
then show *?case*
by (*cases ys*) (*auto simp*: *map-upd-upds-conv-if*)
qed *auto*

lemma *restrict-map-upds*[*simp*]:
⟦ *length xs* = *length ys*; *set xs* ⊆ *D* ⟧
⟹ *m*(*xs* [↦] *ys*)|‘*D* = (*m*|‘(*D* − *set xs*))(*xs* [↦] *ys*)
proof (*induct xs arbitrary*: *m ys*)
case (*Cons a xs*)
then show *?case*
proof (*cases ys*)
case (*Cons z zs*)
with *Cons.hyps Cons.prems* **show** *?thesis*
apply (*simp add*: *insert-absorb flip*: *Diff-insert*)
apply (*auto simp add*: *map-upd-upds-conv-if*)
done
qed *auto*
qed *auto*

71.9 *dom*

lemma *dom-eq-empty-conv* [*simp*]: *dom f* = {} ⟷ *f* = *empty*
by (*auto simp*: *dom-def*)

lemma *domI*: *m a* = *Some b* ⟹ *a* ∈ *dom m*
by (*simp add*: *dom-def*)

lemma *domD*: *a* ∈ *dom m* ⟹ ∃ *b*. *m a* = *Some b*
by (*cases m a*) (*auto simp add*: *dom-def*)

lemma *domIff* [*iff*, *simp del*, *code-unfold*]: *a* ∈ *dom m* ⟷ *m a* ≠ *None*
by (*simp add*: *dom-def*)

lemma *dom-empty* [*simp*]: *dom empty* = {}
by (*simp add*: *dom-def*)

lemma *dom-fun-upd* [*simp*]:
dom(*f*(*x* := *y*)) = (*if y* = *None then dom f* − {*x*} *else insert x* (*dom f*))
by (*auto simp*: *dom-def*)

lemma *inj-on-the-image-dom*: *inj-on the* (*mu ‘ dom mu*)
by (*rule inj-onI*) *force*

lemma *dom-if*:
dom (λ*x*. *if P x then f x else g x*) = *dom f* ∩ {*x*. *P x*} ∪ *dom g* ∩ {*x*. ¬ *P x*}
by (*auto split*: *if-splits*)

lemma *dom-map-of-conv-image-fst*:
dom (*map-of xys*) = *fst ‘ set xys*
by (*induct xys*) (*auto simp add*: *dom-if*)

lemma *dom-map-of-zip* [*simp*]: *length xs* = *length ys* ⟹ *dom* (*map-of* (*zip xs ys*))
= *set xs*
by (*induct rule*: *list-induct2*) (*auto simp*: *dom-if*)

lemma *finite-dom-map-of*: *finite* (*dom* (*map-of l*))
by (*induct l*) (*auto simp*: *dom-def insert-Collect* [*symmetric*])

lemma *dom-map-upds* [*simp*]:
dom(*m*(*xs*[↦]*ys*)) = *set*(*take* (*length ys*) *xs*) ∪ *dom m*
proof (*induct xs arbitrary*: *ys*)
case (*Cons a xs*)
then show *?case*
by (*cases ys*) (*auto simp*: *map-upd-upds-conv-if*)
qed *auto*

lemma *dom-map-add* [*simp*]: *dom* (*m* ++ *n*) = *dom n* ∪ *dom m*
by (*auto simp*: *dom-def*)

lemma *dom-override-on* [*simp*]:
dom (*override-on f g A*) =
(*dom f* − {*a*. *a* ∈ *A* − *dom g*}) ∪ {*a*. *a* ∈ *A* ∩ *dom g*}
by (*auto simp*: *dom-def override-on-def*)

lemma *map-add-comm*: *dom m1* ∩ *dom m2* = {} ⟹ *m1* ++ *m2* = *m2* ++ *m1*
by (*rule ext*) (*force simp*: *map-add-def dom-def split*: *option.split*)

lemma *map-add-dom-app-simps*:
m ∈ *dom l2* ⟹ (*l1* ++ *l2*) *m* = *l2 m*
m ∉ *dom l1* ⟹ (*l1* ++ *l2*) *m* = *l2 m*
m ∉ *dom l2* ⟹ (*l1* ++ *l2*) *m* = *l1 m*
by (*auto simp add*: *map-add-def split*: *option.split-asm*)

lemma *dom-const* [*simp*]:
dom (λ*x*. *Some* (*f x*)) = *UNIV*
by *auto*

lemma *finite-map-freshness*:
finite (*dom* (*f* :: *'a* ⇀ *'b*)) ⟹ ¬ *finite* (*UNIV* :: *'a set*) ⟹
∃ *x*. *f x* = *None*
by (*bestsimp dest*: *ex-new-if-finite*)

lemma *dom-minus*:
f x = *None* ⟹ *dom f* − *insert x A* = *dom f* − *A*
unfolding *dom-def* **by** *simp*

lemma *insert-dom*:
f x = *Some y* ⟹ *insert x* (*dom f*) = *dom f*
unfolding *dom-def* **by** *auto*

lemma *map-of-map-keys*:
set xs = *dom m* ⟹ *map-of* (*map* (λ*k*. (*k*, *the* (*m k*))) *xs*) = *m*
by (*rule ext*) (*auto simp add*: *map-of-map-restrict restrict-map-def*)

lemma *map-of-eqI*:
assumes *set-eq*: *set* (*map fst xs*) = *set* (*map fst ys*)
assumes *map-eq*: ∀ *k*∈*set* (*map fst xs*). *map-of xs k* = *map-of ys k*
shows *map-of xs* = *map-of ys*
proof (*rule ext*)
fix *k* **show** *map-of xs k* = *map-of ys k*
proof (*cases map-of xs k*)
case *None*
then have *k* ∉ *set* (*map fst xs*) **by** (*simp add*: *map-of-eq-None-iff*)
with *set-eq* **have** *k* ∉ *set* (*map fst ys*) **by** *simp*
then have *map-of ys k* = *None* **by** (*simp add*: *map-of-eq-None-iff*)
with *None* **show** *?thesis* **by** *simp*
next
case (*Some v*)
then have *k* ∈ *set* (*map fst xs*) **by** (*auto simp add*: *dom-map-of-conv-image-fst* [*symmetric*])
with *map-eq* **show** *?thesis* **by** *auto*
qed
qed

lemma *map-of-eq-dom*:
assumes *map-of xs* = *map-of ys*
shows *fst* ‘ *set xs* = *fst* ‘ *set ys*
proof −
from *assms* **have** *dom* (*map-of xs*) = *dom* (*map-of ys*) **by** *simp*
then show *?thesis* **by** (*simp add*: *dom-map-of-conv-image-fst*)
qed

lemma *finite-set-of-finite-maps*:
assumes *finite A finite B*
shows *finite* {*m*. *dom m* = *A* ∧ *ran m* ⊆ *B*} (**is** *finite ?S*)

proof −
let *?S′* = {*m*. ∀ *x*. (*x* ∈ *A* ⟶ *m x* ∈ *Some* ‘ *B*) ∧ (*x* ∉ *A* ⟶ *m x* = *None*)}
have *?S* = *?S′*
proof
show *?S* ⊆ *?S′* **by** (*auto simp*: *dom-def ran-def image-def*)
show *?S′* ⊆ *?S*
proof
fix *m* **assume** *m* ∈ *?S′*
hence *1*: *dom m* = *A* **by** *force*
hence *2*: *ran m* ⊆ *B* **using** ‹*m* ∈ *?S′*› **by** (*auto simp*: *dom-def ran-def*)
from *1 2* **show** *m* ∈ *?S* **by** *blast*
qed
qed
with *assms* **show** *?thesis* **by**(*simp add*: *finite-set-of-finite-funs*)
qed

71.10 *ran*

lemma *ranI*: *m a* = *Some b* ⟹ *b* ∈ *ran m*
by (*auto simp*: *ran-def*)

lemma *ran-empty* [*simp*]: *ran empty* = {}
by (*auto simp*: *ran-def*)

lemma *ran-conv-dom*: *ran mu* = *the* ‘ *mu* ‘ *dom mu*
proof (*intro subset-antisym subsetI*)
fix *b*
assume *b* ∈ *ran mu*
then obtain *a* **where** *mu a* = *Some b*
unfolding *ran-def* **by** *blast*
then show *b* ∈ *the* ‘ *mu* ‘ *dom mu*
by *force*
next
fix *b*
assume *b* ∈ *the* ‘ *mu* ‘ *dom mu*
then obtain *a* **where** *a* ∈ *dom mu* **and** *b* = *the* (*mu a*)
by *auto*
then show *b* ∈ *ran mu*
by (*auto intro*: *ranI*)
qed

lemma *ran-map-upd* [*simp*]: *m a* = *None* ⟹ *ran*(*m*(*a*↦*b*)) = *insert b* (*ran m*)
unfolding *ran-def*
by *force*

lemma *fun-upd-None-if-notin-dom*[*simp*]: *k* ∉ *dom m* ⟹ *m*(*k* := *None*) = *m*
by *auto*

lemma *ran-map-upd-Some*:
⟦ *m x* = *Some y*; *inj-on m* (*dom m*); *z* ∉ *ran m* ⟧ ⟹ *ran*(*m*(*x* := *Some z*)) = *ran m* − {*y*} ∪ {*z*}
by(*force simp add*: *ran-def domI inj-onD*)

lemma *ran-map-add*:
assumes *dom m1* ∩ *dom m2* = {}
shows *ran* (*m1* ++ *m2*) = *ran m1* ∪ *ran m2*
proof
show *ran* (*m1* ++ *m2*) ⊆ *ran m1* ∪ *ran m2*
unfolding *ran-def* **by** *auto*
next
show *ran m1* ∪ *ran m2* ⊆ *ran* (*m1* ++ *m2*)
proof −
have (*m1* ++ *m2*) *x* = *Some y* **if** *m1 x* = *Some y* **for** *x y*
using *assms map-add-comm that* **by** *fastforce*
moreover have (*m1* ++ *m2*) *x* = *Some y* **if** *m2 x* = *Some y* **for** *x y*
using *assms that* **by** *auto*
ultimately show *?thesis*
unfolding *ran-def* **by** *blast*
qed
qed

lemma *finite-ran*:
assumes *finite* (*dom p*)
shows *finite* (*ran p*)
proof −
have *ran p* = (λ*x*. *the* (*p x*)) ‘ *dom p*
unfolding *ran-def* **by** *force*
from *this* ‹*finite* (*dom p*)› **show** *?thesis* **by** *auto*
qed

lemma *ran-distinct*:
assumes *dist*: *distinct* (*map fst al*)
shows *ran* (*map-of al*) = *snd* ‘ *set al*
using *assms*
proof (*induct al*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons kv al*)
then have *ran* (*map-of al*) = *snd* ‘ *set al* **by** *simp*
moreover from *Cons.prems* **have** *map-of al* (*fst kv*) = *None*
by (*simp add*: *map-of-eq-None-iff*)
ultimately show *?case* **by** (*simp only*: *map-of.simps ran-map-upd*) *simp*
qed

lemma *ran-map-of-zip*:
assumes *length xs* = *length ys distinct xs*

shows *ran* (*map-of* (*zip xs ys*)) = *set ys*
using *assms* **by** (*simp add*: *ran-distinct set-map*[*symmetric*])

lemma *ran-map-option*: *ran* (λ*x*. *map-option f* (*m x*)) = *f* ‘ *ran m*
by (*auto simp add*: *ran-def*)

71.11 *graph*

lemma *graph-empty*[*simp*]: *graph empty* = {}
unfolding *graph-def* **by** *simp*

lemma *in-graphI*: *m k* = *Some v* ⟹ (*k*, *v*) ∈ *graph m*
unfolding *graph-def* **by** *blast*

lemma *in-graphD*: (*k*, *v*) ∈ *graph m* ⟹ *m k* = *Some v*
unfolding *graph-def* **by** *blast*

lemma *graph-map-upd*[*simp*]: *graph* (*m*(*k* ↦ *v*)) = *insert* (*k*, *v*) (*graph* (*m*(*k* := *None*)))
unfolding *graph-def* **by** (*auto split*: *if-splits*)

lemma *graph-fun-upd-None*: *graph* (*m*(*k* := *None*)) = {*e* ∈ *graph m*. *fst e* ≠ *k*}
unfolding *graph-def* **by** (*auto split*: *if-splits*)

lemma *graph-restrictD*:
assumes (*k*, *v*) ∈ *graph* (*m* |‘ *A*)
shows *k* ∈ *A* **and** *m k* = *Some v*
using *assms* **unfolding** *graph-def*
by (*auto simp*: *restrict-map-def split*: *if-splits*)

lemma *graph-map-comp*[*simp*]: *graph* (*m1* ∘$_m$ *m2*) = *graph m2 O graph m1*
unfolding *graph-def* **by** (*auto simp*: *map-comp-Some-iff relcomp-unfold*)

lemma *graph-map-add*: *dom m1* ∩ *dom m2* = {} ⟹ *graph* (*m1* ++ *m2*) = *graph m1* ∪ *graph m2*
unfolding *graph-def* **using** *map-add-comm* **by** *force*

lemma *graph-eq-to-snd-dom*: *graph m* = (λ*x*. (*x*, *the* (*m x*))) ‘ *dom m*
unfolding *graph-def dom-def* **by** *force*

lemma *fst-graph-eq-dom*: *fst* ‘ *graph m* = *dom m*
unfolding *graph-eq-to-snd-dom* **by** *force*

lemma *graph-domD*: *x* ∈ *graph m* ⟹ *fst x* ∈ *dom m*
using *fst-graph-eq-dom* **by** (*metis imageI*)

lemma *snd-graph-ran*: *snd* ‘ *graph m* = *ran m*
unfolding *graph-def ran-def* **by** *force*

lemma *graph-ranD*: $x \in$ *graph* $m \Longrightarrow$ *snd* $x \in$ *ran* m
using *snd-graph-ran* **by** (*metis imageI*)

lemma *finite-graph-map-of*: *finite* (*graph* (*map-of al*))
unfolding *graph-eq-to-snd-dom finite-dom-map-of*
using *finite-dom-map-of* **by** *blast*

lemma *graph-map-of-if-distinct-dom*: *distinct* (*map fst al*) $\Longrightarrow$ *graph* (*map-of al*) = *set al*
unfolding *graph-def* **by** *auto*

lemma *finite-graph-iff-finite-dom*[*simp*]: *finite* (*graph m*) = *finite* (*dom m*)
by (*metis graph-eq-to-snd-dom finite-imageI fst-graph-eq-dom*)

lemma *inj-on-graph*: *inj-on Map.graph A*
proof (*rule inj-onI*)
fix *x y*
assume *Map.graph x* = *Map.graph y*
hence (*x a* = *Some b*) = (*y a* = *Some b*) **for** *a b*
unfolding *Map.graph-def* **by** *auto*
hence *x k* = *y k* **for** *k*
by (*metis not-None-eq*)
thus *x* = *y* **..**
qed

lemma *inj-on-fst-graph*: *inj-on fst* (*graph m*)
unfolding *graph-def inj-on-def* **by** *force*

71.12 *map-le*

lemma *map-le-empty* [*simp*]: *empty* $\subseteq_m$ *g*
by (*simp add*: *map-le-def*)

lemma *upd-None-map-le* [*simp*]: *f*(*x* := *None*) $\subseteq_m$ *f*
by (*force simp add*: *map-le-def*)

lemma *map-le-upd*[*simp*]: *f* $\subseteq_m$ *g* ==> *f*(*a* := *b*) $\subseteq_m$ *g*(*a* := *b*)
by (*fastforce simp add*: *map-le-def*)

lemma *map-le-imp-upd-le* [*simp*]: *m1* $\subseteq_m$ *m2* $\Longrightarrow$ *m1*(*x* := *None*) $\subseteq_m$ *m2*(*x* $\mapsto$ *y*)
by (*force simp add*: *map-le-def*)

lemma *map-le-upds* [*simp*]:
f $\subseteq_m$ *g* $\Longrightarrow$ *f*(*as* [$\mapsto$] *bs*) $\subseteq_m$ *g*(*as* [$\mapsto$] *bs*)
proof (*induct as arbitrary*: *f g bs*)
case (*Cons a as*)
then show *?case*
by (*cases bs*) (*use Cons* **in** *auto*)

qed *auto*

lemma *map-le-implies-dom-le*: $(f \subseteq_m g) \Longrightarrow (dom\ f \subseteq dom\ g)$
by (*fastforce simp add*: *map-le-def dom-def*)

lemma *map-le-refl* [*simp*]: $f \subseteq_m f$
by (*simp add*: *map-le-def*)

lemma *map-le-trans*[*trans*]: $[\![\ m1 \subseteq_m m2;\ m2 \subseteq_m m3]\!] \Longrightarrow m1 \subseteq_m m3$
by (*auto simp add*: *map-le-def dom-def*)

lemma *map-le-antisym*: $[\![\ f \subseteq_m g;\ g \subseteq_m f\ ]\!] \Longrightarrow f = g$
unfolding *map-le-def*
by (*metis ext domIff*)

lemma *map-le-map-add* [*simp*]: $f \subseteq_m g\ ++\ f$
by (*fastforce simp*: *map-le-def*)

lemma *map-le-iff-map-add-commute*: $f \subseteq_m f\ ++\ g \longleftrightarrow f\ ++\ g = g\ ++\ f$
by (*fastforce simp*: *map-add-def map-le-def fun-eq-iff split*: *option.splits*)

lemma *map-add-le-mapE*: $f\ ++\ g \subseteq_m h \Longrightarrow g \subseteq_m h$
by (*fastforce simp*: *map-le-def map-add-def dom-def*)

lemma *map-add-le-mapI*: $[\![\ f \subseteq_m h;\ g \subseteq_m h\ ]\!] \Longrightarrow f\ ++\ g \subseteq_m h$
by (*auto simp*: *map-le-def map-add-def dom-def split*: *option.splits*)

lemma *map-add-subsumed1*: $f \subseteq_m g \Longrightarrow f{+}{+}g = g$
by (*simp add*: *map-add-le-mapI map-le-antisym*)

lemma *map-add-subsumed2*: $f \subseteq_m g \Longrightarrow g{+}{+}f = g$
by (*metis map-add-subsumed1 map-le-iff-map-add-commute*)

lemma *dom-eq-singleton-conv*: $dom\ f = \{x\} \longleftrightarrow (\exists\, v.\ f = [x \mapsto v])$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
then show *?lhs* **by** (*auto split*: *if-split-asm*)
next
assume *?lhs*
then obtain *v* **where** *v*: $f\ x = Some\ v$ **by** *auto*
show *?rhs*
proof
show $f = [x \mapsto v]$
proof (*rule map-le-antisym*)
show $[x \mapsto v] \subseteq_m f$
using *v* **by** (*auto simp add*: *map-le-def*)
show $f \subseteq_m [x \mapsto v]$
using ‹$dom\ f = \{x\}$› ‹$f\ x = Some\ v$› **by** (*auto simp add*: *map-le-def*)

qed
qed
qed

lemma *map-add-eq-empty-iff*[*simp*]:
(*f*++*g* = *empty*) ⟷ *f* = *empty* ∧ *g* = *empty*
by (*metis map-add-None*)

lemma *empty-eq-map-add-iff*[*simp*]:
(*empty* = *f*++*g*) ⟷ *f* = *empty* ∧ *g* = *empty*
by(*subst map-add-eq-empty-iff*[*symmetric*])(*rule eq-commute*)

71.13 Various

lemma *card-dom-eq-card-range*: *inj-on mu* (*dom mu*) ⟹ *card* (*dom mu*) = *card* (*ran mu*)
by (*simp only*: *ran-conv-dom card-image inj-on-the-image-dom*)

lemma *set-map-of-compr*:
assumes *distinct*: *distinct* (*map fst xs*)
shows *set xs* = {(*k*, *v*). *map-of xs k* = *Some v*}
using *assms*
proof (*induct xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
obtain *k v* **where** *x* = (*k*, *v*) **by** (*cases x*) *blast*
with *Cons.prems* **have** *k* ∉ *dom* (*map-of xs*)
by (*simp add*: *dom-map-of-conv-image-fst*)
then have ∗: *insert* (*k*, *v*) {(*k*, *v*). *map-of xs k* = *Some v*} =
{(*k*′, *v*′). ((*map-of xs*)(*k* ↦ *v*)) *k*′ = *Some v*′}
by (*auto split*: *if-splits*)
from *Cons* **have** *set xs* = {(*k*, *v*). *map-of xs k* = *Some v*} **by** *simp*
with ∗ ‹*x* = (*k*, *v*)› **show** *?case* **by** *simp*
qed

lemma *eq-key-imp-eq-value*:
v1 = *v2*
if *distinct* (*map fst xs*) (*k*, *v1*) ∈ *set xs* (*k*, *v2*) ∈ *set xs*
proof −
from *that* **have** *inj-on fst* (*set xs*)
by (*simp add*: *distinct-map*)
moreover have *fst* (*k*, *v1*) = *fst* (*k*, *v2*)
by *simp*
ultimately have (*k*, *v1*) = (*k*, *v2*)
by (*rule inj-onD*) (*fact that*)+
then show *?thesis*
by *simp*

qed

lemma *map-of-inject-set*:
assumes *distinct*: *distinct* (*map fst xs*) *distinct* (*map fst ys*)
shows *map-of xs* = *map-of ys* ⟷ *set xs* = *set ys* (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs*
moreover from ‹*distinct* (*map fst xs*)› **have** *set xs* = {(*k*, *v*). *map-of xs k* = *Some v*}
by (*rule set-map-of-compr*)
moreover from ‹*distinct* (*map fst ys*)› **have** *set ys* = {(*k*, *v*). *map-of ys k* = *Some v*}
by (*rule set-map-of-compr*)
ultimately show *?rhs* **by** *simp*
next
assume *?rhs* **show** *?lhs*
proof
fix *k*
show *map-of xs k* = *map-of ys k*
proof (*cases map-of xs k*)
case *None*
with ‹*?rhs*› **have** *map-of ys k* = *None*
by (*simp add*: *map-of-eq-None-iff*)
with *None* **show** *?thesis* **by** *simp*
next
case (*Some v*)
with *distinct* ‹*?rhs*› **have** *map-of ys k* = *Some v*
by *simp*
with *Some* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *finite-Map-induct*[*consumes 1*, *case-names empty update*]:
assumes *finite* (*dom m*)
assumes *P Map.empty*
assumes ⋀*k v m*. *finite* (*dom m*) ⟹ *k* ∉ *dom m* ⟹ *P m* ⟹ *P* (*m*(*k* ↦ *v*))
shows *P m*
using *assms*(*1*)
proof(*induction dom m arbitrary*: *m rule*: *finite-induct*)
case *empty*
then show *?case* **using** *assms*(*2*) **unfolding** *dom-def* **by** *simp*
next
case (*insert x F*)
then have *finite* (*dom* (*m*(*x*:=*None*))) *x* ∉ *dom* (*m*(*x*:=*None*)) *P* (*m*(*x*:=*None*))
by (*metis Diff-insert-absorb dom-fun-upd*)+
with *assms*(*3*)[*OF this*] **show** *?case*
by (*metis fun-upd-triv fun-upd-upd option.exhaust*)
qed

hide-const (**open**) *Map.empty Map.graph*

end

72 Finite types as explicit enumerations

theory *Enum*
imports *Map Groups-List*
begin

72.1 Class *enum*

class *enum* =
  fixes *enum* :: *'a list*
  fixes *enum-all* :: (*'a* ⇒ *bool*) ⇒ *bool*
  fixes *enum-ex* :: (*'a* ⇒ *bool*) ⇒ *bool*
  assumes *UNIV-enum*: *UNIV* = *set enum*
    and *enum-distinct*: *distinct enum*
  assumes *enum-all-UNIV*: *enum-all P* ⟷ *Ball UNIV P*
  assumes *enum-ex-UNIV*: *enum-ex P* ⟷ *Bex UNIV P*
   — tailored towards simple instantiation
begin

subclass *finite* **proof**
qed (*simp add*: *UNIV-enum*)

lemma *enum-UNIV*:
  set enum = *UNIV*
  by (*simp only*: *UNIV-enum*)

lemma *in-enum*: *x* ∈ *set enum*
  by (*simp add*: *enum-UNIV*)

lemma *enum-eq-I*:
  assumes ⋀*x*. *x* ∈ *set xs*
  shows *set enum* = *set xs*
proof −
  from *assms UNIV-eq-I* **have** *UNIV* = *set xs* **by** *auto*
  with *enum-UNIV* **show** *?thesis* **by** *simp*
qed

lemma *card-UNIV-length-enum*:
  card (*UNIV* :: *'a set*) = *length enum*
  by (*simp add*: *UNIV-enum distinct-card enum-distinct*)

lemma *enum-all* [*simp*]:
  enum-all = *HOL.All*
  by (*simp add*: *fun-eq-iff enum-all-UNIV*)

lemma *enum-ex* [*simp*]:
enum-ex = *HOL.Ex*
by (*simp add*: *fun-eq-iff enum-ex-UNIV*)

end

72.2 Implementations using *enum*

72.2.1 Unbounded operations and quantifiers

lemma *Collect-code* [*code*]:
Collect P = *set* (*filter P enum*)
by (*simp add*: *enum-UNIV*)

lemma *vimage-code* [*code*]:
f −‘ *B* = *set* (*filter* (λ*x*. *f x* ∈ *B*) *enum-class.enum*)
unfolding *vimage-def Collect-code* **..**

definition *card-UNIV* :: *'a itself* ⇒ *nat*
where
card-UNIV TYPE('*a*) = *card* (*UNIV* :: *'a set*)

lemma [*code*]:
card-UNIV TYPE('*a* :: *enum*) = *card* (*set* (*Enum.enum* :: *'a list*))
by (*simp only*: *card-UNIV-def enum-UNIV*)

lemma *all-code* [*code*]: (∀ *x*. *P x*) ⟷ *enum-all P*
by *simp*

lemma *exists-code* [*code*]: (∃ *x*. *P x*) ⟷ *enum-ex P*
by *simp*

lemma *exists1-code* [*code*]: (∃!*x*. *P x*) ⟷ *list-ex1 P enum*
by (*simp add*: *list-ex1-iff enum-UNIV*)

72.2.2 An executable choice operator

definition
enum-the = *The*

lemma [*code*]:
The P = (*case filter P enum of* [*x*] ⇒ *x* | - ⇒ *enum-the P*)
proof −
{
fix *a*
assume *filter-enum*: *filter P enum* = [*a*]
have *The P* = *a*
proof (*rule the-equality*)
fix *x*

```
    assume P x
    show x = a
    proof (rule ccontr)
     assume x ≠ a
     from filter-enum obtain us vs
       where enum-eq: enum = us @ [a] @ vs
       and ∀ x ∈ set us. ¬ P x
       and ∀ x ∈ set vs. ¬ P x
       and P a
     by (auto simp add: filter-eq-Cons-iff) (simp only: filter-empty-conv[symmetric])
      with ‹P x› in-enum[of x, unfolded enum-eq] ‹x ≠ a› show False by auto
    qed
   next
    from filter-enum show P a by (auto simp add: filter-eq-Cons-iff)
   qed
  }
  from this show ?thesis
   unfolding enum-the-def by (auto split: list.split)
qed

declare [[code abort: enum-the]]

code-printing
  constant enum-the ⇀ (Eval) (fn '- => raise Match)
```

72.2.3 Equality and order on functions

```
instantiation fun :: (enum, equal) equal
begin

definition
  HOL.equal f g ⟷ (∀ x ∈ set enum. f x = g x)

instance proof
qed (simp-all add: equal-fun-def fun-eq-iff enum-UNIV)

end

lemma [code]:
  HOL.equal f g ⟷ enum-all (%x. f x = g x)
  by (auto simp add: equal fun-eq-iff)

lemma [code nbe]:
  HOL.equal (f :: - ⇒ -) f ⟷ True
  by (fact equal-refl)

lemma order-fun [code]:
  fixes f g :: 'a::enum ⇒ 'b::order
  shows f ≤ g ⟷ enum-all (λx. f x ≤ g x)
```

and $f < g \longleftrightarrow f \leq g \wedge$ *enum-ex* $(\lambda x.\ f\ x \neq g\ x)$
by (*simp-all add*: *fun-eq-iff le-fun-def order-less-le*)

72.2.4 Operations on relations

lemma [*code*]:
$Id =$ *image* $(\lambda x.\ (x,\ x))$ (*set Enum.enum*)
by (*auto intro*: *imageI in-enum*)

lemma *tranclp-unfold* [*code*]:
tranclp r a b $\longleftrightarrow (a,\ b) \in$ *trancl* $\{(x,\ y).\ r\ x\ y\}$
by (*simp add*: *trancl-def*)

lemma *rtranclp-rtrancl-eq* [*code*]:
rtranclp r x y $\longleftrightarrow (x,\ y) \in$ *rtrancl* $\{(x,\ y).\ r\ x\ y\}$
by (*simp add*: *rtrancl-def*)

lemma *max-ext-eq* [*code*]:
max-ext R $= \{(X,\ Y).$ *finite* $X \wedge$ *finite* $Y \wedge\ Y \neq \{\} \wedge (\forall x.\ x \in X \longrightarrow (\exists\, xa \in Y.\ (x,\ xa) \in R))\}$
by (*auto simp add*: *max-ext.simps*)

lemma *max-extp-eq* [*code*]:
max-extp r x y $\longleftrightarrow (x,\ y) \in$ *max-ext* $\{(x,\ y).\ r\ x\ y\}$
by (*simp add*: *max-ext-def*)

lemma *mlex-eq* [*code*]:
$f <*mlex*> R = \{(x,\ y).\ f\ x < f\ y \vee (f\ x \leq f\ y \wedge (x,\ y) \in R)\}$
by (*auto simp add*: *mlex-prod-def*)

72.2.5 Bounded accessible part

primrec *bacc* :: $('a \times 'a)$ *set* $\Rightarrow$ *nat* $\Rightarrow$ $'a$ *set*
where
bacc r 0 $= \{x.\ \forall\ y.\ (y,\ x) \notin r\}$
| *bacc r* (*Suc n*) $=$ (*bacc r n* $\cup \{x.\ \forall y.\ (y,\ x) \in r \longrightarrow y \in$ *bacc r n*$\}$)

lemma *bacc-subseteq-acc*:
bacc r n $\subseteq$ *Wellfounded.acc r*
by (*induct n*) (*auto intro*: *acc.intros*)

lemma *bacc-mono*:
$n \leq m \Longrightarrow$ *bacc r n* $\subseteq$ *bacc r m*
by (*induct rule*: *dec-induct*) *auto*

lemma *bacc-upper-bound*:
bacc (*r* :: $('a \times 'a)$ *set*) (*card* (*UNIV* :: $'a$::*finite set*)) $= (\bigcup n.$ *bacc r n*$)$
proof −
have *mono* (*bacc r*) **unfolding** *mono-def* **by** (*simp add*: *bacc-mono*)

moreover have $\forall n.\ bacc\ r\ n = bacc\ r\ (Suc\ n) \longrightarrow bacc\ r\ (Suc\ n) = bacc\ r\ (Suc\ (Suc\ n))$ **by** *auto*
moreover have *finite* (*range* (*bacc r*)) **by** *auto*
ultimately show *?thesis*
by (*intro finite-mono-strict-prefix-implies-finite-fixpoint*)
(*auto intro*: *finite-mono-remains-stable-implies-strict-prefix*)
qed

lemma *acc-subseteq-bacc*:
assumes *finite r*
shows *Wellfounded.acc r* $\subseteq (\bigcup n.\ bacc\ r\ n)$
proof
fix *x*
assume $x \in$ *Wellfounded.acc r*
then have $\exists n.\ x \in bacc\ r\ n$
proof (*induct x arbitrary*: *rule*: *acc.induct*)
case (*accI x*)
then have $\forall y.\ \exists\ n.\ (y, x) \in r \longrightarrow y \in bacc\ r\ n$ **by** *simp*
from *choice*[*OF this*] **obtain** *n* **where** *n*: $\forall y.\ (y, x) \in r \longrightarrow y \in bacc\ r\ (n\ y)$
..
obtain *n* **where** $\bigwedge y.\ (y, x) \in r \Longrightarrow y \in bacc\ r\ n$
proof
fix *y* **assume** *y*: $(y, x) \in r$
with *n* **have** $y \in bacc\ r\ (n\ y)$ **by** *auto*
moreover have $n\ y <= Max\ ((\lambda(y, x).\ n\ y)\ `\ r)$
using *y* ‹*finite r*› **by** (*auto intro*!: *Max-ge*)
note *bacc-mono*[*OF this*, *of r*]
ultimately show $y \in bacc\ r\ (Max\ ((\lambda(y, x).\ n\ y)\ `\ r))$ **by** *auto*
qed
then show *?case*
by (*auto simp add*: *Let-def intro*!: *exI*[*of* - *Suc n*])
qed
then show $x \in (\bigcup n.\ bacc\ r\ n)$ **by** *auto*
qed

lemma *acc-bacc-eq*:
fixes $A :: ('a :: finite \times 'a)\ set$
assumes *finite A*
shows *Wellfounded.acc A* = *bacc A* (*card* (*UNIV* :: *'a set*))
using *assms* **by** (*metis acc-subseteq-bacc bacc-subseteq-acc bacc-upper-bound order-eq-iff*)

lemma [*code*]:
fixes $xs :: ('a::finite \times 'a)\ list$
shows *Wellfounded.acc* (*set xs*) = *bacc* (*set xs*) (*card-UNIV TYPE*(*'a*))
by (*simp add*: *card-UNIV-def acc-bacc-eq*)

72.3 Default instances for *enum*

lemma *map-of-zip-enum-is-Some*:
assumes *length ys* = *length* (*enum* :: *'a*::*enum list*)
shows ∃ *y*. *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x* = *Some y*
proof −
from *assms* **have** *x* ∈ *set* (*enum* :: *'a*::*enum list*) ⟷
(∃ *y*. *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x* = *Some y*)
by (*auto intro*!: *map-of-zip-is-Some*)
then show *?thesis* **using** *enum-UNIV* **by** *auto*
qed

lemma *map-of-zip-enum-inject*:
fixes *xs ys* :: *'b*::*enum list*
assumes *length*: *length xs* = *length* (*enum* :: *'a*::*enum list*)
length ys = *length* (*enum* :: *'a*::*enum list*)
and *map-of*: *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *xs*) = *the* ∘ *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*)
shows *xs* = *ys*
proof −
have *map-of* (*zip* (*enum* :: *'a list*) *xs*) = *map-of* (*zip* (*enum* :: *'a list*) *ys*)
proof
fix *x* :: *'a*
from *length map-of-zip-enum-is-Some* **obtain** *y1 y2*
where *map-of* (*zip* (*enum* :: *'a list*) *xs*) *x* = *Some y1*
and *map-of* (*zip* (*enum* :: *'a list*) *ys*) *x* = *Some y2* **by** *blast*
moreover from *map-of*
have *the* (*map-of* (*zip* (*enum* :: *'a*::*enum list*) *xs*) *x*) = *the* (*map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x*)
by (*auto dest*: *fun-cong*)
ultimately show *map-of* (*zip* (*enum* :: *'a*::*enum list*) *xs*) *x* = *map-of* (*zip* (*enum* :: *'a*::*enum list*) *ys*) *x*
by *simp*
qed
with *length enum-distinct* **show** *xs* = *ys* **by** (*rule map-of-zip-inject*)
qed

definition *all-n-lists* :: ((*'a* :: *enum*) *list* ⇒ *bool*) ⇒ *nat* ⇒ *bool*
where
all-n-lists P n ⟷ (∀ *xs* ∈ *set* (*List.n-lists n enum*). *P xs*)

lemma [*code*]:
all-n-lists P n ⟷ (*if n* = *0 then P* [] *else enum-all* (%*x*. *all-n-lists* (%*xs*. *P* (*x* # *xs*)) (*n* − *1*)))
unfolding *all-n-lists-def enum-all*
by (*cases n*) (*auto simp add*: *enum-UNIV*)

definition *ex-n-lists* :: ((*'a* :: *enum*) *list* ⇒ *bool*) ⇒ *nat* ⇒ *bool*
where
ex-n-lists P n ⟷ (∃ *xs* ∈ *set* (*List.n-lists n enum*). *P xs*)

lemma [*code*]:
ex-n-lists P n ⟷ (*if n = 0 then P* [] *else enum-ex* (%*x*. *ex-n-lists* (%*xs*. *P* (*x* # *xs*)) (*n* − *1*)))
unfolding *ex-n-lists-def enum-ex*
by (*cases n*) (*auto simp add*: *enum-UNIV*)

instantiation *fun* :: (*enum*, *enum*) *enum*
begin

definition
enum = *map* (λ*ys*. *the* ∘ *map-of* (*zip* (*enum*::′*a list*) *ys*)) (*List.n-lists* (*length* (*enum*::′*a*::*enum list*)) *enum*)

definition
enum-all P = *all-n-lists* (λ*bs*. *P* (*the* ∘ *map-of* (*zip enum bs*))) (*length* (*enum* :: ′*a list*))

definition
enum-ex P = *ex-n-lists* (λ*bs*. *P* (*the* ∘ *map-of* (*zip enum bs*))) (*length* (*enum* :: ′*a list*))

instance proof
show *UNIV* = *set* (*enum* :: (′*a* ⇒ ′*b*) *list*)
proof (*rule UNIV-eq-I*)
fix *f* :: ′*a* ⇒ ′*b*
have *f* = *the* ∘ *map-of* (*zip* (*enum* :: ′*a*::*enum list*) (*map f enum*))
by (*auto simp add*: *map-of-zip-map fun-eq-iff intro*: *in-enum*)
then show *f* ∈ *set enum*
by (*auto simp add*: *enum-fun-def set-n-lists intro*: *in-enum*)
qed
next
from *map-of-zip-enum-inject*
show *distinct* (*enum* :: (′*a* ⇒ ′*b*) *list*)
by (*auto intro*!: *inj-onI simp add*: *enum-fun-def*
distinct-map distinct-n-lists enum-distinct set-n-lists)
next
fix *P*
show *enum-all* (*P* :: (′*a* ⇒ ′*b*) ⇒ *bool*) = *Ball UNIV P*
proof
assume *enum-all P*
show *Ball UNIV P*
proof
fix *f* :: ′*a* ⇒ ′*b*
have *f*: *f* = *the* ∘ *map-of* (*zip* (*enum* :: ′*a*::*enum list*) (*map f enum*))
by (*auto simp add*: *map-of-zip-map fun-eq-iff intro*: *in-enum*)
from ‹*enum-all P*› **have** *P* (*the* ∘ *map-of* (*zip enum* (*map f enum*)))
unfolding *enum-all-fun-def all-n-lists-def*
apply (*simp add*: *set-n-lists*)

```
      apply (erule-tac x=map f enum in allE)
      apply (auto intro!: in-enum)
      done
    from this f show P f by auto
  qed
 next
  assume Ball UNIV P
  from this show enum-all P
   unfolding enum-all-fun-def all-n-lists-def by auto
 qed
next
 fix P
 show enum-ex (P :: ('a ⇒ 'b) ⇒ bool) = Bex UNIV P
 proof
  assume enum-ex P
  from this show Bex UNIV P
   unfolding enum-ex-fun-def ex-n-lists-def by auto
 next
  assume Bex UNIV P
  from this obtain f where P f ..
  have f: f = the ∘ map-of (zip (enum :: 'a::enum list) (map f enum))
   by (auto simp add: map-of-zip-map fun-eq-iff intro: in-enum)
  from ‹P f› this have P (the ∘ map-of (zip (enum :: 'a::enum list) (map f
enum)))
   by auto
  from  this show enum-ex P
   unfolding enum-ex-fun-def ex-n-lists-def
   apply (auto simp add: set-n-lists)
   apply (rule-tac x=map f enum in exI)
   apply (auto intro!: in-enum)
   done
 qed
qed

end

lemma enum-fun-code [code]: enum = (let enum-a = (enum :: 'a::{enum, equal}
list)
 in map (λys. the ∘ map-of (zip enum-a ys)) (List.n-lists (length enum-a) enum))
 by (simp add: enum-fun-def Let-def)

lemma enum-all-fun-code [code]:
 enum-all P = (let enum-a = (enum :: 'a::{enum, equal} list)
  in all-n-lists (λbs. P (the ∘ map-of (zip enum-a bs))) (length enum-a))
 by (simp only: enum-all-fun-def Let-def)

lemma enum-ex-fun-code [code]:
 enum-ex P = (let enum-a = (enum :: 'a::{enum, equal} list)
  in ex-n-lists (λbs. P (the ∘ map-of (zip enum-a bs))) (length enum-a))
```

by (*simp only*: *enum-ex-fun-def Let-def*)

instantiation *set* :: (*enum*) *enum*
begin

definition
enum = *map set* (*subseqs enum*)

definition
enum-all P ⟷ (∀ *A*∈*set enum*. *P* (*A*::'*a set*))

definition
enum-ex P ⟷ (∃ *A*∈*set enum*. *P* (*A*::'*a set*))

instance proof
qed (*simp-all add*: *enum-set-def enum-all-set-def enum-ex-set-def subseqs-powset distinct-set-subseqs*
enum-distinct enum-UNIV)

end

instantiation *unit* :: *enum*
begin

definition
enum = [()]

definition
enum-all P = *P* ()

definition
enum-ex P = *P* ()

instance proof
qed (*auto simp add*: *enum-unit-def enum-all-unit-def enum-ex-unit-def*)

end

instantiation *bool* :: *enum*
begin

definition
enum = [*False*, *True*]

definition
enum-all P ⟷ *P False* ∧ *P True*

definition
enum-ex P ⟷ *P False* ∨ *P True*

instance proof
qed (*simp-all only*: *enum-bool-def enum-all-bool-def enum-ex-bool-def UNIV-bool*, *simp-all*)

end

instantiation *prod* :: (*enum*, *enum*) *enum*
begin

definition
enum = *List.product enum enum*

definition
enum-all P = *enum-all* (%*x*. *enum-all* (%*y*. *P* (*x*, *y*)))

definition
enum-ex P = *enum-ex* (%*x*. *enum-ex* (%*y*. *P* (*x*, *y*)))

instance
by *standard*
(*simp-all add*: *enum-prod-def distinct-product*
enum-UNIV enum-distinct enum-all-prod-def enum-ex-prod-def)

end

instantiation *sum* :: (*enum*, *enum*) *enum*
begin

definition
enum = *map Inl enum* @ *map Inr enum*

definition
enum-all P ⟷ *enum-all* (λ*x*. *P* (*Inl x*)) ∧ *enum-all* (λ*x*. *P* (*Inr x*))

definition
enum-ex P ⟷ *enum-ex* (λ*x*. *P* (*Inl x*)) ∨ *enum-ex* (λ*x*. *P* (*Inr x*))

instance proof
qed (*simp-all only*: *enum-sum-def enum-all-sum-def enum-ex-sum-def UNIV-sum*,
auto simp add: *enum-UNIV distinct-map enum-distinct*)

end

instantiation *option* :: (*enum*) *enum*
begin

definition

enum = None # map Some enum

definition
enum-all P $\longleftrightarrow$ *P None* $\wedge$ *enum-all* (λx. *P* (*Some x*))

definition
enum-ex P $\longleftrightarrow$ *P None* $\vee$ *enum-ex* (λx. *P* (*Some x*))

instance proof
qed (*simp-all only*: *enum-option-def enum-all-option-def enum-ex-option-def UNIV-option-conv*, *auto simp add*: *distinct-map enum-UNIV enum-distinct*)

end

72.4 Small finite types

We define small finite types for use in Quickcheck

datatype (*plugins only*: *code quickcheck extraction*) *finite-1* =
a_1

notation (**output**) a_1 ($\langle a_1 \rangle$)

lemma *UNIV-finite-1*:
UNIV = $\{a_1\}$
by (*auto intro*: *finite-1.exhaust*)

instantiation *finite-1* :: *enum*
begin

definition
enum = $[a_1]$

definition
enum-all P = *P* a_1

definition
enum-ex P = *P* a_1

instance proof
qed (*simp-all only*: *enum-finite-1-def enum-all-finite-1-def enum-ex-finite-1-def UNIV-finite-1*, *simp-all*)

end

instantiation *finite-1* :: *linorder*
begin

definition *less-finite-1* :: *finite-1* $\Rightarrow$ *finite-1* $\Rightarrow$ *bool*
where

$x < (y :: \mathit{finite\text{-}1}) \longleftrightarrow \mathit{False}$

definition *less-eq-finite-1* :: *finite-1* ⇒ *finite-1* ⇒ *bool*
where
$x \leq (y :: \mathit{finite\text{-}1}) \longleftrightarrow \mathit{True}$

instance
apply (*intro-classes*)
apply (*auto simp add*: *less-finite-1-def less-eq-finite-1-def*)
apply (*metis* (*full-types*) *finite-1.exhaust*)
done

end

instance *finite-1* :: {*dense-linorder*, *wellorder*}
by *intro-classes* (*simp-all add*: *less-finite-1-def*)

instantiation *finite-1* :: *complete-lattice*
begin

definition [*simp*]: *Inf* = (λ-. a_1)
definition [*simp*]: *Sup* = (λ-. a_1)
definition [*simp*]: *bot* = a_1
definition [*simp*]: *top* = a_1
definition [*simp*]: *inf* = (λ- -. a_1)
definition [*simp*]: *sup* = (λ- -. a_1)

instance by *intro-classes*(*simp-all add*: *less-eq-finite-1-def*)
end

instance *finite-1* :: *complete-distrib-lattice*
by *standard simp-all*

instance *finite-1* :: *complete-linorder* **..**

lemma *finite-1-eq*: *x* = a_1
by(*cases x*) *simp*

simproc-setup *finite-1-eq* (*x*::*finite-1*) = ‹
K (*K* (*fn ct* =>
(*case Thm.term-of ct of*
Const (***const-name***‹a_1›, -) => *NONE*
| - => *SOME* (*mk-meta-eq* @{*thm finite-1-eq*}))))
›

instantiation *finite-1* :: *complete-boolean-algebra*
begin
definition [*simp*]: (−) = (λ- -. a_1)
definition [*simp*]: *uminus* = (λ-. a_1)

instance by *intro-classes simp-all*
end

instantiation *finite-1* ::
{*linordered-ring-strict*, *linordered-comm-semiring-strict*, *ordered-comm-ring*, *ordered-cancel-comm-monoid-diff*, *comm-monoid-mult*, *ordered-ring-abs*, *one*, *modulo*, *sgn*, *inverse*}
begin
definition [*simp*]: *Groups.zero* = a_1
definition [*simp*]: *Groups.one* = a_1
definition [*simp*]: $(+) = (\lambda\text{-}\ \text{-}.\ a_1)$
definition [*simp*]: $(*) = (\lambda\text{-}\ \text{-}.\ a_1)$
definition [*simp*]: $(mod) = (\lambda\text{-}\ \text{-}.\ a_1)$
definition [*simp*]: *abs* = $(\lambda\text{-}.\ a_1)$
definition [*simp*]: *sgn* = $(\lambda\text{-}.\ a_1)$
definition [*simp*]: *inverse* = $(\lambda\text{-}.\ a_1)$
definition [*simp*]: *divide* = $(\lambda\text{-}\ \text{-}.\ a_1)$

instance by *intro-classes*(*simp-all add*: *less-finite-1-def*)
end

declare [[*simproc del*: *finite-1-eq*]]
hide-const (**open**) a_1

datatype (*plugins only*: *code quickcheck extraction*) *finite-2* =
a_1 | a_2

notation (**output**) a_1 ($\langle a_1 \rangle$)
notation (**output**) a_2 ($\langle a_2 \rangle$)

lemma *UNIV-finite-2*:
UNIV = $\{a_1, a_2\}$
by (*auto intro*: *finite-2.exhaust*)

instantiation *finite-2* :: *enum*
begin

definition
enum = $[a_1, a_2]$

definition
enum-all $P \longleftrightarrow P\ a_1 \wedge P\ a_2$

definition
enum-ex $P \longleftrightarrow P\ a_1 \vee P\ a_2$

instance proof
qed (*simp-all only*: *enum-finite-2-def enum-all-finite-2-def enum-ex-finite-2-def UNIV-finite-2*, *simp-all*)

end

instantiation *finite-2* :: *linorder*
begin

definition *less-finite-2* :: *finite-2* ⇒ *finite-2* ⇒ *bool*
where
  $x < y \longleftrightarrow x = a_1 \wedge y = a_2$

definition *less-eq-finite-2* :: *finite-2* ⇒ *finite-2* ⇒ *bool*
where
  $x \leq y \longleftrightarrow x = y \vee x < (y :: \textit{finite-2})$

instance
apply (*intro-classes*)
apply (*auto simp add*: *less-finite-2-def less-eq-finite-2-def*)
apply (*metis finite-2.nchotomy*)+
done

end

instance *finite-2* :: *wellorder*
by(*rule wf-wellorderI*)(*simp add*: *less-finite-2-def*, *intro-classes*)

instantiation *finite-2* :: *complete-lattice*
begin

definition $\bigsqcap A = (\textit{if } a_1 \in A \textit{ then } a_1 \textit{ else } a_2)$
definition $\bigsqcup A = (\textit{if } a_2 \in A \textit{ then } a_2 \textit{ else } a_1)$
definition [*simp*]: $\textit{bot} = a_1$
definition [*simp*]: $\textit{top} = a_2$
definition $x \sqcap y = (\textit{if } x = a_1 \vee y = a_1 \textit{ then } a_1 \textit{ else } a_2)$
definition $x \sqcup y = (\textit{if } x = a_2 \vee y = a_2 \textit{ then } a_2 \textit{ else } a_1)$

lemma *neq-finite-2-a*$_1$*-iff* [*simp*]: $x \neq a_1 \longleftrightarrow x = a_2$
by(*cases x*) *simp-all*

lemma *neq-finite-2-a*$_1$*-iff'* [*simp*]: $a_1 \neq x \longleftrightarrow x = a_2$
by(*cases x*) *simp-all*

lemma *neq-finite-2-a*$_2$*-iff* [*simp*]: $x \neq a_2 \longleftrightarrow x = a_1$
by(*cases x*) *simp-all*

lemma *neq-finite-2-a*$_2$*-iff'* [*simp*]: $a_2 \neq x \longleftrightarrow x = a_1$
by(*cases x*) *simp-all*

instance
proof

fix x :: *finite-2* **and** A
assume $x \in A$
then show $\bigsqcap A \leq x\ x \leq \bigsqcup A$
by(*cases x*; *auto simp add*: *less-eq-finite-2-def less-finite-2-def Inf-finite-2-def Sup-finite-2-def*)+
qed(*auto simp add*: *less-eq-finite-2-def less-finite-2-def inf-finite-2-def sup-finite-2-def Inf-finite-2-def Sup-finite-2-def*)
end

instance *finite-2* :: *complete-linorder* **..**

instance *finite-2* :: *complete-distrib-lattice* **..**

instantiation *finite-2* :: {*field*, *idom-abs-sgn*, *idom-modulo*} **begin**
definition [*simp*]: $0 = a_1$
definition [*simp*]: $1 = a_2$
definition $x + y = (case\ (x,\ y)\ of\ (a_1,\ a_1) \Rightarrow a_1 \mid (a_2,\ a_2) \Rightarrow a_1 \mid \text{-} \Rightarrow a_2)$
definition $uminus = (\lambda x :: finite\text{-}2.\ x)$
definition $(-) = ((+) :: finite\text{-}2 \Rightarrow \text{-})$
definition $x * y = (case\ (x,\ y)\ of\ (a_2,\ a_2) \Rightarrow a_2 \mid \text{-} \Rightarrow a_1)$
definition $inverse = (\lambda x :: finite\text{-}2.\ x)$
definition $divide = ((*) :: finite\text{-}2 \Rightarrow \text{-})$
definition $x\ mod\ y = (case\ (x,\ y)\ of\ (a_2,\ a_1) \Rightarrow a_2 \mid \text{-} \Rightarrow a_1)$
definition $abs = (\lambda x :: finite\text{-}2.\ x)$
definition $sgn = (\lambda x :: finite\text{-}2.\ x)$
instance
by *standard*
(*subproofs*
‹*simp-all add*: *plus-finite-2-def uminus-finite-2-def minus-finite-2-def times-finite-2-def inverse-finite-2-def divide-finite-2-def modulo-finite-2-def abs-finite-2-def sgn-finite-2-def split*: *finite-2.splits*›)
end

lemma *two-finite-2* [*simp*]:
$2 = a_1$
by (*simp add*: *numeral.simps plus-finite-2-def*)

lemma *dvd-finite-2-unfold*:
$x\ dvd\ y \longleftrightarrow x = a_2 \vee y = a_1$
by (*auto simp add*: *dvd-def times-finite-2-def split*: *finite-2.splits*)

instantiation *finite-2* :: {*normalization-semidom*, *unique-euclidean-semiring*} **begin**
definition [*simp*]: $normalize = (id :: finite\text{-}2 \Rightarrow \text{-})$
definition [*simp*]: $unit\text{-}factor = (id :: finite\text{-}2 \Rightarrow \text{-})$
definition [*simp*]: $euclidean\text{-}size\ x = (case\ x\ of\ a_1 \Rightarrow 0 \mid a_2 \Rightarrow 1)$
definition [*simp*]: $division\text{-}segment\ (x :: finite\text{-}2) = 1$

instance
by *standard*
(*subproofs*
‹*auto simp add*: *divide-finite-2-def times-finite-2-def dvd-finite-2-unfold*
split: *finite-2.splits*›)
end

hide-const (**open**) a_1 a_2

datatype (*plugins only*: *code quickcheck extraction*) *finite-3* =
a_1 | a_2 | a_3

notation (**output**) a_1 (‹a_1›)
notation (**output**) a_2 (‹a_2›)
notation (**output**) a_3 (‹a_3›)

lemma *UNIV-finite-3*:
$UNIV = \{a_1, a_2, a_3\}$
by (*auto intro*: *finite-3.exhaust*)

instantiation *finite-3* :: *enum*
begin

definition
$enum = [a_1, a_2, a_3]$

definition
$enum\text{-}all\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3$

definition
$enum\text{-}ex\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3$

instance proof
qed (*simp-all only*: *enum-finite-3-def enum-all-finite-3-def enum-ex-finite-3-def UNIV-finite-3*, *simp-all*)

end

lemma *finite-3-not-eq-unfold*:
$x \neq a_1 \longleftrightarrow x \in \{a_2, a_3\}$
$x \neq a_2 \longleftrightarrow x \in \{a_1, a_3\}$
$x \neq a_3 \longleftrightarrow x \in \{a_1, a_2\}$
by (*cases x*; *simp*)+

instantiation *finite-3* :: *linorder*
begin

definition *less-finite-3* :: *finite-3* ⇒ *finite-3* ⇒ *bool*

where
  $x < y = (case\ x\ of\ a_1 \Rightarrow y \neq a_1 \mid a_2 \Rightarrow y = a_3 \mid a_3 \Rightarrow False)$

definition *less-eq-finite-3* :: *finite-3* ⇒ *finite-3* ⇒ *bool*
where
  $x \leq y \longleftrightarrow x = y \vee x < (y :: finite\text{-}3)$

instance proof (*intro-classes*)
qed (*auto simp add*: *less-finite-3-def less-eq-finite-3-def split*: *finite-3.split-asm*)

end

instance *finite-3* :: *wellorder*
proof(*rule wf-wellorderI*)
  have *inv-image less-than* (*case-finite-3 0 1 2*) = {(*x*, *y*). *x* < *y*}
    by(*auto simp add*: *less-finite-3-def split*: *finite-3.splits*)
  from *this*[*symmetric*] **show** *wf* ... **by** *simp*
qed *intro-classes*

class *finite-lattice* = *finite* + *lattice* + *Inf* + *Sup* + *bot* + *top* +
  assumes *Inf-finite-empty*: *Inf* {} = *Sup UNIV*
  assumes *Inf-finite-insert*: *Inf* (*insert a A*) = *a* ⊓ *Inf A*
  assumes *Sup-finite-empty*: *Sup* {} = *Inf UNIV*
  assumes *Sup-finite-insert*: *Sup* (*insert a A*) = *a* ⊔ *Sup A*
  assumes *bot-finite-def*: *bot* = *Inf UNIV*
  assumes *top-finite-def*: *top* = *Sup UNIV*
begin

subclass *complete-lattice*
proof
  fix *x A*
  show $x \in A \Longrightarrow \bigsqcap A \leq x$
   by (*metis Set.set-insert abel-semigroup.commute local.Inf-finite-insert local.inf.abel-semigroup-axioms local.inf.left-idem local.inf.orderI*)
  show $x \in A \Longrightarrow x \leq \bigsqcup A$
   by (*metis Set.set-insert insert-absorb2 local.Sup-finite-insert local.sup.absorb-iff2*)
next
  fix *A z*
  have $\bigsqcup\ UNIV = z \sqcup \bigsqcup UNIV$
    by (*subst Sup-finite-insert* [*symmetric*], *simp add*: *insert-UNIV*)
  from *this* **have** [*simp*]: $z \leq \bigsqcup UNIV$
    using *local.le-iff-sup* **by** *auto*
  have $(\forall\ x.\ x \in A \longrightarrow z \leq x) \longrightarrow z \leq \bigsqcap A$
    by (*rule finite-induct* [*of A* λ *A* . $(\forall\ x.\ x \in A \longrightarrow z \leq x) \longrightarrow z \leq \bigsqcap A$])
      (*simp-all add*: *Inf-finite-empty Inf-finite-insert*)
  from *this* **show** $(\bigwedge x.\ x \in A \Longrightarrow z \leq x) \Longrightarrow z \leq \bigsqcap A$
    by *simp*

  have $\bigsqcap\ UNIV = z \sqcap \bigsqcap UNIV$

by (*subst Inf-finite-insert* [*symmetric*], *simp add*: *insert-UNIV*)
from *this* **have** [*simp*]: $\bigsqcap UNIV \leq z$
by (*simp add*: *local.inf.absorb-iff2*)
have $(\forall\ x.\ x \in A \longrightarrow x \leq z) \longrightarrow \bigsqcup A \leq z$
by (*rule finite-induct* [*of* $A\ \lambda\ A\ .\ (\forall\ x.\ x \in A \longrightarrow x \leq z) \longrightarrow \bigsqcup A \leq z$], *simp-all add*: *Sup-finite-empty Sup-finite-insert*)
from *this* **show** $(\bigwedge x.\ x \in A \Longrightarrow x \leq z) \Longrightarrow \bigsqcup A \leq z$
by *blast*
next
show $\bigsqcap \{\} = \top$
by (*simp add*: *Inf-finite-empty top-finite-def*)
show $\bigsqcup \{\} = \bot$
by (*simp add*: *Sup-finite-empty bot-finite-def*)
qed
end

class *finite-distrib-lattice* = *finite-lattice* + *distrib-lattice*
begin
lemma *finite-inf-Sup*: $a \sqcap (Sup\ A) = Sup\ \{a \sqcap b \mid b\ .\ b \in A\}$
proof (*rule finite-induct* [*of* $A\ \lambda\ A\ .\ a \sqcap (Sup\ A) = Sup\ \{a \sqcap b \mid b\ .\ b \in A\}$], *simp-all*)
fix $x::'a$
fix F
assume $x \notin F$
assume [*simp*]: $a \sqcap \bigsqcup F = \bigsqcup \{a \sqcap b \mid b.\ b \in F\}$
have [*simp*]: $insert\ (a \sqcap x)\ \{a \sqcap b \mid b.\ b \in F\} = \{a \sqcap b \mid b.\ b = x \vee b \in F\}$
by *blast*
have $a \sqcap (x \sqcup \bigsqcup F) = a \sqcap x \sqcup a \sqcap \bigsqcup F$
by (*simp add*: *inf-sup-distrib1*)
also have $\ldots = a \sqcap x \sqcup \bigsqcup \{a \sqcap b \mid b.\ b \in F\}$
by *simp*
also have $\ldots = \bigsqcup \{a \sqcap b \mid b.\ b = x \vee b \in F\}$
by (*unfold Sup-insert*[*THEN sym*], *simp*)
finally show $a \sqcap (x \sqcup \bigsqcup F) = \bigsqcup \{a \sqcap b \mid b.\ b = x \vee b \in F\}$
by *simp*
qed

lemma *finite-Inf-Sup*: $\bigsqcap (Sup\ `\ A) \leq \bigsqcup (Inf\ `\ \{f\ `\ A \mid f.\ \forall\ Y \in A.\ f\ Y \in\ Y\})$
proof (*rule finite-induct* [*of* $A\ \lambda A.\ \bigsqcap (Sup\ `\ A) \leq \bigsqcup (Inf\ `\ \{f\ `\ A \mid f.\ \forall\ Y \in A.\ f\ Y \in\ Y\})$], *simp-all add*: *finite-UnionD*)
fix $x::'a\ set$
fix F
assume $x \notin F$
have [*simp*]: $\{\bigsqcup x \sqcap b \mid b\ .\ b \in Inf\ `\ \{f\ `\ F \mid f.\ \forall\ Y \in F.\ f\ Y \in\ Y\}\ \} = \{\bigsqcup x \sqcap (Inf\ (f\ `\ F)) \mid f\ .\ (\forall\ Y \in F.\ f\ Y \in\ Y)\}$
by *auto*
define *fa* **where** $fa = (\lambda\ (b::'a)\ f\ Y\ .\ (if\ Y = x\ then\ b\ else\ f\ Y))$
have $\bigwedge f\ b.\ \forall\ Y \in F.\ f\ Y \in\ Y \Longrightarrow b \in x \Longrightarrow insert\ b\ (f\ `\ (F \cap \{Y.\ Y \neq x\})) = insert\ (fa\ b\ f\ x)\ (fa\ b\ f\ `\ F) \wedge fa\ b\ f\ x \in x \wedge (\forall\ Y \in F.\ fa\ b\ f\ Y \in\ Y)$

by (*auto simp add*: *fa-def*)
from *this* **have** *B*: $\bigwedge f\ b.\ \forall\, Y{\in}F.\ f\ Y\ \in\ Y \Longrightarrow b \in x \Longrightarrow fa\ b\ f\ `\ (\{x\} \cup F) \in \{insert\ (f\ x)\ (f\ `\ F)\ |f.\ f\ x \in x \wedge (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\}$
by *blast*
have [*simp*]: $\bigwedge f\ b.\ \forall\, Y{\in}F.\ f\ Y\ \in\ Y \Longrightarrow b \in x \Longrightarrow b \sqcap (\prod x{\in}F.\ f\ x)\ \leq \bigsqcup (Inf\ `\ \{insert\ (f\ x)\ (f\ `\ F)\ |f.\ f\ x \in x \wedge (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\})$
using *B* **apply** (*rule SUP-upper2*)
using ‹$x \notin F$› **apply** (*simp-all add*: *fa-def Inf-union-distrib*)
apply (*simp add*: *image-mono Inf-superset-mono inf.coboundedI2*)
done
assume $\prod (Sup\ `\ F) \leq \bigsqcup (Inf\ `\ \{f\ `\ F\ |f.\ \forall\, Y{\in}F.\ f\ Y\ \in\ Y\})$

from *this* **have** $\bigsqcup x \sqcap \prod (Sup\ `\ F) \leq \bigsqcup x \sqcap \bigsqcup (Inf\ `\ \{f\ `\ F\ |f.\ \forall\, Y{\in}F.\ f\ Y\ \in\ Y\})$
using *inf.coboundedI2* **by** *auto*
also have ... = $Sup\ \{\bigsqcup x \sqcap (Inf\ (f\ `\ F))\ |f\ .\ (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\}$
by (*simp add*: *finite-inf-Sup*)

also have ... = $Sup\ \{Sup\ \{Inf\ (f\ `\ F) \sqcap b \mid b\ .\ b \in x\}\ |f\ .\ (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\}$
by (*subst inf-commute*) (*simp add*: *finite-inf-Sup*)

also have ... $\leq \bigsqcup (Inf\ `\ \{insert\ (f\ x)\ (f\ `\ F)\ |f.\ f\ x \in x \wedge (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\})$
apply (*rule Sup-least*, *clarsimp*)+
apply (*subst inf-commute*, *simp*)
done

finally show $\bigsqcup x \sqcap \prod (Sup\ `\ F) \leq \bigsqcup (Inf\ `\ \{insert\ (f\ x)\ (f\ `\ F)\ |f.\ f\ x \in x \wedge (\forall\, Y{\in}F.\ f\ Y\ \in\ Y)\})$
by *simp*
qed

subclass *complete-distrib-lattice*
by (*standard*, *rule finite-Inf-Sup*)
end

instantiation *finite-3* :: *finite-lattice*
begin

definition $\prod A = (if\ a_1 \in A\ then\ a_1\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_3)$
definition $\bigsqcup A = (if\ a_3 \in A\ then\ a_3\ else\ if\ a_2 \in A\ then\ a_2\ else\ a_1)$
definition [*simp*]: $bot = a_1$
definition [*simp*]: $top = a_3$
definition [*simp*]: $inf = (min :: finite\text{-}3 \Rightarrow \text{-})$
definition [*simp*]: $sup = (max :: finite\text{-}3 \Rightarrow \text{-})$

instance
proof
qed (*auto simp add*: *Inf-finite-3-def Sup-finite-3-def max-def min-def less-eq-finite-3-def less-finite-3-def split*: *finite-3.split*)
end

instance *finite-3* :: *complete-lattice* **..**

instance *finite-3* :: *finite-distrib-lattice*
proof
qed (*auto simp add*: *min-def max-def*)

instance *finite-3* :: *complete-distrib-lattice* **..**

instance *finite-3* :: *complete-linorder* **..**

instantiation *finite-3* :: {*field*, *idom-abs-sgn*, *idom-modulo*} **begin**
definition [*simp*]: *0* = a_1
definition [*simp*]: *1* = a_2
definition
  $x + y$ = (*case* (x, y) *of*
    (a_1, a_1) ⇒ a_1 | (a_2, a_3) ⇒ a_1 | (a_3, a_2) ⇒ a_1
  | (a_1, a_2) ⇒ a_2 | (a_2, a_1) ⇒ a_2 | (a_3, a_3) ⇒ a_2
  | - ⇒ a_3)
definition $-x$ = (*case* x *of* a_1 ⇒ a_1 | a_2 ⇒ a_3 | a_3 ⇒ a_2)
definition $x - y = x + (-y$ :: *finite-3*)
definition $x * y$ = (*case* (x, y) *of* (a_2, a_2) ⇒ a_2 | (a_3, a_3) ⇒ a_2 | (a_2, a_3) ⇒ a_3
| (a_3, a_2) ⇒ a_3 | - ⇒ a_1)
definition *inverse* = (λx :: *finite-3*. x)
definition x *div* $y = x *$ *inverse* (y :: *finite-3*)
definition x *mod* y = (*case* y *of* a_1 ⇒ x | - ⇒ a_1)
definition *abs* = (λx. *case* x *of* a_3 ⇒ a_2 | - ⇒ x)
definition *sgn* = (λx :: *finite-3*. x)
instance
  by *standard*
    (*subproofs*
      ‹*simp-all add*: *plus-finite-3-def uminus-finite-3-def minus-finite-3-def*
        times-finite-3-def
        inverse-finite-3-def divide-finite-3-def modulo-finite-3-def
        abs-finite-3-def sgn-finite-3-def
        less-finite-3-def
        split: *finite-3.splits*›)
end

lemma *two-finite-3* [*simp*]:
  2 = a_3
  by (*simp add*: *numeral.simps plus-finite-3-def*)

lemma *dvd-finite-3-unfold*:
  x *dvd* $y \longleftrightarrow x = a_2 \vee x = a_3 \vee y = a_1$
  by (*cases* x) (*auto simp add*: *dvd-def times-finite-3-def split*: *finite-3.splits*)

instantiation *finite-3* :: {*normalization-semidom*, *unique-euclidean-semiring*} **begin**

definition [*simp*]: *normalize x* = (*case x of* $a_3 \Rightarrow a_2$ | - ⇒ *x*)
definition [*simp*]: *unit-factor* = (*id* :: *finite-3* ⇒ -)
definition [*simp*]: *euclidean-size x* = (*case x of* $a_1 \Rightarrow 0$ | - ⇒ *1*)
definition [*simp*]: *division-segment* (*x* :: *finite-3*) = *1*
instance
proof
  fix *x* :: *finite-3*
  assume *x* ≠ *0*
  then show *is-unit* (*unit-factor x*)
    by (*cases x*) (*simp-all add*: *dvd-finite-3-unfold*)
qed
  (*subproofs*
    ‹*auto simp add*: *divide-finite-3-def times-finite-3-def*
      dvd-finite-3-unfold inverse-finite-3-def plus-finite-3-def
      split: *finite-3.splits*›)
end

hide-const (**open**) a_1 a_2 a_3

datatype (*plugins only*: *code quickcheck extraction*) *finite-4* =
  a_1 | a_2 | a_3 | a_4

notation (**output**) a_1 (‹a_1›)
notation (**output**) a_2 (‹a_2›)
notation (**output**) a_3 (‹a_3›)
notation (**output**) a_4 (‹a_4›)

lemma *UNIV-finite-4*:
  UNIV = {a_1, a_2, a_3, a_4}
  by (*auto intro*: *finite-4.exhaust*)

instantiation *finite-4* :: *enum*
begin

definition
  enum = [a_1, a_2, a_3, a_4]

definition
  enum-all P ⟷ *P* a_1 ∧ *P* a_2 ∧ *P* a_3 ∧ *P* a_4

definition
  enum-ex P ⟷ *P* a_1 ∨ *P* a_2 ∨ *P* a_3 ∨ *P* a_4

instance proof
qed (*simp-all only*: *enum-finite-4-def enum-all-finite-4-def enum-ex-finite-4-def UNIV-finite-4*, *simp-all*)

end

instantiation *finite-4* :: *finite-distrib-lattice* **begin**

$a_1 < a_2$,$a_3 < a_4$, but a_2 and a_3 are incomparable.

definition
$x < y \longleftrightarrow (case\ (x,\ y)\ of$
$(a_1,\ a_1) \Rightarrow False \mid (a_1,\ \text{-}) \Rightarrow True$
$\mid\ (a_2,\ a_4) \Rightarrow True$
$\mid\ (a_3,\ a_4) \Rightarrow True\ \mid \text{-} \Rightarrow False)$

definition
$x \leq y \longleftrightarrow (case\ (x,\ y)\ of$
$(a_1,\ \text{-}) \Rightarrow True$
$\mid (a_2,\ a_2) \Rightarrow True \mid (a_2,\ a_4) \Rightarrow True$
$\mid (a_3,\ a_3) \Rightarrow True \mid (a_3,\ a_4) \Rightarrow True$
$\mid (a_4,\ a_4) \Rightarrow True \mid \text{-} \Rightarrow False)$

definition
$\bigsqcap A = (if\ a_1 \in A \vee a_2 \in A \wedge a_3 \in A\ then\ a_1\ else\ if\ a_2 \in A\ then\ a_2\ else\ if\ a_3 \in A\ then\ a_3\ else\ a_4)$
definition
$\bigsqcup A = (if\ a_4 \in A \vee a_2 \in A \wedge a_3 \in A\ then\ a_4\ else\ if\ a_2 \in A\ then\ a_2\ else\ if\ a_3 \in A\ then\ a_3\ else\ a_1)$
definition [*simp*]: $bot = a_1$
definition [*simp*]: $top = a_4$
definition
$x \sqcap y = (case\ (x,\ y)\ of$
$(a_1,\ \text{-}) \Rightarrow a_1 \mid (\text{-},\ a_1) \Rightarrow a_1 \mid (a_2,\ a_3) \Rightarrow a_1 \mid (a_3,\ a_2) \Rightarrow a_1$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid \text{-} \Rightarrow a_4)$
definition
$x \sqcup y = (case\ (x,\ y)\ of$
$(a_4,\ \text{-}) \Rightarrow a_4 \mid (\text{-},\ a_4) \Rightarrow a_4 \mid (a_2,\ a_3) \Rightarrow a_4 \mid (a_3,\ a_2) \Rightarrow a_4$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid \text{-} \Rightarrow a_1)$

instance
by *standard*
(*subproofs*
‹*auto simp add*: *less-finite-4-def less-eq-finite-4-def Inf-finite-4-def Sup-finite-4-def*

inf-finite-4-def sup-finite-4-def split: *finite-4.splits*›)
end

instance *finite-4* :: *complete-lattice* **..**

instance *finite-4* :: *complete-distrib-lattice* **..**

instantiation *finite-4* :: *complete-boolean-algebra* **begin**
definition $-\ x = (\mathit{case}\ x\ \mathit{of}\ a_1 \Rightarrow a_4 \mid a_2 \Rightarrow a_3 \mid a_3 \Rightarrow a_2 \mid a_4 \Rightarrow a_1)$
definition $x - y = x \sqcap -\ (y :: \mathit{finite\text{-}4})$
instance
by *standard*
(*subproofs*
‹*simp-all add*: *inf-finite-4-def sup-finite-4-def uminus-finite-4-def minus-finite-4-def*

split: *finite-4.splits*›)
end

hide-const (**open**) $a_1\ a_2\ a_3\ a_4$

datatype (*plugins only*: *code quickcheck extraction*) *finite-5* =
$a_1 \mid a_2 \mid a_3 \mid a_4 \mid a_5$

notation (**output**) a_1 (‹a_1›)
notation (**output**) a_2 (‹a_2›)
notation (**output**) a_3 (‹a_3›)
notation (**output**) a_4 (‹a_4›)
notation (**output**) a_5 (‹a_5›)

lemma *UNIV-finite-5*:
$\mathit{UNIV} = \{a_1, a_2, a_3, a_4, a_5\}$
by (*auto intro*: *finite-5.exhaust*)

instantiation *finite-5* :: *enum*
begin

definition
$\mathit{enum} = [a_1, a_2, a_3, a_4, a_5]$

definition
$\mathit{enum\text{-}all}\ P \longleftrightarrow P\ a_1 \wedge P\ a_2 \wedge P\ a_3 \wedge P\ a_4 \wedge P\ a_5$

definition
$\mathit{enum\text{-}ex}\ P \longleftrightarrow P\ a_1 \vee P\ a_2 \vee P\ a_3 \vee P\ a_4 \vee P\ a_5$

instance proof
qed (*simp-all only*: *enum-finite-5-def enum-all-finite-5-def enum-ex-finite-5-def UNIV-finite-5*, *simp-all*)

end

instantiation *finite-5* :: *finite-lattice*
begin

The non-distributive pentagon lattice N_5

definition

$x < y \longleftrightarrow (\mathit{case}\ (x,\ y)\ \mathit{of}$
$(a_1,\ a_1) \Rightarrow \mathit{False} \mid (a_1,\ \text{-}) \Rightarrow \mathit{True}$
$\mid (a_2,\ a_3) \Rightarrow \mathit{True} \mid (a_2,\ a_5) \Rightarrow \mathit{True}$
$\mid (a_3,\ a_5) \Rightarrow \mathit{True}$
$\mid (a_4,\ a_5) \Rightarrow \mathit{True} \mid \text{-} \Rightarrow \mathit{False})$

definition
$x \leq y \longleftrightarrow (\mathit{case}\ (x,\ y)\ \mathit{of}$
$(a_1,\ \text{-}) \Rightarrow \mathit{True}$
$\mid (a_2,\ a_2) \Rightarrow \mathit{True} \mid (a_2,\ a_3) \Rightarrow \mathit{True} \mid (a_2,\ a_5) \Rightarrow \mathit{True}$
$\mid (a_3,\ a_3) \Rightarrow \mathit{True} \mid (a_3,\ a_5) \Rightarrow \mathit{True}$
$\mid (a_4,\ a_4) \Rightarrow \mathit{True} \mid (a_4,\ a_5) \Rightarrow \mathit{True}$
$\mid (a_5,\ a_5) \Rightarrow \mathit{True} \mid \text{-} \Rightarrow \mathit{False})$

definition
$\bigsqcap A =$
$(\mathit{if}\ a_1 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A)\ \mathit{then}\ a_1$
$\mathit{else\ if}\ a_2 \in A\ \mathit{then}\ a_2$
$\mathit{else\ if}\ a_3 \in A\ \mathit{then}\ a_3$
$\mathit{else\ if}\ a_4 \in A\ \mathit{then}\ a_4$
$\mathit{else}\ a_5)$
definition
$\bigsqcup A =$
$(\mathit{if}\ a_5 \in A \vee a_4 \in A \wedge (a_2 \in A \vee a_3 \in A)\ \mathit{then}\ a_5$
$\mathit{else\ if}\ a_3 \in A\ \mathit{then}\ a_3$
$\mathit{else\ if}\ a_2 \in A\ \mathit{then}\ a_2$
$\mathit{else\ if}\ a_4 \in A\ \mathit{then}\ a_4$
$\mathit{else}\ a_1)$
definition [*simp*]: $\mathit{bot} = a_1$
definition [*simp*]: $\mathit{top} = a_5$
definition
$x \sqcap y = (\mathit{case}\ (x,\ y)\ \mathit{of}$
$(a_1,\ \text{-}) \Rightarrow a_1 \mid (\text{-},\ a_1) \Rightarrow a_1 \mid (a_2,\ a_4) \Rightarrow a_1 \mid (a_4,\ a_2) \Rightarrow a_1 \mid (a_3,\ a_4) \Rightarrow a_1 \mid (a_4,\ a_3) \Rightarrow a_1$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid (a_4,\ \text{-}) \Rightarrow a_4 \mid (\text{-},\ a_4) \Rightarrow a_4$
$\mid \text{-} \Rightarrow a_5)$
definition
$x \sqcup y = (\mathit{case}\ (x,\ y)\ \mathit{of}$
$(a_5,\ \text{-}) \Rightarrow a_5 \mid (\text{-},\ a_5) \Rightarrow a_5 \mid (a_2,\ a_4) \Rightarrow a_5 \mid (a_4,\ a_2) \Rightarrow a_5 \mid (a_3,\ a_4) \Rightarrow a_5 \mid (a_4,\ a_3) \Rightarrow a_5$
$\mid (a_3,\ \text{-}) \Rightarrow a_3 \mid (\text{-},\ a_3) \Rightarrow a_3$
$\mid (a_2,\ \text{-}) \Rightarrow a_2 \mid (\text{-},\ a_2) \Rightarrow a_2$
$\mid (a_4,\ \text{-}) \Rightarrow a_4 \mid (\text{-},\ a_4) \Rightarrow a_4$
$\mid \text{-} \Rightarrow a_1)$

instance
by *standard*

(*subproofs*
‹*auto simp add*: *less-eq-finite-5-def less-finite-5-def inf-finite-5-def sup-finite-5-def*

Inf-finite-5-def Sup-finite-5-def split: *finite-5.splits if-split-asm*›)
end

instance *finite-5* :: *complete-lattice* **..**

hide-const (**open**) a_1 a_2 a_3 a_4 a_5

72.5 Closing up

hide-type (**open**) *finite-1 finite-2 finite-3 finite-4 finite-5*
hide-const (**open**) *enum enum-all enum-ex all-n-lists ex-n-lists ntrancl*

end

73 Character and string types

theory *String*
imports *Enum Bit-Operations Code-Numeral*
begin

73.1 Strings as list of bytes

When modelling strings, we follow the approach given in https://utf8everywhere.org/:

- Strings are a list of bytes (8 bit).
- Byte values from 0 to 127 are US-ASCII.
- Byte values from 128 to 255 are uninterpreted blobs.

73.1.1 Bytes as datatype

datatype *char* =
Char (*digit0*: *bool*) (*digit1*: *bool*) (*digit2*: *bool*) (*digit3*: *bool*)
(*digit4*: *bool*) (*digit5*: *bool*) (*digit6*: *bool*) (*digit7*: *bool*)

context *comm-semiring-1*
begin

definition *of-char* :: ‹*char* ⇒ *'a*›
where ‹*of-char c* = *horner-sum of-bool 2* [*digit0 c*, *digit1 c*, *digit2 c*, *digit3 c*, *digit4 c*, *digit5 c*, *digit6 c*, *digit7 c*]›

lemma *of-char-Char* [*simp*]:
  ‹*of-char* (*Char b0 b1 b2 b3 b4 b5 b6 b7*) =
    horner-sum of-bool 2 [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*]›
  by (*simp add*: *of-char-def*)

end

lemma (**in** *comm-semiring-1*) *of-nat-of-char*:
  ‹*of-nat* (*of-char c*) = *of-char c*›
  by (*cases c*) *simp*

lemma (**in** *comm-ring-1*) *of-int-of-char*:
  ‹*of-int* (*of-char c*) = *of-char c*›
  by (*cases c*) *simp*

lemma *nat-of-char* [*simp*]:
  ‹*nat* (*of-char c*) = *of-char c*›
  by (*cases c*) (*simp only*: *of-char-Char nat-horner-sum*)

context *linordered-euclidean-semiring-bit-operations*
begin

definition *char-of* :: ‹'*a* ⇒ *char*›
  where ‹*char-of n* = *Char* (*bit n 0*) (*bit n 1*) (*bit n 2*) (*bit n 3*) (*bit n 4*) (*bit n 5*) (*bit n 6*) (*bit n 7*)›

lemma *char-of-take-bit-eq*:
  ‹*char-of* (*take-bit n m*) = *char-of m*› **if** ‹*n* ≥ *8*›
  using *that* **by** (*simp add*: *char-of-def bit-take-bit-iff*)

lemma *char-of-char* [*simp*]:
  ‹*char-of* (*of-char c*) = *c*›
  by (*simp only*: *of-char-def char-of-def bit-horner-sum-bit-iff*) *simp*

lemma *char-of-comp-of-char* [*simp*]:
  char-of ∘ *of-char* = *id*
  by (*simp add*: *fun-eq-iff*)

lemma *inj-of-char*:
  ‹*inj of-char*›
proof (*rule injI*)
  fix *c d*
  assume *of-char c* = *of-char d*
  then have *char-of* (*of-char c*) = *char-of* (*of-char d*)
    by *simp*
  then show *c* = *d*
    by *simp*

qed

lemma *of-char-eqI*:
‹*c* = *d*› **if** ‹*of-char c* = *of-char d*›
using *that inj-of-char* **by** (*simp add*: *inj-eq*)

lemma *of-char-eq-iff* [*simp*]:
‹*of-char c* = *of-char d* ⟷ *c* = *d*›
by (*auto intro*: *of-char-eqI*)

lemma *of-char-of* [*simp*]:
‹*of-char* (*char-of a*) = *a mod 256*›
proof −
have ‹[*0*..<*8*] = [*0*, *Suc 0*, *2*, *3*, *4*, *5*, *6*, *7* :: *nat*]›
by (*simp add*: *upt-eq-Cons-conv*)
then have ‹[*bit a 0*, *bit a 1*, *bit a 2*, *bit a 3*, *bit a 4*, *bit a 5*, *bit a 6*, *bit a 7*] = *map* (*bit a*) [*0*..<*8*]›
by *simp*
then have ‹*of-char* (*char-of a*) = *take-bit 8 a*›
by (*simp only*: *char-of-def of-char-def char.sel horner-sum-bit-eq-take-bit*)
then show *?thesis*
by (*simp add*: *take-bit-eq-mod*)
qed

lemma *char-of-mod-256* [*simp*]:
‹*char-of* (*n mod 256*) = *char-of n*›
by (*rule of-char-eqI*) *simp*

lemma *of-char-mod-256* [*simp*]:
‹*of-char c mod 256* = *of-char c*›
proof −
have ‹*of-char* (*char-of* (*of-char c*)) *mod 256* = *of-char* (*char-of* (*of-char c*))›
by (*simp only*: *of-char-of*) *simp*
then show *?thesis*
by *simp*
qed

lemma *char-of-quasi-inj* [*simp*]:
‹*char-of m* = *char-of n* ⟷ *m mod 256* = *n mod 256*› (**is** ‹*?P* ⟷ *?Q*›)
proof
assume *?Q*
then show *?P*
by (*auto intro*: *of-char-eqI*)
next
assume *?P*
then have ‹*of-char* (*char-of m*) = *of-char* (*char-of n*)›
by *simp*
then show *?Q*
by *simp*

qed

lemma *char-of-eq-iff*:
‹*char-of n = c ⟷ take-bit 8 n = of-char c*›
by (*auto intro*: *of-char-eqI simp add*: *take-bit-eq-mod*)

lemma *char-of-nat* [*simp*]:
‹*char-of* (*of-nat n*) = *char-of n*›
by (*simp add*: *char-of-def String.char-of-def drop-bit-of-nat bit-simps possible-bit-def*)

end

lemma *inj-on-char-of-nat* [*simp*]:
inj-on char-of {*0*::*nat*..<*256*}
by (*rule inj-onI*) *simp*

lemma *nat-of-char-less-256* [*simp*]:
of-char c < (*256* :: *nat*)
proof −
have *of-char c mod* (*256* :: *nat*) < *256*
by *arith*
then show *?thesis* **by** *simp*
qed

lemma *range-nat-of-char*:
range of-char = {*0*::*nat*..<*256*}
proof (*rule*; *rule*)
fix *n* :: *nat*
assume *n* ∈ *range of-char*
then show *n* ∈ {*0*..<*256*}
by *auto*
next
fix *n* :: *nat*
assume *n* ∈ {*0*..<*256*}
then have *n* = *of-char* (*char-of n*)
by *simp*
then show *n* ∈ *range of-char*
by (*rule range-eqI*)
qed

lemma *UNIV-char-of-nat*:
UNIV = *char-of* ‘ {*0*::*nat*..<*256*}
proof −
have *range* (*of-char* :: *char* ⇒ *nat*) = *of-char* ‘ *char-of* ‘ {*0*::*nat*..<*256*}
by (*simp add*: *image-image range-nat-of-char*)
with *inj-of-char* [**where** *?'a* = *nat*] **show** *?thesis*
by (*simp add*: *inj-image-eq-iff*)
qed

lemma *card-UNIV-char*:
card (*UNIV* :: *char set*) = *256*
by (*auto simp add*: *UNIV-char-of-nat card-image*)

context
includes *lifting-syntax* **and** *integer.lifting* **and** *natural.lifting*
begin

lemma [*transfer-rule*]:
‹(*pcr-integer* ===> (=)) *char-of char-of*›
by (*unfold char-of-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-integer*) *of-char of-char*›
by (*unfold of-char-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹(*pcr-natural* ===> (=)) *char-of char-of*›
by (*unfold char-of-def*) *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-natural*) *of-char of-char*›
by (*unfold of-char-def*) *transfer-prover*

end

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

lifting-update *natural.lifting*
lifting-forget *natural.lifting*

lemma *size-char-eq-0* [*simp*, *code*]:
‹*size c* = *0*› **for** *c* :: *char*
by (*cases c*) *simp*

lemma *size′-char-eq-0* [*simp*, *code*]:
‹*size-char c* = *0*›
by (*cases c*) *simp*

syntax
-Char :: *str-position* ⇒ *char* (‹(‹*open-block notation*=‹*literal char*›› *CHR* -)›)
-Char-ord :: *num-const* ⇒ *char* (‹(‹*open-block notation*=‹*literal char code*›› *CHR* -)›)
syntax-consts
-Char -Char-ord ⇌ *Char*

type-synonym *string* = *char list*

syntax
-String :: *str-position* ⇒ *string* (‹(‹*open-block notation*=‹*literal string*››-)›)

ML-file ‹*Tools/string-syntax.ML*›

instantiation *char* :: *enum*
begin

definition
Enum.enum = [
CHR 0x00, *CHR 0x01*, *CHR 0x02*, *CHR 0x03*,
CHR 0x04, *CHR 0x05*, *CHR 0x06*, *CHR 0x07*,
CHR 0x08, *CHR 0x09*, *CHR ''⏎''*, *CHR 0x0B*,
CHR 0x0C, *CHR 0x0D*, *CHR 0x0E*, *CHR 0x0F*,
CHR 0x10, *CHR 0x11*, *CHR 0x12*, *CHR 0x13*,
CHR 0x14, *CHR 0x15*, *CHR 0x16*, *CHR 0x17*,
CHR 0x18, *CHR 0x19*, *CHR 0x1A*, *CHR 0x1B*,
CHR 0x1C, *CHR 0x1D*, *CHR 0x1E*, *CHR 0x1F*,
CHR '' '', *CHR ''!''*, *CHR 0x22*, *CHR ''#''*,
CHR ''$'', *CHR ''%''*, *CHR ''&''*, *CHR 0x27*,
CHR ''('', *CHR '')''*, *CHR ''*''*, *CHR ''+''*,
CHR '','', *CHR ''−''*, *CHR ''.''*, *CHR ''/''*,
CHR ''0'', *CHR ''1''*, *CHR ''2''*, *CHR ''3''*,
CHR ''4'', *CHR ''5''*, *CHR ''6''*, *CHR ''7''*,
CHR ''8'', *CHR ''9''*, *CHR '':''*, *CHR '';''*,
CHR ''<'', *CHR ''=''*, *CHR ''>''*, *CHR ''?''*,
CHR ''@'', *CHR ''A''*, *CHR ''B''*, *CHR ''C''*,
CHR ''D'', *CHR ''E''*, *CHR ''F''*, *CHR ''G''*,
CHR ''H'', *CHR ''I''*, *CHR ''J''*, *CHR ''K''*,
CHR ''L'', *CHR ''M''*, *CHR ''N''*, *CHR ''O''*,
CHR ''P'', *CHR ''Q''*, *CHR ''R''*, *CHR ''S''*,
CHR ''T'', *CHR ''U''*, *CHR ''V''*, *CHR ''W''*,
CHR ''X'', *CHR ''Y''*, *CHR ''Z''*, *CHR ''[''*,
CHR 0x5C, *CHR '']''*, *CHR ''^''*, *CHR ''-''*,
CHR 0x60, *CHR ''a''*, *CHR ''b''*, *CHR ''c''*,
CHR ''d'', *CHR ''e''*, *CHR ''f''*, *CHR ''g''*,
CHR ''h'', *CHR ''i''*, *CHR ''j''*, *CHR ''k''*,
CHR ''l'', *CHR ''m''*, *CHR ''n''*, *CHR ''o''*,
CHR ''p'', *CHR ''q''*, *CHR ''r''*, *CHR ''s''*,
CHR ''t'', *CHR ''u''*, *CHR ''v''*, *CHR ''w''*,
CHR ''x'', *CHR ''y''*, *CHR ''z''*, *CHR ''{''*,
CHR ''|'', *CHR ''}''*, *CHR ''~''*, *CHR 0x7F*,
CHR 0x80, *CHR 0x81*, *CHR 0x82*, *CHR 0x83*,
CHR 0x84, *CHR 0x85*, *CHR 0x86*, *CHR 0x87*,
CHR 0x88, *CHR 0x89*, *CHR 0x8A*, *CHR 0x8B*,
CHR 0x8C, *CHR 0x8D*, *CHR 0x8E*, *CHR 0x8F*,
CHR 0x90, *CHR 0x91*, *CHR 0x92*, *CHR 0x93*,
CHR 0x94, *CHR 0x95*, *CHR 0x96*, *CHR 0x97*,
CHR 0x98, *CHR 0x99*, *CHR 0x9A*, *CHR 0x9B*,

CHR 0x9C, *CHR 0x9D*, *CHR 0x9E*, *CHR 0x9F*,
CHR 0xA0, *CHR 0xA1*, *CHR 0xA2*, *CHR 0xA3*,
CHR 0xA4, *CHR 0xA5*, *CHR 0xA6*, *CHR 0xA7*,
CHR 0xA8, *CHR 0xA9*, *CHR 0xAA*, *CHR 0xAB*,
CHR 0xAC, *CHR 0xAD*, *CHR 0xAE*, *CHR 0xAF*,
CHR 0xB0, *CHR 0xB1*, *CHR 0xB2*, *CHR 0xB3*,
CHR 0xB4, *CHR 0xB5*, *CHR 0xB6*, *CHR 0xB7*,
CHR 0xB8, *CHR 0xB9*, *CHR 0xBA*, *CHR 0xBB*,
CHR 0xBC, *CHR 0xBD*, *CHR 0xBE*, *CHR 0xBF*,
CHR 0xC0, *CHR 0xC1*, *CHR 0xC2*, *CHR 0xC3*,
CHR 0xC4, *CHR 0xC5*, *CHR 0xC6*, *CHR 0xC7*,
CHR 0xC8, *CHR 0xC9*, *CHR 0xCA*, *CHR 0xCB*,
CHR 0xCC, *CHR 0xCD*, *CHR 0xCE*, *CHR 0xCF*,
CHR 0xD0, *CHR 0xD1*, *CHR 0xD2*, *CHR 0xD3*,
CHR 0xD4, *CHR 0xD5*, *CHR 0xD6*, *CHR 0xD7*,
CHR 0xD8, *CHR 0xD9*, *CHR 0xDA*, *CHR 0xDB*,
CHR 0xDC, *CHR 0xDD*, *CHR 0xDE*, *CHR 0xDF*,
CHR 0xE0, *CHR 0xE1*, *CHR 0xE2*, *CHR 0xE3*,
CHR 0xE4, *CHR 0xE5*, *CHR 0xE6*, *CHR 0xE7*,
CHR 0xE8, *CHR 0xE9*, *CHR 0xEA*, *CHR 0xEB*,
CHR 0xEC, *CHR 0xED*, *CHR 0xEE*, *CHR 0xEF*,
CHR 0xF0, *CHR 0xF1*, *CHR 0xF2*, *CHR 0xF3*,
CHR 0xF4, *CHR 0xF5*, *CHR 0xF6*, *CHR 0xF7*,
CHR 0xF8, *CHR 0xF9*, *CHR 0xFA*, *CHR 0xFB*,
CHR 0xFC, *CHR 0xFD*, *CHR 0xFE*, *CHR 0xFF*]

definition
Enum.enum-all P $\longleftrightarrow$ *list-all P* (*Enum.enum* :: *char list*)

definition
Enum.enum-ex P $\longleftrightarrow$ *list-ex P* (*Enum.enum* :: *char list*)

lemma *enum-char-unfold*:
Enum.enum = *map char-of* [*0*..<*256*]
proof −
have *map* (*of-char* :: *char* ⇒ *nat*) *Enum.enum* = [*0*..<*256*]
by (*simp add*: *enum-char-def of-char-def upt-conv-Cons-Cons numeral-2-eq-2* [*symmetric*])
then have *map char-of* (*map* (*of-char* :: *char* ⇒ *nat*) *Enum.enum*) =
map char-of [*0*..<*256*]
by *simp*
then show *?thesis*
by *simp*
qed

instance proof
show *UNIV*: *UNIV* = *set* (*Enum.enum* :: *char list*)
by (*simp add*: *enum-char-unfold UNIV-char-of-nat atLeast0LessThan*)
show *distinct* (*Enum.enum* :: *char list*)

by (*auto simp add*: *enum-char-unfold distinct-map intro*: *inj-onI*)
show $\bigwedge P$. *Enum.enum-all P* $\longleftrightarrow$ *Ball* (*UNIV* :: *char set*) *P*
by (*simp add*: *UNIV enum-all-char-def list-all-iff*)
show $\bigwedge P$. *Enum.enum-ex P* $\longleftrightarrow$ *Bex* (*UNIV* :: *char set*) *P*
by (*simp add*: *UNIV enum-ex-char-def list-ex-iff*)
qed

end

lemma *linorder-char*:
class.linorder ($\lambda c\ d$. *of-char c* $\leq$ (*of-char d* :: *nat*)) ($\lambda c\ d$. *of-char c* $<$ (*of-char d* :: *nat*))
by *standard auto*

Optimized version for execution

definition *char-of-integer* :: *integer* $\Rightarrow$ *char*
where [*code-abbrev*]: *char-of-integer* = *char-of*

definition *integer-of-char* :: *char* $\Rightarrow$ *integer*
where [*code-abbrev*]: *integer-of-char* = *of-char*

lemma *char-of-integer-code* [*code*]:
char-of-integer k = (*let*
(*q0*, *b0*) = *bit-cut-integer k*;
(*q1*, *b1*) = *bit-cut-integer q0*;
(*q2*, *b2*) = *bit-cut-integer q1*;
(*q3*, *b3*) = *bit-cut-integer q2*;
(*q4*, *b4*) = *bit-cut-integer q3*;
(*q5*, *b5*) = *bit-cut-integer q4*;
(*q6*, *b6*) = *bit-cut-integer q5*;
(-, *b7*) = *bit-cut-integer q6*
in Char b0 b1 b2 b3 b4 b5 b6 b7)
by (*simp add*: *bit-cut-integer-def char-of-integer-def char-of-def div-mult2-numeral-eq bit-iff-odd-drop-bit drop-bit-eq-div*)

lemma *integer-of-char-code* [*code*]:
integer-of-char (*Char b0 b1 b2 b3 b4 b5 b6 b7*) =
((((((*of-bool b7* $*$ *2* $+$ *of-bool b6*) $*$ *2* $+$
of-bool b5) $*$ *2* $+$ *of-bool b4*) $*$ *2* $+$
of-bool b3) $*$ *2* $+$ *of-bool b2*) $*$ *2* $+$
of-bool b1) $*$ *2* $+$ *of-bool b0*
by (*simp add*: *integer-of-char-def of-char-def*)

73.2 Strings as dedicated type for target language code generation

73.2.1 Logical specification

context
begin

qualified definition *ascii-of* :: *char* ⇒ *char*
where *ascii-of c* = *Char* (*digit0 c*) (*digit1 c*) (*digit2 c*) (*digit3 c*) (*digit4 c*) (*digit5 c*) (*digit6 c*) *False*

qualified lemma *ascii-of-Char* [*simp*]:
ascii-of (*Char b0 b1 b2 b3 b4 b5 b6 b7*) = *Char b0 b1 b2 b3 b4 b5 b6 False*
by (*simp add*: *ascii-of-def*)

qualified lemma *digit0-ascii-of-iff* [*simp*]:
digit0 (*String.ascii-of c*) ⟷ *digit0 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit1-ascii-of-iff* [*simp*]:
digit1 (*String.ascii-of c*) ⟷ *digit1 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit2-ascii-of-iff* [*simp*]:
digit2 (*String.ascii-of c*) ⟷ *digit2 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit3-ascii-of-iff* [*simp*]:
digit3 (*String.ascii-of c*) ⟷ *digit3 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit4-ascii-of-iff* [*simp*]:
digit4 (*String.ascii-of c*) ⟷ *digit4 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit5-ascii-of-iff* [*simp*]:
digit5 (*String.ascii-of c*) ⟷ *digit5 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *digit6-ascii-of-iff* [*simp*]:
digit6 (*String.ascii-of c*) ⟷ *digit6 c*
by (*simp add*: *String.ascii-of-def*)

qualified lemma *not-digit7-ascii-of* [*simp*]:
¬ *digit7* (*ascii-of c*)
by (*simp add*: *ascii-of-def*)

qualified lemma *ascii-of-idem*:
ascii-of c = *c* **if** ¬ *digit7 c*
using *that* **by** (*cases c*) *simp*

qualified typedef *literal* = {*cs*. ∀ *c*∈*set cs*. ¬ *digit7 c*}
morphisms *explode Abs-literal*
proof
show [] ∈ {*cs*. ∀ *c*∈*set cs*. ¬ *digit7 c*}

by *simp*
qed

qualified setup-lifting *type-definition-literal*

qualified lift-definition *implode* :: *string* ⇒ *literal*
is *map ascii-of*
by *auto*

qualified lemma *implode-explode-eq* [*simp*]:
String.implode (*String.explode s*) = *s*
proof *transfer*
fix *cs*
show *map ascii-of cs* = *cs* **if** ∀ *c*∈*set cs*. ¬ *digit7 c*
using *that*
by (*induction cs*) (*simp-all add*: *ascii-of-idem*)
qed

qualified lemma *explode-implode-eq* [*simp*]:
String.explode (*String.implode cs*) = *map ascii-of cs*
by *transfer rule*

end

context *linordered-euclidean-semiring-bit-operations*
begin

context
begin

qualified lemma *char-of-ascii-of* [*simp*]:
‹*of-char* (*String.ascii-of c*) = *take-bit* 7 (*of-char c*)›
by (*cases c*) (*simp only*: *String.ascii-of-Char of-char-Char take-bit-horner-sum-bit-eq*, *simp*)

qualified lemma *ascii-of-char-of*:
‹*String.ascii-of* (*char-of a*) = *char-of* (*take-bit* 7 *a*)›
by (*simp add*: *char-of-def bit-simps*)

end

end

73.2.2 Syntactic representation

Logical ground representations for literals are:

1. *0* for the empty literal;

2. *Literal b0 ... b6 s* for a literal starting with one character and continued by another literal.

   Syntactic representations for literals are:

3. Printable text as string prefixed with *STR*;

4. A single ascii value as numerical hexadecimal value prefixed with *STR*.

instantiation *String.literal* :: *zero*
begin

context
begin

qualified lift-definition *zero-literal* :: *String.literal*
  is *Nil*
  by *simp*

instance ..

end

end

context
begin

qualified abbreviation (**output**) *empty-literal* :: *String.literal*
  where *empty-literal* ≡ *0*

qualified lift-definition *Literal* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *String.literal* ⇒ *String.literal*
  is λ*b0 b1 b2 b3 b4 b5 b6 cs. Char b0 b1 b2 b3 b4 b5 b6 False # cs*
  by *auto*

qualified lemma *Literal-eq-iff* [*simp*]:
  Literal b0 b1 b2 b3 b4 b5 b6 s = Literal c0 c1 c2 c3 c4 c5 c6 t
    ⟷ (*b0* ⟷ *c0*) ∧ (*b1* ⟷ *c1*) ∧ (*b2* ⟷ *c2*) ∧ (*b3* ⟷ *c3*)
      ∧ (*b4* ⟷ *c4*) ∧ (*b5* ⟷ *c5*) ∧ (*b6* ⟷ *c6*) ∧ *s* = *t*
  by *transfer simp*

qualified lemma *empty-neq-Literal* [*simp*]:
  empty-literal ≠ *Literal b0 b1 b2 b3 b4 b5 b6 s*
  by *transfer simp*

qualified lemma *Literal-neq-empty* [*simp*]:
  Literal b0 b1 b2 b3 b4 b5 b6 s ≠ *empty-literal*
  by *transfer simp*

end

code-datatype *0* :: *String.literal String.Literal*

syntax
-Literal :: *str-position* ⇒ *String.literal*
(‹(‹*open-block notation=‹literal string››STR* -)›)
-Ascii :: *num-const* ⇒ *String.literal*
(‹(‹*open-block notation=‹literal char code››STR* -)›)
syntax-consts
-Literal -Ascii ⇌ *String.Literal*

ML-file ‹*Tools/literal.ML*›

73.2.3 Operations

instantiation *String.literal* :: *plus*
begin

context
begin

qualified lift-definition *plus-literal* :: *String.literal* ⇒ *String.literal* ⇒ *String.literal*
is (@)
by *auto*

instance ..

end

end

instance *String.literal* :: *monoid-add*
by (*standard*; *transfer*) *simp-all*

lemma *add-Literal-assoc*:
‹*String.Literal b0 b1 b2 b3 b4 b5 b6 t* + *s* = *String.Literal b0 b1 b2 b3 b4 b5 b6* (*t* + *s*)›
by *transfer simp*

instantiation *String.literal* :: *size*
begin

context
includes *literal.lifting*
begin

lift-definition *size-literal* :: *String.literal* ⇒ *nat*
is *length* **.**

end

instance ..

end

instantiation *String.literal* :: *equal*
begin

context
begin

qualified lift-definition *equal-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool*
  is *HOL.equal* **.**

instance
  by (*standard*; *transfer*) (*simp add*: *equal*)

end

end

instantiation *String.literal* :: *linorder*
begin

context
begin

qualified lift-definition *less-eq-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool*
  is *ord.lexordp-eq* (λ*c d*. *of-char c* < (*of-char d* :: *nat*))
  .

qualified lift-definition *less-literal* :: *String.literal* ⇒ *String.literal* ⇒ *bool*
  is *ord.lexordp* (λ*c d*. *of-char c* < (*of-char d* :: *nat*))
  .

instance proof −
  from *linorder-char* **interpret** *linorder ord.lexordp-eq* (λ*c d*. *of-char c* < (*of-char d* :: *nat*))
    ord.lexordp (λ*c d*. *of-char c* < (*of-char d* :: *nat*)) :: *string* ⇒ *string* ⇒ *bool*
    by (*rule linorder.lexordp-linorder*)
  show *PROP ?thesis*
    by (*standard*; *transfer*) (*simp-all add*: *less-le-not-le linear*)
qed

end

end

lemma *infinite-literal*:
infinite (*UNIV* :: *String.literal set*)
proof −
define *S* **where** *S* = *range* (λ*n*. *replicate n CHR* ''*A*'')
have *inj-on String.implode S*
proof (*rule inj-onI*)
fix *cs ds*
assume *String.implode cs* = *String.implode ds*
then have *String.explode* (*String.implode cs*) = *String.explode* (*String.implode ds*)
by *simp*
moreover assume *cs* ∈ *S* **and** *ds* ∈ *S*
ultimately show *cs* = *ds*
by (*auto simp add*: *S-def*)
qed
moreover have *infinite S*
by (*auto simp add*: *S-def dest*: *finite-range-imageI* [*of - length*])
ultimately have *infinite* (*String.implode* ‘ *S*)
by (*simp add*: *finite-image-iff*)
then show *?thesis*
by (*auto intro*: *finite-subset*)
qed

lemma *add-literal-code* [*code*]:
‹*STR* '''' + *s* = *s*›
‹*s* + *STR* '''' = *s*›
‹*String.Literal b0 b1 b2 b3 b4 b5 b6 t* + *s* = *String.Literal b0 b1 b2 b3 b4 b5 b6* (*t* + *s*)›
by (*simp-all add*: *add-Literal-assoc*)

73.2.4 Executable conversions

context
begin

qualified lift-definition *asciis-of-literal* :: *String.literal* ⇒ *integer list*
is *map of-char*
.

qualified lemma *asciis-of-zero* [*simp*, *code*]:
asciis-of-literal 0 = []
by *transfer simp*

qualified lemma *asciis-of-Literal* [*simp*, *code*]:
asciis-of-literal (*String.Literal b0 b1 b2 b3 b4 b5 b6 s*) =
of-char (*Char b0 b1 b2 b3 b4 b5 b6 False*) # *asciis-of-literal s*
by *transfer simp*

qualified lift-definition *literal-of-asciis* :: *integer list* ⇒ *String.literal*
is *map* (*String.ascii-of* ∘ *char-of*)
by *auto*

qualified lemma *literal-of-asciis-Nil* [*simp*, *code*]:
literal-of-asciis [] = *0*
by *transfer simp*

qualified lemma *literal-of-asciis-Cons* [*simp*, *code*]:
literal-of-asciis (*k* # *ks*) = (*case char-of k*
of Char b0 b1 b2 b3 b4 b5 b6 b7 ⇒ *String.Literal b0 b1 b2 b3 b4 b5 b6* (*literal-of-asciis ks*))
by (*simp add*: *char-of-def*) (*transfer*, *simp add*: *char-of-def*)

qualified lemma *literal-of-asciis-of-literal* [*simp*]:
literal-of-asciis (*asciis-of-literal s*) = *s*
proof *transfer*
fix *cs*
assume ∀ *c*∈*set cs*. ¬ *digit7 c*
then show *map* (*String.ascii-of* ∘ *char-of*) (*map of-char cs*) = *cs*
by (*induction cs*) (*simp-all add*: *String.ascii-of-idem*)
qed

qualified lemma *explode-code* [*code*]:
String.explode s = *map char-of* (*asciis-of-literal s*)
by *transfer simp*

qualified lemma *implode-code* [*code*]:
String.implode cs = *literal-of-asciis* (*map of-char cs*)
by *transfer simp*

qualified lemma *equal-literal* [*code*]:
HOL.equal (*String.Literal b0 b1 b2 b3 b4 b5 b6 s*)
(*String.Literal a0 a1 a2 a3 a4 a5 a6 r*)
⟷ (*b0* ⟷ *a0*) ∧ (*b1* ⟷ *a1*) ∧ (*b2* ⟷ *a2*) ∧ (*b3* ⟷ *a3*)
∧ (*b4* ⟷ *a4*) ∧ (*b5* ⟷ *a5*) ∧ (*b6* ⟷ *a6*) ∧ (*s* = *r*)
by (*simp add*: *equal*)

end

73.2.5 Technical code generation setup

Alternative constructor for generated computations

context
begin

qualified definition *Literal′* :: *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *String.literal* ⇒ *String.literal*
where [*simp*]: *Literal′* = *String.Literal*

lemma [*code*]:
  ‹*Literal′ b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis*
    [*foldr* (*λb k. of-bool b + k * 2*) [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*] *0*] + *s*›
proof −
  have ‹*foldr* (*λb k. of-bool b + k * 2*) [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*] *0 = of-char* (*Char b0 b1 b2 b3 b4 b5 b6 False*)›
    by *simp*
  moreover have ‹*Literal′ b0 b1 b2 b3 b4 b5 b6 s = String.literal-of-asciis*
    [*of-char* (*Char b0 b1 b2 b3 b4 b5 b6 False*)] + *s*›
   by (*unfold Literal′-def*) (*transfer*, *simp only*: *list.simps comp-apply char-of-char*, *simp*)
  ultimately show *?thesis*
    by *simp*
qed

lemma [*code-computation-unfold*]:
  String.Literal = Literal′
  by *simp*

end

code-reserved
  (*SML*) *string String Char Str-Literal*
  and (*OCaml*) *string String Char Str-Literal*
  and (*Haskell*) *Str-Literal*
  and (*Scala*) *String Str-Literal*

code-identifier
  code-module *String* ⇀
    (*SML*) *Str* **and** (*OCaml*) *Str* **and** (*Haskell*) *Str* **and** (*Scala*) *Str*

code-printing
  type-constructor *String.literal* ⇀
    (*SML*) *string*
    and (*OCaml*) *string*
    and (*Haskell*) *String*
    and (*Scala*) *String*
| **constant** *STR ''''* ⇀
    (*SML*)
    and (*OCaml*)
    and (*Haskell*)
    and (*Scala*)

setup ‹
  fold Literal.add-code [*SML*, *OCaml*, *Haskell*, *Scala*]
›

code-printing

```
  code-module Str-Literal ⇀
    (SML) ‹structure Str-Literal : sig
  type int = IntInf.int
  val literal-of-asciis : int list -> string
  val asciis-of-literal : string -> int list
end = struct

open IntInf;

fun map f [] = []
  | map f (x :: xs) = f x :: map f xs; (* deliberate clone not relying on List.- structure
*)

fun check-ascii k =
  if 0 <= k andalso k < 128
  then k
  else raise Fail Non−ASCII character in literal;

val char-of-ascii = Char.chr o toInt o (fn k => k mod 128);

val ascii-of-char = check-ascii o fromInt o Char.ord;

val literal-of-asciis = String.implode o map char-of-ascii;

val asciis-of-literal = map ascii-of-char o String.explode;

end;› for constant String.literal-of-asciis String.asciis-of-literal
    and (OCaml) ‹module Str-Literal : sig
  val literal-of-asciis : Z.t list -> string
  val asciis-of-literal: string -> Z.t list
end = struct

(* deliberate clones not relying on List.- module *)

let rec length xs = match xs with
    [] -> 0
  | x :: xs -> 1 + length xs;;

let rec nth xs n = match xs with
  (x :: xs) -> if n <= 0 then x else nth xs (n − 1);;

let rec map-range f n =
  if n <= 0
    then []
    else
      let m = n − 1
    in map-range f m @ [f m];;

let implode f xs =
```

```
  String.init (length xs) (fun n -> f (nth xs n));;

let explode f s =
  map-range (fun n -> f (String.get s n)) (String.length s);;

let z-128 = Z.of-int 128;;

let check-ascii k =
  if 0 <= k && k < 128
  then k
  else failwith Non−ASCII character in literal;;

let char-of-ascii k = Char.chr (Z.to-int (Z.rem k z-128));;

let ascii-of-char c = Z.of-int (check-ascii (Char.code c));;

let literal-of-asciis ks = implode char-of-ascii ks;;

let asciis-of-literal s = explode ascii-of-char s;;

end;;› for constant String.literal-of-asciis String.asciis-of-literal
    and (Haskell) ‹module Str-Literal(literalOfAsciis, asciisOfLiteral) where

check-ascii :: Int -> Int
check-ascii k
  | (0 <= k && k < 128) = k
  | otherwise = error Non−ASCII character in literal

charOfAscii :: Integer -> Char
charOfAscii = toEnum . Prelude.fromInteger . (\k -> k `mod ` 128)

asciiOfChar :: Char -> Integer
asciiOfChar = toInteger . check-ascii . fromEnum

literalOfAsciis :: [Integer] -> [Char]
literalOfAsciis = map charOfAscii

asciisOfLiteral :: [Char] -> [Integer]
asciisOfLiteral = map asciiOfChar
› for constant String.literal-of-asciis String.asciis-of-literal
    and (Scala) ‹object Str-Literal {

private def checkAscii(k : Int) : Int =
  0 <= k && k < 128 match {
    case true => k
    case false => sys.error(Non−ASCII character in literal)
  }

private def charOfAscii(k : BigInt) : Char =
```

```
  (k % 128).charValue

private def asciiOfChar(c : Char) : BigInt =
  BigInt(checkAscii(c.toInt))

def literalOfAsciis(ks : List[BigInt]) : String =
  ks.map(charOfAscii).mkString

def asciisOfLiteral(s : String) : List[BigInt] =
  s.toList.map(asciiOfChar)

}
› for constant String.literal-of-asciis String.asciis-of-literal
| constant ‹(+) :: String.literal ⇒ String.literal ⇒ String.literal› ⇀
    (SML) infixl 18 ^
    and (OCaml) infixr 6 ^
    and (Haskell) infixr 5 ++
    and (Scala) infixl 7 +
| constant String.literal-of-asciis ⇀
    (SML) Str'-Literal.literal'-of'-asciis
    and (OCaml) Str'-Literal.literal'-of'-asciis
    and (Haskell) Str'-Literal.literalOfAsciis
    and (Scala) Str'-Literal.literalOfAsciis
| constant String.asciis-of-literal ⇀
    (SML) Str'-Literal.asciis'-of'-literal
    and (OCaml) Str'-Literal.asciis'-of'-literal
    and (Haskell) Str'-Literal.asciisOfLiteral
    and (Scala) Str'-Literal.asciisOfLiteral
| class-instance String.literal :: equal ⇀
    (Haskell) -
| constant ‹HOL.equal :: String.literal ⇒ String.literal ⇒ bool› ⇀
    (SML) !((- : string) = -)
    and (OCaml) !((- : string) = -)
    and (Haskell) infix 4 ==
    and (Scala) infixl 5 ==
| constant ‹(≤) :: String.literal ⇒ String.literal ⇒ bool› ⇀
    (SML) !((- : string) <= -)
    and (OCaml) !((- : string) <= -)
    and (Haskell) infix 4 <=
      — Order operations for String.literal work in Haskell only if no type class
instance needs to be generated, because String = [Char] in Haskell and char list
need not have the same order as String.literal.
    and (Scala) infixl 4 <=
    and (Eval) infixl 6 <=
| constant (<) :: String.literal ⇒ String.literal ⇒ bool ⇀
    (SML) !((- : string) < -)
    and (OCaml) !((- : string) < -)
    and (Haskell) infix 4 <
    and (Scala) infixl 4 <
```

and (*Eval*) **infixl** *6* <

73.2.6 Code generation utility

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path Code*)›

definition *abort* :: *String.literal* ⇒ (*unit* ⇒ *'a*) ⇒ *'a*
where [*simp*]: *abort - f* = *f* ()

declare [[*code drop*: *Code.abort*]]

lemma *abort-cong*:
msg = *msg'* ⟹ *Code.abort msg f* = *Code.abort msg' f*
by *simp*

setup ‹*Sign.map-naming Name-Space.parent-path*›

setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm Code.abort-cong*})›

code-printing
constant *Code.abort* ⇀
(*SML*) !(*raise*/ *Fail*/ -)
and (*OCaml*) *failwith*
and (*Haskell*) !(*error*/ ::/ *forall a.*/ *String* −> (() −> *a*) −> *a*)
and (*Scala*) !{/ *sys.error*((-));/ ((-)).*apply*(())/ }

73.2.7 Finally

lifting-update *literal.lifting*
lifting-forget *literal.lifting*

end

74 Reflecting Pure types into HOL

theory *Typerep*
imports *String*
begin

datatype *typerep* = *Typerep String.literal typerep list*

class *typerep* =
fixes *typerep* :: *'a itself* ⇒ *typerep*
begin

definition *typerep-of* :: *'a* ⇒ *typerep* **where**
[*simp*]: *typerep-of x* = *typerep TYPE*(*'a*)

end

```
syntax
  -TYPEREP :: type => logic  (‹(‹indent=1 notation=‹mixfix TYPEREP›› TYPEREP/(1'(-')))›)
syntax-consts
  -TYPEREP ⇌ typerep

parse-translation ‹
  let
    fun typerep-tr (*-TYPEREP*) [ty] =
          Syntax.const const-syntax‹typerep› $
              (Syntax.const syntax-const‹-constrain› $ Syntax.const const-syn-
tax‹Pure.type› $
                (Syntax.const type-syntax‹itself› $ ty))
      | typerep-tr (*-TYPEREP*) ts = raise TERM (typerep-tr, ts);
  in [(syntax-const‹-TYPEREP›, K typerep-tr)] end
›

typed-print-translation ‹
  let
    fun typerep-tr' ctxt (*typerep*) Type‹fun Type‹itself T› -›
            (Const (const-syntax‹Pure.type›, -) :: ts) =
          Term.list-comb
            (Syntax.const syntax-const‹-TYPEREP› $ Syntax-Phases.term-of-typ
ctxt T, ts)
      | typerep-tr' - T ts = raise Match;
  in [(const-syntax‹typerep›, typerep-tr')] end
›

setup ‹
let

fun add-typerep tyco thy =
  let
    val sorts = replicate (Sign.arity-number thy tyco) sort‹typerep›;
    val vs = Name.invent-types-global sorts;
    val ty = Type (tyco, map TFree vs);
    val lhs = Const‹typerep ty› $ Free (T, Term.itselfT ty);
    val rhs = Const‹Typerep› $ HOLogic.mk-literal tyco
      $ HOLogic.mk-list Type‹typerep› (map (HOLogic.mk-typerep o TFree) vs);
    val eq = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs, rhs));
  in
    thy
    |> Class.instantiation ([tyco], vs, sort‹typerep›)
    |> `(fn lthy => Syntax.check-term lthy eq)
   |-> (fn eq => Specification.definition {verbose = false} NONE [] [] (Binding.empty-atts,
eq))
    |> snd
    |> Class.prove-instantiation-exit (fn ctxt => Class.intro-classes-tac ctxt [])
  end;
```

fun ensure-typerep tyco thy =
if not (Sorts.has-instance (Sign.classes-of thy) tyco ***sort*** ‹*typerep*›)
andalso Sorts.has-instance (Sign.classes-of thy) tyco ***sort*** ‹*type*›
then add-typerep tyco thy else thy;

in

add-typerep ***type-name*** ‹*fun*›
#> Typedef.interpretation (Local-Theory.background-theory o ensure-typerep)
#> Code.type-interpretation ensure-typerep

end
›

lemma [*code*]:
HOL.equal (Typerep tyco1 tys1) (Typerep tyco2 tys2) ⟷ *HOL.equal tyco1 tyco2*
∧ *list-all2 HOL.equal tys1 tys2*
by (*auto simp add*: *eq-equal* [*symmetric*] *list-all2-eq* [*symmetric*])

lemma [*code nbe*]:
HOL.equal (x :: typerep) x ⟷ *True*
by (*fact equal-refl*)

code-printing
type-constructor *typerep* ⇀ (*Eval*) *Term.typ*
| **constant** *Typerep* ⇀ (*Eval*) *Term.Type*/ (-, -)

code-reserved
(*Eval*) *Term*

hide-const (**open**) *typerep Typerep*

end

75 Predicates as enumerations

theory *Predicate*
imports *String*
begin

75.1 The type of predicate enumerations (a monad)

datatype (*plugins only*: *extraction*) (*dead* 'a) *pred* = *Pred* (*eval*: 'a ⇒ *bool*)

lemma *pred-eqI*:
(⋀*w*. *eval P w* ⟷ *eval Q w*) ⟹ *P* = *Q*
by (*cases P*, *cases Q*) (*auto simp add*: *fun-eq-iff*)

lemma *pred-eq-iff*:
$P = Q \Longrightarrow (\bigwedge w.\ eval\ P\ w \longleftrightarrow eval\ Q\ w)$
by (*simp add*: *pred-eqI*)

instantiation *pred* :: (*type*) *complete-lattice*
begin

definition
$P \leq Q \longleftrightarrow eval\ P \leq eval\ Q$

definition
$P < Q \longleftrightarrow eval\ P < eval\ Q$

definition
$\bot = Pred\ \bot$

lemma *eval-bot* [*simp*]:
$eval\ \bot\ = \bot$
by (*simp add*: *bot-pred-def*)

definition
$\top = Pred\ \top$

lemma *eval-top* [*simp*]:
$eval\ \top\ = \top$
by (*simp add*: *top-pred-def*)

definition
$P \sqcap Q = Pred\ (eval\ P \sqcap eval\ Q)$

lemma *eval-inf* [*simp*]:
$eval\ (P \sqcap Q) = eval\ P \sqcap eval\ Q$
by (*simp add*: *inf-pred-def*)

definition
$P \sqcup Q = Pred\ (eval\ P \sqcup eval\ Q)$

lemma *eval-sup* [*simp*]:
$eval\ (P \sqcup Q) = eval\ P \sqcup eval\ Q$
by (*simp add*: *sup-pred-def*)

definition
$\bigsqcap A = Pred\ (\bigsqcap (eval\ `\ A))$

lemma *eval-Inf* [*simp*]:
$eval\ (\bigsqcap A) = \bigsqcap (eval\ `\ A)$
by (*simp add*: *Inf-pred-def*)

definition

$\bigsqcup A = Pred\ (\bigsqcup (eval\ `\ A))$

lemma *eval-Sup* [*simp*]:
eval $(\bigsqcup A) = \bigsqcup (eval\ `\ A)$
by (*simp add*: *Sup-pred-def*)

instance proof
qed (*auto intro*!: *pred-eqI simp add*: *less-eq-pred-def less-pred-def le-fun-def less-fun-def*)

end

lemma *eval-INF* [*simp*]:
eval $(\bigsqcap (f\ `\ A)) = \bigsqcap ((eval \circ f)\ `\ A)$
by (*simp add*: *image-comp*)

lemma *eval-SUP* [*simp*]:
eval $(\bigsqcup (f\ `\ A)) = \bigsqcup ((eval \circ f)\ `\ A)$
by (*simp add*: *image-comp*)

instantiation *pred* :: (*type*) *complete-boolean-algebra*
begin

definition
$-\ P = Pred\ (-\ eval\ P)$

lemma *eval-compl* [*simp*]:
eval $(-\ P) = -\ eval\ P$
by (*simp add*: *uminus-pred-def*)

definition
$P - Q = Pred\ (eval\ P - eval\ Q)$

lemma *eval-minus* [*simp*]:
eval $(P - Q) = eval\ P - eval\ Q$
by (*simp add*: *minus-pred-def*)

instance proof
fix A::$'a$ *pred set set*
show $\bigsqcap (Sup\ `\ A) \leq \bigsqcup (Inf\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y\ \in\ Y\})$
proof (*simp add*: *less-eq-pred-def Sup-fun-def Inf-fun-def*, *safe*)
fix w
assume A: $\forall\, x{\in}A.\ \exists f{\in}x.\ eval\ f\ w$
define F **where** $F = (\lambda\ x\ .\ SOME\ f\ .\ f\ \in\ x\ \wedge\ eval\ f\ w)$
have [*simp*]: $(\forall f{\in}\ (F\ `\ A).\ eval\ f\ w)$
by (*metis* (*no-types*, *lifting*) *A F-def image-iff some-eq-ex*)
have $(\exists f.\ F\ `\ A = f\ `\ A\ \wedge\ (\forall\ Y{\in}A.\ f\ Y\ \in\ Y)) \wedge (\forall f{\in}(F\ `\ A).\ eval\ f\ w)$
using A **by** (*simp*, *metis* (*no-types*, *lifting*) *F-def someI*)+
from *this* **show** $\exists\, x.\ (\exists f.\ x = f\ `\ A\ \wedge\ (\forall\ Y{\in}A.\ f\ Y\ \in\ Y)) \wedge (\forall f{\in}x.\ eval\ f\ w)$
by (*rule exI* [*of* - $F\ `\ A$])

qed
qed (*auto intro*!: *pred-eqI*)

end

definition *single* :: *'a* ⇒ *'a pred* **where**
single x = *Pred* ((=) *x*)

lemma *eval-single* [*simp*]:
eval (*single x*) = (=) *x*
by (*simp add*: *single-def*)

definition *bind* :: *'a pred* ⇒ (*'a* ⇒ *'b pred*) ⇒ *'b pred* (**infixl** ‹⨠› *70*) **where**
P ⨠ *f* = (⨆(*f* ‘ {*x*. *eval P x*}))

lemma *eval-bind* [*simp*]:
eval (*P* ⨠ *f*) = *eval* (⨆(*f* ‘ {*x*. *eval P x*}))
by (*simp add*: *bind-def*)

lemma *bind-bind*:
(*P* ⨠ *Q*) ⨠ *R* = *P* ⨠ (λ*x*. *Q x* ⨠ *R*)
by (*rule pred-eqI*) *auto*

lemma *bind-single*:
P ⨠ *single* = *P*
by (*rule pred-eqI*) *auto*

lemma *single-bind*:
single x ⨠ *P* = *P x*
by (*rule pred-eqI*) *auto*

lemma *bottom-bind*:
⊥ ⨠ *P* = ⊥
by (*rule pred-eqI*) *auto*

lemma *sup-bind*:
(*P* ⊔ *Q*) ⨠ *R* = *P* ⨠ *R* ⊔ *Q* ⨠ *R*
by (*rule pred-eqI*) *auto*

lemma *Sup-bind*:
(⨆ *A* ⨠ *f*) = ⨆((λ*x*. *x* ⨠ *f*) ‘ *A*)
by (*rule pred-eqI*) *auto*

lemma *pred-iffI*:
assumes ⋀*x*. *eval A x* ⟹ *eval B x*
and ⋀*x*. *eval B x* ⟹ *eval A x*
shows *A* = *B*
using *assms* **by** (*auto intro*: *pred-eqI*)

lemma *singleI*: *eval* (*single x*) *x*
by *simp*

lemma *singleI-unit*: *eval* (*single* ()) *x*
by *simp*

lemma *singleE*: *eval* (*single x*) *y* ⟹ (*y* = *x* ⟹ *P*) ⟹ *P*
by *simp*

lemma *singleE'*: *eval* (*single x*) *y* ⟹ (*x* = *y* ⟹ *P*) ⟹ *P*
by *simp*

lemma *bindI*: *eval P x* ⟹ *eval* (*Q x*) *y* ⟹ *eval* (*P* ⤜ *Q*) *y*
by *auto*

lemma *bindE*: *eval* (*R* ⤜ *Q*) *y* ⟹ (⋀*x*. *eval R x* ⟹ *eval* (*Q x*) *y* ⟹ *P*) ⟹ *P*
by *auto*

lemma *botE*: *eval* ⊥ *x* ⟹ *P*
by *auto*

lemma *supI1*: *eval A x* ⟹ *eval* (*A* ⊔ *B*) *x*
by *auto*

lemma *supI2*: *eval B x* ⟹ *eval* (*A* ⊔ *B*) *x*
by *auto*

lemma *supE*: *eval* (*A* ⊔ *B*) *x* ⟹ (*eval A x* ⟹ *P*) ⟹ (*eval B x* ⟹ *P*) ⟹ *P*
by *auto*

lemma *single-not-bot* [*simp*]:
single x ≠ ⊥
by (*auto simp add*: *single-def bot-pred-def fun-eq-iff*)

lemma *not-bot*:
assumes *A* ≠ ⊥
obtains *x* **where** *eval A x*
using *assms* **by** (*cases A*) (*auto simp add*: *bot-pred-def*)

75.2 Emptiness check and definite choice

definition *is-empty* :: *'a pred* ⇒ *bool* **where**
is-empty A ⟷ *A* = ⊥

lemma *is-empty-bot*:
is-empty ⊥
by (*simp add*: *is-empty-def*)

lemma *not-is-empty-single*:
  ¬ *is-empty* (*single x*)
  by (*auto simp add*: *is-empty-def single-def bot-pred-def fun-eq-iff*)

lemma *is-empty-sup*:
  is-empty (*A* ⊔ *B*) ⟷ *is-empty A* ∧ *is-empty B*
  by (*auto simp add*: *is-empty-def*)

definition *singleton* :: (*unit* ⇒ *'a*) ⇒ *'a pred* ⇒ *'a* **where**
  singleton default A = (*if* ∃!*x*. *eval A x then THE x*. *eval A x else default* ()) **for** *default*

lemma *singleton-eqI*:
  ∃!*x*. *eval A x* ⟹ *eval A x* ⟹ *singleton default A* = *x* **for** *default*
  by (*auto simp add*: *singleton-def*)

lemma *eval-singletonI*:
  ∃!*x*. *eval A x* ⟹ *eval A* (*singleton default A*) **for** *default*
proof −
  assume *assm*: ∃!*x*. *eval A x*
  then obtain *x* **where** *x*: *eval A x* **..**
  with *assm* **have** *singleton default A* = *x* **by** (*rule singleton-eqI*)
  with *x* **show** *?thesis* **by** *simp*
qed

lemma *single-singleton*:
  ∃!*x*. *eval A x* ⟹ *single* (*singleton default A*) = *A* **for** *default*
proof −
  assume *assm*: ∃!*x*. *eval A x*
  then have *eval A* (*singleton default A*)
    by (*rule eval-singletonI*)
  moreover from *assm* **have** ⋀*x*. *eval A x* ⟹ *singleton default A* = *x*
    by (*rule singleton-eqI*)
  ultimately have *eval* (*single* (*singleton default A*)) = *eval A*
    by (*simp* (*no-asm-use*) *add*: *single-def fun-eq-iff*) *blast*
  then have ⋀*x*. *eval* (*single* (*singleton default A*)) *x* = *eval A x*
    by *simp*
  then show *?thesis* **by** (*rule pred-eqI*)
qed

lemma *singleton-undefinedI*:
  ¬ (∃!*x*. *eval A x*) ⟹ *singleton default A* = *default* () **for** *default*
  by (*simp add*: *singleton-def*)

lemma *singleton-bot*:
  singleton default ⊥ = *default* () **for** *default*
  by (*auto simp add*: *bot-pred-def intro*: *singleton-undefinedI*)

lemma *singleton-single*:

singleton default (*single x*) = *x* **for** *default*
by (*auto simp add*: *intro*: *singleton-eqI singleI elim*: *singleE*)

lemma *singleton-sup-single-single*:
singleton default (*single x* ⊔ *single y*) = (*if x* = *y then x else default* ()) **for** *default*
proof (*cases x* = *y*)
case *True* **then show** *?thesis* **by** (*simp add*: *singleton-single*)
next
case *False*
have *eval* (*single x* ⊔ *single y*) *x*
and *eval* (*single x* ⊔ *single y*) *y*
by (*auto intro*: *supI1 supI2 singleI*)
with *False* **have** ¬ (∃!*z*. *eval* (*single x* ⊔ *single y*) *z*)
by *blast*
then have *singleton default* (*single x* ⊔ *single y*) = *default* ()
by (*rule singleton-undefinedI*)
with *False* **show** *?thesis* **by** *simp*
qed

lemma *singleton-sup-aux*:
singleton default (*A* ⊔ *B*) = (*if A* = ⊥ *then singleton default B*
else if B = ⊥ *then singleton default A*
else singleton default
(*single* (*singleton default A*) ⊔ *single* (*singleton default B*))) **for** *default*
proof (*cases* (∃!*x*. *eval A x*) ∧ (∃!*y*. *eval B y*))
case *True* **then show** *?thesis* **by** (*simp add*: *single-singleton*)
next
case *False*
from *False* **have** *A-or-B*:
singleton default A = *default* () ∨ *singleton default B* = *default* ()
by (*auto intro*!: *singleton-undefinedI*)
then have *rhs*: *singleton default*
(*single* (*singleton default A*) ⊔ *single* (*singleton default B*)) = *default* ()
by (*auto simp add*: *singleton-sup-single-single singleton-single*)
from *False* **have** *not-unique*:
¬ (∃!*x*. *eval A x*) ∨ ¬ (∃!*y*. *eval B y*) **by** *simp*
show *?thesis* **proof** (*cases A* ≠ ⊥ ∧ *B* ≠ ⊥)
case *True*
then obtain *a b* **where** *a*: *eval A a* **and** *b*: *eval B b*
by (*blast elim*: *not-bot*)
with *True not-unique* **have** ¬ (∃!*x*. *eval* (*A* ⊔ *B*) *x*)
by (*auto simp add*: *sup-pred-def bot-pred-def*)
then have *singleton default* (*A* ⊔ *B*) = *default* () **by** (*rule singleton-undefinedI*)
with *True rhs* **show** *?thesis* **by** *simp*
next
case *False* **then show** *?thesis* **by** *auto*
qed
qed

lemma *singleton-sup*:
singleton default $(A \sqcup B)$ = (*if* $A = \bot$ *then singleton default* B
else if $B = \bot$ *then singleton default* A
else if singleton default A = *singleton default* B *then singleton default* A *else default* ()) **for** *default*
using *singleton-sup-aux* [*of default* A B] **by** (*simp only*: *singleton-sup-single-single*)

75.3 Derived operations

definition *if-pred* :: *bool* ⇒ *unit pred* **where**
if-pred-eq: *if-pred* b = (*if* b *then single* () *else* $\bot$)

definition *holds* :: *unit pred* ⇒ *bool* **where**
holds-eq: *holds* P = *eval* P ()

definition *not-pred* :: *unit pred* ⇒ *unit pred* **where**
not-pred-eq: *not-pred* P = (*if eval* P () *then* $\bot$ *else single* ())

lemma *if-predI*: $P \Longrightarrow$ *eval* (*if-pred* P) ()
unfolding *if-pred-eq* **by** (*auto intro*: *singleI*)

lemma *if-predE*: *eval* (*if-pred* b) $x \Longrightarrow (b \Longrightarrow x = () \Longrightarrow P) \Longrightarrow P$
unfolding *if-pred-eq* **by** (*cases* b) (*auto elim*: *botE*)

lemma *not-predI*: $\neg P \Longrightarrow$ *eval* (*not-pred* (*Pred* ($\lambda u.$ P))) ()
unfolding *not-pred-eq* **by** (*auto intro*: *singleI*)

lemma *not-predI'*: $\neg$ *eval* P () $\Longrightarrow$ *eval* (*not-pred* P) ()
unfolding *not-pred-eq* **by** (*auto intro*: *singleI*)

lemma *not-predE*: *eval* (*not-pred* (*Pred* ($\lambda u.$ P))) $x \Longrightarrow (\neg P \Longrightarrow thesis) \Longrightarrow thesis$
unfolding *not-pred-eq*
by (*auto split*: *if-split-asm elim*: *botE*)

lemma *not-predE'*: *eval* (*not-pred* P) $x \Longrightarrow (\neg$ *eval* P $x \Longrightarrow thesis) \Longrightarrow thesis$
unfolding *not-pred-eq*
by (*auto split*: *if-split-asm elim*: *botE*)
lemma f () = *False* ∨ f () = *True*
by *simp*

lemma *closure-of-bool-cases* [*no-atp*]:
fixes f :: *unit* ⇒ *bool*
assumes $f = (\lambda u.\ False) \Longrightarrow P\ f$
assumes $f = (\lambda u.\ True) \Longrightarrow P\ f$
shows $P\ f$
proof −
have $f = (\lambda u.\ False) \vee f = (\lambda u.\ True)$

```
    apply (cases f ())
    apply (rule disjI2)
    apply (rule ext)
    apply (simp add: unit-eq)
    apply (rule disjI1)
    apply (rule ext)
    apply (simp add: unit-eq)
    done
  from this assms show ?thesis by blast
qed

lemma unit-pred-cases:
  assumes P ⊥
  assumes P (single ())
  shows P Q
using assms unfolding bot-pred-def bot-fun-def bot-bool-def empty-def single-def
proof (cases Q)
  fix f
  assume P (Pred (λu. False)) P (Pred (λu. () = u))
  then have P (Pred f)
    by (cases - f rule: closure-of-bool-cases) simp-all
  moreover assume Q = Pred f
  ultimately show P Q by simp
qed

lemma holds-if-pred:
  holds (if-pred b) = b
unfolding if-pred-eq holds-eq
by (cases b) (auto intro: singleI elim: botE)

lemma if-pred-holds:
  if-pred (holds P) = P
unfolding if-pred-eq holds-eq
by (rule unit-pred-cases) (auto intro: singleI elim: botE)

lemma is-empty-holds:
  is-empty P ⟷ ¬ holds P
unfolding is-empty-def holds-eq
by (rule unit-pred-cases) (auto elim: botE intro: singleI)

definition map :: ('a ⇒ 'b) ⇒ 'a pred ⇒ 'b pred where
  map f P = P ⤜ (single ∘ f)

lemma eval-map [simp]:
  eval (map f P) = (⨆x∈{x. eval P x}. (λy. f x = y))
  by (simp add: map-def comp-def image-comp)

functor map: map
  by (rule ext, rule pred-eqI, auto)+
```

75.4 Implementation

datatype (*plugins only*: *code extraction*) (*dead* *'a*) *seq* =
  Empty
| *Insert* *'a* *'a pred*
| *Join* *'a pred* *'a seq*

primrec *pred-of-seq* :: *'a seq* ⇒ *'a pred* **where**
  pred-of-seq Empty = ⊥
| *pred-of-seq* (*Insert x P*) = *single x* ⊔ *P*
| *pred-of-seq* (*Join P xq*) = *P* ⊔ *pred-of-seq xq*

definition *Seq* :: (*unit* ⇒ *'a seq*) ⇒ *'a pred* **where**
  Seq f = *pred-of-seq* (*f* ())

code-datatype *Seq*

primrec *member* :: *'a seq* ⇒ *'a* ⇒ *bool* **where**
  member Empty x ⟷ *False*
| *member* (*Insert y P*) *x* ⟷ *x* = *y* ∨ *eval P x*
| *member* (*Join P xq*) *x* ⟷ *eval P x* ∨ *member xq x*

lemma *eval-member*:
  member xq = *eval* (*pred-of-seq xq*)
proof (*induct xq*)
  case *Empty* **show** *?case*
  by (*auto simp add*: *fun-eq-iff elim*: *botE*)
next
  case *Insert* **show** *?case*
  by (*auto simp add*: *fun-eq-iff elim*: *supE singleE intro*: *supI1 supI2 singleI*)
next
  case *Join* **then show** *?case*
  by (*auto simp add*: *fun-eq-iff elim*: *supE intro*: *supI1 supI2*)
qed

lemma *eval-code* [*code*]: *eval* (*Seq f*) = *member* (*f* ())
  unfolding *Seq-def* **by** (*rule sym*, *rule eval-member*)

lemma *single-code* [*code*]:
  single x = *Seq* (λ*u*. *Insert x* ⊥)
  unfolding *Seq-def* **by** *simp*

primrec *apply* :: (*'a* ⇒ *'b pred*) ⇒ *'a seq* ⇒ *'b seq* **where**
  apply f Empty = *Empty*
| *apply f* (*Insert x P*) = *Join* (*f x*) (*Join* (*P* ⤜ *f*) *Empty*)
| *apply f* (*Join P xq*) = *Join* (*P* ⤜ *f*) (*apply f xq*)

lemma *apply-bind*:
  pred-of-seq (*apply f xq*) = *pred-of-seq xq* ⤜ *f*
proof (*induct xq*)

```
  case Empty show ?case
    by (simp add: bottom-bind)
next
  case Insert show ?case
    by (simp add: single-bind sup-bind)
next
  case Join then show ?case
    by (simp add: sup-bind)
qed

lemma bind-code [code]:
  Seq g ⤜ f = Seq (λu. apply f (g ()))
  unfolding Seq-def by (rule sym, rule apply-bind)

lemma bot-set-code [code]:
  ⊥ = Seq (λu. Empty)
  unfolding Seq-def by simp

primrec adjunct :: 'a pred ⇒ 'a seq ⇒ 'a seq where
  adjunct P Empty = Join P Empty
| adjunct P (Insert x Q) = Insert x (Q ⊔ P)
| adjunct P (Join Q xq) = Join Q (adjunct P xq)

lemma adjunct-sup:
  pred-of-seq (adjunct P xq) = P ⊔ pred-of-seq xq
  by (induct xq) (simp-all add: sup-assoc sup-commute sup-left-commute)

lemma sup-code [code]:
  Seq f ⊔ Seq g = Seq (λu. case f ()
    of Empty ⇒ g ()
     | Insert x P ⇒ Insert x (P ⊔ Seq g)
     | Join P xq ⇒ adjunct (Seq g) (Join P xq))
proof (cases f ())
  case Empty
  thus ?thesis
    unfolding Seq-def by (simp add: sup-commute [of ⊥])
next
  case Insert
  thus ?thesis
    unfolding Seq-def by (simp add: sup-assoc)
next
  case Join
  thus ?thesis
    unfolding Seq-def
    by (simp add: adjunct-sup sup-assoc sup-commute sup-left-commute)
qed

primrec contained :: 'a seq ⇒ 'a pred ⇒ bool where
  contained Empty Q ⟷ True
```

| *contained* (*Insert x P*) *Q* ⟷ *eval Q x* ∧ *P* ≤ *Q*
| *contained* (*Join P xq*) *Q* ⟷ *P* ≤ *Q* ∧ *contained xq Q*

lemma *single-less-eq-eval*:
single x ≤ *P* ⟷ *eval P x*
by (*auto simp add*: *less-eq-pred-def le-fun-def*)

lemma *contained-less-eq*:
contained xq Q ⟷ *pred-of-seq xq* ≤ *Q*
by (*induct xq*) (*simp-all add*: *single-less-eq-eval*)

lemma *less-eq-pred-code* [*code*]:
Seq f ≤ *Q* = (*case f* ()
of Empty ⇒ *True*
| *Insert x P* ⇒ *eval Q x* ∧ *P* ≤ *Q*
| *Join P xq* ⇒ *P* ≤ *Q* ∧ *contained xq Q*)
by (*cases f* ())
(*simp-all add*: *Seq-def single-less-eq-eval contained-less-eq*)

instantiation *pred* :: (*type*) *equal*
begin

definition *equal-pred*
where [*simp*]: *HOL.equal P Q* ⟷ *P* = (*Q* :: *'a pred*)

instance by *standard simp*

end

lemma [*code nbe*]:
HOL.equal P P ⟷ *True* **for** *P* :: *'a pred*
by (*fact equal-refl*)

lemma [*code*]:
HOL.equal P Q ⟷ *P* ≤ *Q* ∧ *Q* ≤ *P* **for** *P Q* :: *'a pred*
by *auto*

lemma [*code*]:
case-pred f P = *f* (*eval P*)
by (*fact pred.case-eq-if*)

lemma [*code*]:
rec-pred f P = *f* (*eval P*)
by (*cases P*) *simp*

inductive *eq* :: *'a* ⇒ *'a* ⇒ *bool* **where** *eq x x*

lemma *eq-is-eq*: *eq x y* ≡ (*x* = *y*)
by (*rule eq-reflection*) (*auto intro*: *eq.intros elim*: *eq.cases*)

primrec *null* :: *'a seq* ⇒ *bool* **where**
null Empty ⟷ *True*
| *null* (*Insert x P*) ⟷ *False*
| *null* (*Join P xq*) ⟷ *is-empty P* ∧ *null xq*

lemma *null-is-empty*:
null xq ⟷ *is-empty* (*pred-of-seq xq*)
by (*induct xq*) (*simp-all add*: *is-empty-bot not-is-empty-single is-empty-sup*)

lemma *is-empty-code* [*code*]:
is-empty (*Seq f*) ⟷ *null* (*f* ())
by (*simp add*: *null-is-empty Seq-def*)

primrec *the-only* :: (*unit* ⇒ *'a*) ⇒ *'a seq* ⇒ *'a* **where**
the-only default Empty = *default* () **for** *default*
| *the-only default* (*Insert x P*) =
(*if is-empty P then x else let y* = *singleton default P in if x* = *y then x else default* ()) **for** *default*
| *the-only default* (*Join P xq*) =
(*if is-empty P then the-only default xq else if null xq then singleton default P*
else let x = *singleton default P*; *y* = *the-only default xq in*
if x = *y then x else default* ()) **for** *default*

lemma *the-only-singleton*:
the-only default xq = *singleton default* (*pred-of-seq xq*) **for** *default*
by (*induct xq*)
(*auto simp add*: *singleton-bot singleton-single is-empty-def*
null-is-empty Let-def singleton-sup)

lemma *singleton-code* [*code*]:
singleton default (*Seq f*) =
(*case f* () *of*
Empty ⇒ *default* ()
| *Insert x P* ⇒ *if is-empty P then x*
else let y = *singleton default P in*
if x = *y then x else default* ()
| *Join P xq* ⇒ *if is-empty P then the-only default xq*
else if null xq then singleton default P
else let x = *singleton default P*; *y* = *the-only default xq in*
if x = *y then x else default* ()) **for** *default*
by (*cases f* ())
(*auto simp add*: *Seq-def the-only-singleton is-empty-def*
null-is-empty singleton-bot singleton-single singleton-sup Let-def)

definition *the* :: *'a pred* ⇒ *'a* **where**
the A = (*THE x*. *eval A x*)

lemma *the-eqI*:

```
  (THE x. eval P x) = x ⟹ the P = x
  by (simp add: the-def)

lemma the-eq [code]: the A = singleton (λx. Code.abort (STR ''not-unique'') (λ-. the A)) A
  by (rule the-eqI) (simp add: singleton-def the-def)

code-reflect Predicate
  datatypes pred = Seq and seq = Empty | Insert | Join

ML ‹
signature PREDICATE =
sig
  val anamorph: ('a -> ('b * 'a) option) -> int -> 'a -> 'b list * 'a
  datatype 'a pred = Seq of (unit -> 'a seq)
  and 'a seq = Empty | Insert of 'a * 'a pred | Join of 'a pred * 'a seq
  val map: ('a -> 'b) -> 'a pred -> 'b pred
  val yield: 'a pred -> ('a * 'a pred) option
  val yieldn: int -> 'a pred -> 'a list * 'a pred
end;

structure Predicate : PREDICATE =
struct

fun anamorph f k x =
 (if k = 0 then ([], x)
  else case f x
   of NONE => ([], x)
    | SOME (v, y) => let
        val k' = k - 1;
        val (vs, z) = anamorph f k' y
      in (v :: vs, z) end);

datatype pred = datatype Predicate.pred
datatype seq = datatype Predicate.seq

fun map f = @{code Predicate.map} f;

fun yield (Seq f) = next (f ())
and next Empty = NONE
  | next (Insert (x, P)) = SOME (x, P)
  | next (Join (P, xq)) = (case yield P
     of NONE => next xq
      | SOME (x, Q) => SOME (x, Seq (fn - => Join (Q, xq))));

fun yieldn k = anamorph yield k;

end;
›
```

Conversion from and to sets

definition *pred-of-set* :: *'a set* ⇒ *'a pred* **where**
pred-of-set = *Pred* ∘ (λ*A x*. *x* ∈ *A*)

lemma *eval-pred-of-set* [*simp*]:
eval (*pred-of-set A*) *x* ⟷ *x* ∈*A*
by (*simp add*: *pred-of-set-def*)

definition *set-of-pred* :: *'a pred* ⇒ *'a set* **where**
set-of-pred = *Collect* ∘ *eval*

lemma *member-set-of-pred* [*simp*]:
x ∈ *set-of-pred P* ⟷ *Predicate.eval P x*
by (*simp add*: *set-of-pred-def*)

definition *set-of-seq* :: *'a seq* ⇒ *'a set* **where**
set-of-seq = *set-of-pred* ∘ *pred-of-seq*

lemma *member-set-of-seq* [*simp*]:
x ∈ *set-of-seq xq* = *Predicate.member xq x*
by (*simp add*: *set-of-seq-def eval-member*)

lemma *of-pred-code* [*code*]:
set-of-pred (*Predicate.Seq f*) = (*case f* () *of*
Predicate.Empty ⇒ {}
| *Predicate.Insert x P* ⇒ *insert x* (*set-of-pred P*)
| *Predicate.Join P xq* ⇒ *set-of-pred P* ∪ *set-of-seq xq*)
by (*auto split*: *seq.split simp add*: *eval-code*)

lemma *of-seq-code* [*code*]:
set-of-seq Predicate.Empty = {}
set-of-seq (*Predicate.Insert x P*) = *insert x* (*set-of-pred P*)
set-of-seq (*Predicate.Join P xq*) = *set-of-pred P* ∪ *set-of-seq xq*
by *auto*

Lazy Evaluation of an indexed function

function *iterate-upto* :: (*natural* ⇒ *'a*) ⇒ *natural* ⇒ *natural* ⇒ *'a Predicate.pred*
where
iterate-upto f n m =
Predicate.Seq (%*u*. *if n* > *m then Predicate.Empty*
else Predicate.Insert (*f n*) (*iterate-upto f* (*n* + *1*) *m*))
by *pat-completeness auto*

termination by (*relation measure* (%(*f*, *n*, *m*). *nat-of-natural* (*m* + *1* − *n*)))
(*auto simp add*: *less-natural-def*)

Misc

declare *Inf-set-fold* [**where** *'a* = *'a Predicate.pred*, *code*]
declare *Sup-set-fold* [**where** *'a* = *'a Predicate.pred*, *code*]

lemma *pred-of-set-fold-sup*:
assumes *finite A*
shows *pred-of-set A = Finite-Set.fold sup bot* (*Predicate.single ‘ A*) (**is** *?lhs = ?rhs*)
proof (*rule sym*)
interpret *comp-fun-idem sup* :: *'a Predicate.pred ⇒ 'a Predicate.pred ⇒ 'a Predicate.pred*
by (*fact comp-fun-idem-sup*)
from ‹*finite A*› **show** *?rhs = ?lhs* **by** (*induct A*) (*auto intro*!: *pred-eqI*)
qed

lemma *pred-of-set-set-fold-sup*:
pred-of-set (*set xs*) = *fold sup* (*List.map Predicate.single xs*) *bot*
proof −
interpret *comp-fun-idem sup* :: *'a Predicate.pred ⇒ 'a Predicate.pred ⇒ 'a Predicate.pred*
by (*fact comp-fun-idem-sup*)
show *?thesis* **by** (*simp add*: *pred-of-set-fold-sup fold-set-fold* [*symmetric*])
qed

lemma *pred-of-set-set-foldr-sup* [*code*]:
pred-of-set (*set xs*) = *foldr sup* (*List.map Predicate.single xs*) *bot*
by (*simp add*: *pred-of-set-set-fold-sup ac-simps foldr-fold fun-eq-iff*)

no-notation *bind* (**infixl** ‹≫=› *70*)

hide-type (**open**) *pred seq*
hide-const (**open**) *Pred eval single bind is-empty singleton if-pred not-pred holds Empty Insert Join Seq member pred-of-seq apply adjunct null the-only eq map the iterate-upto*
hide-fact (**open**) *null-def member-def*

end

76 Lazy sequences

theory *Lazy-Sequence*
imports *Predicate*
begin

76.1 Type of lazy sequences

datatype (*plugins only*: *code extraction*) (*dead 'a*) *lazy-sequence* =
lazy-sequence-of-list 'a list

primrec *list-of-lazy-sequence* :: *'a lazy-sequence ⇒ 'a list*

where
list-of-lazy-sequence (*lazy-sequence-of-list xs*) = *xs*

lemma *lazy-sequence-of-list-of-lazy-sequence* [*simp*]:
lazy-sequence-of-list (*list-of-lazy-sequence xq*) = *xq*
by (*cases xq*) *simp-all*

lemma *lazy-sequence-eqI*:
list-of-lazy-sequence xq = *list-of-lazy-sequence yq* ⟹ *xq* = *yq*
by (*cases xq*, *cases yq*) *simp*

lemma *lazy-sequence-eq-iff*:
xq = *yq* ⟷ *list-of-lazy-sequence xq* = *list-of-lazy-sequence yq*
by (*auto intro*: *lazy-sequence-eqI*)

lemma *case-lazy-sequence* [*simp*]:
case-lazy-sequence f xq = *f* (*list-of-lazy-sequence xq*)
by (*cases xq*) *auto*

lemma *rec-lazy-sequence* [*simp*]:
rec-lazy-sequence f xq = *f* (*list-of-lazy-sequence xq*)
by (*cases xq*) *auto*

definition *Lazy-Sequence* :: (*unit* ⇒ ('*a* × '*a lazy-sequence*) *option*) ⇒ '*a lazy-sequence*
where
Lazy-Sequence f = *lazy-sequence-of-list* (*case f* () *of*
None ⇒ []
| *Some* (*x*, *xq*) ⇒ *x* # *list-of-lazy-sequence xq*)

code-datatype *Lazy-Sequence*

declare [[*code drop*: *list-of-lazy-sequence rec-lazy-sequence case-lazy-sequence*]]

lemma *list-of-Lazy-Sequence* [*simp*]:
list-of-lazy-sequence (*Lazy-Sequence f*) = (*case f* () *of*
None ⇒ []
| *Some* (*x*, *xq*) ⇒ *x* # *list-of-lazy-sequence xq*)
by (*simp add*: *Lazy-Sequence-def*)

definition *yield* :: '*a lazy-sequence* ⇒ ('*a* × '*a lazy-sequence*) *option*
where
yield xq = (*case list-of-lazy-sequence xq of*
[] ⇒ *None*
| *x* # *xs* ⇒ *Some* (*x*, *lazy-sequence-of-list xs*))

lemma *yield-Seq* [*simp*, *code*]:
yield (*Lazy-Sequence f*) = *f* ()
by (*cases f* ()) (*simp-all add*: *yield-def split-def*)

lemma *case-yield-eq* [*simp*]: *case-option g h* (*yield xq*) =
case-list g (λ*x*. *curry h x* ∘ *lazy-sequence-of-list*) (*list-of-lazy-sequence xq*)
by (*cases list-of-lazy-sequence xq*) (*simp-all add*: *yield-def*)

lemma *equal-lazy-sequence-code* [*code*]:
HOL.equal xq yq = (*case* (*yield xq*, *yield yq*) *of*
(*None*, *None*) ⇒ *True*
| (*Some* (*x*, *xq'*), *Some* (*y*, *yq'*)) ⇒ *HOL.equal x y* ∧ *HOL.equal xq yq*
| - ⇒ *False*)
by (*simp-all add*: *lazy-sequence-eq-iff equal-eq split*: *list.splits*)

lemma [*code nbe*]:
HOL.equal (*x* :: *'a lazy-sequence*) *x* ⟷ *True*
by (*fact equal-refl*)

definition *empty* :: *'a lazy-sequence*
where
empty = *lazy-sequence-of-list* []

lemma *list-of-lazy-sequence-empty* [*simp*]:
list-of-lazy-sequence empty = []
by (*simp add*: *empty-def*)

lemma *empty-code* [*code*]:
empty = *Lazy-Sequence* (λ-. *None*)
by (*simp add*: *lazy-sequence-eq-iff*)

definition *single* :: *'a* ⇒ *'a lazy-sequence*
where
single x = *lazy-sequence-of-list* [*x*]

lemma *list-of-lazy-sequence-single* [*simp*]:
list-of-lazy-sequence (*single x*) = [*x*]
by (*simp add*: *single-def*)

lemma *single-code* [*code*]:
single x = *Lazy-Sequence* (λ-. *Some* (*x*, *empty*))
by (*simp add*: *lazy-sequence-eq-iff*)

definition *append* :: *'a lazy-sequence* ⇒ *'a lazy-sequence* ⇒ *'a lazy-sequence*
where
append xq yq = *lazy-sequence-of-list* (*list-of-lazy-sequence xq* @ *list-of-lazy-sequence yq*)

lemma *list-of-lazy-sequence-append* [*simp*]:
list-of-lazy-sequence (*append xq yq*) = *list-of-lazy-sequence xq* @ *list-of-lazy-sequence yq*
by (*simp add*: *append-def*)

lemma *append-code* [*code*]:
append xq yq = *Lazy-Sequence* (λ-. *case yield xq of*
None ⇒ *yield yq*
| *Some* (*x*, *xq'*) ⇒ *Some* (*x*, *append xq' yq*))
by (*simp-all add*: *lazy-sequence-eq-iff split*: *list.splits*)

definition *map* :: ('*a* ⇒ '*b*) ⇒ '*a lazy-sequence* ⇒ '*b lazy-sequence*
where
map f xq = *lazy-sequence-of-list* (*List.map f* (*list-of-lazy-sequence xq*))

lemma *list-of-lazy-sequence-map* [*simp*]:
list-of-lazy-sequence (*map f xq*) = *List.map f* (*list-of-lazy-sequence xq*)
by (*simp add*: *map-def*)

lemma *map-code* [*code*]:
map f xq =
Lazy-Sequence (λ-. *map-option* (λ(*x*, *xq'*). (*f x*, *map f xq'*)) (*yield xq*))
by (*simp-all add*: *lazy-sequence-eq-iff split*: *list.splits*)

definition *flat* :: '*a lazy-sequence lazy-sequence* ⇒ '*a lazy-sequence*
where
flat xqq = *lazy-sequence-of-list* (*concat* (*List.map list-of-lazy-sequence* (*list-of-lazy-sequence xqq*)))

lemma *list-of-lazy-sequence-flat* [*simp*]:
list-of-lazy-sequence (*flat xqq*) = *concat* (*List.map list-of-lazy-sequence* (*list-of-lazy-sequence xqq*))
by (*simp add*: *flat-def*)

lemma *flat-code* [*code*]:
flat xqq = *Lazy-Sequence* (λ-. *case yield xqq of*
None ⇒ *None*
| *Some* (*xq*, *xqq'*) ⇒ *yield* (*append xq* (*flat xqq'*)))
by (*simp add*: *lazy-sequence-eq-iff split*: *list.splits*)

definition *bind* :: '*a lazy-sequence* ⇒ ('*a* ⇒ '*b lazy-sequence*) ⇒ '*b lazy-sequence*
where
bind xq f = *flat* (*map f xq*)

definition *if-seq* :: *bool* ⇒ *unit lazy-sequence*
where
if-seq b = (*if b then single* () *else empty*)

definition *those* :: '*a option lazy-sequence* ⇒ '*a lazy-sequence option*
where
those xq = *map-option lazy-sequence-of-list* (*List.those* (*list-of-lazy-sequence xq*))

function *iterate-upto* :: (*natural* ⇒ '*a*) ⇒ *natural* ⇒ *natural* ⇒ '*a lazy-sequence*
where

iterate-upto f n m =
Lazy-Sequence (λ-. if n > m then None else Some (f n, iterate-upto f (n + 1) m))
by *pat-completeness auto*

termination by (*relation measure* (λ(*f*, *n*, *m*). *nat-of-natural* (*m* + 1 − *n*)))
(*auto simp add*: *less-natural-def*)

definition *not-seq* :: *unit lazy-sequence* ⇒ *unit lazy-sequence*
where
not-seq xq = (*case yield xq of*
None ⇒ *single* ()
| *Some* ((), *xq*) ⇒ *empty*)

76.2 Code setup

code-reflect *Lazy-Sequence*
datatypes *lazy-sequence* = *Lazy-Sequence*

ML ‹
signature LAZY-SEQUENCE =
sig
*datatype 'a lazy-sequence = Lazy-Sequence of (unit -> ('a * 'a Lazy-Sequence.lazy-sequence) option)*
val map: *('a -> 'b) -> 'a lazy-sequence -> 'b lazy-sequence*
val yield: *'a lazy-sequence -> ('a * 'a lazy-sequence) option*
val yieldn: *int -> 'a lazy-sequence -> 'a list * 'a lazy-sequence*
end;

structure Lazy-Sequence : LAZY-SEQUENCE =
struct

datatype lazy-sequence = datatype Lazy-Sequence.lazy-sequence;

fun map f = @{code Lazy-Sequence.map} f;

fun yield P = @{code Lazy-Sequence.yield} P;

fun yieldn k = Predicate.anamorph yield k;

end;
›

76.3 Generator Sequences

76.3.1 General lazy sequence operation

definition *product* :: *'a lazy-sequence* ⇒ *'b lazy-sequence* ⇒ (*'a* × *'b*) *lazy-sequence*
where
product s1 s2 = bind s1 (λa. bind s2 (λb. single (a, b)))

76.3.2 Small lazy typeclasses

class *small-lazy* =
  fixes *small-lazy* :: *natural* ⇒ *'a lazy-sequence*

instantiation *unit* :: *small-lazy*
begin

definition *small-lazy d* = *single* ()

instance ..

end

instantiation *int* :: *small-lazy*
begin

maybe optimise this expression -> append (single x) xs == cons x xs Performance difference?

function *small-lazy'* :: *int* ⇒ *int* ⇒ *int lazy-sequence*
where
  small-lazy' d i = (*if d* < *i then empty*
    else append (*single i*) (*small-lazy' d* (*i* + *1*)))
    by *pat-completeness auto*

termination
  by (*relation measure* (%(*d*, *i*). *nat* (*d* + *1* − *i*))) *auto*

definition
  small-lazy d = *small-lazy'* (*int* (*nat-of-natural d*)) (− (*int* (*nat-of-natural d*)))

instance ..

end

instantiation *prod* :: (*small-lazy*, *small-lazy*) *small-lazy*
begin

definition
  small-lazy d = *product* (*small-lazy d*) (*small-lazy d*)

instance ..

end

instantiation *list* :: (*small-lazy*) *small-lazy*
begin

fun *small-lazy-list* :: *natural* ⇒ *'a list lazy-sequence*
where

small-lazy-list d = append (single []) (if d > 0 then bind (product (small-lazy (d − 1)) (small-lazy (d − 1))) (λ(x, xs). single (x # xs)) else empty)

instance ..

end

76.4 With Hit Bound Value

assuming in negative context

type-synonym *'a hit-bound-lazy-sequence = 'a option lazy-sequence*

definition *hit-bound :: 'a hit-bound-lazy-sequence*
where
hit-bound = Lazy-Sequence (λ-. Some (None, empty))

lemma *list-of-lazy-sequence-hit-bound [simp]:*
list-of-lazy-sequence hit-bound = [None]
by *(simp add: hit-bound-def)*

definition *hb-single :: 'a ⇒ 'a hit-bound-lazy-sequence*
where
hb-single x = Lazy-Sequence (λ-. Some (Some x, empty))

definition *hb-map :: ('a ⇒ 'b) ⇒ 'a hit-bound-lazy-sequence ⇒ 'b hit-bound-lazy-sequence*
where
hb-map f xq = map (map-option f) xq

lemma *hb-map-code [code]:*
hb-map f xq = Lazy-Sequence (λ-. map-option (λ(x, xq'). (map-option f x, hb-map f xq')) (yield xq))
using *map-code [of map-option f xq]* **by** *(simp add: hb-map-def)*

definition *hb-flat :: 'a hit-bound-lazy-sequence hit-bound-lazy-sequence ⇒ 'a hit-bound-lazy-sequence*
where
hb-flat xqq = lazy-sequence-of-list (concat (List.map ((λx. case x of None ⇒ [None] | Some xs ⇒ xs) ∘ map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq)))

lemma *list-of-lazy-sequence-hb-flat [simp]:*
list-of-lazy-sequence (hb-flat xqq) = concat (List.map ((λx. case x of None ⇒ [None] | Some xs ⇒ xs) ∘ map-option list-of-lazy-sequence) (list-of-lazy-sequence xqq))
by *(simp add: hb-flat-def)*

lemma *hb-flat-code [code]:*
hb-flat xqq = Lazy-Sequence (λ-. case yield xqq of

```
    None ⇒ None
  | Some (xq, xqq') ⇒ yield
      (append (case xq of None ⇒ hit-bound | Some xq ⇒ xq) (hb-flat xqq')))
  by (simp add: lazy-sequence-eq-iff split: list.splits option.splits)

definition hb-bind :: 'a hit-bound-lazy-sequence ⇒ ('a ⇒ 'b hit-bound-lazy-sequence)
⇒ 'b hit-bound-lazy-sequence
where
  hb-bind xq f = hb-flat (hb-map f xq)

definition hb-if-seq :: bool ⇒ unit hit-bound-lazy-sequence
where
  hb-if-seq b = (if b then hb-single () else empty)

definition hb-not-seq :: unit hit-bound-lazy-sequence ⇒ unit lazy-sequence
where
  hb-not-seq xq = (case yield xq of
    None ⇒ single ()
  | Some (x, xq) ⇒ empty)

hide-const (open) yield empty single append flat map bind
  if-seq those iterate-upto not-seq product

hide-fact (open) yield-def empty-def single-def append-def flat-def map-def bind-def
  if-seq-def those-def not-seq-def product-def

end
```

77 Depth-Limited Sequences with failure element

```
theory Limited-Sequence
imports Lazy-Sequence
begin
```

77.1 Depth-Limited Sequence

```
type-synonym 'a dseq = natural ⇒ bool ⇒ 'a lazy-sequence option

definition empty :: 'a dseq
where
  empty = (λ- -. Some Lazy-Sequence.empty)

definition single :: 'a ⇒ 'a dseq
where
  single x = (λ- -. Some (Lazy-Sequence.single x))

definition eval :: 'a dseq ⇒ natural ⇒ bool ⇒ 'a lazy-sequence option
where
  [simp]: eval f i pol = f i pol
```

definition *yield* :: *'a dseq* ⇒ *natural* ⇒ *bool* ⇒ (*'a* × *'a dseq*) *option*
where
yield f i pol = (*case eval f i pol of*
None ⇒ *None*
| *Some s* ⇒ (*map-option* ∘ *apsnd*) (λ*r* - -. *Some r*) (*Lazy-Sequence.yield s*))

definition *map-seq* :: (*'a* ⇒ *'b dseq*) ⇒ *'a lazy-sequence* ⇒ *'b dseq*
where
map-seq f xq i pol = *map-option Lazy-Sequence.flat*
(*Lazy-Sequence.those* (*Lazy-Sequence.map* (λ*x*. *f x i pol*) *xq*))

lemma *map-seq-code* [*code*]:
map-seq f xq i pol = (*case Lazy-Sequence.yield xq of*
None ⇒ *Some Lazy-Sequence.empty*
| *Some* (*x*, *xq'*) ⇒ (*case eval* (*f x*) *i pol of*
None ⇒ *None*
| *Some yq* ⇒ (*case map-seq f xq' i pol of*
None ⇒ *None*
| *Some zq* ⇒ *Some* (*Lazy-Sequence.append yq zq*))))
by (*cases xq*)
(*auto simp add*: *map-seq-def Lazy-Sequence.those-def lazy-sequence-eq-iff split*: *list.splits option.splits*)

definition *bind* :: *'a dseq* ⇒ (*'a* ⇒ *'b dseq*) ⇒ *'b dseq*
where
bind x f = (λ*i pol*.
if i = *0 then*
(*if pol then Some Lazy-Sequence.empty else None*)
else
(*case x* (*i* − *1*) *pol of*
None ⇒ *None*
| *Some xq* ⇒ *map-seq f xq i pol*))

definition *union* :: *'a dseq* ⇒ *'a dseq* ⇒ *'a dseq*
where
union x y = (λ*i pol*. *case* (*x i pol*, *y i pol*) *of*
(*Some xq*, *Some yq*) ⇒ *Some* (*Lazy-Sequence.append xq yq*)
| - ⇒ *None*)

definition *if-seq* :: *bool* ⇒ *unit dseq*
where
if-seq b = (*if b then single* () *else empty*)

definition *not-seq* :: *unit dseq* ⇒ *unit dseq*
where
not-seq x = (λ*i pol*. *case x i* (¬ *pol*) *of*
None ⇒ *Some Lazy-Sequence.empty*
| *Some xq* ⇒ (*case Lazy-Sequence.yield xq of*

```
    None ⇒ Some (Lazy-Sequence.single ())
  | Some - ⇒ Some (Lazy-Sequence.empty)))

definition map :: ('a ⇒ 'b) ⇒ 'a dseq ⇒ 'b dseq
where
  map f g = (λi pol. case g i pol of
    None ⇒ None
  | Some xq ⇒ Some (Lazy-Sequence.map f xq))
```

77.2 Positive Depth-Limited Sequence

```
type-synonym 'a pos-dseq = natural ⇒ 'a Lazy-Sequence.lazy-sequence

definition pos-empty :: 'a pos-dseq
where
  pos-empty = (λi. Lazy-Sequence.empty)

definition pos-single :: 'a ⇒ 'a pos-dseq
where
  pos-single x = (λi. Lazy-Sequence.single x)

definition pos-bind :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
where
  pos-bind x f = (λi. Lazy-Sequence.bind (x i) (λa. f a i))

definition pos-decr-bind :: 'a pos-dseq ⇒ ('a ⇒ 'b pos-dseq) ⇒ 'b pos-dseq
where
  pos-decr-bind x f = (λi.
    if i = 0 then
      Lazy-Sequence.empty
    else
      Lazy-Sequence.bind (x (i − 1)) (λa. f a i))

definition pos-union :: 'a pos-dseq ⇒ 'a pos-dseq ⇒ 'a pos-dseq
where
  pos-union xq yq = (λi. Lazy-Sequence.append (xq i) (yq i))

definition pos-if-seq :: bool ⇒ unit pos-dseq
where
  pos-if-seq b = (if b then pos-single () else pos-empty)

definition pos-iterate-upto :: (natural ⇒ 'a) ⇒ natural ⇒ natural ⇒ 'a pos-dseq
where
  pos-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto f n m)

definition pos-map :: ('a ⇒ 'b) ⇒ 'a pos-dseq ⇒ 'b pos-dseq
where
  pos-map f xq = (λi. Lazy-Sequence.map f (xq i))
```

77.3 Negative Depth-Limited Sequence

type-synonym *'a neg-dseq = natural ⇒ 'a Lazy-Sequence.hit-bound-lazy-sequence*

definition *neg-empty :: 'a neg-dseq*
where
neg-empty = (λi. Lazy-Sequence.empty)

definition *neg-single :: 'a ⇒ 'a neg-dseq*
where
neg-single x = (λi. Lazy-Sequence.hb-single x)

definition *neg-bind :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq*
where
neg-bind x f = (λi. hb-bind (x i) (λa. f a i))

definition *neg-decr-bind :: 'a neg-dseq ⇒ ('a ⇒ 'b neg-dseq) ⇒ 'b neg-dseq*
where
neg-decr-bind x f = (λi.
if i = 0 then
Lazy-Sequence.hit-bound
else
hb-bind (x (i − 1)) (λa. f a i))

definition *neg-union :: 'a neg-dseq ⇒ 'a neg-dseq ⇒ 'a neg-dseq*
where
neg-union x y = (λi. Lazy-Sequence.append (x i) (y i))

definition *neg-if-seq :: bool ⇒ unit neg-dseq*
where
neg-if-seq b = (if b then neg-single () else neg-empty)

definition *neg-iterate-upto*
where
neg-iterate-upto f n m = (λi. Lazy-Sequence.iterate-upto (λi. Some (f i)) n m)

definition *neg-map :: ('a ⇒ 'b) ⇒ 'a neg-dseq ⇒ 'b neg-dseq*
where
neg-map f xq = (λi. Lazy-Sequence.hb-map f (xq i))

77.4 Negation

definition *pos-not-seq :: unit neg-dseq ⇒ unit pos-dseq*
where
pos-not-seq xq = (λi. Lazy-Sequence.hb-not-seq (xq (3 ∗ i)))

definition *neg-not-seq :: unit pos-dseq ⇒ unit neg-dseq*
where
neg-not-seq x = (λi. case Lazy-Sequence.yield (x i) of
None ⇒ Lazy-Sequence.hb-single ()

| *Some* ((), *xq*) ⇒ *Lazy-Sequence.empty*)

ML ‹
signature LIMITED-SEQUENCE =
sig
type 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence option
val map : ('a -> 'b) -> 'a dseq -> 'b dseq
*val yield : 'a dseq -> Code-Numeral.natural -> bool -> ('a * 'a dseq) option*
*val yieldn : int -> 'a dseq -> Code-Numeral.natural -> bool -> 'a list * 'a dseq*
end;

structure Limited-Sequence : LIMITED-SEQUENCE =
struct

type 'a dseq = Code-Numeral.natural -> bool -> 'a Lazy-Sequence.lazy-sequence option

fun map f = @{code Limited-Sequence.map} f;

fun yield f = @{code Limited-Sequence.yield} f;

fun yieldn n f i pol = (case f i pol of
NONE => ([], fn - => fn - => NONE)
| SOME s => let val (xs, s') = Lazy-Sequence.yieldn n s in (xs, fn - => fn - => SOME s') end);

end;
›

code-reserved
(*Eval*) *Limited-Sequence*

hide-const (**open**) *yield empty single eval map-seq bind union if-seq not-seq map*
pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-seq pos-iterate-upto pos-not-seq pos-map
neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-seq neg-iterate-upto neg-not-seq neg-map

hide-fact (**open**) *yield-def empty-def single-def eval-def map-seq-def bind-def union-def*
if-seq-def not-seq-def map-def
pos-empty-def pos-single-def pos-bind-def pos-union-def pos-if-seq-def pos-iterate-upto-def pos-not-seq-def pos-map-def
neg-empty-def neg-single-def neg-bind-def neg-union-def neg-if-seq-def neg-iterate-upto-def neg-not-seq-def neg-map-def

end

78 Term evaluation using the generic code generator

theory *Code-Evaluation*
imports *Typerep Limited-Sequence*
keywords *value* :: *diag*
begin

78.1 Term representation

78.1.1 Terms and class *term-of*

datatype (*plugins only*: *extraction*) *term* = *dummy-term*

definition *Const* :: *String.literal* ⇒ *typerep* ⇒ *term* **where**
Const - - = *dummy-term*

definition *App* :: *term* ⇒ *term* ⇒ *term* **where**
App - - = *dummy-term*

definition *Abs* :: *String.literal* ⇒ *typerep* ⇒ *term* ⇒ *term* **where**
Abs - - - = *dummy-term*

definition *Free* :: *String.literal* ⇒ *typerep* ⇒ *term* **where**
Free - - = *dummy-term*

code-datatype *Const App Abs Free*

class *term-of* = *typerep* +
fixes *term-of* :: *'a* ⇒ *term*

lemma *term-of-anything*: *term-of x* ≡ *t*
by (*rule eq-reflection*) (*cases term-of x*, *cases t*, *simp*)

definition *valapp* :: (*'a* ⇒ *'b*) × (*unit* ⇒ *term*)
⇒ *'a* × (*unit* ⇒ *term*) ⇒ *'b* × (*unit* ⇒ *term*) **where**
valapp f x = (*fst f* (*fst x*), λ*u*. *App* (*snd f* ()) (*snd x* ()))

lemma *valapp-code* [*code*, *code-unfold*]:
valapp (*f*, *tf*) (*x*, *tx*) = (*f x*, λ*u*. *App* (*tf* ()) (*tx* ()))
by (*simp only*: *valapp-def fst-conv snd-conv*)

78.1.2 Syntax

definition *termify* :: *'a* ⇒ *term* **where**
[*code drop*]: *termify x* = *dummy-term*

abbreviation *valtermify* :: *'a* ⇒ *'a* × (*unit* ⇒ *term*) **where**
valtermify x ≡ (*x*, λ*u*. *termify x*)

bundle *term-syntax*
begin
notation *App* (**infixl** ‹<·>› *70*) **and** *valapp* (**infixl** ‹{·}› *70*)
end

78.2 Tools setup and evaluation

context
begin

qualified definition *TERM-OF* :: *'a*::*term-of itself*
where
TERM-OF = *snd* (*Code-Evaluation.term-of* :: *'a* ⇒ -, *TYPE*(*'a*))

qualified definition *TERM-OF-EQUAL* :: *'a*::*term-of itself*
where
TERM-OF-EQUAL = *snd* (λ(*a*::*'a*). (*Code-Evaluation.term-of a*, *HOL.eq a*), *TYPE*(*'a*))

end

lemma *eq-eq-TrueD*:
fixes *x y* :: *'a*::{}
assumes (*x* ≡ *y*) ≡ *Trueprop True*
shows *x* ≡ *y*
using *assms* **by** *simp*

code-printing
type-constructor *term* ⇀ (*Eval*) *Term.term*
| **constant** *Const* ⇀ (*Eval*) *Term.Const*/ ((-), (-))
| **constant** *App* ⇀ (*Eval*) *Term.*$/ ((-), (-))
| **constant** *Abs* ⇀ (*Eval*) *Term.Abs*/ ((-), (-), (-))
| **constant** *Free* ⇀ (*Eval*) *Term.Free*/ ((-), (-))

ML-file ‹*Tools*/*code-evaluation.ML*›

code-reserved
(*Eval*) *Code-Evaluation*

ML-file ‹~~/*src*/*HOL*/*Tools*/*value-command.ML*›

78.3 Dedicated *term-of* instances

instantiation *fun* :: (*typerep*, *typerep*) *term-of*
begin

definition
term-of (*f* :: *'a* ⇒ *'b*) =
Const (*STR ''Pure.dummy-pattern''*)

(*Typerep.Typerep* (*STR ''fun''*) [*Typerep.typerep TYPE*('*a*), *Typerep.typerep TYPE*('*b*)])

instance ..

end

declare [[*code drop*:
term-of :: *typerep* ⇒ -
term-of :: *term* ⇒ -
term-of :: *integer* ⇒ -
term-of :: *String.literal* ⇒ -
term-of :: - *Predicate.pred* ⇒ -
term-of :: - *Predicate.seq* ⇒ -]]

code-printing
constant *term-of* :: *integer* ⇒ *term* ⇀ (*Eval*) *HOLogic.mk'-number*/ *HOLogic.code'-integerT*
| **constant** *term-of* :: *String.literal* ⇒ *term* ⇀ (*Eval*) *HOLogic.mk'-literal*

lemma *term-of-integer* [*unfolded typerep-fun-def typerep-num-def typerep-integer-def*, *code*]:
term-of (*i* :: *integer*) =
(*if i* > *0 then*
App (*Const* (*STR ''Num.numeral-class.numeral''*) (*TYPEREP*(*num* ⇒ *integer*)))
(*term-of* (*num-of-integer i*))
else if i = *0 then Const* (*STR ''Groups.zero-class.zero''*) *TYPEREP*(*integer*)
else
App (*Const* (*STR ''Groups.uminus-class.uminus''*) *TYPEREP*(*integer* ⇒ *integer*))
(*term-of* (− *i*)))
by (*rule term-of-anything* [*THEN meta-eq-to-obj-eq*])

code-reserved
(*Eval*) *HOLogic*

78.4 Generic reification

ML-file ‹~~/*src*/*HOL*/*Tools*/*reification.ML*›

78.5 Diagnostic

definition *tracing* :: *String.literal* ⇒ '*a* ⇒ '*a* **where**
tracing s x = *x*

code-printing
constant *tracing* :: *String.literal* => '*a* => '*a* ⇀ (*Eval*) *Code'-Evaluation.tracing*

hide-const *dummy-term valapp*

hide-const (**open**) *Const App Abs Free termify valtermify term-of tracing*

end

79 A simple counterexample generator performing random testing

theory *Quickcheck-Random*
imports *Random Code-Evaluation Enum*
begin

setup ‹*Code-Target.add-derived-target* (*Quickcheck*, [(*Code-Runtime.target*, *I*)])›

79.1 Catching Match exceptions

axiomatization *catch-match* :: *'a => 'a => 'a*

code-printing
  constant *catch-match* ⇀ (*Quickcheck*) ((-) *handle Match =>* -)

code-reserved
  (*Quickcheck*) *Match*

79.2 The *random* class

class *random* = *typerep* +
  fixes *random* :: *natural* ⇒ *Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*

79.3 Fundamental and numeric types

instantiation *bool* :: *random*
begin

context
  includes *state-combinator-syntax*
begin

definition
  random i = *Random.range 2* ○→
   (λ*k*. *Pair* (*if k = 0 then Code-Evaluation.valtermify False else Code-Evaluation.valtermify True*))

instance ..

end

end

instantiation *itself* :: (*typerep*) *random*

begin

definition
random-itself :: *natural* ⇒ *Random.seed* ⇒ (*'a itself* × (*unit* ⇒ *term*)) × *Random.seed*
where *random-itself* - = *Pair* (*Code-Evaluation.valtermify TYPE('a)*)

instance ..

end

instantiation *char* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
random - = *Random.select* (*Enum.enum* :: *char list*) ○→ (λ*c*. *Pair* (*c*, λ*u*. *Code-Evaluation.term-of c*))

instance ..

end

end

instantiation *String.literal* :: *random*
begin

definition
random - = *Pair* (*STR ''''*, λ*u*. *Code-Evaluation.term-of* (*STR ''''*))

instance ..

end

instantiation *nat* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition *random-nat* :: *natural* ⇒ *Random.seed*
⇒ (*nat* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-nat i = *Random.range* (*i* + *1*) ○→ (λ*k*. *Pair* (

let n = nat-of-natural k
in (n, λ-. Code-Evaluation.term-of n)))

instance ..

end

end

instantiation *int* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
*random i = Random.range (2 * i + 1) ○→ (λk. Pair (*
let j = (if k ≥ i then int (nat-of-natural (k − i)) else − (int (nat-of-natural (i − k))))
in (j, λ-. Code-Evaluation.term-of j)))

instance ..

end

end

instantiation *natural* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition *random-natural* :: *natural* ⇒ *Random.seed*
⇒ (*natural* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-natural i = Random.range (i + 1) ○→ (λn. Pair (n, λ-. Code-Evaluation.term-of n))

instance ..

end

end

instantiation *integer* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition *random-integer* :: *natural* ⇒ *Random.seed*
⇒ (*integer* × (*unit* ⇒ *Code-Evaluation.term*)) × *Random.seed*
where
random-integer i = *Random.range* (*2* ∗ *i* + *1*) ∘→ (λ*k*. *Pair* (
let j = (*if k* ≥ *i then integer-of-natural* (*k* − *i*) *else* − (*integer-of-natural* (*i* − *k*)))
in (*j*, λ-. *Code-Evaluation.term-of j*)))

instance ..

end

end

79.4 Complex generators

Towards *'a* ⇒ *'b*

axiomatization *random-fun-aux* :: *typerep* ⇒ *typerep* ⇒ (*'a* ⇒ *'a* ⇒ *bool*) ⇒ (*'a* ⇒ *term*)
⇒ (*Random.seed* ⇒ (*'b* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ (*Random.seed* ⇒ *Random.seed* × *Random.seed*)
⇒ *Random.seed* ⇒ ((*'a* ⇒ *'b*) × (*unit* ⇒ *term*)) × *Random.seed*

definition *random-fun-lift* :: (*Random.seed* ⇒ (*'b* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *Random.seed* ⇒ ((*'a*::*term-of* ⇒ *'b*::*typerep*) × (*unit* ⇒ *term*)) × *Random.seed*
where
random-fun-lift f =
random-fun-aux TYPEREP(*'a*) *TYPEREP*(*'b*) (=) *Code-Evaluation.term-of f Random.split-seed*

instantiation *fun* :: ({*equal*, *term-of*}, *random*) *random*
begin

definition
random-fun :: *natural* ⇒ *Random.seed* ⇒ ((*'a* ⇒ *'b*) × (*unit* ⇒ *term*)) × *Random.seed*
where *random i* = *random-fun-lift* (*random i*)

instance ..

end

Towards type copies and datatypes

context
includes *state-combinator-syntax*
begin

definition *collapse* :: $('a \Rightarrow ('a \Rightarrow 'b \times 'a) \times 'a) \Rightarrow 'a \Rightarrow 'b \times 'a$
where *collapse f* = $(f \circ\!\!\rightarrow id)$

end

definition *beyond* :: *natural* ⇒ *natural* ⇒ *natural*
where *beyond k l* = (*if l* > *k then l else 0*)

lemma *beyond-zero*: *beyond k 0* = *0*
by (*simp add*: *beyond-def*)

context
includes *term-syntax*
begin

definition [*code-unfold*]:
valterm-emptyset = *Code-Evaluation.valtermify* ({} :: ('*a* :: *typerep*) *set*)

definition [*code-unfold*]:
valtermify-insert x s = *Code-Evaluation.valtermify insert* {·} (*x* :: ('*a* :: *typerep* ∗ -)) {·} *s*

end

instantiation *set* :: (*random*) *random*
begin

context
includes *state-combinator-syntax*
begin

fun *random-aux-set*
where
random-aux-set 0 j = *collapse* (*Random.select-weight* [(*1*, *Pair valterm-emptyset*)])
| *random-aux-set* (*Code-Numeral.Suc i*) *j* =
collapse (*Random.select-weight*
[(*1*, *Pair valterm-emptyset*),
(*Code-Numeral.Suc i*,
random j ∘→ (%*x*. *random-aux-set i j* ∘→ (%*s*. *Pair* (*valtermify-insert x s*))))])

lemma [*code*]:
random-aux-set i j =
collapse (*Random.select-weight* [(*1*, *Pair valterm-emptyset*),
(*i*, *random j* ∘→ (%*x*. *random-aux-set* (*i* − *1*) *j* ∘→ (%*s*. *Pair* (*valtermify-insert*

x s))))])
proof (*induct i rule*: *natural.induct*)
case *zero*
show *?case* **by** (*subst select-weight-drop-zero* [*symmetric*])
(*simp add*: *random-aux-set.simps* [*simplified*] *less-natural-def*)
next
case (*Suc i*)
show *?case* **by** (*simp only*: *random-aux-set.simps*(*2*) [*of i*] *Suc-natural-minus-one*)
qed

definition *random-set i* = *random-aux-set i i*

instance ..

end

end

lemma *random-aux-rec*:
fixes *random-aux* :: *natural* ⇒ *'a*
assumes *random-aux 0* = *rhs 0*
and ⋀*k*. *random-aux* (*Code-Numeral.Suc k*) = *rhs* (*Code-Numeral.Suc k*)
shows *random-aux k* = *rhs k*
using *assms* **by** (*rule natural.induct*)

79.5 Deriving random generators for datatypes

ML-file ‹*Tools/Quickcheck/quickcheck-common.ML*›
ML-file ‹*Tools/Quickcheck/random-generators.ML*›

79.6 Code setup

code-printing
constant *random-fun-aux* ⇀ (*Quickcheck*) *Random'-Generators.random'-fun*
— With enough criminal energy this can be abused to derive *False*; for this reason we use a distinguished target *Quickcheck* not spoiling the regular trusted code generation

code-reserved
(*Quickcheck*) *Random-Generators*

hide-const (**open**) *catch-match random collapse beyond random-fun-aux random-fun-lift*

hide-fact (**open**) *collapse-def beyond-def random-fun-lift-def*

end

80 The Random-Predicate Monad

theory *Random-Pred*
imports *Quickcheck-Random*
begin

fun *iter'* :: *'a itself* ⇒ *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ (*'a*::*random*) *Predicate.pred*
where
iter' T nrandom sz seed = (*if nrandom* = *0 then bot-class.bot else*
let ((*x*, -), *seed'*) = *Quickcheck-Random.random sz seed*
in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter' T* (*nrandom* − *1*) *sz seed'*)))

definition *iter* :: *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ (*'a*::*random*) *Predicate.pred*
where
iter nrandom sz seed = *iter'* (*TYPE*(*'a*)) *nrandom sz seed*

lemma [*code*]:
iter nrandom sz seed = (*if nrandom* = *0 then bot-class.bot else*
let ((*x*, -), *seed'*) = *Quickcheck-Random.random sz seed*
in Predicate.Seq (%*u*. *Predicate.Insert x* (*iter* (*nrandom* − *1*) *sz seed'*)))
unfolding *iter-def iter'.simps* [*of* - *nrandom*] **..**

type-synonym *'a random-pred* = *Random.seed* ⇒ (*'a Predicate.pred* × *Random.seed*)

definition *empty* :: *'a random-pred*
where *empty* = *Pair bot*

definition *single* :: *'a* => *'a random-pred*
where *single x* = *Pair* (*Predicate.single x*)

definition *bind* :: *'a random-pred* ⇒ (*'a* ⇒ *'b random-pred*) ⇒ *'b random-pred*
where
bind R f = (λ*s*. *let*
(*P*, *s'*) = *R s*;
(*s1*, *s2*) = *Random.split-seed s'*
in (*Predicate.bind P* (%*a*. *fst* (*f a s1*)), *s2*))

definition *union* :: *'a random-pred* ⇒ *'a random-pred* ⇒ *'a random-pred*
where
union R1 R2 = (λ*s*. *let*
(*P1*, *s'*) = *R1 s*; (*P2*, *s''*) = *R2 s'*
in (*sup-class.sup P1 P2*, *s''*))

definition *if-randompred* :: *bool* ⇒ *unit random-pred*
where
if-randompred b = (*if b then single* () *else empty*)

definition *iterate-upto* :: (*natural* ⇒ *'a*) => *natural* ⇒ *natural* ⇒ *'a random-pred*
where
iterate-upto f n m = *Pair* (*Predicate.iterate-upto f n m*)

definition *not-randompred* :: *unit random-pred* ⇒ *unit random-pred*
where
not-randompred P = (λ*s*. *let*
(*P'*, *s'*) = *P s*
in if Predicate.eval P' () *then* (*Orderings.bot*, *s'*) *else* (*Predicate.single* (), *s'*))

definition *Random* :: (*Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*) ⇒ *'a random-pred*
where *Random g* = *scomp g* (*Pair* ∘ (*Predicate.single* ∘ *fst*))

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a random-pred* ⇒ *'b random-pred*
where *map f P* = *bind P* (*single* ∘ *f*)

hide-const (**open**) *iter'* *iter empty single bind union if-randompred*
iterate-upto not-randompred Random map

hide-fact *iter'.simps*

hide-fact (**open**) *iter-def empty-def single-def bind-def union-def*
if-randompred-def iterate-upto-def not-randompred-def Random-def map-def

end

81 Various kind of sequences inside the random monad

theory *Random-Sequence*
imports *Random-Pred*
begin

type-synonym *'a random-dseq* = *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ (*'a Limited-Sequence.dseq* × *Random.seed*)

definition *empty* :: *'a random-dseq*
where
empty = (%*nrandom size*. *Pair* (*Limited-Sequence.empty*))

definition *single* :: *'a* => *'a random-dseq*
where
single x = (%*nrandom size*. *Pair* (*Limited-Sequence.single x*))

definition *bind* :: *'a random-dseq* => (*'a* ⇒ *'b random-dseq*) ⇒ *'b random-dseq*
where
bind R f = (λ*nrandom size s*. *let*

```
    (P, s') = R nrandom size s;
    (s1, s2) = Random.split-seed s'
  in (Limited-Sequence.bind P (%a. fst (f a nrandom size s1)), s2))

definition union :: 'a random-dseq => 'a random-dseq => 'a random-dseq
where
  union R1 R2 = (λnrandom size s. let
    (S1, s') = R1 nrandom size s; (S2, s'') = R2 nrandom size s'
  in (Limited-Sequence.union S1 S2, s''))

definition if-random-dseq :: bool => unit random-dseq
where
  if-random-dseq b = (if b then single () else empty)

definition not-random-dseq :: unit random-dseq => unit random-dseq
where
  not-random-dseq R = (λnrandom size s. let
    (S, s') = R nrandom size s
   in (Limited-Sequence.not-seq S, s'))

definition map :: ('a => 'b) => 'a random-dseq => 'b random-dseq
where
  map f P = bind P (single ∘ f)

fun Random :: (natural ⇒ Random.seed ⇒ (('a × (unit ⇒ term)) × Random.seed))
⇒ 'a random-dseq
where
  Random g nrandom = (%size. if nrandom <= 0 then (Pair Limited-Sequence.empty)
else
      (scomp (g size) (%r. scomp (Random g (nrandom − 1) size) (%rs. Pair
(Limited-Sequence.union (Limited-Sequence.single (fst r)) rs)))))


type-synonym 'a pos-random-dseq = natural ⇒ natural ⇒ Random.seed ⇒ 'a
Limited-Sequence.pos-dseq

definition pos-empty :: 'a pos-random-dseq
where
  pos-empty = (%nrandom size seed. Limited-Sequence.pos-empty)

definition pos-single :: 'a => 'a pos-random-dseq
where
  pos-single x = (%nrandom size seed. Limited-Sequence.pos-single x)

definition pos-bind :: 'a pos-random-dseq => ('a ⇒ 'b pos-random-dseq) ⇒ 'b
pos-random-dseq
where
  pos-bind R f = (λnrandom size seed. Limited-Sequence.pos-bind (R nrandom size
seed) (%a. f a nrandom size seed))
```

definition *pos-decr-bind* :: *'a pos-random-dseq* => (*'a* ⇒ *'b pos-random-dseq*) ⇒ *'b pos-random-dseq*
where
pos-decr-bind R f = (λ*nrandom size seed. Limited-Sequence.pos-decr-bind* (*R nrandom size seed*) (%*a. f a nrandom size seed*))

definition *pos-union* :: *'a pos-random-dseq* => *'a pos-random-dseq* => *'a pos-random-dseq*
where
pos-union R1 R2 = (λ*nrandom size seed. Limited-Sequence.pos-union* (*R1 nrandom size seed*) (*R2 nrandom size seed*))

definition *pos-if-random-dseq* :: *bool* => *unit pos-random-dseq*
where
pos-if-random-dseq b = (*if b then pos-single* () *else pos-empty*)

definition *pos-iterate-upto* :: (*natural* => *'a*) => *natural* => *natural* => *'a pos-random-dseq*
where
pos-iterate-upto f n m = (λ*nrandom size seed. Limited-Sequence.pos-iterate-upto f n m*)

definition *pos-map* :: (*'a* => *'b*) => *'a pos-random-dseq* => *'b pos-random-dseq*
where
pos-map f P = *pos-bind P* (*pos-single* ∘ *f*)

fun *iter* :: (*Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *natural* ⇒ *Random.seed* ⇒ *'a Lazy-Sequence.lazy-sequence*
where
iter random nrandom seed =
(*if nrandom* = *0 then Lazy-Sequence.empty else Lazy-Sequence.Lazy-Sequence* (%*u. let* ((*x*, -), *seed'*) = *random seed in Some* (*x*, *iter random* (*nrandom* − *1*) *seed'*)))

definition *pos-Random* :: (*natural* ⇒ *Random.seed* ⇒ (*'a* × (*unit* ⇒ *term*)) × *Random.seed*)
⇒ *'a pos-random-dseq*
where
pos-Random g = (%*nrandom size seed depth. iter* (*g size*) *nrandom seed*)

type-synonym *'a neg-random-dseq* = *natural* ⇒ *natural* ⇒ *Random.seed* ⇒ *'a Limited-Sequence.neg-dseq*

definition *neg-empty* :: *'a neg-random-dseq*
where
neg-empty = (%*nrandom size seed. Limited-Sequence.neg-empty*)

definition *neg-single* :: *'a* => *'a neg-random-dseq*

where
neg-single x = (%nrandom size seed. Limited-Sequence.neg-single x)

definition *neg-bind :: 'a neg-random-dseq => ('a ⇒ 'b neg-random-dseq) ⇒ 'b neg-random-dseq*
where
neg-bind R f = (λnrandom size seed. Limited-Sequence.neg-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-decr-bind :: 'a neg-random-dseq => ('a ⇒ 'b neg-random-dseq) ⇒ 'b neg-random-dseq*
where
neg-decr-bind R f = (λnrandom size seed. Limited-Sequence.neg-decr-bind (R nrandom size seed) (%a. f a nrandom size seed))

definition *neg-union :: 'a neg-random-dseq => 'a neg-random-dseq => 'a neg-random-dseq*
where
neg-union R1 R2 = (λnrandom size seed. Limited-Sequence.neg-union (R1 nrandom size seed) (R2 nrandom size seed))

definition *neg-if-random-dseq :: bool => unit neg-random-dseq*
where
neg-if-random-dseq b = (if b then neg-single () else neg-empty)

definition *neg-iterate-upto :: (natural => 'a) => natural => natural => 'a neg-random-dseq*
where
neg-iterate-upto f n m = (λnrandom size seed. Limited-Sequence.neg-iterate-upto f n m)

definition *neg-not-random-dseq :: unit pos-random-dseq => unit neg-random-dseq*
where
neg-not-random-dseq R = (λnrandom size seed. Limited-Sequence.neg-not-seq (R nrandom size seed))

definition *neg-map :: ('a => 'b) => 'a neg-random-dseq => 'b neg-random-dseq*
where
neg-map f P = neg-bind P (neg-single ∘ f)

definition *pos-not-random-dseq :: unit neg-random-dseq => unit pos-random-dseq*
where
pos-not-random-dseq R = (λnrandom size seed. Limited-Sequence.pos-not-seq (R nrandom size seed))

hide-const (**open**)
empty single bind union if-random-dseq not-random-dseq map Random pos-empty pos-single pos-bind pos-decr-bind pos-union pos-if-random-dseq pos-iterate-upto pos-not-random-dseq pos-map iter pos-Random

neg-empty neg-single neg-bind neg-decr-bind neg-union neg-if-random-dseq neg-iterate-upto neg-not-random-dseq neg-map

hide-fact (**open**) *empty-def single-def bind-def union-def if-random-dseq-def not-random-dseq-def map-def Random.simps*
pos-empty-def pos-single-def pos-bind-def pos-decr-bind-def pos-union-def pos-if-random-dseq-def pos-iterate-upto-def pos-not-random-dseq-def pos-map-def iter.simps pos-Random-def neg-empty-def neg-single-def neg-bind-def neg-decr-bind-def neg-union-def neg-if-random-dseq-def neg-iterate-upto-def neg-not-random-dseq-def neg-map-def

end

82 A simple counterexample generator performing exhaustive testing

theory *Quickcheck-Exhaustive*
imports *Quickcheck-Random*
keywords *quickcheck-generator* :: *thy-decl*
begin

82.1 Basic operations for exhaustive generators

definition *orelse* :: *'a option ⇒ 'a option ⇒ 'a option* (**infixr** ‹*orelse*› *55*)
where [*code-unfold*]: *x orelse y = (case x of Some x' ⇒ Some x' | None ⇒ y)*

82.2 Exhaustive generator type classes

class *exhaustive = term-of +*
fixes *exhaustive* :: *('a ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option*

class *full-exhaustive = term-of +*
fixes *full-exhaustive* ::
('a × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ natural ⇒ (bool × term list) option

instantiation *natural* :: *full-exhaustive*
begin

function *full-exhaustive-natural'* ::
(natural × (unit ⇒ term) ⇒ (bool × term list) option) ⇒
natural ⇒ natural ⇒ (bool × term list) option
where *full-exhaustive-natural' f d i =*
(if d < i then None
else (f (i, λ-. Code-Evaluation.term-of i)) orelse (full-exhaustive-natural' f d (i + 1)))
by *pat-completeness auto*

termination

by (*relation measure* (λ(-, *d*, *i*). *nat-of-natural* (*d* + *1* − *i*))) (*auto simp add*: *less-natural-def*)

definition *full-exhaustive f d* = *full-exhaustive-natural′ f d 0*

instance ..

end

instantiation *natural* :: *exhaustive*
begin

function *exhaustive-natural′* ::
(*natural* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-natural′ f d i* =
(*if d* < *i then None*
else (*f i orelse exhaustive-natural′ f d* (*i* + *1*)))
by *pat-completeness auto*

termination
by (*relation measure* (λ(-, *d*, *i*). *nat-of-natural* (*d* + *1* − *i*))) (*auto simp add*: *less-natural-def*)

definition *exhaustive f d* = *exhaustive-natural′ f d 0*

instance ..

end

instantiation *integer* :: *exhaustive*
begin

function *exhaustive-integer′* ::
(*integer* ⇒ (*bool* × *term list*) *option*) ⇒ *integer* ⇒ *integer* ⇒ (*bool* × *term list*) *option*
where *exhaustive-integer′ f d i* =
(*if d* < *i then None else* (*f i orelse exhaustive-integer′ f d* (*i* + *1*)))
by *pat-completeness auto*

termination
by (*relation measure* (λ(-, *d*, *i*). *nat-of-integer* (*d* + *1* − *i*)))
(*auto simp add*: *less-integer-def nat-of-integer-def*)

definition *exhaustive f d* = *exhaustive-integer′ f* (*integer-of-natural d*) (− (*integer-of-natural d*))

instance ..

end

instantiation *integer* :: *full-exhaustive*
begin

function *full-exhaustive-integer'* ::
(*integer* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
integer ⇒ *integer* ⇒ (*bool* × *term list*) *option*
where *full-exhaustive-integer' f d i* =
(*if d* < *i then None*
else
(*case f* (*i*, λ-. *Code-Evaluation.term-of i*) *of*
Some t ⇒ *Some t*
| *None* ⇒ *full-exhaustive-integer' f d* (*i* + *1*)))
by *pat-completeness auto*

termination
by (*relation measure* (λ(-, *d*, *i*). *nat-of-integer* (*d* + *1* − *i*)))
(*auto simp add*: *less-integer-def nat-of-integer-def*)

definition *full-exhaustive f d* =
full-exhaustive-integer' f (*integer-of-natural d*) (− (*integer-of-natural d*))

instance ..

end

instantiation *nat* :: *exhaustive*
begin

definition *exhaustive f d* = *exhaustive* (λ*x*. *f* (*nat-of-natural x*)) *d*

instance ..

end

instantiation *nat* :: *full-exhaustive*
begin

definition *full-exhaustive f d* =
full-exhaustive (λ(*x*, *xt*). *f* (*nat-of-natural x*, λ-. *Code-Evaluation.term-of* (*nat-of-natural x*))) *d*

instance ..

end

instantiation *int* :: *exhaustive*
begin

function *exhaustive-int'* ::
  (*int* ⇒ (*bool* × *term list*) *option*) ⇒ *int* ⇒ *int* ⇒ (*bool* × *term list*) *option*
 where *exhaustive-int' f d i* =
  (*if d* < *i then None else* (*f i orelse exhaustive-int' f d* (*i* + *1*)))
by *pat-completeness auto*

termination
 by (*relation measure* (λ(-, *d*, *i*). *nat* (*d* + *1* − *i*))) *auto*

definition *exhaustive f d* =
 exhaustive-int' f (*int-of-integer* (*integer-of-natural d*))
  (− (*int-of-integer* (*integer-of-natural d*)))

instance ..

end

instantiation *int* :: *full-exhaustive*
begin

function *full-exhaustive-int'* ::
  (*int* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
   int ⇒ *int* ⇒ (*bool* × *term list*) *option*
 where *full-exhaustive-int' f d i* =
  (*if d* < *i then None*
   else
    (*case f* (*i*, λ-. *Code-Evaluation.term-of i*) *of*
      Some t ⇒ *Some t*
    | *None* ⇒ *full-exhaustive-int' f d* (*i* + *1*)))
by *pat-completeness auto*

termination
 by (*relation measure* (λ(-, *d*, *i*). *nat* (*d* + *1* − *i*))) *auto*

definition *full-exhaustive f d* =
 full-exhaustive-int' f (*int-of-integer* (*integer-of-natural d*))
  (− (*int-of-integer* (*integer-of-natural d*)))

instance ..

end

instantiation *prod* :: (*exhaustive*, *exhaustive*) *exhaustive*
begin

definition *exhaustive f d* = *exhaustive* (λ*x*. *exhaustive* (λ*y*. *f* ((*x*, *y*))) *d*) *d*

instance ..

end

context
includes *term-syntax*
begin

definition
[*code-unfold*]: *valtermify-pair x y =*
Code-Evaluation.valtermify (Pair :: 'a::typerep ⇒ 'b::typerep ⇒ 'a × 'b) {·} x {·} y

end

instantiation *prod* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive f d =*
full-exhaustive (λx. full-exhaustive (λy. f (valtermify-pair x y)) d) d

instance ..

end

instantiation *set* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-set*
where
exhaustive-set f i =
(if i = 0 then None
else
f {} orelse
exhaustive-set
(λA. f A orelse exhaustive (λx. if x ∈ A then None else f (insert x A)) (i − 1)) (i − 1))

instance ..

end

instantiation *set* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-set*
where
full-exhaustive-set f i =
(if i = 0 then None
else

f valterm-emptyset orelse
full-exhaustive-set
(λ*A*. *f A orelse Quickcheck-Exhaustive.full-exhaustive*
(λ*x*. *if fst x ∈ fst A then None else f* (*valtermify-insert x A*)) (*i* − *1*)) (*i* − *1*))

instance ..

end

instantiation *fun* :: ({*equal*,*exhaustive*}, *exhaustive*) *exhaustive*
begin

fun *exhaustive-fun′* ::
((′*a* ⇒ ′*b*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
exhaustive-fun′ f i d =
(*exhaustive* (λ*b*. *f* (λ-. *b*)) *d*) *orelse*
(*if i* > *1 then*
exhaustive-fun′
(λ*g*. *exhaustive* (λ*a*. *exhaustive* (λ*b*. *f* (*g*(*a* := *b*))) *d*) *d*) (*i* − *1*) *d else None*)

definition *exhaustive-fun* ::
((′*a* ⇒ ′*b*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-fun f d* = *exhaustive-fun′ f d d*

instance ..

end

definition [*code-unfold*]:
valtermify-absdummy =
(λ(*v*, *t*).
(λ-::′*a*. *v*,
λ*u*::*unit*. *Code-Evaluation.Abs* (*STR* ′′*x*′′) (*Typerep.typerep TYPE*(′*a*::*typerep*)) (*t* ())))

context
includes *term-syntax*
begin

definition
[*code-unfold*]: *valtermify-fun-upd g a b* =
Code-Evaluation.valtermify
(*fun-upd* :: (′*a*::*typerep* ⇒ ′*b*::*typerep*) ⇒ ′*a* ⇒ ′*b* ⇒ ′*a* ⇒ ′*b*) {·} *g* {·} *a* {·} *b*

end

instantiation *fun* :: ({*equal*,*full-exhaustive*}, *full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fun'* ::
(('*a* ⇒ '*b*) × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
natural ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-fun' f i d =
full-exhaustive (λ*v*. *f* (*valtermify-absdummy v*)) *d orelse*
(*if i* > *1 then*
full-exhaustive-fun'
(λ*g*. *full-exhaustive*
(λ*a*. *full-exhaustive* (λ*b*. *f* (*valtermify-fun-upd g a b*)) *d*) *d*) (*i* − *1*) *d*
else None)

definition *full-exhaustive-fun* ::
(('*a* ⇒ '*b*) × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
natural ⇒ (*bool* × *term list*) *option*
where *full-exhaustive-fun f d* = *full-exhaustive-fun' f d d*

instance ..

end

82.2.1 A smarter enumeration scheme for functions over finite datatypes

class *check-all* = *enum* + *term-of* +
fixes *check-all* :: ('*a* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ (*bool* ∗ *term list*) *option*
fixes *enum-term-of* :: '*a itself* ⇒ *unit* ⇒ *term list*

fun *check-all-n-lists* :: ('*a*::*check-all list* × (*unit* ⇒ *term list*) ⇒
(*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* ∗ *term list*) *option*
where
check-all-n-lists f n =
(*if n* = *0 then f* ([], (λ-. [])
else check-all (λ(*x*, *xt*).
check-all-n-lists (λ(*xs*, *xst*). *f* ((*x* # *xs*), (λ-. (*xt* () # *xst* ()))))) (*n* − *1*)))

context
includes *term-syntax*
begin

definition
[*code-unfold*]: *termify-fun-upd g a b* =
(*Code-Evaluation.termify*
(*fun-upd* :: ('*a*::*typerep* ⇒ '*b*::*typerep*) ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇒ '*b*) <·> *g* <·> *a*

```
<·> b)

end

definition mk-map-term ::
  (unit ⇒ typerep) ⇒ (unit ⇒ typerep) ⇒
    (unit ⇒ term list) ⇒ (unit ⇒ term list) ⇒ unit ⇒ term
  where mk-map-term T1 T2 domm rng =
    (λ-.
      let
        T1 = T1 ();
        T2 = T2 ();
        update-term =
          (λg (a, b).
            Code-Evaluation.App (Code-Evaluation.App (Code-Evaluation.App
             (Code-Evaluation.Const (STR ''Fun.fun-upd'')
                (Typerep.Typerep (STR ''fun'') [Typerep.Typerep (STR ''fun'') [T1,
T2],
                  Typerep.Typerep (STR ''fun'') [T1,
                   Typerep.Typerep (STR ''fun'') [T2, Typerep.Typerep (STR ''fun'')
[T1, T2]]]]))
                  g) a) b)
      in
        List.foldl update-term
          (Code-Evaluation.Abs (STR ''x'') T1
            (Code-Evaluation.Const (STR ''HOL.undefined'') T2)) (zip (domm ())
(rng ())))

instantiation fun :: ({equal,check-all}, check-all) check-all
begin

definition
  check-all f =
    (let
      mk-term =
        mk-map-term
          (λ-. Typerep.typerep (TYPE('a)))
          (λ-. Typerep.typerep (TYPE('b)))
          (enum-term-of (TYPE('a)));
      enum = (Enum.enum :: 'a list)
    in
      check-all-n-lists
        (λ(ys, yst). f (the ∘ map-of (zip enum ys), mk-term yst))
        (natural-of-nat (length enum)))

definition enum-term-of-fun :: ('a ⇒ 'b) itself ⇒ unit ⇒ term list
  where enum-term-of-fun =
    (λ- -.
      let
```

```
    enum-term-of-a = enum-term-of (TYPE('a));
    mk-term =
      mk-map-term
        (λ-. Typerep.typerep (TYPE('a)))
        (λ-. Typerep.typerep (TYPE('b)))
        enum-term-of-a
  in
    map (λys. mk-term (λ-. ys) ())
     (List.n-lists (length (enum-term-of-a ())) (enum-term-of (TYPE('b)) ()))
```

instance ..

end

context
  includes *term-syntax*
begin

fun *check-all-subsets* ::
  (('a::typerep) *set* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒
    ('a × (*unit* ⇒ *term*)) *list* ⇒ (*bool* × *term list*) *option*
where
  check-all-subsets f [] = *f valterm-emptyset*
| *check-all-subsets f* (*x* # *xs*) =
   check-all-subsets (λs. *case f s of Some ts* ⇒ *Some ts* | *None* ⇒ *f* (*valtermify-insert x s*)) *xs*

definition
  [*code-unfold*]: *term-emptyset* = *Code-Evaluation.termify* ({} :: ('a::typerep) *set*)

definition
  [*code-unfold*]: *termify-insert x s* =
    Code-Evaluation.termify (*insert* :: ('a::typerep) ⇒ 'a *set* ⇒ 'a *set*) <·> *x* <·> *s*

definition *setify* :: ('a::typerep) *itself* ⇒ *term list* ⇒ *term*
where
  setify T ts = *foldr* (*termify-insert T*) *ts* (*term-emptyset T*)

end

instantiation *set* :: (*check-all*) *check-all*
begin

definition
  check-all-set f =
     check-all-subsets f
      (*zip* (*Enum.enum* :: 'a *list*)
       (*map* (λa. λu :: *unit*. *a*) (*Quickcheck-Exhaustive.enum-term-of* (*TYPE* ('a))

())))

definition *enum-term-of-set* :: *'a set itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-set - - =*
map (*setify* (*TYPE*('*a*))) (*subseqs* (*Quickcheck-Exhaustive.enum-term-of* (*TYPE*('*a*))
()))

instance ..

end

instantiation *unit* :: *check-all*
begin

definition *check-all f* = *f* (*Code-Evaluation.valtermify* ())

definition *enum-term-of-unit* :: *unit itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-unit* = (λ- -. [*Code-Evaluation.term-of* ()])

instance ..

end

instantiation *bool* :: *check-all*
begin

definition
check-all f =
(*case f* (*Code-Evaluation.valtermify False*) *of*
Some x' ⇒ *Some x'*
| *None* ⇒ *f* (*Code-Evaluation.valtermify True*))

definition *enum-term-of-bool* :: *bool itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-bool* = (λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *bool list*))

instance ..

end

context
includes *term-syntax*
begin

definition [*code-unfold*]:
termify-pair x y =
Code-Evaluation.termify (*Pair* :: *'a*::*typerep* ⇒ *'b* :: *typerep* ⇒ *'a* ∗ *'b*) <·> *x* <·> *y*

end

instantiation *prod* :: (*check-all*, *check-all*) *check-all*
begin

definition *check-all f* = *check-all* (λ*x*. *check-all* (λ*y*. *f* (*valtermify-pair x y*)))

definition *enum-term-of-prod* :: ('*a* ∗ '*b*) *itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-prod* =
(λ- -.
map (λ(*x*, *y*). *termify-pair TYPE*('*a*) *TYPE*('*b*) *x y*)
(*List.product* (*enum-term-of* (*TYPE*('*a*)) ()) (*enum-term-of* (*TYPE*('*b*))
())))

instance ..

end

context
includes *term-syntax*
begin

definition
[*code-unfold*]: *valtermify-Inl x* =
Code-Evaluation.valtermify (*Inl* :: '*a*::*typerep* ⇒ '*a* + '*b* :: *typerep*) {·} *x*

definition
[*code-unfold*]: *valtermify-Inr x* =
Code-Evaluation.valtermify (*Inr* :: '*b*::*typerep* ⇒ '*a*::*typerep* + '*b*) {·} *x*

end

instantiation *sum* :: (*check-all*, *check-all*) *check-all*
begin

definition
check-all f = *check-all* (λ*a*. *f* (*valtermify-Inl a*)) *orelse check-all* (λ*b*. *f* (*valtermify-Inr b*))

definition *enum-term-of-sum* :: ('*a* + '*b*) *itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-sum* =
(λ- -.
let
T1 = *Typerep.typerep* (*TYPE*('*a*));
T2 = *Typerep.typerep* (*TYPE*('*b*))
in
map
(*Code-Evaluation.App* (*Code-Evaluation.Const* (*STR* ''*Sum-Type.Inl*''))

```
      (Typerep.Typerep (STR ''fun'') [T1, Typerep.Typerep (STR ''Sum-Type.sum'')
[T1, T2]]))
       (enum-term-of (TYPE('a)) ()) @
      map
       (Code-Evaluation.App (Code-Evaluation.Const (STR ''Sum-Type.Inr'')
      (Typerep.Typerep (STR ''fun'') [T2, Typerep.Typerep (STR ''Sum-Type.sum'')
[T1, T2]])))
       (enum-term-of (TYPE('b)) ()))

instance ..

end

instantiation char :: check-all
begin

primrec check-all-char' ::
  (char × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ char list ⇒ (bool × term
list) option
  where check-all-char' f [] = None
  | check-all-char' f (c # cs) = f (c, λ-. Code-Evaluation.term-of c)
      orelse check-all-char' f cs

definition check-all-char ::
  (char × (unit ⇒ term) ⇒ (bool × term list) option) ⇒ (bool × term list) option
  where check-all f = check-all-char' f Enum.enum

definition enum-term-of-char :: char itself ⇒ unit ⇒ term list
where
  enum-term-of-char = (λ- -. map Code-Evaluation.term-of (Enum.enum :: char
list))

instance ..

end

instantiation option :: (check-all) check-all
begin

definition
  check-all f =
    f (Code-Evaluation.valtermify (None :: 'a option)) orelse
    check-all
      (λ(x, t).
        f
          (Some x,
            λ-. Code-Evaluation.App
              (Code-Evaluation.Const (STR ''Option.option.Some'')
                (Typerep.Typerep (STR ''fun'')
```

```
            [Typerep.typerep TYPE('a),
           Typerep.Typerep (STR ''Option.option'') [Typerep.typerep TYPE('a)]]))
(t ()))))

definition enum-term-of-option :: 'a option itself ⇒ unit ⇒ term list
  where enum-term-of-option =
    (λ - -.
      Code-Evaluation.term-of (None :: 'a option) #
      (map
        (Code-Evaluation.App
          (Code-Evaluation.Const (STR ''Option.option.Some'')
            (Typerep.Typerep (STR ''fun'')
              [Typerep.typerep TYPE('a),
             Typerep.Typerep (STR ''Option.option'') [Typerep.typerep TYPE('a)]])))
        (enum-term-of (TYPE('a)) ())))

instance ..

end


instantiation Enum.finite-1 :: check-all
begin

definition check-all f = f (Code-Evaluation.valtermify Enum.finite-1.a1)

definition enum-term-of-finite-1 :: Enum.finite-1 itself ⇒ unit ⇒ term list
  where enum-term-of-finite-1 = (λ- -. [Code-Evaluation.term-of Enum.finite-1.a1])

instance ..

end

instantiation Enum.finite-2 :: check-all
begin

definition
  check-all f =
    (f (Code-Evaluation.valtermify Enum.finite-2.a1) orelse
     f (Code-Evaluation.valtermify Enum.finite-2.a2))

definition enum-term-of-finite-2 :: Enum.finite-2 itself ⇒ unit ⇒ term list
  where enum-term-of-finite-2 =
    (λ- -. map Code-Evaluation.term-of (Enum.enum :: Enum.finite-2 list))

instance ..

end
```

instantiation *Enum.finite-3* :: *check-all*
begin

definition
check-all f =
(*f* (*Code-Evaluation.valtermify Enum.finite-3*.a_1) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-3*.a_2) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-3*.a_3))

definition *enum-term-of-finite-3* :: *Enum.finite-3 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-3* =
(λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-3 list*))

instance ..

end

instantiation *Enum.finite-4* :: *check-all*
begin

definition
check-all f =
f (*Code-Evaluation.valtermify Enum.finite-4*.a_1) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_2) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_3) *orelse*
f (*Code-Evaluation.valtermify Enum.finite-4*.a_4)

definition *enum-term-of-finite-4* :: *Enum.finite-4 itself* ⇒ *unit* ⇒ *term list*
where *enum-term-of-finite-4* =
(λ- -. *map Code-Evaluation.term-of* (*Enum.enum* :: *Enum.finite-4 list*))

instance ..

end

82.3 Bounded universal quantifiers

class *bounded-forall* =
fixes *bounded-forall* :: ('*a* ⇒ *bool*) ⇒ *natural* ⇒ *bool*

82.4 Fast exhaustive combinators

class *fast-exhaustive* = *term-of* +
fixes *fast-exhaustive* :: ('*a* ⇒ *unit*) ⇒ *natural* ⇒ *unit*

axiomatization *throw-Counterexample* :: *term list* ⇒ *unit*
axiomatization *catch-Counterexample* :: *unit* ⇒ *term list option*

code-printing
constant *throw-Counterexample* ⇀

(*Quickcheck*) *raise* (*Exhaustive'-Generators.Counterexample* -)
| **constant** *catch-Counterexample* ⇀
(*Quickcheck*) (((-); *NONE*) *handle Exhaustive'-Generators.Counterexample ts* ⇒ *SOME ts*)

82.5 Continuation passing style functions as plus monad

type-synonym *'a cps* = (*'a* ⇒ *term list option*) ⇒ *term list option*

definition *cps-empty* :: *'a cps*
where *cps-empty* = (λ*cont*. *None*)

definition *cps-single* :: *'a* ⇒ *'a cps*
where *cps-single v* = (λ*cont*. *cont v*)

definition *cps-bind* :: *'a cps* ⇒ (*'a* ⇒ *'b cps*) ⇒ *'b cps*
where *cps-bind m f* = (λ*cont*. *m* (λ*a*. (*f a*) *cont*))

definition *cps-plus* :: *'a cps* ⇒ *'a cps* ⇒ *'a cps*
where *cps-plus a b* = (λ*c*. *case a c of None* ⇒ *b c* | *Some x* ⇒ *Some x*)

definition *cps-if* :: *bool* ⇒ *unit cps*
where *cps-if b* = (*if b then cps-single* () *else cps-empty*)

definition *cps-not* :: *unit cps* ⇒ *unit cps*
where *cps-not n* = (λ*c*. *case n* (λ*u*. *Some* []) *of None* ⇒ *c* () | *Some* - ⇒ *None*)

type-synonym *'a pos-bound-cps* =
(*'a* ⇒ (*bool* ∗ *term list*) *option*) ⇒ *natural* ⇒ (*bool* ∗ *term list*) *option*

definition *pos-bound-cps-empty* :: *'a pos-bound-cps*
where *pos-bound-cps-empty* = (λ*cont i*. *None*)

definition *pos-bound-cps-single* :: *'a* ⇒ *'a pos-bound-cps*
where *pos-bound-cps-single v* = (λ*cont i*. *cont v*)

definition *pos-bound-cps-bind* :: *'a pos-bound-cps* ⇒ (*'a* ⇒ *'b pos-bound-cps*) ⇒ *'b pos-bound-cps*
where *pos-bound-cps-bind m f* = (λ*cont i*. *if i* = *0 then None else* (*m* (λ*a*. (*f a*) *cont i*) (*i* − *1*)))

definition *pos-bound-cps-plus* :: *'a pos-bound-cps* ⇒ *'a pos-bound-cps* ⇒ *'a pos-bound-cps*
where *pos-bound-cps-plus a b* = (λ*c i*. *case a c i of None* ⇒ *b c i* | *Some x* ⇒ *Some x*)

definition *pos-bound-cps-if* :: *bool* ⇒ *unit pos-bound-cps*
where *pos-bound-cps-if b* = (*if b then pos-bound-cps-single* () *else pos-bound-cps-empty*)

datatype (*plugins only*: *code extraction*) (*dead 'a*) *unknown* =

Unknown | *Known 'a*

datatype (*plugins only*: *code extraction*) (*dead 'a*) *three-valued* =
Unknown-value | *Value 'a* | *No-value*

type-synonym *'a neg-bound-cps* =
('a unknown ⇒ term list three-valued) ⇒ natural ⇒ term list three-valued

definition *neg-bound-cps-empty* :: *'a neg-bound-cps*
where *neg-bound-cps-empty* = (*λcont i. No-value*)

definition *neg-bound-cps-single* :: *'a ⇒ 'a neg-bound-cps*
where *neg-bound-cps-single v* = (*λcont i. cont* (*Known v*))

definition *neg-bound-cps-bind* :: *'a neg-bound-cps ⇒ ('a ⇒ 'b neg-bound-cps) ⇒ 'b neg-bound-cps*
where *neg-bound-cps-bind m f* =
(*λcont i.*
if i = 0 then cont Unknown
else m (*λa. case a of Unknown ⇒ cont Unknown* | *Known a' ⇒ f a' cont i*) (*i − 1*))

definition *neg-bound-cps-plus* :: *'a neg-bound-cps ⇒ 'a neg-bound-cps ⇒ 'a neg-bound-cps*
where *neg-bound-cps-plus a b* =
(*λc i.*
case a c i of
No-value ⇒ b c i
| *Value x ⇒ Value x*
| *Unknown-value ⇒*
(*case b c i of*
No-value ⇒ Unknown-value
| *Value x ⇒ Value x*
| *Unknown-value ⇒ Unknown-value*))

definition *neg-bound-cps-if* :: *bool ⇒ unit neg-bound-cps*
where *neg-bound-cps-if b* = (*if b then neg-bound-cps-single* () *else neg-bound-cps-empty*)

definition *neg-bound-cps-not* :: *unit pos-bound-cps ⇒ unit neg-bound-cps*
where *neg-bound-cps-not n* =
(*λc i. case n* (*λu. Some* (*True*, [])) *i of None ⇒ c* (*Known* ()) | *Some* - ⇒ *No-value*)

definition *pos-bound-cps-not* :: *unit neg-bound-cps ⇒ unit pos-bound-cps*
where *pos-bound-cps-not n* =
(*λc i. case n* (*λu. Value* []) *i of No-value ⇒ c* () | *Value* - ⇒ *None* | *Unknown-value ⇒ None*)

82.6 Defining generators for any first-order data type

axiomatization *unknown* :: *'a*

notation (**output**) *unknown* (‹*?*›)

ML-file ‹*Tools/Quickcheck/exhaustive-generators.ML*›

declare [[*quickcheck-batch-tester* = *exhaustive*]]

82.7 Defining generators for abstract types

ML-file ‹*Tools/Quickcheck/abstract-generators.ML*›

hide-fact (**open**) *orelse-def*
no-notation *orelse* (**infixr** ‹*orelse*› *55*)

hide-const *valtermify-absdummy valtermify-fun-upd*
valterm-emptyset valtermify-insert
valtermify-pair valtermify-Inl valtermify-Inr
termify-fun-upd term-emptyset termify-insert termify-pair setify

hide-const (**open**)
exhaustive full-exhaustive
exhaustive-int' full-exhaustive-int'
exhaustive-integer' full-exhaustive-integer'
exhaustive-natural' full-exhaustive-natural'
throw-Counterexample catch-Counterexample
check-all enum-term-of
orelse unknown mk-map-term check-all-n-lists check-all-subsets

hide-type (**open**) *cps pos-bound-cps neg-bound-cps unknown three-valued*

hide-const (**open**) *cps-empty cps-single cps-bind cps-plus cps-if cps-not*
pos-bound-cps-empty pos-bound-cps-single pos-bound-cps-bind
pos-bound-cps-plus pos-bound-cps-if pos-bound-cps-not
neg-bound-cps-empty neg-bound-cps-single neg-bound-cps-bind
neg-bound-cps-plus neg-bound-cps-if neg-bound-cps-not
Unknown Known Unknown-value Value No-value

end

83 A compiler for predicates defined by introduction rules

theory *Predicate-Compile*
imports *Random-Sequence Quickcheck-Exhaustive*
keywords
code-pred :: *thy-goal* **and**

values :: *diag*
begin

ML-file ‹*Tools/Predicate-Compile/predicate-compile-aux.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-compilations.ML*›
ML-file ‹*Tools/Predicate-Compile/core-data.ML*›
ML-file ‹*Tools/Predicate-Compile/mode-inference.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-proof.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-core.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-data.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-fun.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-pred.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile-specialisation.ML*›
ML-file ‹*Tools/Predicate-Compile/predicate-compile.ML*›

83.1 Set membership as a generator predicate

Introduce a new constant for membership to allow fine-grained control in code equations.

context
begin

qualified definition *contains* :: *'a set* ⇒ *'a* ⇒ *bool*
where *contains A x* ⟷ *x* ∈ *A*

qualified definition *contains-pred* :: *'a set* ⇒ *'a* ⇒ *unit Predicate.pred*
where *contains-pred A x* = (*if x* ∈ *A then Predicate.single* () *else bot*)

qualified lemma *pred-of-setE*:
assumes *Predicate.eval* (*pred-of-set A*) *x*
obtains *contains A x*
using *assms* **by**(*simp add*: *contains-def*)

qualified lemma *pred-of-setI*: *contains A x* ⟹ *Predicate.eval* (*pred-of-set A*) *x*
by(*simp add*: *contains-def*)

qualified lemma *pred-of-set-eq*: *pred-of-set* ≡ λ*A*. *Predicate.Pred* (*contains A*)
by(*simp add*: *contains-def*[*abs-def*] *pred-of-set-def o-def*)

qualified lemma *containsI*: *x* ∈ *A* ⟹ *contains A x*
by(*simp add*: *contains-def*)

qualified lemma *containsE*: **assumes** *contains A x*
obtains *A' x'* **where** *A* = *A' x* = *x' x* ∈ *A*
using *assms* **by**(*simp add*: *contains-def*)

qualified lemma *contains-predI*: *contains A x* ⟹ *Predicate.eval* (*contains-pred A x*) ()
by(*simp add*: *contains-pred-def contains-def*)

qualified lemma *contains-predE*:
assumes *Predicate.eval* (*contains-pred A x*) *y*
obtains *contains A x*
using *assms* **by**(*simp add*: *contains-pred-def contains-def split*: *if-split-asm*)

qualified lemma *contains-pred-eq*: *contains-pred* ≡ λ*A x*. *Predicate.Pred* (λ*y*. *contains A x*)
by(*rule eq-reflection*)(*auto simp add*: *contains-pred-def fun-eq-iff contains-def intro*: *pred-eqI*)

qualified lemma *contains-pred-notI*:
¬ *contains A x* ⟹ *Predicate.eval* (*Predicate.not-pred* (*contains-pred A x*)) ()
by(*simp add*: *contains-pred-def contains-def not-pred-eq*)

end

setup ‹
let
val Fun = *Predicate-Compile-Aux.Fun*
val Input = *Predicate-Compile-Aux.Input*
val Output = *Predicate-Compile-Aux.Output*
val Bool = *Predicate-Compile-Aux.Bool*
val io = *Fun* (*Input*, *Fun* (*Output*, *Bool*))
val ii = *Fun* (*Input*, *Fun* (*Input*, *Bool*))
in
Core-Data.PredData.map (*Graph.new-node*
(***const-name*** ‹*Predicate-Compile.contains*›,
Core-Data.PredData {
pos = *Position.thread-data* (),
intros = [(*NONE*, @{*thm Predicate-Compile.containsI*})],
elim = *SOME* @{*thm Predicate-Compile.containsE*},
preprocessed = *true*,
function-names = [(*Predicate-Compile-Aux.Pred*,
[(*io*, ***const-name*** ‹*pred-of-set*›), (*ii*, ***const-name*** ‹*Predicate-Compile.contains-pred*›)])],

predfun-data = [
(*io*, *Core-Data.PredfunData* {
elim = @{*thm Predicate-Compile.pred-of-setE*},
intro = @{*thm Predicate-Compile.pred-of-setI*},
neg-intro = *NONE*,
definition = @{*thm Predicate-Compile.pred-of-set-eq*}
}),
(*ii*, *Core-Data.PredfunData* {
elim = @{*thm Predicate-Compile.contains-predE*},
intro = @{*thm Predicate-Compile.contains-predI*},
neg-intro = *SOME* @{*thm Predicate-Compile.contains-pred-notI*},
definition = @{*thm Predicate-Compile.contains-pred-eq*}
})],

needs-random = []}))
end
›

end

84 Counterexample generator performing narrowing-based testing

theory *Quickcheck-Narrowing*
imports *Quickcheck-Random*
keywords *find-unused-assms* :: *diag*
begin

84.1 Counterexample generator

84.1.1 Code generation setup

setup ‹*Code-Target.add-derived-target* (*Haskell-Quickcheck*, [(*Code-Haskell.target*, *I*)])›

code-printing
code-module *Typerep* ⇀ (*Haskell-Quickcheck*) ‹
module Typerep(*Typerep*(..)) *where*

data Typerep = *Typerep String* [*Typerep*]
› **for type-constructor** *typerep* **constant** *Typerep.Typerep*
| **type-constructor** *typerep* ⇀ (*Haskell-Quickcheck*) *Typerep.Typerep*
| **constant** *Typerep.Typerep* ⇀ (*Haskell-Quickcheck*) *Typerep.Typerep*

code-reserved
(*Haskell-Quickcheck*) *Typerep*

code-printing
type-constructor *integer* ⇀ (*Haskell-Quickcheck*) *Prelude.Int*
| **constant** *0*::*integer* ⇀
(*Haskell-Quickcheck*) !(*0*/ ::/ *Prelude.Int*)

setup ‹
let
val target = *Haskell-Quickcheck*;
fun print - = *Code-Haskell.print-numeral Prelude.Int*;
in
Numeral.add-code ***const-name*** ‹*Code-Numeral.Pos*› *I print target*
#> *Numeral.add-code* ***const-name*** ‹*Code-Numeral.Neg*› (~) *print target*
end
›

code-printing

constant *Code-Numeral.push-bit* ⇀
(*Haskell-Quickcheck*) *Bit'-Shifts.drop'*
| **constant** *Code-Numeral.drop-bit* ⇀
(*Haskell-Quickcheck*) *Bit'-Shifts.push'*

84.1.2 Narrowing's deep representation of types and terms

datatype (*plugins only*: *code extraction*) *narrowing-type* =
Narrowing-sum-of-products narrowing-type list list

datatype (*plugins only*: *code extraction*) *narrowing-term* =
Narrowing-variable integer list narrowing-type
| *Narrowing-constructor integer narrowing-term list*

datatype (*plugins only*: *code extraction*) (*dead* 'a) *narrowing-cons* =
Narrowing-cons narrowing-type (*narrowing-term list* ⇒ 'a) *list*

primrec *map-cons* :: ('a => 'b) => 'a *narrowing-cons* => 'b *narrowing-cons*
where
map-cons f (*Narrowing-cons ty cs*) = *Narrowing-cons ty* (*map* (λ*c*. *f* ∘ *c*) *cs*)

84.1.3 From narrowing's deep representation of terms to *HOL.Code-Evaluation*'s terms

class *partial-term-of* = *typerep* +
fixes *partial-term-of* :: 'a *itself* => *narrowing-term* => *Code-Evaluation.term*

lemma *partial-term-of-anything*: *partial-term-of x nt* ≡ *t*
by (*rule eq-reflection*) (*cases partial-term-of x nt*, *cases t*, *simp*)

84.1.4 Auxilary functions for Narrowing

consts *nth* :: 'a *list* => *integer* => 'a

code-printing constant *nth* ⇀ (*Haskell-Quickcheck*) **infixl** *9* !!

consts *error* :: *char list* => 'a

code-printing constant *error* ⇀ (*Haskell-Quickcheck*) *error*

consts *toEnum* :: *integer* => *char*

code-printing constant *toEnum* ⇀ (*Haskell-Quickcheck*) *Prelude.toEnum*

consts *marker* :: *char*

code-printing constant *marker* ⇀ (*Haskell-Quickcheck*) '\'0'

84.1.5 Narrowing's basic operations

type-synonym *'a narrowing = integer => 'a narrowing-cons*

definition *cons :: 'a => 'a narrowing*
where
cons a d = (Narrowing-cons (Narrowing-sum-of-products [[]]) [(λ-. a)])

fun *conv :: (narrowing-term list => 'a) list => narrowing-term => 'a*
where
conv cs (Narrowing-variable p -) = error (marker # map toEnum p)
| conv cs (Narrowing-constructor i xs) = (nth cs i) xs

fun *non-empty :: narrowing-type => bool*
where
non-empty (Narrowing-sum-of-products ps) = (¬ (List.null ps))

definition *apply :: ('a => 'b) narrowing => 'a narrowing => 'b narrowing*
where
apply f a d = (if d > 0 then
(case f d of Narrowing-cons (Narrowing-sum-of-products ps) cfs ⇒
case a (d − 1) of Narrowing-cons ta cas ⇒
let
shallow = non-empty ta;
cs = [(λ(x # xs) ⇒ cf xs (conv cas x)). shallow, cf ← cfs]
in Narrowing-cons (Narrowing-sum-of-products [ta # p. shallow, p ← ps])
cs)
else Narrowing-cons (Narrowing-sum-of-products []) [])

definition *sum :: 'a narrowing => 'a narrowing => 'a narrowing*
where
sum a b d =
(case a d of Narrowing-cons (Narrowing-sum-of-products ssa) ca ⇒
case b d of Narrowing-cons (Narrowing-sum-of-products ssb) cb ⇒
Narrowing-cons (Narrowing-sum-of-products (ssa @ ssb)) (ca @ cb))

lemma [*fundef-cong*]:
assumes *a d = a' d b d = b' d d = d'*
shows *sum a b d = sum a' b' d'*
using *assms* **unfolding** *sum-def* **by** (*auto split*: *narrowing-cons.split narrowing-type.split*)

lemma [*fundef-cong*]:
assumes *f d = f' d* (⋀*d'. 0 ≤ d' ∧ d' < d ⟹ a d' = a' d'*)
assumes *d = d'*
shows *apply f a d = apply f' a' d'*
proof −
note *assms*
moreover have *0 < d' ⟹ 0 ≤ d' − 1*
by (*simp add*: *less-integer-def less-eq-integer-def*)
ultimately show *?thesis*

by (*auto simp add*: *apply-def Let-def*
split: *narrowing-cons.split narrowing-type.split*)
qed

84.1.6 Narrowing generator type class

class *narrowing* =
fixes *narrowing* :: *integer* => *'a narrowing-cons*

datatype (*plugins only*: *code extraction*) *property* =
Universal narrowing-type (*narrowing-term* => *property*) *narrowing-term* => *Code-Evaluation.term*
| *Existential narrowing-type* (*narrowing-term* => *property*) *narrowing-term* => *Code-Evaluation.term*
| *Property bool*

definition *exists* :: (*'a* :: {*narrowing*, *partial-term-of*} => *property*) => *property*
where
exists f = (*case narrowing* (*100* :: *integer*) *of Narrowing-cons ty cs* ⇒ *Existential ty* (λ *t*. *f* (*conv cs t*)) (*partial-term-of* (*TYPE*(*'a*))))

definition *all* :: (*'a* :: {*narrowing*, *partial-term-of*} => *property*) => *property*
where
all f = (*case narrowing* (*100* :: *integer*) *of Narrowing-cons ty cs* ⇒ *Universal ty* (λ*t*. *f* (*conv cs t*)) (*partial-term-of* (*TYPE*(*'a*))))

84.1.7 class *is-testable*

The class *is-testable* ensures that all necessary type instances are generated.

class *is-testable*

instance *bool* :: *is-testable* **..**

instance *fun* :: ({*term-of*, *narrowing*, *partial-term-of*}, *is-testable*) *is-testable* **..**

definition *ensure-testable* :: *'a* :: *is-testable* => *'a* :: *is-testable*
where
ensure-testable f = *f*

84.1.8 Defining a simple datatype to represent functions in an incomplete and redundant way

datatype (*plugins only*: *code quickcheck-narrowing extraction*) (*dead 'a*, *dead 'b*) *ffun* =
Constant 'b
| *Update 'a 'b* (*'a*, *'b*) *ffun*

primrec *eval-ffun* :: (*'a*, *'b*) *ffun* => *'a* => *'b*

where
eval-ffun (*Constant c*) *x* = *c*
| *eval-ffun* (*Update x' y f*) *x* = (*if x* = *x' then y else eval-ffun f x*)

hide-type (**open**) *ffun*
hide-const (**open**) *Constant Update eval-ffun*

datatype (*plugins only*: *code quickcheck-narrowing extraction*) (*dead 'b*) *cfun* = *Constant 'b*

primrec *eval-cfun* :: *'b cfun* => *'a* => *'b*
where
eval-cfun (*Constant c*) *y* = *c*

hide-type (**open**) *cfun*
hide-const (**open**) *Constant eval-cfun Abs-cfun Rep-cfun*

84.1.9 Setting up the counterexample generator

external-file ‹~~/*src/HOL/Tools/Quickcheck/Narrowing-Engine.hs*›
external-file ‹~~/*src/HOL/Tools/Quickcheck/PNF-Narrowing-Engine.hs*›
ML-file ‹*Tools/Quickcheck/narrowing-generators.ML*›

definition *narrowing-dummy-partial-term-of* :: (*'a* :: *partial-term-of*) *itself* => *narrowing-term* => *term*
where
narrowing-dummy-partial-term-of = *partial-term-of*

definition *narrowing-dummy-narrowing* :: *integer* => (*'a* :: *narrowing*) *narrowing-cons*
where
narrowing-dummy-narrowing = *narrowing*

lemma [*code*]:
ensure-testable f =
(*let*
x = *narrowing-dummy-narrowing* :: *integer* => *bool narrowing-cons*;
y = *narrowing-dummy-partial-term-of* :: *bool itself* => *narrowing-term* => *term*;
z = (*conv* :: - => - => *unit*) *in f*)
unfolding *Let-def ensure-testable-def* **..**

84.2 Narrowing for sets

instantiation *set* :: (*narrowing*) *narrowing*
begin

definition *narrowing-set* = *Quickcheck-Narrowing.apply* (*Quickcheck-Narrowing.cons set*) *narrowing*

instance ..

end

84.3 Narrowing for integers

definition *drawn-from* :: *'a list* ⇒ *'a narrowing-cons*
where
drawn-from xs =
Narrowing-cons (*Narrowing-sum-of-products* (*map* (λ-. []) *xs*)) (*map* (λ*x* -. *x*) *xs*)

function *around-zero* :: *int* ⇒ *int list*
where
around-zero i = (*if i* < *0 then* [] *else* (*if i* = *0 then* [*0*] *else around-zero* (*i* − *1*) @ [*i*, −*i*]))
by *pat-completeness auto*
termination by (*relation measure nat*) *auto*

declare *around-zero.simps* [*simp del*]

lemma *length-around-zero*:
assumes *i* >= *0*
shows *length* (*around-zero i*) = *2* ∗ *nat i* + *1*
proof (*induct rule*: *int-ge-induct* [*OF assms*])
case *1*
from *1* **show** *?case* **by** (*simp add*: *around-zero.simps*)
next
case (*2 i*)
from *2* **show** *?case*
by (*simp add*: *around-zero.simps* [*of i* + *1*])
qed

instantiation *int* :: *narrowing*
begin

definition
narrowing-int d = (*let* (*u* :: - ⇒ - ⇒ *unit*) = *conv*; *i* = *int-of-integer d*
in drawn-from (*around-zero i*))

instance ..

end

lemma [*code*]:
partial-term-of (*ty* :: *int itself*) (*Narrowing-variable p t*) ≡
Code-Evaluation.Free (*STR ''-''*) (*Typerep.Typerep* (*STR ''Int.int''*) [])
partial-term-of (*ty* :: *int itself*) (*Narrowing-constructor i* []) ≡
(*if i mod 2* = *0*

```
    then Code-Evaluation.term-of (− (int-of-integer i) div 2)
    else Code-Evaluation.term-of ((int-of-integer i + 1) div 2))
  by (rule partial-term-of-anything)+

instantiation integer :: narrowing
begin

definition
  narrowing-integer d = (let (u :: - ⇒ - ⇒ unit) = conv; i = int-of-integer d
    in drawn-from (map integer-of-int (around-zero i)))

instance ..

end

lemma [code]:
  partial-term-of (ty :: integer itself) (Narrowing-variable p t) ≡
   Code-Evaluation.Free (STR ''-'') (Typerep.Typerep (STR ''Code-Numeral.integer'')
[])
  partial-term-of (ty :: integer itself) (Narrowing-constructor i []) ≡
    (if i mod 2 = 0
    then Code-Evaluation.term-of (− i div 2)
    else Code-Evaluation.term-of ((i + 1) div 2))
  by (rule partial-term-of-anything)+

code-printing constant Code-Evaluation.term-of :: integer ⇒ term → (Haskell-Quickcheck)

  (let { t = Typerep.Typerep Code'-Numeral.integer [];
    mkFunT s t = Typerep.Typerep fun [s, t];
    numT = Typerep.Typerep Num.num [];
    mkBit 0 = Generated'-Code.Const Num.num.Bit0 (mkFunT numT numT);
    mkBit 1 = Generated'-Code.Const Num.num.Bit1 (mkFunT numT numT);
    mkNumeral 1 = Generated'-Code.Const Num.num.One numT;
    mkNumeral i = let { q = i `Prelude.div` 2; r = i `Prelude.mod` 2 }
      in Generated'-Code.App (mkBit r) (mkNumeral q);
    mkNumber 0 = Generated'-Code.Const Groups.zero'-class.zero t;
    mkNumber 1 = Generated'-Code.Const Groups.one'-class.one t;
    mkNumber i = if i > 0 then
        Generated'-Code.App
          (Generated'-Code.Const Num.numeral'-class.numeral
            (mkFunT numT t))
          (mkNumeral i)
      else
        Generated'-Code.App
          (Generated'-Code.Const Groups.uminus'-class.uminus (mkFunT t t))
          (mkNumber (− i)); } in mkNumber)
```

84.4 The *find-unused-assms* command

ML-file ‹*Tools/Quickcheck/find-unused-assms.ML*›

84.5 Closing up

hide-type *narrowing-type narrowing-term narrowing-cons property*
hide-const *map-cons nth error toEnum marker empty Narrowing-cons conv non-empty ensure-testable all exists drawn-from around-zero*
hide-const (**open**) *Narrowing-variable Narrowing-constructor apply sum cons*
hide-fact *empty-def cons-def conv.simps non-empty.simps apply-def sum-def ensure-testable-def all-def exists-def*

end

theory *Mirabelle*
imports *Sledgehammer Predicate-Compile Presburger*
begin

ML-file ‹*Tools/Mirabelle/mirabelle-util.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle.ML*›

ML ‹
signature MIRABELLE-ACTION = sig
*val make-action : Mirabelle.action-context -> string * Mirabelle.action*
end
›

ML-file ‹*Tools/Mirabelle/mirabelle-arith.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-order.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-metis.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-presburger.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-quickcheck.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-sledgehammer-filter.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-sledgehammer.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-try0.ML*›
ML-file ‹*Tools/Mirabelle/mirabelle-isar-annotate.ML*›

end

85 Program extraction for HOL

theory *Extraction*
imports *Option*
begin

85.1 Setup

setup ‹

Extraction.add-types
[(*bool*, ([], *NONE*))] #>
Extraction.set-preprocessor (*fn thy* =>
Proofterm.rewrite-proof-notypes
([], *Rewrite-HOL-Proof.elim-cong* :: *Proof-Rewrite-Rules.rprocs true*) *o*
Proofterm.rewrite-proof thy
(*Rewrite-HOL-Proof.rews*,
Proof-Rewrite-Rules.rprocs true @ [*Proof-Rewrite-Rules.expand-of-class thy*])
o
Proof-Rewrite-Rules.elim-vars (*curry Const* ***const-name*** ‹*default*›))
›

lemmas [*extraction-expand*] =
meta-spec atomize-eq atomize-all atomize-imp atomize-conj
allE rev-mp conjE Eq-TrueI Eq-FalseI eqTrueI eqTrueE eq-cong2
notE′ impE′ impE iffE imp-cong simp-thms eq-True eq-False
induct-forall-eq induct-implies-eq induct-equal-eq induct-conj-eq
induct-atomize induct-atomize′ induct-rulify induct-rulify′
induct-rulify-fallback induct-trueI
True-implies-equals implies-True-equals TrueE
False-implies-equals implies-False-swap

lemmas [*extraction-expand-def*] =
HOL.induct-forall-def HOL.induct-implies-def HOL.induct-equal-def HOL.induct-conj-def
HOL.induct-true-def HOL.induct-false-def

datatype (*plugins only*: *code extraction*) *sumbool* = *Left* | *Right*

85.2 Type of extracted program

extract-type
typeof (*Trueprop P*) ≡ *typeof P*

typeof P ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*('*Q*))

typeof Q ≡ *Type* (*TYPE*(*Null*)) ⟹ *typeof* (*P* ⟶ *Q*) ≡ *Type* (*TYPE*(*Null*))

typeof P ≡ *Type* (*TYPE*('*P*)) ⟹ *typeof Q* ≡ *Type* (*TYPE*('*Q*)) ⟹
typeof (*P* ⟶ *Q*) ≡ *Type* (*TYPE*('*P* ⇒ '*Q*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∀ *x*. *P x*) ≡ *Type* (*TYPE*(*Null*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*('*P*))) ⟹
typeof (∀ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a* ⇒ '*P*))

(λ*x*. *typeof* (*P x*)) ≡ (λ*x*. *Type* (*TYPE*(*Null*))) ⟹
typeof (∃ *x*::'*a*. *P x*) ≡ *Type* (*TYPE*('*a*))

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE('P))) \Longrightarrow$
$\quad typeof\ (\exists\, x{::}'a.\ P\ x) \equiv Type\ (TYPE('a \times 'P))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE(sumbool))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('Q\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('P\ option))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \vee Q) \equiv Type\ (TYPE('P + 'Q))$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('Q))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('P))$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow typeof\ Q \equiv Type\ (TYPE('Q)) \Longrightarrow$
$\quad typeof\ (P \wedge Q) \equiv Type\ (TYPE('P \times 'Q))$

$typeof\ (P = Q) \equiv typeof\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P))$

$typeof\ (x \in P) \equiv typeof\ P$

85.3 Realizability

realizability
$(realizes\ t\ (Trueprop\ P)) \equiv (Trueprop\ (realizes\ t\ P))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (realizes\ Null\ P \longrightarrow realizes\ t\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE('P))) \Longrightarrow$
$\ (typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x{::}'P.\ realizes\ x\ P \longrightarrow realizes\ Null\ Q)$

$(realizes\ t\ (P \longrightarrow Q)) \equiv (\forall\, x.\ realizes\ x\ P \longrightarrow realizes\ (t\ x)\ Q)$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$
$\quad (realizes\ t\ (\forall\, x.\ P\ x)) \equiv (\forall\, x.\ realizes\ Null\ (P\ x))$

$(realizes\ t\ (\forall\, x.\ P\ x)) \equiv (\forall\, x.\ realizes\ (t\ x)\ (P\ x))$

$(\lambda x.\ typeof\ (P\ x)) \equiv (\lambda x.\ Type\ (TYPE(Null))) \Longrightarrow$

$(realizes\ t\ (\exists\, x.\ P\ x)) \equiv (realizes\ Null\ (P\ t))$

$(realizes\ t\ (\exists\, x.\ P\ x)) \equiv (realizes\ (snd\ t)\ (P\ (fst\ t)))$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Left \Rightarrow realizes\ Null\ P \mid Right \Rightarrow realizes\ Null\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ P \mid Some\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ None \Rightarrow realizes\ Null\ Q \mid Some\ p \Rightarrow realizes\ p\ P)$

$(realizes\ t\ (P \vee Q)) \equiv$
$(case\ t\ of\ Inl\ p \Rightarrow realizes\ p\ P \mid Inr\ q \Rightarrow realizes\ q\ Q)$

$(typeof\ P) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ Null\ P \wedge realizes\ t\ Q)$

$(typeof\ Q) \equiv (Type\ (TYPE(Null))) \Longrightarrow$
$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ t\ P \wedge realizes\ Null\ Q)$

$(realizes\ t\ (P \wedge Q)) \equiv (realizes\ (fst\ t)\ P \wedge realizes\ (snd\ t)\ Q)$

$typeof\ P \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv \neg\ realizes\ Null\ P$

$typeof\ P \equiv Type\ (TYPE('P)) \Longrightarrow$
$realizes\ t\ (\neg\ P) \equiv (\forall\, x{::}'P.\ \neg\ realizes\ x\ P)$

$typeof\ (P{::}bool) \equiv Type\ (TYPE(Null)) \Longrightarrow$
$typeof\ Q \equiv Type\ (TYPE(Null)) \Longrightarrow$
$realizes\ t\ (P = Q) \equiv realizes\ Null\ P = realizes\ Null\ Q$

$(realizes\ t\ (P = Q)) \equiv (realizes\ t\ ((P \longrightarrow Q) \wedge (Q \longrightarrow P)))$

85.4 Computational content of basic inference rules

theorem *disjE-realizer*:
assumes *r*: $case\ x\ of\ Inl\ p \Rightarrow P\ p \mid Inr\ q \Rightarrow Q\ q$
and *r1*: $\bigwedge p.\ P\ p \Longrightarrow R\ (f\ p)$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows $R\ (case\ x\ of\ Inl\ p \Rightarrow f\ p \mid Inr\ q \Rightarrow g\ q)$
proof (*cases x*)
case *Inl*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)

next
case *Inr*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer2*:
assumes *r*: *case x of None ⇒ P | Some q ⇒ Q q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $\bigwedge q.\ Q\ q \Longrightarrow R\ (g\ q)$
shows *R* (*case x of None ⇒ f | Some q ⇒ g q*)
proof (*cases x*)
case *None*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Some*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *disjE-realizer3*:
assumes *r*: *case x of Left ⇒ P | Right ⇒ Q*
and *r1*: $P \Longrightarrow R\ f$ **and** *r2*: $Q \Longrightarrow R\ g$
shows *R* (*case x of Left ⇒ f | Right ⇒ g*)
proof (*cases x*)
case *Left*
with *r* **show** *?thesis* **by** *simp* (*rule r1*)
next
case *Right*
with *r* **show** *?thesis* **by** *simp* (*rule r2*)
qed

theorem *conjI-realizer*:
$P\ p \Longrightarrow Q\ q \Longrightarrow P\ (fst\ (p,\ q)) \wedge Q\ (snd\ (p,\ q))$
by *simp*

theorem *exI-realizer*:
$P\ y\ x \Longrightarrow P\ (snd\ (x,\ y))\ (fst\ (x,\ y))$ **by** *simp*

theorem *exE-realizer*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q\ (f\ x\ y)) \Longrightarrow Q\ (let\ (x,\ y) = p\ in\ f\ x\ y)$
by (*cases p*) (*simp add*: *Let-def*)

theorem *exE-realizer'*: $P\ (snd\ p)\ (fst\ p) \Longrightarrow$
$(\bigwedge x\ y.\ P\ y\ x \Longrightarrow Q) \Longrightarrow Q$ **by** (*cases p*) *simp*

realizers
impI (*P*, *Q*): *λpq. pq*
$\boldsymbol{\lambda}$(*c*: -) (*d*: -) *P Q pq* (*h*: -). *allI* · - · *c* · ($\boldsymbol{\lambda}$*x*. *impI* · - · - · (*h* · *x*))

impI (*P*): *Null*
$\boldsymbol{\lambda}$(*c*: -) *P Q* (*h*: -). *allI* · - · *c* · ($\boldsymbol{\lambda}$*x*. *impI* · - · - · (*h* · *x*))

impI (*Q*): $\lambda q.\ q\ \boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ q.\ impI \cdot \text{-} \cdot \text{-}$

impI: *Null impI*

mp (*P*, *Q*): $\lambda pq.\ pq$
$\boldsymbol{\lambda}(c\colon \text{-})\ (d\colon \text{-})\ P\ Q\ pq\ (h\colon \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet c \bullet h)$

mp (*P*): *Null*
$\boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ (h\colon \text{-})\ p.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet (spec \cdot \text{-} \cdot p \bullet c \bullet h)$

mp (*Q*): $\lambda q.\ q\ \boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ q.\ mp \cdot \text{-} \cdot \text{-} \cdot \text{-}$

mp: *Null mp*

allI (*P*): $\lambda p.\ p\ \boldsymbol{\lambda}(c\colon \text{-})\ P\ (d\colon \text{-})\ p.\ allI \cdot \text{-} \cdot d$

allI: *Null allI*

spec (*P*): $\lambda x\ p.\ p\ x\ \boldsymbol{\lambda}(c\colon \text{-})\ P\ x\ (d\colon \text{-})\ p.\ spec \cdot \text{-} \cdot x \bullet d$

spec: *Null spec*

exI (*P*): $\lambda x\ p.\ (x,\ p)\ \boldsymbol{\lambda}(c\colon \text{-})\ P\ x\ (d\colon \text{-})\ p.\ exI\text{-}realizer \cdot P \cdot p \cdot x \bullet c \bullet d$

exI: $\lambda x.\ x\ \boldsymbol{\lambda} P\ x\ (c\colon \text{-})\ (h\colon \text{-}).\ h$

exE (*P*, *Q*): $\lambda p\ pq.\ let\ (x,\ y) = p\ in\ pq\ x\ y$
$\boldsymbol{\lambda}(c\colon \text{-})\ (d\colon \text{-})\ P\ Q\ (e\colon \text{-})\ p\ (h\colon \text{-})\ pq.\ exE\text{-}realizer \cdot P \cdot p \cdot Q \cdot pq \bullet c \bullet e \bullet d \bullet h$

exE (*P*): *Null*
$\boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ (d\colon \text{-})\ p.\ exE\text{-}realizer' \cdot \text{-} \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet c \bullet d$

exE (*Q*): $\lambda x\ pq.\ pq\ x$
$\boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ (d\colon \text{-})\ x\ (h1\colon \text{-})\ pq\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

exE: *Null*
$\boldsymbol{\lambda} P\ Q\ (c\colon \text{-})\ x\ (h1\colon \text{-})\ (h2\colon \text{-}).\ h2 \cdot x \bullet h1$

conjI (*P*, *Q*): *Pair*
$\boldsymbol{\lambda}(c\colon \text{-})\ (d\colon \text{-})\ P\ Q\ p\ (h\colon \text{-})\ q.\ conjI\text{-}realizer \cdot P \cdot p \cdot Q \cdot q \bullet c \bullet d \bullet h$

conjI (*P*): $\lambda p.\ p$
$\boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ p.\ conjI \cdot \text{-} \cdot \text{-} \cdot \text{-}$

conjI (*Q*): $\lambda q.\ q$
$\boldsymbol{\lambda}(c\colon \text{-})\ P\ Q\ (h\colon \text{-})\ q.\ conjI \cdot \text{-} \cdot \text{-} \cdot \text{-} \bullet h$

conjI: *Null conjI*

conjunct1 (*P*, *Q*): *fst*
λ(*c*: -) (*d*: -) *P Q pq*. *conjunct1* · - · -

conjunct1 (*P*): λ*p*. *p*
λ(*c*: -) *P Q p*. *conjunct1* · - · -

conjunct1 (*Q*): *Null*
λ(*c*: -) *P Q q*. *conjunct1* · - · -

conjunct1: *Null conjunct1*

conjunct2 (*P*, *Q*): *snd*
λ(*c*: -) (*d*: -) *P Q pq*. *conjunct2* · - · -

conjunct2 (*P*): *Null*
λ(*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2 (*Q*): λ*p*. *p*
λ(*c*: -) *P Q p*. *conjunct2* · - · -

conjunct2: *Null conjunct2*

disjI1 (*P*, *Q*): *Inl*
λ(*c*: -) (*d*: -) *P Q p*. *iffD2* · - · - · (*sum.case-1* · *P* · - · *p* · *arity-type-bool* · *c* · *d*)

disjI1 (*P*): *Some*
λ(*c*: -) *P Q p*. *iffD2* · - · - · (*option.case-2* · - · *P* · *p* · *arity-type-bool* · *c*)

disjI1 (*Q*): *None*
λ(*c*: -) *P Q*. *iffD2* · - · - · (*option.case-1* · - · - · *arity-type-bool* · *c*)

disjI1: *Left*
λ*P Q*. *iffD2* · - · - · (*sumbool.case-1* · - · - · *arity-type-bool*)

disjI2 (*P*, *Q*): *Inr*
λ(*d*: -) (*c*: -) *Q P q*. *iffD2* · - · - · (*sum.case-2* · - · *Q* · *q* · *arity-type-bool* · *c* · *d*)

disjI2 (*P*): *None*
λ(*c*: -) *Q P*. *iffD2* · - · - · (*option.case-1* · - · - · *arity-type-bool* · *c*)

disjI2 (*Q*): *Some*
λ(*c*: -) *Q P q*. *iffD2* · - · - · (*option.case-2* · - · *Q* · *q* · *arity-type-bool* · *c*)

disjI2: *Right*
λ*Q P*. *iffD2* · - · - · (*sumbool.case-2* · - · - · *arity-type-bool*)

disjE (*P*, *Q*, *R*): *λpq pr qr.*
(*case pq of Inl p ⇒ pr p | Inr q ⇒ qr q*)
λ(*c*: -) (*d*: -) (*e*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr.*
disjE-realizer · - · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *e* · *h1* · *h2*

disjE (*Q*, *R*): *λpq pr qr.*
(*case pq of None ⇒ pr | Some q ⇒ qr q*)
λ(*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr.*
disjE-realizer2 · - · - · - · *pq* · *R* · *pr* · *qr* · *c* · *d* · *h1* · *h2*

disjE (*P*, *R*): *λpq pr qr.*
(*case pq of None ⇒ qr | Some p ⇒ pr p*)
λ(*c*: -) (*d*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr* (*h3*: -).
disjE-realizer2 · - · - · - · *pq* · *R* · *qr* · *pr* · *c* · *d* · *h1* · *h3* · *h2*

disjE (*R*): *λpq pr qr.*
(*case pq of Left ⇒ pr | Right ⇒ qr*)
λ(*c*: -) *P Q R pq* (*h1*: -) *pr* (*h2*: -) *qr.*
disjE-realizer3 · - · - · - · *pq* · *R* · *pr* · *qr* · *c* · *h1* · *h2*

disjE (*P*, *Q*): *Null*
λ(*c*: -) (*d*: -) *P Q R pq. disjE-realizer* · - · - · - · *pq* · (*λx. R*) · - · - · - · *c* · *d* · *arity-type-bool*

disjE (*Q*): *Null*
λ(*c*: -) *P Q R pq. disjE-realizer2* · - · - · - · *pq* · (*λx. R*) · - · - · - · *c* · *arity-type-bool*

disjE (*P*): *Null*
λ(*c*: -) *P Q R pq* (*h1*: -) (*h2*: -) (*h3*: -).
disjE-realizer2 · - · - · - · *pq* · (*λx. R*) · - · - · - · *c* · *arity-type-bool* · *h1* · *h3* · *h2*

disjE: *Null*
λ*P Q R pq. disjE-realizer3* · - · - · - · *pq* · (*λx. R*) · - · - · - · *arity-type-bool*

FalseE (*P*): *default*
λ(*c*: -) *P. FalseE* · -

FalseE: *Null FalseE*

notI (*P*): *Null*
λ(*c*: -) *P* (*h*: -). *allI* · - · - · *c* · (**λ***x. notI* · - · - · (*h* · *x*))

notI: *Null notI*

notE (*P*, *R*): *λp. default*
λ(*c*: -) (*d*: -) *P R* (*h*: -) *p. notE* · - · - · - · (*spec* · - · - · *p* · *c* · *h*)

notE (*P*): *Null*
λ(*c*: -) *P R* (*h*: -) *p. notE* · - · - · - · (*spec* · - · - · *p* · *c* · *h*)

notE (*R*): *default*
  λ(*c*: -) *P R*. *notE* · - · - -

notE: *Null notE*

subst (*P*): *λs t ps*. *ps*
  λ(*c*: -) *s t P* (*d*: -) (*h*: -) *ps*. *subst* · *s* · *t* · *P ps* · *d* · *h*

subst: *Null subst*

iffD1 (*P*, *Q*): *fst*
  λ(*d*: -) (*c*: -) *Q P pq* (*h*: -) *p*.
    mp · - · - · - · (*spec* · - · *p* · *d* · (*conjunct1* · - · - · - · *h*))

iffD1 (*P*): *λp*. *p*
  λ(*c*: -) *Q P p* (*h*: -). *mp* · - · - · - · (*conjunct1* · - · - · - · *h*)

iffD1 (*Q*): *Null*
  λ(*c*: -) *Q P q1* (*h*: -) *q2*.
    mp · - · - · - · (*spec* · - · *q2* · *c* · (*conjunct1* · - · - · - · *h*))

iffD1: *Null iffD1*

iffD2 (*P*, *Q*): *snd*
  λ(*c*: -) (*d*: -) *P Q pq* (*h*: -) *q*.
    mp · - · - · - · (*spec* · - · *q* · *d* · (*conjunct2* · - · - · - · *h*))

iffD2 (*P*): *λp*. *p*
  λ(*c*: -) *P Q p* (*h*: -). *mp* · - · - · - · (*conjunct2* · - · - · - · *h*)

iffD2 (*Q*): *Null*
  λ(*c*: -) *P Q q1* (*h*: -) *q2*.
    mp · - · - · - · (*spec* · - · *q2* · *c* · (*conjunct2* · - · - · - · *h*))

iffD2: *Null iffD2*

iffI (*P*, *Q*): *Pair*
  λ(*c*: -) (*d*: -) *P Q pq* (*h1* : -) *qp* (*h2* : -). *conjI-realizer* ·
    (*λpq*. ∀ *x*. *P x* ⟶ *Q* (*pq x*)) · *pq* ·
    (*λqp*. ∀ *x*. *Q x* ⟶ *P* (*qp x*)) · *qp* ·
    (*arity-type-fun* · *c* · *d*) ·
    (*arity-type-fun* · *d* · *c*) ·
    (*allI* · - · *c* · (**λ***x*. *impI* · - · - · - · (*h1* · *x*))) ·
    (*allI* · - · *d* · (**λ***x*. *impI* · - · - · - · (*h2* · *x*)))

iffI (*P*): *λp*. *p*
  λ(*c*: -) *P Q* (*h1* : -) *p* (*h2* : -). *conjI* · - · - · -
    (*allI* · - · *c* · (**λ***x*. *impI* · - · - · - · (*h1* · *x*))) ·

(*impI* · - · - · *h2*)

iffI (*Q*): λ*q*. *q*
λ(*c*: -) *P Q q* (*h1* : -) (*h2* : -). *conjI* · - · - ·
(*impI* · - · - · *h1*) ·
(*allI* · - · *c* · (**λ***x*. *impI* · - · - · (*h2* · *x*)))

iffI: *Null iffI*

end

86 Extensible records with structural subtyping

theory *Record*
imports *Quickcheck-Exhaustive*
keywords
record :: *thy-defn* **and**
print-record :: *diag*
begin

86.1 Introduction

Records are isomorphic to compound tuple types. To implement efficient records, we make this isomorphism explicit. Consider the record access/update simplification *alpha* (*beta-update f rec*) = *alpha rec* for distinct fields alpha and beta of some record rec with n fields. There are $n \hat{} 2$ such theorems, which prohibits storage of all of them for large n. The rules can be proved on the fly by case decomposition and simplification in O(n) time. By creating O(n) isomorphic-tuple types while defining the record, however, we can prove the access/update simplification in $O(log(n) \hat{} 2)$ time.

The O(n) cost of case decomposition is not because O(n) steps are taken, but rather because the resulting rule must contain O(n) new variables and an O(n) size concrete record construction. To sidestep this cost, we would like to avoid case decomposition in proving access/update theorems.

Record types are defined as isomorphic to tuple types. For instance, a record type with fields $'a$, $'b$, $'c$ and $'d$ might be introduced as isomorphic to $'a \times ('b \times ('c \times 'd))$. If we balance the tuple tree to $('a \times 'b) \times ('c \times 'd)$ then accessors can be defined by converting to the underlying type then using O(log(n)) fst or snd operations. Updators can be defined similarly, if we introduce a *fst-update* and *snd-update* function. Furthermore, we can prove the access/update theorem in O(log(n)) steps by using simple rewrites on fst, snd, *fst-update* and *snd-update*.

The catch is that, although O(log(n)) steps were taken, the underlying type we converted to is a tuple tree of size O(n). Processing this term type wastes performance. We avoid this for large n by taking each subtree of size K and

defining a new type isomorphic to that tuple subtree. A record can now be defined as isomorphic to a tuple tree of these O(n/K) new types, or, if $n > K*K$, we can repeat the process, until the record can be defined in terms of a tuple tree of complexity less than the constant K.

If we prove the access/update theorem on this type with the analogous steps to the tuple tree, we consume $O(log(n)\hat{}2)$ time as the intermediate terms are $O(log(n))$ in size and the types needed have size bounded by K. To enable this analogous traversal, we define the functions seen below: *iso-tuple-fst*, *iso-tuple-snd*, *iso-tuple-fst-update* and *iso-tuple-snd-update*. These functions generalise tuple operations by taking a parameter that encapsulates a tuple isomorphism. The rewrites needed on these functions now need an additional assumption which is that the isomorphism works.

These rewrites are typically used in a structured way. They are here presented as the introduction rule *isomorphic-tuple.intros* rather than as a rewrite rule set. The introduction form is an optimisation, as net matching can be performed at one term location for each step rather than the simplifier searching the term for possible pattern matches. The rule set is used as it is viewed outside the locale, with the locale assumption (that the isomorphism is valid) left as a rule assumption. All rules are structured to aid net matching, using either a point-free form or an encapsulating predicate.

86.2 Operators and lemmas for types isomorphic to tuples

datatype (*dead* 'a, *dead* 'b, *dead* 'c) *tuple-isomorphism* =
  Tuple-Isomorphism 'a ⇒ 'b × 'c 'b × 'c ⇒ 'a

primrec
  repr :: ('a, 'b, 'c) *tuple-isomorphism* ⇒ 'a ⇒ 'b × 'c **where**
  repr (*Tuple-Isomorphism r a*) = *r*

primrec
  abst :: ('a, 'b, 'c) *tuple-isomorphism* ⇒ 'b × 'c ⇒ 'a **where**
  abst (*Tuple-Isomorphism r a*) = *a*

definition
  iso-tuple-fst :: ('a, 'b, 'c) *tuple-isomorphism* ⇒ 'a ⇒ 'b **where**
  iso-tuple-fst isom = *fst* ∘ *repr isom*

definition
  iso-tuple-snd :: ('a, 'b, 'c) *tuple-isomorphism* ⇒ 'a ⇒ 'c **where**
  iso-tuple-snd isom = *snd* ∘ *repr isom*

definition
  iso-tuple-fst-update ::
    ('a, 'b, 'c) *tuple-isomorphism* ⇒ ('b ⇒ 'b) ⇒ ('a ⇒ 'a) **where**
  iso-tuple-fst-update isom f = *abst isom* ∘ *apfst f* ∘ *repr isom*

definition
iso-tuple-snd-update ::
(*'a*, *'b*, *'c*) *tuple-isomorphism* ⇒ (*'c* ⇒ *'c*) ⇒ (*'a* ⇒ *'a*) **where**
iso-tuple-snd-update isom f = *abst isom* ∘ *apsnd f* ∘ *repr isom*

definition
iso-tuple-cons ::
(*'a*, *'b*, *'c*) *tuple-isomorphism* ⇒ *'b* ⇒ *'c* ⇒ *'a* **where**
iso-tuple-cons isom = *curry* (*abst isom*)

86.3 Logical infrastructure for records

definition
iso-tuple-surjective-proof-assist :: *'a* ⇒ *'b* ⇒ (*'a* ⇒ *'b*) ⇒ *bool* **where**
iso-tuple-surjective-proof-assist x y f ⟷ *f x* = *y*

definition
iso-tuple-update-accessor-cong-assist ::
((*'b* ⇒ *'b*) ⇒ (*'a* ⇒ *'a*)) ⇒ (*'a* ⇒ *'b*) ⇒ *bool* **where**
iso-tuple-update-accessor-cong-assist upd ac ⟷
(∀ *f v*. *upd* (λ*x*. *f* (*ac v*)) *v* = *upd f v*) ∧ (∀ *v*. *upd id v* = *v*)

definition
iso-tuple-update-accessor-eq-assist ::
((*'b* ⇒ *'b*) ⇒ (*'a* ⇒ *'a*)) ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇒ (*'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ *bool*
where
iso-tuple-update-accessor-eq-assist upd ac v f v' x ⟷
upd f v = *v'* ∧ *ac v* = *x* ∧ *iso-tuple-update-accessor-cong-assist upd ac*

lemma *update-accessor-congruence-foldE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*
and *r*: *r* = *r'* **and** *v*: *ac r'* = *v'*
and *f*: ⋀*v*. *v'* = *v* ⟹ *f v* = *f' v*
shows *upd f r* = *upd f' r'*
using *uac r v* [*symmetric*]
apply (*subgoal-tac upd* (λ*x*. *f* (*ac r'*)) *r'* = *upd* (λ*x*. *f'* (*ac r'*)) *r'*)
apply (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)
apply (*simp add*: *f*)
done

lemma *update-accessor-congruence-unfoldE*:
iso-tuple-update-accessor-cong-assist upd ac ⟹
r = *r'* ⟹ *ac r'* = *v'* ⟹ (⋀*v*. *v* = *v'* ⟹ *f v* = *f' v*) ⟹
upd f r = *upd f' r'*
apply (*erule*(2) *update-accessor-congruence-foldE*)
apply *simp*
done

lemma *iso-tuple-update-accessor-cong-assist-id*:
iso-tuple-update-accessor-cong-assist upd ac $\Longrightarrow$ *upd id* = *id*
by *rule* (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)

lemma *update-accessor-noopE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*
and *ac*: *f* (*ac x*) = *ac x*
shows *upd f x* = *x*
using *uac*
by (*simp add*: *ac iso-tuple-update-accessor-cong-assist-id* [*OF uac*, *unfolded id-def*]
cong: *update-accessor-congruence-unfoldE* [*OF uac*])

lemma *update-accessor-noop-compE*:
assumes *uac*: *iso-tuple-update-accessor-cong-assist upd ac*
and *ac*: *f* (*ac x*) = *ac x*
shows *upd* (*g* $\circ$ *f*) *x* = *upd g x*
by (*simp add*: *ac cong*: *update-accessor-congruence-unfoldE*[*OF uac*])

lemma *update-accessor-cong-assist-idI*:
iso-tuple-update-accessor-cong-assist id id
by (*simp add*: *iso-tuple-update-accessor-cong-assist-def*)

lemma *update-accessor-cong-assist-triv*:
iso-tuple-update-accessor-cong-assist upd ac $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd ac
by *assumption*

lemma *update-accessor-accessor-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow$ *ac v* = *x*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *update-accessor-updator-eqE*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow$ *upd f v* = *v'*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-update-accessor-eq-assist-idI*:
v' = *f v* $\Longrightarrow$ *iso-tuple-update-accessor-eq-assist id id v f v' v*
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def update-accessor-cong-assist-idI*)

lemma *iso-tuple-update-accessor-eq-assist-triv*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow$
iso-tuple-update-accessor-eq-assist upd ac v f v' x
by *assumption*

lemma *iso-tuple-update-accessor-cong-from-eq*:
iso-tuple-update-accessor-eq-assist upd ac v f v' x $\Longrightarrow$
iso-tuple-update-accessor-cong-assist upd ac
by (*simp add*: *iso-tuple-update-accessor-eq-assist-def*)

lemma *iso-tuple-surjective-proof-assistI*:
f x = y ⟹ iso-tuple-surjective-proof-assist x y f
by (*simp add*: *iso-tuple-surjective-proof-assist-def*)

lemma *iso-tuple-surjective-proof-assist-idE*:
iso-tuple-surjective-proof-assist x y id ⟹ x = y
by (*simp add*: *iso-tuple-surjective-proof-assist-def*)

locale *isomorphic-tuple* =
fixes *isom* :: (*'a*, *'b*, *'c*) *tuple-isomorphism*
assumes *repr-inv*: ⋀*x*. *abst isom* (*repr isom x*) = *x*
and *abst-inv*: ⋀*y*. *repr isom* (*abst isom y*) = *y*
begin

lemma *repr-inj*: *repr isom x = repr isom y ⟷ x = y*
by (*auto dest*: *arg-cong* [*of repr isom x repr isom y abst isom*]
simp add: *repr-inv*)

lemma *abst-inj*: *abst isom x = abst isom y ⟷ x = y*
by (*auto dest*: *arg-cong* [*of abst isom x abst isom y repr isom*]
simp add: *abst-inv*)

lemmas *simps = Let-def repr-inv abst-inv repr-inj abst-inj*

lemma *iso-tuple-access-update-fst-fst*:
f ∘ h g = j ∘ f ⟹
(*f ∘ iso-tuple-fst isom*) ∘ (*iso-tuple-fst-update isom ∘ h*) *g* =
j ∘ (*f ∘ iso-tuple-fst isom*)
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-fst-def simps*
fun-eq-iff)

lemma *iso-tuple-access-update-snd-snd*:
f ∘ h g = j ∘ f ⟹
(*f ∘ iso-tuple-snd isom*) ∘ (*iso-tuple-snd-update isom ∘ h*) *g* =
j ∘ (*f ∘ iso-tuple-snd isom*)
by (*clarsimp simp*: *iso-tuple-snd-update-def iso-tuple-snd-def simps*
fun-eq-iff)

lemma *iso-tuple-access-update-fst-snd*:
(*f ∘ iso-tuple-fst isom*) ∘ (*iso-tuple-snd-update isom ∘ h*) *g* =
id ∘ (*f ∘ iso-tuple-fst isom*)
by (*clarsimp simp*: *iso-tuple-snd-update-def iso-tuple-fst-def simps*
fun-eq-iff)

lemma *iso-tuple-access-update-snd-fst*:
(*f ∘ iso-tuple-snd isom*) ∘ (*iso-tuple-fst-update isom ∘ h*) *g* =
id ∘ (*f ∘ iso-tuple-snd isom*)
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-def simps*
fun-eq-iff)

lemma *iso-tuple-update-swap-fst-fst*:
h f ∘ j g = j g ∘ h f ⟹
(iso-tuple-fst-update isom ∘ h) f ∘ (iso-tuple-fst-update isom ∘ j) g =
(iso-tuple-fst-update isom ∘ j) g ∘ (iso-tuple-fst-update isom ∘ h) f
by (*clarsimp simp*: *iso-tuple-fst-update-def simps apfst-compose fun-eq-iff*)

lemma *iso-tuple-update-swap-snd-snd*:
h f ∘ j g = j g ∘ h f ⟹
(iso-tuple-snd-update isom ∘ h) f ∘ (iso-tuple-snd-update isom ∘ j) g =
(iso-tuple-snd-update isom ∘ j) g ∘ (iso-tuple-snd-update isom ∘ h) f
by (*clarsimp simp*: *iso-tuple-snd-update-def simps apsnd-compose fun-eq-iff*)

lemma *iso-tuple-update-swap-fst-snd*:
(iso-tuple-snd-update isom ∘ h) f ∘ (iso-tuple-fst-update isom ∘ j) g =
(iso-tuple-fst-update isom ∘ j) g ∘ (iso-tuple-snd-update isom ∘ h) f
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-update-def*
simps fun-eq-iff)

lemma *iso-tuple-update-swap-snd-fst*:
(iso-tuple-fst-update isom ∘ h) f ∘ (iso-tuple-snd-update isom ∘ j) g =
(iso-tuple-snd-update isom ∘ j) g ∘ (iso-tuple-fst-update isom ∘ h) f
by (*clarsimp simp*: *iso-tuple-fst-update-def iso-tuple-snd-update-def simps*
fun-eq-iff)

lemma *iso-tuple-update-compose-fst-fst*:
h f ∘ j g = k (f ∘ g) ⟹
(iso-tuple-fst-update isom ∘ h) f ∘ (iso-tuple-fst-update isom ∘ j) g =
(iso-tuple-fst-update isom ∘ k) (f ∘ g)
by (*clarsimp simp*: *iso-tuple-fst-update-def simps apfst-compose fun-eq-iff*)

lemma *iso-tuple-update-compose-snd-snd*:
h f ∘ j g = k (f ∘ g) ⟹
(iso-tuple-snd-update isom ∘ h) f ∘ (iso-tuple-snd-update isom ∘ j) g =
(iso-tuple-snd-update isom ∘ k) (f ∘ g)
by (*clarsimp simp*: *iso-tuple-snd-update-def simps apsnd-compose fun-eq-iff*)

lemma *iso-tuple-surjective-proof-assist-step*:
iso-tuple-surjective-proof-assist v a (iso-tuple-fst isom ∘ f) ⟹
iso-tuple-surjective-proof-assist v b (iso-tuple-snd isom ∘ f) ⟹
iso-tuple-surjective-proof-assist v (iso-tuple-cons isom a b) f
by (*clarsimp simp*: *iso-tuple-surjective-proof-assist-def simps*
iso-tuple-fst-def iso-tuple-snd-def iso-tuple-cons-def)

lemma *iso-tuple-fst-update-accessor-cong-assist*:
assumes *iso-tuple-update-accessor-cong-assist f g*
shows *iso-tuple-update-accessor-cong-assist*
(iso-tuple-fst-update isom ∘ f) (g ∘ iso-tuple-fst isom)
proof −

from *assms* **have** *f id = id*
by (*rule iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** *?thesis*
by (*clarsimp simp*: *iso-tuple-update-accessor-cong-assist-def simps*
iso-tuple-fst-update-def iso-tuple-fst-def)
qed

lemma *iso-tuple-snd-update-accessor-cong-assist*:
assumes *iso-tuple-update-accessor-cong-assist f g*
shows *iso-tuple-update-accessor-cong-assist*
(*iso-tuple-snd-update isom ∘ f*) (*g ∘ iso-tuple-snd isom*)
proof −
from *assms* **have** *f id = id*
by (*rule iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** *?thesis*
by (*clarsimp simp*: *iso-tuple-update-accessor-cong-assist-def simps*
iso-tuple-snd-update-def iso-tuple-snd-def)
qed

lemma *iso-tuple-fst-update-accessor-eq-assist*:
assumes *iso-tuple-update-accessor-eq-assist f g a u a′ v*
shows *iso-tuple-update-accessor-eq-assist*
(*iso-tuple-fst-update isom ∘ f*) (*g ∘ iso-tuple-fst isom*)
(*iso-tuple-cons isom a b*) *u* (*iso-tuple-cons isom a′ b*) *v*
proof −
from *assms* **have** *f id = id*
by (*auto simp add*: *iso-tuple-update-accessor-eq-assist-def*
intro: *iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** *?thesis*
by (*clarsimp simp*: *iso-tuple-update-accessor-eq-assist-def*
iso-tuple-fst-update-def iso-tuple-fst-def
iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma *iso-tuple-snd-update-accessor-eq-assist*:
assumes *iso-tuple-update-accessor-eq-assist f g b u b′ v*
shows *iso-tuple-update-accessor-eq-assist*
(*iso-tuple-snd-update isom ∘ f*) (*g ∘ iso-tuple-snd isom*)
(*iso-tuple-cons isom a b*) *u* (*iso-tuple-cons isom a b′*) *v*
proof −
from *assms* **have** *f id = id*
by (*auto simp add*: *iso-tuple-update-accessor-eq-assist-def*
intro: *iso-tuple-update-accessor-cong-assist-id*)
with *assms* **show** *?thesis*
by (*clarsimp simp*: *iso-tuple-update-accessor-eq-assist-def*
iso-tuple-snd-update-def iso-tuple-snd-def
iso-tuple-update-accessor-cong-assist-def iso-tuple-cons-def simps)
qed

lemma *iso-tuple-cons-conj-eqI*:
  $a = c \wedge b = d \wedge P \longleftrightarrow Q \Longrightarrow$
    iso-tuple-cons isom a b = iso-tuple-cons isom c d $\wedge$ $P \longleftrightarrow Q$
  by (*clarsimp simp*: *iso-tuple-cons-def simps*)

lemmas *intros* =
  iso-tuple-access-update-fst-fst
  iso-tuple-access-update-snd-snd
  iso-tuple-access-update-fst-snd
  iso-tuple-access-update-snd-fst
  iso-tuple-update-swap-fst-fst
  iso-tuple-update-swap-snd-snd
  iso-tuple-update-swap-fst-snd
  iso-tuple-update-swap-snd-fst
  iso-tuple-update-compose-fst-fst
  iso-tuple-update-compose-snd-snd
  iso-tuple-surjective-proof-assist-step
  iso-tuple-fst-update-accessor-eq-assist
  iso-tuple-snd-update-accessor-eq-assist
  iso-tuple-fst-update-accessor-cong-assist
  iso-tuple-snd-update-accessor-cong-assist
  iso-tuple-cons-conj-eqI

end

lemma *isomorphic-tuple-intro*:
  fixes *repr abst*
  assumes *repr-inj*: $\bigwedge x\ y.$ *repr x = repr y* $\longleftrightarrow$ $x = y$
    and *abst-inv*: $\bigwedge z.$ *repr* (*abst z*) $= z$
    and *v*: $v \equiv$ *Tuple-Isomorphism repr abst*
  shows *isomorphic-tuple v*
proof
  fix *x* **have** *repr* (*abst* (*repr x*)) = *repr x*
    by (*simp add*: *abst-inv*)
  then show *Record.abst v* (*Record.repr v x*) = *x*
    by (*simp add*: *v repr-inj*)
next
  fix *y*
  show *Record.repr v* (*Record.abst v y*) = *y*
    by (*simp add*: *v*) (*fact abst-inv*)
qed

definition
  tuple-iso-tuple $\equiv$ *Tuple-Isomorphism id id*

lemma *tuple-iso-tuple*:
  isomorphic-tuple tuple-iso-tuple
  by (*simp add*: *isomorphic-tuple-intro* [*OF - - reflexive*] *tuple-iso-tuple-def*)

lemma *refl-conj-eq*: $Q = R \Longrightarrow P \wedge Q \longleftrightarrow P \wedge R$
by *simp*

lemma *iso-tuple-UNIV-I*: $x \in UNIV \equiv True$
by *simp*

lemma *iso-tuple-True-simp*: $(True \Longrightarrow PROP\ P) \equiv PROP\ P$
by *simp*

lemma *prop-subst*: $s = t \Longrightarrow PROP\ P\ t \Longrightarrow PROP\ P\ s$
by *simp*

lemma *K-record-comp*: $(\lambda x.\ c) \circ f = (\lambda x.\ c)$
by (*simp add*: *comp-def*)

86.4 Concrete record syntax

nonterminal
ident **and**
field-type **and**
field-types **and**
field **and**
fields **and**
field-update **and**
field-updates

open-bundle *record-syntax*
begin

syntax
-constify :: *id* => *ident* (‹-›)
-constify :: *longid* => *ident* (‹-›)

-field-type :: *ident* => *type* => *field-type* (‹(‹*indent=2 notation=‹infix field type›*›- ::/ -)›)
:: *field-type* => *field-types* (‹-›)
-field-types :: *field-type* => *field-types* => *field-types* (‹-,/ -›)
-record-type :: *field-types* => *type* (‹(‹*indent=3 notation=‹mixfix record type›*›(⦇-⦈)›)
-record-type-scheme :: *field-types* => *type* => *type* (‹(‹*indent=3 notation=‹mixfix record type›*›(⦇-,/ (‹*indent=2 notation=‹infix more type›*›... ::/ -)⦈)›)

-field :: *ident* => *'a* => *field* (‹(‹*indent=2 notation=‹infix field value›*›(‹*open-block markup=‹const›*›-) =/ -)›)
:: *field* => *fields* (‹-›)
-fields :: *field* => *fields* => *fields* (‹-,/ -›)
-record :: *fields* => *'a* (‹(‹*indent=3 notation=‹mixfix record value›*›(⦇-⦈)›)
-record-scheme :: *fields* => *'a* => *'a* (‹(‹*indent=3 notation=‹mixfix*

record value‹›(⁅-,/ (‹*indent=2 notation=‹infix more value*››... =/ -)⁆)›)

-field-update :: *ident* => *'a* => *field-update* (‹(‹*indent=2 notation=‹infix field update*››(‹*open-block markup=‹const*››-) :=/ -)›)
:: *field-update* => *field-updates* (‹-›)
-field-updates :: *field-update* => *field-updates* => *field-updates* (‹-,/ -›)
-record-update :: *'a* => *field-updates* => *'b* (‹(‹*open-block notation=‹mixfix record update*››-/(*3*(⁅-⁆))›) [*900*, *0*] *900*)

syntax (*ASCII*)
-record-type :: *field-types* => *type* (‹(‹*indent=3 notation=‹mixfix record type*››'(| - |'))›)
-record-type-scheme :: *field-types* => *type* => *type* (‹(‹*indent=3 notation=‹mixfix record type*››'(| -,/ (‹*indent=2 notation=‹infix more type*››... ::/ -) |'))›)
-record :: *fields* => *'a* (‹(‹*indent=3 notation=‹mixfix record value*››'(| - |'))›)
-record-scheme :: *fields* => *'a* => *'a* (‹(‹*indent=3 notation=‹mixfix record value*››'(| -,/ (‹*indent=2 notation=‹infix more value*››... =/ -) |'))›)
-record-update :: *'a* => *field-updates* => *'b* (‹(‹*open-block notation=‹mixfix record update*››-/(*3*'(| - |')))› [*900*, *0*] *900*)

end

86.5 Record package

ML-file ‹*Tools/record.ML*›

hide-const (**open**) *Tuple-Isomorphism repr abst iso-tuple-fst iso-tuple-snd iso-tuple-fst-update iso-tuple-snd-update iso-tuple-cons iso-tuple-surjective-proof-assist iso-tuple-update-accessor-cong-assist iso-tuple-update-accessor-eq-assist tuple-iso-tuple*

end

87 Greatest common divisor and least common multiple

theory *GCD*
imports *Groups-List Code-Numeral*
begin

87.1 Abstract bounded quasi semilattices as common foundation

locale *bounded-quasi-semilattice* = *abel-semigroup* +
fixes *top* :: *'a* (‹⊤›) **and** *bot* :: *'a* (‹⊥›)
and *normalize* :: *'a* ⇒ *'a*

assumes *idem-normalize* [*simp*]: $a * a = \mathit{normalize}\ a$
and *normalize-left-idem* [*simp*]: $\mathit{normalize}\ a * b = a * b$
and *normalize-idem* [*simp*]: $\mathit{normalize}\ (a * b) = a * b$
and *normalize-top* [*simp*]: $\mathit{normalize}\ \top = \top$
and *normalize-bottom* [*simp*]: $\mathit{normalize}\ \bot = \bot$
and *top-left-normalize* [*simp*]: $\top * a = \mathit{normalize}\ a$
and *bottom-left-bottom* [*simp*]: $\bot * a = \bot$
begin

lemma *left-idem* [*simp*]:
$a * (a * b) = a * b$
using *assoc* [*of a a b*, *symmetric*] **by** *simp*

lemma *right-idem* [*simp*]:
$(a * b) * b = a * b$
using *left-idem* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *comp-fun-idem*: *comp-fun-idem f*
by *standard* (*simp-all add*: *fun-eq-iff ac-simps*)

interpretation *comp-fun-idem f*
by (*fact comp-fun-idem*)

lemma *top-right-normalize* [*simp*]:
$a * \top = \mathit{normalize}\ a$
using *top-left-normalize* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *bottom-right-bottom* [*simp*]:
$a * \bot = \bot$
using *bottom-left-bottom* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *normalize-right-idem* [*simp*]:
$a * \mathit{normalize}\ b = a * b$
using *normalize-left-idem* [*of b a*] **by** (*simp add*: *ac-simps*)

end

locale *bounded-quasi-semilattice-set* = *bounded-quasi-semilattice*
begin

interpretation *comp-fun-idem f*
by (*fact comp-fun-idem*)

definition $F :: 'a\ \mathit{set} \Rightarrow 'a$
where
eq-fold: $F\ A = (\mathit{if}\ \mathit{finite}\ A\ \mathit{then}\ \mathit{Finite\text{-}Set.fold}\ f\ \top\ A\ \mathit{else}\ \bot)$

lemma *infinite* [*simp*]:
$\mathit{infinite}\ A \Longrightarrow F\ A = \bot$

by (*simp add*: *eq-fold*)

lemma *set-eq-fold* [*code*]:
F (*set xs*) = *fold f xs* ⊤
by (*simp add*: *eq-fold fold-set-fold*)

lemma *empty* [*simp*]:
F {} = ⊤
by (*simp add*: *eq-fold*)

lemma *insert* [*simp*]:
F (*insert a A*) = *a* ∗ *F A*
by (*cases finite A*) (*simp-all add*: *eq-fold*)

lemma *normalize* [*simp*]:
normalize (*F A*) = *F A*
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *in-idem*:
assumes *a* ∈ *A*
shows *a* ∗ *F A* = *F A*
using *assms* **by** (*induct A rule*: *infinite-finite-induct*)
(*auto simp*: *left-commute* [*of a*])

lemma *union*:
F (*A* ∪ *B*) = *F A* ∗ *F B*
by (*induct A rule*: *infinite-finite-induct*)
(*simp-all add*: *ac-simps*)

lemma *remove*:
assumes *a* ∈ *A*
shows *F A* = *a* ∗ *F* (*A* − {*a*})
proof −
from *assms* **obtain** *B* **where** *A* = *insert a B* **and** *a* ∉ *B*
by (*blast dest*: *mk-disjoint-insert*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *insert-remove*:
F (*insert a A*) = *a* ∗ *F* (*A* − {*a*})
by (*cases a* ∈ *A*) (*simp-all add*: *insert-absorb remove*)

lemma *subset*:
assumes *B* ⊆ *A*
shows *F B* ∗ *F A* = *F A*
using *assms* **by** (*simp add*: *union* [*symmetric*] *Un-absorb1*)

end

87.2 Abstract GCD and LCM

class *gcd* = *zero* + *one* + *dvd* +
  fixes *gcd* :: *'a* ⇒ *'a* ⇒ *'a*
    and *lcm* :: *'a* ⇒ *'a* ⇒ *'a*

class *Gcd* = *gcd* +
  fixes *Gcd* :: *'a set* ⇒ *'a*
    and *Lcm* :: *'a set* ⇒ *'a*

syntax
  -GCD1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder GCD›› GCD -./ -)› [*0*, *10*] *10*)
  -GCD :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder GCD›› GCD -∈-./ -)› [*0*, *0*, *10*] *10*)
  -LCM1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder LCM›› LCM -./ -)› [*0*, *10*] *10*)
  -LCM :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent=3 notation=‹binder LCM›› LCM -∈-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
  -GCD1 -GCD ⇌ *Gcd* **and**
  -LCM1 -LCM ⇌ *Lcm*

translations
  GCD x y. f ⇌ *GCD x. GCD y. f*
  GCD x. f ⇌ *CONST Gcd* (*CONST range* (λ*x*. *f*))
  GCD x∈A. f ⇌ *CONST Gcd* ((λ*x*. *f*) ‘ *A*)
  LCM x y. f ⇌ *LCM x. LCM y. f*
  LCM x. f ⇌ *CONST Lcm* (*CONST range* (λ*x*. *f*))
  LCM x∈A. f ⇌ *CONST Lcm* ((λ*x*. *f*) ‘ *A*)

class *semiring-gcd* = *normalization-semidom* + *gcd* +
  assumes *gcd-dvd1* [*iff*]: *gcd a b dvd a*
    and *gcd-dvd2* [*iff*]: *gcd a b dvd b*
    and *gcd-greatest*: *c dvd a* ⟹ *c dvd b* ⟹ *c dvd gcd a b*
    and *normalize-gcd* [*simp*]: *normalize* (*gcd a b*) = *gcd a b*
    and *lcm-gcd*: *lcm a b* = *normalize* (*a* ∗ *b div gcd a b*)
begin

lemma *gcd-greatest-iff* [*simp*]: *a dvd gcd b c* ⟷ *a dvd b* ∧ *a dvd c*
  by (*blast intro*!: *gcd-greatest intro*: *dvd-trans*)

lemma *gcd-dvdI1*: *a dvd c* ⟹ *gcd a b dvd c*
  by (*rule dvd-trans*) (*rule gcd-dvd1*)

lemma *gcd-dvdI2*: *b dvd c* ⟹ *gcd a b dvd c*
  by (*rule dvd-trans*) (*rule gcd-dvd2*)

lemma *dvd-gcdD1*: *a dvd gcd b c* ⟹ *a dvd b*

using *gcd-dvd1* [*of b c*] **by** (*blast intro*: *dvd-trans*)

lemma *dvd-gcdD2*: *a dvd gcd b c* ⟹ *a dvd c*
using *gcd-dvd2* [*of b c*] **by** (*blast intro*: *dvd-trans*)

lemma *gcd-0-left* [*simp*]: *gcd 0 a = normalize a*
by (*rule associated-eqI*) *simp-all*

lemma *gcd-0-right* [*simp*]: *gcd a 0 = normalize a*
by (*rule associated-eqI*) *simp-all*

lemma *gcd-eq-0-iff* [*simp*]: *gcd a b = 0* ⟷ *a = 0* ∧ *b = 0*
(**is** *?P* ⟷ *?Q*)
proof
assume *?P*
then have *0 dvd gcd a b*
by *simp*
then have *0 dvd a* **and** *0 dvd b*
by (*blast intro*: *dvd-trans*)+
then show *?Q*
by *simp*
next
assume *?Q*
then show *?P*
by *simp*
qed

lemma *unit-factor-gcd*: *unit-factor* (*gcd a b*) = (*if a = 0* ∧ *b = 0 then 0 else 1*)
proof (*cases gcd a b = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *unit-factor* (*gcd a b*) ∗ *normalize* (*gcd a b*) = *gcd a b*
by (*rule unit-factor-mult-normalize*)
then have *unit-factor* (*gcd a b*) ∗ *gcd a b* = *gcd a b*
by *simp*
then have *unit-factor* (*gcd a b*) ∗ *gcd a b div gcd a b* = *gcd a b div gcd a b*
by *simp*
with *False* **show** *?thesis*
by *simp*
qed

lemma *is-unit-gcd-iff* [*simp*]:
is-unit (*gcd a b*) ⟷ *gcd a b = 1*
by (*cases a = 0* ∧ *b = 0*) (*auto simp*: *unit-factor-gcd dest*: *is-unit-unit-factor*)

sublocale *gcd*: *abel-semigroup gcd*
proof

fix *a b c*
show *gcd a b = gcd b a*
by (*rule associated-eqI*) *simp-all*
from *gcd-dvd1* **have** *gcd (gcd a b) c dvd a*
by (*rule dvd-trans*) *simp*
moreover from *gcd-dvd1* **have** *gcd (gcd a b) c dvd b*
by (*rule dvd-trans*) *simp*
ultimately have *P1*: *gcd (gcd a b) c dvd gcd a (gcd b c)*
by (*auto intro*!: *gcd-greatest*)
from *gcd-dvd2* **have** *gcd a (gcd b c) dvd b*
by (*rule dvd-trans*) *simp*
moreover from *gcd-dvd2* **have** *gcd a (gcd b c) dvd c*
by (*rule dvd-trans*) *simp*
ultimately have *P2*: *gcd a (gcd b c) dvd gcd (gcd a b) c*
by (*auto intro*!: *gcd-greatest*)
from *P1 P2* **show** *gcd (gcd a b) c = gcd a (gcd b c)*
by (*rule associated-eqI*) *simp-all*
qed

sublocale *gcd*: *bounded-quasi-semilattice gcd 0 1 normalize*
proof
show *gcd a a = normalize a* **for** *a*
proof −
have *a dvd gcd a a*
by (*rule gcd-greatest*) *simp-all*
then show *?thesis*
by (*auto intro*: *associated-eqI*)
qed
show *gcd (normalize a) b = gcd a b* **for** *a b*
using *gcd-dvd1* [*of normalize a b*]
by (*auto intro*: *associated-eqI*)
show *gcd 1 a = 1* **for** *a*
by (*rule associated-eqI*) *simp-all*
qed *simp-all*

lemma *gcd-self*: *gcd a a = normalize a*
by (*fact gcd.idem-normalize*)

lemma *gcd-left-idem*: *gcd a (gcd a b) = gcd a b*
by (*fact gcd.left-idem*)

lemma *gcd-right-idem*: *gcd (gcd a b) b = gcd a b*
by (*fact gcd.right-idem*)

lemma *gcdI*:
assumes *c dvd a* **and** *c dvd b*
and *greatest*: $\bigwedge d.$ *d dvd a* $\Longrightarrow$ *d dvd b* $\Longrightarrow$ *d dvd c*
and *normalize c = c*
shows *c = gcd a b*

by (*rule associated-eqI*) (*auto simp*: *assms intro*: *gcd-greatest*)

lemma *gcd-unique*:
d dvd a ∧ *d dvd b* ∧ *normalize d* = *d* ∧ (∀ *e*. *e dvd a* ∧ *e dvd b* ⟶ *e dvd d*) ⟷ *d* = *gcd a b*
by *rule* (*auto intro*: *gcdI simp*: *gcd-greatest*)

lemma *gcd-dvd-prod*: *gcd a b dvd k* ∗ *b*
using *mult-dvd-mono* [*of 1*] **by** *auto*

lemma *gcd-proj2-if-dvd*: *b dvd a* ⟹ *gcd a b* = *normalize b*
by (*rule gcdI* [*symmetric*]) *simp-all*

lemma *gcd-proj1-if-dvd*: *a dvd b* ⟹ *gcd a b* = *normalize a*
by (*rule gcdI* [*symmetric*]) *simp-all*

lemma *gcd-proj1-iff*: *gcd m n* = *normalize m* ⟷ *m dvd n*
proof
assume ∗: *gcd m n* = *normalize m*
show *m dvd n*
proof (*cases m* = *0*)
case *True*
with ∗ **show** *?thesis* **by** *simp*
next
case [*simp*]: *False*
from ∗ **have** ∗∗: *m* = *gcd m n* ∗ *unit-factor m*
by (*simp add*: *unit-eq-div2*)
show *?thesis*
by (*subst* ∗∗) (*simp add*: *mult-unit-dvd-iff*)
qed
next
assume *m dvd n*
then show *gcd m n* = *normalize m*
by (*rule gcd-proj1-if-dvd*)
qed

lemma *gcd-proj2-iff*: *gcd m n* = *normalize n* ⟷ *n dvd m*
using *gcd-proj1-iff* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *gcd-mult-left*: *gcd* (*c* ∗ *a*) (*c* ∗ *b*) = *normalize* (*c* ∗ *gcd a b*)
proof (*cases c* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have ∗: *c* ∗ *gcd a b dvd gcd* (*c* ∗ *a*) (*c* ∗ *b*)
by (*auto intro*: *gcd-greatest*)
moreover from *False* ∗ **have** *gcd* (*c* ∗ *a*) (*c* ∗ *b*) *dvd c* ∗ *gcd a b*
by (*metis div-dvd-iff-mult dvd-mult-left gcd-dvd1 gcd-dvd2 gcd-greatest mult-commute*)

ultimately have *normalize* (*gcd* (*c* ∗ *a*) (*c* ∗ *b*)) = *normalize* (*c* ∗ *gcd a b*)
by (*auto intro*: *associated-eqI*)
then show *?thesis*
by (*simp add*: *normalize-mult*)
qed

lemma *gcd-mult-right*: *gcd* (*a* ∗ *c*) (*b* ∗ *c*) = *normalize* (*gcd b a* ∗ *c*)
using *gcd-mult-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *dvd-lcm1* [*iff*]: *a dvd lcm a b*
by (*metis div-mult-swap dvd-mult2 dvd-normalize-iff dvd-refl gcd-dvd2 lcm-gcd*)

lemma *dvd-lcm2* [*iff*]: *b dvd lcm a b*
by (*metis dvd-div-mult dvd-mult dvd-normalize-iff dvd-refl gcd-dvd1 lcm-gcd*)

lemma *dvd-lcmI1*: *a dvd b* ⟹ *a dvd lcm b c*
by (*rule dvd-trans*) (*assumption*, *blast*)

lemma *dvd-lcmI2*: *a dvd c* ⟹ *a dvd lcm b c*
by (*rule dvd-trans*) (*assumption*, *blast*)

lemma *lcm-dvdD1*: *lcm a b dvd c* ⟹ *a dvd c*
using *dvd-lcm1* [*of a b*] **by** (*blast intro*: *dvd-trans*)

lemma *lcm-dvdD2*: *lcm a b dvd c* ⟹ *b dvd c*
using *dvd-lcm2* [*of a b*] **by** (*blast intro*: *dvd-trans*)

lemma *lcm-least*:
assumes *a dvd c* **and** *b dvd c*
shows *lcm a b dvd c*
proof (*cases c* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have ∗: *is-unit* (*unit-factor c*)
by *simp*
show *?thesis*
proof (*cases gcd a b* = *0*)
case *True*
with *assms* **show** *?thesis* **by** *simp*
next
case *False*
have *a* ∗ *b dvd normalize* (*c* ∗ *gcd a b*)
using *assms* **by** (*subst gcd-mult-left* [*symmetric*]) (*auto intro*!: *gcd-greatest simp*: *mult-ac*)
with *False* **have** (*a* ∗ *b div gcd a b*) *dvd c*
by (*subst div-dvd-iff-mult*) *auto*
thus *?thesis* **by** (*simp add*: *lcm-gcd*)

qed
qed

lemma *lcm-least-iff* [*simp*]: *lcm a b dvd c* ⟷ *a dvd c* ∧ *b dvd c*
by (*blast intro*!: *lcm-least intro*: *dvd-trans*)

lemma *normalize-lcm* [*simp*]: *normalize* (*lcm a b*) = *lcm a b*
by (*simp add*: *lcm-gcd dvd-normalize-div*)

lemma *lcm-0-left* [*simp*]: *lcm 0 a* = *0*
by (*simp add*: *lcm-gcd*)

lemma *lcm-0-right* [*simp*]: *lcm a 0* = *0*
by (*simp add*: *lcm-gcd*)

lemma *lcm-eq-0-iff*: *lcm a b* = *0* ⟷ *a* = *0* ∨ *b* = *0*
(**is** *?P* ⟷ *?Q*)
proof
assume *?P*
then have *0 dvd lcm a b*
by *simp*
also have *lcm a b dvd* (*a* ∗ *b*)
by *simp*
finally show *?Q*
by *auto*
next
assume *?Q*
then show *?P*
by *auto*
qed

lemma *zero-eq-lcm-iff*: *0* = *lcm a b* ⟷ *a* = *0* ∨ *b* = *0*
using *lcm-eq-0-iff*[*of a b*] **by** *auto*

lemma *lcm-eq-1-iff* [*simp*]: *lcm a b* = *1* ⟷ *is-unit a* ∧ *is-unit b*
by (*auto intro*: *associated-eqI*)

lemma *unit-factor-lcm*: *unit-factor* (*lcm a b*) = (*if a* = *0* ∨ *b* = *0 then 0 else 1*)
using *lcm-eq-0-iff*[*of a b*] **by** (*cases lcm a b* = *0*) (*auto simp*: *lcm-gcd*)

sublocale *lcm*: *abel-semigroup lcm*
proof
fix *a b c*
show *lcm a b* = *lcm b a*
by (*simp add*: *lcm-gcd ac-simps normalize-mult dvd-normalize-div*)
have *lcm* (*lcm a b*) *c dvd lcm a* (*lcm b c*)
and *lcm a* (*lcm b c*) *dvd lcm* (*lcm a b*) *c*
by (*auto intro*: *lcm-least*
dvd-trans [*of b lcm b c lcm a* (*lcm b c*)]

```
      dvd-trans [of c lcm b c lcm a (lcm b c)]
      dvd-trans [of a lcm a b lcm (lcm a b) c]
      dvd-trans [of b lcm a b lcm (lcm a b) c])
  then show lcm (lcm a b) c = lcm a (lcm b c)
    by (rule associated-eqI) simp-all
qed

sublocale lcm: bounded-quasi-semilattice lcm 1 0 normalize
proof
  show lcm a a = normalize a for a
  proof −
    have lcm a a dvd a
      by (rule lcm-least) simp-all
    then show ?thesis
      by (auto intro: associated-eqI)
  qed
  show lcm (normalize a) b = lcm a b for a b
    using dvd-lcm1 [of normalize a b] unfolding normalize-dvd-iff
    by (auto intro: associated-eqI)
  show lcm 1 a = normalize a for a
    by (rule associated-eqI) simp-all
qed simp-all

lemma lcm-self: lcm a a = normalize a
  by (fact lcm.idem-normalize)

lemma lcm-left-idem: lcm a (lcm a b) = lcm a b
  by (fact lcm.left-idem)

lemma lcm-right-idem: lcm (lcm a b) b = lcm a b
  by (fact lcm.right-idem)

lemma gcd-lcm:
  assumes a ≠ 0 and b ≠ 0
  shows gcd a b = normalize (a * b div lcm a b)
proof −
  from assms have [simp]: a * b div gcd a b ≠ 0
    by (subst dvd-div-eq-0-iff) auto
  let ?u = unit-factor (a * b div gcd a b)
  have gcd a b * normalize (a * b div gcd a b) =
        gcd a b * ((a * b div gcd a b) * (1 div ?u))
    by simp
  also have ... = a * b div ?u
    by (subst mult.assoc [symmetric]) auto
  also have ... dvd a * b
    by (subst div-unit-dvd-iff) auto
  finally have gcd a b dvd ((a * b) div lcm a b)
    by (intro dvd-mult-imp-div) (auto simp: lcm-gcd)
  moreover have a * b div lcm a b dvd a and a * b div lcm a b dvd b
```

using *assms* **by** (*subst div-dvd-iff-mult*; *simp add*: *lcm-eq-0-iff mult.commute*[*of b lcm a b*])+
ultimately have *normalize* (*gcd a b*) = *normalize* (*a* ∗ *b div lcm a b*)
apply −
apply (*rule associated-eqI*)
using *assms*
apply (*auto simp*: *div-dvd-iff-mult zero-eq-lcm-iff*)
done
thus *?thesis* **by** *simp*
qed

lemma *lcm-1-left*: *lcm 1 a* = *normalize a*
by (*fact lcm.top-left-normalize*)

lemma *lcm-1-right*: *lcm a 1* = *normalize a*
by (*fact lcm.top-right-normalize*)

lemma *lcm-mult-left*: *lcm* (*c* ∗ *a*) (*c* ∗ *b*) = *normalize* (*c* ∗ *lcm a b*)
proof (*cases c* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have ∗: *lcm* (*c* ∗ *a*) (*c* ∗ *b*) *dvd c* ∗ *lcm a b*
by (*auto intro*: *lcm-least*)
moreover have *lcm a b dvd lcm* (*c* ∗ *a*) (*c* ∗ *b*) *div c*
by (*intro lcm-least*) (*auto intro*!: *dvd-mult-imp-div simp*: *mult-ac*)
hence *c* ∗ *lcm a b dvd lcm* (*c* ∗ *a*) (*c* ∗ *b*)
using *False* **by** (*subst* (*asm*) *dvd-div-iff-mult*) (*auto simp*: *mult-ac intro*: *dvd-lcmI1*)
ultimately have *normalize* (*lcm* (*c* ∗ *a*) (*c* ∗ *b*)) = *normalize* (*c* ∗ *lcm a b*)
by (*auto intro*: *associated-eqI*)
then show *?thesis*
by (*simp add*: *normalize-mult*)
qed

lemma *lcm-mult-right*: *lcm* (*a* ∗ *c*) (*b* ∗ *c*) = *normalize* (*lcm b a* ∗ *c*)
using *lcm-mult-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *lcm-mult-unit1*: *is-unit a* ⟹ *lcm* (*b* ∗ *a*) *c* = *lcm b c*
by (*rule associated-eqI*) (*simp-all add*: *mult-unit-dvd-iff dvd-lcmI1*)

lemma *lcm-mult-unit2*: *is-unit a* ⟹ *lcm b* (*c* ∗ *a*) = *lcm b c*
using *lcm-mult-unit1* [*of a c b*] **by** (*simp add*: *ac-simps*)

lemma *lcm-div-unit1*:
is-unit a ⟹ *lcm* (*b div a*) *c* = *lcm b c*
by (*erule is-unitE* [*of* - *b*]) (*simp add*: *lcm-mult-unit1*)

lemma *lcm-div-unit2*: *is-unit a* ⟹ *lcm b* (*c div a*) = *lcm b c*

by (*erule is-unitE* [*of* - *c*]) (*simp add*: *lcm-mult-unit2*)

lemma *normalize-lcm-left*: *lcm* (*normalize a*) *b* = *lcm a b*
by (*fact lcm.normalize-left-idem*)

lemma *normalize-lcm-right*: *lcm a* (*normalize b*) = *lcm a b*
by (*fact lcm.normalize-right-idem*)

lemma *comp-fun-idem-gcd*: *comp-fun-idem gcd*
by *standard* (*simp-all add*: *fun-eq-iff ac-simps*)

lemma *comp-fun-idem-lcm*: *comp-fun-idem lcm*
by *standard* (*simp-all add*: *fun-eq-iff ac-simps*)

lemma *gcd-dvd-antisym*: *gcd a b dvd gcd c d* ⟹ *gcd c d dvd gcd a b* ⟹ *gcd a b* = *gcd c d*
proof (*rule gcdI*)
assume ∗: *gcd a b dvd gcd c d*
and ∗∗: *gcd c d dvd gcd a b*
have *gcd c d dvd c*
by *simp*
with ∗ **show** *gcd a b dvd c*
by (*rule dvd-trans*)
have *gcd c d dvd d*
by *simp*
with ∗ **show** *gcd a b dvd d*
by (*rule dvd-trans*)
show *normalize* (*gcd a b*) = *gcd a b*
by *simp*
fix *l* **assume** *l dvd c* **and** *l dvd d*
then have *l dvd gcd c d*
by (*rule gcd-greatest*)
from *this* **and** ∗∗ **show** *l dvd gcd a b*
by (*rule dvd-trans*)
qed

declare *unit-factor-lcm* [*simp*]

lemma *lcmI*:
assumes *a dvd c* **and** *b dvd c* **and** ⋀*d*. *a dvd d* ⟹ *b dvd d* ⟹ *c dvd d*
and *normalize c* = *c*
shows *c* = *lcm a b*
by (*rule associated-eqI*) (*auto simp*: *assms intro*: *lcm-least*)

lemma *gcd-dvd-lcm* [*simp*]: *gcd a b dvd lcm a b*
using *gcd-dvd2* **by** (*rule dvd-lcmI2*)

lemmas *lcm-0* = *lcm-0-right*

lemma *lcm-unique*:
a dvd d ∧ *b dvd d* ∧ *normalize d* = *d* ∧ (∀ *e*. *a dvd e* ∧ *b dvd e* ⟶ *d dvd e*) ⟷ *d* = *lcm a b*
by *rule* (*auto intro*: *lcmI simp*: *lcm-least lcm-eq-0-iff*)

lemma *lcm-proj1-if-dvd*:
assumes *b dvd a* **shows** *lcm a b* = *normalize a*
proof −
have *normalize* (*lcm a b*) = *normalize a*
by (*rule associatedI*) (*use assms* **in** *auto*)
thus *?thesis* **by** *simp*
qed

lemma *lcm-proj2-if-dvd*: *a dvd b* ⟹ *lcm a b* = *normalize b*
using *lcm-proj1-if-dvd* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *lcm-proj1-iff*: *lcm m n* = *normalize m* ⟷ *n dvd m*
proof
assume ∗: *lcm m n* = *normalize m*
show *n dvd m*
proof (*cases m* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case [*simp*]: *False*
from ∗ **have** ∗∗: *m* = *lcm m n* ∗ *unit-factor m*
by (*simp add*: *unit-eq-div2*)
show *?thesis* **by** (*subst* ∗∗) *simp*
qed
next
assume *n dvd m*
then show *lcm m n* = *normalize m*
by (*rule lcm-proj1-if-dvd*)
qed

lemma *lcm-proj2-iff*: *lcm m n* = *normalize n* ⟷ *m dvd n*
using *lcm-proj1-iff* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *gcd-mono*: *a dvd c* ⟹ *b dvd d* ⟹ *gcd a b dvd gcd c d*
by (*simp add*: *gcd-dvdI1 gcd-dvdI2*)

lemma *lcm-mono*: *a dvd c* ⟹ *b dvd d* ⟹ *lcm a b dvd lcm c d*
by (*simp add*: *dvd-lcmI1 dvd-lcmI2*)

lemma *dvd-productE*:
assumes *p dvd a* ∗ *b*
obtains *x y* **where** *p* = *x* ∗ *y x dvd a y dvd b*
proof (*cases a* = *0*)
case *True*

thus *?thesis* **by** (*intro that*[*of p 1*]) *simp-all*
next
case *False*
define *x y* **where** $x = gcd\ a\ p$ **and** $y = p\ div\ x$
have $p = x * y$ **by** (*simp add*: *x-def y-def*)
moreover have *x dvd a* **by** (*simp add*: *x-def*)
moreover from *assms* **have** *p dvd gcd* $(b * a)$ $(b * p)$
by (*intro gcd-greatest*) (*simp-all add*: *mult.commute*)
hence *p dvd b* $*$ *gcd a p* **by** (*subst* (*asm*) *gcd-mult-left*) *auto*
with *False* **have** *y dvd b*
by (*simp add*: *x-def y-def div-dvd-iff-mult assms*)
ultimately show *?thesis* **by** (*rule that*)
qed

lemma *gcd-mult-unit1*:
assumes *is-unit a* **shows** *gcd* $(b * a)$ $c = gcd\ b\ c$
proof −
have *gcd* $(b * a)$ *c dvd b*
using *assms dvd-mult-unit-iff* **by** *blast*
then show *?thesis*
by (*rule gcdI*) *simp-all*
qed

lemma *gcd-mult-unit2*: *is-unit a* $\Longrightarrow$ *gcd b* $(c * a) = gcd\ b\ c$
using *gcd.commute gcd-mult-unit1* **by** *auto*

lemma *gcd-div-unit1*: *is-unit a* $\Longrightarrow$ *gcd* $(b\ div\ a)$ $c = gcd\ b\ c$
by (*erule is-unitE* [*of* - *b*]) (*simp add*: *gcd-mult-unit1*)

lemma *gcd-div-unit2*: *is-unit a* $\Longrightarrow$ *gcd b* $(c\ div\ a) = gcd\ b\ c$
by (*erule is-unitE* [*of* - *c*]) (*simp add*: *gcd-mult-unit2*)

lemma *normalize-gcd-left*: *gcd* (*normalize a*) $b = gcd\ a\ b$
by (*fact gcd.normalize-left-idem*)

lemma *normalize-gcd-right*: *gcd a* (*normalize b*) $= gcd\ a\ b$
by (*fact gcd.normalize-right-idem*)

lemma *gcd-add1* [*simp*]: *gcd* $(m + n)$ $n = gcd\ m\ n$
by (*rule gcdI* [*symmetric*]) (*simp-all add*: *dvd-add-left-iff*)

lemma *gcd-add2* [*simp*]: *gcd m* $(m + n) = gcd\ m\ n$
using *gcd-add1* [*of n m*] **by** (*simp add*: *ac-simps*)

lemma *gcd-add-mult*: *gcd m* $(k * m + n) = gcd\ m\ n$
by (*rule gcdI* [*symmetric*]) (*simp-all add*: *dvd-add-right-iff*)

end

class *ring-gcd* = *comm-ring-1* + *semiring-gcd*
begin

lemma *gcd-neg1* [*simp*]: *gcd* (−*a*) *b* = *gcd a b*
by (*rule sym*, *rule gcdI*) (*simp-all add*: *gcd-greatest*)

lemma *gcd-neg2* [*simp*]: *gcd a* (−*b*) = *gcd a b*
by (*rule sym*, *rule gcdI*) (*simp-all add*: *gcd-greatest*)

lemma *gcd-neg-numeral-1* [*simp*]: *gcd* (− *numeral n*) *a* = *gcd* (*numeral n*) *a*
by (*fact gcd-neg1*)

lemma *gcd-neg-numeral-2* [*simp*]: *gcd a* (− *numeral n*) = *gcd a* (*numeral n*)
by (*fact gcd-neg2*)

lemma *gcd-diff1*: *gcd* (*m* − *n*) *n* = *gcd m n*
by (*subst diff-conv-add-uminus*, *subst gcd-neg2*[*symmetric*], *subst gcd-add1*, *simp*)

lemma *gcd-diff2*: *gcd* (*n* − *m*) *n* = *gcd m n*
by (*subst gcd-neg1*[*symmetric*]) (*simp only*: *minus-diff-eq gcd-diff1*)

lemma *lcm-neg1* [*simp*]: *lcm* (−*a*) *b* = *lcm a b*
by (*rule sym*, *rule lcmI*) (*simp-all add*: *lcm-least lcm-eq-0-iff*)

lemma *lcm-neg2* [*simp*]: *lcm a* (−*b*) = *lcm a b*
by (*rule sym*, *rule lcmI*) (*simp-all add*: *lcm-least lcm-eq-0-iff*)

lemma *lcm-neg-numeral-1* [*simp*]: *lcm* (− *numeral n*) *a* = *lcm* (*numeral n*) *a*
by (*fact lcm-neg1*)

lemma *lcm-neg-numeral-2* [*simp*]: *lcm a* (− *numeral n*) = *lcm a* (*numeral n*)
by (*fact lcm-neg2*)

end

class *semiring-Gcd* = *semiring-gcd* + *Gcd* +
assumes *Gcd-dvd*: *a* ∈ *A* ⟹ *Gcd A dvd a*
and *Gcd-greatest*: (⋀*b*. *b* ∈ *A* ⟹ *a dvd b*) ⟹ *a dvd Gcd A*
and *normalize-Gcd* [*simp*]: *normalize* (*Gcd A*) = *Gcd A*
assumes *dvd-Lcm*: *a* ∈ *A* ⟹ *a dvd Lcm A*
and *Lcm-least*: (⋀*b*. *b* ∈ *A* ⟹ *b dvd a*) ⟹ *Lcm A dvd a*
and *normalize-Lcm* [*simp*]: *normalize* (*Lcm A*) = *Lcm A*
begin

lemma *Lcm-Gcd*: *Lcm A* = *Gcd* {*b*. ∀ *a*∈*A*. *a dvd b*}
by (*rule associated-eqI*) (*auto intro*: *Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least*)

lemma *Gcd-Lcm*: *Gcd A* = *Lcm* {*b*. ∀ *a*∈*A*. *b dvd a*}
by (*rule associated-eqI*) (*auto intro*: *Gcd-dvd dvd-Lcm Gcd-greatest Lcm-least*)

lemma *Gcd-empty* [*simp*]: *Gcd* {} = *0*
by (*rule dvd-0-left*, *rule Gcd-greatest*) *simp*

lemma *Lcm-empty* [*simp*]: *Lcm* {} = *1*
by (*auto intro*: *associated-eqI Lcm-least*)

lemma *Gcd-insert* [*simp*]: *Gcd* (*insert a A*) = *gcd a* (*Gcd A*)
proof −
have *Gcd* (*insert a A*) *dvd gcd a* (*Gcd A*)
by (*auto intro*: *Gcd-dvd Gcd-greatest*)
moreover have *gcd a* (*Gcd A*) *dvd Gcd* (*insert a A*)
proof (*rule Gcd-greatest*)
fix *b*
assume *b* ∈ *insert a A*
then show *gcd a* (*Gcd A*) *dvd b*
proof
assume *b* = *a*
then show *?thesis*
by *simp*
next
assume *b* ∈ *A*
then have *Gcd A dvd b*
by (*rule Gcd-dvd*)
moreover have *gcd a* (*Gcd A*) *dvd Gcd A*
by *simp*
ultimately show *?thesis*
by (*blast intro*: *dvd-trans*)
qed
qed
ultimately show *?thesis*
by (*auto intro*: *associated-eqI*)
qed

lemma *Lcm-insert* [*simp*]: *Lcm* (*insert a A*) = *lcm a* (*Lcm A*)
proof (*rule sym*)
have *lcm a* (*Lcm A*) *dvd Lcm* (*insert a A*)
by (*auto intro*: *dvd-Lcm Lcm-least*)
moreover have *Lcm* (*insert a A*) *dvd lcm a* (*Lcm A*)
proof (*rule Lcm-least*)
fix *b*
assume *b* ∈ *insert a A*
then show *b dvd lcm a* (*Lcm A*)
proof
assume *b* = *a*
then show *?thesis* **by** *simp*
next
assume *b* ∈ *A*
then have *b dvd Lcm A*

```
      by (rule dvd-Lcm)
    moreover have Lcm A dvd lcm a (Lcm A)
      by simp
    ultimately show ?thesis
      by (blast intro: dvd-trans)
  qed
 qed
 ultimately show lcm a (Lcm A) = Lcm (insert a A)
  by (rule associated-eqI) (simp-all add: lcm-eq-0-iff)
qed

lemma LcmI:
 assumes ⋀a. a ∈ A ⟹ a dvd b
  and ⋀c. (⋀a. a ∈ A ⟹ a dvd c) ⟹ b dvd c
  and normalize b = b
 shows b = Lcm A
 by (rule associated-eqI) (auto simp: assms dvd-Lcm intro: Lcm-least)

lemma Lcm-subset: A ⊆ B ⟹ Lcm A dvd Lcm B
 by (blast intro: Lcm-least dvd-Lcm)

lemma Lcm-Un: Lcm (A ∪ B) = lcm (Lcm A) (Lcm B)
proof −
 have ⋀d. ⟦Lcm A dvd d; Lcm B dvd d⟧ ⟹ Lcm (A ∪ B) dvd d
  by (meson UnE Lcm-least dvd-Lcm dvd-trans)
 then show ?thesis
  by (meson Lcm-subset lcm-unique normalize-Lcm sup.cobounded1 sup.cobounded2)
qed

lemma Gcd-0-iff [simp]: Gcd A = 0 ⟷ A ⊆ {0}
 (is ?P ⟷ ?Q)
proof
 assume ?P
 show ?Q
 proof
  fix a
  assume a ∈ A
  then have Gcd A dvd a
   by (rule Gcd-dvd)
  with ‹?P› have a = 0
   by simp
  then show a ∈ {0}
   by simp
 qed
next
 assume ?Q
 have 0 dvd Gcd A
 proof (rule Gcd-greatest)
  fix a
```

```
    assume a ∈ A
    with ‹?Q› have a = 0
      by auto
    then show 0 dvd a
      by simp
  qed
  then show ?P
    by simp
qed

lemma Lcm-1-iff [simp]: Lcm A = 1 ⟷ (∀ a∈A. is-unit a)
  (is ?P ⟷ ?Q)
proof
  assume ?P
  show ?Q
  proof
    fix a
    assume a ∈ A
    then have a dvd Lcm A
      by (rule dvd-Lcm)
    with ‹?P› show is-unit a
      by simp
  qed
next
  assume ?Q
  then have is-unit (Lcm A)
    by (blast intro: Lcm-least)
  then have normalize (Lcm A) = 1
    by (rule is-unit-normalize)
  then show ?P
    by simp
qed

lemma unit-factor-Lcm: unit-factor (Lcm A) = (if Lcm A = 0 then 0 else 1)
proof (cases Lcm A = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  with unit-factor-normalize have unit-factor (normalize (Lcm A)) = 1
    by blast
  with False show ?thesis
    by simp
qed

lemma unit-factor-Gcd: unit-factor (Gcd A) = (if Gcd A = 0 then 0 else 1)
  by (simp add: Gcd-Lcm unit-factor-Lcm)
```

lemma *GcdI*:
  assumes $\bigwedge a.\ a \in A \Longrightarrow b\ dvd\ a$
    and $\bigwedge c.\ (\bigwedge a.\ a \in A \Longrightarrow c\ dvd\ a) \Longrightarrow c\ dvd\ b$
    and *normalize* $b = b$
  shows $b = Gcd\ A$
  by (*rule associated-eqI*) (*auto simp*: *assms Gcd-dvd intro*: *Gcd-greatest*)

lemma *Gcd-eq-1-I*:
  assumes *is-unit a* **and** $a \in A$
  shows $Gcd\ A = 1$
proof −
  from *assms* **have** *is-unit* $(Gcd\ A)$
    by (*blast intro*: *Gcd-dvd dvd-unit-imp-unit*)
  then have *normalize* $(Gcd\ A) = 1$
    by (*rule is-unit-normalize*)
  then show *?thesis*
    by *simp*
qed

lemma *Lcm-eq-0-I*:
  assumes $0 \in A$
  shows $Lcm\ A = 0$
proof −
  from *assms* **have** *0 dvd Lcm A*
    by (*rule dvd-Lcm*)
  then show *?thesis*
    by *simp*
qed

lemma *Gcd-UNIV* [*simp*]: $Gcd\ UNIV = 1$
  using *dvd-refl* **by** (*rule Gcd-eq-1-I*) *simp*

lemma *Lcm-UNIV* [*simp*]: $Lcm\ UNIV = 0$
  by (*rule Lcm-eq-0-I*) *simp*

lemma *Lcm-0-iff*:
  assumes *finite A*
  shows $Lcm\ A = 0 \longleftrightarrow 0 \in A$
proof (*cases* $A = \{\}$)
  case *True*
  then show *?thesis* **by** *simp*
next
  case *False*
  with *assms* **show** *?thesis*
    by (*induct A rule*: *finite-ne-induct*) (*auto simp*: *lcm-eq-0-iff*)
qed

lemma *Gcd-image-normalize* [*simp*]: $Gcd\ (normalize\ `\ A) = Gcd\ A$
proof −

```
  have Gcd (normalize ' A) dvd a if a ∈ A for a
  proof −
    from that obtain B where A = insert a B
      by blast
    moreover have gcd (normalize a) (Gcd (normalize ' B)) dvd normalize a
      by (rule gcd-dvd1)
    ultimately show Gcd (normalize ' A) dvd a
      by simp
  qed
  then have Gcd (normalize ' A) dvd Gcd A and Gcd A dvd Gcd (normalize ' A)
    by (auto intro!: Gcd-greatest intro: Gcd-dvd)
  then show ?thesis
    by (auto intro: associated-eqI)
qed

lemma Gcd-eqI:
  assumes normalize a = a
  assumes ⋀b. b ∈ A ⟹ a dvd b
    and ⋀c. (⋀b. b ∈ A ⟹ c dvd b) ⟹ c dvd a
  shows Gcd A = a
  using assms by (blast intro: associated-eqI Gcd-greatest Gcd-dvd normalize-Gcd)

lemma dvd-GcdD: x dvd Gcd A ⟹ y ∈ A ⟹ x dvd y
  using Gcd-dvd dvd-trans by blast

lemma dvd-Gcd-iff: x dvd Gcd A ⟷ (∀ y∈A. x dvd y)
  by (blast dest: dvd-GcdD intro: Gcd-greatest)

lemma Gcd-mult: Gcd ((∗) c ' A) = normalize (c ∗ Gcd A)
proof (cases c = 0)
  case True
  then show ?thesis by auto
next
  case [simp]: False
  have Gcd ((∗) c ' A) div c dvd Gcd A
    by (intro Gcd-greatest, subst div-dvd-iff-mult)
       (auto intro!: Gcd-greatest Gcd-dvd simp: mult.commute[of - c])
  then have Gcd ((∗) c ' A) dvd c ∗ Gcd A
    by (subst (asm) div-dvd-iff-mult) (auto intro: Gcd-greatest simp: mult-ac)
  moreover have c ∗ Gcd A dvd Gcd ((∗) c ' A)
    by (intro Gcd-greatest) (auto intro: mult-dvd-mono Gcd-dvd)
  ultimately have normalize (Gcd ((∗) c ' A)) = normalize (c ∗ Gcd A)
    by (rule associatedI)
  then show ?thesis by simp
qed

lemma Lcm-eqI:
  assumes normalize a = a
    and ⋀b. b ∈ A ⟹ b dvd a
```

and $\bigwedge c.\ (\bigwedge b.\ b \in A \Longrightarrow b\ dvd\ c) \Longrightarrow a\ dvd\ c$
shows $Lcm\ A = a$
using *assms* **by** (*blast intro*: *associated-eqI Lcm-least dvd-Lcm normalize-Lcm*)

lemma *Lcm-dvdD*: $Lcm\ A\ dvd\ x \Longrightarrow y \in A \Longrightarrow y\ dvd\ x$
using *dvd-Lcm dvd-trans* **by** *blast*

lemma *Lcm-dvd-iff*: $Lcm\ A\ dvd\ x \longleftrightarrow (\forall\, y{\in}A.\ y\ dvd\ x)$
by (*blast dest*: *Lcm-dvdD intro*: *Lcm-least*)

lemma *Lcm-mult*:
assumes $A \neq \{\}$
shows $Lcm\ ((*)\ c\ `\ A) = normalize\ (c * Lcm\ A)$
proof (*cases* $c = 0$)
case *True*
with *assms* **have** $(*)\ c\ `\ A = \{0\}$
by *auto*
with *True* **show** *?thesis* **by** *auto*
next
case [*simp*]: *False*
from *assms* **obtain** x **where** x: $x \in A$
by *blast*
have $c\ dvd\ c * x$
by *simp*
also from x **have** $c * x\ dvd\ Lcm\ ((*)\ c\ `\ A)$
by (*intro dvd-Lcm*) *auto*
finally have *dvd*: $c\ dvd\ Lcm\ ((*)\ c\ `\ A)$ **.**
moreover have $Lcm\ A\ dvd\ Lcm\ ((*)\ c\ `\ A)\ div\ c$
by (*intro Lcm-least dvd-mult-imp-div*)
(*auto intro*!: *Lcm-least dvd-Lcm simp*: *mult.commute*[*of* - *c*])
ultimately have $c * Lcm\ A\ dvd\ Lcm\ ((*)\ c\ `\ A)$
by *auto*
moreover have $Lcm\ ((*)\ c\ `\ A)\ dvd\ c * Lcm\ A$
by (*intro Lcm-least*) (*auto intro*: *mult-dvd-mono dvd-Lcm*)
ultimately have $normalize\ (c * Lcm\ A) = normalize\ (Lcm\ ((*)\ c\ `\ A))$
by (*rule associatedI*)
then show *?thesis* **by** *simp*
qed

lemma *Lcm-no-units*: $Lcm\ A = Lcm\ (A - \{a.\ is\text{-}unit\ a\})$
proof −
have $(A - \{a.\ is\text{-}unit\ a\}) \cup \{a{\in}A.\ is\text{-}unit\ a\} = A$
by *blast*
then have $Lcm\ A = lcm\ (Lcm\ (A - \{a.\ is\text{-}unit\ a\}))\ (Lcm\ \{a{\in}A.\ is\text{-}unit\ a\})$
by (*simp add*: *Lcm-Un* [*symmetric*])
also have $Lcm\ \{a{\in}A.\ is\text{-}unit\ a\} = 1$
by *simp*
finally show *?thesis*
by *simp*

qed

lemma *Lcm-0-iff'*: $Lcm\ A = 0 \longleftrightarrow (\nexists l.\ l \neq 0 \wedge (\forall a{\in}A.\ a\ dvd\ l))$
  by (*metis Lcm-least dvd-0-left dvd-Lcm*)

lemma *Lcm-no-multiple*: $(\forall m.\ m \neq 0 \longrightarrow (\exists a{\in}A.\ \neg\ a\ dvd\ m)) \Longrightarrow Lcm\ A = 0$
  by (*auto simp*: *Lcm-0-iff'*)

lemma *Lcm-singleton* [*simp*]: $Lcm\ \{a\} = normalize\ a$
  by *simp*

lemma *Lcm-2* [*simp*]: $Lcm\ \{a, b\} = lcm\ a\ b$
  by *simp*

lemma *Gcd-1*: $1 \in A \Longrightarrow Gcd\ A = 1$
  by (*auto intro*!: *Gcd-eq-1-I*)

lemma *Gcd-singleton* [*simp*]: $Gcd\ \{a\} = normalize\ a$
  by *simp*

lemma *Gcd-2* [*simp*]: $Gcd\ \{a, b\} = gcd\ a\ b$
  by *simp*

lemma *Gcd-mono*:
  assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x\ dvd\ g\ x$
  shows $(GCD\ x{\in}A.\ f\ x)\ dvd\ (GCD\ x{\in}A.\ g\ x)$
proof (*intro Gcd-greatest*, *safe*)
  fix x **assume** $x \in A$
  hence $(GCD\ x{\in}A.\ f\ x)\ dvd\ f\ x$
    by (*intro Gcd-dvd*) *auto*
  also have $f\ x\ dvd\ g\ x$
    using ‹$x \in A$› *assms* **by** *blast*
  finally show $(GCD\ x{\in}A.\ f\ x)\ dvd\ \ldots$ **.**
qed

lemma *Lcm-mono*:
  assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x\ dvd\ g\ x$
  shows $(LCM\ x{\in}A.\ f\ x)\ dvd\ (LCM\ x{\in}A.\ g\ x)$
proof (*intro Lcm-least*, *safe*)
  fix x **assume** $x \in A$
  hence $f\ x\ dvd\ g\ x$ **by** (*rule assms*)
  also have $g\ x\ dvd\ (LCM\ x{\in}A.\ g\ x)$
    using ‹$x \in A$› **by** (*intro dvd-Lcm*) *auto*
  finally show $f\ x\ dvd\ \ldots$ **.**
qed

end

87.3 An aside: GCD and LCM on finite sets for incomplete gcd rings

context *semiring-gcd*
begin

sublocale *Gcd-fin*: *bounded-quasi-semilattice-set gcd 0 1 normalize*
defines
Gcd-fin (‹Gcd_{fin}›) = *Gcd-fin.F* :: *'a set* ⇒ *'a* **..**

abbreviation *gcd-list* :: *'a list* ⇒ *'a*
where *gcd-list xs* ≡ Gcd_{fin} (*set xs*)

sublocale *Lcm-fin*: *bounded-quasi-semilattice-set lcm 1 0 normalize*
defines
Lcm-fin (‹Lcm_{fin}›) = *Lcm-fin.F* **..**

abbreviation *lcm-list* :: *'a list* ⇒ *'a*
where *lcm-list xs* ≡ Lcm_{fin} (*set xs*)

lemma *Gcd-fin-dvd*:
a ∈ *A* ⟹ Gcd_{fin} *A dvd a*
by (*induct A rule*: *infinite-finite-induct*)
(*auto intro*: *dvd-trans*)

lemma *dvd-Lcm-fin*:
a ∈ *A* ⟹ *a dvd* Lcm_{fin} *A*
by (*induct A rule*: *infinite-finite-induct*)
(*auto intro*: *dvd-trans*)

lemma *Gcd-fin-greatest*:
a dvd Gcd_{fin} *A* **if** *finite A* **and** ⋀*b*. *b* ∈ *A* ⟹ *a dvd b*
using *that* **by** (*induct A*) *simp-all*

lemma *Lcm-fin-least*:
Lcm_{fin} *A dvd a* **if** *finite A* **and** ⋀*b*. *b* ∈ *A* ⟹ *b dvd a*
using *that* **by** (*induct A*) *simp-all*

lemma *gcd-list-greatest*:
a dvd gcd-list bs **if** ⋀*b*. *b* ∈ *set bs* ⟹ *a dvd b*
by (*rule Gcd-fin-greatest*) (*simp-all add*: *that*)

lemma *lcm-list-least*:
lcm-list bs dvd a **if** ⋀*b*. *b* ∈ *set bs* ⟹ *b dvd a*
by (*rule Lcm-fin-least*) (*simp-all add*: *that*)

lemma *dvd-Gcd-fin-iff*:
b dvd Gcd_{fin} *A* ⟷ (∀ *a*∈*A*. *b dvd a*) **if** *finite A*
using *that* **by** (*auto intro*: *Gcd-fin-greatest Gcd-fin-dvd dvd-trans* [*of b* Gcd_{fin} *A*])

lemma *dvd-gcd-list-iff*:
b dvd gcd-list xs $\longleftrightarrow$ ($\forall$ *a*$\in$*set xs. b dvd a*)
by (*simp add*: *dvd-Gcd-fin-iff*)

lemma *Lcm-fin-dvd-iff*:
Lcm_{fin} *A dvd b* $\longleftrightarrow$ ($\forall$ *a*$\in$*A. a dvd b*) **if** *finite A*
using *that* **by** (*auto intro*: *Lcm-fin-least dvd-Lcm-fin dvd-trans* [*of* - Lcm_{fin} *A b*])

lemma *lcm-list-dvd-iff*:
lcm-list xs dvd b $\longleftrightarrow$ ($\forall$ *a*$\in$*set xs. a dvd b*)
by (*simp add*: *Lcm-fin-dvd-iff*)

lemma *Gcd-fin-mult*:
Gcd_{fin} (*image* (*times b*) *A*) = *normalize* (*b* $*$ Gcd_{fin} *A*) **if** *finite A*
using *that* **by** *induction* (*auto simp*: *gcd-mult-left*)

lemma *Lcm-fin-mult*:
Lcm_{fin} (*image* (*times b*) *A*) = *normalize* (*b* $*$ Lcm_{fin} *A*) **if** *A* $\neq$ {}
proof (*cases b* = *0*)
case *True*
moreover from *that* **have** *times 0* ‘ *A* = {*0*}
by *auto*
ultimately show *?thesis*
by *simp*
next
case *False*
show *?thesis* **proof** (*cases finite A*)
case *False*
moreover have *inj-on* (*times b*) *A*
using ‹*b* $\neq$ *0*› **by** (*rule inj-on-mult*)
ultimately have *infinite* (*times b* ‘ *A*)
by (*simp add*: *finite-image-iff*)
with *False* **show** *?thesis*
by *simp*
next
case *True*
then show *?thesis* **using** *that*
by (*induct A rule*: *finite-ne-induct*) (*auto simp*: *lcm-mult-left*)
qed
qed

lemma *unit-factor-Gcd-fin*:
unit-factor (Gcd_{fin} *A*) = *of-bool* (Gcd_{fin} *A* $\neq$ *0*)
by (*rule normalize-idem-imp-unit-factor-eq*) *simp*

lemma *unit-factor-Lcm-fin*:
unit-factor (Lcm_{fin} *A*) = *of-bool* (Lcm_{fin} *A* $\neq$ *0*)

by (*rule normalize-idem-imp-unit-factor-eq*) *simp*

lemma *is-unit-Gcd-fin-iff* [*simp*]:
is-unit (Gcd_{fin} *A*) ⟷ Gcd_{fin} *A* = *1*
by (*rule normalize-idem-imp-is-unit-iff*) *simp*

lemma *is-unit-Lcm-fin-iff* [*simp*]:
is-unit (Lcm_{fin} *A*) ⟷ Lcm_{fin} *A* = *1*
by (*rule normalize-idem-imp-is-unit-iff*) *simp*

lemma *Gcd-fin-0-iff*:
Gcd_{fin} *A* = *0* ⟷ *A* ⊆ {*0*} ∧ *finite A*
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *Lcm-fin-0-iff*:
Lcm_{fin} *A* = *0* ⟷ *0* ∈ *A* **if** *finite A*
using *that* **by** (*induct A*) (*auto simp*: *lcm-eq-0-iff*)

lemma *Lcm-fin-1-iff*:
Lcm_{fin} *A* = *1* ⟷ (∀ *a*∈*A*. *is-unit a*) ∧ *finite A*
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

end

context *semiring-Gcd*
begin

lemma *Gcd-fin-eq-Gcd* [*simp*]:
Gcd_{fin} *A* = *Gcd A* **if** *finite A* **for** *A* :: *'a set*
using *that* **by** *induct simp-all*

lemma *Gcd-set-eq-fold*:
Gcd (*set xs*) = *fold gcd xs 0*
by (*simp flip*: *Gcd-fin.set-eq-fold*)

lemma [*code*]:
Gcd (*set xs*) = Gcd_{fin} (*set xs*)
by *simp*

lemma *Lcm-fin-eq-Lcm* [*simp*]:
Lcm_{fin} *A* = *Lcm A* **if** *finite A* **for** *A* :: *'a set*
using *that* **by** *induct simp-all*

lemma *Lcm-set-eq-fold*:
Lcm (*set xs*) = *fold lcm xs 1*
by (*simp flip*: *Lcm-fin.set-eq-fold*)

lemma [*code*]:
Lcm (*set xs*) = Lcm_{fin} (*set xs*)

by *simp*

end

87.4 Coprimality

context *semiring-gcd*
begin

lemma *coprime-imp-gcd-eq-1* [*simp*]:
gcd a b = 1 **if** *coprime a b*
proof −
define *t r s* **where** *t = gcd a b* **and** *r = a div t* **and** *s = b div t*
then have *a = t * r* **and** *b = t * s*
by *simp-all*
with *that* **have** *coprime (t * r) (t * s)*
by *simp*
then show *?thesis*
by (*simp add*: *t-def*)
qed

lemma *gcd-eq-1-imp-coprime* [*dest*!]:
coprime a b **if** *gcd a b = 1*
proof (*rule coprimeI*)
fix *c*
assume *c dvd a* **and** *c dvd b*
then have *c dvd gcd a b*
by (*rule gcd-greatest*)
with *that* **show** *is-unit c*
by *simp*
qed

lemma *coprime-iff-gcd-eq-1* [*presburger*, *code*]:
coprime a b ⟷ *gcd a b = 1*
by *rule* (*simp-all add*: *gcd-eq-1-imp-coprime*)

lemma *is-unit-gcd* [*simp*]:
is-unit (gcd a b) ⟷ *coprime a b*
by (*simp add*: *coprime-iff-gcd-eq-1*)

lemma *coprime-add-one-left* [*simp*]: *coprime (a + 1) a*
by (*simp add*: *gcd-eq-1-imp-coprime ac-simps*)

lemma *coprime-add-one-right* [*simp*]: *coprime a (a + 1)*
using *coprime-add-one-left* [*of a*] **by** (*simp add*: *ac-simps*)

lemma *coprime-mult-left-iff* [*simp*]:
*coprime (a * b) c* ⟷ *coprime a c* ∧ *coprime b c*
proof

```
  assume coprime (a * b) c
  with coprime-common-divisor [of a * b c]
  have *: is-unit d if d dvd a * b and d dvd c for d
    using that by blast
  have coprime a c
    by (rule coprimeI, rule *) simp-all
  moreover have coprime b c
    by (rule coprimeI, rule *) simp-all
  ultimately show coprime a c ∧ coprime b c ..
next
  assume coprime a c ∧ coprime b c
  then have coprime a c coprime b c
    by simp-all
  show coprime (a * b) c
  proof (rule coprimeI)
    fix d
    assume d dvd a * b
    then obtain r s where d: d = r * s r dvd a s dvd b
      by (rule dvd-productE)
    assume d dvd c
    with d have r * s dvd c
      by simp
    then have r dvd c s dvd c
      by (auto intro: dvd-mult-left dvd-mult-right)
    from ‹coprime a c› ‹r dvd a› ‹r dvd c›
    have is-unit r
      by (rule coprime-common-divisor)
    moreover from ‹coprime b c› ‹s dvd b› ‹s dvd c›
    have is-unit s
      by (rule coprime-common-divisor)
    ultimately show is-unit d
      by (simp add: d is-unit-mult-iff)
  qed
qed

lemma coprime-mult-right-iff [simp]:
  coprime c (a * b) ⟷ coprime c a ∧ coprime c b
  using coprime-mult-left-iff [of a b c] by (simp add: ac-simps)

lemma coprime-power-left-iff [simp]:
  coprime (a ^ n) b ⟷ coprime a b ∨ n = 0
proof (cases n = 0)
  case True
  then show ?thesis
    by simp
next
  case False
  then have n > 0
    by simp
```

then show *?thesis*
by (*induction n rule*: *nat-induct-non-zero*) *simp-all*
qed

lemma *coprime-power-right-iff* [*simp*]:
coprime a (*b* ^ *n*) ⟷ *coprime a b* ∨ *n* = *0*
using *coprime-power-left-iff* [*of b n a*] **by** (*simp add*: *ac-simps*)

lemma *prod-coprime-left*:
coprime (∏ *i*∈*A*. *f i*) *a* **if** ⋀*i*. *i* ∈ *A* ⟹ *coprime* (*f i*) *a*
using *that* **by** (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *prod-coprime-right*:
coprime a (∏ *i*∈*A*. *f i*) **if** ⋀*i*. *i* ∈ *A* ⟹ *coprime a* (*f i*)
using *that prod-coprime-left* [*of A f a*] **by** (*simp add*: *ac-simps*)

lemma *prod-list-coprime-left*:
coprime (*prod-list xs*) *a* **if** ⋀*x*. *x* ∈ *set xs* ⟹ *coprime x a*
using *that* **by** (*induct xs*) *simp-all*

lemma *prod-list-coprime-right*:
coprime a (*prod-list xs*) **if** ⋀*x*. *x* ∈ *set xs* ⟹ *coprime a x*
using *that prod-list-coprime-left* [*of xs a*] **by** (*simp add*: *ac-simps*)

lemma *coprime-dvd-mult-left-iff*:
a dvd b ∗ *c* ⟷ *a dvd b* **if** *coprime a c*
proof
assume *a dvd b*
then show *a dvd b* ∗ *c*
by *simp*
next
assume *a dvd b* ∗ *c*
show *a dvd b*
proof (*cases b* = *0*)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then have *unit*: *is-unit* (*unit-factor b*)
by *simp*
from ‹*coprime a c*›
have *gcd* (*b* ∗ *a*) (*b* ∗ *c*) ∗ *unit-factor b* = *b*
by (*subst gcd-mult-left*) (*simp add*: *ac-simps*)
moreover from ‹*a dvd b* ∗ *c*›
have *a dvd gcd* (*b* ∗ *a*) (*b* ∗ *c*) ∗ *unit-factor b*
by (*simp add*: *dvd-mult-unit-iff unit*)
ultimately show *?thesis*
by *simp*

```
  qed
qed

lemma coprime-dvd-mult-right-iff:
  a dvd c * b ⟷ a dvd b if coprime a c
  using that coprime-dvd-mult-left-iff [of a c b] by (simp add: ac-simps)

lemma divides-mult:
  a * b dvd c if a dvd c and b dvd c and coprime a b
proof −
  from ‹b dvd c› obtain b' where c = b * b' ..
  with ‹a dvd c› have a dvd b' * b
    by (simp add: ac-simps)
  with ‹coprime a b› have a dvd b'
    by (simp add: coprime-dvd-mult-left-iff)
  then obtain a' where b' = a * a' ..
  with ‹c = b * b'› have c = (a * b) * a'
    by (simp add: ac-simps)
  then show ?thesis ..
qed

lemma div-gcd-coprime:
  assumes a ≠ 0 ∨ b ≠ 0
  shows coprime (a div gcd a b) (b div gcd a b)
proof −
  let ?g = gcd a b
  let ?a' = a div ?g
  let ?b' = b div ?g
  let ?g' = gcd ?a' ?b'
  have dvdg: ?g dvd a ?g dvd b
    by simp-all
  have dvdg': ?g' dvd ?a' ?g' dvd ?b'
    by simp-all
  from dvdg dvdg' obtain ka kb ka' kb' where
    kab: a = ?g * ka b = ?g * kb ?a' = ?g' * ka' ?b' = ?g' * kb'
    unfolding dvd-def by blast
  from this [symmetric] have ?g * ?a' = (?g * ?g') * ka' ?g * ?b' = (?g * ?g') *
kb'
    by (simp-all add: mult.assoc mult.left-commute [of gcd a b])
  then have dvdgg':?g * ?g' dvd a ?g* ?g' dvd b
   by (auto simp: dvd-mult-div-cancel [OF dvdg(1)] dvd-mult-div-cancel [OF dvdg(2)]
dvd-def)
  have ?g ≠ 0
    using assms by simp
  moreover from gcd-greatest [OF dvdgg'] have ?g * ?g' dvd ?g .
  ultimately show ?thesis
    using dvd-times-left-cancel-iff [of gcd a b - 1]
    by simp (simp only: coprime-iff-gcd-eq-1)
qed
```

lemma *gcd-coprime*:
assumes *c*: *gcd a b* ≠ *0*
and *a*: *a* = *a′* ∗ *gcd a b*
and *b*: *b* = *b′* ∗ *gcd a b*
shows *coprime a′ b′*
proof −
from *c* **have** *a* ≠ *0* ∨ *b* ≠ *0*
by *simp*
with *div-gcd-coprime* **have** *coprime* (*a div gcd a b*) (*b div gcd a b*) **.**
also from *assms* **have** *a div gcd a b* = *a′*
using *dvd-div-eq-mult gcd-dvd1* **by** *blast*
also from *assms* **have** *b div gcd a b* = *b′*
using *dvd-div-eq-mult gcd-dvd1* **by** *blast*
finally show *?thesis* **.**
qed

lemma *gcd-coprime-exists*:
assumes *gcd a b* ≠ *0*
shows ∃ *a′ b′*. *a* = *a′* ∗ *gcd a b* ∧ *b* = *b′* ∗ *gcd a b* ∧ *coprime a′ b′*
proof −
have *coprime* (*a div gcd a b*) (*b div gcd a b*)
using *assms div-gcd-coprime* **by** *auto*
then show *?thesis*
by *force*
qed

lemma *pow-divides-pow-iff* [*simp*]:
a ^ *n dvd b* ^ *n* ⟷ *a dvd b* **if** *n* > *0*
proof (*cases gcd a b* = *0*)
case *True*
then show *?thesis*
by *simp*
next
case *False*
show *?thesis*
proof
let *?d* = *gcd a b*
from ‹*n* > *0*› **obtain** *m* **where** *m*: *n* = *Suc m*
by (*cases n*) *simp-all*
from *False* **have** *zn*: *?d* ^ *n* ≠ *0*
by (*rule power-not-zero*)
from *gcd-coprime-exists* [*OF False*]
obtain *a′ b′* **where** *ab′*: *a* = *a′* ∗ *?d b* = *b′* ∗ *?d coprime a′ b′*
by *blast*
assume *a* ^ *n dvd b* ^ *n*
then have (*a′* ∗ *?d*) ^ *n dvd* (*b′* ∗ *?d*) ^ *n*
by (*simp add*: *ab′*(*1*,*2*)[*symmetric*])
then have *?d*^*n* ∗ *a′*^*n dvd ?d*^*n* ∗ *b′*^*n*

```
    by (simp only: power-mult-distrib ac-simps)
  with zn have a' ^ n dvd b' ^ n
    by simp
  then have a' dvd b' ^ n
    using dvd-trans[of a' a'^n b'^n] by (simp add: m)
  then have a' dvd b' ^ m * b'
    by (simp add: m ac-simps)
  moreover have coprime a' (b' ^ n)
    using ‹coprime a' b'› by simp
  then have a' dvd b'
    using ‹a' dvd b' ^ n› coprime-dvd-mult-left-iff dvd-mult by blast
  then have a' * ?d dvd b' * ?d
    by (rule mult-dvd-mono) simp
  with ab'(1,2) show a dvd b
    by simp
 next
  assume a dvd b
  with ‹n > 0› show a ^ n dvd b ^ n
    by (induction rule: nat-induct-non-zero)
      (simp-all add: mult-dvd-mono)
 qed
qed

lemma coprime-crossproduct:
 fixes a b c d :: 'a
 assumes coprime a d and coprime b c
 shows normalize a * normalize c = normalize b * normalize d ⟷
   normalize a = normalize b ∧ normalize c = normalize d
   (is ?lhs ⟷ ?rhs)
proof
 assume ?rhs
 then show ?lhs by simp
next
 assume ?lhs
 from ‹?lhs› have normalize a dvd normalize b * normalize d
   by (auto intro: dvdI dest: sym)
 with ‹coprime a d› have a dvd b
   by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
 from ‹?lhs› have normalize b dvd normalize a * normalize c
   by (auto intro: dvdI dest: sym)
 with ‹coprime b c› have b dvd a
   by (simp add: coprime-dvd-mult-left-iff normalize-mult [symmetric])
 from ‹?lhs› have normalize c dvd normalize d * normalize b
   by (auto intro: dvdI dest: sym simp add: mult.commute)
 with ‹coprime b c› have c dvd d
  by (simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric])
 from ‹?lhs› have normalize d dvd normalize c * normalize a
   by (auto intro: dvdI dest: sym simp add: mult.commute)
 with ‹coprime a d› have d dvd c
```

```
  by (simp add: coprime-dvd-mult-left-iff coprime-commute normalize-mult [symmetric])
 from ‹a dvd b› ‹b dvd a› have normalize a = normalize b
  by (rule associatedI)
 moreover from ‹c dvd d› ‹d dvd c› have normalize c = normalize d
  by (rule associatedI)
 ultimately show ?rhs ..
qed

lemma gcd-mult-left-left-cancel:
 gcd (c * a) b = gcd a b if coprime b c
proof -
 have coprime (gcd b (a * c)) c
  by (rule coprimeI) (auto intro: that coprime-common-divisor)
 then have gcd b (a * c) dvd a
  using coprime-dvd-mult-left-iff [of gcd b (a * c) c a]
  by simp
 then show ?thesis
  by (auto intro: associated-eqI simp add: ac-simps)
qed

lemma gcd-mult-left-right-cancel:
 gcd (a * c) b = gcd a b if coprime b c
 using that gcd-mult-left-left-cancel [of b c a]
 by (simp add: ac-simps)

lemma gcd-mult-right-left-cancel:
 gcd a (c * b) = gcd a b if coprime a c
 using that gcd-mult-left-left-cancel [of a c b]
 by (simp add: ac-simps)

lemma gcd-mult-right-right-cancel:
 gcd a (b * c) = gcd a b if coprime a c
 using that gcd-mult-right-left-cancel [of a c b]
 by (simp add: ac-simps)

lemma gcd-exp-weak:
 gcd (a ^ n) (b ^ n) = normalize (gcd a b ^ n)
proof (cases a = 0 ∧ b = 0 ∨ n = 0)
 case True
 then show ?thesis
  by (cases n) simp-all
next
 case False
 then have coprime (a div gcd a b) (b div gcd a b) and n > 0
  by (auto intro: div-gcd-coprime)
 then have coprime ((a div gcd a b) ^ n) ((b div gcd a b) ^ n)
  by simp
 then have 1 = gcd ((a div gcd a b) ^ n) ((b div gcd a b) ^ n)
  by simp
```

then have *normalize* (*gcd a b* ^ *n*) = *normalize* (*gcd a b* ^ *n* * ...)
by *simp*
also have ... = *gcd* (*gcd a b* ^ *n* * (*a div gcd a b*) ^ *n*) (*gcd a b* ^ *n* * (*b div gcd a b*) ^ *n*)
by (*rule gcd-mult-left* [*symmetric*])
also have (*gcd a b*) ^ *n* * (*a div gcd a b*) ^ *n* = *a* ^ *n*
by (*simp add*: *ac-simps div-power dvd-power-same*)
also have (*gcd a b*) ^ *n* * (*b div gcd a b*) ^ *n* = *b* ^ *n*
by (*simp add*: *ac-simps div-power dvd-power-same*)
finally show *?thesis* **by** *simp*
qed

lemma *division-decomp*:
assumes *a dvd b* * *c*
shows ∃ *b' c'*. *a* = *b'* * *c'* ∧ *b' dvd b* ∧ *c' dvd c*
proof (*cases gcd a b* = *0*)
case *True*
then have *a* = *0* ∧ *b* = *0*
by *simp*
then have *a* = *0* * *c* ∧ *0 dvd b* ∧ *c dvd c*
by *simp*
then show *?thesis* **by** *blast*
next
case *False*
let *?d* = *gcd a b*
from *gcd-coprime-exists* [*OF False*]
obtain *a' b'* **where** *ab'*: *a* = *a'* * *?d b* = *b'* * *?d coprime a' b'*
by *blast*
from *ab'*(*1*) **have** *a' dvd a* **..**
with *assms* **have** *a' dvd b* * *c*
using *dvd-trans* [*of a' a b* * *c*] **by** *simp*
from *assms ab'*(*1*,*2*) **have** *a'* * *?d dvd* (*b'* * *?d*) * *c*
by *simp*
then have *?d* * *a' dvd ?d* * (*b'* * *c*)
by (*simp add*: *mult-ac*)
with ‹*?d* ≠ *0*› **have** *a' dvd b'* * *c*
by *simp*
then have *a' dvd c*
using ‹*coprime a' b'*› **by** (*simp add*: *coprime-dvd-mult-right-iff*)
with *ab'*(*1*) **have** *a* = *?d* * *a'* ∧ *?d dvd b* ∧ *a' dvd c*
by (*simp add*: *ac-simps*)
then show *?thesis* **by** *blast*
qed

lemma *lcm-coprime*: *coprime a b* ⟹ *lcm a b* = *normalize* (*a* * *b*)
by (*subst lcm-gcd*) *simp*

end

context *ring-gcd*
begin

lemma *coprime-minus-left-iff* [*simp*]:
  coprime (− *a*) *b* ⟷ *coprime a b*
  by (*rule*; *rule coprimeI*) (*auto intro*: *coprime-common-divisor*)

lemma *coprime-minus-right-iff* [*simp*]:
  coprime a (− *b*) ⟷ *coprime a b*
  using *coprime-minus-left-iff* [*of b a*] **by** (*simp add*: *ac-simps*)

lemma *coprime-diff-one-left* [*simp*]: *coprime* (*a* − *1*) *a*
  using *coprime-add-one-right* [*of a* − *1*] **by** *simp*

lemma *coprime-doff-one-right* [*simp*]: *coprime a* (*a* − *1*)
  using *coprime-diff-one-left* [*of a*] **by** (*simp add*: *ac-simps*)

end

context *semiring-Gcd*
begin

lemma *Lcm-coprime*:
  assumes *finite A*
    and *A* ≠ {}
    and ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *a* ≠ *b* ⟹ *coprime a b*
  shows *Lcm A* = *normalize* (∏ *A*)
  using *assms*
proof (*induct rule*: *finite-ne-induct*)
  case *singleton*
  then show *?case* **by** *simp*
next
  case (*insert a A*)
  have *Lcm* (*insert a A*) = *lcm a* (*Lcm A*)
    by *simp*
  also from *insert* **have** *Lcm A* = *normalize* (∏ *A*)
    by *blast*
  also have *lcm a* ... = *lcm a* (∏ *A*)
    by (*cases* ∏ *A* = *0*) (*simp-all add*: *lcm-div-unit2*)
  also from *insert* **have** *coprime a* (∏ *A*)
    by (*subst coprime-commute*, *intro prod-coprime-left*) *auto*
  with *insert* **have** *lcm a* (∏ *A*) = *normalize* (∏ (*insert a A*))
    by (*simp add*: *lcm-coprime*)
  finally show *?case* **.**
qed

lemma *Lcm-coprime′*:
  card A ≠ *0* ⟹
    (⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *a* ≠ *b* ⟹ *coprime a b*) ⟹

Lcm A = normalize (∏ *A*)
by (*rule Lcm-coprime*) (*simp-all add*: *card-eq-0-iff*)

end

And some consequences: cancellation modulo *m*

lemma *mult-mod-cancel-right*:
fixes *m* :: *'a*::{*euclidean-ring-cancel*,*semiring-gcd*}
assumes *eq*: (*a* ∗ *n*) *mod m* = (*b* ∗ *n*) *mod m* **and** *coprime m n*
shows *a mod m* = *b mod m*
proof −
have *m dvd* (*a*∗*n* − *b*∗*n*)
using *eq mod-eq-dvd-iff* **by** *blast*
then have *m dvd a*−*b*
by (*metis* ‹*coprime m n*› *coprime-dvd-mult-left-iff left-diff-distrib'*)
then show *?thesis*
using *mod-eq-dvd-iff* **by** *blast*
qed

lemma *mult-mod-cancel-left*:
fixes *m* :: *'a*::{*euclidean-ring-cancel*,*semiring-gcd*}
assumes (*n* ∗ *a*) *mod m* = (*n* ∗ *b*) *mod m* **and** *coprime m n*
shows *a mod m* = *b mod m*
by (*metis assms mult.commute mult-mod-cancel-right*)

87.5 GCD and LCM for multiplicative normalisation functions

class *semiring-gcd-mult-normalize* = *semiring-gcd* + *normalization-semidom-multiplicative*
begin

lemma *mult-gcd-left*: *c* ∗ *gcd a b* = *unit-factor c* ∗ *gcd* (*c* ∗ *a*) (*c* ∗ *b*)
by (*simp add*: *gcd-mult-left normalize-mult mult.assoc* [*symmetric*])

lemma *mult-gcd-right*: *gcd a b* ∗ *c* = *gcd* (*a* ∗ *c*) (*b* ∗ *c*) ∗ *unit-factor c*
using *mult-gcd-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *gcd-mult-distrib'*: *normalize c* ∗ *gcd a b* = *gcd* (*c* ∗ *a*) (*c* ∗ *b*)
by (*subst gcd-mult-left*) (*simp-all add*: *normalize-mult*)

lemma *gcd-mult-distrib*: *k* ∗ *gcd a b* = *gcd* (*k* ∗ *a*) (*k* ∗ *b*) ∗ *unit-factor k*
proof−
have *normalize k* ∗ *gcd a b* = *gcd* (*k* ∗ *a*) (*k* ∗ *b*)
by (*simp add*: *gcd-mult-distrib'*)
then have *normalize k* ∗ *gcd a b* ∗ *unit-factor k* = *gcd* (*k* ∗ *a*) (*k* ∗ *b*) ∗ *unit-factor k*
by *simp*
then have *normalize k* ∗ *unit-factor k* ∗ *gcd a b* = *gcd* (*k* ∗ *a*) (*k* ∗ *b*) ∗ *unit-factor k*

by (*simp only*: *ac-simps*)
then show *?thesis*
by *simp*
qed

lemma *gcd-mult-lcm* [*simp*]: *gcd a b * lcm a b = normalize a * normalize b*
by (*simp add*: *lcm-gcd normalize-mult dvd-normalize-div*)

lemma *lcm-mult-gcd* [*simp*]: *lcm a b * gcd a b = normalize a * normalize b*
using *gcd-mult-lcm* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *mult-lcm-left*: *c * lcm a b = unit-factor c * lcm (c * a) (c * b)*
by (*simp add*: *lcm-mult-left mult.assoc* [*symmetric*] *normalize-mult*)

lemma *mult-lcm-right*: *lcm a b * c = lcm (a * c) (b * c) * unit-factor c*
using *mult-lcm-left* [*of c a b*] **by** (*simp add*: *ac-simps*)

lemma *lcm-gcd-prod*: *lcm a b * gcd a b = normalize (a * b)*
by (*simp add*: *lcm-gcd dvd-normalize-div normalize-mult*)

lemma *lcm-mult-distrib'*: *normalize c * lcm a b = lcm (c * a) (c * b)*
by (*subst lcm-mult-left*) (*simp add*: *normalize-mult*)

lemma *lcm-mult-distrib*: *k * lcm a b = lcm (k * a) (k * b) * unit-factor k*
proof−
have *normalize k * lcm a b = lcm (k * a) (k * b)*
by (*simp add*: *lcm-mult-distrib'*)
then have *normalize k * lcm a b * unit-factor k = lcm (k * a) (k * b) * unit-factor k*
by *simp*
then have *normalize k * unit-factor k * lcm a b = lcm (k * a) (k * b) * unit-factor k*
by (*simp only*: *ac-simps*)
then show *?thesis*
by *simp*
qed

lemma *coprime-crossproduct'*:
fixes *a b c d*
assumes *b ≠ 0*
assumes *unit-factors*: *unit-factor b = unit-factor d*
assumes *coprime*: *coprime a b coprime c d*
shows *a * d = b * c ⟷ a = c ∧ b = d*
proof *safe*
assume *eq*: *a * d = b * c*
hence *normalize a * normalize d = normalize c * normalize b*
by (*simp only*: *normalize-mult* [*symmetric*] *mult-ac*)
with *coprime* **have** *normalize b = normalize d*
by (*subst* (*asm*) *coprime-crossproduct*) *simp-all*

from *this* **and** *unit-factors* **show** $b = d$
by (*rule normalize-unit-factor-eqI*)
from *eq* **have** $a * d = c * d$ **by** (*simp only*: ‹$b = d$› *mult-ac*)
with ‹$b \neq 0$› ‹$b = d$› **show** $a = c$ **by** *simp*
qed (*simp-all add*: *mult-ac*)

lemma *gcd-exp* [*simp*]:
gcd $(a \hat{} n)$ $(b \hat{} n)$ = *gcd a b* $\hat{}$ *n*
using *gcd-exp-weak*[*of a n b*] **by** (*simp add*: *normalize-power*)

end

87.6 GCD and LCM on *nat* and *int*

instantiation *nat* :: *gcd*
begin

fun *gcd-nat* :: *nat* ⇒ *nat* ⇒ *nat*
where *gcd-nat x y* = (*if y* = *0 then x else gcd y* (*x mod y*))

definition *lcm-nat* :: *nat* ⇒ *nat* ⇒ *nat*
where *lcm-nat x y* = *x* ∗ *y div* (*gcd x y*)

instance ..

end

instantiation *int* :: *gcd*
begin

definition *gcd-int* :: *int* ⇒ *int* ⇒ *int*
where *gcd-int x y* = *int* (*gcd* (*nat* |*x*|) (*nat* |*y*|))

definition *lcm-int* :: *int* ⇒ *int* ⇒ *int*
where *lcm-int x y* = *int* (*lcm* (*nat* |*x*|) (*nat* |*y*|))

instance ..

end

lemma *gcd-int-int-eq* [*simp*]:
gcd (*int m*) (*int n*) = *int* (*gcd m n*)
by (*simp add*: *gcd-int-def*)

lemma *gcd-nat-abs-left-eq* [*simp*]:
gcd (*nat* |*k*|) *n* = *nat* (*gcd k* (*int n*))
by (*simp add*: *gcd-int-def*)

lemma *gcd-nat-abs-right-eq* [*simp*]:

gcd n (*nat* |*k*|) = *nat* (*gcd* (*int n*) *k*)
by (*simp add*: *gcd-int-def*)

lemma *abs-gcd-int* [*simp*]:
|*gcd x y*| = *gcd x y*
for *x y* :: *int*
by (*simp only*: *gcd-int-def*)

lemma *gcd-abs1-int* [*simp*]:
gcd |*x*| *y* = *gcd x y*
for *x y* :: *int*
by (*simp only*: *gcd-int-def*) *simp*

lemma *gcd-abs2-int* [*simp*]:
gcd x |*y*| = *gcd x y*
for *x y* :: *int*
by (*simp only*: *gcd-int-def*) *simp*

lemma *lcm-int-int-eq* [*simp*]:
lcm (*int m*) (*int n*) = *int* (*lcm m n*)
by (*simp add*: *lcm-int-def*)

lemma *lcm-nat-abs-left-eq* [*simp*]:
lcm (*nat* |*k*|) *n* = *nat* (*lcm k* (*int n*))
by (*simp add*: *lcm-int-def*)

lemma *lcm-nat-abs-right-eq* [*simp*]:
lcm n (*nat* |*k*|) = *nat* (*lcm* (*int n*) *k*)
by (*simp add*: *lcm-int-def*)

lemma *lcm-abs1-int* [*simp*]:
lcm |*x*| *y* = *lcm x y*
for *x y* :: *int*
by (*simp only*: *lcm-int-def*) *simp*

lemma *lcm-abs2-int* [*simp*]:
lcm x |*y*| = *lcm x y*
for *x y* :: *int*
by (*simp only*: *lcm-int-def*) *simp*

lemma *abs-lcm-int* [*simp*]: |*lcm i j*| = *lcm i j*
for *i j* :: *int*
by (*simp only*: *lcm-int-def*)

lemma *gcd-nat-induct* [*case-names base step*]:
fixes *m n* :: *nat*
assumes ⋀*m*. *P m 0*
and ⋀*m n*. *0* < *n* ⟹ *P n* (*m mod n*) ⟹ *P m n*
shows *P m n*

proof (*induction m n rule*: *gcd-nat.induct*)
  case (*1 x y*)
  then show *?case*
    using *assms neq0-conv* **by** *blast*
qed

lemma *gcd-neg1-int* [*simp*]: *gcd* (− *x*) *y* = *gcd x y*
  for *x y* :: *int*
  by (*simp only*: *gcd-int-def*) *simp*

lemma *gcd-neg2-int* [*simp*]: *gcd x* (− *y*) = *gcd x y*
  for *x y* :: *int*
  by (*simp only*: *gcd-int-def*) *simp*

lemma *gcd-cases-int*:
  fixes *x y* :: *int*
  assumes *x* ≥ *0* ⟹ *y* ≥ *0* ⟹ *P* (*gcd x y*)
    and *x* ≥ *0* ⟹ *y* ≤ *0* ⟹ *P* (*gcd x* (− *y*))
    and *x* ≤ *0* ⟹ *y* ≥ *0* ⟹ *P* (*gcd* (− *x*) *y*)
    and *x* ≤ *0* ⟹ *y* ≤ *0* ⟹ *P* (*gcd* (− *x*) (− *y*))
  shows *P* (*gcd x y*)
  using *assms* **by** *auto arith*

lemma *gcd-ge-0-int* [*simp*]: *gcd* (*x*::*int*) *y* >= *0*
  for *x y* :: *int*
  by (*simp add*: *gcd-int-def*)

lemma *lcm-neg1-int*: *lcm* (− *x*) *y* = *lcm x y*
  for *x y* :: *int*
  by (*simp only*: *lcm-int-def*) *simp*

lemma *lcm-neg2-int*: *lcm x* (− *y*) = *lcm x y*
  for *x y* :: *int*
  by (*simp only*: *lcm-int-def*) *simp*

lemma *lcm-cases-int*:
  fixes *x y* :: *int*
  assumes *x* ≥ *0* ⟹ *y* ≥ *0* ⟹ *P* (*lcm x y*)
    and *x* ≥ *0* ⟹ *y* ≤ *0* ⟹ *P* (*lcm x* (− *y*))
    and *x* ≤ *0* ⟹ *y* ≥ *0* ⟹ *P* (*lcm* (− *x*) *y*)
    and *x* ≤ *0* ⟹ *y* ≤ *0* ⟹ *P* (*lcm* (− *x*) (− *y*))
  shows *P* (*lcm x y*)
  using *assms* **by** (*auto simp*: *lcm-neg1-int lcm-neg2-int*) *arith*

lemma *lcm-ge-0-int* [*simp*]: *lcm x y* ≥ *0*
  for *x y* :: *int*
  by (*simp only*: *lcm-int-def*)

lemma *gcd-0-nat*: *gcd x 0* = *x*

for x :: *nat*
by *simp*

lemma *gcd-0-int* [*simp*]: $gcd\ x\ 0 = |x|$
for x :: *int*
by (*auto simp*: *gcd-int-def*)

lemma *gcd-0-left-nat*: $gcd\ 0\ x = x$
for x :: *nat*
by *simp*

lemma *gcd-0-left-int* [*simp*]: $gcd\ 0\ x = |x|$
for x :: *int*
by (*auto simp*: *gcd-int-def*)

lemma *gcd-red-nat*: $gcd\ x\ y = gcd\ y\ (x\ mod\ y)$
for $x\ y$ :: *nat*
by (*cases* $y = 0$) *auto*

Weaker, but useful for the simplifier.

lemma *gcd-non-0-nat*: $y \neq 0 \Longrightarrow gcd\ x\ y = gcd\ y\ (x\ mod\ y)$
for $x\ y$ :: *nat*
by *simp*

lemma *gcd-1-nat* [*simp*]: $gcd\ m\ 1 = 1$
for m :: *nat*
by *simp*

lemma *gcd-Suc-0* [*simp*]: $gcd\ m\ (Suc\ 0) = Suc\ 0$
for m :: *nat*
by *simp*

lemma *gcd-1-int* [*simp*]: $gcd\ m\ 1 = 1$
for m :: *int*
by (*simp add*: *gcd-int-def*)

lemma *gcd-idem-nat*: $gcd\ x\ x = x$
for x :: *nat*
by *simp*

lemma *gcd-idem-int*: $gcd\ x\ x = |x|$
for x :: *int*
by (*auto simp*: *gcd-int-def*)

declare *gcd-nat.simps* [*simp del*]

gcd m n divides m and n. The conjunctions don't seem provable separately.

instance *nat* :: *semiring-gcd*
proof

```
  fix m n :: nat
  show gcd m n dvd m and gcd m n dvd n
  proof (induct m n rule: gcd-nat-induct)
    case (step m n)
    then have gcd n (m mod n) dvd m
      by (metis dvd-mod-imp-dvd)
    with step show gcd m n dvd m
      by (simp add: gcd-non-0-nat)
  qed (simp-all add: gcd-0-nat gcd-non-0-nat)
next
  fix m n k :: nat
  assume k dvd m and k dvd n
  then show k dvd gcd m n
    by (induct m n rule: gcd-nat-induct) (simp-all add: gcd-non-0-nat dvd-mod
gcd-0-nat)
qed (simp-all add: lcm-nat-def)

instance int :: ring-gcd
proof
  fix k l r :: int
  show [simp]: gcd k l dvd k gcd k l dvd l
    using gcd-dvd1 [of nat |k| nat |l|]
      gcd-dvd2 [of nat |k| nat |l|]
    by simp-all
  show lcm k l = normalize (k * l div gcd k l)
    using lcm-gcd [of nat |k| nat |l|]
    by (simp add: nat-eq-iff of-nat-div abs-mult abs-div)
  assume r dvd k r dvd l
  then show r dvd gcd k l
    using gcd-greatest [of nat |r| nat |k| nat |l|]
    by simp
qed simp

lemma gcd-le1-nat [simp]: a ≠ 0 ⟹ gcd a b ≤ a
  for a b :: nat
  by (rule dvd-imp-le) auto

lemma gcd-le2-nat [simp]: b ≠ 0 ⟹ gcd a b ≤ b
  for a b :: nat
  by (rule dvd-imp-le) auto

lemma gcd-le1-int [simp]: a > 0 ⟹ gcd a b ≤ a
  for a b :: int
  by (rule zdvd-imp-le) auto

lemma gcd-le2-int [simp]: b > 0 ⟹ gcd a b ≤ b
  for a b :: int
  by (rule zdvd-imp-le) auto
```

lemma *gcd-pos-nat* [*simp*]: *gcd m n > 0* ⟷ *m ≠ 0 ∨ n ≠ 0*
for *m n* :: *nat*
using *gcd-eq-0-iff* [*of m n*] **by** *arith*

lemma *gcd-pos-int* [*simp*]: *gcd m n > 0* ⟷ *m ≠ 0 ∨ n ≠ 0*
for *m n* :: *int*
using *gcd-eq-0-iff* [*of m n*] *gcd-ge-0-int* [*of m n*] **by** *arith*

lemma *gcd-unique-nat*: *d dvd a ∧ d dvd b ∧* (∀ *e*. *e dvd a ∧ e dvd b* ⟶ *e dvd d*) ⟷ *d = gcd a b*
for *d a* :: *nat*
using *gcd-unique* **by** *fastforce*

lemma *gcd-unique-int*:
d ≥ 0 ∧ d dvd a ∧ d dvd b ∧ (∀ *e*. *e dvd a ∧ e dvd b* ⟶ *e dvd d*) ⟷ *d = gcd a b*
for *d a* :: *int*
using *zdvd-antisym-nonneg* **by** *auto*

interpretation *gcd-nat*:
semilattice-neutr-order gcd 0::*nat Rings.dvd* λ*m n*. *m dvd n ∧ m ≠ n*
by *standard* (*auto simp*: *gcd-unique-nat* [*symmetric*] *intro*: *dvd-antisym dvd-trans*)

lemma *gcd-proj1-if-dvd-int* [*simp*]: *x dvd y* ⟹ *gcd x y = |x|*
for *x y* :: *int*
by (*metis abs-dvd-iff gcd-0-left-int gcd-unique-int*)

lemma *gcd-proj2-if-dvd-int* [*simp*]: *y dvd x* ⟹ *gcd x y = |y|*
for *x y* :: *int*
by (*metis gcd-proj1-if-dvd-int gcd.commute*)

Multiplication laws.

lemma *gcd-mult-distrib-nat*: *k ∗ gcd m n = gcd* (*k ∗ m*) (*k ∗ n*)
for *k m n* :: *nat*
— [1, page 27]
by (*simp add*: *gcd-mult-left*)

lemma *gcd-mult-distrib-int*: *|k| ∗ gcd m n = gcd* (*k ∗ m*) (*k ∗ n*)
for *k m n* :: *int*
by (*simp add*: *gcd-mult-left abs-mult*)

Addition laws.

lemma *gcd-diff1-nat*: *m ≥ n* ⟹ *gcd* (*m − n*) *n = gcd m n*
for *m n* :: *nat*
by (*subst gcd-add1* [*symmetric*]) *auto*

lemma *gcd-diff2-nat*: *n ≥ m* ⟹ *gcd* (*n − m*) *n = gcd m n*
for *m n* :: *nat*

by (*metis gcd.commute gcd-add2 gcd-diff1-nat le-add-diff-inverse2*)

lemma *gcd-non-0-int*:
fixes *x y* :: *int*
assumes *y* > *0* **shows** *gcd x y* = *gcd y* (*x mod y*)
proof (*cases x mod y* = *0*)
case *False*
then have *neg*: *x mod y* = *y* − (− *x*) *mod y*
by (*simp add*: *zmod-zminus1-eq-if*)
have *xy*: *0* ≤ *x mod y*
by (*simp add*: *assms*)
show *?thesis*
proof (*cases x* < *0*)
case *True*
have *nat* (− *x mod y*) ≤ *nat y*
by (*simp add*: *assms dual-order.order-iff-strict*)
moreover have *gcd* (*nat* (− *x*)) (*nat y*) = *gcd* (*nat* (− *x mod y*)) (*nat y*)
using *True assms gcd-non-0-nat nat-mod-distrib* **by** *auto*
ultimately have *gcd* (*nat* (− *x*)) (*nat y*) = *gcd* (*nat y*) (*nat* (*x mod y*))
using *assms*
by (*simp add*: *neg nat-diff-distrib′*) (*metis gcd.commute gcd-diff2-nat*)
with *assms* ‹*0* ≤ *x mod y*› **show** *?thesis*
by (*simp add*: *True dual-order.order-iff-strict gcd-int-def*)
next
case *False*
with *assms xy* **have** *gcd* (*nat x*) (*nat y*) = *gcd* (*nat y*) (*nat x mod nat y*)
using *gcd-red-nat* **by** *blast*
with *False assms* **show** *?thesis*
by (*simp add*: *gcd-int-def nat-mod-distrib*)
qed
qed (*use assms* **in** *auto*)

lemma *gcd-red-int*: *gcd x y* = *gcd y* (*x mod y*)
for *x y* :: *int*
proof (*cases y 0*::*int rule*: *linorder-cases*)
case *less*
with *gcd-non-0-int* [*of* − *y* − *x*] **show** *?thesis*
by *auto*
next
case *greater*
with *gcd-non-0-int* [*of y x*] **show** *?thesis*
by *auto*
qed *auto*

lemma *finite-divisors-nat* [*simp*]:

fixes $m :: nat$
assumes $m > 0$
shows *finite* $\{d.\ d\ dvd\ m\}$
proof−
from *assms* **have** $\{d.\ d\ dvd\ m\} \subseteq \{d.\ d \leq m\}$
by (*auto dest*: *dvd-imp-le*)
then show *?thesis*
using *finite-Collect-le-nat* **by** (*rule finite-subset*)
qed

lemma *finite-divisors-int* [*simp*]:
fixes $i :: int$
assumes $i \neq 0$
shows *finite* $\{d.\ d\ dvd\ i\}$
proof −
have $\{d.\ |d| \leq |i|\} = \{-|i|..|i|\}$
by (*auto simp*: *abs-if*)
then have *finite* $\{d.\ |d| \leq |i|\}$
by *simp*
from *finite-subset* [*OF - this*] **show** *?thesis*
using *assms* **by** (*simp add*: *dvd-imp-le-int subset-iff*)
qed

lemma *Max-divisors-self-nat* [*simp*]: $n \neq 0 \Longrightarrow Max\ \{d{::}nat.\ d\ dvd\ n\} = n$
by (*fastforce intro*: *antisym Max-le-iff*[*THEN iffD2*] *simp*: *dvd-imp-le*)

lemma *Max-divisors-self-int* [*simp*]:
assumes $n \neq 0$ **shows** $Max\ \{d{::}int.\ d\ dvd\ n\} = |n|$
proof (*rule antisym*)
show $Max\ \{d.\ d\ dvd\ n\} \leq |n|$
using *assms* **by** (*auto intro*: *abs-le-D1 dvd-imp-le-int intro*!: *Max-le-iff* [*THEN iffD2*])
qed (*simp add*: *assms*)

lemma *gcd-is-Max-divisors-nat*:
fixes $m\ n :: nat$
assumes $n > 0$ **shows** $gcd\ m\ n = Max\ \{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$
proof (*rule Max-eqI*[*THEN sym*], *simp-all*)
show *finite* $\{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$
by (*simp add*: ‹$n > 0$›)
show $\bigwedge y.\ y\ dvd\ m \wedge y\ dvd\ n \Longrightarrow y \leq gcd\ m\ n$
by (*simp add*: ‹$n > 0$› *dvd-imp-le*)
qed

lemma *gcd-is-Max-divisors-int*:
fixes $m\ n :: int$
assumes $n \neq 0$ **shows** $gcd\ m\ n = Max\ \{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$
proof (*rule Max-eqI*[*THEN sym*], *simp-all*)
show *finite* $\{d.\ d\ dvd\ m \wedge d\ dvd\ n\}$

by (*simp add*: ‹$n \neq 0$›)
show $\bigwedge y.\ y\ dvd\ m \wedge y\ dvd\ n \Longrightarrow y \leq gcd\ m\ n$
by (*simp add*: ‹$n \neq 0$› *zdvd-imp-le*)
qed

lemma *gcd-code-int* [*code*]: $gcd\ k\ l = |if\ l = 0\ then\ k\ else\ gcd\ l\ (|k|\ mod\ |l|)|$
for $k\ l :: int$
using *gcd-red-int* [*of* $|k|\ |l|$] **by** *simp*

lemma *coprime-Suc-left-nat* [*simp*]:
$coprime\ (Suc\ n)\ n$
using *coprime-add-one-left* [*of n*] **by** *simp*

lemma *coprime-Suc-right-nat* [*simp*]:
$coprime\ n\ (Suc\ n)$
using *coprime-Suc-left-nat* [*of n*] **by** (*simp add*: *ac-simps*)

lemma *coprime-diff-one-left-nat* [*simp*]:
$coprime\ (n - 1)\ n$ **if** $n > 0$ **for** $n :: nat$
using *that coprime-Suc-right-nat* [*of* $n - 1$] **by** *simp*

lemma *coprime-diff-one-right-nat* [*simp*]:
$coprime\ n\ (n - 1)$ **if** $n > 0$ **for** $n :: nat$
using *that coprime-diff-one-left-nat* [*of n*] **by** (*simp add*: *ac-simps*)

lemma *coprime-crossproduct-nat*:
fixes $a\ b\ c\ d :: nat$
assumes $coprime\ a\ d$ **and** $coprime\ b\ c$
shows $a * c = b * d \longleftrightarrow a = b \wedge c = d$
using *assms coprime-crossproduct* [*of a d b c*] **by** *simp*

lemma *coprime-crossproduct-int*:
fixes $a\ b\ c\ d :: int$
assumes $coprime\ a\ d$ **and** $coprime\ b\ c$
shows $|a| * |c| = |b| * |d| \longleftrightarrow |a| = |b| \wedge |c| = |d|$
using *assms coprime-crossproduct* [*of a d b c*] **by** *simp*

87.7 Bezout's theorem

Function *bezw* returns a pair of witnesses to Bezout's theorem – see the theorems that follow the definition.

fun $bezw :: nat \Rightarrow nat \Rightarrow int * int$
where $bezw\ x\ y =$
$(if\ y = 0\ then\ (1,\ 0)$
$else$
$(snd\ (bezw\ y\ (x\ mod\ y)),$
$fst\ (bezw\ y\ (x\ mod\ y)) - snd\ (bezw\ y\ (x\ mod\ y)) * int(x\ div\ y)))$

lemma *bezw-0* [*simp*]: $bezw\ x\ 0 = (1,\ 0)$

by *simp*

lemma *bezw-non-0*:
y > *0* ⟹ *bezw x y* =
(*snd* (*bezw y* (*x mod y*)), *fst* (*bezw y* (*x mod y*)) − *snd* (*bezw y* (*x mod y*)) ∗ *int*(*x div y*))
by *simp*

declare *bezw.simps* [*simp del*]

lemma *bezw-aux*: *int* (*gcd x y*) = *fst* (*bezw x y*) ∗ *int x* + *snd* (*bezw x y*) ∗ *int y*
proof (*induct x y rule*: *gcd-nat-induct*)
case (*step m n*)
then have *fst* (*bezw m n*) ∗ *int m* + *snd* (*bezw m n*) ∗ *int n* − *int* (*gcd m n*)
= *int m* ∗ *snd* (*bezw n* (*m mod n*)) −
(*int* (*m mod n*) ∗ *snd* (*bezw n* (*m mod n*)) + *int n* ∗ (*int* (*m div n*) ∗ *snd* (*bezw n* (*m mod n*))))
by (*simp add*: *bezw-non-0 gcd-non-0-nat field-simps*)
also have ... = *int m* ∗ *snd* (*bezw n* (*m mod n*)) − (*int* (*m mod n*) + *int* (*n* ∗ (*m div n*))) ∗ *snd* (*bezw n* (*m mod n*))
by (*simp add*: *distrib-right*)
also have ... = *0*
by (*metis cancel-comm-monoid-add-class.diff-cancel mod-mult-div-eq of-nat-add*)
finally show *?case*
by *simp*
qed *auto*

lemma *bezout-int*: ∃ *u v*. *u* ∗ *x* + *v* ∗ *y* = *gcd x y*
for *x y* :: *int*
proof −
have *aux*: *x* ≥ *0* ⟹ *y* ≥ *0* ⟹ ∃ *u v*. *u* ∗ *x* + *v* ∗ *y* = *gcd x y* **for** *x y* :: *int*
apply (*rule-tac x* = *fst* (*bezw* (*nat x*) (*nat y*)) **in** *exI*)
apply (*rule-tac x* = *snd* (*bezw* (*nat x*) (*nat y*)) **in** *exI*)
by (*simp add*: *bezw-aux gcd-int-def*)
consider *x* ≥ *0 y* ≥ *0* | *x* ≥ *0 y* ≤ *0* | *x* ≤ *0 y* ≥ *0* | *x* ≤ *0 y* ≤ *0*
using *linear* **by** *blast*
then show *?thesis*
proof *cases*
case *1*
then show *?thesis* **by** (*rule aux*)
next
case *2*
then show *?thesis*
using *aux* [*of x* −*y*]
by (*metis gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le*)
next
case *3*

```
    then show ?thesis
      using aux [of −x y]
    by (metis gcd.commute gcd-neg2-int mult.commute mult-minus-right neg-0-le-iff-le)
  next
    case 4
    then show ?thesis
      using aux [of −x −y]
    by (metis diff-0 diff-ge-0-iff-ge gcd-neg1-int gcd-neg2-int mult.commute mult-minus-right)
  qed
qed
```

Versions of Bezout for *nat*, by Amine Chaieb.

```
lemma Euclid-induct [case-names swap zero add]:
  fixes P :: nat ⇒ nat ⇒ bool
  assumes c: ⋀a b. P a b ⟷ P b a
    and z: ⋀a. P a 0
    and add: ⋀a b. P a b ⟶ P a (a + b)
  shows P a b
proof (induct a + b arbitrary: a b rule: less-induct)
  case less
  consider (eq) a = b | (lt) a < b a + b − a < a + b | b = 0 | b + a − b < a + b
    by arith
  show ?case
  proof (cases a b rule: linorder-cases)
    case equal
    with add [rule-format, OF z [rule-format, of a]] show ?thesis by simp
  next
    case lt: less
    then consider a = 0 | a + b − a < a + b by arith
    then show ?thesis
    proof cases
      case 1
      with z c show ?thesis by blast
    next
      case 2
      also have ∗: a + b − a = a + (b − a) using lt by arith
      finally have a + (b − a) < a + b .
      then have P a (a + (b − a)) by (rule add [rule-format, OF less])
      then show ?thesis by (simp add: ∗[symmetric])
    qed
  next
    case gt: greater
    then consider b = 0 | b + a − b < a + b by arith
    then show ?thesis
    proof cases
      case 1
      with z c show ?thesis by blast
    next
      case 2
```

```
    also have *: b + a - b = b + (a - b) using gt by arith
    finally have b + (a - b) < a + b .
    then have P b (b + (a - b)) by (rule add [rule-format, OF less])
    then have P b a by (simp add: *[symmetric])
    with c show ?thesis by blast
   qed
  qed
qed

lemma bezout-lemma-nat:
  fixes d::nat
  shows ⟦d dvd a; d dvd b; a * x = b * y + d ∨ b * x = a * y + d⟧
    ⟹ ∃x y. d dvd a ∧ d dvd a + b ∧ (a * x = (a + b) * y + d ∨ (a + b) * x =
a * y + d)
  apply auto
  apply (metis add-mult-distrib2 left-add-mult-distrib)
  apply (rule-tac x=x in exI)
  by (metis add-mult-distrib2 mult.commute add.assoc)

lemma bezout-add-nat:
  ∃(d::nat) x y. d dvd a ∧ d dvd b ∧ (a * x = b * y + d ∨ b * x = a * y + d)
proof (induct a b rule: Euclid-induct)
  case (swap a b)
  then show ?case
    by blast
next
  case (zero a)
  then show ?case
    by fastforce
next
  case (add a b)
  then show ?case
    by (meson bezout-lemma-nat)
qed

lemma bezout1-nat: ∃(d::nat) x y. d dvd a ∧ d dvd b ∧ (a * x - b * y = d ∨ b *
x - a * y = d)
  using bezout-add-nat[of a b]  by (metis add-diff-cancel-left')

lemma bezout-add-strong-nat:
  fixes a b :: nat
  assumes a: a ≠ 0
  shows ∃d x y. d dvd a ∧ d dvd b ∧ a * x = b * y + d
proof -
  consider d x y where d dvd a d dvd b a * x = b * y + d
        | d x y where d dvd a d dvd b b * x = a * y + d
    using bezout-add-nat [of a b] by blast
  then show ?thesis
  proof cases
```

```
  case 1
  then show ?thesis by blast
next
  case H: 2
  show ?thesis
  proof (cases b = 0)
    case True
    with H show ?thesis by simp
  next
    case False
    then have bp: b > 0 by simp
    with dvd-imp-le [OF H(2)] consider d = b | d < b
      by atomize-elim auto
    then show ?thesis
    proof cases
      case 1
      with a H show ?thesis
        by (metis Suc-pred add.commute mult.commute mult-Suc-right neq0-conv)
    next
      case 2
      show ?thesis
      proof (cases x = 0)
        case True
        with a H show ?thesis by simp
      next
        case x0: False
        then have xp: x > 0 by simp
        from ‹d < b› have d ≤ b - 1 by simp
        then have d * b ≤ b * (b - 1) by simp
        with xp mult-mono[of 1 x d * b b * (b - 1)]
        have dble: d * b ≤ x * b * (b - 1) using bp by simp
        from H(3) have d + (b - 1) * (b * x) = d + (b - 1) * (a * y + d)
          by simp
        then have d + (b - 1) * a * y + (b - 1) * d = d + (b - 1) * b * x
          by (simp only: mult.assoc distrib-left)
        then have a * ((b - 1) * y) + d * (b - 1 + 1) = d + x * b * (b - 1)
          by algebra
        then have a * ((b - 1) * y) = d + x * b * (b - 1) - d * b
          using bp by simp
        then have a * ((b - 1) * y) = d + (x * b * (b - 1) - d * b)
          by (simp only: diff-add-assoc[OF dble, of d, symmetric])
        then have a * ((b - 1) * y) = b * (x * (b - 1) - d) + d
          by (simp only: diff-mult-distrib2 ac-simps)
        with H(1,2) show ?thesis
          by blast
      qed
    qed
  qed
qed
```

qed

lemma *bezout-nat*:
fixes a :: *nat*
assumes *a*: $a \neq 0$
shows $\exists x\ y.\ a * x = b * y + gcd\ a\ b$
proof −
obtain $d\ x\ y$ **where** *d*: $d\ dvd\ a\ d\ dvd\ b$ **and** *eq*: $a * x = b * y + d$
using *bezout-add-strong-nat* [*OF a*, *of b*] **by** *blast*
from d **have** $d\ dvd\ gcd\ a\ b$
by *simp*
then obtain k **where** *k*: $gcd\ a\ b = d * k$
unfolding *dvd-def* **by** *blast*
from *eq* **have** $a * x * k = (b * y + d) * k$
by *auto*
then have $a * (x * k) = b * (y * k) + gcd\ a\ b$
by (*algebra add*: *k*)
then show *?thesis*
by *blast*
qed

87.8 LCM properties on *nat* and *int*

lemma *lcm-altdef-int* [*code*]: $lcm\ a\ b = |a| * |b|\ div\ gcd\ a\ b$
for $a\ b$ :: *int*
by (*simp add*: *abs-mult lcm-gcd abs-div*)

lemma *prod-gcd-lcm-nat*: $m * n = gcd\ m\ n * lcm\ m\ n$
for $m\ n$ :: *nat*
by (*simp add*: *lcm-gcd*)

lemma *prod-gcd-lcm-int*: $|m| * |n| = gcd\ m\ n * lcm\ m\ n$
for $m\ n$ :: *int*
by (*simp add*: *lcm-gcd abs-div abs-mult*)

lemma *lcm-pos-nat*: $m > 0 \Longrightarrow n > 0 \Longrightarrow lcm\ m\ n > 0$
for $m\ n$ :: *nat*
using *lcm-eq-0-iff* [*of m n*] **by** *auto*

lemma *lcm-pos-int*: $m \neq 0 \Longrightarrow n \neq 0 \Longrightarrow lcm\ m\ n > 0$
for $m\ n$ :: *int*
by (*simp add*: *less-le lcm-eq-0-iff*)

lemma *lcm-unique-nat*:
$a\ dvd\ d \wedge b\ dvd\ d \wedge (\forall e.\ a\ dvd\ e \wedge b\ dvd\ e \longrightarrow d\ dvd\ e) \longleftrightarrow d = lcm\ a\ b$
for $a\ b\ d$ :: *nat*
by (*auto intro*: *dvd-antisym lcm-least*)

lemma *lcm-unique-int*:

$d \geq 0 \wedge a \text{ dvd } d \wedge b \text{ dvd } d \wedge (\forall e.\ a \text{ dvd } e \wedge b \text{ dvd } e \longrightarrow d \text{ dvd } e) \longleftrightarrow d = \text{lcm } a\ b$
for $a\ b\ d :: int$
using *lcm-least zdvd-antisym-nonneg* **by** *auto*

lemma *lcm-proj2-if-dvd-nat* [*simp*]: $x \text{ dvd } y \Longrightarrow \text{lcm } x\ y = y$
for $x\ y :: nat$
by (*simp add*: *lcm-proj2-if-dvd*)

lemma *lcm-proj2-if-dvd-int* [*simp*]: $x \text{ dvd } y \Longrightarrow \text{lcm } x\ y = |y|$
for $x\ y :: int$
by (*simp add*: *lcm-proj2-if-dvd*)

lemma *lcm-proj1-if-dvd-nat* [*simp*]: $x \text{ dvd } y \Longrightarrow \text{lcm } y\ x = y$
for $x\ y :: nat$
by (*subst lcm.commute*) (*erule lcm-proj2-if-dvd-nat*)

lemma *lcm-proj1-if-dvd-int* [*simp*]: $x \text{ dvd } y \Longrightarrow \text{lcm } y\ x = |y|$
for $x\ y :: int$
by (*subst lcm.commute*) (*erule lcm-proj2-if-dvd-int*)

lemma *lcm-proj1-iff-nat* [*simp*]: $\text{lcm } m\ n = m \longleftrightarrow n \text{ dvd } m$
for $m\ n :: nat$
by (*metis lcm-proj1-if-dvd-nat lcm-unique-nat*)

lemma *lcm-proj2-iff-nat* [*simp*]: $\text{lcm } m\ n = n \longleftrightarrow m \text{ dvd } n$
for $m\ n :: nat$
by (*metis lcm-proj2-if-dvd-nat lcm-unique-nat*)

lemma *lcm-proj1-iff-int* [*simp*]: $\text{lcm } m\ n = |m| \longleftrightarrow n \text{ dvd } m$
for $m\ n :: int$
by (*metis dvd-abs-iff lcm-proj1-if-dvd-int lcm-unique-int*)

lemma *lcm-proj2-iff-int* [*simp*]: $\text{lcm } m\ n = |n| \longleftrightarrow m \text{ dvd } n$
for $m\ n :: int$
by (*metis dvd-abs-iff lcm-proj2-if-dvd-int lcm-unique-int*)

lemma *lcm-1-iff-nat* [*simp*]: $\text{lcm } m\ n = Suc\ 0 \longleftrightarrow m = Suc\ 0 \wedge n = Suc\ 0$
for $m\ n :: nat$
using *lcm-eq-1-iff* [*of m n*] **by** *simp*

lemma *lcm-1-iff-int* [*simp*]: $\text{lcm } m\ n = 1 \longleftrightarrow (m = 1 \vee m = -1) \wedge (n = 1 \vee n = -1)$
for $m\ n :: int$
by *auto*

87.9 The complete divisibility lattice on *nat* and *int*

Lifting *gcd* and *lcm* to sets (*Gcd* / *Lcm*). *Gcd* is defined via *Lcm* to facilitate the proof that we have a complete lattice.

instantiation *nat* :: *semiring-Gcd*
begin

interpretation *semilattice-neutr-set lcm 1*::*nat*
  by *standard simp-all*

definition *Lcm M* = (*if finite M then F M else 0*) **for** *M* :: *nat set*

lemma *Lcm-nat-empty*: *Lcm* {} = (*1*::*nat*)
  by (*simp add*: *Lcm-nat-def del*: *One-nat-def*)

lemma *Lcm-nat-insert*: *Lcm* (*insert n M*) = *lcm n* (*Lcm M*) **for** *n* :: *nat*
  by (*cases finite M*) (*auto simp*: *Lcm-nat-def simp del*: *One-nat-def*)

lemma *Lcm-nat-infinite*: *infinite M* ⟹ *Lcm M = 0* **for** *M* :: *nat set*
  by (*simp add*: *Lcm-nat-def*)

lemma *dvd-Lcm-nat* [*simp*]:
  fixes *M* :: *nat set*
  assumes *m* ∈ *M*
  shows *m dvd Lcm M*
proof −
  from *assms* **have** *insert m M = M*
    by *auto*
  moreover have *m dvd Lcm* (*insert m M*)
    by (*simp add*: *Lcm-nat-insert*)
  ultimately show *?thesis*
    by *simp*
qed

lemma *Lcm-dvd-nat* [*simp*]:
  fixes *M* :: *nat set*
  assumes ∀ *m*∈*M*. *m dvd n*
  shows *Lcm M dvd n*
proof (*cases n > 0*)
  case *False*
  then show *?thesis* **by** *simp*
next
  case *True*
  then have *finite* {*d*. *d dvd n*}
    by (*rule finite-divisors-nat*)
  moreover have *M* ⊆ {*d*. *d dvd n*}
    using *assms* **by** *fast*
  ultimately have *finite M*
    by (*rule rev-finite-subset*)

then show *?thesis*
using *assms* **by** (*induct M*) (*simp-all add*: *Lcm-nat-empty Lcm-nat-insert*)
qed

definition *Gcd M* = *Lcm* {*d*. ∀ *m*∈*M*. *d dvd m*} **for** *M* :: *nat set*

instance
proof
fix *N* :: *nat set*
fix *n* :: *nat*
show *Gcd N dvd n* **if** *n* ∈ *N*
using *that* **by** (*induct N rule*: *infinite-finite-induct*) (*auto simp*: *Gcd-nat-def*)
show *n dvd Gcd N* **if** ⋀*m*. *m* ∈ *N* ⟹ *n dvd m*
using *that* **by** (*induct N rule*: *infinite-finite-induct*) (*auto simp*: *Gcd-nat-def*)
show *n dvd Lcm N* **if** *n* ∈ *N*
using *that* **by** (*induct N rule*: *infinite-finite-induct*) *auto*
show *Lcm N dvd n* **if** ⋀*m*. *m* ∈ *N* ⟹ *m dvd n*
using *that* **by** (*induct N rule*: *infinite-finite-induct*) *auto*
show *normalize* (*Gcd N*) = *Gcd N* **and** *normalize* (*Lcm N*) = *Lcm N*
by *simp-all*
qed

end

lemma *Gcd-nat-eq-one*: *1* ∈ *N* ⟹ *Gcd N* = *1*
for *N* :: *nat set*
by (*rule Gcd-eq-1-I*) *auto*

instance *nat* :: *semiring-gcd-mult-normalize*
by *intro-classes* (*auto simp*: *unit-factor-nat-def*)

Alternative characterizations of Gcd:

lemma *Gcd-eq-Max*:
fixes *M* :: *nat set*
assumes *finite* (*M*::*nat set*) **and** *M* ≠ {} **and** *0* ∉ *M*
shows *Gcd M* = *Max* (⋂ *m*∈*M*. {*d*. *d dvd m*})
proof (*rule antisym*)
from *assms* **obtain** *m* **where** *m* ∈ *M* **and** *m* > *0*
by *auto*
from ‹*m* > *0*› **have** *finite* {*d*. *d dvd m*}
by (*blast intro*: *finite-divisors-nat*)
with ‹*m* ∈ *M*› **have** *fin*: *finite* (⋂ *m*∈*M*. {*d*. *d dvd m*})
by *blast*
from *fin* **show** *Gcd M* ≤ *Max* (⋂ *m*∈*M*. {*d*. *d dvd m*})
by (*auto intro*: *Max-ge Gcd-dvd*)
from *fin* **show** *Max* (⋂ *m*∈*M*. {*d*. *d dvd m*}) ≤ *Gcd M*
proof (*rule Max.boundedI*, *simp-all*)
show (⋂ *m*∈*M*. {*d*. *d dvd m*}) ≠ {}
by *auto*

show $\bigwedge a.\ \forall x{\in}M.\ a\ dvd\ x \Longrightarrow a \leq Gcd\ M$
by (*meson Gcd-dvd Gcd-greatest* ‹$0 < m$› ‹$m \in M$› *dvd-imp-le dvd-pos-nat*)
qed
qed

lemma *Gcd-remove0-nat*: $Gcd\ M = Gcd\ (M - \{0\})$
for M :: *nat set*
proof−
have $(\forall\ m \in M.\ b\ dvd\ m) \longleftrightarrow (\forall\ m \in (M - \{0\}).\ b\ dvd\ m)$ **for** b
by *blast*+
thus *?thesis*
unfolding *Gcd-Lcm* **by** *presburger*
qed

lemma *Lcm-in-lcm-closed-set-nat*:
fixes M :: *nat set*
assumes *finite* M $M \neq \{\}$ $\bigwedge m\ n.\ [\![m \in M;\ n \in M]\!] \Longrightarrow lcm\ m\ n \in M$
shows $Lcm\ M \in M$
using *assms*
proof (*induction M rule*: *finite-linorder-min-induct*)
case (*insert x M*)
then have $\bigwedge m\ n.\ m \in M \Longrightarrow n \in M \Longrightarrow lcm\ m\ n \in M$
by (*metis dvd-lcm1 gr0I insert-iff lcm-pos-nat nat-dvd-not-less*)
with *insert* **show** *?case*
by *simp* (*metis Lcm-nat-empty One-nat-def dvd-1-left dvd-lcm2*)
qed *auto*

lemma *Lcm-eq-Max-nat*:
fixes M :: *nat set*
assumes M: *finite* M $M \neq \{\}$ $0 \notin M$ **and** *lcm*: $\bigwedge m\ n.\ [\![m \in M;\ n \in M]\!] \Longrightarrow$ $lcm\ m\ n \in M$
shows $Lcm\ M = Max\ M$
proof (*rule antisym*)
show $Lcm\ M \leq Max\ M$
by (*simp add*: *Lcm-in-lcm-closed-set-nat* ‹*finite* M› ‹$M \neq \{\}$› *lcm*)
show $Max\ M \leq Lcm\ M$
by (*meson Lcm-0-iff Max-in M dvd-Lcm dvd-imp-le le-0-eq not-le*)
qed

lemma *mult-inj-if-coprime-nat*:
inj-on f A $\Longrightarrow$ *inj-on g B* $\Longrightarrow$ $(\bigwedge a\ b.\ [\![a{\in}A;\ b{\in}B]\!] \Longrightarrow coprime\ (f\ a)\ (g\ b)) \Longrightarrow$
inj-on $(\lambda(a,\ b).\ f\ a * g\ b)\ (A \times B)$
for f :: $'a \Rightarrow nat$ **and** g :: $'b \Rightarrow nat$
by (*auto simp*: *inj-on-def coprime-crossproduct-nat simp del*: *One-nat-def*)

87.9.1 Setwise GCD and LCM for integers

instantiation *int* :: *Gcd*
begin

definition *Gcd-int* :: *int set* ⇒ *int*
where *Gcd K* = *int* (*GCD k*∈*K*. (*nat* ∘ *abs*) *k*)

definition *Lcm-int* :: *int set* ⇒ *int*
where *Lcm K* = *int* (*LCM k*∈*K*. (*nat* ∘ *abs*) *k*)

instance ..

end

lemma *Gcd-int-eq* [*simp*]:
(*GCD n*∈*N*. *int n*) = *int* (*Gcd N*)
by (*simp add*: *Gcd-int-def image-image*)

lemma *Gcd-nat-abs-eq* [*simp*]:
(*GCD k*∈*K*. *nat* |*k*|) = *nat* (*Gcd K*)
by (*simp add*: *Gcd-int-def*)

lemma *abs-Gcd-eq* [*simp*]:
|*Gcd K*| = *Gcd K* **for** *K* :: *int set*
by (*simp only*: *Gcd-int-def*)

lemma *uminus-Gcd-eq* [*simp*]:
fixes *K*::*int set*
shows *Gcd* (*uminus* ' *K*) = *Gcd K*
unfolding *Gcd-int-def o-def* **by** (*simp add*: *image-image*)

lemma *Gcd-int-greater-eq-0* [*simp*]:
Gcd K ≥ *0*
for *K* :: *int set*
using *abs-ge-zero* [*of Gcd K*] **by** *simp*

lemma *Gcd-abs-eq* [*simp*]:
(*GCD k*∈*K*. |*k*|) = *Gcd K*
for *K* :: *int set*
by (*simp only*: *Gcd-int-def image-image*) *simp*

lemma *Lcm-int-eq* [*simp*]:
(*LCM n*∈*N*. *int n*) = *int* (*Lcm N*)
by (*simp add*: *Lcm-int-def image-image*)

lemma *Lcm-nat-abs-eq* [*simp*]:
(*LCM k*∈*K*. *nat* |*k*|) = *nat* (*Lcm K*)
by (*simp add*: *Lcm-int-def*)

lemma *abs-Lcm-eq* [*simp*]:
|*Lcm K*| = *Lcm K* **for** *K* :: *int set*
by (*simp only*: *Lcm-int-def*)

lemma *Lcm-int-greater-eq-0* [*simp*]:
  Lcm K $\geq$ *0*
  for *K* :: *int set*
  using *abs-ge-zero* [*of Lcm K*] **by** *simp*

lemma *Lcm-abs-eq* [*simp*]:
  (*LCM k*$\in$*K*. |*k*|) = *Lcm K*
  for *K* :: *int set*
  by (*simp only*: *Lcm-int-def image-image*) *simp*

instance *int* :: *semiring-Gcd*
proof
  fix *K* :: *int set* **and** *k* :: *int*
  show *Gcd K dvd k* **and** *k dvd Lcm K* **if** *k* $\in$ *K*
    using *that Gcd-dvd* [*of nat* |*k*| (*nat* $\circ$ *abs*) ‘ *K*]
      dvd-Lcm [*of nat* |*k*| (*nat* $\circ$ *abs*) ‘ *K*]
    by (*simp-all add*: *comp-def*)
  show *k dvd Gcd K* **if** $\bigwedge$*l*. *l* $\in$ *K* $\Longrightarrow$ *k dvd l*
  proof −
    have *nat* |*k*| *dvd* (*GCD k*$\in$*K*. *nat* |*k*|)
      by (*rule Gcd-greatest*) (*use that* **in** *auto*)
    then show *?thesis* **by** *simp*
  qed
  show *Lcm K dvd k* **if** $\bigwedge$*l*. *l* $\in$ *K* $\Longrightarrow$ *l dvd k*
  proof −
    have (*LCM k*$\in$*K*. *nat* |*k*|) *dvd nat* |*k*|
      by (*rule Lcm-least*) (*use that* **in** *auto*)
    then show *?thesis* **by** *simp*
  qed
qed (*simp-all add*: *sgn-mult*)

instance *int* :: *semiring-gcd-mult-normalize*
  by *intro-classes* (*auto simp*: *sgn-mult*)

87.10 GCD and LCM on *integer*

instantiation *integer* :: *gcd*
begin

context
  includes *integer.lifting*
begin

lift-definition *gcd-integer* :: *integer* $\Rightarrow$ *integer* $\Rightarrow$ *integer* **is** *gcd* **.**

lift-definition *lcm-integer* :: *integer* $\Rightarrow$ *integer* $\Rightarrow$ *integer* **is** *lcm* **.**

end

instance ..

end

lifting-update *integer.lifting*
lifting-forget *integer.lifting*

context
  includes *integer.lifting*
begin

lemma *gcd-code-integer* [*code*]: *gcd k l* = |*if l* = (*0*::*integer*) *then k else gcd l* (|*k*| *mod* |*l*|)|
  by *transfer* (*fact gcd-code-int*)

lemma *lcm-code-integer* [*code*]: *lcm a b* = |*a*| ∗ |*b*| *div gcd a b*
  for *a b* :: *integer*
  by *transfer* (*fact lcm-altdef-int*)

end

code-printing
  constant *gcd* :: *integer* ⇒ - ⇀
    (*OCaml*) !(*fun k l -> if Z.equal k Z.zero then/ Z.abs l else if Z.equal/ l Z.zero then Z.abs k else Z.gcd k l*)
  and (*Haskell*) *Prelude.gcd*
  and (*Scala*) *-.gcd'*((-)')
  — There is no gcd operation in the SML standard library, so no code setup for SML

Some code equations

lemmas *Gcd-nat-set-eq-fold* [*code*] = *Gcd-set-eq-fold* [**where** *?'a* = *nat*]
lemmas *Lcm-nat-set-eq-fold* [*code*] = *Lcm-set-eq-fold* [**where** *?'a* = *nat*]
lemmas *Gcd-int-set-eq-fold* [*code*] = *Gcd-set-eq-fold* [**where** *?'a* = *int*]
lemmas *Lcm-int-set-eq-fold* [*code*] = *Lcm-set-eq-fold* [**where** *?'a* = *int*]

Fact aliases.

lemma *lcm-0-iff-nat* [*simp*]: *lcm m n* = *0* ⟷ *m* = *0* ∨ *n* = *0*
  for *m n* :: *nat*
  by (*fact lcm-eq-0-iff*)

lemma *lcm-0-iff-int* [*simp*]: *lcm m n* = *0* ⟷ *m* = *0* ∨ *n* = *0*
  for *m n* :: *int*
  by (*fact lcm-eq-0-iff*)

lemma *dvd-lcm-I1-nat* [*simp*]: *k dvd m* ⟹ *k dvd lcm m n*
  for *k m n* :: *nat*
  by (*fact dvd-lcmI1*)

lemma *dvd-lcm-I2-nat* [*simp*]: *k dvd n* ⟹ *k dvd lcm m n*
for *k m n* :: *nat*
by (*fact dvd-lcmI2*)

lemma *dvd-lcm-I1-int* [*simp*]: *i dvd m* ⟹ *i dvd lcm m n*
for *i m n* :: *int*
by (*fact dvd-lcmI1*)

lemma *dvd-lcm-I2-int* [*simp*]: *i dvd n* ⟹ *i dvd lcm m n*
for *i m n* :: *int*
by (*fact dvd-lcmI2*)

lemmas *Gcd-dvd-nat* [*simp*] = *Gcd-dvd* [**where** *?'a* = *nat*]
lemmas *Gcd-dvd-int* [*simp*] = *Gcd-dvd* [**where** *?'a* = *int*]
lemmas *Gcd-greatest-nat* [*simp*] = *Gcd-greatest* [**where** *?'a* = *nat*]
lemmas *Gcd-greatest-int* [*simp*] = *Gcd-greatest* [**where** *?'a* = *int*]

lemma *dvd-Lcm-int* [*simp*]: *m* ∈ *M* ⟹ *m dvd Lcm M*
for *M* :: *int set*
by (*fact dvd-Lcm*)

lemma *gcd-neg-numeral-1-int* [*simp*]: *gcd* (− *numeral n* :: *int*) *x* = *gcd* (*numeral n*) *x*
by (*fact gcd-neg1-int*)

lemma *gcd-neg-numeral-2-int* [*simp*]: *gcd x* (− *numeral n* :: *int*) = *gcd x* (*numeral n*)
by (*fact gcd-neg2-int*)

lemma *gcd-proj1-if-dvd-nat* [*simp*]: *x dvd y* ⟹ *gcd x y* = *x*
for *x y* :: *nat*
by (*fact gcd-nat.absorb1*)

lemma *gcd-proj2-if-dvd-nat* [*simp*]: *y dvd x* ⟹ *gcd x y* = *y*
for *x y* :: *nat*
by (*fact gcd-nat.absorb2*)

lemma *Gcd-in*:
fixes *A* :: *nat set*
assumes ⋀*a b*. *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *gcd a b* ∈ *A*
assumes *A* ≠ {}
shows *Gcd A* ∈ *A*
proof (*cases A* = {*0*})
case *False*
with *assms* **obtain** *x* **where** *x* ∈ *A x* > *0*
by *auto*
thus *Gcd A* ∈ *A*
proof (*induction x rule*: *less-induct*)

```
    case (less x)
    show ?case
    proof (cases x = Gcd A)
      case False
      have ∃ y∈A. ¬x dvd y
        using False less.prems by (metis Gcd-dvd Gcd-greatest-nat gcd-nat.asym)
      then obtain y where y: y ∈ A ¬x dvd y
        by blast
      have gcd x y ∈ A
        by (rule assms(1)) (use ‹x ∈ A› y in auto)
      moreover have gcd x y < x
        using ‹x > 0› y by (metis gcd-dvd1 gcd-dvd2 nat-dvd-not-less nat-neq-iff)
      moreover have gcd x y > 0
        using ‹x > 0› by auto
      ultimately show ?thesis using less.IH by blast
    qed (use less in auto)
  qed
qed auto

lemma bezout-gcd-nat':
  fixes a b :: nat
  shows ∃ x y. b * y ≤ a * x ∧ a * x − b * y = gcd a b ∨ a * y ≤ b * x ∧ b * x
− a * y = gcd a b
  using bezout-nat[of a b]
  by (metis add-diff-cancel-left' diff-zero gcd.commute gcd-0-nat
          le-add-same-cancel1 mult.right-neutral zero-le)

lemmas Lcm-eq-0-I-nat [simp] = Lcm-eq-0-I [where ?'a = nat]
lemmas Lcm-0-iff-nat [simp] = Lcm-0-iff [where ?'a = nat]
lemmas Lcm-least-int [simp] = Lcm-least [where ?'a = int]
```

87.11 Characteristic of a semiring

```
definition (in semiring-1) semiring-char :: 'a itself ⇒ nat
  where semiring-char - = Gcd {n. of-nat n = (0 :: 'a)}

syntax -type-char :: type => nat (‹(‹indent=1 notation=‹mixfix CHAR››CHAR/(1'(-')))›)
syntax-consts -type-char ⇌ semiring-char
translations CHAR('t) ⇀ CONST semiring-char (CONST Pure.type :: 't itself)
print-translation ‹
  let
    fun char-type-tr' ctxt [Const (const-syntax ‹Pure.type›, Type (-, [T]))] =
      Syntax.const syntax-const ‹-type-char› $ Syntax-Phases.term-of-typ ctxt T
  in [(const-syntax ‹semiring-char›, char-type-tr')] end
›

context semiring-1
begin
```

lemma *of-nat-CHAR* [*simp*]: *of-nat CHAR('a) = (0 :: 'a)*
proof −
  have *CHAR('a) ∈ {n. of-nat n = (0::'a)}*
    unfolding *semiring-char-def*
  proof (*rule Gcd-in*, *clarify*)
    fix *a b :: nat*
    assume ∗: *of-nat a = (0 :: 'a) of-nat b = (0 :: 'a)*
    show *of-nat (gcd a b) = (0 :: 'a)*
    proof (*cases a = 0*)
      case *False*
      with *bezout-nat* **obtain** *x y* **where** *a ∗ x = b ∗ y + gcd a b*
        by *blast*
      hence *of-nat (a ∗ x) = (of-nat (b ∗ y + gcd a b) :: 'a)*
        by (*rule arg-cong*)
      thus *of-nat (gcd a b) = (0 :: 'a)*
        using ∗ **by** *simp*
    qed (*use* ∗ **in** *auto*)
  next
    have *of-nat 0 = (0 :: 'a)*
      by *simp*
    thus *{n. of-nat n = (0 :: 'a)} ≠ {}*
      by *blast*
  qed
  thus *?thesis*
    by *simp*
qed

lemma *of-nat-eq-0-iff-char-dvd*: *of-nat n = (0 :: 'a) ⟷ CHAR('a) dvd n*
proof
  assume *of-nat n = (0 :: 'a)*
  thus *CHAR('a) dvd n*
    unfolding *semiring-char-def* **by** (*intro Gcd-dvd*) *auto*
next
  assume *CHAR('a) dvd n*
  then obtain *m* **where** *n = CHAR('a) ∗ m*
    by *auto*
  thus *of-nat n = (0 :: 'a)*
    by *simp*
qed

lemma *CHAR-eqI*:
  assumes *of-nat c = (0 :: 'a)*
  assumes ⋀*x*. *of-nat x = (0 :: 'a) ⟹ c dvd x*
  shows *CHAR('a) = c*
  using *assms* **by** (*intro dvd-antisym*) (*auto simp*: *of-nat-eq-0-iff-char-dvd*)

lemma *CHAR-eq0-iff*: *CHAR('a) = 0 ⟷ (∀ n>0. of-nat n ≠ (0::'a))*
  by (*auto simp*: *of-nat-eq-0-iff-char-dvd*)

lemma *CHAR-pos-iff*: $CHAR('a) > 0 \longleftrightarrow (\exists\, n{>}0.\ of\text{-}nat\ n = (0::'a))$
using *CHAR-eq0-iff neq0-conv* **by** *blast*

lemma *CHAR-eq-posI*:
assumes $c > 0$ *of-nat* $c = (0 :: 'a)$ $\bigwedge x.\ x > 0 \Longrightarrow x < c \Longrightarrow of\text{-}nat\ x \neq (0 :: 'a)$
shows $CHAR('a) = c$
proof (*rule antisym*)
from *assms* **have** $CHAR('a) > 0$
by (*auto simp*: *CHAR-pos-iff*)
from *assms*(*3*)[*OF this*] **show** $CHAR('a) \geq c$
by *force*
next
have $CHAR('a)$ *dvd c*
using *assms* **by** (*auto simp*: *of-nat-eq-0-iff-char-dvd*)
thus $CHAR('a) \leq c$
using ‹$c > 0$› **by** (*intro dvd-imp-le*) *auto*
qed

end

lemma (**in** *semiring-char-0*) *CHAR-eq-0* [*simp*]: $CHAR('a) = 0$
by (*simp add*: *CHAR-eq0-iff*)

lemma *CHAR-not-1* [*simp*]: $CHAR('a :: \{semiring\text{-}1,\ zero\text{-}neq\text{-}one\}) \neq Suc\ 0$
by (*metis One-nat-def of-nat-1 of-nat-CHAR zero-neq-one*)

lemma (**in** *idom*) *CHAR-not-1'* [*simp*]: $CHAR('a) \neq Suc\ 0$
using *local.of-nat-CHAR* **by** *fastforce*

lemma (**in** *ring-1*) *uminus-CHAR-2*:
assumes $CHAR('a) = 2$
shows $-(x :: 'a) = x$
proof −
have $x + x = 2 * x$
by (*simp add*: *mult-2*)
also have $2 = (0 :: 'a)$
using *assms local.of-nat-CHAR* **by** *auto*
finally show *?thesis*
by (*simp add*: *add-eq-0-iff2*)
qed

lemma (**in** *ring-1*) *minus-CHAR-2*:
assumes $CHAR('a) = 2$
shows $(x - y :: 'a) = x + y$
proof −
have $x - y = x + (-y)$
by *simp*

also have $-y = y$
by (*rule uminus-CHAR-2*) *fact*
finally show *?thesis* **.**
qed

lemma (**in** *semiring-1-cancel*) *of-nat-eq-iff-char-dvd*:
assumes $m < n$
shows $\mathit{of\text{-}nat}\ m = (\mathit{of\text{-}nat}\ n :: 'a) \longleftrightarrow \mathit{CHAR}('a)\ \mathit{dvd}\ (n - m)$
proof
assume $*$: $\mathit{of\text{-}nat}\ m = (\mathit{of\text{-}nat}\ n :: 'a)$
have $\mathit{of\text{-}nat}\ n = (\mathit{of\text{-}nat}\ m + \mathit{of\text{-}nat}\ (n - m) :: 'a)$
using *assms* **by** (*metis le-add-diff-inverse local.of-nat-add nless-le*)
hence $\mathit{of\text{-}nat}\ (n - m) = (0 :: 'a)$
by (*simp add*: $*$)
thus $\mathit{CHAR}('a)\ \mathit{dvd}\ (n - m)$
by (*simp add*: *of-nat-eq-0-iff-char-dvd*)
next
assume $\mathit{CHAR}('a)\ \mathit{dvd}\ (n - m)$
hence $\mathit{of\text{-}nat}\ (n - m) = (0 :: 'a)$
by (*simp add*: *of-nat-eq-0-iff-char-dvd*)
hence $\mathit{of\text{-}nat}\ m = (\mathit{of\text{-}nat}\ m + \mathit{of\text{-}nat}\ (n - m) :: 'a)$
by *simp*
also have $\ldots = \mathit{of\text{-}nat}\ n$
using *assms* **by** (*metis le-add-diff-inverse local.of-nat-add nless-le*)
finally show $\mathit{of\text{-}nat}\ m = (\mathit{of\text{-}nat}\ n :: 'a)$ **.**
qed

lemma (**in** *ring-1*) *of-int-eq-0-iff-char-dvd*:
$(\mathit{of\text{-}int}\ n = (0 :: 'a)) = (\mathit{int}\ \mathit{CHAR}('a)\ \mathit{dvd}\ n)$
proof (*cases* $n \geq 0$)
case *True*
hence $(\mathit{of\text{-}int}\ n = (0 :: 'a)) \longleftrightarrow (\mathit{of\text{-}nat}\ (\mathit{nat}\ n)) = (0 :: 'a)$
by *auto*
also have $\ldots \longleftrightarrow \mathit{CHAR}('a)\ \mathit{dvd}\ \mathit{nat}\ n$
by (*subst of-nat-eq-0-iff-char-dvd*) *auto*
also have $\ldots \longleftrightarrow \mathit{int}\ \mathit{CHAR}('a)\ \mathit{dvd}\ n$
using *True* **by** *presburger*
finally show *?thesis* **.**
next
case *False*
hence $(\mathit{of\text{-}int}\ n = (0 :: 'a)) \longleftrightarrow -(\mathit{of\text{-}nat}\ (\mathit{nat}\ (-n))) = (0 :: 'a)$
by *auto*
also have $\ldots \longleftrightarrow \mathit{CHAR}('a)\ \mathit{dvd}\ \mathit{nat}\ (-n)$
by (*auto simp*: *of-nat-eq-0-iff-char-dvd*)
also have $\ldots \longleftrightarrow \mathit{int}\ \mathit{CHAR}('a)\ \mathit{dvd}\ n$
using *False dvd-nat-abs-iff*[*of* $\mathit{CHAR}('a)\ n$] **by** *simp*
finally show *?thesis* **.**
qed

lemma (**in** *semiring-1-cancel*) *finite-imp-CHAR-pos*:
assumes *finite* (*UNIV* :: *'a set*)
shows *CHAR*(*'a*) > *0*
proof −
have ∃ *n*∈*UNIV*. *infinite* {*m* ∈ *UNIV*. *of-nat m* = (*of-nat n* :: *'a*)}
proof (*rule pigeonhole-infinite*)
show *infinite* (*UNIV* :: *nat set*)
by *simp*
show *finite* (*range* (*of-nat* :: *nat* ⇒ *'a*))
by (*rule finite-subset*[*OF - assms*]) *auto*
qed
then obtain *n* :: *nat* **where** *infinite* {*m* ∈ *UNIV*. *of-nat m* = (*of-nat n* :: *'a*)}
by *blast*
hence ¬({*m* ∈ *UNIV*. *of-nat m* = (*of-nat n* :: *'a*)} ⊆ {*n*})
by (*intro notI*) (*use finite-subset* **in** *blast*)
then obtain *m* **where** *m* ≠ *n of-nat m* = (*of-nat n* :: *'a*)
by *blast*
thus *?thesis*
proof (*induction m n rule*: *linorder-wlog*)
case (*le m n*)
hence *CHAR*(*'a*) *dvd* (*n* − *m*)
using *of-nat-eq-iff-char-dvd*[*of m n*] **by** *auto*
thus *?thesis*
using *le* **by** (*intro Nat.gr0I*) *auto*
qed (*simp-all add*: *eq-commute*)
qed

end

88 Nitpick: Yet Another Counterexample Generator for Isabelle/HOL

theory *Nitpick*
imports *Record GCD*
keywords
nitpick :: *diag* **and**
nitpick-params :: *thy-decl*
begin

datatype (*plugins only*: *extraction*) (*dead 'a*, *dead 'b*) *fun-box* = *FunBox 'a* ⇒ *'b*
datatype (*plugins only*: *extraction*) (*dead 'a*, *dead 'b*) *pair-box* = *PairBox 'a 'b*
datatype (*plugins only*: *extraction*) (*dead 'a*) *word* = *Word 'a set*

typedecl *bisim-iterator*
typedecl *unsigned-bit*
typedecl *signed-bit*

consts

unknown :: *'a*
is-unknown :: *'a ⇒ bool*
bisim :: *bisim-iterator ⇒ 'a ⇒ 'a ⇒ bool*
bisim-iterator-max :: *bisim-iterator*
Quot :: *'a ⇒ 'b*
safe-The :: *('a ⇒ bool) ⇒ 'a*

Alternative definitions.

lemma *Ex1-unfold*[*nitpick-unfold*]: *Ex1 P ≡ ∃ x. {x. P x} = {x}*
apply (*rule eq-reflection*)
apply (*simp add*: *Ex1-def set-eq-iff*)
apply (*rule iffI*)
apply (*erule exE*)
apply (*erule conjE*)
apply (*rule-tac x = x* **in** *exI*)
apply (*rule allI*)
apply (*rename-tac y*)
apply (*erule-tac x = y* **in** *allE*)
by *auto*

lemma *rtrancl-unfold*[*nitpick-unfold*]: $r^* \equiv (r^+)^=$
by (*simp only*: *rtrancl-trancl-reflcl*)

lemma *rtranclp-unfold*[*nitpick-unfold*]: *rtranclp r a b ≡ (a = b ∨ tranclp r a b)*
by (*rule eq-reflection*) (*auto dest*: *rtranclpD*)

lemma *tranclp-unfold*[*nitpick-unfold*]:
tranclp r a b ≡ (a, b) ∈ trancl {(x, y). r x y}
by (*simp add*: *trancl-def*)

lemma [*nitpick-simp*]:
of-nat n = (if n = 0 then 0 else 1 + of-nat (n − 1))
by (*cases n*) *auto*

definition *prod* :: *'a set ⇒ 'b set ⇒ ('a × 'b) set* **where**
prod A B = {(a, b). a ∈ A ∧ b ∈ B}

definition *refl'* :: *('a × 'a) set ⇒ bool* **where**
refl' r ≡ ∀ x. (x, x) ∈ r

definition *wf'* :: *('a × 'a) set ⇒ bool* **where**
wf' r ≡ acyclic r ∧ (finite r ∨ unknown)

definition *card'* :: *'a set ⇒ nat* **where**
card' A ≡ if finite A then length (SOME xs. set xs = A ∧ distinct xs) else 0

definition *sum'* :: *('a ⇒ 'b*::*comm-monoid-add) ⇒ 'a set ⇒ 'b* **where**
sum' f A ≡ if finite A then sum-list (map f (SOME xs. set xs = A ∧ distinct xs))
else 0

inductive *fold-graph'* :: $('a \Rightarrow 'b \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ set \Rightarrow 'b \Rightarrow bool$ **where**
fold-graph' f z {} z |
$[\![x \in A;$ *fold-graph' f z* $(A - \{x\})\ y]\!] \Longrightarrow$ *fold-graph' f z A* $(f\ x\ y)$

The following lemmas are not strictly necessary but they help the *specialize* optimization.

lemma *The-psimp*[*nitpick-psimp*]: $P = (=)\ x \Longrightarrow The\ P = x$
by *auto*

lemma *Eps-psimp*[*nitpick-psimp*]:
$[\![P\ x;\ \neg\ P\ y;\ Eps\ P = y]\!] \Longrightarrow Eps\ P = x$
apply (*cases P* (*Eps P*))
apply *auto*
apply (*erule contrapos-np*)
by (*rule someI*)

lemma *case-unit-unfold*[*nitpick-unfold*]:
case-unit x u $\equiv x$
apply (*subgoal-tac u* $= ()$)
apply (*simp only*: *unit.case*)
by *simp*

declare *unit.case*[*nitpick-simp del*]

lemma *case-nat-unfold*[*nitpick-unfold*]:
case-nat x f n $\equiv$ *if n* $= 0$ *then x else f* $(n - 1)$
apply (*rule eq-reflection*)
by (*cases n*) *auto*

declare *nat.case*[*nitpick-simp del*]

lemma *size-list-simp*[*nitpick-simp*]:
size-list f xs $=$ (*if xs* $= []$ *then 0 else Suc* (*f* (*hd xs*) $+$ *size-list f* (*tl xs*)))
size xs $=$ (*if xs* $= []$ *then 0 else Suc* (*size* (*tl xs*)))
by (*cases xs*) *auto*

Auxiliary definitions used to provide an alternative representation for *rat* and *real*.

fun *nat-gcd* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
nat-gcd x y $=$ (*if y* $= 0$ *then x else nat-gcd y* (*x mod y*))

declare *nat-gcd.simps* [*simp del*]

definition *nat-lcm* :: $nat \Rightarrow nat \Rightarrow nat$ **where**
nat-lcm x y $= x * y$ *div* (*nat-gcd x y*)

lemma *gcd-eq-nitpick-gcd* [*nitpick-unfold*]:
gcd x y $=$ *Nitpick.nat-gcd x y*

by (*induct x y rule*: *nat-gcd.induct*)
(*simp add*: *gcd-nat.simps Nitpick.nat-gcd.simps*)

lemma *lcm-eq-nitpick-lcm* [*nitpick-unfold*]:
lcm x y = *Nitpick.nat-lcm x y*
by (*simp only*: *lcm-nat-def Nitpick.nat-lcm-def gcd-eq-nitpick-gcd*)

definition *Frac* :: *int* × *int* ⇒ *bool* **where**
Frac ≡ λ(*a*, *b*). *b* > *0* ∧ *coprime a b*

consts
Abs-Frac :: *int* × *int* ⇒ *'a*
Rep-Frac :: *'a* ⇒ *int* × *int*

definition *zero-frac* :: *'a* **where**
zero-frac ≡ *Abs-Frac* (*0*, *1*)

definition *one-frac* :: *'a* **where**
one-frac ≡ *Abs-Frac* (*1*, *1*)

definition *num* :: *'a* ⇒ *int* **where**
num ≡ *fst* ∘ *Rep-Frac*

definition *denom* :: *'a* ⇒ *int* **where**
denom ≡ *snd* ∘ *Rep-Frac*

function *norm-frac* :: *int* ⇒ *int* ⇒ *int* × *int* **where**
norm-frac a b =
(*if b* < *0 then norm-frac* (− *a*) (− *b*)
else if a = *0* ∨ *b* = *0 then* (*0*, *1*)
else let c = *gcd a b in* (*a div c*, *b div c*))
by *pat-completeness auto*
termination by (*relation measure* (λ(-, *b*). *if b* < *0 then 1 else 0*)) *auto*

declare *norm-frac.simps*[*simp del*]

definition *frac* :: *int* ⇒ *int* ⇒ *'a* **where**
frac a b ≡ *Abs-Frac* (*norm-frac a b*)

definition *plus-frac* :: *'a* ⇒ *'a* ⇒ *'a* **where**
[*nitpick-simp*]: *plus-frac q r* = (*let d* = *lcm* (*denom q*) (*denom r*) *in*
frac (*num q* ∗ (*d div denom q*) + *num r* ∗ (*d div denom r*)) *d*)

definition *times-frac* :: *'a* ⇒ *'a* ⇒ *'a* **where**
[*nitpick-simp*]: *times-frac q r* = *frac* (*num q* ∗ *num r*) (*denom q* ∗ *denom r*)

definition *uminus-frac* :: *'a* ⇒ *'a* **where**
uminus-frac q ≡ *Abs-Frac* (− *num q*, *denom q*)

definition *number-of-frac* :: *int* ⇒ *'a* **where**
number-of-frac n ≡ *Abs-Frac* (*n*, *1*)

definition *inverse-frac* :: *'a* ⇒ *'a* **where**
inverse-frac q ≡ *frac* (*denom q*) (*num q*)

definition *less-frac* :: *'a* ⇒ *'a* ⇒ *bool* **where**
[*nitpick-simp*]: *less-frac q r* ⟷ *num* (*plus-frac q* (*uminus-frac r*)) < *0*

definition *less-eq-frac* :: *'a* ⇒ *'a* ⇒ *bool* **where**
[*nitpick-simp*]: *less-eq-frac q r* ⟷ *num* (*plus-frac q* (*uminus-frac r*)) ≤ *0*

definition *of-frac* :: *'a* ⇒ *'b*::{*inverse*,*ring-1*} **where**
of-frac q ≡ *of-int* (*num q*) / *of-int* (*denom q*)

axiomatization *wf-wfrec* :: (*'a* × *'a*) *set* ⇒ ((*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*

definition *wf-wfrec'* :: (*'a* × *'a*) *set* ⇒ ((*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b* **where**
[*nitpick-simp*]: *wf-wfrec' R F x* = *F* (*Wfrec.cut* (*wf-wfrec R F*) *R x*) *x*

definition *wfrec'* :: (*'a* × *'a*) *set* ⇒ ((*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b* **where**
wfrec' R F x ≡ *if wf R then wf-wfrec' R F x else THE y. wfrec-rel R* (λ*f x. F* (*Wfrec.cut f R x*) *x*) *x y*

ML-file ‹*Tools/Nitpick/kodkod.ML*›
ML-file ‹*Tools/Nitpick/kodkod-sat.ML*›
ML-file ‹*Tools/Nitpick/nitpick-util.ML*›
ML-file ‹*Tools/Nitpick/nitpick-hol.ML*›
ML-file ‹*Tools/Nitpick/nitpick-mono.ML*›
ML-file ‹*Tools/Nitpick/nitpick-preproc.ML*›
ML-file ‹*Tools/Nitpick/nitpick-scope.ML*›
ML-file ‹*Tools/Nitpick/nitpick-peephole.ML*›
ML-file ‹*Tools/Nitpick/nitpick-rep.ML*›
ML-file ‹*Tools/Nitpick/nitpick-nut.ML*›
ML-file ‹*Tools/Nitpick/nitpick-kodkod.ML*›
ML-file ‹*Tools/Nitpick/nitpick-model.ML*›
ML-file ‹*Tools/Nitpick/nitpick.ML*›
ML-file ‹*Tools/Nitpick/nitpick-commands.ML*›
ML-file ‹*Tools/Nitpick/nitpick-tests.ML*›

setup ‹
Nitpick-HOL.register-ersatz-global
[(***const-name*** ‹*card*›, ***const-name*** ‹*card'*›),
(***const-name*** ‹*sum*›, ***const-name*** ‹*sum'*›),
(***const-name*** ‹*fold-graph*›, ***const-name*** ‹*fold-graph'*›),
(***const-abbrev*** ‹*wf*›, ***const-name*** ‹*wf'*›),
(***const-name*** ‹*wf-wfrec*›, ***const-name*** ‹*wf-wfrec'*›),
(***const-name*** ‹*wfrec*›, ***const-name*** ‹*wfrec'*›)]
›

hide-const (**open**) *unknown is-unknown bisim bisim-iterator-max Quot safe-The FunBox PairBox Word prod*
refl′ wf′ card′ sum′ fold-graph′ nat-gcd nat-lcm Frac Abs-Frac Rep-Frac
zero-frac one-frac num denom norm-frac frac plus-frac times-frac uminus-frac number-of-frac
inverse-frac less-frac less-eq-frac of-frac wf-wfrec wf-wfrec wfrec′

hide-type (**open**) *bisim-iterator fun-box pair-box unsigned-bit signed-bit word*

hide-fact (**open**) *Ex1-unfold rtrancl-unfold rtranclp-unfold tranclp-unfold prod-def refl′-def wf′-def*
card′-def sum′-def The-psimp Eps-psimp case-unit-unfold case-nat-unfold
size-list-simp nat-lcm-def Frac-def zero-frac-def one-frac-def
num-def denom-def frac-def plus-frac-def times-frac-def uminus-frac-def
number-of-frac-def inverse-frac-def less-frac-def less-eq-frac-def of-frac-def wf-wfrec′-def
wfrec′-def

end

theory *Nunchaku*
imports *Nitpick*
keywords
nunchaku :: *diag* **and**
nunchaku-params :: *thy-decl*
begin

consts *unreachable* :: *'a*

definition *The-unsafe* :: *('a ⇒ bool) ⇒ 'a* **where**
The-unsafe = The

definition *rmember* :: *'a set ⇒ 'a ⇒ bool* **where**
rmember A x ⟷ x ∈ A

ML-file ‹*Tools/Nunchaku/nunchaku-util.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-collect.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-problem.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-translate.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-model.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-reconstruct.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-display.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-tool.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku.ML*›
ML-file ‹*Tools/Nunchaku/nunchaku-commands.ML*›

hide-const (**open**) *unreachable The-unsafe rmember*

end

89 Greatest Fixpoint (Codatatype) Operation on Bounded Natural Functors

theory *BNF-Greatest-Fixpoint*
imports *BNF-Fixpoint-Base String*
keywords
codatatype :: *thy-defn* **and**
primcorecursive :: *thy-goal-defn* **and**
primcorec :: *thy-defn*
begin

alias *proj* = *Equiv-Relations.proj*

lemma *one-pointE*: $[\![\bigwedge x.\ s = x \Longrightarrow P]\!] \Longrightarrow P$
by *simp*

lemma *obj-sumE*: $[\![\forall x.\ s = Inl\ x \longrightarrow P;\ \forall x.\ s = Inr\ x \longrightarrow P]\!] \Longrightarrow P$
by (*cases s*) *auto*

lemma *not-TrueE*: $\neg\ True \Longrightarrow P$
by (*erule notE*, *rule TrueI*)

lemma *neq-eq-eq-contradict*: $[\![t \neq u;\ s = t;\ s = u]\!] \Longrightarrow P$
by *fast*

lemma *converse-Times*: $(A \times B)^{-1} = B \times A$
by *fast*

lemma *equiv-proj*:
assumes *e*: *equiv A R* **and** *m*: $z \in R$
shows (*proj R* $\circ$ *fst*) *z* = (*proj R* $\circ$ *snd*) *z*
proof −
from *m* **have** *z*: (*fst z*, *snd z*) $\in R$ **by** *auto*
with *e* **have** $\bigwedge x.$ (*fst z*, *x*) $\in R \Longrightarrow$ (*snd z*, *x*) $\in R$ $\bigwedge x.$ (*snd z*, *x*) $\in R \Longrightarrow$ (*fst z*, *x*) $\in R$
unfolding *equiv-def sym-def trans-def* **by** *blast+*
then show *?thesis* **unfolding** *proj-def*[*abs-def*] **by** *auto*
qed

definition *image2* **where** *image2 A f g* = $\{(f\ a,\ g\ a) \mid a.\ a \in A\}$

lemma *Id-on-Gr*: *Id-on A* = *Gr A id*
unfolding *Id-on-def Gr-def* **by** *auto*

lemma *image2-eqI*: $[\![b = f\ x;\ c = g\ x;\ x \in A]\!] \Longrightarrow (b,\ c) \in$ *image2 A f g*

unfolding *image2-def* **by** *auto*

lemma *IdD*: $(a, b) \in Id \Longrightarrow a = b$
by *auto*

lemma *image2-Gr*: *image2 A f g* $= (Gr\ A\ f)^{-1}\ O\ (Gr\ A\ g)$
unfolding *image2-def Gr-def* **by** *auto*

lemma *GrD1*: $(x, fx) \in Gr\ A\ f \Longrightarrow x \in A$
unfolding *Gr-def* **by** *simp*

lemma *GrD2*: $(x, fx) \in Gr\ A\ f \Longrightarrow f\ x = fx$
unfolding *Gr-def* **by** *simp*

lemma *Gr-incl*: $Gr\ A\ f \subseteq A \times B \longleftrightarrow f\ `\ A \subseteq B$
unfolding *Gr-def* **by** *auto*

lemma *subset-Collect-iff*: $B \subseteq A \Longrightarrow (B \subseteq \{x \in A.\ P\ x\}) = (\forall x \in B.\ P\ x)$
by *blast*

lemma *subset-CollectI*: $B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow Q\ x \Longrightarrow P\ x) \Longrightarrow (\{x \in B.\ Q\ x\} \subseteq \{x \in A.\ P\ x\})$
by *blast*

lemma *in-rel-Collect-case-prod-eq*: *in-rel* (*Collect* (*case-prod X*)) $= X$
unfolding *fun-eq-iff* **by** *auto*

lemma *Collect-case-prod-in-rel-leI*: $X \subseteq Y \Longrightarrow X \subseteq$ *Collect* (*case-prod* (*in-rel Y*))
by *auto*

lemma *Collect-case-prod-in-rel-leE*: $X \subseteq$ *Collect* (*case-prod* (*in-rel Y*)) $\Longrightarrow (X \subseteq Y \Longrightarrow R) \Longrightarrow R$
by *force*

lemma *conversep-in-rel*: $(in\text{-}rel\ R)^{-1-1} = in\text{-}rel\ (R^{-1})$
unfolding *fun-eq-iff* **by** *auto*

lemma *relcompp-in-rel*: *in-rel R OO in-rel S* $=$ *in-rel* $(R\ O\ S)$
unfolding *fun-eq-iff* **by** *auto*

lemma *in-rel-Gr*: *in-rel* $(Gr\ A\ f) = Grp\ A\ f$
unfolding *Gr-def Grp-def fun-eq-iff* **by** *auto*

definition *relImage* **where**
$relImage\ R\ f \equiv \{(f\ a1, f\ a2) \mid a1\ a2.\ (a1,a2) \in R\}$

definition *relInvImage* **where**
$relInvImage\ A\ R\ f \equiv \{(a1, a2) \mid a1\ a2.\ a1 \in A \wedge a2 \in A \wedge (f\ a1, f\ a2) \in R\}$

lemma *relImage-Gr*:
  $[\![R \subseteq A \times A]\!] \Longrightarrow relImage\ R\ f = (Gr\ A\ f)^{-1}\ O\ R\ O\ Gr\ A\ f$
  unfolding *relImage-def Gr-def relcomp-def* **by** *auto*

lemma *relInvImage-Gr*: $[\![R \subseteq B \times B]\!] \Longrightarrow relInvImage\ A\ R\ f = Gr\ A\ f\ O\ R\ O\ (Gr\ A\ f)^{-1}$
  unfolding *Gr-def relcomp-def image-def relInvImage-def* **by** *auto*

lemma *relImage-mono*:
  $R1 \subseteq R2 \Longrightarrow relImage\ R1\ f \subseteq relImage\ R2\ f$
  unfolding *relImage-def* **by** *auto*

lemma *relInvImage-mono*:
  $R1 \subseteq R2 \Longrightarrow relInvImage\ A\ R1\ f \subseteq relInvImage\ A\ R2\ f$
  unfolding *relInvImage-def* **by** *auto*

lemma *relInvImage-Id-on*:
  $(\bigwedge a1\ a2.\ f\ a1 = f\ a2 \longleftrightarrow a1 = a2) \Longrightarrow relInvImage\ A\ (Id\text{-}on\ B)\ f \subseteq Id$
  unfolding *relInvImage-def Id-on-def* **by** *auto*

lemma *relInvImage-UNIV-relImage*:
  $R \subseteq relInvImage\ UNIV\ (relImage\ R\ f)\ f$
  unfolding *relInvImage-def relImage-def* **by** *auto*

lemma *relImage-proj*:
  assumes *equiv A R*
  shows *relImage R (proj R)* $\subseteq$ *Id-on (A//R)*
  unfolding *relImage-def Id-on-def*
  using *proj-iff*[*OF assms*] *equiv-class-eq-iff*[*OF assms*]
  by (*auto simp*: *proj-preserves*)

lemma *relImage-relInvImage*:
  assumes $R \subseteq f\ `\ A \times f\ `\ A$
  shows *relImage (relInvImage A R f) f* = *R*
  using *assms* **unfolding** *relImage-def relInvImage-def* **by** *fast*

lemma *subst-Pair*: $P\ x\ y \Longrightarrow a = (x, y) \Longrightarrow P\ (fst\ a)\ (snd\ a)$
  by *simp*

lemma *fst-diag-id*: $(fst \circ (\lambda x.\ (x, x)))\ z = id\ z$ **by** *simp*
lemma *snd-diag-id*: $(snd \circ (\lambda x.\ (x, x)))\ z = id\ z$ **by** *simp*

lemma *fst-diag-fst*: $fst \circ ((\lambda x.\ (x, x)) \circ fst) = fst$ **by** *auto*
lemma *snd-diag-fst*: $snd \circ ((\lambda x.\ (x, x)) \circ fst) = fst$ **by** *auto*
lemma *fst-diag-snd*: $fst \circ ((\lambda x.\ (x, x)) \circ snd) = snd$ **by** *auto*
lemma *snd-diag-snd*: $snd \circ ((\lambda x.\ (x, x)) \circ snd) = snd$ **by** *auto*

definition *Succ* **where** $Succ\ Kl\ kl = \{k\ .\ kl\ @\ [k] \in Kl\}$

definition *Shift* **where** *Shift Kl k* = {*kl*. *k* # *kl* ∈ *Kl*}
definition *shift* **where** *shift lab k* = (λ*kl*. *lab* (*k* # *kl*))

lemma *empty-Shift*: ⟦[] ∈ *Kl*; *k* ∈ *Succ Kl* []⟧ ⟹ [] ∈ *Shift Kl k*
unfolding *Shift-def Succ-def* **by** *simp*

lemma *SuccD*: *k* ∈ *Succ Kl kl* ⟹ *kl* @ [*k*] ∈ *Kl*
unfolding *Succ-def* **by** *simp*

lemmas *SuccE* = *SuccD*[*elim-format*]

lemma *SuccI*: *kl* @ [*k*] ∈ *Kl* ⟹ *k* ∈ *Succ Kl kl*
unfolding *Succ-def* **by** *simp*

lemma *ShiftD*: *kl* ∈ *Shift Kl k* ⟹ *k* # *kl* ∈ *Kl*
unfolding *Shift-def* **by** *simp*

lemma *Succ-Shift*: *Succ* (*Shift Kl k*) *kl* = *Succ Kl* (*k* # *kl*)
unfolding *Succ-def Shift-def* **by** *auto*

lemma *length-Cons*: *length* (*x* # *xs*) = *Suc* (*length xs*)
by *simp*

lemma *length-append-singleton*: *length* (*xs* @ [*x*]) = *Suc* (*length xs*)
by *simp*

definition *toCard-pred A r f* ≡ *inj-on f A* ∧ *f* ‘ *A* ⊆ *Field r* ∧ *Card-order r*
definition *toCard A r* ≡ *SOME f*. *toCard-pred A r f*

lemma *ex-toCard-pred*:
⟦|*A*| ≤*o r*; *Card-order r*⟧ ⟹ ∃ *f*. *toCard-pred A r f*
unfolding *toCard-pred-def*
using *card-of-ordLeq*[*of A Field r*]
ordLeq-ordIso-trans[*OF - card-of-unique*[*of Field r r*], *of* |*A*|]
by *blast*

lemma *toCard-pred-toCard*:
⟦|*A*| ≤*o r*; *Card-order r*⟧ ⟹ *toCard-pred A r* (*toCard A r*)
unfolding *toCard-def* **using** *someI-ex*[*OF ex-toCard-pred*] **.**

lemma *toCard-inj*: ⟦|*A*| ≤*o r*; *Card-order r*; *x* ∈ *A*; *y* ∈ *A*⟧ ⟹ *toCard A r x* = *toCard A r y* ⟷ *x* = *y*
using *toCard-pred-toCard* **unfolding** *inj-on-def toCard-pred-def* **by** *blast*

definition *fromCard A r k* ≡ *SOME b*. *b* ∈ *A* ∧ *toCard A r b* = *k*

lemma *fromCard-toCard*:
⟦|*A*| ≤*o r*; *Card-order r*; *b* ∈ *A*⟧ ⟹ *fromCard A r* (*toCard A r b*) = *b*

unfolding *fromCard-def* **by** (*rule some-equality*) (*auto simp add*: *toCard-inj*)

lemma *Inl-Field-csum*: $a \in Field\ r \Longrightarrow Inl\ a \in Field\ (r +c s)$
unfolding *Field-card-of csum-def* **by** *auto*

lemma *Inr-Field-csum*: $a \in Field\ s \Longrightarrow Inr\ a \in Field\ (r +c s)$
unfolding *Field-card-of csum-def* **by** *auto*

lemma *rec-nat-0-imp*: $f = rec\text{-}nat\ f1\ (\lambda n\ rec.\ f2\ n\ rec) \Longrightarrow f\ 0 = f1$
by *auto*

lemma *rec-nat-Suc-imp*: $f = rec\text{-}nat\ f1\ (\lambda n\ rec.\ f2\ n\ rec) \Longrightarrow f\ (Suc\ n) = f2\ n\ (f\ n)$
by *auto*

lemma *rec-list-Nil-imp*: $f = rec\text{-}list\ f1\ (\lambda x\ xs\ rec.\ f2\ x\ xs\ rec) \Longrightarrow f\ [] = f1$
by *auto*

lemma *rec-list-Cons-imp*: $f = rec\text{-}list\ f1\ (\lambda x\ xs\ rec.\ f2\ x\ xs\ rec) \Longrightarrow f\ (x \#\ xs) = f2\ x\ xs\ (f\ xs)$
by *auto*

lemma *not-arg-cong-Inr*: $x \neq y \Longrightarrow Inr\ x \neq Inr\ y$
by *simp*

definition *image2p* **where**
$image2p\ f\ g\ R = (\lambda x\ y.\ \exists x'\ y'.\ R\ x'\ y' \wedge f\ x' = x \wedge g\ y' = y)$

lemma *image2pI*: $R\ x\ y \Longrightarrow image2p\ f\ g\ R\ (f\ x)\ (g\ y)$
unfolding *image2p-def* **by** *blast*

lemma *image2pE*: $[\![image2p\ f\ g\ R\ fx\ gy; (\bigwedge x\ y.\ fx = f\ x \Longrightarrow gy = g\ y \Longrightarrow R\ x\ y \Longrightarrow P)]\!] \Longrightarrow P$
unfolding *image2p-def* **by** *blast*

lemma *rel-fun-iff-geq-image2p*: $rel\text{-}fun\ R\ S\ f\ g = (image2p\ f\ g\ R \leq S)$
unfolding *rel-fun-def image2p-def* **by** *auto*

lemma *rel-fun-image2p*: $rel\text{-}fun\ R\ (image2p\ f\ g\ R)\ f\ g$
unfolding *rel-fun-def image2p-def* **by** *auto*

89.1 Equivalence relations, quotients, and Hilbert's choice

lemma *equiv-Eps-in*:
$[\![equiv\ A\ r; X \in A//r]\!] \Longrightarrow Eps\ (\lambda x.\ x \in X) \in X$
apply (*rule someI2-ex*)
using *in-quotient-imp-non-empty* **by** *blast*

lemma *equiv-Eps-preserves*:

assumes *ECH*: *equiv A r* **and** *X*: $X \in A//r$
shows *Eps* $(\lambda x.\ x \in X) \in A$
apply (*rule in-mono*[*rule-format*])
using *assms* **apply** (*rule in-quotient-imp-subset*)
by (*rule equiv-Eps-in*) (*rule assms*)+

lemma *proj-Eps*:
assumes *equiv A r* **and** $X \in A//r$
shows *proj r* (*Eps* $(\lambda x.\ x \in X)) = X$
unfolding *proj-def*
proof *auto*
fix *x* **assume** *x*: $x \in X$
thus (*Eps* $(\lambda x.\ x \in X)$, *x*) $\in r$ **using** *assms equiv-Eps-in in-quotient-imp-in-rel*
by *fast*
next
fix *x* **assume** (*Eps* $(\lambda x.\ x \in X)$,*x*) $\in r$
thus $x \in X$ **using** *in-quotient-imp-closed*[*OF assms equiv-Eps-in*[*OF assms*]] **by**
fast
qed

definition *univ* **where** *univ f X* == *f* (*Eps* $(\lambda x.\ x \in X)$)

lemma *univ-commute*:
assumes *ECH*: *equiv A r* **and** *RES*: *f respects r* **and** *x*: $x \in A$
shows (*univ f*) (*proj r x*) = *f x*
proof (*unfold univ-def*)
have *prj*: *proj r x* $\in A//r$ **using** *x proj-preserves* **by** *fast*
hence *Eps* $(\lambda y.\ y \in$ *proj r x*$) \in A$ **using** *ECH equiv-Eps-preserves* **by** *fast*
moreover have *proj r* (*Eps* $(\lambda y.\ y \in$ *proj r x*$))$ = *proj r x* **using** *ECH prj*
proj-Eps **by** *fast*
ultimately have (*x*, *Eps* $(\lambda y.\ y \in$ *proj r x*$)) \in r$ **using** *x ECH proj-iff* **by** *fast*
thus *f* (*Eps* $(\lambda y.\ y \in$ *proj r x*$))$ = *f x* **using** *RES* **unfolding** *congruent-def* **by**
fastforce
qed

lemma *univ-preserves*:
assumes *ECH*: *equiv A r* **and** *RES*: *f respects r* **and** *PRES*: $\forall x \in A.\ f\ x \in B$
shows $\forall X \in A//r.$ *univ f X* $\in B$
proof
fix *X* **assume** $X \in A//r$
then obtain *x* **where** *x*: $x \in A$ **and** *X*: *X* = *proj r x* **using** *ECH proj-image*[*of*
r A] **by** *blast*
hence *univ f X* = *f x* **using** *ECH RES univ-commute* **by** *fastforce*
thus *univ f X* $\in B$ **using** *x PRES* **by** *simp*
qed

lemma *card-suc-ordLess-imp-ordLeq*:
assumes *ORD*: *Card-order r Card-order r′ card-order r′*
and *LESS*: $r <o$ *card-suc r′*

shows $r \leq o\ r'$
proof −
have *Card-order* (*card-suc* r') **by** (*rule Card-order-card-suc*[*OF ORD*(*3*)])
then have *cardSuc* $r \leq o$ *card-suc* r' **using** *cardSuc-least ORD LESS* **by** *blast*
then have *cardSuc* $r \leq o$ *cardSuc* r' **using** *cardSuc-ordIso-card-suc ordIso-symmetric ordLeq-ordIso-trans ORD*(*3*) **by** *blast*
then show *?thesis* **using** *cardSuc-mono-ordLeq ORD* **by** *blast*
qed

lemma *natLeq-ordLess-cinfinite*: ⟦*Cinfinite r*; *card-order r*⟧ ⟹ *natLeq* <*o card-suc r*
using *natLeq-ordLeq-cinfinite card-suc-greater ordLeq-ordLess-trans* **by** *blast*

corollary *natLeq-ordLess-cinfinite'*: ⟦*Cinfinite r'*; *card-order r'*; *r* ≡ *card-suc r'*⟧ ⟹ *natLeq* <*o r*
using *natLeq-ordLess-cinfinite* **by** *blast*

ML-file ‹*Tools/BNF/bnf-gfp-util.ML*›
ML-file ‹*Tools/BNF/bnf-gfp-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-gfp.ML*›
ML-file ‹*Tools/BNF/bnf-gfp-rec-sugar-tactics.ML*›
ML-file ‹*Tools/BNF/bnf-gfp-rec-sugar.ML*›

end

90 Filters on predicates

theory *Filter*
imports *Set-Interval Lifting-Set*
begin

90.1 Filters

This definition also allows non-proper filters.

locale *is-filter* =
fixes *F* :: ('*a* ⇒ *bool*) ⇒ *bool*
assumes *True*: *F* (λ*x*. *True*)
assumes *conj*: *F* (λ*x*. *P x*) ⟹ *F* (λ*x*. *Q x*) ⟹ *F* (λ*x*. *P x* ∧ *Q x*)
assumes *mono*: ∀ *x*. *P x* ⟶ *Q x* ⟹ *F* (λ*x*. *P x*) ⟹ *F* (λ*x*. *Q x*)

typedef '*a filter* = {*F* :: ('*a* ⇒ *bool*) ⇒ *bool*. *is-filter F*}
proof
show (λ*x*. *True*) ∈ *?filter* **by** (*auto intro*: *is-filter.intro*)
qed

lemma *is-filter-Rep-filter*: *is-filter* (*Rep-filter F*)
using *Rep-filter* [*of F*] **by** *simp*

lemma *Abs-filter-inverse'*:
assumes *is-filter F* **shows** *Rep-filter* (*Abs-filter F*) = *F*
using *assms* **by** (*simp add*: *Abs-filter-inverse*)

90.1.1 Eventually

definition *eventually* :: ('*a* ⇒ *bool*) ⇒ '*a filter* ⇒ *bool*
where *eventually P F* ⟷ *Rep-filter F P*

syntax
-eventually :: *pttrn* => '*a filter* => *bool* => *bool* (‹(‹*indent=3 notation=‹binder* ∀ *F*››∀ *F* *- in -./ -*)› [*0*, *0*, *10*] *10*)
syntax-consts
-eventually == *eventually*
translations
∀ *F*x *in F*. *P* == *CONST eventually* (λx. *P*) *F*

lemma *eventually-Abs-filter*:
assumes *is-filter F* **shows** *eventually P* (*Abs-filter F*) = *F P*
unfolding *eventually-def* **using** *assms* **by** (*simp add*: *Abs-filter-inverse*)

lemma *filter-eq-iff*:
shows *F* = *F'* ⟷ (∀ *P*. *eventually P F* = *eventually P F'*)
unfolding *Rep-filter-inject* [*symmetric*] *fun-eq-iff eventually-def* **..**

lemma *eventually-True* [*simp*]: *eventually* (λx. *True*) *F*
unfolding *eventually-def*
by (*rule is-filter.True* [*OF is-filter-Rep-filter*])

lemma *always-eventually*: ∀ *x*. *P x* ⟹ *eventually P F*
proof −
assume ∀ *x*. *P x* **hence** *P* = (λx. *True*) **by** (*simp add*: *ext*)
thus *eventually P F* **by** *simp*
qed

lemma *eventuallyI*: (⋀*x*. *P x*) ⟹ *eventually P F*
by (*auto intro*: *always-eventually*)

lemma *filter-eqI*: (⋀*P*. *eventually P F* ⟷ *eventually P G*) ⟹ *F* = *G*
by (*auto simp*: *filter-eq-iff*)

lemma *eventually-mono*:
⟦*eventually P F*; ⋀*x*. *P x* ⟹ *Q x*⟧ ⟹ *eventually Q F*
unfolding *eventually-def*
by (*blast intro*: *is-filter.mono* [*OF is-filter-Rep-filter*])

lemma *eventually-conj*:
assumes *P*: *eventually* (λx. *P x*) *F*
assumes *Q*: *eventually* (λx. *Q x*) *F*

shows *eventually* (λx. *P x* ∧ *Q x*) *F*
using *assms* **unfolding** *eventually-def*
by (*rule is-filter.conj* [*OF is-filter-Rep-filter*])

lemma *eventually-mp*:
assumes *eventually* (λx. *P x* ⟶ *Q x*) *F*
assumes *eventually* (λx. *P x*) *F*
shows *eventually* (λx. *Q x*) *F*
proof −
have *eventually* (λx. (*P x* ⟶ *Q x*) ∧ *P x*) *F*
using *assms* **by** (*rule eventually-conj*)
then show *?thesis*
by (*blast intro*: *eventually-mono*)
qed

lemma *eventually-rev-mp*:
assumes *eventually* (λx. *P x*) *F*
assumes *eventually* (λx. *P x* ⟶ *Q x*) *F*
shows *eventually* (λx. *Q x*) *F*
using *assms*(*2*) *assms*(*1*) **by** (*rule eventually-mp*)

lemma *eventually-conj-iff*:
eventually (λx. *P x* ∧ *Q x*) *F* ⟷ *eventually P F* ∧ *eventually Q F*
by (*auto intro*: *eventually-conj elim*: *eventually-rev-mp*)

lemma *eventually-elim2*:
assumes *eventually* (λi. *P i*) *F*
assumes *eventually* (λi. *Q i*) *F*
assumes ⋀i. *P i* ⟹ *Q i* ⟹ *R i*
shows *eventually* (λi. *R i*) *F*
using *assms* **by** (*auto elim*!: *eventually-rev-mp*)

lemma *eventually-cong*:
assumes *eventually P F* **and** ⋀x. *P x* ⟹ *Q x* ⟷ *R x*
shows *eventually Q F* ⟷ *eventually R F*
using *assms eventually-elim2* **by** *blast*

lemma *eventually-ball-finite-distrib*:
finite A ⟹ (*eventually* (λx. ∀ y∈*A*. *P x y*) *net*) ⟷ (∀ y∈*A*. *eventually* (λx. *P x y*) *net*)
by (*induction A rule*: *finite-induct*) (*auto simp*: *eventually-conj-iff*)

lemma *eventually-ball-finite*:
finite A ⟹ ∀ y∈*A*. *eventually* (λx. *P x y*) *net* ⟹ *eventually* (λx. ∀ y∈*A*. *P x y*) *net*
by (*auto simp*: *eventually-ball-finite-distrib*)

lemma *eventually-all-finite*:
fixes *P* :: *'a* ⇒ *'b*::*finite* ⇒ *bool*

assumes $\bigwedge y$. *eventually* $(\lambda x.\ P\ x\ y)$ *net*
shows *eventually* $(\lambda x.\ \forall\, y.\ P\ x\ y)$ *net*
using *eventually-ball-finite* [*of UNIV P*] *assms* **by** *simp*

lemma *eventually-ex*: $(\forall_F x$ *in* $F.\ \exists\, y.\ P\ x\ y) \longleftrightarrow (\exists\, Y.\ \forall_F x$ *in* $F.\ P\ x\ (Y\ x))$
proof
assume $\forall_F x$ *in* $F.\ \exists\, y.\ P\ x\ y$
then have $\forall_F x$ *in* $F.\ P\ x$ (*SOME y. P x y*)
by (*auto intro*: *someI-ex eventually-mono*)
then show $\exists\, Y.\ \forall_F x$ *in* $F.\ P\ x\ (Y\ x)$
by *auto*
qed (*auto intro*: *eventually-mono*)

lemma *not-eventually-impI*: *eventually P F* $\Longrightarrow \neg$ *eventually Q F* $\Longrightarrow \neg$ *eventually* $(\lambda x.\ P\ x \longrightarrow Q\ x)\ F$
by (*auto intro*: *eventually-mp*)

lemma *not-eventuallyD*: $\neg$ *eventually P F* $\Longrightarrow \exists\, x.\ \neg\ P\ x$
by (*metis always-eventually*)

lemma *eventually-subst*:
assumes *eventually* $(\lambda n.\ P\ n = Q\ n)\ F$
shows *eventually P F* = *eventually Q F* (**is** *?L* = *?R*)
proof −
from *assms* **have** *eventually* $(\lambda x.\ P\ x \longrightarrow Q\ x)\ F$
and *eventually* $(\lambda x.\ Q\ x \longrightarrow P\ x)\ F$
by (*auto elim*: *eventually-mono*)
then show *?thesis* **by** (*auto elim*: *eventually-elim2*)
qed

90.2 Frequently as dual to eventually

definition *frequently* :: $('a \Rightarrow bool) \Rightarrow$ *'a filter* $\Rightarrow$ *bool*
where *frequently P F* $\longleftrightarrow \neg$ *eventually* $(\lambda x.\ \neg\ P\ x)\ F$

syntax
-frequently :: *pttrn* $\Rightarrow$ *'a filter* $\Rightarrow$ *bool* $\Rightarrow$ *bool* (‹(‹*indent=3 notation=‹binder* $\exists_F$››$\exists_F$ *- in -./ -*)› [*0*, *0*, *10*] *10*)
syntax-consts
-frequently == *frequently*
translations
$\exists_F x$ *in* $F.\ P$ == *CONST frequently* $(\lambda x.\ P)\ F$

lemma *not-frequently-False* [*simp*]: $\neg\ (\exists_F x$ *in* $F.$ *False*)
by (*simp add*: *frequently-def*)

lemma *frequently-ex*: $\exists_F x$ *in* $F.\ P\ x \Longrightarrow \exists\, x.\ P\ x$
by (*auto simp*: *frequently-def dest*: *not-eventuallyD*)

lemma *frequentlyE*: **assumes** *frequently P F* **obtains** *x* **where** *P x*
using *frequently-ex*[*OF assms*] **by** *auto*

lemma *frequently-mp*:
assumes *ev*: $\forall_F x$ *in F*. *P x* $\longrightarrow$ *Q x* **and** *P*: $\exists_F x$ *in F*. *P x* **shows** $\exists_F x$ *in F*. *Q x*
proof −
from *ev* **have** *eventually* (λx. $\neg$ *Q x* $\longrightarrow$ $\neg$ *P x*) *F*
by (*rule eventually-rev-mp*) (*auto intro*!: *always-eventually*)
from *eventually-mp*[*OF this*] *P* **show** *?thesis*
by (*auto simp*: *frequently-def*)
qed

lemma *frequently-rev-mp*:
assumes $\exists_F x$ *in F*. *P x*
assumes $\forall_F x$ *in F*. *P x* $\longrightarrow$ *Q x*
shows $\exists_F x$ *in F*. *Q x*
using *assms*(*2*) *assms*(*1*) **by** (*rule frequently-mp*)

lemma *frequently-mono*: ($\forall x$. *P x* $\longrightarrow$ *Q x*) $\Longrightarrow$ *frequently P F* $\Longrightarrow$ *frequently Q F*
using *frequently-mp*[*of P Q*] **by** (*simp add*: *always-eventually*)

lemma *frequently-elim1*: $\exists_F x$ *in F*. *P x* $\Longrightarrow$ ($\bigwedge i$. *P i* $\Longrightarrow$ *Q i*) $\Longrightarrow$ $\exists_F x$ *in F*. *Q x*
by (*metis frequently-mono*)

lemma *frequently-disj-iff*: ($\exists_F x$ *in F*. *P x* $\vee$ *Q x*) $\longleftrightarrow$ ($\exists_F x$ *in F*. *P x*) $\vee$ ($\exists_F x$ *in F*. *Q x*)
by (*simp add*: *frequently-def eventually-conj-iff*)

lemma *frequently-disj*: $\exists_F x$ *in F*. *P x* $\Longrightarrow$ $\exists_F x$ *in F*. *Q x* $\Longrightarrow$ $\exists_F x$ *in F*. *P x* $\vee$ *Q x*
by (*simp add*: *frequently-disj-iff*)

lemma *frequently-bex-finite-distrib*:
assumes *finite A* **shows** ($\exists_F x$ *in F*. $\exists$ *y*$\in$*A*. *P x y*) $\longleftrightarrow$ ($\exists$ *y*$\in$*A*. $\exists_F x$ *in F*. *P x y*)
using *assms* **by** *induction* (*auto simp*: *frequently-disj-iff*)

lemma *frequently-bex-finite*: *finite A* $\Longrightarrow$ $\exists_F x$ *in F*. $\exists$ *y*$\in$*A*. *P x y* $\Longrightarrow$ $\exists$ *y*$\in$*A*. $\exists_F x$ *in F*. *P x y*
by (*simp add*: *frequently-bex-finite-distrib*)

lemma *frequently-all*: ($\exists_F x$ *in F*. $\forall y$. *P x y*) $\longleftrightarrow$ ($\forall$ *Y*. $\exists_F x$ *in F*. *P x* (*Y x*))
using *eventually-ex*[*of* λx *y*. $\neg$ *P x y F*] **by** (*simp add*: *frequently-def*)

lemma
shows *not-eventually*: $\neg$ *eventually P F* $\longleftrightarrow$ ($\exists_F x$ *in F*. $\neg$ *P x*)

and *not-frequently*: $\neg$ *frequently P F* $\longleftrightarrow$ $(\forall_F x$ *in F*. $\neg$ *P x*$)$
by (*auto simp*: *frequently-def*)

lemma *frequently-imp-iff*:
$(\exists_F x$ *in F*. *P x* $\longrightarrow$ *Q x*$)$ $\longleftrightarrow$ (*eventually P F* $\longrightarrow$ *frequently Q F*)
unfolding *imp-conv-disj frequently-disj-iff not-eventually*[*symmetric*] **..**

lemma *frequently-eventually-conj*:
assumes $\exists_F x$ *in F*. *P x*
assumes $\forall_F x$ *in F*. *Q x*
shows $\exists_F x$ *in F*. *Q x* $\wedge$ *P x*
using *assms eventually-elim2* **by** (*force simp add*: *frequently-def*)

lemma *frequently-cong*:
assumes *ev*: *eventually P F* **and** *QR*: $\bigwedge x$. *P x* $\Longrightarrow$ *Q x* $\longleftrightarrow$ *R x*
shows *frequently Q F* $\longleftrightarrow$ *frequently R F*
unfolding *frequently-def*
using *QR* **by** (*auto intro*!: *eventually-cong* [*OF ev*])

lemma *frequently-eventually-frequently*:
frequently P F $\Longrightarrow$ *eventually Q F* $\Longrightarrow$ *frequently* $(\lambda x$. *P x* $\wedge$ *Q x*$)$ *F*
using *frequently-cong* [*of Q F P* λx. *P x* $\wedge$ *Q x*] **by** *meson*

lemma *eventually-frequently-const-simps* [*simp*]:
$(\exists_F x$ *in F*. *P x* $\wedge$ *C*$)$ $\longleftrightarrow$ $(\exists_F x$ *in F*. *P x*$)$ $\wedge$ *C*
$(\exists_F x$ *in F*. *C* $\wedge$ *P x*$)$ $\longleftrightarrow$ *C* $\wedge$ $(\exists_F x$ *in F*. *P x*$)$
$(\forall_F x$ *in F*. *P x* $\vee$ *C*$)$ $\longleftrightarrow$ $(\forall_F x$ *in F*. *P x*$)$ $\vee$ *C*
$(\forall_F x$ *in F*. *C* $\vee$ *P x*$)$ $\longleftrightarrow$ *C* $\vee$ $(\forall_F x$ *in F*. *P x*$)$
$(\forall_F x$ *in F*. *P x* $\longrightarrow$ *C*$)$ $\longleftrightarrow$ $((\exists_F x$ *in F*. *P x*$)$ $\longrightarrow$ *C*$)$
$(\forall_F x$ *in F*. *C* $\longrightarrow$ *P x*$)$ $\longleftrightarrow$ $(C \longrightarrow (\forall_F x$ *in F*. *P x*$))$
by (*cases C*; *simp add*: *not-frequently*)+

lemmas *eventually-frequently-simps* =
eventually-frequently-const-simps
not-eventually
eventually-conj-iff
eventually-ball-finite-distrib
eventually-ex
not-frequently
frequently-disj-iff
frequently-bex-finite-distrib
frequently-all
frequently-imp-iff

ML ‹
fun eventually-elim-tac facts =
CONTEXT-SUBGOAL (*fn* (*goal*, *i*) => *fn* (*ctxt*, *st*) =>
let
val mp-facts = *facts RL* @{*thms eventually-rev-mp*}

val rule =
@{thm eventuallyI}
|> fold (fn mp-fact => fn th => th RS mp-fact) mp-facts
|> funpow (length facts) (fn th => @{thm impI} RS th)
val cases-prop =
Thm.prop-of (Rule-Cases.internalize-params (rule RS Goal.init (Thm.cterm-of ctxt goal)))
val cases = Rule-Cases.make-common ctxt cases-prop [((elim, []), [])]
in CONTEXT-CASES cases (resolve-tac ctxt [rule] i) (ctxt, st) end
›

method-setup *eventually-elim* = ‹
Scan.succeed (fn - => CONTEXT-METHOD (fn facts => eventually-elim-tac facts 1))
› *elimination of eventually quantifiers*

90.2.1 Finer-than relation

$F \leq F'$ means that filter F is finer than filter F'.

instantiation *filter* :: (*type*) *complete-lattice*
begin

definition *le-filter-def*:
$F \leq F' \longleftrightarrow (\forall P.\ \mathit{eventually}\ P\ F' \longrightarrow \mathit{eventually}\ P\ F)$

definition
$(F :: 'a\ \mathit{filter}) < F' \longleftrightarrow F \leq F' \wedge \neg\ F' \leq F$

definition
$\mathit{top} = \mathit{Abs\text{-}filter}\ (\lambda P.\ \forall x.\ P\ x)$

definition
$\mathit{bot} = \mathit{Abs\text{-}filter}\ (\lambda P.\ \mathit{True})$

definition
$\mathit{sup}\ F\ F' = \mathit{Abs\text{-}filter}\ (\lambda P.\ \mathit{eventually}\ P\ F \wedge \mathit{eventually}\ P\ F')$

definition
$\mathit{inf}\ F\ F' = \mathit{Abs\text{-}filter}$
$(\lambda P.\ \exists Q\ R.\ \mathit{eventually}\ Q\ F \wedge \mathit{eventually}\ R\ F' \wedge (\forall x.\ Q\ x \wedge R\ x \longrightarrow P\ x))$

definition
$\mathit{Sup}\ S = \mathit{Abs\text{-}filter}\ (\lambda P.\ \forall F{\in}S.\ \mathit{eventually}\ P\ F)$

definition
$\mathit{Inf}\ S = \mathit{Sup}\ \{F::'a\ \mathit{filter}.\ \forall F'{\in}S.\ F \leq F'\}$

lemma *eventually-top* [*simp*]: $\mathit{eventually}\ P\ \mathit{top} \longleftrightarrow (\forall x.\ P\ x)$
unfolding *top-filter-def*

by (*rule eventually-Abs-filter*, *rule is-filter.intro*, *auto*)

lemma *eventually-bot* [*simp*]: *eventually P bot*
unfolding *bot-filter-def*
by (*subst eventually-Abs-filter*, *rule is-filter.intro*, *auto*)

lemma *eventually-sup*:
eventually P (*sup F F'*) ⟷ *eventually P F* ∧ *eventually P F'*
unfolding *sup-filter-def*
by (*rule eventually-Abs-filter*, *rule is-filter.intro*)
(*auto elim*!: *eventually-rev-mp*)

lemma *eventually-inf*:
eventually P (*inf F F'*) ⟷
(∃ *Q R*. *eventually Q F* ∧ *eventually R F'* ∧ (∀ *x*. *Q x* ∧ *R x* ⟶ *P x*))
unfolding *inf-filter-def*
apply (*rule eventually-Abs-filter* [*OF is-filter.intro*])
apply (*blast intro*: *eventually-True*)
apply (*force elim*!: *eventually-conj*)+
done

lemma *eventually-Sup*:
eventually P (*Sup S*) ⟷ (∀ *F*∈*S*. *eventually P F*)
unfolding *Sup-filter-def*
apply (*rule eventually-Abs-filter* [*OF is-filter.intro*])
apply (*auto intro*: *eventually-conj elim*!: *eventually-rev-mp*)
done

instance proof
fix *F F' F''* :: *'a filter* **and** *S* :: *'a filter set*
{ show *F* < *F'* ⟷ *F* ≤ *F'* ∧ ¬ *F'* ≤ *F*
by (*rule less-filter-def*) **}**
{ show *F* ≤ *F*
unfolding *le-filter-def* **by** *simp* **}**
{ assume *F* ≤ *F'* **and** *F'* ≤ *F''* **thus** *F* ≤ *F''*
unfolding *le-filter-def* **by** *simp* **}**
{ assume *F* ≤ *F'* **and** *F'* ≤ *F* **thus** *F* = *F'*
unfolding *le-filter-def filter-eq-iff* **by** *fast* **}**
{ show *inf F F'* ≤ *F* **and** *inf F F'* ≤ *F'*
unfolding *le-filter-def eventually-inf* **by** (*auto intro*: *eventually-True*) **}**
{ assume *F* ≤ *F'* **and** *F* ≤ *F''* **thus** *F* ≤ *inf F' F''*
unfolding *le-filter-def eventually-inf*
by (*auto intro*: *eventually-mono* [*OF eventually-conj*]) **}**
{ show *F* ≤ *sup F F'* **and** *F'* ≤ *sup F F'*
unfolding *le-filter-def eventually-sup* **by** *simp-all* **}**
{ assume *F* ≤ *F''* **and** *F'* ≤ *F''* **thus** *sup F F'* ≤ *F''*
unfolding *le-filter-def eventually-sup* **by** *simp* **}**
{ assume *F''* ∈ *S* **thus** *Inf S* ≤ *F''*
unfolding *le-filter-def Inf-filter-def eventually-Sup Ball-def* **by** *simp* **}**

```
  { assume ⋀F'. F' ∈ S ⟹ F ≤ F' thus F ≤ Inf S
    unfolding le-filter-def Inf-filter-def eventually-Sup Ball-def by simp }
  { assume F ∈ S thus F ≤ Sup S
    unfolding le-filter-def eventually-Sup by simp }
  { assume ⋀F. F ∈ S ⟹ F ≤ F' thus Sup S ≤ F'
    unfolding le-filter-def eventually-Sup by simp }
  { show Inf {} = (top::'a filter)
    by (auto simp: top-filter-def Inf-filter-def Sup-filter-def)
      (metis (full-types) top-filter-def always-eventually eventually-top) }
  { show Sup {} = (bot::'a filter)
    by (auto simp: bot-filter-def Sup-filter-def) }
qed

end

instance filter :: (type) distrib-lattice
proof
  fix F G H :: 'a filter
  show sup F (inf G H) = inf (sup F G) (sup F H)
  proof (rule order.antisym)
    show inf (sup F G) (sup F H) ≤ sup F (inf G H)
      unfolding le-filter-def eventually-sup
    proof safe
      fix P assume 1: eventually P F and 2: eventually P (inf G H)
      from 2 obtain Q R
        where QR: eventually Q G eventually R H ⋀x. Q x ⟹ R x ⟹ P x
        by (auto simp: eventually-inf)
      define Q' where Q' = (λx. Q x ∨ P x)
      define R' where R' = (λx. R x ∨ P x)
      from 1 have eventually Q' F
        by (elim eventually-mono) (auto simp: Q'-def)
      moreover from 1 have eventually R' F
        by (elim eventually-mono) (auto simp: R'-def)
      moreover from QR(1) have eventually Q' G
        by (elim eventually-mono) (auto simp: Q'-def)
      moreover from QR(2) have eventually R' H
        by (elim eventually-mono)(auto simp: R'-def)
      moreover from QR have P x if Q' x R' x for x
        using that by (auto simp: Q'-def R'-def)
      ultimately show eventually P (inf (sup F G) (sup F H))
        by (auto simp: eventually-inf eventually-sup)
    qed
  qed (auto intro: inf.coboundedI1 inf.coboundedI2)
qed


lemma filter-leI:
  (⋀P. eventually P F' ⟹ eventually P F) ⟹ F ≤ F'
  unfolding le-filter-def by simp
```

lemma *filter-leD*:
  $F \leq F' \Longrightarrow$ *eventually* $P\ F' \Longrightarrow$ *eventually* $P\ F$
  unfolding *le-filter-def* **by** *simp*

lemma *frequently-mono-filter*: *frequently* $P\ F \Longrightarrow F \leq F' \Longrightarrow$ *frequently* $P\ F'$
  using *filter-leD*[*of* $F\ F'\ \lambda x.\ \neg P\ x$] **by** (*auto simp*: *frequently-def*)

lemma *eventually-False*:
  eventually $(\lambda x.\ False)\ F \longleftrightarrow F = bot$
  unfolding *filter-eq-iff* **by** (*auto elim*: *eventually-rev-mp*)

lemma *eventually-frequently*: $F \neq bot \Longrightarrow$ *eventually* $P\ F \Longrightarrow$ *frequently* $P\ F$
  using *eventually-conj*[*of* $P\ F\ \lambda x.\ \neg\ P\ x$]
  by (*auto simp add*: *frequently-def eventually-False*)

lemma *eventually-frequentlyE*:
  assumes *eventually* $P\ F$
  assumes *eventually* $(\lambda x.\ \neg\ P\ x \vee Q\ x)\ F\ F{\neq}bot$
  shows *frequently* $Q\ F$
proof −
  have *eventually* $Q\ F$
   using *eventually-conj*[*OF assms*(*1*,*2*),*simplified*] **by** (*auto elim*:*eventually-mono*)
  then show *?thesis* **using** *eventually-frequently*[*OF* ‹$F{\neq}bot$›] **by** *auto*
qed

lemma *eventually-const-iff*: *eventually* $(\lambda x.\ P)\ F \longleftrightarrow P \vee F = bot$
  by (*cases* P) (*auto simp*: *eventually-False*)

lemma *eventually-const*[*simp*]: $F \neq bot \Longrightarrow$ *eventually* $(\lambda x.\ P)\ F \longleftrightarrow P$
  by (*simp add*: *eventually-const-iff*)

lemma *frequently-const-iff*: *frequently* $(\lambda x.\ P)\ F \longleftrightarrow P \wedge F \neq bot$
  by (*simp add*: *frequently-def eventually-const-iff*)

lemma *frequently-const*[*simp*]: $F \neq bot \Longrightarrow$ *frequently* $(\lambda x.\ P)\ F \longleftrightarrow P$
  by (*simp add*: *frequently-const-iff*)

lemma *eventually-happens*: *eventually* P *net* $\Longrightarrow$ *net* $= bot \vee (\exists\, x.\ P\ x)$
  by (*metis frequentlyE eventually-frequently*)

lemma *eventually-happens'*:
  assumes $F \neq bot$ *eventually* $P\ F$
  shows $\exists\, x.\ P\ x$
  using *assms eventually-frequently frequentlyE* **by** *blast*

abbreviation (*input*) *trivial-limit* :: *'a filter* ⇒ *bool*
  where *trivial-limit* $F \equiv F = bot$

lemma *trivial-limit-def*: *trivial-limit F* $\longleftrightarrow$ *eventually* (λx. *False*) *F*
by (*rule eventually-False* [*symmetric*])

lemma *False-imp-not-eventually*: ($\forall\, x.\ \neg\ P\ x\ ) \Longrightarrow \neg$ *trivial-limit net* $\Longrightarrow \neg$ *eventually* (λx. *P x*) *net*
by (*simp add*: *eventually-False*)

lemma *trivial-limit-eventually*: *trivial-limit net* $\Longrightarrow$ *eventually P net*
by *simp*

lemma *trivial-limit-eq*: *trivial-limit net* $\longleftrightarrow$ ($\forall\, P.$ *eventually P net*)
by (*simp add*: *filter-eq-iff*)

lemma *eventually-Inf*: *eventually P* (*Inf B*) $\longleftrightarrow$ ($\exists\, X{\subseteq}B.$ *finite X* $\wedge$ *eventually P* (*Inf X*))
proof −
let *?F* = $\lambda P.\ \exists\, X{\subseteq}B.$ *finite X* $\wedge$ *eventually P* (*Inf X*)

have *eventually-F*: *eventually P* (*Abs-filter ?F*) $\longleftrightarrow$ *?F P* **for** *P*
proof (*rule eventually-Abs-filter is-filter.intro*)+
show *?F* (λx. *True*)
by (*rule exI*[*of* - {}]) (*simp add*: *le-fun-def*)
next
fix *P Q*
assume *?F P ?F Q*
then obtain *X Y* **where**
$X \subseteq B$ *finite X eventually P* ($\bigsqcap$ *X*)
$Y \subseteq B$ *finite Y eventually Q* ($\bigsqcap$ *Y*) **by** *blast*
then show *?F* ($\lambda x.\ P\ x \wedge Q\ x$)
by (*intro exI*[*of* - $X \cup Y$]) (*auto simp*: *Inf-union-distrib eventually-inf*)
next
fix *P Q*
assume *?F P*
then obtain *X* **where** $X \subseteq B$ *finite X eventually P* ($\bigsqcap$ *X*)
by *blast*
moreover assume $\forall\, x.\ P\ x \longrightarrow Q\ x$
ultimately show *?F Q*
by (*intro exI*[*of* - *X*]) (*auto elim*: *eventually-mono*)
qed

have *Inf B* = *Abs-filter ?F*
proof (*intro antisym Inf-greatest*)
show *Inf B* $\leq$ *Abs-filter ?F*
by (*auto simp*: *le-filter-def eventually-F dest*: *Inf-superset-mono*)
next
fix *F* **assume** $F \in B$ **then show** *Abs-filter ?F* $\leq F$
by (*auto simp add*: *le-filter-def eventually-F intro*!: *exI*[*of* - {*F*}])
qed
then show *?thesis*

by (*simp add*: *eventually-F*)
qed

lemma *eventually-INF*: *eventually P* (⨅ *b*∈*B*. *F b*) ⟷ (∃ *X*⊆*B*. *finite X* ∧ *eventually P* (⨅ *b*∈*X*. *F b*))
unfolding *eventually-Inf* [*of P F*‘*B*]
by (*metis finite-imageI image-mono finite-subset-image*)

lemma *Inf-filter-not-bot*:
fixes *B* :: *'a filter set*
shows (⋀*X*. *X* ⊆ *B* ⟹ *finite X* ⟹ *Inf X* ≠ *bot*) ⟹ *Inf B* ≠ *bot*
unfolding *trivial-limit-def eventually-Inf*[*of* - *B*]
bot-bool-def [*symmetric*] *bot-fun-def* [*symmetric*] *bot-unique* **by** *simp*

lemma *INF-filter-not-bot*:
fixes *F* :: *'i* ⇒ *'a filter*
shows (⋀*X*. *X* ⊆ *B* ⟹ *finite X* ⟹ (⨅ *b*∈*X*. *F b*) ≠ *bot*) ⟹ (⨅ *b*∈*B*. *F b*) ≠ *bot*
unfolding *trivial-limit-def eventually-INF* [*of* - - *B*]
bot-bool-def [*symmetric*] *bot-fun-def* [*symmetric*] *bot-unique* **by** *simp*

lemma *eventually-Inf-base*:
assumes *B* ≠ {} **and** *base*: ⋀*F G*. *F* ∈ *B* ⟹ *G* ∈ *B* ⟹ ∃ *x*∈*B*. *x* ≤ *inf F G*
shows *eventually P* (*Inf B*) ⟷ (∃ *b*∈*B*. *eventually P b*)
proof (*subst eventually-Inf*, *safe*)
fix *X* **assume** *finite X X* ⊆ *B*
then have ∃ *b*∈*B*. ∀ *x*∈*X*. *b* ≤ *x*
proof *induct*
case *empty* **then show** *?case*
using ‹*B* ≠ {}› **by** *auto*
next
case (*insert x X*)
then obtain *b* **where** *b* ∈ *B* ⋀*x*. *x* ∈ *X* ⟹ *b* ≤ *x*
by *auto*
with ‹*insert x X* ⊆ *B*› *base*[*of b x*] **show** *?case*
by (*auto intro*: *order-trans*)
qed
then obtain *b* **where** *b* ∈ *B b* ≤ *Inf X*
by (*auto simp*: *le-Inf-iff*)
then show *eventually P* (*Inf X*) ⟹ *Bex B* (*eventually P*)
by (*intro bexI*[*of* - *b*]) (*auto simp*: *le-filter-def*)
qed (*auto intro*!: *exI*[*of* - {*x*} **for** *x*])

lemma *eventually-INF-base*:
B ≠ {} ⟹ (⋀*a b*. *a* ∈ *B* ⟹ *b* ∈ *B* ⟹ ∃ *x*∈*B*. *F x* ≤ *inf* (*F a*) (*F b*)) ⟹
eventually P (⨅ *b*∈*B*. *F b*) ⟷ (∃ *b*∈*B*. *eventually P* (*F b*))
by (*subst eventually-Inf-base*) *auto*

lemma *eventually-INF1*: *i* ∈ *I* ⟹ *eventually P* (*F i*) ⟹ *eventually P* (⨅ *i*∈*I*.

F i)
using *filter-leD*[*OF INF-lower*] **.**

lemma *eventually-INF-finite*:
assumes *finite A*
shows *eventually P* (⨅ *x*∈*A*. *F x*) ⟷
(∃ *Q*. (∀ *x*∈*A*. *eventually* (*Q x*) (*F x*)) ∧ (∀ *y*. (∀ *x*∈*A*. *Q x y*) ⟶ *P y*))
using *assms*
proof (*induction arbitrary*: *P rule*: *finite-induct*)
case (*insert a A P*)
from *insert.hyps* **have** [*simp*]: *x* ≠ *a* **if** *x* ∈ *A* **for** *x*
using *that* **by** *auto*
have *eventually P* (⨅ *x*∈*insert a A*. *F x*) ⟷
(∃ *Q R S*. *eventually Q* (*F a*) ∧ (((∀ *x*∈*A*. *eventually* (*S x*) (*F x*)) ∧
(∀ *y*. (∀ *x*∈*A*. *S x y*) ⟶ *R y*)) ∧ (∀ *x*. *Q x* ∧ *R x* ⟶ *P x*)))
unfolding *ex-simps* **by** (*simp add*: *eventually-inf insert.IH*)
also have ... ⟷ (∃ *Q*. (∀ *x*∈*insert a A*. *eventually* (*Q x*) (*F x*)) ∧
(∀ *y*. (∀ *x*∈*insert a A*. *Q x y*) ⟶ *P y*))
proof (*safe*, *goal-cases*)
case (*1 Q R S*)
thus *?case* **using** *1* **by** (*intro exI*[*of* - *S*(*a* := *Q*)]) *auto*
next
case (*2 Q*)
show *?case*
by (*rule exI*[*of* - *Q a*], *rule exI*[*of* - λ*y*. ∀ *x*∈*A*. *Q x y*],
rule exI[*of* - *Q*(*a* := (λ-. *True*))]) (*use 2* **in** *auto*)
qed
finally show *?case* **.**
qed *auto*

lemma *eventually-le-le*:
fixes *P* :: *'a* ⇒ (*'b* :: *preorder*)
assumes *eventually* (λ*x*. *P x* ≤ *Q x*) *F*
assumes *eventually* (λ*x*. *Q x* ≤ *R x*) *F*
shows *eventually* (λ*x*. *P x* ≤ *R x*) *F*
using *assms* **by** *eventually-elim* (*rule order-trans*)

90.2.2 Map function for filters

definition *filtermap* :: (*'a* ⇒ *'b*) ⇒ *'a filter* ⇒ *'b filter*
where *filtermap f F* = *Abs-filter* (λ*P*. *eventually* (λ*x*. *P* (*f x*)) *F*)

lemma *eventually-filtermap*:
eventually P (*filtermap f F*) = *eventually* (λ*x*. *P* (*f x*)) *F*
unfolding *filtermap-def*
apply (*rule eventually-Abs-filter* [*OF is-filter.intro*])
apply (*auto elim*!: *eventually-rev-mp*)
done

lemma *eventually-comp-filtermap*:
eventually $(P \circ f)$ $F \longleftrightarrow$ *eventually* P (*filtermap* f F)
unfolding *comp-def* **using** *eventually-filtermap* **by** *auto*

lemma *filtermap-compose*: *filtermap* $(f \circ g)$ F = *filtermap* f (*filtermap* g F)
unfolding *filter-eq-iff* **by** (*simp add*: *eventually-filtermap*)

lemma *filtermap-ident*: *filtermap* $(\lambda x.\ x)$ $F = F$
by (*simp add*: *filter-eq-iff eventually-filtermap*)

lemma *filtermap-filtermap*:
filtermap f (*filtermap* g F) = *filtermap* $(\lambda x.\ f\ (g\ x))$ F
by (*simp add*: *filter-eq-iff eventually-filtermap*)

lemma *filtermap-mono*: $F \le F' \Longrightarrow$ *filtermap* f $F \le$ *filtermap* f F'
unfolding *le-filter-def eventually-filtermap* **by** *simp*

lemma *filtermap-bot* [*simp*]: *filtermap* f *bot* = *bot*
by (*simp add*: *filter-eq-iff eventually-filtermap*)

lemma *filtermap-bot-iff*: *filtermap* f F = *bot* $\longleftrightarrow$ F = *bot*
by (*simp add*: *trivial-limit-def eventually-filtermap*)

lemma *filtermap-sup*: *filtermap* f (*sup F1 F2*) = *sup* (*filtermap* f *F1*) (*filtermap* f *F2*)
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-sup*)

lemma *filtermap-SUP*: *filtermap* f $(\bigsqcup b{\in}B.\ F\ b)$ = $(\bigsqcup b{\in}B.$ *filtermap* f $(F\ b))$
by (*simp add*: *filter-eq-iff eventually-Sup eventually-filtermap*)

lemma *filtermap-inf*: *filtermap* f (*inf F1 F2*) $\le$ *inf* (*filtermap* f *F1*) (*filtermap* f *F2*)
by (*intro inf-greatest filtermap-mono inf-sup-ord*)

lemma *filtermap-INF*: *filtermap* f $(\bigsqcap b{\in}B.\ F\ b) \le (\bigsqcap b{\in}B.$ *filtermap* f $(F\ b))$
by (*rule INF-greatest*, *rule filtermap-mono*, *erule INF-lower*)

lemma *frequently-filtermap*:
frequently P (*filtermap* f F) = *frequently* $(\lambda x.\ P\ (f\ x))$ F
by (*simp add*: *frequently-def eventually-filtermap*)

90.2.3 Contravariant map function for filters

definition *filtercomap* :: $('a \Rightarrow 'b) \Rightarrow 'b$ *filter* $\Rightarrow 'a$ *filter* **where**
filtercomap f F = *Abs-filter* $(\lambda P.\ \exists Q.$ *eventually* Q $F \wedge (\forall x.\ Q\ (f\ x) \longrightarrow P\ x))$

lemma *eventually-filtercomap*:
eventually P (*filtercomap* f F) $\longleftrightarrow$ $(\exists Q.$ *eventually* Q $F \wedge (\forall x.\ Q\ (f\ x) \longrightarrow P\ x))$

unfolding *filtercomap-def*
proof (*intro eventually-Abs-filter*, *unfold-locales*, *goal-cases*)
case *1*
show *?case* **by** (*auto intro*!: *exI*[*of* - λ-. *True*])
next
case (*2 P Q*)
then obtain *P' Q'* **where** *P'Q'*:
eventually P' F ∀*x*. *P'* (*f x*) ⟶ *P x*
eventually Q' F ∀*x*. *Q'* (*f x*) ⟶ *Q x*
by (*elim exE conjE*)
show *?case*
by (*rule exI*[*of* - λ*x*. *P' x* ∧ *Q' x*]) (*use P'Q'* **in** ‹*auto intro*!: *eventually-conj*›)
next
case (*3 P Q*)
thus *?case* **by** *blast*
qed

lemma *filtercomap-ident*: *filtercomap* (λ*x*. *x*) *F* = *F*
by (*auto simp*: *filter-eq-iff eventually-filtercomap elim*!: *eventually-mono*)

lemma *filtercomap-filtercomap*: *filtercomap f* (*filtercomap g F*) = *filtercomap* (λ*x*. *g* (*f x*)) *F*
unfolding *filter-eq-iff* **by** (*auto simp*: *eventually-filtercomap*)

lemma *filtercomap-mono*: *F* ≤ *F'* ⟹ *filtercomap f F* ≤ *filtercomap f F'*
by (*auto simp*: *eventually-filtercomap le-filter-def*)

lemma *filtercomap-bot* [*simp*]: *filtercomap f bot* = *bot*
by (*auto simp*: *filter-eq-iff eventually-filtercomap*)

lemma *filtercomap-top* [*simp*]: *filtercomap f top* = *top*
by (*auto simp*: *filter-eq-iff eventually-filtercomap*)

lemma *filtercomap-inf*: *filtercomap f* (*inf F1 F2*) = *inf* (*filtercomap f F1*) (*filtercomap f F2*)
unfolding *filter-eq-iff*
proof *safe*
fix *P*
assume *eventually P* (*filtercomap f* (*F1* ⊓ *F2*))
then obtain *Q R S* **where** ∗:
eventually Q F1 eventually R F2 ⋀*x*. *Q x* ⟹ *R x* ⟹ *S x* ⋀*x*. *S* (*f x*) ⟹ *P x*
unfolding *eventually-filtercomap eventually-inf* **by** *blast*
from ∗ **have** *eventually* (λ*x*. *Q* (*f x*)) (*filtercomap f F1*)
eventually (λ*x*. *R* (*f x*)) (*filtercomap f F2*)
by (*auto simp*: *eventually-filtercomap*)
with ∗ **show** *eventually P* (*filtercomap f F1* ⊓ *filtercomap f F2*)
unfolding *eventually-inf* **by** *blast*
next

fix P
assume *eventually* P (*inf* (*filtercomap* f *F1*) (*filtercomap* f *F2*))
then obtain Q Q' R R' **where** $*$:
eventually Q *F1 eventually* R *F2* $\bigwedge x.\ Q\ (f\ x) \Longrightarrow Q'\ x\ \bigwedge x.\ R\ (f\ x) \Longrightarrow R'\ x$
$\bigwedge x.\ Q'\ x \Longrightarrow R'\ x \Longrightarrow P\ x$
unfolding *eventually-filtercomap eventually-inf* **by** *blast*
from $*$ **have** *eventually* $(\lambda x.\ Q\ x \wedge R\ x)$ (*F1* $\sqcap$ *F2*) **by** (*auto simp*: *eventually-inf*)
with $*$ **show** *eventually* P (*filtercomap* f (*F1* $\sqcap$ *F2*))
by (*auto simp*: *eventually-filtercomap*)
qed

lemma *filtercomap-sup*: *filtercomap* f (*sup F1 F2*) $\geq$ *sup* (*filtercomap* f *F1*) (*filtercomap* f *F2*)
by (*intro sup-least filtercomap-mono inf-sup-ord*)

lemma *filtercomap-INF*: *filtercomap* f $(\bigsqcap b{\in}B.\ F\ b) = (\bigsqcap b{\in}B.$ *filtercomap* f $(F\ b))$
proof $-$
have $*$: *filtercomap* f $(\bigsqcap b{\in}B.\ F\ b) = (\bigsqcap b{\in}B.$ *filtercomap* f $(F\ b))$ **if** *finite* B **for** B
using *that* **by** *induction* (*simp-all add*: *filtercomap-inf*)
show *?thesis* **unfolding** *filter-eq-iff*
proof
fix P
have *eventually* P $(\bigsqcap b{\in}B.$ *filtercomap* f $(F\ b)) \longleftrightarrow$
$(\exists X.\ (X \subseteq B \wedge$ *finite* $X) \wedge$ *eventually* P $(\bigsqcap b{\in}X.$ *filtercomap* f $(F\ b)))$
by (*subst eventually-INF*) *blast*
also have $\ldots \longleftrightarrow (\exists X.\ (X \subseteq B \wedge$ *finite* $X) \wedge$ *eventually* P (*filtercomap* f $(\bigsqcap b{\in}X.\ F\ b)))$
by (*rule ex-cong*) (*simp add*: $*$)
also have $\ldots \longleftrightarrow$ *eventually* P (*filtercomap* f $(\bigsqcap (F\ `\ B)))$
unfolding *eventually-filtercomap* **by** (*subst eventually-INF*) *blast*
finally show *eventually* P (*filtercomap* f $(\bigsqcap (F\ `\ B))) =$
eventually P $(\bigsqcap b{\in}B.$ *filtercomap* f $(F\ b))$ **..**
qed
qed

lemma *filtercomap-SUP*:
filtercomap f $(\bigsqcup b{\in}B.\ F\ b) \geq (\bigsqcup b{\in}B.$ *filtercomap* f $(F\ b))$
by (*intro SUP-least filtercomap-mono SUP-upper*)

lemma *filtermap-le-iff-le-filtercomap*: *filtermap* f $F \leq G \longleftrightarrow F \leq$ *filtercomap* f G
unfolding *le-filter-def eventually-filtermap eventually-filtercomap*
using *eventually-mono* **by** *auto*

lemma *filtercomap-neq-bot*:
assumes $\bigwedge P.$ *eventually* P $F \Longrightarrow \exists x.\ P\ (f\ x)$
shows *filtercomap* f $F \neq$ *bot*

using *assms* **by** (*auto simp*: *trivial-limit-def eventually-filtercomap*)

lemma *filtercomap-neq-bot-surj*:
assumes $F \neq bot$ **and** *surj f*
shows *filtercomap f F* $\neq$ *bot*
proof (*rule filtercomap-neq-bot*)
fix P **assume** $*$: *eventually P F*
show $\exists x.\ P\ (f\ x)$
proof (*rule ccontr*)
assume $**$: $\neg(\exists x.\ P\ (f\ x))$
from $*$ **have** *eventually* (λ-. *False*) *F*
proof *eventually-elim*
case (*elim x*)
from ‹*surj f*› **obtain** *y* **where** $x = f\ y$ **by** *auto*
with *elim* **and** $**$ **show** *False* **by** *auto*
qed
with *assms* **show** *False* **by** (*simp add*: *trivial-limit-def*)
qed
qed

lemma *eventually-filtercomapI* [*intro*]:
assumes *eventually P F*
shows *eventually* ($\lambda x.\ P\ (f\ x)$) (*filtercomap f F*)
using *assms* **by** (*auto simp*: *eventually-filtercomap*)

lemma *filtermap-filtercomap*: *filtermap f* (*filtercomap f F*) $\leq F$
by (*auto simp*: *le-filter-def eventually-filtermap eventually-filtercomap*)

lemma *filtercomap-filtermap*: *filtercomap f* (*filtermap f F*) $\geq F$
unfolding *le-filter-def eventually-filtermap eventually-filtercomap*
by (*auto elim*!: *eventually-mono*)

90.2.4 Standard filters

definition *principal* :: *'a set* $\Rightarrow$ *'a filter* **where**
principal S = *Abs-filter* ($\lambda P.\ \forall x{\in}S.\ P\ x$)

lemma *eventually-principal*: *eventually P* (*principal S*) $\longleftrightarrow$ ($\forall x{\in}S.\ P\ x$)
unfolding *principal-def*
by (*rule eventually-Abs-filter*, *rule is-filter.intro*) *auto*

lemma *eventually-inf-principal*: *eventually P* (*inf F* (*principal s*)) $\longleftrightarrow$ *eventually* ($\lambda x.\ x \in s \longrightarrow P\ x$) *F*
unfolding *eventually-inf eventually-principal* **by** (*auto elim*: *eventually-mono*)

lemma *principal-UNIV*[*simp*]: *principal UNIV* = *top*
by (*auto simp*: *filter-eq-iff eventually-principal*)

lemma *principal-empty*[*simp*]: *principal* {} = *bot*

by (*auto simp*: *filter-eq-iff eventually-principal*)

lemma *principal-eq-bot-iff*: *principal X* = *bot* ⟷ *X* = {}
by (*auto simp add*: *filter-eq-iff eventually-principal*)

lemma *principal-le-iff*[*iff*]: *principal A* ≤ *principal B* ⟷ *A* ⊆ *B*
by (*auto simp*: *le-filter-def eventually-principal*)

lemma *le-principal*: *F* ≤ *principal A* ⟷ *eventually* (λ*x*. *x* ∈ *A*) *F*
unfolding *le-filter-def eventually-principal*
by (*force elim*: *eventually-mono*)

lemma *principal-inject*[*iff*]: *principal A* = *principal B* ⟷ *A* = *B*
unfolding *eq-iff* **by** *simp*

lemma *sup-principal*[*simp*]: *sup* (*principal A*) (*principal B*) = *principal* (*A* ∪ *B*)
unfolding *filter-eq-iff eventually-sup eventually-principal* **by** *auto*

lemma *inf-principal*[*simp*]: *inf* (*principal A*) (*principal B*) = *principal* (*A* ∩ *B*)
unfolding *filter-eq-iff eventually-inf eventually-principal*
by (*auto intro*: *exI*[*of* - λ*x*. *x* ∈ *A*] *exI*[*of* - λ*x*. *x* ∈ *B*])

lemma *SUP-principal*[*simp*]: (⨆*i*∈*I*. *principal* (*A i*)) = *principal* (⋃*i*∈*I*. *A i*)
unfolding *filter-eq-iff eventually-Sup* **by** (*auto simp*: *eventually-principal*)

lemma *INF-principal-finite*: *finite X* ⟹ (⨅*x*∈*X*. *principal* (*f x*)) = *principal* (⋂*x*∈*X*. *f x*)
by (*induct X rule*: *finite-induct*) *auto*

lemma *filtermap-principal*[*simp*]: *filtermap f* (*principal A*) = *principal* (*f* ‘ *A*)
unfolding *filter-eq-iff eventually-filtermap eventually-principal* **by** *simp*

lemma *filtercomap-principal*[*simp*]: *filtercomap f* (*principal A*) = *principal* (*f* −‘ *A*)
unfolding *filter-eq-iff eventually-filtercomap eventually-principal* **by** *fast*

90.2.5 Order filters

definition *at-top* :: (′*a*::*order*) *filter*
where *at-top* = (⨅*k*. *principal* {*k* ..})

lemma *at-top-sub*: *at-top* = (⨅*k*∈{*c*::′*a*::*linorder*..}. *principal* {*k* ..})
by (*auto intro*!: *INF-eq max.cobounded1 max.cobounded2 simp*: *at-top-def*)

lemma *eventually-at-top-linorder*: *eventually P at-top* ⟷ (∃ *N*::′*a*::*linorder*. ∀ *n*≥*N*. *P n*)
unfolding *at-top-def*
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal intro*: *max.cobounded1 max.cobounded2*)

lemma *eventually-filtercomap-at-top-linorder*:
eventually P (filtercomap f at-top) $\longleftrightarrow$ $(\exists N::'a::linorder.\ \forall x.\ f\ x \geq N \longrightarrow P\ x)$
by (*auto simp*: *eventually-filtercomap eventually-at-top-linorder*)

lemma *eventually-at-top-linorderI*:
fixes *c*::*'a*::*linorder*
assumes $\bigwedge x.\ c \leq x \Longrightarrow P\ x$
shows *eventually P at-top*
using *assms* **by** (*auto simp*: *eventually-at-top-linorder*)

lemma *eventually-ge-at-top* [*simp*]:
eventually $(\lambda x.\ (c::'-::linorder) \leq x)$ *at-top*
unfolding *eventually-at-top-linorder* **by** *auto*

lemma *eventually-at-top-dense*: *eventually P at-top* $\longleftrightarrow$ $(\exists N::'a::\{no\text{-}top, linorder\}.\ \forall n{>}N.\ P\ n)$
proof −
have *eventually P* $(\bigsqcap k.\ principal\ \{k <..\})$ $\longleftrightarrow$ $(\exists N::'a.\ \forall n{>}N.\ P\ n)$
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal intro*: *max.cobounded1 max.cobounded2*)
also have $(\bigsqcap k.\ principal\ \{k::'a <..\}) = at\text{-}top$
unfolding *at-top-def*
by (*intro INF-eq*) (*auto intro*: *less-imp-le simp*: *Ici-subset-Ioi-iff gt-ex*)
finally show *?thesis* **.**
qed

lemma *eventually-filtercomap-at-top-dense*:
eventually P (filtercomap f at-top) $\longleftrightarrow$ $(\exists N::'a::\{no\text{-}top, linorder\}.\ \forall x.\ f\ x > N \longrightarrow P\ x)$
by (*auto simp*: *eventually-filtercomap eventually-at-top-dense*)

lemma *eventually-at-top-not-equal* [*simp*]: *eventually* $(\lambda x::'a::\{no\text{-}top, linorder\}.\ x \neq c)$ *at-top*
unfolding *eventually-at-top-dense* **by** *auto*

lemma *eventually-gt-at-top* [*simp*]: *eventually* $(\lambda x.\ (c::'-::\{no\text{-}top, linorder\}) < x)$ *at-top*
unfolding *eventually-at-top-dense* **by** *auto*

lemma *eventually-all-ge-at-top*:
assumes *eventually P* (*at-top* :: (*'a* :: *linorder*) *filter*)
shows *eventually* $(\lambda x.\ \forall y{\geq}x.\ P\ y)$ *at-top*
proof −
from *assms* **obtain** *x* **where** $\bigwedge y.\ y \geq x \Longrightarrow P\ y$ **by** (*auto simp*: *eventually-at-top-linorder*)
hence $\forall z{\geq}y.\ P\ z$ **if** $y \geq x$ **for** *y* **using** *that* **by** *simp*
thus *?thesis* **by** (*auto simp*: *eventually-at-top-linorder*)
qed

definition *at-bot* :: (*'a*::*order*) *filter*
where *at-bot* = (⨅ *k*. *principal* {.. *k*})

lemma *at-bot-sub*: *at-bot* = (⨅ *k*∈{.. *c*::*'a*::*linorder*}. *principal* {.. *k*})
by (*auto intro*!: *INF-eq min.cobounded1 min.cobounded2 simp*: *at-bot-def*)

lemma *eventually-at-bot-linorder*:
fixes *P* :: *'a*::*linorder* ⇒ *bool* **shows** *eventually P at-bot* ⟷ (∃ *N*. ∀ *n*≤*N*. *P n*)
unfolding *at-bot-def*
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal intro*: *min.cobounded1 min.cobounded2*)

lemma *eventually-at-bot-linorderI*:
fixes *c*::*'a*::*linorder*
assumes ⋀*x*. *x* ≤ *c* ⟹ *P x*
shows *eventually P at-bot*
using *assms* **by** (*auto simp*: *eventually-at-bot-linorder*)

lemma *eventually-filtercomap-at-bot-linorder*:
eventually P (*filtercomap f at-bot*) ⟷ (∃ *N*::*'a*::*linorder*. ∀ *x*. *f x* ≤ *N* ⟶ *P x*)
by (*auto simp*: *eventually-filtercomap eventually-at-bot-linorder*)

lemma *eventually-le-at-bot* [*simp*]:
eventually (λ*x*. *x* ≤ (*c*::-::*linorder*)) *at-bot*
unfolding *eventually-at-bot-linorder* **by** *auto*

lemma *eventually-at-bot-dense*: *eventually P at-bot* ⟷ (∃ *N*::*'a*::{*no-bot*, *linorder*}. ∀ *n*<*N*. *P n*)
proof −
have *eventually P* (⨅ *k*. *principal* {..< *k*}) ⟷ (∃ *N*::*'a*. ∀ *n*<*N*. *P n*)
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal intro*: *min.cobounded1 min.cobounded2*)
also have (⨅ *k*. *principal* {..< *k*::*'a*}) = *at-bot*
unfolding *at-bot-def*
by (*intro INF-eq*) (*auto intro*: *less-imp-le simp*: *Iic-subset-Iio-iff lt-ex*)
finally show *?thesis* **.**
qed

lemma *eventually-filtercomap-at-bot-dense*:
eventually P (*filtercomap f at-bot*) ⟷ (∃ *N*::*'a*::{*no-bot*, *linorder*}. ∀ *x*. *f x* < *N* ⟶ *P x*)
by (*auto simp*: *eventually-filtercomap eventually-at-bot-dense*)

lemma *eventually-at-bot-not-equal* [*simp*]: *eventually* (λ*x*::*'a*::{*no-bot*, *linorder*}. *x* ≠ *c*) *at-bot*
unfolding *eventually-at-bot-dense* **by** *auto*

lemma *eventually-gt-at-bot* [*simp*]:

eventually (λ*x*. *x* < (*c*::-::*unbounded-dense-linorder*)) *at-bot*
unfolding *eventually-at-bot-dense* **by** *auto*

lemma *trivial-limit-at-bot-linorder* [*simp*]: ¬ *trivial-limit* (*at-bot* ::('*a*::*linorder*) *filter*)
unfolding *trivial-limit-def*
by (*metis eventually-at-bot-linorder order-refl*)

lemma *trivial-limit-at-top-linorder* [*simp*]: ¬ *trivial-limit* (*at-top* ::('*a*::*linorder*) *filter*)
unfolding *trivial-limit-def*
by (*metis eventually-at-top-linorder order-refl*)

90.3 Sequentially

abbreviation *sequentially* :: *nat filter*
where *sequentially* ≡ *at-top*

lemma *eventually-sequentially*:
eventually P sequentially ⟷ (∃ *N*. ∀ *n*≥*N*. *P n*)
by (*rule eventually-at-top-linorder*)

lemma *frequently-sequentially*:
frequently P sequentially ⟷ (∀ *N*. ∃ *n*≥*N*. *P n*)
by (*simp add*: *frequently-def eventually-sequentially*)

lemma *sequentially-bot* [*simp*, *intro*]: *sequentially* ≠ *bot*
unfolding *filter-eq-iff eventually-sequentially* **by** *auto*

lemmas *trivial-limit-sequentially* = *sequentially-bot*

lemma *eventually-False-sequentially* [*simp*]:
¬ *eventually* (λ*n*. *False*) *sequentially*
by (*simp add*: *eventually-False*)

lemma *le-sequentially*:
F ≤ *sequentially* ⟷ (∀ *N*. *eventually* (λ*n*. *N* ≤ *n*) *F*)
by (*simp add*: *at-top-def le-INF-iff le-principal*)

lemma *eventually-sequentiallyI* [*intro?*]:
assumes ⋀*x*. *c* ≤ *x* ⟹ *P x*
shows *eventually P sequentially*
using *assms* **by** (*auto simp*: *eventually-sequentially*)

lemma *eventually-sequentially-Suc* [*simp*]: *eventually* (λ*i*. *P* (*Suc i*)) *sequentially* ⟷ *eventually P sequentially*
unfolding *eventually-sequentially* **by** (*metis Suc-le-D Suc-le-mono le-Suc-eq*)

lemma *eventually-sequentially-seg* [*simp*]: *eventually* (λ*n*. *P* (*n* + *k*)) *sequentially*

$\longleftrightarrow$ *eventually P sequentially*
using *eventually-sequentially-Suc*[*of* $\lambda n.\ P\ (n + k)$ **for** k] **by** (*induction k*) *auto*

lemma *filtermap-sequentually-ne-bot*: *filtermap f sequentially* $\neq$ *bot*
by (*simp add*: *filtermap-bot-iff*)

90.4 Increasing finite subsets

definition *finite-subsets-at-top* **where**
finite-subsets-at-top A = ($\bigsqcap$ X∈{X. *finite X* $\wedge$ $X \subseteq A$}. *principal* {Y. *finite Y* $\wedge$ $X \subseteq Y$ $\wedge$ $Y \subseteq A$})

abbreviation *finite-sets-at-top* $\equiv$ *finite-subsets-at-top UNIV*

lemma *eventually-finite-subsets-at-top*:
eventually P (*finite-subsets-at-top A*) $\longleftrightarrow$
($\exists X.$ *finite X* $\wedge$ $X \subseteq A$ $\wedge$ ($\forall Y.$ *finite Y* $\wedge$ $X \subseteq Y$ $\wedge$ $Y \subseteq A \longrightarrow P\ Y$))
unfolding *finite-subsets-at-top-def*
proof (*subst eventually-INF-base*, *goal-cases*)
show {X. *finite X* $\wedge$ $X \subseteq A$} $\neq$ {} **by** *auto*
next
case (*2 B C*)
thus *?case* **by** (*intro bexI*[*of* - $B \cup C$]) *auto*
qed (*simp-all add*: *eventually-principal*)

lemma *eventually-finite-subsets-at-top-weakI* [*intro*]:
assumes $\bigwedge X.$ *finite X* $\Longrightarrow$ $X \subseteq A$ $\Longrightarrow$ *P X*
shows *eventually P* (*finite-subsets-at-top A*)
proof −
have *eventually* ($\lambda X.$ *finite X* $\wedge$ $X \subseteq A$) (*finite-subsets-at-top A*)
by (*auto simp*: *eventually-finite-subsets-at-top*)
thus *?thesis* **by** *eventually-elim* (*use assms* **in** *auto*)
qed

lemma *finite-subsets-at-top-neq-bot* [*simp*]: *finite-subsets-at-top A* $\neq$ *bot*
proof −
have ¬*eventually* ($\lambda x.$ *False*) (*finite-subsets-at-top A*)
by (*auto simp*: *eventually-finite-subsets-at-top*)
thus *?thesis* **by** *auto*
qed

lemma *filtermap-image-finite-subsets-at-top*:
assumes *inj-on f A*
shows *filtermap* ((‘) *f*) (*finite-subsets-at-top A*) = *finite-subsets-at-top* (*f* ‘ *A*)
unfolding *filter-eq-iff eventually-filtermap*
proof (*safe*, *goal-cases*)
case (*1 P*)
then obtain X **where** X: *finite X* $X \subseteq A$ $\bigwedge Y.$ *finite Y* $\Longrightarrow$ $X \subseteq Y$ $\Longrightarrow$ $Y \subseteq A$ $\Longrightarrow$ *P* (*f* ‘ *Y*)

unfolding *eventually-finite-subsets-at-top* **by** *force*
show *?case* **unfolding** *eventually-finite-subsets-at-top eventually-filtermap*
proof (*rule exI*[*of - f ‘ X*], *intro conjI allI impI*, *goal-cases*)
case (*3 Y*)
with *assms* **and** *X*(*1*,*2*) **have** *P* (*f ‘* (*f* −‘ *Y* ∩ *A*)) **using** *X*(*1*,*2*)
by (*intro X*(*3*) *finite-vimage-IntI*) *auto*
also have *f ‘* (*f* −‘ *Y* ∩ *A*) = *Y* **using** *assms 3* **by** *blast*
finally show *?case* **.**
qed (*insert assms X*(*1*,*2*), *auto intro*!: *finite-vimage-IntI*)
next
case (*2 P*)
then obtain *X* **where** *X*: *finite X X* ⊆ *f ‘ A* ⋀*Y*. *finite Y* ⟹ *X* ⊆ *Y* ⟹ *Y* ⊆ *f ‘ A* ⟹ *P Y*
unfolding *eventually-finite-subsets-at-top* **by** *force*
show *?case* **unfolding** *eventually-finite-subsets-at-top eventually-filtermap*
proof (*rule exI*[*of - f* −‘ *X* ∩ *A*], *intro conjI allI impI*, *goal-cases*)
case (*3 Y*)
with *X*(*1*,*2*) **and** *assms* **show** *?case* **by** (*intro X*(*3*)) *force*+
qed (*insert assms X*(*1*), *auto intro*!: *finite-vimage-IntI*)
qed

lemma *eventually-finite-subsets-at-top-finite*:
assumes *finite A*
shows *eventually P* (*finite-subsets-at-top A*) ⟷ *P A*
unfolding *eventually-finite-subsets-at-top* **using** *assms* **by** *force*

lemma *finite-subsets-at-top-finite*: *finite A* ⟹ *finite-subsets-at-top A* = *principal* {*A*}
by (*auto simp*: *filter-eq-iff eventually-finite-subsets-at-top-finite eventually-principal*)

90.5 The cofinite filter

definition *cofinite* = *Abs-filter* (λ*P*. *finite* {*x*. ¬ *P x*})

abbreviation *Inf-many* :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** ‹∃ ∞› *10*)
where *Inf-many P* ≡ *frequently P cofinite*

abbreviation *Alm-all* :: (*'a* ⇒ *bool*) ⇒ *bool* (**binder** ‹∀ ∞› *10*)
where *Alm-all P* ≡ *eventually P cofinite*

notation (*ASCII*)
Inf-many (**binder** ‹*INFM* › *10*) **and**
Alm-all (**binder** ‹*MOST* › *10*)

lemma *eventually-cofinite*: *eventually P cofinite* ⟷ *finite* {*x*. ¬ *P x*}
unfolding *cofinite-def*
proof (*rule eventually-Abs-filter*, *rule is-filter.intro*)
fix *P Q* :: *'a* ⇒ *bool* **assume** *finite* {*x*. ¬ *P x*} *finite* {*x*. ¬ *Q x*}
from *finite-UnI*[*OF this*] **show** *finite* {*x*. ¬ (*P x* ∧ *Q x*)}

by (*rule rev-finite-subset*) *auto*
next
fix *P Q* :: *'a* ⇒ *bool* **assume** *P*: *finite* {*x*. ¬ *P x*} **and** ∗: ∀ *x*. *P x* ⟶ *Q x*
from ∗ **show** *finite* {*x*. ¬ *Q x*}
by (*intro finite-subset*[*OF* - *P*]) *auto*
qed *simp*

lemma *frequently-cofinite*: *frequently P cofinite* ⟷ ¬ *finite* {*x*. *P x*}
by (*simp add*: *frequently-def eventually-cofinite*)

lemma *cofinite-bot*[*simp*]: *cofinite* = (*bot*::*'a filter*) ⟷ *finite* (*UNIV* :: *'a set*)
unfolding *trivial-limit-def eventually-cofinite* **by** *simp*

lemma *cofinite-eq-sequentially*: *cofinite* = *sequentially*
unfolding *filter-eq-iff eventually-sequentially eventually-cofinite*
proof *safe*
fix *P* :: *nat* ⇒ *bool* **assume** [*simp*]: *finite* {*x*. ¬ *P x*}
show ∃ *N*. ∀ *n*≥*N*. *P n*
proof *cases*
assume {*x*. ¬ *P x*} ≠ {} **then show** *?thesis*
by (*intro exI*[*of* - *Suc* (*Max* {*x*. ¬ *P x*})]) (*auto simp*: *Suc-le-eq*)
qed *auto*
next
fix *P* :: *nat* ⇒ *bool* **and** *N* :: *nat* **assume** ∀ *n*≥*N*. *P n*
then have {*x*. ¬ *P x*} ⊆ {..< *N*}
by (*auto simp*: *not-le*)
then show *finite* {*x*. ¬ *P x*}
by (*blast intro*: *finite-subset*)
qed

90.5.1 Product of filters

definition *prod-filter* :: *'a filter* ⇒ *'b filter* ⇒ (*'a* × *'b*) *filter* (**infixr** ‹×$_F$› *80*)
where
prod-filter F G =
(⨅(*P*, *Q*)∈{(*P*, *Q*). *eventually P F* ∧ *eventually Q G*}. *principal* {(*x*, *y*). *P x* ∧ *Q y*})

lemma *eventually-prod-filter*: *eventually P* (*F* ×$_F$ *G*) ⟷
(∃ *Pf Pg*. *eventually Pf F* ∧ *eventually Pg G* ∧ (∀ *x y*. *Pf x* ⟶ *Pg y* ⟶ *P* (*x*, *y*)))
unfolding *prod-filter-def*
proof (*subst eventually-INF-base*, *goal-cases*)
case *2*
moreover have *eventually Pf F* ⟹ *eventually Qf F* ⟹ *eventually Pg G* ⟹ *eventually Qg G* ⟹
∃ *P Q*. *eventually P F* ∧ *eventually Q G* ∧
Collect P × *Collect Q* ⊆ *Collect Pf* × *Collect Pg* ∩ *Collect Qf* × *Collect Qg*
for *Pf Pg Qf Qg*

by (*intro conjI exI*[*of - inf Pf Qf*] *exI*[*of - inf Pg Qg*])
(*auto simp*: *inf-fun-def eventually-conj*)
ultimately show *?case*
by *auto*
qed (*auto simp*: *eventually-principal intro*: *eventually-True*)

lemma *eventually-prod1*:
assumes $B \neq bot$
shows $(\forall_F\ (x,\ y)\ in\ A \times_F B.\ P\ x) \longleftrightarrow (\forall_F\ x\ in\ A.\ P\ x)$
unfolding *eventually-prod-filter*
proof *safe*
fix *R Q*
assume $*$: $\forall_F\ x\ in\ A.\ R\ x\ \forall_F\ x\ in\ B.\ Q\ x\ \forall x\ y.\ R\ x \longrightarrow Q\ y \longrightarrow P\ x$
with ‹$B \neq bot$› **obtain** *y* **where** *Q y* **by** (*auto dest*: *eventually-happens*)
with $*$ **show** *eventually P A*
by (*force elim*: *eventually-mono*)
next
assume *eventually P A*
then show $\exists\, Pf\ Pg.\ eventually\ Pf\ A \wedge eventually\ Pg\ B \wedge (\forall x\ y.\ Pf\ x \longrightarrow Pg\ y \longrightarrow P\ x)$
by (*intro exI*[*of - P*] *exI*[*of - λx. True*]) *auto*
qed

lemma *eventually-prod2*:
assumes $A \neq bot$
shows $(\forall_F\ (x,\ y)\ in\ A \times_F B.\ P\ y) \longleftrightarrow (\forall_F\ y\ in\ B.\ P\ y)$
unfolding *eventually-prod-filter*
proof *safe*
fix *R Q*
assume $*$: $\forall_F\ x\ in\ A.\ R\ x\ \forall_F\ x\ in\ B.\ Q\ x\ \forall x\ y.\ R\ x \longrightarrow Q\ y \longrightarrow P\ y$
with ‹$A \neq bot$› **obtain** *x* **where** *R x* **by** (*auto dest*: *eventually-happens*)
with $*$ **show** *eventually P B*
by (*force elim*: *eventually-mono*)
next
assume *eventually P B*
then show $\exists\, Pf\ Pg.\ eventually\ Pf\ A \wedge eventually\ Pg\ B \wedge (\forall x\ y.\ Pf\ x \longrightarrow Pg\ y \longrightarrow P\ y)$
by (*intro exI*[*of - P*] *exI*[*of - λx. True*]) *auto*
qed

lemma *eventually-eventually-prod-filter1*:
assumes *eventually P* $(F \times_F G)$
shows *eventually* $(\lambda x.\ eventually\ (\lambda y.\ P\ (x,\ y))\ G)\ F$
proof −
from *assms* **obtain** *Pf Pg* **where**
$*$: *eventually Pf F eventually Pg G* $\bigwedge x\ y.\ Pf\ x \Longrightarrow Pg\ y \Longrightarrow P\ (x,\ y)$
unfolding *eventually-prod-filter* **by** *auto*
show *?thesis*
using $*(1)$

```
  proof eventually-elim
    case x: (elim x)
    show ?case
      using ∗(2) by eventually-elim (use x ∗(3) in auto)
  qed
qed

lemma eventually-eventually-prod-filter2:
  assumes eventually P (F ×_F G)
  shows   eventually (λy. eventually (λx. P (x, y)) F) G
proof −
  from assms obtain Pf Pg where
    ∗: eventually Pf F eventually Pg G ⋀x y. Pf x ⟹ Pg y ⟹ P (x, y)
    unfolding eventually-prod-filter by auto
  show ?thesis
    using ∗(2)
  proof eventually-elim
    case y: (elim y)
    show ?case
      using ∗(1) by eventually-elim (use y ∗(3) in auto)
  qed
qed

lemma INF-filter-bot-base:
  fixes F :: 'a ⇒ 'b filter
  assumes ∗: ⋀i j. i ∈ I ⟹ j ∈ I ⟹ ∃k∈I. F k ≤ F i ⊓ F j
  shows (∏ i∈I. F i) = bot ⟷ (∃ i∈I. F i = bot)
proof (cases ∃ i∈I. F i = bot)
  case True
  then have (∏ i∈I. F i) ≤ bot
    by (auto intro: INF-lower2)
  with True show ?thesis
    by (auto simp: bot-unique)
next
  case False
  moreover have (∏ i∈I. F i) ≠ bot
  proof (cases I = {})
    case True
    then show ?thesis
      by (auto simp add: filter-eq-iff)
  next
    case False': False
    show ?thesis
    proof (rule INF-filter-not-bot)
      fix J
      assume finite J J ⊆ I
      then have ∃ k∈I. F k ≤ (∏ i∈J. F i)
      proof (induct J)
        case empty
```

```
      then show ?case
        using ‹I ≠ {}› by auto
    next
      case (insert i J)
      then obtain k where k ∈ I F k ≤ (∏ i∈J. F i) by auto
      with insert ∗[of i k] show ?case
        by auto
    qed
    with False show (∏ i∈J. F i) ≠ ⊥
      by (auto simp: bot-unique)
  qed
 qed
 ultimately show ?thesis
  by auto
qed

lemma Collect-empty-eq-bot: Collect P = {} ⟷ P = ⊥
 by auto

lemma prod-filter-eq-bot: A ×F B = bot ⟷ A = bot ∨ B = bot
 unfolding trivial-limit-def
proof
 assume ∀F x in A ×F B. False
 then obtain Pf Pg
   where Pf: eventually (λx. Pf x) A and Pg: eventually (λy. Pg y) B
   and ∗: ∀x y. Pf x ⟶ Pg y ⟶ False
   unfolding eventually-prod-filter by fast
 from ∗ have (∀x. ¬ Pf x) ∨ (∀y. ¬ Pg y) by fast
 with Pf Pg show (∀F x in A. False) ∨ (∀F x in B. False) by auto
next
 assume (∀F x in A. False) ∨ (∀F x in B. False)
 then show ∀F x in A ×F B. False
   unfolding eventually-prod-filter by (force intro: eventually-True)
qed

lemma prod-filter-mono: F ≤ F' ⟹ G ≤ G' ⟹ F ×F G ≤ F' ×F G'
 by (auto simp: le-filter-def eventually-prod-filter)

lemma prod-filter-mono-iff:
 assumes nAB: A ≠ bot B ≠ bot
 shows A ×F B ≤ C ×F D ⟷ A ≤ C ∧ B ≤ D
proof safe
 assume ∗: A ×F B ≤ C ×F D
 with assms have A ×F B ≠ bot
   by (auto simp: bot-unique prod-filter-eq-bot)
 with ∗ have C ×F D ≠ bot
   by (auto simp: bot-unique)
 then have nCD: C ≠ bot D ≠ bot
   by (auto simp: prod-filter-eq-bot)
```

show $A \leq C$
proof (*rule filter-leI*)
fix P **assume** *eventually* P C **with** $*$[*THEN filter-leD*, *of* $\lambda(x,\ y).\ P\ x$] **show** *eventually* P A
using *nAB nCD* **by** (*simp add*: *eventually-prod1 eventually-prod2*)
qed

show $B \leq D$
proof (*rule filter-leI*)
fix P **assume** *eventually* P D **with** $*$[*THEN filter-leD*, *of* $\lambda(x,\ y).\ P\ y$] **show** *eventually* P B
using *nAB nCD* **by** (*simp add*: *eventually-prod1 eventually-prod2*)
qed
qed (*intro prod-filter-mono*)

lemma *eventually-prod-same*: *eventually* P $(F \times_F F) \longleftrightarrow$
$(\exists\, Q.\ \textit{eventually}\ Q\ F \wedge (\forall\, x\ y.\ Q\ x \longrightarrow Q\ y \longrightarrow P\ (x,\ y)))$
unfolding *eventually-prod-filter* **by** (*blast intro*!: *eventually-conj*)

lemma *eventually-prod-sequentially*:
eventually P (*sequentially* $\times_F$ *sequentially*) $\longleftrightarrow$ $(\exists\, N.\ \forall\, m \geq N.\ \forall\, n \geq N.\ P\ (n,\ m))$
unfolding *eventually-prod-same eventually-sequentially* **by** *auto*

lemma *principal-prod-principal*: *principal* $A \times_F$ *principal* $B =$ *principal* $(A \times B)$
unfolding *filter-eq-iff eventually-prod-filter eventually-principal*
by (*fast intro*: *exI*[*of* - $\lambda x.\ x \in A$] *exI*[*of* - $\lambda x.\ x \in B$])

lemma *le-prod-filterI*:
filtermap fst $F \leq A \Longrightarrow$ *filtermap snd* $F \leq B \Longrightarrow F \leq A \times_F B$
unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force elim*: *eventually-elim2*)

lemma *filtermap-fst-prod-filter*: *filtermap fst* $(A \times_F B) \leq A$
unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force intro*: *eventually-True*)

lemma *filtermap-snd-prod-filter*: *filtermap snd* $(A \times_F B) \leq B$
unfolding *le-filter-def eventually-filtermap eventually-prod-filter*
by (*force intro*: *eventually-True*)

lemma *prod-filter-INF*:
assumes $I \neq \{\}$ **and** $J \neq \{\}$
shows $(\bigsqcap i{\in}I.\ A\ i) \times_F (\bigsqcap j{\in}J.\ B\ j) = (\bigsqcap i{\in}I.\ \bigsqcap j{\in}J.\ A\ i \times_F B\ j)$
proof (*rule antisym*)
from ‹$I \neq \{\}$› **obtain** i **where** $i \in I$ **by** *auto*
from ‹$J \neq \{\}$› **obtain** j **where** $j \in J$ **by** *auto*

show $(\prod i{\in}I.\ \prod j{\in}J.\ A\ i \times_F B\ j) \leq (\prod i{\in}I.\ A\ i) \times_F (\prod j{\in}J.\ B\ j)$
by (*fast intro*: *le-prod-filterI INF-greatest INF-lower2*
order-trans[*OF filtermap-INF*] ‹$i \in I$› ‹$j \in J$›
filtermap-fst-prod-filter filtermap-snd-prod-filter)
show $(\prod i{\in}I.\ A\ i) \times_F (\prod j{\in}J.\ B\ j) \leq (\prod i{\in}I.\ \prod j{\in}J.\ A\ i \times_F B\ j)$
by (*intro INF-greatest prod-filter-mono INF-lower*)
qed

lemma *filtermap-Pair*: *filtermap* $(\lambda x.\ (f\ x,\ g\ x))\ F \leq$ *filtermap* $f\ F \times_F$ *filtermap* $g\ F$
by (*rule le-prod-filterI*, *simp-all add*: *filtermap-filtermap*)

lemma *eventually-prodI*: *eventually* $P\ F \Longrightarrow$ *eventually* $Q\ G \Longrightarrow$ *eventually* $(\lambda x.\ P\ (fst\ x) \wedge Q\ (snd\ x))\ (F \times_F G)$
unfolding *eventually-prod-filter* **by** *auto*

lemma *prod-filter-INF1*: $I \neq \{\} \Longrightarrow (\prod i{\in}I.\ A\ i) \times_F B = (\prod i{\in}I.\ A\ i \times_F B)$
using *prod-filter-INF*[*of* I $\{B\}$ A $\lambda x.\ x$] **by** *simp*

lemma *prod-filter-INF2*: $J \neq \{\} \Longrightarrow A \times_F (\prod i{\in}J.\ B\ i) = (\prod i{\in}J.\ A \times_F B\ i)$
using *prod-filter-INF*[*of* $\{A\}$ J $\lambda x.\ x\ B$] **by** *simp*

lemma *prod-filtermap1*: *prod-filter* $(filtermap\ f\ F)\ G =$ *filtermap* $(apfst\ f)$ (*prod-filter* $F\ G$)
unfolding *filter-eq-iff eventually-filtermap eventually-prod-filter*
apply *safe*
subgoal by *auto*
subgoal for $P\ Q\ R$ **by**(*rule exI*[**where** $x{=}\lambda y.\ \exists x.\ y = f\ x \wedge Q\ x$])(*auto intro*: *eventually-mono*)
done

lemma *prod-filtermap2*: *prod-filter* F $(filtermap\ g\ G) =$ *filtermap* $(apsnd\ g)$ (*prod-filter* $F\ G$)
unfolding *filter-eq-iff eventually-filtermap eventually-prod-filter*
apply *safe*
subgoal by *auto*
subgoal for $P\ Q\ R$ **by**(*auto intro*: *exI*[**where** $x{=}\lambda y.\ \exists x.\ y = g\ x \wedge R\ x$] *eventually-mono*)
done

lemma *prod-filter-assoc*:
prod-filter (*prod-filter* $F\ G$) $H =$ *filtermap* $(\lambda(x,\ y,\ z).\ ((x,\ y),\ z))$ (*prod-filter* F (*prod-filter* $G\ H$))
apply(*clarsimp simp add*: *filter-eq-iff eventually-filtermap eventually-prod-filter*; *safe*)
subgoal for $P\ Q\ R\ S\ T$ **by**(*auto 4 4 intro*: *exI*[**where** $x{=}\lambda(a,\ b).\ T\ a \wedge S\ b$])
subgoal for $P\ Q\ R\ S\ T$ **by**(*auto 4 3 intro*: *exI*[**where** $x{=}\lambda(a,\ b).\ Q\ a \wedge S\ b$])
done

lemma *prod-filter-principal-singleton*: *prod-filter* (*principal* {*x*}) *F* = *filtermap* (*Pair x*) *F*
by(*fastforce simp add*: *filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim*: *eventually-mono intro*: *exI*[**where** *x*=λ*a*. *a* = *x*])

lemma *prod-filter-principal-singleton2*: *prod-filter F* (*principal* {*x*}) = *filtermap* (λ*a*. (*a*, *x*)) *F*
by(*fastforce simp add*: *filter-eq-iff eventually-prod-filter eventually-principal eventually-filtermap elim*: *eventually-mono intro*: *exI*[**where** *x*=λ*a*. *a* = *x*])

lemma *prod-filter-commute*: *prod-filter F G* = *filtermap prod.swap* (*prod-filter G F*)
by(*auto simp add*: *filter-eq-iff eventually-prod-filter eventually-filtermap*)

90.6 Limits

definition *filterlim* :: ('*a* ⇒ '*b*) ⇒ '*b filter* ⇒ '*a filter* ⇒ *bool* **where**
filterlim f F2 F1 ⟷ *filtermap f F1* ≤ *F2*

syntax
-LIM :: *pttrns* ⇒ '*a* ⇒ '*b* ⇒ '*a* ⇒ *bool* (‹(‹*indent=3 notation=‹binder LIM›› LIM* (-)/ (-)./ (-) :> (-))› [*1000*, *10*, *0*, *10*] *10*)
syntax-consts
-LIM == *filterlim*
translations
LIM x F1. *f* :> *F2* == *CONST filterlim* (λ*x*. *f*) *F2 F1*

lemma *filterlim-filtercomapI*: *filterlim f F G* ⟹ *filterlim* (λ*x*. *f* (*g x*)) *F* (*filtercomap g G*)
unfolding *filterlim-def*
by (*metis order-trans filtermap-filtercomap filtermap-filtermap filtermap-mono*)

lemma *filterlim-top* [*simp*]: *filterlim f top F*
by (*simp add*: *filterlim-def*)

lemma *filterlim-iff*:
(*LIM x F1*. *f x* :> *F2*) ⟷ (∀ *P*. *eventually P F2* ⟶ *eventually* (λ*x*. *P* (*f x*)) *F1*)
unfolding *filterlim-def le-filter-def eventually-filtermap* **..**

lemma *filterlim-compose*:
filterlim g F3 F2 ⟹ *filterlim f F2 F1* ⟹ *filterlim* (λ*x*. *g* (*f x*)) *F3 F1*
unfolding *filterlim-def filtermap-filtermap*[*symmetric*] **by** (*metis filtermap-mono order-trans*)

lemma *filterlim-mono*:
filterlim f F2 F1 ⟹ *F2* ≤ *F2'* ⟹ *F1'* ≤ *F1* ⟹ *filterlim f F2' F1'*
unfolding *filterlim-def* **by** (*metis filtermap-mono order-trans*)

lemma *filterlim-ident*: *LIM x F. x :> F*
by (*simp add*: *filterlim-def filtermap-ident*)

lemma *filterlim-cong*:
F1 = F1' ⟹ F2 = F2' ⟹ eventually (*λx. f x = g x*) *F2 ⟹ filterlim f F1 F2 = filterlim g F1' F2'*
by (*auto simp*: *filterlim-def le-filter-def eventually-filtermap elim*: *eventually-elim2*)

lemma *filterlim-mono-eventually*:
assumes *filterlim f F G* **and** *ord*: *F ≤ F' G' ≤ G*
assumes *eq*: *eventually* (*λx. f x = f' x*) *G'*
shows *filterlim f' F' G'*
proof −
have *filterlim f F' G'*
by (*simp add*: *filterlim-mono*[*OF - ord*] *assms*)
then show *?thesis*
by (*rule filterlim-cong*[*OF refl refl eq, THEN iffD1*])
qed

lemma *filtermap-mono-strong*: *inj f ⟹ filtermap f F ≤ filtermap f G ⟷ F ≤ G*
apply (*safe intro*!: *filtermap-mono*)
apply (*auto simp*: *le-filter-def eventually-filtermap*)
apply (*erule-tac x=λx. P* (*inv f x*) **in** *allE*)
apply *auto*
done

lemma *eventually-compose-filterlim*:
assumes *eventually P F filterlim f F G*
shows *eventually* (*λx. P* (*f x*)) *G*
using *assms* **by** (*simp add*: *filterlim-iff*)

lemma *filtermap-eq-strong*: *inj f ⟹ filtermap f F = filtermap f G ⟷ F = G*
by (*simp add*: *filtermap-mono-strong eq-iff*)

lemma *filtermap-fun-inverse*:
assumes *g*: *filterlim g F G*
assumes *f*: *filterlim f G F*
assumes *ev*: *eventually* (*λx. f* (*g x*) = *x*) *G*
shows *filtermap f F = G*
proof (*rule antisym*)
show *filtermap f F ≤ G*
using *f* **unfolding** *filterlim-def* **.**
have *G = filtermap f* (*filtermap g G*)
using *ev* **by** (*auto elim*: *eventually-elim2 simp*: *filter-eq-iff eventually-filtermap*)
also have *. . . ≤ filtermap f F*
using *g* **by** (*intro filtermap-mono*) (*simp add*: *filterlim-def*)
finally show *G ≤ filtermap f F* **.**
qed

lemma *filterlim-principal*:
(*LIM x F. f x :> principal S*) ⟷ (*eventually* (λ*x. f x* ∈ *S*) *F*)
unfolding *filterlim-def eventually-filtermap le-principal* **..**

lemma *filterlim-filtercomap* [*intro*]: *filterlim f F* (*filtercomap f F*)
unfolding *filterlim-def* **by** (*rule filtermap-filtercomap*)

lemma *filterlim-inf*:
(*LIM x F1. f x :> inf F2 F3*) ⟷ ((*LIM x F1. f x :> F2*) ∧ (*LIM x F1. f x :> F3*))
unfolding *filterlim-def* **by** *simp*

lemma *filterlim-INF*:
(*LIM x F. f x :>* (⨅ *b*∈*B. G b*)) ⟷ (∀ *b*∈*B. LIM x F. f x :> G b*)
unfolding *filterlim-def le-INF-iff* **..**

lemma *filterlim-INF-INF*:
(⋀*m. m* ∈ *J* ⟹ ∃ *i*∈*I. filtermap f* (*F i*) ≤ *G m*) ⟹ *LIM x* (⨅ *i*∈*I. F i*). *f x* :> (⨅ *j*∈*J. G j*)
unfolding *filterlim-def* **by** (*rule order-trans*[*OF filtermap-INF INF-mono*])

lemma *filterlim-INF'*: *x* ∈ *A* ⟹ *filterlim f F* (*G x*) ⟹ *filterlim f F* (⨅ *x*∈*A. G x*)
unfolding *filterlim-def* **by** (*rule order.trans*[*OF filtermap-mono*[*OF INF-lower*]])

lemma *filterlim-filtercomap-iff*: *filterlim f* (*filtercomap g G*) *F* ⟷ *filterlim* (*g* ∘ *f*) *G F*
by (*simp add*: *filterlim-def filtermap-le-iff-le-filtercomap filtercomap-filtercomap o-def*)

lemma *filterlim-iff-le-filtercomap*: *filterlim f F G* ⟷ *G* ≤ *filtercomap f F*
by (*simp add*: *filterlim-def filtermap-le-iff-le-filtercomap*)

lemma *filterlim-base*:
(⋀*m x. m* ∈ *J* ⟹ *i m* ∈ *I*) ⟹ (⋀*m x. m* ∈ *J* ⟹ *x* ∈ *F* (*i m*) ⟹ *f x* ∈ *G m*) ⟹
LIM x (⨅ *i*∈*I. principal* (*F i*)). *f x* :> (⨅ *j*∈*J. principal* (*G j*))
by (*force intro*!: *filterlim-INF-INF simp*: *image-subset-iff*)

lemma *filterlim-base-iff*:
assumes *I* ≠ {} **and** *chain*: ⋀*i j. i* ∈ *I* ⟹ *j* ∈ *I* ⟹ *F i* ⊆ *F j* ∨ *F j* ⊆ *F i*
shows (*LIM x* (⨅ *i*∈*I. principal* (*F i*)). *f x* :> ⨅ *j*∈*J. principal* (*G j*)) ⟷
(∀ *j*∈*J*. ∃ *i*∈*I*. ∀ *x*∈*F i. f x* ∈ *G j*)
unfolding *filterlim-INF filterlim-principal*
proof (*subst eventually-INF-base*)
fix *i j* **assume** *i* ∈ *I j* ∈ *I*
with *chain*[*OF this*] **show** ∃ *x*∈*I. principal* (*F x*) ≤ *inf* (*principal* (*F i*)) (*principal* (*F j*))

by *auto*
qed (*auto simp*: *eventually-principal* ‹$I \neq \{\}$›)

lemma *filterlim-filtermap*: *filterlim f F1* (*filtermap g F2*) = *filterlim* (λ*x*. *f* (*g x*)) *F1 F2*
unfolding *filterlim-def filtermap-filtermap* **..**

lemma *filterlim-sup*:
filterlim f F F1 ⟹ *filterlim f F F2* ⟹ *filterlim f F* (*sup F1 F2*)
unfolding *filterlim-def filtermap-sup* **by** *auto*

lemma *filterlim-sequentially-Suc*:
(*LIM x sequentially*. *f* (*Suc x*) :> *F*) ⟷ (*LIM x sequentially*. *f x* :> *F*)
unfolding *filterlim-iff* **by** (*subst eventually-sequentially-Suc*) *simp*

lemma *filterlim-Suc*: *filterlim Suc sequentially sequentially*
by (*simp add*: *filterlim-iff eventually-sequentially*)

lemma *filterlim-If*:
LIM x inf F (*principal* {*x*. *P x*}). *f x* :> *G* ⟹
LIM x inf F (*principal* {*x*. ¬ *P x*}). *g x* :> *G* ⟹
LIM x F. *if P x then f x else g x* :> *G*
unfolding *filterlim-iff eventually-inf-principal* **by** (*auto simp*: *eventually-conj-iff*)

lemma *filterlim-Pair*:
LIM x F. *f x* :> *G* ⟹ *LIM x F*. *g x* :> *H* ⟹ *LIM x F*. (*f x*, *g x*) :> *G* ×$_F$ *H*
unfolding *filterlim-def*
by (*rule order-trans*[*OF filtermap-Pair prod-filter-mono*])

90.7 Limits to *at-top* and *at-bot*

lemma *filterlim-at-top*:
fixes *f* :: *'a* ⇒ (*'b*::*linorder*)
shows (*LIM x F*. *f x* :> *at-top*) ⟷ (∀ *Z*. *eventually* (λ*x*. *Z* ≤ *f x*) *F*)
by (*auto simp*: *filterlim-iff eventually-at-top-linorder elim*!: *eventually-mono*)

lemma *filterlim-at-top-mono*:
LIM x F. *f x* :> *at-top* ⟹ *eventually* (λ*x*. *f x* ≤ (*g x*::*'a*::*linorder*)) *F* ⟹
LIM x F. *g x* :> *at-top*
by (*auto simp*: *filterlim-at-top elim*: *eventually-elim2 intro*: *order-trans*)

lemma *filterlim-at-top-dense*:
fixes *f* :: *'a* ⇒ (*'b*::*unbounded-dense-linorder*)
shows (*LIM x F*. *f x* :> *at-top*) ⟷ (∀ *Z*. *eventually* (λ*x*. *Z* < *f x*) *F*)
by (*metis eventually-mono*[*of* - *F*] *eventually-gt-at-top order-less-imp-le*
filterlim-at-top[*of f F*] *filterlim-iff*[*of f at-top F*])

lemma *filterlim-at-top-ge*:
fixes *f* :: *'a* ⇒ (*'b*::*linorder*) **and** *c* :: *'b*

shows (*LIM x F. f x :> at-top*) $\longleftrightarrow$ ($\forall Z{\geq}c$. *eventually* ($\lambda x.\ Z \leq f\ x$) *F*)
unfolding *at-top-sub*[*of c*] *filterlim-INF* **by** (*auto simp add*: *filterlim-principal*)

lemma *filterlim-at-top-at-top*:
fixes *f* :: *'a*::*linorder* $\Rightarrow$ *'b*::*linorder*
assumes *mono*: $\bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
assumes *bij*: $\bigwedge x.\ P\ x \Longrightarrow f\ (g\ x) = x\ \bigwedge x.\ P\ x \Longrightarrow Q\ (g\ x)$
assumes *Q*: *eventually Q at-top*
assumes *P*: *eventually P at-top*
shows *filterlim f at-top at-top*
proof −
from *P* **obtain** *x* **where** *x*: $\bigwedge y.\ x \leq y \Longrightarrow P\ y$
unfolding *eventually-at-top-linorder* **by** *auto*
show *?thesis*
proof (*intro filterlim-at-top-ge*[*THEN iffD2*] *allI impI*)
fix *z* **assume** $x \leq z$
with *x* **have** *P z* **by** *auto*
have *eventually* ($\lambda x.\ g\ z \leq x$) *at-top*
by (*rule eventually-ge-at-top*)
with *Q* **show** *eventually* ($\lambda x.\ z \leq f\ x$) *at-top*
by *eventually-elim* (*metis mono bij* ‹*P z*›)
qed
qed

lemma *filterlim-at-top-gt*:
fixes *f* :: *'a* $\Rightarrow$ (*'b*::*unbounded-dense-linorder*) **and** *c* :: *'b*
shows (*LIM x F. f x :> at-top*) $\longleftrightarrow$ ($\forall Z{>}c$. *eventually* ($\lambda x.\ Z \leq f\ x$) *F*)
by (*metis filterlim-at-top order-less-le-trans gt-ex filterlim-at-top-ge*)

lemma *filterlim-at-bot*:
fixes *f* :: *'a* $\Rightarrow$ (*'b*::*linorder*)
shows (*LIM x F. f x :> at-bot*) $\longleftrightarrow$ ($\forall Z$. *eventually* ($\lambda x.\ f\ x \leq Z$) *F*)
by (*auto simp*: *filterlim-iff eventually-at-bot-linorder elim*!: *eventually-mono*)

lemma *filterlim-at-bot-dense*:
fixes *f* :: *'a* $\Rightarrow$ (*'b*::{*dense-linorder*, *no-bot*})
shows (*LIM x F. f x :> at-bot*) $\longleftrightarrow$ ($\forall Z$. *eventually* ($\lambda x.\ f\ x < Z$) *F*)
proof (*auto simp add*: *filterlim-at-bot*[*of f F*])
fix *Z* :: *'b*
from *lt-ex* [*of Z*] **obtain** *Z'* **where** *1*: $Z' < Z$ **..**
assume $\forall Z$. *eventually* ($\lambda x.\ f\ x \leq Z$) *F*
hence *eventually* ($\lambda x.\ f\ x \leq Z'$) *F* **by** *auto*
thus *eventually* ($\lambda x.\ f\ x < Z$) *F*
by (*rule eventually-mono*) (*use 1* **in** *auto*)
next
fix *Z* :: *'b*
show $\forall Z$. *eventually* ($\lambda x.\ f\ x < Z$) *F* $\Longrightarrow$ *eventually* ($\lambda x.\ f\ x \leq Z$) *F*
by (*drule spec* [*of - Z*], *erule eventually-mono*, *auto simp add*: *less-imp-le*)
qed

lemma *filterlim-at-bot-le*:
fixes $f :: 'a \Rightarrow ('b::linorder)$ **and** $c :: 'b$
shows $(LIM\ x\ F.\ f\ x :> at\text{-}bot) \longleftrightarrow (\forall Z{\leq}c.\ eventually\ (\lambda x.\ Z \geq f\ x)\ F)$
unfolding *filterlim-at-bot*
proof *safe*
fix Z **assume** $*$: $\forall Z{\leq}c.\ eventually\ (\lambda x.\ Z \geq f\ x)\ F$
with $*[THEN\ spec,\ of\ min\ Z\ c]$ **show** $eventually\ (\lambda x.\ Z \geq f\ x)\ F$
by (*auto elim*!: *eventually-mono*)
qed *simp*

lemma *filterlim-at-bot-lt*:
fixes $f :: 'a \Rightarrow ('b::unbounded\text{-}dense\text{-}linorder)$ **and** $c :: 'b$
shows $(LIM\ x\ F.\ f\ x :> at\text{-}bot) \longleftrightarrow (\forall Z{<}c.\ eventually\ (\lambda x.\ Z \geq f\ x)\ F)$
by (*metis filterlim-at-bot filterlim-at-bot-le lt-ex order-le-less-trans*)

lemma *filterlim-at-top-div-const-nat*:
assumes $c > 0$
shows $filterlim\ (\lambda x::nat.\ x\ div\ c)\ at\text{-}top\ at\text{-}top$
unfolding *filterlim-at-top*
proof
fix $C :: nat$
have $*$: $n\ div\ c \geq C$ **if** $n \geq C * c$ **for** n
using *assms that* **by** (*metis div-le-mono div-mult-self-is-m*)
have $eventually\ (\lambda n.\ n \geq C * c)\ at\text{-}top$
by (*rule eventually-ge-at-top*)
thus $eventually\ (\lambda n.\ n\ div\ c \geq C)\ at\text{-}top$
by *eventually-elim* (*use* $*$ **in** *auto*)
qed

lemma *filterlim-finite-subsets-at-top*:
$filterlim\ f\ (finite\text{-}subsets\text{-}at\text{-}top\ A)\ F \longleftrightarrow$
$(\forall X.\ finite\ X \wedge X \subseteq A \longrightarrow eventually\ (\lambda y.\ finite\ (f\ y) \wedge X \subseteq f\ y \wedge f\ y \subseteq A)\ F)$
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
thus *?rhs*
proof (*safe*, *goal-cases*)
case (*1 X*)
hence $*$: $(\forall_F\ x\ in\ F.\ P\ (f\ x))$ **if** $eventually\ P\ (finite\text{-}subsets\text{-}at\text{-}top\ A)$ **for** P
using *that* **by** (*auto simp*: *filterlim-def le-filter-def eventually-filtermap*)
have $\forall_F\ Y\ in\ finite\text{-}subsets\text{-}at\text{-}top\ A.\ finite\ Y \wedge X \subseteq Y \wedge Y \subseteq A$
using *1* **unfolding** *eventually-finite-subsets-at-top* **by** *force*
thus *?case* **by** (*intro* $*$) *auto*
qed
next
assume *rhs*: *?rhs*
show *?lhs* **unfolding** *filterlim-def le-filter-def eventually-finite-subsets-at-top*

proof (*safe*, *goal-cases*)
case (*1 P X*)
with *rhs* **have** $\forall_F\ y$ *in* F. *finite* $(f\ y) \wedge X \subseteq f\ y \wedge f\ y \subseteq A$ **by** *auto*
thus *eventually P* (*filtermap f F*) **unfolding** *eventually-filtermap*
by *eventually-elim* (*insert 1*, *auto*)
qed
qed

lemma *filterlim-atMost-at-top*:
filterlim $(\lambda n.\ \{..n\})$ (*finite-subsets-at-top* (*UNIV* :: *nat set*)) *at-top*
unfolding *filterlim-finite-subsets-at-top*
proof (*safe*, *goal-cases*)
case (*1 X*)
then obtain *n* **where** *n*: $X \subseteq \{..n\}$ **by** (*auto simp*: *finite-nat-set-iff-bounded-le*)
show *?case* **using** *eventually-ge-at-top*[*of n*]
by *eventually-elim* (*insert n*, *auto*)
qed

lemma *filterlim-lessThan-at-top*:
filterlim $(\lambda n.\ \{..<n\})$ (*finite-subsets-at-top* (*UNIV* :: *nat set*)) *at-top*
unfolding *filterlim-finite-subsets-at-top*
proof (*safe*, *goal-cases*)
case (*1 X*)
then obtain *n* **where** *n*: $X \subseteq \{..<n\}$ **by** (*auto simp*: *finite-nat-set-iff-bounded*)
show *?case* **using** *eventually-ge-at-top*[*of n*]
by *eventually-elim* (*insert n*, *auto*)
qed

lemma *filterlim-minus-const-nat-at-top*:
filterlim $(\lambda n.\ n - c)$ *sequentially sequentially*
unfolding *filterlim-at-top*
proof
fix *a* :: *nat*
show *eventually* $(\lambda n.\ n - c \geq a)$ *at-top*
using *eventually-ge-at-top*[*of* $a + c$] **by** *eventually-elim auto*
qed

lemma *filterlim-add-const-nat-at-top*:
filterlim $(\lambda n.\ n + c)$ *sequentially sequentially*
unfolding *filterlim-at-top*
proof
fix *a* :: *nat*
show *eventually* $(\lambda n.\ n + c \geq a)$ *at-top*
using *eventually-ge-at-top*[*of a*] **by** *eventually-elim auto*
qed

90.8 Setup *'a filter* for lifting and transfer

lemma *filtermap-id* [*simp*, *id-simps*]: *filtermap id* = *id*

by(*simp add*: *fun-eq-iff id-def filtermap-ident*)

lemma *filtermap-id'* [*simp*]: *filtermap* (λ*x*. *x*) = (λ*F*. *F*)
using *filtermap-id* **unfolding** *id-def* **.**

context includes *lifting-syntax*
begin

definition *map-filter-on* :: *'a set* ⇒ (*'a* ⇒ *'b*) ⇒ *'a filter* ⇒ *'b filter* **where**
map-filter-on X f F = *Abs-filter* (λ*P*. *eventually* (λ*x*. *P* (*f x*) ∧ *x* ∈ *X*) *F*)

lemma *is-filter-map-filter-on*:
is-filter (λ*P*. ∀ *F x in F*. *P* (*f x*) ∧ *x* ∈ *X*) ⟷ *eventually* (λ*x*. *x* ∈ *X*) *F*
proof(*rule iffI*; *unfold-locales*)
show ∀ *F x in F*. *True* ∧ *x* ∈ *X* **if** *eventually* (λ*x*. *x* ∈ *X*) *F* **using** *that* **by** *simp*
show ∀ *F x in F*. (*P* (*f x*) ∧ *Q* (*f x*)) ∧ *x* ∈ *X* **if** ∀ *F x in F*. *P* (*f x*) ∧ *x* ∈ *X*
∀ *F x in F*. *Q* (*f x*) ∧ *x* ∈ *X* **for** *P Q*
using *eventually-conj*[*OF that*] **by**(*auto simp add*: *conj-ac cong*: *conj-cong*)
show ∀ *F x in F*. *Q* (*f x*) ∧ *x* ∈ *X* **if** ∀ *x*. *P x* ⟶ *Q x* ∀ *F x in F*. *P* (*f x*) ∧ *x*
∈ *X* **for** *P Q*
using *that*(*2*) **by**(*rule eventually-mono*)(*use that*(*1*) **in** *auto*)
show *eventually* (λ*x*. *x* ∈ *X*) *F* **if** *is-filter* (λ*P*. ∀ *F x in F*. *P* (*f x*) ∧ *x* ∈ *X*)
using *is-filter.True*[*OF that*] **by** *simp*
qed

lemma *eventually-map-filter-on*: *eventually P* (*map-filter-on X f F*) = (∀ *F x in*
F. *P* (*f x*) ∧ *x* ∈ *X*)
if *eventually* (λ*x*. *x* ∈ *X*) *F*
by(*simp add*: *is-filter-map-filter-on map-filter-on-def eventually-Abs-filter that*)

lemma *map-filter-on-UNIV*: *map-filter-on UNIV* = *filtermap*
by(*simp add*: *map-filter-on-def filtermap-def fun-eq-iff*)

lemma *map-filter-on-comp*: *map-filter-on X f* (*map-filter-on Y g F*) = *map-filter-on*
Y (*f* ∘ *g*) *F*
if *g* ‘ *Y* ⊆ *X* **and** *eventually* (λ*x*. *x* ∈ *Y*) *F*
unfolding *map-filter-on-def* **using** *that*(*1*)
by(*auto simp add*: *eventually-Abs-filter that*(*2*) *is-filter-map-filter-on intro*!: *arg-cong*[**where**
f=*Abs-filter*] *arg-cong2*[**where** *f*=*eventually*])

inductive *rel-filter* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'a filter* ⇒ *'b filter* ⇒ *bool* **for** *R F G*
where
rel-filter R F G **if** *eventually* (*case-prod R*) *Z map-filter-on* {(*x*, *y*). *R x y*} *fst Z*
= *F map-filter-on* {(*x*, *y*). *R x y*} *snd Z* = *G*

lemma *rel-filter-eq* [*relator-eq*]: *rel-filter* (=) = (=)
proof(*intro ext iffI*)+
show *F* = *G* **if** *rel-filter* (=) *F G* **for** *F G* **using** *that*
by *cases*(*clarsimp simp add*: *filter-eq-iff eventually-map-filter-on split-def cong*:

rev-conj-cong)
show *rel-filter* (=) *F G* **if** *F* = *G* **for** *F G* **unfolding** ‹*F* = *G*›
proof
let *?Z* = *map-filter-on UNIV* (λ*x*. (*x*, *x*)) *G*
have [*simp*]: *range* (λ*x*. (*x*, *x*)) ⊆ {(*x*, *y*). *x* = *y*} **by** *auto*
show *map-filter-on* {(*x*, *y*). *x* = *y*} *fst ?Z* = *G* **and** *map-filter-on* {(*x*, *y*). *x* = *y*} *snd ?Z* = *G*
by(*simp-all add*: *map-filter-on-comp*)(*simp-all add*: *map-filter-on-UNIV o-def*)
show ∀ *F* (*x*, *y*) *in ?Z*. *x* = *y* **by**(*simp add*: *eventually-map-filter-on*)
qed
qed

lemma *rel-filter-mono* [*relator-mono*]: *rel-filter A* ≤ *rel-filter B* **if** *le*: *A* ≤ *B*
proof(*clarify elim*!: *rel-filter.cases*)
show *rel-filter B* (*map-filter-on* {(*x*, *y*). *A x y*} *fst Z*) (*map-filter-on* {(*x*, *y*). *A x y*} *snd Z*)
(**is** *rel-filter* - *?X ?Y*) **if** ∀ *F* (*x*, *y*) *in Z*. *A x y* **for** *Z*
proof
let *?Z* = *map-filter-on* {(*x*, *y*). *A x y*} *id Z*
show ∀ *F* (*x*, *y*) *in ?Z*. *B x y* **using** *le that*
by(*simp add*: *eventually-map-filter-on le-fun-def split-def conj-commute cong*: *conj-cong*)
have [*simp*]: {(*x*, *y*). *A x y*} ⊆ {(*x*, *y*). *B x y*} **using** *le* **by** *auto*
show *map-filter-on* {(*x*, *y*). *B x y*} *fst ?Z* = *?X map-filter-on* {(*x*, *y*). *B x y*} *snd ?Z* = *?Y*
using *le that* **by**(*simp-all add*: *le-fun-def map-filter-on-comp*)
qed
qed

lemma *rel-filter-conversep*: *rel-filter A*$^{-1-1}$ = (*rel-filter A*)$^{-1-1}$
proof(*safe intro*!: *ext elim*!: *rel-filter.cases*)
show ∗: *rel-filter A* (*map-filter-on* {(*x*, *y*). *A*$^{-1-1}$ *x y*} *snd Z*) (*map-filter-on* {(*x*, *y*). *A*$^{-1-1}$ *x y*} *fst Z*)
(**is** *rel-filter* - *?X ?Y*) **if** ∀ *F* (*x*, *y*) *in Z*. *A*$^{-1-1}$ *x y* **for** *A Z*
proof
let *?Z* = *map-filter-on* {(*x*, *y*). *A y x*} *prod.swap Z*
show ∀ *F* (*x*, *y*) *in ?Z*. *A x y* **using** *that* **by**(*simp add*: *eventually-map-filter-on*)
have [*simp*]: *prod.swap* ‘ {(*x*, *y*). *A y x*} ⊆ {(*x*, *y*). *A x y*} **by** *auto*
show *map-filter-on* {(*x*, *y*). *A x y*} *fst ?Z* = *?X map-filter-on* {(*x*, *y*). *A x y*} *snd ?Z* = *?Y*
using *that* **by**(*simp-all add*: *map-filter-on-comp o-def*)
qed
show *rel-filter A*$^{-1-1}$ (*map-filter-on* {(*x*, *y*). *A x y*} *snd Z*) (*map-filter-on* {(*x*, *y*). *A x y*} *fst Z*)
if ∀ *F* (*x*, *y*) *in Z*. *A x y* **for** *Z* **using** ∗[*of A*$^{-1-1}$ *Z*] *that* **by** *simp*
qed

lemma *rel-filter-distr* [*relator-distr*]:
rel-filter A OO rel-filter B = *rel-filter* (*A OO B*)

proof(*safe intro*!: *ext elim*!: *rel-filter.cases*)
let *?AB* = {(*x*, *y*). (*A OO B*) *x y*}
show (*rel-filter A OO rel-filter B*)
(*map-filter-on* {(*x*, *y*). (*A OO B*) *x y*} *fst Z*) (*map-filter-on* {(*x*, *y*). (*A OO B*) *x y*} *snd Z*)
(**is** (- *OO* -) *?F ?H*) **if** ∀$_F$ (*x*, *y*) *in Z*. (*A OO B*) *x y* **for** *Z*
proof
let *?G* = *map-filter-on ?AB* (λ(*x*, *y*). *SOME z*. *A x z* ∧ *B z y*) *Z*
show *rel-filter A ?F ?G*
proof
let *?Z* = *map-filter-on ?AB* (λ(*x*, *y*). (*x*, *SOME z*. *A x z* ∧ *B z y*)) *Z*
show ∀$_F$ (*x*, *y*) *in ?Z*. *A x y* **using** *that*
by(*auto simp add*: *eventually-map-filter-on split-def elim*!: *eventually-mono intro*: *someI2*)
have [*simp*]: (λ*p*. (*fst p*, *SOME z*. *A* (*fst p*) *z* ∧ *B z* (*snd p*))) ‘ {*p*. (*A OO B*) (*fst p*) (*snd p*)} ⊆ {*p*. *A* (*fst p*) (*snd p*)} **by**(*auto intro*: *someI2*)
show *map-filter-on* {(*x*, *y*). *A x y*} *fst ?Z* = *?F map-filter-on* {(*x*, *y*). *A x y*} *snd ?Z* = *?G*
using *that* **by**(*simp-all add*: *map-filter-on-comp split-def o-def*)
qed
show *rel-filter B ?G ?H*
proof
let *?Z* = *map-filter-on ?AB* (λ(*x*, *y*). (*SOME z*. *A x z* ∧ *B z y*, *y*)) *Z*
show ∀$_F$ (*x*, *y*) *in ?Z*. *B x y* **using** *that*
by(*auto simp add*: *eventually-map-filter-on split-def elim*!: *eventually-mono intro*: *someI2*)
have [*simp*]: (λ*p*. (*SOME z*. *A* (*fst p*) *z* ∧ *B z* (*snd p*), *snd p*)) ‘ {*p*. (*A OO B*) (*fst p*) (*snd p*)} ⊆ {*p*. *B* (*fst p*) (*snd p*)} **by**(*auto intro*: *someI2*)
show *map-filter-on* {(*x*, *y*). *B x y*} *fst ?Z* = *?G map-filter-on* {(*x*, *y*). *B x y*} *snd ?Z* = *?H*
using *that* **by**(*simp-all add*: *map-filter-on-comp split-def o-def*)
qed
qed

fix *F G*
assume *F*: ∀$_F$ (*x*, *y*) *in F*. *A x y* **and** *G*: ∀$_F$ (*x*, *y*) *in G*. *B x y*
and *eq*: *map-filter-on* {(*x*, *y*). *B x y*} *fst G* = *map-filter-on* {(*x*, *y*). *A x y*} *snd F* (**is** *?Y2* = *?Y1*)
let *?X* = *map-filter-on* {(*x*, *y*). *A x y*} *fst F*
and *?Z* = (*map-filter-on* {(*x*, *y*). *B x y*} *snd G*)
have *step*: ∃ *P'*≤*P*. ∃ *Q'* ≤ *Q*. *eventually P' F* ∧ *eventually Q' G* ∧ {*y*. ∃ *x*. *P'* (*x*, *y*)} = {*y*. ∃ *z*. *Q'* (*y*, *z*)}
if *P*: *eventually P F* **and** *Q*: *eventually Q G* **for** *P Q*
proof −
let *?P* = λ(*x*, *y*). *P* (*x*, *y*) ∧ *A x y* **and** *?Q* = λ(*y*, *z*). *Q* (*y*, *z*) ∧ *B y z*
define *P'* **where** *P'* ≡ λ(*x*, *y*). *?P* (*x*, *y*) ∧ (∃ *z*. *?Q* (*y*, *z*))
define *Q'* **where** *Q'* ≡ λ(*y*, *z*). *?Q* (*y*, *z*) ∧ (∃ *x*. *?P* (*x*, *y*))
have *P'* ≤ *P Q'* ≤ *Q* {*y*. ∃ *x*. *P'* (*x*, *y*)} = {*y*. ∃ *z*. *Q'* (*y*, *z*)}
by(*auto simp add*: *P'-def Q'-def*)

moreover
from *P Q F G* **have** *P'*: *eventually ?P F* **and** *Q'*: *eventually ?Q G*
by(*simp-all add*: *eventually-conj-iff split-def*)
from *P' F* **have** $\forall_F$ *y in ?Y1*. $\exists$ *x*. *P* (*x*, *y*) $\wedge$ *A x y*
by(*auto simp add*: *eventually-map-filter-on elim*!: *eventually-mono*)
from *this*[*folded eq*] **obtain** *Q''* **where** *Q''*: *eventually Q'' G*
and *Q''P*: {*y*. $\exists$ *z*. *Q''* (*y*, *z*)} $\subseteq$ {*y*. $\exists$ *x*. *?P* (*x*, *y*)}
using *G* **by**(*fastforce simp add*: *eventually-map-filter-on*)
have *eventually* (*inf Q'' ?Q*) *G* **using** *Q'' Q'* **by**(*auto intro*: *eventually-conj simp add*: *inf-fun-def*)
then have *eventually Q' G* **using** *Q''P* **by**(*auto elim*!: *eventually-mono simp add*: *Q'-def*)
moreover
from *Q' G* **have** $\forall_F$ *y in ?Y2*. $\exists$ *z*. *Q* (*y*, *z*) $\wedge$ *B y z*
by(*auto simp add*: *eventually-map-filter-on elim*!: *eventually-mono*)
from *this*[*unfolded eq*] **obtain** *P''* **where** *P''*: *eventually P'' F*
and *P''Q*: {*y*. $\exists$ *x*. *P''* (*x*, *y*)} $\subseteq$ {*y*. $\exists$ *z*. *?Q* (*y*, *z*)}
using *F* **by**(*fastforce simp add*: *eventually-map-filter-on*)
have *eventually* (*inf P'' ?P*) *F* **using** *P'' P'* **by**(*auto intro*: *eventually-conj simp add*: *inf-fun-def*)
then have *eventually P' F* **using** *P''Q* **by**(*auto elim*!: *eventually-mono simp add*: *P'-def*)
ultimately show *?thesis* **by** *blast*
qed

show *rel-filter* (*A OO B*) *?X ?Z*
proof
let *?Y* = λ*Y*. $\exists$ *X Z*. *eventually X ?X* $\wedge$ *eventually Z ?Z* $\wedge$ (λ(*x*, *z*). *X x* $\wedge$ *Z z* $\wedge$ (*A OO B*) *x z*) $\leq$ *Y*
have *Y*: *is-filter ?Y*
proof
show *?Y* (λ-. *True*) **by**(*auto simp add*: *le-fun-def intro*: *eventually-True*)
show *?Y* (λ*x*. *P x* $\wedge$ *Q x*) **if** *?Y P ?Y Q* **for** *P Q* **using** *that*
apply *clarify*
apply(*intro exI conjI*; (*elim eventually-rev-mp*; *fold imp-conjL*; *intro always-eventually allI*; *rule imp-refl*)*?*)
apply *auto*
done
show *?Y Q* **if** *?Y P* $\forall$ *x*. *P x* $\longrightarrow$ *Q x* **for** *P Q* **using** *that* **by** *blast*
qed
define *Y* **where** *Y* = *Abs-filter ?Y*
have *eventually-Y*: *eventually P Y* $\longleftrightarrow$ *?Y P* **for** *P*
using *eventually-Abs-filter*[*OF Y*, *of P*] **by**(*simp add*: *Y-def*)
show *YY*: $\forall_F$ (*x*, *y*) *in Y*. (*A OO B*) *x y* **using** *F G*
by(*auto simp add*: *eventually-Y eventually-map-filter-on eventually-conj-iff intro*!: *eventually-True*)
have *?Y* (λ(*x*, *z*). *P x* $\wedge$ (*A OO B*) *x z*) $\longleftrightarrow$ ($\forall_F$ (*x*, *y*) *in F*. *P x* $\wedge$ *A x y*) (**is** *?lhs* = *?rhs*) **for** *P*
proof

show *?lhs* **if** *?rhs* **using** *G F that*
by(*auto 4 3 intro*: *exI*[**where** *x*=λ-. *True*] *simp add*: *eventually-map-filter-on split-def*)
assume *?lhs*
then obtain *X Z* **where** ∀ $_F$ (*x*, *y*) *in F*. *X x* ∧ *A x y*
and ∀ $_F$ (*x*, *y*) *in G*. *Z y* ∧ *B x y*
and (λ(*x*, *z*). *X x* ∧ *Z z* ∧ (*A OO B*) *x z*) ≤ (λ(*x*, *z*). *P x* ∧ (*A OO B*) *x z*)
using *F G* **by**(*auto simp add*: *eventually-map-filter-on split-def*)
from *step*[*OF this*(*1*, *2*)] *this*(*3*)
show *?rhs* **by**(*clarsimp elim*!: *eventually-rev-mp simp add*: *le-fun-def*)(*fastforce intro*: *always-eventually*)
qed
then show *map-filter-on ?AB fst Y* = *?X*
by(*simp add*: *filter-eq-iff YY eventually-map-filter-on*)(*simp add*: *eventually-Y eventually-map-filter-on F G*; *simp add*: *split-def*)

have *?Y* (λ(*x*, *z*). *P z* ∧ (*A OO B*) *x z*) ⟷ (∀ $_F$ (*x*, *y*) *in G*. *P y* ∧ *B x y*) (**is** *?lhs* = *?rhs*) **for** *P*
proof
show *?lhs* **if** *?rhs* **using** *G F that*
by(*auto 4 3 intro*: *exI*[**where** *x*=λ-. *True*] *simp add*: *eventually-map-filter-on split-def*)
assume *?lhs*
then obtain *X Z* **where** ∀ $_F$ (*x*, *y*) *in F*. *X x* ∧ *A x y*
and ∀ $_F$ (*x*, *y*) *in G*. *Z y* ∧ *B x y*
and (λ(*x*, *z*). *X x* ∧ *Z z* ∧ (*A OO B*) *x z*) ≤ (λ(*x*, *z*). *P z* ∧ (*A OO B*) *x z*)
using *F G* **by**(*auto simp add*: *eventually-map-filter-on split-def*)
from *step*[*OF this*(*1*, *2*)] *this*(*3*)
show *?rhs* **by**(*clarsimp elim*!: *eventually-rev-mp simp add*: *le-fun-def*)(*fastforce intro*: *always-eventually*)
qed
then show *map-filter-on ?AB snd Y* = *?Z*
by(*simp add*: *filter-eq-iff YY eventually-map-filter-on*)(*simp add*: *eventually-Y eventually-map-filter-on F G*; *simp add*: *split-def*)
qed
qed

lemma *filtermap-parametric*: ((*A* ===> *B*) ===> *rel-filter A* ===> *rel-filter B*) *filtermap filtermap*
proof(*intro rel-funI*; *erule rel-filter.cases*; *hypsubst*)
fix *f g Z*
assume *fg*: (*A* ===> *B*) *f g* **and** *Z*: ∀ $_F$ (*x*, *y*) *in Z*. *A x y*
have *rel-filter B* (*map-filter-on* {(*x*, *y*). *A x y*} (*f* ∘ *fst*) *Z*) (*map-filter-on* {(*x*, *y*). *A x y*} (*g* ∘ *snd*) *Z*)
(**is** *rel-filter* - *?F ?G*)
proof
let *?Z* = *map-filter-on* {(*x*, *y*). *A x y*} (*map-prod f g*) *Z*
show ∀ $_F$ (*x*, *y*) *in ?Z*. *B x y* **using** *fg Z*
by(*auto simp add*: *eventually-map-filter-on split-def elim*!: *eventually-mono*

rel-funD)
have [*simp*]: *map-prod f g ' {p. A (fst p) (snd p)} ⊆ {p. B (fst p) (snd p)}*
using *fg* **by**(*auto dest*: *rel-funD*)
show *map-filter-on {(x, y). B x y} fst ?Z = ?F map-filter-on {(x, y). B x y} snd ?Z = ?G*
using *Z* **by**(*auto simp add*: *map-filter-on-comp split-def*)
qed
thus *rel-filter B (filtermap f (map-filter-on {(x, y). A x y} fst Z)) (filtermap g (map-filter-on {(x, y). A x y} snd Z))*
using *Z* **by**(*simp add*: *map-filter-on-UNIV*[*symmetric*] *map-filter-on-comp*)
qed

lemma *rel-filter-Grp*: *rel-filter (Grp UNIV f) = Grp UNIV (filtermap f)*
proof((*intro antisym predicate2I*; (*elim GrpE*; *hypsubst*)*?*), *rule GrpI*[*OF - UNIV-I*])
fix *F G*
assume *rel-filter (Grp UNIV f) F G*
hence *rel-filter (=) (filtermap f F) (filtermap id G)*
by(*rule filtermap-parametric*[*THEN rel-funD*, *THEN rel-funD*, *rotated*])(*simp add*: *Grp-def rel-fun-def*)
thus *filtermap f F = G* **by**(*simp add*: *rel-filter-eq*)
next
fix *F* :: *'a filter*
have *rel-filter (=) F F* **by**(*simp add*: *rel-filter-eq*)
hence *rel-filter (Grp UNIV f) (filtermap id F) (filtermap f F)*
by(*rule filtermap-parametric*[*THEN rel-funD*, *THEN rel-funD*, *rotated*])(*simp add*: *Grp-def rel-fun-def*)
thus *rel-filter (Grp UNIV f) F (filtermap f F)* **by** *simp*
qed

lemma *Quotient-filter* [*quot-map*]:
Quotient R Abs Rep T ⟹ Quotient (rel-filter R) (filtermap Abs) (filtermap Rep) (rel-filter T)
unfolding *Quotient-alt-def5 rel-filter-eq*[*symmetric*] *rel-filter-Grp*[*symmetric*]
by(*simp add*: *rel-filter-distr*[*symmetric*] *rel-filter-conversep*[*symmetric*] *rel-filter-mono*)

lemma *left-total-rel-filter* [*transfer-rule*]: *left-total A ⟹ left-total (rel-filter A)*
unfolding *left-total-alt-def rel-filter-eq*[*symmetric*] *rel-filter-conversep*[*symmetric*] *rel-filter-distr*
by(*rule rel-filter-mono*)

lemma *right-total-rel-filter* [*transfer-rule*]: *right-total A ⟹ right-total (rel-filter A)*
using *left-total-rel-filter*[*of* A^{-1-1}] **by**(*simp add*: *rel-filter-conversep*)

lemma *bi-total-rel-filter* [*transfer-rule*]: *bi-total A ⟹ bi-total (rel-filter A)*
unfolding *bi-total-alt-def* **by**(*simp add*: *left-total-rel-filter right-total-rel-filter*)

lemma *left-unique-rel-filter* [*transfer-rule*]: *left-unique A ⟹ left-unique (rel-filter A)*

unfolding *left-unique-alt-def rel-filter-eq*[*symmetric*] *rel-filter-conversep*[*symmetric*] *rel-filter-distr*
by(*rule rel-filter-mono*)

lemma *right-unique-rel-filter* [*transfer-rule*]:
  right-unique A ⟹ *right-unique* (*rel-filter A*)
using *left-unique-rel-filter*[*of* A^{-1-1}] **by**(*simp add*: *rel-filter-conversep*)

lemma *bi-unique-rel-filter* [*transfer-rule*]: *bi-unique A* ⟹ *bi-unique* (*rel-filter A*)
by(*simp add*: *bi-unique-alt-def left-unique-rel-filter right-unique-rel-filter*)

lemma *eventually-parametric* [*transfer-rule*]:
  ((*A* ===> (=)) ===> *rel-filter A* ===> (=)) *eventually eventually*
  unfolding *rel-fun-def*
 by(*force elim*!: *rel-filter.cases eventually-rev-mp simp add*: *eventually-map-filter-on intro*: *always-eventually*)

lemma *frequently-parametric* [*transfer-rule*]: ((*A* ===> (=)) ===> *rel-filter A* ===> (=)) *frequently frequently*
  unfolding *frequently-def*[*abs-def*] **by** *transfer-prover*

lemma *is-filter-parametric* [*transfer-rule*]:
  assumes [*transfer-rule*]: *bi-total A*
  assumes [*transfer-rule*]: *bi-unique A*
  shows (((*A* ===> (=)) ===> (=)) ===> (=)) *is-filter is-filter*
  unfolding *is-filter-def* **by** *transfer-prover*

lemma *top-filter-parametric* [*transfer-rule*]: *rel-filter A top top* **if** *bi-total A*
proof
  let *?Z* = *principal* {(*x*, *y*). *A x y*}
  show ∀$_F$ (*x*, *y*) *in ?Z*. *A x y* **by**(*simp add*: *eventually-principal*)
  show *map-filter-on* {(*x*, *y*). *A x y*} *fst ?Z* = *top map-filter-on* {(*x*, *y*). *A x y*} *snd ?Z* = *top*
   using *that* **by**(*auto simp add*: *filter-eq-iff eventually-map-filter-on eventually-principal bi-total-def*)
qed

lemma *bot-filter-parametric* [*transfer-rule*]: *rel-filter A bot bot*
proof
  show ∀$_F$ (*x*, *y*) *in bot*. *A x y* **by** *simp*
  show *map-filter-on* {(*x*, *y*). *A x y*} *fst bot* = *bot map-filter-on* {(*x*, *y*). *A x y*} *snd bot* = *bot*
    by(*simp-all add*: *filter-eq-iff eventually-map-filter-on*)
qed

lemma *principal-parametric* [*transfer-rule*]: (*rel-set A* ===> *rel-filter A*) *principal principal*
proof(*rule rel-funI rel-filter.intros*)+
  fix *S S*′

assume ∗: *rel-set A S S′*
define *SS′* **where** *SS′* = *S* × *S′* ∩ {(*x*, *y*). *A x y*}
have *SS′*: *SS′* ⊆ {(*x*, *y*). *A x y*} **and** [*simp*]: *S* = *fst* ‘ *SS′ S′* = *snd* ‘ *SS′*
using ∗ **by**(*auto 4 3 dest*: *rel-setD1 rel-setD2 intro*: *rev-image-eqI simp add*: *SS′-def*)
let *?Z* = *principal SS′*
show ∀ *F* (*x*, *y*) *in ?Z*. *A x y* **using** *SS′* **by**(*auto simp add*: *eventually-principal*)
then show *map-filter-on* {(*x*, *y*). *A x y*} *fst ?Z* = *principal S*
and *map-filter-on* {(*x*, *y*). *A x y*} *snd ?Z* = *principal S′*
by(*auto simp add*: *filter-eq-iff eventually-map-filter-on eventually-principal*)
qed

lemma *sup-filter-parametric* [*transfer-rule*]:
(*rel-filter A* ===> *rel-filter A* ===> *rel-filter A*) *sup sup*
proof(*intro rel-funI*; *elim rel-filter.cases*; *hypsubst*)
show *rel-filter A*
(*map-filter-on* {(*x*, *y*). *A x y*} *fst FG* ⊔ *map-filter-on* {(*x*, *y*). *A x y*} *fst FG′*)
(*map-filter-on* {(*x*, *y*). *A x y*} *snd FG* ⊔ *map-filter-on* {(*x*, *y*). *A x y*} *snd FG′*)
(**is** *rel-filter* - (*sup ?F ?G*) (*sup ?F′ ?G′*))
if ∀ *F* (*x*, *y*) *in FG*. *A x y* ∀ *F* (*x*, *y*) *in FG′*. *A x y* **for** *FG FG′*
proof
let *?Z* = *sup FG FG′*
show ∀ *F* (*x*, *y*) *in ?Z*. *A x y* **by**(*simp add*: *eventually-sup that*)
then show *map-filter-on* {(*x*, *y*). *A x y*} *fst ?Z* = *sup ?F ?G*
and *map-filter-on* {(*x*, *y*). *A x y*} *snd ?Z* = *sup ?F′ ?G′*
by(*simp-all add*: *filter-eq-iff eventually-map-filter-on eventually-sup*)
qed
qed

lemma *Sup-filter-parametric* [*transfer-rule*]: (*rel-set* (*rel-filter A*) ===> *rel-filter A*) *Sup Sup*
proof(*rule rel-funI*)
fix *S S′*
define *SS′* **where** *SS′* = *S* × *S′* ∩ {(*F*, *G*). *rel-filter A F G*}
assume *rel-set* (*rel-filter A*) *S S′*
then have *SS′*: *SS′* ⊆ {(*F*, *G*). *rel-filter A F G*} **and** [*simp*]: *S* = *fst* ‘ *SS′ S′* = *snd* ‘ *SS′*
by(*auto 4 3 dest*: *rel-setD1 rel-setD2 intro*: *rev-image-eqI simp add*: *SS′-def*)
from *SS′* **obtain** *Z* **where** *Z*: ⋀*F G*. (*F*, *G*) ∈ *SS′* ⟹
(∀ *F* (*x*, *y*) *in Z F G*. *A x y*) ∧
id F = *map-filter-on* {(*x*, *y*). *A x y*} *fst* (*Z F G*) ∧
id G = *map-filter-on* {(*x*, *y*). *A x y*} *snd* (*Z F G*)
unfolding *rel-filter.simps* **by** *atomize-elim*((*rule choice allI*)+; *auto*)
have *id*: *eventually P F* = *eventually P* (*id F*) *eventually Q G* = *eventually Q* (*id G*)
if (*F*, *G*) ∈ *SS′* **for** *P Q F G* **by** *simp-all*
show *rel-filter A* (*Sup S*) (*Sup S′*)
proof
let *?Z* = ⨆(*F*, *G*)∈*SS′*. *Z F G*

show $*$: $\forall_F$ (x, y) *in ?Z*. *A x y* **using** *Z* **by**(*auto simp add*: *eventually-Sup*)
show *map-filter-on* {(x, y). *A x y*} *fst ?Z* = *Sup S map-filter-on* {(x, y). *A x y*} *snd ?Z* = *Sup S'*
unfolding *filter-eq-iff*
by(*auto 4 4 simp add*: *id eventually-Sup eventually-map-filter-on* $*$[*simplified eventually-Sup*] *simp del*: *id-apply dest*: *Z*)
qed
qed

context
fixes *A* :: *'a* ⇒ *'b* ⇒ *bool*
assumes [*transfer-rule*]: *bi-unique A*
begin

lemma *le-filter-parametric* [*transfer-rule*]:
(*rel-filter A* ===> *rel-filter A* ===> (=)) (≤) (≤)
unfolding *le-filter-def*[*abs-def*] **by** *transfer-prover*

lemma *less-filter-parametric* [*transfer-rule*]:
(*rel-filter A* ===> *rel-filter A* ===> (=)) (<) (<)
unfolding *less-filter-def*[*abs-def*] **by** *transfer-prover*

context
assumes [*transfer-rule*]: *bi-total A*
begin

lemma *Inf-filter-parametric* [*transfer-rule*]:
(*rel-set* (*rel-filter A*) ===> *rel-filter A*) *Inf Inf*
unfolding *Inf-filter-def*[*abs-def*] **by** *transfer-prover*

lemma *inf-filter-parametric* [*transfer-rule*]:
(*rel-filter A* ===> *rel-filter A* ===> *rel-filter A*) *inf inf*
proof(*intro rel-funI*)+
fix *F F' G G'*
assume [*transfer-rule*]: *rel-filter A F F' rel-filter A G G'*
have *rel-filter A* (*Inf* {*F*, *G*}) (*Inf* {*F'*, *G'*}) **by** *transfer-prover*
thus *rel-filter A* (*inf F G*) (*inf F' G'*) **by** *simp*
qed

end

end

end

context
includes *lifting-syntax*
begin

lemma *prod-filter-parametric* [*transfer-rule*]:
(*rel-filter R* ===> *rel-filter S* ===> *rel-filter* (*rel-prod R S*)) *prod-filter prod-filter*
proof(*intro rel-funI*; *elim rel-filter.cases*; *hypsubst*)
fix *F G*
assume *F*: $\forall_F$ (*x*, *y*) *in F*. *R x y* **and** *G*: $\forall_F$ (*x*, *y*) *in G*. *S x y*
show *rel-filter* (*rel-prod R S*)
(*map-filter-on* {(*x*, *y*). *R x y*} *fst F* $\times_F$ *map-filter-on* {(*x*, *y*). *S x y*} *fst G*)
(*map-filter-on* {(*x*, *y*). *R x y*} *snd F* $\times_F$ *map-filter-on* {(*x*, *y*). *S x y*} *snd G*)
(**is** *rel-filter ?RS ?F ?G*)
proof
let *?Z* = *filtermap* (λ((*a*, *b*), (*a'*, *b'*)). ((*a*, *a'*), (*b*, *b'*))) (*prod-filter F G*)
show ∗: $\forall_F$ (*x*, *y*) *in ?Z*. *rel-prod R S x y* **using** *F G*
by(*auto simp add*: *eventually-filtermap split-beta eventually-prod-filter*)
show *map-filter-on* {(*x*, *y*). *?RS x y*} *fst ?Z* = *?F*
using *F G*
apply(*clarsimp simp add*: *filter-eq-iff eventually-map-filter-on* ∗)
apply(*simp add*: *eventually-filtermap split-beta eventually-prod-filter*)
apply(*subst eventually-map-filter-on*; *simp*)+
apply(*rule iffI*; *clarsimp*)
subgoal for *P P' P''*
apply(*rule exI*[**where** *x*=λ*a*. ∃ *b*. *P'* (*a*, *b*) ∧ *R a b*]; *rule conjI*)
subgoal by(*fastforce elim*: *eventually-rev-mp eventually-mono*)
subgoal
by(*rule exI*[**where** *x*=λ*a*. ∃ *b*. *P''* (*a*, *b*) ∧ *S a b*])(*fastforce elim*: *eventually-rev-mp eventually-mono*)
done
subgoal by *fastforce*
done
show *map-filter-on* {(*x*, *y*). *?RS x y*} *snd ?Z* = *?G*
using *F G*
apply(*clarsimp simp add*: *filter-eq-iff eventually-map-filter-on* ∗)
apply(*simp add*: *eventually-filtermap split-beta eventually-prod-filter*)
apply(*subst eventually-map-filter-on*; *simp*)+
apply(*rule iffI*; *clarsimp*)
subgoal for *P P' P''*
apply(*rule exI*[**where** *x*=λ*b*. ∃ *a*. *P'* (*a*, *b*) ∧ *R a b*]; *rule conjI*)
subgoal by(*fastforce elim*: *eventually-rev-mp eventually-mono*)
subgoal
by(*rule exI*[**where** *x*=λ*b*. ∃ *a*. *P''* (*a*, *b*) ∧ *S a b*])(*fastforce elim*: *eventually-rev-mp eventually-mono*)
done
subgoal by *fastforce*
done
qed
qed

end

Code generation for filters

definition *abstract-filter* :: (*unit* ⇒ *'a filter*) ⇒ *'a filter*
  where [*simp*]: *abstract-filter f* = *f* ()

code-datatype *principal abstract-filter*

hide-const (**open**) *abstract-filter*

declare *filterlim-principal* [*code*]
declare *principal-prod-principal* [*code*]
declare *filtermap-principal* [*code*]
declare *filtercomap-principal* [*code*]
declare *eventually-principal* [*code*]
declare *inf-principal* [*code*]
declare *sup-principal* [*code*]
declare *principal-le-iff* [*code*]

lemma *Rep-filter-iff-eventually* [*simp*, *code*]:
  Rep-filter F P ⟷ *eventually P F*
  by (*simp add*: *eventually-def*)

lemma *bot-eq-principal-empty* [*code*]:
  bot = *principal* {}
  by *simp*

lemma *top-eq-principal-UNIV* [*code*]:
  top = *principal UNIV*
  by *simp*

instantiation *filter* :: (*equal*) *equal*
begin

definition *equal-filter* :: *'a filter* ⇒ *'a filter* ⇒ *bool*
  where *equal-filter F F'* ⟷ *F* = *F'*

lemma *equal-filter* [*code*]:
  HOL.equal (*principal A*) (*principal B*) ⟷ *A* = *B*
  by (*simp add*: *equal-filter-def*)

instance
  by *standard* (*simp add*: *equal-filter-def*)

end

end

91 Conditionally-complete Lattices

theory *Conditionally-Complete-Lattices*
imports *Finite-Set Lattices-Big Set-Interval*

begin

locale *preordering-bdd* = *preordering*
begin

definition *bdd* :: ‹*'a set* ⇒ *bool*›
  where *unfold*: ‹*bdd A* ⟷ (∃ *M*. ∀ *x* ∈ *A*. *x* ≤ *M*)›

lemma *empty* [*simp*, *intro*]:
  ‹*bdd* {}›
  by (*simp add*: *unfold*)

lemma *I* [*intro*]:
  ‹*bdd A*› **if** ‹⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *M*›
  using *that* **by** (*auto simp add*: *unfold*)

lemma *E*:
  assumes ‹*bdd A*›
  obtains *M* **where** ‹⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *M*›
  using *assms that* **by** (*auto simp add*: *unfold*)

lemma *I2*:
  ‹*bdd* (*f* ‘ *A*)› **if** ‹⋀*x*. *x* ∈ *A* ⟹ *f x* ≤ *M*›
  using *that* **by** (*auto simp add*: *unfold*)

lemma *mono*:
  ‹*bdd A*› **if** ‹*bdd B*› ‹*A* ⊆ *B*›
  using *that* **by** (*auto simp add*: *unfold*)

lemma *Int1* [*simp*]:
  ‹*bdd* (*A* ∩ *B*)› **if** ‹*bdd A*›
  using *mono that* **by** *auto*

lemma *Int2* [*simp*]:
  ‹*bdd* (*A* ∩ *B*)› **if** ‹*bdd B*›
  using *mono that* **by** *auto*

end

91.1 Preorders

context *preorder*
begin

sublocale *bdd-above*: *preordering-bdd* ‹(≤)› ‹(<)›
  defines *bdd-above-primitive-def*: *bdd-above* = *bdd-above.bdd* **..**

sublocale *bdd-below*: *preordering-bdd* ‹(≥)› ‹(>)›
  defines *bdd-below-primitive-def*: *bdd-below* = *bdd-below.bdd* **..**

lemma *bdd-above-def*: ‹*bdd-above* $A \longleftrightarrow (\exists M.\ \forall x \in A.\ x \leq M)$›
by (*fact bdd-above.unfold*)

lemma *bdd-below-def*: ‹*bdd-below* $A \longleftrightarrow (\exists M.\ \forall x \in A.\ M \leq x)$›
by (*fact bdd-below.unfold*)

lemma *bdd-aboveI*: $(\bigwedge x.\ x \in A \Longrightarrow x \leq M) \Longrightarrow$ *bdd-above* A
by (*fact bdd-above.I*)

lemma *bdd-belowI*: $(\bigwedge x.\ x \in A \Longrightarrow m \leq x) \Longrightarrow$ *bdd-below* A
by (*fact bdd-below.I*)

lemma *bdd-aboveI2*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \leq M) \Longrightarrow$ *bdd-above* $(f`A)$
by (*fact bdd-above.I2*)

lemma *bdd-belowI2*: $(\bigwedge x.\ x \in A \Longrightarrow m \leq f\ x) \Longrightarrow$ *bdd-below* $(f`A)$
by (*fact bdd-below.I2*)

lemma *bdd-above-empty*: *bdd-above* $\{\}$
by (*fact bdd-above.empty*)

lemma *bdd-below-empty*: *bdd-below* $\{\}$
by (*fact bdd-below.empty*)

lemma *bdd-above-mono*: *bdd-above* $B \Longrightarrow A \subseteq B \Longrightarrow$ *bdd-above* A
by (*fact bdd-above.mono*)

lemma *bdd-below-mono*: *bdd-below* $B \Longrightarrow A \subseteq B \Longrightarrow$ *bdd-below* A
by (*fact bdd-below.mono*)

lemma *bdd-above-Int1*: *bdd-above* $A \Longrightarrow$ *bdd-above* $(A \cap B)$
by (*fact bdd-above.Int1*)

lemma *bdd-above-Int2*: *bdd-above* $B \Longrightarrow$ *bdd-above* $(A \cap B)$
by (*fact bdd-above.Int2*)

lemma *bdd-below-Int1*: *bdd-below* $A \Longrightarrow$ *bdd-below* $(A \cap B)$
by (*fact bdd-below.Int1*)

lemma *bdd-below-Int2*: *bdd-below* $B \Longrightarrow$ *bdd-below* $(A \cap B)$
by (*fact bdd-below.Int2*)

lemma *bdd-above-Ioo* [*simp*, *intro*]: *bdd-above* $\{a <..< b\}$
by (*auto simp add*: *bdd-above-def intro*!: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-above-Ico* [*simp*, *intro*]: *bdd-above* $\{a ..< b\}$
by (*auto simp add*: *bdd-above-def intro*!: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-above-Iio* [*simp*, *intro*]: *bdd-above* {..< *b*}
by (*auto simp add*: *bdd-above-def intro*: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-above-Ioc* [*simp*, *intro*]: *bdd-above* {*a* <.. *b*}
by (*auto simp add*: *bdd-above-def intro*: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-above-Icc* [*simp*, *intro*]: *bdd-above* {*a* .. *b*}
by (*auto simp add*: *bdd-above-def intro*: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-above-Iic* [*simp*, *intro*]: *bdd-above* {.. *b*}
by (*auto simp add*: *bdd-above-def intro*: *exI*[*of* - *b*] *less-imp-le*)

lemma *bdd-below-Ioo* [*simp*, *intro*]: *bdd-below* {*a* <..< *b*}
by (*auto simp add*: *bdd-below-def intro*!: *exI*[*of* - *a*] *less-imp-le*)

lemma *bdd-below-Ioc* [*simp*, *intro*]: *bdd-below* {*a* <.. *b*}
by (*auto simp add*: *bdd-below-def intro*!: *exI*[*of* - *a*] *less-imp-le*)

lemma *bdd-below-Ioi* [*simp*, *intro*]: *bdd-below* {*a* <..}
by (*auto simp add*: *bdd-below-def intro*: *exI*[*of* - *a*] *less-imp-le*)

lemma *bdd-below-Ico* [*simp*, *intro*]: *bdd-below* {*a* ..< *b*}
by (*auto simp add*: *bdd-below-def intro*: *exI*[*of* - *a*] *less-imp-le*)

lemma *bdd-below-Icc* [*simp*, *intro*]: *bdd-below* {*a* .. *b*}
by (*auto simp add*: *bdd-below-def intro*: *exI*[*of* - *a*] *less-imp-le*)

lemma *bdd-below-Ici* [*simp*, *intro*]: *bdd-below* {*a* ..}
by (*auto simp add*: *bdd-below-def intro*: *exI*[*of* - *a*] *less-imp-le*)

end

context *order-top*
begin

lemma *bdd-above-top* [*simp*, *intro*!]: *bdd-above A*
by (*rule bdd-aboveI* [*of* - *top*]) *simp*

end

context *order-bot*
begin

lemma *bdd-below-bot* [*simp*, *intro*!]: *bdd-below A*
by (*rule bdd-belowI* [*of* - *bot*]) *simp*

end

lemma *bdd-above-image-mono*: *mono f* ⟹ *bdd-above A* ⟹ *bdd-above* (*f*‘*A*)

by (*auto simp*: *bdd-above-def mono-def*)

lemma *bdd-below-image-mono*: *mono f* $\Longrightarrow$ *bdd-below A* $\Longrightarrow$ *bdd-below* (*f*‘*A*)
by (*auto simp*: *bdd-below-def mono-def*)

lemma *bdd-above-image-antimono*: *antimono f* $\Longrightarrow$ *bdd-below A* $\Longrightarrow$ *bdd-above* (*f*‘*A*)
by (*auto simp*: *bdd-above-def bdd-below-def antimono-def*)

lemma *bdd-below-image-antimono*: *antimono f* $\Longrightarrow$ *bdd-above A* $\Longrightarrow$ *bdd-below* (*f*‘*A*)
by (*auto simp*: *bdd-above-def bdd-below-def antimono-def*)

lemma
fixes *X* :: *'a*::*ordered-ab-group-add set*
shows *bdd-above-uminus*[*simp*]: *bdd-above* (*uminus* ‘ *X*) $\longleftrightarrow$ *bdd-below X*
and *bdd-below-uminus*[*simp*]: *bdd-below* (*uminus* ‘ *X*) $\longleftrightarrow$ *bdd-above X*
using *bdd-above-image-antimono*[*of uminus X*] *bdd-below-image-antimono*[*of uminus uminus*‘*X*]
using *bdd-below-image-antimono*[*of uminus X*] *bdd-above-image-antimono*[*of uminus uminus*‘*X*]
by (*auto simp*: *antimono-def image-image*)

91.2 Lattices

context *lattice*
begin

lemma *bdd-above-insert* [*simp*]: *bdd-above* (*insert a A*) = *bdd-above A*
by (*auto simp*: *bdd-above-def intro*: *le-supI2 sup-ge1*)

lemma *bdd-below-insert* [*simp*]: *bdd-below* (*insert a A*) = *bdd-below A*
by (*auto simp*: *bdd-below-def intro*: *le-infI2 inf-le1*)

lemma *bdd-finite* [*simp*]:
assumes *finite A* **shows** *bdd-above-finite*: *bdd-above A* **and** *bdd-below-finite*: *bdd-below A*
using *assms* **by** (*induct rule*: *finite-induct*, *auto*)

lemma *bdd-above-Un* [*simp*]: *bdd-above* (*A* $\cup$ *B*) = (*bdd-above A* $\wedge$ *bdd-above B*)
proof
assume *bdd-above* (*A* $\cup$ *B*)
thus *bdd-above A* $\wedge$ *bdd-above B* **unfolding** *bdd-above-def* **by** *auto*
next
assume *bdd-above A* $\wedge$ *bdd-above B*
then obtain *a b* **where** $\forall x{\in}A.\ x \leq a\ \forall x{\in}B.\ x \leq b$ **unfolding** *bdd-above-def* **by** *auto*
hence $\forall x \in A \cup B.\ x \leq sup\ a\ b$ **by** (*auto intro*: *Un-iff le-supI1 le-supI2*)
thus *bdd-above* (*A* $\cup$ *B*) **unfolding** *bdd-above-def* **..**

qed

lemma *bdd-below-Un* [*simp*]: *bdd-below* $(A \cup B)$ = (*bdd-below* $A \wedge$ *bdd-below* B)
proof
assume *bdd-below* $(A \cup B)$
thus *bdd-below* $A \wedge$ *bdd-below* B **unfolding** *bdd-below-def* **by** *auto*
next
assume *bdd-below* $A \wedge$ *bdd-below* B
then obtain a b **where** $\forall x{\in}A.\ a \leq x\ \forall x{\in}B.\ b \leq x$ **unfolding** *bdd-below-def*
by *auto*
hence $\forall x \in A \cup B.$ *inf* $a\ b \leq x$ **by** (*auto intro*: *Un-iff le-infI1 le-infI2*)
thus *bdd-below* $(A \cup B)$ **unfolding** *bdd-below-def* **..**
qed

lemma *bdd-above-image-sup*[*simp*]:
bdd-above $((\lambda x.\ sup\ (f\ x)\ (g\ x))\ `\ A) \longleftrightarrow$ *bdd-above* $(f`A) \wedge$ *bdd-above* $(g`A)$
by (*auto simp*: *bdd-above-def intro*: *le-supI1 le-supI2*)

lemma *bdd-below-image-inf*[*simp*]:
bdd-below $((\lambda x.\ inf\ (f\ x)\ (g\ x))\ `\ A) \longleftrightarrow$ *bdd-below* $(f`A) \wedge$ *bdd-below* $(g`A)$
by (*auto simp*: *bdd-below-def intro*: *le-infI1 le-infI2*)

lemma *bdd-below-UN*[*simp*]: *finite* $I \Longrightarrow$ *bdd-below* $(\bigcup i{\in}I.\ A\ i) = (\forall i \in I.$ *bdd-below* $(A\ i))$
by (*induction I rule*: *finite.induct*) *auto*

lemma *bdd-above-UN*[*simp*]: *finite* $I \Longrightarrow$ *bdd-above* $(\bigcup i{\in}I.\ A\ i) = (\forall i \in I.$ *bdd-above* $(A\ i))$
by (*induction I rule*: *finite.induct*) *auto*

end

To avoid name classes with the *complete-lattice*-class we prefix *Sup* and *Inf* in theorem names with c.

91.3 Conditionally complete lattices

class *conditionally-complete-lattice* = *lattice* + *Sup* + *Inf* +
assumes *cInf-lower*: $x \in X \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* $X \leq x$
and *cInf-greatest*: $X \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow z \leq x) \Longrightarrow z \leq$ *Inf* X
assumes *cSup-upper*: $x \in X \Longrightarrow$ *bdd-above* $X \Longrightarrow x \leq$ *Sup* X
and *cSup-least*: $X \neq \{\} \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow x \leq z) \Longrightarrow$ *Sup* $X \leq z$
begin

lemma *cSup-upper2*: $x \in X \Longrightarrow y \leq x \Longrightarrow$ *bdd-above* $X \Longrightarrow y \leq$ *Sup* X
by (*metis cSup-upper order-trans*)

lemma *cInf-lower2*: $x \in X \Longrightarrow x \leq y \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* $X \leq y$
by (*metis cInf-lower order-trans*)

lemma *cSup-mono*: $B \neq \{\} \Longrightarrow$ *bdd-above* $A \Longrightarrow (\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ b \leq a) \Longrightarrow$ *Sup* $B \leq$ *Sup* A
by (*metis cSup-least cSup-upper2*)

lemma *cInf-mono*: $B \neq \{\} \Longrightarrow$ *bdd-below* $A \Longrightarrow (\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b) \Longrightarrow$ *Inf* $A \leq$ *Inf* B
by (*metis cInf-greatest cInf-lower2*)

lemma *cSup-subset-mono*: $A \neq \{\} \Longrightarrow$ *bdd-above* $B \Longrightarrow A \subseteq B \Longrightarrow$ *Sup* $A \leq$ *Sup* B
by (*metis cSup-least cSup-upper subsetD*)

lemma *cInf-superset-mono*: $A \neq \{\} \Longrightarrow$ *bdd-below* $B \Longrightarrow A \subseteq B \Longrightarrow$ *Inf* $B \leq$ *Inf* A
by (*metis cInf-greatest cInf-lower subsetD*)

lemma *cSup-eq-maximum*: $z \in X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow x \leq z) \Longrightarrow$ *Sup* $X = z$
by (*intro order.antisym cSup-upper*[*of z X*] *cSup-least*[*of X z*]) *auto*

lemma *cInf-eq-minimum*: $z \in X \Longrightarrow (\bigwedge x.\ x \in X \Longrightarrow z \leq x) \Longrightarrow$ *Inf* $X = z$
by (*intro order.antisym cInf-lower*[*of z X*] *cInf-greatest*[*of X z*]) *auto*

lemma *cSup-le-iff*: $S \neq \{\} \Longrightarrow$ *bdd-above* $S \Longrightarrow$ *Sup* $S \leq a \longleftrightarrow (\forall\, x{\in}S.\ x \leq a)$
by (*metis order-trans cSup-upper cSup-least*)

lemma *le-cInf-iff*: $S \neq \{\} \Longrightarrow$ *bdd-below* $S \Longrightarrow a \leq$ *Inf* $S \longleftrightarrow (\forall\, x{\in}S.\ a \leq x)$
by (*metis order-trans cInf-lower cInf-greatest*)

lemma *cSup-eq-non-empty*:
assumes *1*: $X \neq \{\}$
assumes *2*: $\bigwedge x.\ x \in X \Longrightarrow x \leq a$
assumes *3*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow x \leq y) \Longrightarrow a \leq y$
shows *Sup* $X = a$
by (*intro 3 1 order.antisym cSup-least*) (*auto intro*: *2 1 cSup-upper*)

lemma *cInf-eq-non-empty*:
assumes *1*: $X \neq \{\}$
assumes *2*: $\bigwedge x.\ x \in X \Longrightarrow a \leq x$
assumes *3*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow y \leq x) \Longrightarrow y \leq a$
shows *Inf* $X = a$
by (*intro 3 1 order.antisym cInf-greatest*) (*auto intro*: *2 1 cInf-lower*)

lemma *cInf-cSup*: $S \neq \{\} \Longrightarrow$ *bdd-below* $S \Longrightarrow$ *Inf* $S =$ *Sup* $\{x.\ \forall\, s{\in}S.\ x \leq s\}$
by (*rule cInf-eq-non-empty*) (*auto intro*!: *cSup-upper cSup-least simp*: *bdd-below-def*)

lemma *cSup-cInf*: $S \neq \{\} \Longrightarrow$ *bdd-above* $S \Longrightarrow$ *Sup* $S =$ *Inf* $\{x.\ \forall\, s{\in}S.\ s \leq x\}$
by (*rule cSup-eq-non-empty*) (*auto intro*!: *cInf-lower cInf-greatest simp*: *bdd-above-def*)

lemma *cSup-insert*: $X \neq \{\} \Longrightarrow$ *bdd-above* $X \Longrightarrow$ *Sup* (*insert a X*) = *sup a* (*Sup X*)
by (*intro cSup-eq-non-empty*) (*auto intro*: *le-supI2 cSup-upper cSup-least*)

lemma *cInf-insert*: $X \neq \{\} \Longrightarrow$ *bdd-below* $X \Longrightarrow$ *Inf* (*insert a X*) = *inf a* (*Inf X*)
by (*intro cInf-eq-non-empty*) (*auto intro*: *le-infI2 cInf-lower cInf-greatest*)

lemma *cSup-singleton* [*simp*]: *Sup* $\{x\} = x$
by (*intro cSup-eq-maximum*) *auto*

lemma *cInf-singleton* [*simp*]: *Inf* $\{x\} = x$
by (*intro cInf-eq-minimum*) *auto*

lemma *cSup-insert-If*: *bdd-above* $X \Longrightarrow$ *Sup* (*insert a X*) = (*if* $X = \{\}$ *then a else sup a* (*Sup X*))
using *cSup-insert*[*of X*] **by** *simp*

lemma *cInf-insert-If*: *bdd-below* $X \Longrightarrow$ *Inf* (*insert a X*) = (*if* $X = \{\}$ *then a else inf a* (*Inf X*))
using *cInf-insert*[*of X*] **by** *simp*

lemma *le-cSup-finite*: *finite* $X \Longrightarrow x \in X \Longrightarrow x \leq$ *Sup X*
proof (*induct X arbitrary*: *x rule*: *finite-induct*)
case (*insert x X y*) **then show** *?case*
by (*cases* $X = \{\}$) (*auto simp*: *cSup-insert intro*: *le-supI2*)
qed *simp*

lemma *cInf-le-finite*: *finite* $X \Longrightarrow x \in X \Longrightarrow$ *Inf X* $\leq x$
proof (*induct X arbitrary*: *x rule*: *finite-induct*)
case (*insert x X y*) **then show** *?case*
by (*cases* $X = \{\}$) (*auto simp*: *cInf-insert intro*: *le-infI2*)
qed *simp*

lemma *cSup-eq-Sup-fin*: *finite* $X \Longrightarrow X \neq \{\} \Longrightarrow$ *Sup X* = *Sup-fin X*
by (*induct X rule*: *finite-ne-induct*) (*simp-all add*: *cSup-insert*)

lemma *cInf-eq-Inf-fin*: *finite* $X \Longrightarrow X \neq \{\} \Longrightarrow$ *Inf X* = *Inf-fin X*
by (*induct X rule*: *finite-ne-induct*) (*simp-all add*: *cInf-insert*)

lemma *cSup-atMost*[*simp*]: *Sup* $\{..x\} = x$
by (*auto intro*!: *cSup-eq-maximum*)

lemma *cSup-greaterThanAtMost*[*simp*]: $y < x \Longrightarrow$ *Sup* $\{y<..x\} = x$
by (*auto intro*!: *cSup-eq-maximum*)

lemma *cSup-atLeastAtMost*[*simp*]: $y \leq x \Longrightarrow$ *Sup* $\{y..x\} = x$
by (*auto intro*!: *cSup-eq-maximum*)

lemma *cInf-atLeast*[*simp*]: *Inf* $\{x..\} = x$

by (*auto intro*!: *cInf-eq-minimum*)

lemma *cInf-atLeastLessThan*[*simp*]: $y < x \Longrightarrow Inf\ \{y..<x\} = y$
by (*auto intro*!: *cInf-eq-minimum*)

lemma *cInf-atLeastAtMost*[*simp*]: $y \leq x \Longrightarrow Inf\ \{y..x\} = y$
by (*auto intro*!: *cInf-eq-minimum*)

lemma *cINF-lower*: *bdd-below* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow \bigsqcap (f\ `\ A) \leq f\ x$
using *cInf-lower* [*of - f ` A*] **by** *simp*

lemma *cINF-greatest*: $A \neq \{\} \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow m \leq f\ x) \Longrightarrow m \leq \bigsqcap (f\ `\ A)$
using *cInf-greatest* [*of f ` A*] **by** *auto*

lemma *cSUP-upper*: $x \in A \Longrightarrow$ *bdd-above* $(f\ `\ A) \Longrightarrow f\ x \leq \bigsqcup (f\ `\ A)$
using *cSup-upper* [*of - f ` A*] **by** *simp*

lemma *cSUP-least*: $A \neq \{\} \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq M) \Longrightarrow \bigsqcup (f\ `\ A) \leq M$
using *cSup-least* [*of f ` A*] **by** *auto*

lemma *cINF-lower2*: *bdd-below* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow f\ x \leq u \Longrightarrow \bigsqcap (f\ `\ A) \leq u$
by (*auto intro*: *cINF-lower order-trans*)

lemma *cSUP-upper2*: *bdd-above* $(f\ `\ A) \Longrightarrow x \in A \Longrightarrow u \leq f\ x \Longrightarrow u \leq \bigsqcup (f\ `\ A)$
by (*auto intro*: *cSUP-upper order-trans*)

lemma *cSUP-const* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcup x{\in}A.\ c) = c$
by (*intro order.antisym cSUP-least*) (*auto intro*: *cSUP-upper*)

lemma *cINF-const* [*simp*]: $A \neq \{\} \Longrightarrow (\bigsqcap x{\in}A.\ c) = c$
by (*intro order.antisym cINF-greatest*) (*auto intro*: *cINF-lower*)

lemma *le-cINF-iff*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f\ `\ A) \Longrightarrow u \leq \bigsqcap (f\ `\ A) \longleftrightarrow (\forall x{\in}A.\ u \leq f\ x)$
by (*metis cINF-greatest cINF-lower order-trans*)

lemma *cSUP-le-iff*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f\ `\ A) \Longrightarrow \bigsqcup (f\ `\ A) \leq u \longleftrightarrow (\forall x{\in}A.\ f\ x \leq u)$
by (*metis cSUP-least cSUP-upper order-trans*)

lemma *less-cINF-D*: *bdd-below* $(f`A) \Longrightarrow y < (\bigsqcap i{\in}A.\ f\ i) \Longrightarrow i \in A \Longrightarrow y < f\ i$
by (*metis cINF-lower less-le-trans*)

lemma *cSUP-lessD*: *bdd-above* $(f`A) \Longrightarrow (\bigsqcup i{\in}A.\ f\ i) < y \Longrightarrow i \in A \Longrightarrow f\ i < y$
by (*metis cSUP-upper le-less-trans*)

lemma *cINF-insert*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f\ `\ A) \Longrightarrow \bigsqcap (f\ `\ insert\ a\ A) = inf\ (f\ a)\ (\bigsqcap (f\ `\ A))$
by (*simp add*: *cInf-insert*)

lemma *cSUP-insert*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f \mathbin{`} A) \Longrightarrow \bigsqcup (f \mathbin{`} \mathit{insert}\ a\ A) = \mathit{sup}\ (f\ a)\ (\bigsqcup (f \mathbin{`} A))$
by (*simp add*: *cSup-insert*)

lemma *cINF-mono*: $B \neq \{\} \Longrightarrow$ *bdd-below* $(f \mathbin{`} A) \Longrightarrow (\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow \bigsqcap (f \mathbin{`} A) \leq \bigsqcap (g \mathbin{`} B)$
using *cInf-mono* [*of g* ‘ *B f* ‘ *A*] **by** *auto*

lemma *cSUP-mono*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(g \mathbin{`} B) \Longrightarrow (\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow \bigsqcup (f \mathbin{`} A) \leq \bigsqcup (g \mathbin{`} B)$
using *cSup-mono* [*of f* ‘ *A g* ‘ *B*] **by** *auto*

lemma *cINF-superset-mono*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(g \mathbin{`} B) \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow g\ x \leq f\ x) \Longrightarrow \bigsqcap (g \mathbin{`} B) \leq \bigsqcap (f \mathbin{`} A)$
by (*rule cINF-mono*) *auto*

lemma *cSUP-subset-mono*:
$[\![A \neq \{\};$ *bdd-above* $(g \mathbin{`} B);\ A \subseteq B;\ \bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x]\!] \Longrightarrow \bigsqcup\ (f \mathbin{`} A) \leq \bigsqcup\ (g \mathbin{`} B)$
by (*rule cSUP-mono*) *auto*

lemma *less-eq-cInf-inter*: *bdd-below* $A \Longrightarrow$ *bdd-below* $B \Longrightarrow A \cap B \neq \{\} \Longrightarrow \mathit{inf}\ (\mathit{Inf}\ A)\ (\mathit{Inf}\ B) \leq \mathit{Inf}\ (A \cap B)$
by (*metis cInf-superset-mono lattice-class.inf-sup-ord*(*1*) *le-infI1*)

lemma *cSup-inter-less-eq*: *bdd-above* $A \Longrightarrow$ *bdd-above* $B \Longrightarrow A \cap B \neq \{\} \Longrightarrow \mathit{Sup}\ (A \cap B) \leq \mathit{sup}\ (\mathit{Sup}\ A)\ (\mathit{Sup}\ B)$
by (*metis cSup-subset-mono lattice-class.inf-sup-ord*(*1*) *le-supI1*)

lemma *cInf-union-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-below* $A \Longrightarrow B \neq \{\} \Longrightarrow$ *bdd-below* $B \Longrightarrow \mathit{Inf}\ (A \cup B) = \mathit{inf}\ (\mathit{Inf}\ A)\ (\mathit{Inf}\ B)$
by (*intro order.antisym le-infI cInf-greatest cInf-lower*) (*auto intro*: *le-infI1 le-infI2 cInf-lower*)

lemma *cINF-union*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f \mathbin{`} A) \Longrightarrow B \neq \{\} \Longrightarrow$ *bdd-below* $(f \mathbin{`} B) \Longrightarrow \bigsqcap\ (f \mathbin{`} (A \cup B)) = \bigsqcap\ (f \mathbin{`} A) \sqcap \bigsqcap\ (f \mathbin{`} B)$
using *cInf-union-distrib* [*of f* ‘ *A f* ‘ *B*] **by** (*simp add*: *image-Un*)

lemma *cSup-union-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-above* $A \Longrightarrow B \neq \{\} \Longrightarrow$ *bdd-above* $B \Longrightarrow \mathit{Sup}\ (A \cup B) = \mathit{sup}\ (\mathit{Sup}\ A)\ (\mathit{Sup}\ B)$
by (*intro order.antisym le-supI cSup-least cSup-upper*) (*auto intro*: *le-supI1 le-supI2 cSup-upper*)

lemma *cSUP-union*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f \mathbin{`} A) \Longrightarrow B \neq \{\} \Longrightarrow$ *bdd-above* $(f \mathbin{`} B) \Longrightarrow \bigsqcup\ (f \mathbin{`} (A \cup B)) = \bigsqcup\ (f \mathbin{`} A) \sqcup \bigsqcup\ (f \mathbin{`} B)$
using *cSup-union-distrib* [*of f* ‘ *A f* ‘ *B*] **by** (*simp add*: *image-Un*)

lemma *cINF-inf-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow$ *bdd-below* $(g`A) \Longrightarrow \bigsqcap$

$(f \text{ ` } A) \sqcap \bigsqcap (g \text{ ` } A) = (\bigsqcap a{\in}A.\ inf\ (f\ a)\ (g\ a))$
by (*intro order.antisym le-infI cINF-greatest cINF-lower2*)
(*auto intro*: *le-infI1 le-infI2 cINF-greatest cINF-lower le-infI*)

lemma *SUP-sup-distrib*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow$ *bdd-above* $(g`A) \Longrightarrow \bigsqcup$
$(f \text{ ` } A) \sqcup \bigsqcup (g \text{ ` } A) = (\bigsqcup a{\in}A.\ sup\ (f\ a)\ (g\ a))$
by (*intro order.antisym le-supI cSUP-least cSUP-upper2*)
(*auto intro*: *le-supI1 le-supI2 cSUP-least cSUP-upper le-supI*)

lemma *cInf-le-cSup*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $A \Longrightarrow$ *bdd-below* $A \Longrightarrow$ *Inf* $A \leq$ *Sup* A
by (*auto intro*!: *cSup-upper2*[*of SOME a*. $a \in A$] *intro*: *someI cInf-lower*)

context
fixes $f :: 'a \Rightarrow 'b$::*conditionally-complete-lattice*
assumes *mono f*
begin

lemma *mono-cInf*: $[\![$*bdd-below* A; $A{\neq}\{\}]\!] \Longrightarrow f\ (Inf\ A) \leq (INF\ x{\in}A.\ f\ x)$
by (*simp add*: ‹*mono f*› *conditionally-complete-lattice-class.cINF-greatest cInf-lower monoD*)

lemma *mono-cSup*: $[\![$*bdd-above* A; $A{\neq}\{\}]\!] \Longrightarrow (SUP\ x{\in}A.\ f\ x) \leq f\ (Sup\ A)$
by (*simp add*: ‹*mono f*› *conditionally-complete-lattice-class.cSUP-least cSup-upper monoD*)

lemma *mono-cINF*: $[\![$*bdd-below* $(A`I)$; $I{\neq}\{\}]\!] \Longrightarrow f\ (INF\ i{\in}I.\ A\ i) \leq (INF\ x{\in}I.\ f\ (A\ x))$
by (*simp add*: ‹*mono f*› *conditionally-complete-lattice-class.cINF-greatest cINF-lower monoD*)

lemma *mono-cSUP*: $[\![$*bdd-above* $(A`I)$; $I{\neq}\{\}]\!] \Longrightarrow (SUP\ x{\in}I.\ f\ (A\ x)) \leq f\ (SUP\ i{\in}I.\ A\ i)$
by (*simp add*: ‹*mono f*› *conditionally-complete-lattice-class.cSUP-least cSUP-upper monoD*)

end

end

The special case of well-orderings

lemma *wellorder-InfI*:
fixes $k :: 'a$::{*wellorder*,*conditionally-complete-lattice*}
assumes $k \in A$ **shows** *Inf* $A \in A$
using *wellorder-class.LeastI* [*of* $\lambda x.\ x \in A\ k$]
by (*simp add*: *Least-le assms cInf-eq-minimum*)

lemma *wellorder-Inf-le1*:
fixes $k :: 'a$::{*wellorder*,*conditionally-complete-lattice*}

assumes $k \in A$ **shows** $Inf\ A \leq k$
by (*meson Least-le assms bdd-below.I cInf-lower*)

91.4 Complete lattices

instance *complete-lattice* $\subseteq$ *conditionally-complete-lattice*
by *standard* (*auto intro*: *Sup-upper Sup-least Inf-lower Inf-greatest*)

lemma *cSup-eq*:
fixes $a :: 'a :: \{$*conditionally-complete-lattice*, *no-bot*$\}$
assumes *upper*: $\bigwedge x.\ x \in X \Longrightarrow x \leq a$
assumes *least*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow x \leq y) \Longrightarrow a \leq y$
shows $Sup\ X = a$
proof *cases*
assume $X = \{\}$ **with** *lt-ex*[*of a*] *least* **show** *?thesis* **by** (*auto simp*: *less-le-not-le*)
qed (*intro cSup-eq-non-empty assms*)

lemma *cSup-unique*:
fixes $b :: 'a :: \{$*conditionally-complete-lattice*, *no-bot*$\}$
assumes $\bigwedge c.\ (\forall\, x \in s.\ x \leq c) \longleftrightarrow b \leq c$
shows $Sup\ s = b$
by (*metis assms cSup-eq order.refl*)

lemma *cInf-eq*:
fixes $a :: 'a :: \{$*conditionally-complete-lattice*, *no-top*$\}$
assumes *upper*: $\bigwedge x.\ x \in X \Longrightarrow a \leq x$
assumes *least*: $\bigwedge y.\ (\bigwedge x.\ x \in X \Longrightarrow y \leq x) \Longrightarrow y \leq a$
shows $Inf\ X = a$
proof *cases*
assume $X = \{\}$ **with** *gt-ex*[*of a*] *least* **show** *?thesis* **by** (*auto simp*: *less-le-not-le*)
qed (*intro cInf-eq-non-empty assms*)

lemma *cInf-unique*:
fixes $b :: 'a :: \{$*conditionally-complete-lattice*, *no-top*$\}$
assumes $\bigwedge c.\ (\forall\, x \in s.\ x \geq c) \longleftrightarrow b \geq c$
shows $Inf\ s = b$
by (*meson assms cInf-eq order.refl*)

class *conditionally-complete-linorder* = *conditionally-complete-lattice* + *linorder*
begin

lemma *less-cSup-iff*:
$X \neq \{\} \Longrightarrow$ *bdd-above* $X \Longrightarrow y < Sup\ X \longleftrightarrow (\exists\, x{\in}X.\ y < x)$
by (*rule iffI*) (*metis cSup-least not-less*, *metis cSup-upper less-le-trans*)

lemma *cInf-less-iff*: $X \neq \{\} \Longrightarrow$ *bdd-below* $X \Longrightarrow Inf\ X < y \longleftrightarrow (\exists\, x{\in}X.\ x < y)$
by (*rule iffI*) (*metis cInf-greatest not-less*, *metis cInf-lower le-less-trans*)

lemma *cINF-less-iff*: $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow (\bigsqcap i{\in}A.\ f\ i) < a \longleftrightarrow$

$(\exists\, x{\in}A.\ f\ x < a)$
using *cInf-less-iff*[*of f*‘*A*] **by** *auto*

lemma *less-cSUP-iff*: $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow a < (\bigsqcup i{\in}A.\ f\ i) \longleftrightarrow (\exists\, x{\in}A.\ a < f\ x)$
using *less-cSup-iff*[*of f*‘*A*] **by** *auto*

lemma *less-cSupE*:
assumes $y <$ *Sup* X $X \neq \{\}$ **obtains** x **where** $x \in X$ $y < x$
by (*metis cSup-least assms not-le that*)

lemma *less-cSupD*:
$X \neq \{\} \Longrightarrow z <$ *Sup* $X \Longrightarrow \exists\, x{\in}X.\ z < x$
by (*metis less-cSup-iff not-le-imp-less bdd-above-def*)

lemma *cInf-lessD*:
$X \neq \{\} \Longrightarrow$ *Inf* $X < z \Longrightarrow \exists\, x{\in}X.\ x < z$
by (*metis cInf-less-iff not-le-imp-less bdd-below-def*)

lemma *complete-interval*:
assumes $a < b$ **and** $P\ a$ **and** $\neg\ P\ b$
shows $\exists\, c.\ a \leq c \wedge c \leq b \wedge (\forall\, x.\ a \leq x \wedge x < c \longrightarrow P\ x) \wedge$
$(\forall\, d.\ (\forall\, x.\ a \leq x \wedge x < d \longrightarrow P\ x) \longrightarrow d \leq c)$
proof (*rule exI* [**where** $x =$ *Sup* $\{d.\ \forall\, x.\ a \leq x \wedge x < d \longrightarrow P\ x\}$], *safe*)
show $a \leq$ *Sup* $\{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\}$
by (*rule cSup-upper*, *auto simp*: *bdd-above-def*)
(*metis* ‹$a < b$› ‹$\neg\ P\ b$› *linear less-le*)
next
show *Sup* $\{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\} \leq b$
by (*rule cSup-least*)
(*use* ‹$a{<}b$› ‹$\neg\ P\ b$› **in** ‹*auto simp add*: *less-le-not-le*›)
next
fix x
assume *x*: $a \leq x$ **and** *lt*: $x <$ *Sup* $\{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\}$
show $P\ x$
by (*rule less-cSupE* [*OF lt*]) (*use less-le-not-le x* **in** ‹*auto*›)
next
fix d
assume *0*: $\forall\, x.\ a \leq x \wedge x < d \longrightarrow P\ x$
then have $d \in \{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\}$
by *auto*
moreover have *bdd-above* $\{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\}$
unfolding *bdd-above-def* **using** ‹$a{<}b$› ‹$\neg\ P\ b$› *linear*
by (*simp add*: *less-le*) *blast*
ultimately show $d \leq$ *Sup* $\{d.\ \forall\, c.\ a \leq c \wedge c < d \longrightarrow P\ c\}$
by (*auto simp*: *cSup-upper*)
qed

end

91.5 Instances

instance *complete-linorder* < *conditionally-complete-linorder*
..

lemma *cSup-eq-Max*: *finite* (*X*::'*a*::*conditionally-complete-linorder set*) ⟹ *X* ≠ {} ⟹ *Sup X* = *Max X*
using *cSup-eq-Sup-fin*[*of X*] **by** (*simp add*: *Sup-fin-Max*)

lemma *cInf-eq-Min*: *finite* (*X*::'*a*::*conditionally-complete-linorder set*) ⟹ *X* ≠ {} ⟹ *Inf X* = *Min X*
using *cInf-eq-Inf-fin*[*of X*] **by** (*simp add*: *Inf-fin-Min*)

lemma *cSup-lessThan*[*simp*]: *Sup* {..<*x*::'*a*::{*conditionally-complete-linorder*, *no-bot*, *dense-linorder*}} = *x*
by (*auto intro*!: *cSup-eq-non-empty intro*: *dense-le*)

lemma *cSup-greaterThanLessThan*[*simp*]: *y* < *x* ⟹ *Sup* {*y*<..<*x*::'*a*::{*conditionally-complete-linorder*, *dense-linorder*}} = *x*
by (*auto intro*!: *cSup-eq-non-empty intro*: *dense-le-bounded*)

lemma *cSup-atLeastLessThan*[*simp*]: *y* < *x* ⟹ *Sup* {*y*..<*x*::'*a*::{*conditionally-complete-linorder*, *dense-linorder*}} = *x*
by (*auto intro*!: *cSup-eq-non-empty intro*: *dense-le-bounded*)

lemma *cInf-greaterThan*[*simp*]: *Inf* {*x*::'*a*::{*conditionally-complete-linorder*, *no-top*, *dense-linorder*} <..} = *x*
by (*auto intro*!: *cInf-eq-non-empty intro*: *dense-ge*)

lemma *cInf-greaterThanAtMost*[*simp*]: *y* < *x* ⟹ *Inf* {*y*<..*x*::'*a*::{*conditionally-complete-linorder*, *dense-linorder*}} = *y*
by (*auto intro*!: *cInf-eq-non-empty intro*: *dense-ge-bounded*)

lemma *cInf-greaterThanLessThan*[*simp*]: *y* < *x* ⟹ *Inf* {*y*<..<*x*::'*a*::{*conditionally-complete-linorder*, *dense-linorder*}} = *y*
by (*auto intro*!: *cInf-eq-non-empty intro*: *dense-ge-bounded*)

lemma *Sup-inverse-eq-inverse-Inf*:
fixes *f*::'*b* ⇒ '*a*::{*conditionally-complete-linorder*,*linordered-field*}
assumes *bdd-above* (*range f*) *L* > *0* **and** *geL*: ⋀*x*. *f x* ≥ *L*
shows (*SUP x*. *1* / *f x*) = *1* / (*INF x*. *f x*)
proof (*rule antisym*)
have *bdd-f*: *bdd-below* (*range f*)
by (*meson assms bdd-belowI2*)
have *Inf* (*range f*) ≥ *L*
by (*simp add*: *cINF-greatest geL*)
have *bdd-invf*: *bdd-above* (*range* (λ*x*. *1* / *f x*))
proof (*rule bdd-aboveI2*)
show ⋀*x*. *1* / *f x* ≤ *1*/*L*
using *assms* **by** (*auto simp*: *divide-simps*)

```
  qed
  moreover have le-inverse-Inf: 1 / f x ≤ 1 / Inf (range f) for x
  proof −
    have Inf (range f) ≤ f x
      by (simp add: bdd-f cInf-lower)
    then show ?thesis
      using assms ‹L ≤ Inf (range f)› by (auto simp: divide-simps)
  qed
  ultimately show ∗: (SUP x. 1 / f x) ≤ 1 / Inf (range f)
    by (auto simp: cSup-le-iff cINF-lower)
  have 1 / (SUP x. 1 / f x) ≤ f y for y
  proof (cases (SUP x. 1 / f x) < 0)
    case True
    with assms show ?thesis
      by (meson less-asym' order-trans linorder-not-le zero-le-divide-1-iff)
  next
    case False
    have 1 / f y ≤ (SUP x. 1 / f x)
      by (simp add: bdd-invf cSup-upper)
    with False assms show ?thesis
      by (metis (no-types) div-by-1 divide-divide-eq-right dual-order.strict-trans1
inverse-eq-divide
          inverse-le-imp-le mult.left-neutral)
  qed
  then have 1 / (SUP x. 1 / f x) ≤ Inf (range f)
    using bdd-f by (simp add: le-cInf-iff)
  moreover have (SUP x. 1 / f x) > 0
    using assms cSUP-upper [OF - bdd-invf] by (meson UNIV-I less-le-trans
zero-less-divide-1-iff)
  ultimately show 1 / Inf (range f) ≤ (SUP t. 1 / f t)
    using ‹L ≤ Inf (range f)› ‹L>0› by (auto simp: field-simps)
qed

lemma Inf-inverse-eq-inverse-Sup:
  fixes f::'b ⇒ 'a::{conditionally-complete-linorder,linordered-field}
  assumes bdd-above (range f) L > 0 and geL: ⋀x. f x ≥ L
  shows (INF x. 1 / f x) = 1 / (SUP x. f x)
proof −
  obtain M where M>0 and M: ⋀x. f x ≤ M
    by (meson assms cSup-upper dual-order.strict-trans1 rangeI)
  have bdd: bdd-above (range (inverse ∘ f))
    using assms le-imp-inverse-le by (auto simp: bdd-above-def)
  have f x > 0 for x
    using ‹L>0› geL order-less-le-trans by blast
  then have [simp]: 1 / inverse(f x) = f x 1 / M ≤ 1 / f x for x
    using M ‹M>0› by (auto simp: divide-simps)
  show ?thesis
    using Sup-inverse-eq-inverse-Inf [OF bdd, of inverse M] ‹M>0›
    by (simp add: inverse-eq-divide)
```

qed

lemma *Inf-insert-finite*:
fixes $S :: 'a$::*conditionally-complete-linorder set*
shows *finite* $S \Longrightarrow$ *Inf* (*insert* x S) = (*if* $S = \{\}$ *then* x *else min* x (*Inf* S))
by (*simp add*: *cInf-eq-Min*)

lemma *Sup-insert-finite*:
fixes $S :: 'a$::*conditionally-complete-linorder set*
shows *finite* $S \Longrightarrow$ *Sup* (*insert* x S) = (*if* $S = \{\}$ *then* x *else max* x (*Sup* S))
by (*simp add*: *cSup-insert sup-max*)

lemma *finite-imp-less-Inf*:
fixes $a :: 'a$::*conditionally-complete-linorder*
shows $[\![$*finite* X; $x \in X$; $\bigwedge x.\ x \in X \Longrightarrow a < x]\!] \Longrightarrow a <$ *Inf* X
by (*induction* X *rule*: *finite-induct*) (*simp-all add*: *cInf-eq-Min Inf-insert-finite*)

lemma *finite-less-Inf-iff*:
fixes $a :: 'a ::$ *conditionally-complete-linorder*
shows $[\![$*finite* X; $X \neq \{\}]\!] \Longrightarrow a <$ *Inf* $X \longleftrightarrow (\forall x \in X.\ a < x)$
by (*auto simp*: *cInf-eq-Min*)

lemma *finite-imp-Sup-less*:
fixes $a :: 'a$::*conditionally-complete-linorder*
shows $[\![$*finite* X; $x \in X$; $\bigwedge x.\ x \in X \Longrightarrow a > x]\!] \Longrightarrow a >$ *Sup* X
by (*induction* X *rule*: *finite-induct*) (*simp-all add*: *cSup-eq-Max Sup-insert-finite*)

lemma *finite-Sup-less-iff*:
fixes $a :: 'a ::$ *conditionally-complete-linorder*
shows $[\![$*finite* X; $X \neq \{\}]\!] \Longrightarrow a >$ *Sup* $X \longleftrightarrow (\forall x \in X.\ a > x)$
by (*auto simp*: *cSup-eq-Max*)

class *linear-continuum* = *conditionally-complete-linorder* + *dense-linorder* +
assumes *UNIV-not-singleton*: $\exists\, a\ b::'a.\ a \neq b$
begin

lemma *ex-gt-or-lt*: $\exists\, b.\ a < b \vee b < a$
by (*metis UNIV-not-singleton neq-iff*)

end

context
fixes $f::'a \Rightarrow 'b::\{$*conditionally-complete-linorder*,*ordered-ab-group-add*$\}$
begin

lemma *bdd-above-uminus-image*: *bdd-above* $((\lambda x.\ -\ f\ x)\ `\ A) \longleftrightarrow$ *bdd-below* ($f$ ` A)
by (*metis bdd-above-uminus image-image*)

lemma *bdd-below-uminus-image*: *bdd-below* $((\lambda x.\ -\ f\ x)\ `\ A) \longleftrightarrow$ *bdd-above* $(f\ `\ A)$
by (*metis bdd-below-uminus image-image*)

lemma *uminus-cSUP*:
assumes *bdd-above* $(f\ `\ A)$ $A \neq \{\}$
shows $-\ (SUP\ x{\in}A.\ f\ x) = (INF\ x{\in}A.\ -\ f\ x)$
proof (*rule antisym*)
show $(INF\ x{\in}A.\ -\ f\ x) \leq -\ Sup\ (f\ `\ A)$
by (*metis cINF-lower cSUP-least bdd-below-uminus-image assms le-minus-iff*)
have $*$: $\bigwedge x.\ x \in A \Longrightarrow f\ x \leq Sup\ (f\ `\ A)$
by (*simp add*: *assms cSup-upper*)
then show $-\ Sup\ (f\ `\ A) \leq (INF\ x{\in}A.\ -\ f\ x)$
by (*simp add*: *assms cINF-greatest*)
qed

end

context
fixes $f{::}'a \Rightarrow 'b{::}\{$*conditionally-complete-linorder*,*ordered-ab-group-add*$\}$
begin

lemma *uminus-cINF*:
assumes *bdd-below* $(f\ `\ A)$ $A \neq \{\}$
shows $-\ (INF\ x{\in}A.\ f\ x) = (SUP\ x{\in}A.\ -\ f\ x)$
by (*metis* (*mono-tags*, *lifting*) *INF-cong uminus-cSUP assms bdd-above-uminus-image minus-equation-iff*)

lemma *Sup-add-eq*:
assumes *bdd-above* $(f\ `\ A)$ $A \neq \{\}$
shows $(SUP\ x{\in}A.\ a + f\ x) = a + (SUP\ x{\in}A.\ f\ x)$ (**is** *?L=?R*)
proof (*rule antisym*)
have *bdd*: *bdd-above* $((\lambda x.\ a + f\ x)\ `\ A)$
by (*metis assms bdd-above-image-mono image-image mono-add*)
with *assms* **show** $?L \leq ?R$
by (*simp add*: *assms cSup-le-iff cSUP-upper*)
have $\bigwedge x.\ x \in A \Longrightarrow f\ x \leq (SUP\ x{\in}A.\ a + f\ x) - a$
by (*simp add*: *bdd cSup-upper le-diff-eq*)
with $\langle A \neq \{\}\rangle$ **have** $\bigsqcup\ (f\ `\ A) \leq (\bigsqcup x{\in}A.\ a + f\ x) - a$
by (*simp add*: *cSUP-least*)
then show $?R \leq ?L$
by (*metis add.commute le-diff-eq*)
qed

lemma *Inf-add-eq*: — you don't get a shorter proof by duality
assumes *bdd-below* $(f\ `\ A)$ $A \neq \{\}$
shows $(INF\ x{\in}A.\ a + f\ x) = a + (INF\ x{\in}A.\ f\ x)$ (**is** *?L=?R*)
proof (*rule antisym*)

show *?R* ≤ *?L*
using *assms mono-add mono-cINF* **by** *blast*
have *bdd*: *bdd-below* ((λ*x*. *a* + *f x*) ' *A*)
by (*metis add-left-mono assms*(*1*) *bdd-below.E bdd-below.I2 imageI*)
with *assms* **have** ⋀*x*. *x* ∈ *A* ⟹ *f x* ≥ (*INF x*∈*A*. *a* + *f x*) − *a*
by (*simp add*: *cInf-lower diff-le-eq*)
with ‹*A* ≠ {}› **have** (⨅*x*∈*A*. *a* + *f x*) − *a* ≤ ⨅ (*f* ' *A*)
by (*simp add*: *cINF-greatest*)
with *assms* **show** *?L* ≤ *?R*
by (*metis add.commute diff-le-eq*)
qed

end

instantiation *nat* :: *conditionally-complete-linorder*
begin

definition *Sup* (*X*::*nat set*) = (*if X*={} *then 0 else Max X*)
definition *Inf* (*X*::*nat set*) = (*LEAST n*. *n* ∈ *X*)

lemma *bdd-above-nat*: *bdd-above X* ⟷ *finite* (*X*::*nat set*)
proof
assume *bdd-above X*
then obtain *z* **where** *X* ⊆ {.. *z*}
by (*auto simp*: *bdd-above-def*)
then show *finite X*
by (*rule finite-subset*) *simp*
qed *simp*

instance
proof
fix *x* :: *nat*
fix *X* :: *nat set*
show *Inf X* ≤ *x* **if** *x* ∈ *X bdd-below X*
using *that* **by** (*simp add*: *Inf-nat-def Least-le*)
show *x* ≤ *Inf X* **if** *X* ≠ {} ⋀*y*. *y* ∈ *X* ⟹ *x* ≤ *y*
using *that* **unfolding** *Inf-nat-def ex-in-conv*[*symmetric*] **by** (*rule LeastI2-ex*)
show *x* ≤ *Sup X* **if** *x* ∈ *X bdd-above X*
using *that* **by** (*auto simp add*: *Sup-nat-def bdd-above-nat*)
show *Sup X* ≤ *x* **if** *X* ≠ {} ⋀*y*. *y* ∈ *X* ⟹ *y* ≤ *x*
proof −
from *that* **have** *bdd-above X*
by (*auto simp*: *bdd-above-def*)
with *that* **show** *?thesis*
by (*simp add*: *Sup-nat-def bdd-above-nat*)
qed
qed

end

lemma *Inf-nat-def1*:
fixes *K*::*nat set*
assumes $K \neq \{\}$
shows *Inf K* $\in$ *K*
by (*auto simp add*: *Min-def Inf-nat-def*) (*meson LeastI assms bot.extremum-unique subsetI*)

lemma *Sup-nat-empty* [*simp*]: *Sup* $\{\}$ = (*0*::*nat*)
by (*auto simp add*: *Sup-nat-def*)

lemma *Sup-nat-in-if*:
fixes *x* :: *nat* **and** *X* :: *nat set*
assumes $X \neq \{\}$ *bdd-above X*
shows *Sup X* $\in$ *X*
using *Sup-nat-def assms*(*1*,*2*) *bdd-above-nat* **by** *force*

instantiation *int* :: *conditionally-complete-linorder*
begin

definition *Sup* (*X*::*int set*) = (*THE x*. *x* $\in$ *X* $\wedge$ ($\forall$ *y*$\in$*X*. *y* $\leq$ *x*))
definition *Inf* (*X*::*int set*) = $-$ (*Sup* (*uminus* ‘ *X*))

instance
proof
{ fix *x* :: *int* **and** *X* :: *int set* **assume** $X \neq \{\}$ *bdd-above X*
then obtain *x y* **where** $X \subseteq \{..y\}$ *x* $\in$ *X*
by (*auto simp*: *bdd-above-def*)
then have $*$: *finite* ($X \cap \{x..y\}$) $X \cap \{x..y\} \neq \{\}$ **and** $x \leq y$
by (*auto simp*: *subset-eq*)
have $\exists!x{\in}X.\ (\forall\, y{\in}X.\ y \leq x)$
proof
{ fix *z* **assume** *z* $\in$ *X*
have $z \leq$ *Max* ($X \cap \{x..y\}$)
proof *cases*
assume $x \leq z$ **with** ‹*z* $\in$ *X*› ‹$X \subseteq \{..y\}$› $*$(*1*) **show** *?thesis*
by (*auto intro*!: *Max-ge*)
next
assume $\neg\ x \leq z$
then have $z < x$ **by** *simp*
also have $x \leq$ *Max* ($X \cap \{x..y\}$)
using ‹*x* $\in$ *X*› $*$(*1*) ‹$x \leq y$› **by** (*intro Max-ge*) *auto*
finally show *?thesis* **by** *simp*
qed }
note *le* = *this*
with *Max-in*[*OF* $*$] **show** *ex*: *Max* ($X \cap \{x..y\}$) $\in$ *X* $\wedge$ ($\forall\, z{\in}X.\ z \leq$ *Max* ($X \cap \{x..y\}$)) **by** *auto*

```
    fix z assume *: z ∈ X ∧ (∀ y∈X. y ≤ z)
    with le have z ≤ Max (X ∩ {x..y})
      by auto
    moreover have Max (X ∩ {x..y}) ≤ z
      using * ex by auto
    ultimately show z = Max (X ∩ {x..y})
      by auto
  qed
  then have Sup X ∈ X ∧ (∀ y∈X. y ≤ Sup X)
    unfolding Sup-int-def by (rule theI') }
 note Sup-int = this

 { fix x :: int and X :: int set assume x ∈ X bdd-above X then show x ≤ Sup
X
    using Sup-int[of X] by auto }
 note le-Sup = this
 { fix x :: int and X :: int set assume X ≠ {} ⋀y. y ∈ X ⟹ y ≤ x then show
Sup X ≤ x
    using Sup-int[of X] by (auto simp: bdd-above-def) }
 note Sup-le = this

 { fix x :: int and X :: int set assume x ∈ X bdd-below X then show Inf X ≤ x
    using le-Sup[of −x uminus ‘ X] by (auto simp: Inf-int-def) }
 { fix x :: int and X :: int set assume X ≠ {} ⋀y. y ∈ X ⟹ x ≤ y then show
x ≤ Inf X
    using Sup-le[of uminus ‘ X −x] by (force simp: Inf-int-def) }
qed

end

lemma Sup-int-in-if:
 fixes x :: int and X :: int set
 assumes X ≠ {} bdd-above X
 shows Sup X ∈ X
proof −
 from assms obtain x y where X ⊆ {..y} x ∈ X
   by (auto simp: bdd-above-def)
 then have *: finite (X ∩ {x..y}) X ∩ {x..y} ≠ {} and x ≤ y
   by (auto simp: subset-eq)
 have z ≤ Max (X ∩ {x..y}) if z ∈ X for z
 proof cases
   assume x ≤ z with ‹z ∈ X› ‹X ⊆ {..y}› *(1) show ?thesis
     by (auto intro!: Max-ge)
 next
   assume ¬ x ≤ z
   then have z < x by simp
   also have x ≤ Max (X ∩ {x..y})
     using ‹x ∈ X› *(1) ‹x ≤ y› by (intro Max-ge) auto
```

```
    finally show ?thesis by simp
  qed
  with Max-in[OF *] have ∃ x∈X. (∀ y∈X. y ≤ x) by blast
  then show Sup X ∈ X
    using cSup-eq-maximum by blast
qed

lemma Inf-int-in-if:
  fixes S :: int set
  assumes S ≠ {} bdd-below S
  shows Inf S ∈ S
unfolding Inf-int-def
using assms bdd-above-uminus Sup-int-in-if[of uminus ‘ S]
  imageE[of ⨅ (uminus ‘ S) uminus S]
by fastforce

lemma interval-cases:
  fixes S :: 'a :: conditionally-complete-linorder set
  assumes ivl: ⋀a b x. a ∈ S ⟹ b ∈ S ⟹ a ≤ x ⟹ x ≤ b ⟹ x ∈ S
  shows ∃ a b. S = {} ∨
    S = UNIV ∨
    S = {..<b} ∨
    S = {..b} ∨
    S = {a<..} ∨
    S = {a..} ∨
    S = {a<..<b} ∨
    S = {a<..b} ∨
    S = {a..<b} ∨
    S = {a..b}
proof −
  define lower upper where lower = {x. ∃ s∈S. s ≤ x} and upper = {x. ∃ s∈S.
x ≤ s}
  with ivl have S = lower ∩ upper
    by auto
  moreover
  have ∃ a. upper = UNIV ∨ upper = {} ∨ upper = {.. a} ∨ upper = {..< a}
  proof cases
    assume *: bdd-above S ∧ S ≠ {}
    from * have upper ⊆ {.. Sup S}
      by (auto simp: upper-def intro: cSup-upper2)
    moreover from * have {..< Sup S} ⊆ upper
      by (force simp add: less-cSup-iff upper-def subset-eq Ball-def)
    ultimately have upper = {.. Sup S} ∨ upper = {..< Sup S}
      unfolding ivl-disj-un(2)[symmetric] by auto
    then show ?thesis by auto
  next
    assume ¬ (bdd-above S ∧ S ≠ {})
    then have upper = UNIV ∨ upper = {}
      by (auto simp: upper-def bdd-above-def not-le dest: less-imp-le)
```

```
    then show ?thesis
      by auto
  qed
  moreover
  have ∃ b. lower = UNIV ∨ lower = {} ∨ lower = {b ..} ∨ lower = {b <..}
  proof cases
    assume *: bdd-below S ∧ S ≠ {}
    from * have lower ⊆ {Inf S ..}
      by (auto simp: lower-def intro: cInf-lower2)
    moreover from * have {Inf S <..} ⊆ lower
      by (force simp add: cInf-less-iff lower-def subset-eq Ball-def)
    ultimately have lower = {Inf S ..} ∨ lower = {Inf S <..}
      unfolding ivl-disj-un(1)[symmetric] by auto
    then show ?thesis by auto
  next
    assume ¬ (bdd-below S ∧ S ≠ {})
    then have lower = UNIV ∨ lower = {}
      by (auto simp: lower-def bdd-below-def not-le dest: less-imp-le)
    then show ?thesis
      by auto
  qed
  ultimately show ?thesis
    unfolding greaterThanAtMost-def greaterThanLessThan-def atLeastAtMost-def
atLeastLessThan-def
    by (metis inf-bot-left inf-bot-right inf-top.left-neutral inf-top.right-neutral)
qed

lemma cSUP-eq-cINF-D:
  fixes f :: - ⇒ 'b::conditionally-complete-lattice
  assumes eq: (⨆x∈A. f x) = (⨅x∈A. f x)
     and bdd: bdd-above (f ‘ A) bdd-below (f ‘ A)
     and a: a ∈ A
  shows f a = (⨅x∈A. f x)
proof (rule antisym)
  show f a ≤ ⨅ (f ‘ A)
    by (metis a bdd(1) eq cSUP-upper)
  show ⨅ (f ‘ A) ≤ f a
    using a bdd by (auto simp: cINF-lower)
qed

lemma cSUP-UNION:
  fixes f :: - ⇒ 'b::conditionally-complete-lattice
  assumes ne: A ≠ {} ⋀x. x ∈ A ⟹ B(x) ≠ {}
      and bdd-UN: bdd-above (⋃x∈A. f ‘ B x)
  shows (⨆z ∈ ⋃x∈A. B x. f z) = (⨆x∈A. ⨆z∈B x. f z)
proof −
  have bdd: ⋀x. x ∈ A ⟹ bdd-above (f ‘ B x)
    using bdd-UN by (meson UN-upper bdd-above-mono)
  obtain M where ⋀x y. x ∈ A ⟹ y ∈ B(x) ⟹ f y ≤ M
```

using *bdd-UN* **by** (*auto simp*: *bdd-above-def*)
then have *bdd2*: *bdd-above* ((λ*x*. ⨆*z*∈*B x*. *f z*) ‘ *A*)
unfolding *bdd-above-def* **by** (*force simp*: *bdd cSUP-le-iff ne*(*2*))
have (⨆ *z* ∈ ⋃ *x*∈*A*. *B x*. *f z*) ≤ (⨆ *x*∈*A*. ⨆ *z*∈*B x*. *f z*)
using *assms* **by** (*fastforce simp add*: *intro*!: *cSUP-least intro*: *cSUP-upper2 simp*: *bdd2 bdd*)
moreover have (⨆ *x*∈*A*. ⨆ *z*∈*B x*. *f z*) ≤ (⨆ *z* ∈ ⋃ *x*∈*A*. *B x*. *f z*)
using *assms* **by** (*fastforce simp add*: *intro*!: *cSUP-least intro*: *cSUP-upper simp*: *image-UN bdd-UN*)
ultimately show *?thesis*
by (*rule order-antisym*)
qed

lemma *cINF-UNION*:
fixes *f* :: - ⇒ *'b*::*conditionally-complete-lattice*
assumes *ne*: *A* ≠ {} ⋀*x*. *x* ∈ *A* ⟹ *B*(*x*) ≠ {}
and *bdd-UN*: *bdd-below* (⋃ *x*∈*A*. *f* ‘ *B x*)
shows (⨅ *z* ∈ ⋃ *x*∈*A*. *B x*. *f z*) = (⨅ *x*∈*A*. ⨅ *z*∈*B x*. *f z*)
proof −
have *bdd*: ⋀*x*. *x* ∈ *A* ⟹ *bdd-below* (*f* ‘ *B x*)
using *bdd-UN* **by** (*meson UN-upper bdd-below-mono*)
obtain *M* **where** ⋀*x y*. *x* ∈ *A* ⟹ *y* ∈ *B*(*x*) ⟹ *f y* ≥ *M*
using *bdd-UN* **by** (*auto simp*: *bdd-below-def*)
then have *bdd2*: *bdd-below* ((λ*x*. ⨅ *z*∈*B x*. *f z*) ‘ *A*)
unfolding *bdd-below-def* **by** (*force simp*: *bdd le-cINF-iff ne*(*2*))
have (⨅ *z* ∈ ⋃ *x*∈*A*. *B x*. *f z*) ≤ (⨅ *x*∈*A*. ⨅ *z*∈*B x*. *f z*)
using *assms* **by** (*fastforce simp add*: *intro*!: *cINF-greatest intro*: *cINF-lower simp*: *bdd2 bdd*)
moreover have (⨅ *x*∈*A*. ⨅ *z*∈*B x*. *f z*) ≤ (⨅ *z* ∈ ⋃ *x*∈*A*. *B x*. *f z*)
using *assms* **by** (*fastforce simp add*: *intro*!: *cINF-greatest intro*: *cINF-lower2 simp*: *bdd bdd-UN bdd2*)
ultimately show *?thesis*
by (*rule order-antisym*)
qed

lemma *cSup-abs-le*:
fixes *S* :: (*'a*::{*linordered-idom*,*conditionally-complete-linorder*}) *set*
shows *S* ≠ {} ⟹ (⋀*x*. *x*∈*S* ⟹ |*x*| ≤ *a*) ⟹ |*Sup S*| ≤ *a*
apply (*auto simp add*: *abs-le-iff intro*: *cSup-least*)
by (*metis bdd-aboveI cSup-upper neg-le-iff-le order-trans*)

end

92 Factorial Function, Rising Factorials

theory *Factorial*
imports *Groups-List*
begin

92.1 Factorial Function

context *semiring-char-0*
begin

definition *fact* :: *nat* ⇒ *'a*
  where *fact-prod*: *fact n* = *of-nat* (∏ {*1..n*})

lemma *fact-prod-Suc*: *fact n* = *of-nat* (*prod Suc* {*0..<n*})
  unfolding *fact-prod* **using** *atLeast0LessThan prod.atLeast1-atMost-eq* **by** *auto*

lemma *fact-prod-rev*: *fact n* = *of-nat* (∏ *i* = *0..<n*. *n* − *i*)
proof −
  have *prod Suc* {*0..<n*} = ∏ {*1..n*}
    by (*simp add*: *atLeast0LessThan prod.atLeast1-atMost-eq*)
  then have *prod Suc* {*0..<n*} = *prod* ((−) (*n* + *1*)) {*1..n*}
    using *prod.atLeastAtMost-rev* [*of* λ*i*. *i 1 n*] **by** *presburger*
  then show *?thesis*
   unfolding *fact-prod-Suc* **by** (*simp add*: *atLeast0LessThan prod.atLeast1-atMost-eq*)
qed

lemma *fact-0* [*simp*]: *fact 0* = *1*
  by (*simp add*: *fact-prod*)

lemma *fact-1* [*simp*]: *fact 1* = *1*
  by (*simp add*: *fact-prod*)

lemma *fact-Suc-0* [*simp*]: *fact* (*Suc 0*) = *1*
  by (*simp add*: *fact-prod*)

lemma *fact-Suc* [*simp*]: *fact* (*Suc n*) = *of-nat* (*Suc n*) ∗ *fact n*
  by (*simp add*: *fact-prod atLeastAtMostSuc-conv algebra-simps*)

lemma *fact-2* [*simp*]: *fact 2* = *2*
  by (*simp add*: *numeral-2-eq-2*)

lemma *fact-split*: *k* ≤ *n* ⟹ *fact n* = *of-nat* (*prod Suc* {*n* − *k..<n*}) ∗ *fact* (*n* − *k*)
  by (*simp add*: *fact-prod-Suc prod.union-disjoint* [*symmetric*]
    ivl-disj-un ac-simps of-nat-mult [*symmetric*])

end

lemma *of-nat-fact* [*simp*]: *of-nat* (*fact n*) = *fact n*
  by (*simp add*: *fact-prod*)

lemma *of-int-fact* [*simp*]: *of-int* (*fact n*) = *fact n*
  by (*simp only*: *fact-prod of-int-of-nat-eq*)

lemma *fact-reduce*: *n* > *0* ⟹ *fact n* = *of-nat n* ∗ *fact* (*n* − *1*)

by (*cases n*) *auto*

lemma *fact-nonzero* [*simp*]: *fact n* ≠ (*0*::*'a*::{*semiring-char-0*,*semiring-no-zero-divisors*})
using *of-nat-0-neq* **by** (*induct n*) *auto*

lemma *fact-mono-nat*: *m* ≤ *n* ⟹ *fact m* ≤ (*fact n* :: *nat*)
by (*induct n*) (*auto simp*: *le-Suc-eq*)

lemma *fact-in-Nats*: *fact n* ∈ ℕ
by (*induct n*) *auto*

lemma *fact-in-Ints*: *fact n* ∈ ℤ
by (*induct n*) *auto*

context
assumes *SORT-CONSTRAINT*(*'a*::*linordered-semidom*)
begin

lemma *fact-mono*: *m* ≤ *n* ⟹ *fact m* ≤ (*fact n* :: *'a*)
by (*metis of-nat-fact of-nat-le-iff fact-mono-nat*)

lemma *fact-ge-1* [*simp*]: *fact n* ≥ (*1* :: *'a*)
by (*metis le0 fact-0 fact-mono*)

lemma *fact-gt-zero* [*simp*]: *fact n* > (*0* :: *'a*)
using *fact-ge-1 less-le-trans zero-less-one* **by** *blast*

lemma *fact-ge-zero* [*simp*]: *fact n* ≥ (*0* :: *'a*)
by (*simp add*: *less-imp-le*)

lemma *fact-not-neg* [*simp*]: ¬ *fact n* < (*0* :: *'a*)
by (*simp add*: *not-less-iff-gr-or-eq*)

lemma *fact-le-power*: *fact n* ≤ (*of-nat* (*n*^*n*) :: *'a*)
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have ∗: *fact n* ≤ (*of-nat* (*Suc n* ^ *n*) ::*'a*)
by (*rule order-trans*) (*simp add*: *power-mono del*: *of-nat-power*)
have *fact* (*Suc n*) = (*of-nat* (*Suc n*) ∗ *fact n* ::*'a*)
by (*simp add*: *algebra-simps*)
also have ... ≤ *of-nat* (*Suc n*) ∗ *of-nat* (*Suc n* ^ *n*)
by (*simp add*: ∗ *ordered-comm-semiring-class.comm-mult-left-mono del*: *of-nat-power*)
also have ... ≤ *of-nat* (*Suc n* ^ *Suc n*)
by (*metis of-nat-mult order-refl power-Suc*)
finally show *?case* **.**
qed

end

lemma *fact-less-mono-nat*: $0 < m \Longrightarrow m < n \Longrightarrow fact\ m < (fact\ n :: nat)$
by (*induct n*) (*auto simp*: *less-Suc-eq*)

lemma *fact-less-mono*: $0 < m \Longrightarrow m < n \Longrightarrow fact\ m < (fact\ n :: 'a::linordered\text{-}semidom)$
by (*metis of-nat-fact of-nat-less-iff fact-less-mono-nat*)

lemma *fact-ge-Suc-0-nat* [*simp*]: $fact\ n \geq Suc\ 0$
by (*metis One-nat-def fact-ge-1*)

lemma *dvd-fact*: $1 \leq m \Longrightarrow m \leq n \Longrightarrow m\ dvd\ fact\ n$
by (*induct n*) (*auto simp*: *dvdI le-Suc-eq*)

lemma *fact-ge-self*: $fact\ n \geq n$
by (*cases n = 0*) (*simp-all add*: *dvd-imp-le dvd-fact*)

lemma *fact-dvd*: $n \leq m \Longrightarrow fact\ n\ dvd\ (fact\ m :: 'a::linordered\text{-}semidom)$
by (*induct m*) (*auto simp*: *le-Suc-eq*)

lemma *fact-mod*: $m \leq n \Longrightarrow fact\ n\ mod\ (fact\ m :: 'a::\{semidom\text{-}modulo, linordered\text{-}semidom\}) = 0$
by (*simp add*: *mod-eq-0-iff-dvd fact-dvd*)

lemma *fact-eq-fact-times*:
assumes $m \geq n$
shows $fact\ m = fact\ n * \prod \{Suc\ n..m\}$
unfolding *fact-prod*
by (*metis add.commute assms le-add1 le-add-diff-inverse of-nat-id plus-1-eq-Suc prod.ub-add-nat*)

lemma *fact-div-fact*:
assumes $m \geq n$
shows $fact\ m\ div\ fact\ n = \prod \{n + 1..m\}$
by (*simp add*: *fact-eq-fact-times* [*OF assms*])

lemma *fact-num-eq-if*: $fact\ m = (if\ m = 0\ then\ 1\ else\ of\text{-}nat\ m * fact\ (m - 1))$
by (*cases m*) *auto*

lemma *fact-div-fact-le-pow*:
assumes $r \leq n$
shows $fact\ n\ div\ fact\ (n - r) \leq n\ \hat{}\ r$
proof −
have $r \leq n \Longrightarrow \prod \{n - r..n\} = (n - r) * \prod \{Suc\ (n - r)..n\}$ **for** r
by (*subst prod.insert*[*symmetric*]) (*auto simp*: *atLeastAtMost-insertL*)
with *assms* **show** *?thesis*
by (*induct r rule*: *nat.induct*) (*auto simp add*: *fact-div-fact Suc-diff-Suc mult-le-mono*)
qed

lemma *prod-Suc-fact*: *prod Suc {0..<n} = fact n*
by (*simp add*: *fact-prod-Suc*)

lemma *prod-Suc-Suc-fact*: *prod Suc {Suc 0..<n} = fact n*
proof (*cases n = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have *prod Suc {Suc 0..<n} = Suc 0 * prod Suc {Suc 0..<n}*
by *simp*
also have *... = prod Suc (insert 0 {Suc 0..<n})*
by *simp*
also have *insert 0 {Suc 0..<n} = {0..<n}*
using *False* **by** *auto*
finally show *?thesis*
by (*simp add*: *fact-prod-Suc*)
qed

lemma *fact-numeral*: *fact (numeral k) = numeral k * fact (pred-numeral k)*
— Evaluation for specific numerals
by (*metis fact-Suc numeral-eq-Suc of-nat-numeral*)

92.2 Pochhammer's symbol: generalized rising factorial

See https://en.wikipedia.org/wiki/Pochhammer_symbol.

context *comm-semiring-1*
begin

definition *pochhammer* :: *'a ⇒ nat ⇒ 'a*
where *pochhammer-prod*: *pochhammer a n = prod (λi. a + of-nat i) {0..<n}*

lemma *pochhammer-prod-rev*: *pochhammer a n = prod (λi. a + of-nat (n − i))*
{1..n}
using *prod.atLeastLessThan-rev-at-least-Suc-atMost [of λi. a + of-nat i 0 n]*
by (*simp add*: *pochhammer-prod*)

lemma *pochhammer-Suc-prod*: *pochhammer a (Suc n) = prod (λi. a + of-nat i)*
{0..n}
by (*simp add*: *pochhammer-prod atLeastLessThanSuc-atLeastAtMost*)

lemma *pochhammer-Suc-prod-rev*: *pochhammer a (Suc n) = prod (λi. a + of-nat*
(n − i)) {0..n}
using *prod.atLeast-Suc-atMost-Suc-shift*
by (*simp add*: *pochhammer-prod-rev prod.atLeast-Suc-atMost-Suc-shift del*: *prod.cl-ivl-Suc*)

lemma *pochhammer-0* [*simp*]: *pochhammer a 0 = 1*
by (*simp add*: *pochhammer-prod*)

lemma *pochhammer-1* [*simp*]: *pochhammer a 1 = a*
by (*simp add*: *pochhammer-prod lessThan-Suc*)

lemma *pochhammer-Suc0* [*simp*]: *pochhammer a (Suc 0) = a*
by (*simp add*: *pochhammer-prod lessThan-Suc*)

lemma *pochhammer-Suc*: *pochhammer a (Suc n) = pochhammer a n * (a + of-nat n)*
by (*simp add*: *pochhammer-prod atLeast0-lessThan-Suc ac-simps*)

end

lemma *pochhammer-nonneg*:
fixes *x* :: *'a* :: *linordered-semidom*
shows *x > 0 ⟹ pochhammer x n ≥ 0*
by (*induction n*) (*auto simp*: *pochhammer-Suc intro*!: *mult-nonneg-nonneg add-nonneg-nonneg*)

lemma *pochhammer-pos*:
fixes *x* :: *'a* :: *linordered-semidom*
shows *x > 0 ⟹ pochhammer x n > 0*
by (*induction n*) (*auto simp*: *pochhammer-Suc intro*!: *mult-pos-pos add-pos-nonneg*)

context *comm-semiring-1*
begin

lemma *pochhammer-of-nat*: *pochhammer (of-nat x) n = of-nat (pochhammer x n)*
by (*simp add*: *pochhammer-prod Factorial.pochhammer-prod*)

end

context *comm-ring-1*
begin

lemma *pochhammer-of-int*: *pochhammer (of-int x) n = of-int (pochhammer x n)*
by (*simp add*: *pochhammer-prod Factorial.pochhammer-prod*)

end

lemma *pochhammer-rec*: *pochhammer a (Suc n) = a * pochhammer (a + 1) n*
by (*simp add*: *pochhammer-prod prod.atLeast0-lessThan-Suc-shift ac-simps del*: *prod.op-ivl-Suc*)

lemma *pochhammer-rec'*: *pochhammer z (Suc n) = (z + of-nat n) * pochhammer z n*
by (*simp add*: *pochhammer-prod prod.atLeast0-lessThan-Suc ac-simps*)

lemma *pochhammer-fact*: *fact n = pochhammer 1 n*
by (*simp add*: *pochhammer-prod fact-prod-Suc*)

lemma *pochhammer-of-nat-eq-0-lemma*: $k > n \Longrightarrow$ *pochhammer* $(-$ (*of-nat* $n ::$ $'a::$ *idom*)) $k = 0$
by (*auto simp add*: *pochhammer-prod*)

lemma *pochhammer-of-nat-eq-0-lemma'*:
assumes *kn*: $k \leq n$
shows *pochhammer* $(-$ (*of-nat* $n ::$ $'a::\{$*idom*,*ring-char-0*$\}$)) $k \neq 0$
proof (*cases k*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc h*)
then show *?thesis*
apply (*simp add*: *pochhammer-Suc-prod*)
using *Suc kn*
apply (*auto simp add*: *algebra-simps*)
done
qed

lemma *pochhammer-of-nat-eq-0-iff*:
pochhammer $(-$ (*of-nat* $n ::$ $'a::\{$*idom*,*ring-char-0*$\}$)) $k = 0 \longleftrightarrow k > n$
(**is** *?l* = *?r*)
using *pochhammer-of-nat-eq-0-lemma*[*of n k*, **where** *?'a*=$'a$]
pochhammer-of-nat-eq-0-lemma'[*of k n*, **where** *?'a* = $'a$]
by (*auto simp add*: *not-le*[*symmetric*])

lemma *pochhammer-0-left*:
pochhammer 0 n = (*if n = 0 then 1 else 0*)
by (*induction n*) (*simp-all add*: *pochhammer-rec*)

lemma *pochhammer-eq-0-iff*: *pochhammer a n* = $(0::'a::$*field-char-0*$) \longleftrightarrow (\exists k <$ *n*. *a* = $-$ *of-nat k*)
by (*auto simp add*: *pochhammer-prod eq-neg-iff-add-eq-0*)

lemma *pochhammer-eq-0-mono*:
pochhammer a n = $(0::'a::$*field-char-0*$) \Longrightarrow m \geq n \Longrightarrow$ *pochhammer a m* = *0*
unfolding *pochhammer-eq-0-iff* **by** *auto*

lemma *pochhammer-neq-0-mono*:
pochhammer a m $\neq$ $(0::'a::$*field-char-0*$) \Longrightarrow m \geq n \Longrightarrow$ *pochhammer a n* $\neq$ *0*
unfolding *pochhammer-eq-0-iff* **by** *auto*

lemma *pochhammer-minus*:
pochhammer $(-$ *b*$)$ *k* = $((-$ *1*$)$ $\hat{}$ *k* $::$ $'a::$*comm-ring-1*$)$ $*$ *pochhammer* $(b -$ *of-nat k* + *1*$)$ *k*
proof (*cases k*)
case *0*
then show *?thesis* **by** *simp*

next
case (*Suc h*)
have *eq*: ((− *1*) ^ *Suc h* :: *'a*) = (∏ *i* = *0*..*h*. − *1*)
using *prod-constant* [**where** *A*={*0*.. *h*} **and** *y*=− *1* :: *'a*]
by *auto*
with *Suc* **show** *?thesis*
using *pochhammer-Suc-prod-rev* [*of b* − *of-nat k* + *1*]
by (*auto simp add*: *pochhammer-Suc-prod prod.distrib* [*symmetric*] *eq of-nat-diff simp del*: *prod-constant*)
qed

lemma *pochhammer-minus'*:
pochhammer (*b* − *of-nat k* + *1*) *k* = ((− *1*) ^ *k* :: *'a*::*comm-ring-1*) ∗ *pochhammer* (− *b*) *k*
by (*simp add*: *pochhammer-minus*)

lemma *pochhammer-same*: *pochhammer* (− *of-nat n*) *n* =
((− *1*) ^ *n* :: *'a*::{*semiring-char-0*,*comm-ring-1*,*semiring-no-zero-divisors*}) ∗ *fact n*
unfolding *pochhammer-minus*
by (*simp add*: *of-nat-diff pochhammer-fact*)

lemma *pochhammer-product'*: *pochhammer z* (*n* + *m*) = *pochhammer z n* ∗ *pochhammer* (*z* + *of-nat n*) *m*
proof (*induct n arbitrary*: *z*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n z*)
have *pochhammer z* (*Suc n*) ∗ *pochhammer* (*z* + *of-nat* (*Suc n*)) *m* =
z ∗ (*pochhammer* (*z* + *1*) *n* ∗ *pochhammer* (*z* + *1* + *of-nat n*) *m*)
by (*simp add*: *pochhammer-rec ac-simps*)
also note *Suc*[*symmetric*]
also have *z* ∗ *pochhammer* (*z* + *1*) (*n* + *m*) = *pochhammer z* (*Suc* (*n* + *m*))
by (*subst pochhammer-rec*) *simp*
finally show *?case*
by *simp*
qed

lemma *pochhammer-product*:
m ≤ *n* ⟹ *pochhammer z n* = *pochhammer z m* ∗ *pochhammer* (*z* + *of-nat m*) (*n* − *m*)
using *pochhammer-product'*[*of z m n* − *m*] **by** *simp*

lemma *pochhammer-times-pochhammer-half*:
fixes *z* :: *'a*::*field-char-0*
shows *pochhammer z* (*Suc n*) ∗ *pochhammer* (*z* + *1*/*2*) (*Suc n*) = (∏ *k*=*0*..*2*∗*n*+*1*. *z* + *of-nat k* / *2*)
proof (*induct n*)

```
  case 0
  then show ?case
    by (simp add: atLeast0-atMost-Suc)
next
  case (Suc n)
  define n' where n' = Suc n
  have pochhammer z (Suc n') * pochhammer (z + 1 / 2) (Suc n') =
      (pochhammer z n' * pochhammer (z + 1 / 2) n') * ((z + of-nat n') * (z +
1/2 + of-nat n'))
    (is - = - * ?A)
    by (simp-all add: pochhammer-rec' mult-ac)
  also have ?A = (z + of-nat (Suc (2 * n + 1)) / 2) * (z + of-nat (Suc (Suc (2
* n + 1))) / 2)
    (is - = ?B)
    by (simp add: field-simps n'-def)
  also note Suc[folded n'-def]
  also have (∏ k=0..2 * n + 1. z + of-nat k / 2) * ?B = (∏ k=0..2 * Suc n +
1. z + of-nat k / 2)
    by (simp add: atLeast0-atMost-Suc)
  finally show ?case
    by (simp add: n'-def)
qed

lemma pochhammer-double:
  fixes z :: 'a::field-char-0
  shows pochhammer (2 * z) (2 * n) = of-nat (2^(2*n)) * pochhammer z n *
pochhammer (z+1/2) n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have pochhammer (2 * z) (2 * (Suc n)) = pochhammer (2 * z) (2 * n) *
      (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1)
    by (simp add: pochhammer-rec' ac-simps)
  also note Suc
  also have of-nat (2 ^ (2 * n)) * pochhammer z n * pochhammer (z + 1/2) n *
       (2 * (z + of-nat n)) * (2 * (z + of-nat n) + 1) =
     of-nat (2 ^ (2 * (Suc n))) * pochhammer z (Suc n) * pochhammer (z + 1/2)
(Suc n)
    by (simp add: field-simps pochhammer-rec')
  finally show ?case .
qed

lemma fact-double:
  fact (2 * n) = (2 ^ (2 * n) * pochhammer (1 / 2) n * fact n :: 'a::field-char-0)
  using pochhammer-double[of 1/2::'a n] by (simp add: pochhammer-fact)

lemma pochhammer-absorb-comp: (r − of-nat k) * pochhammer (− r) k = r *
```

pochhammer $(-r + 1)$ *k*
(**is** *?lhs* = *?rhs*)
for *r* :: *'a*::*comm-ring-1*
proof −
have *?lhs* = − *pochhammer* (− *r*) (*Suc k*)
by (*subst pochhammer-rec'*) (*simp add*: *algebra-simps*)
also have ... = *?rhs*
by (*subst pochhammer-rec*) *simp*
finally show *?thesis* **.**
qed

92.3 Misc

lemma *fact-code* [*code*]:
fact n = (*of-nat* (*fold-atLeastAtMost-nat* ((∗)) *2 n 1*) :: *'a*::*semiring-char-0*)
proof −
have *fact n* = (*of-nat* (∏ {*1*..*n*}) :: *'a*)
by (*simp add*: *fact-prod*)
also have ∏ {*1*..*n*} = ∏ {*2*..*n*}
by (*intro prod.mono-neutral-right*) *auto*
also have ... = *fold-atLeastAtMost-nat* ((∗)) *2 n 1*
by (*simp add*: *prod-atLeastAtMost-code*)
finally show *?thesis* **.**
qed

lemma *pochhammer-code* [*code*]:
pochhammer a n =
(*if n* = *0 then 1*
else fold-atLeastAtMost-nat (λ*n acc*. (*a* + *of-nat n*) ∗ *acc*) *0* (*n* − *1*) *1*)
by (*cases n*)
(*simp-all add*: *pochhammer-prod prod-atLeastAtMost-code* [*symmetric*]
atLeastLessThanSuc-atLeastAtMost)

end

93 Binomial Coefficients, Binomial Theorem, Inclusion-exclusion Principle

theory *Binomial*
imports *Presburger Factorial*
begin

93.1 Binomial coefficients

This development is based on the work of Andy Gordon and Florian Kammueller.

Combinatorial definition

definition *binomial* :: *nat* ⇒ *nat* ⇒ *nat*
  where *binomial n k* = *card* {*K*∈*Pow* {*0*..<*n*}. *card K* = *k*}

open-bundle *binomial-syntax*
begin
notation *binomial* (**infix** ‹*choose*› *64*)
end

lemma *binomial-right-mono*:
  assumes *m* ≤ *n* **shows** *m choose k* ≤ *n choose k*
proof −
  have {*K*. *K* ⊆ {*0*..<*m*} ∧ *card K* = *k*} ⊆ {*K*. *K* ⊆ {*0*..<*n*} ∧ *card K* = *k*}
    using *assms* **by** *auto*
  then show *?thesis*
    by (*simp add*: *binomial-def card-mono*)
qed

theorem *n-subsets*:
  assumes *finite A*
  shows *card* {*B*. *B* ⊆ *A* ∧ *card B* = *k*} = *card A choose k*
proof −
  from *assms* **obtain** *f* **where** *bij*: *bij-betw f* {*0*..<*card A*} *A*
    by (*blast dest*: *ex-bij-betw-nat-finite*)
  then have [*simp*]: *card* (*f* ‘ *C*) = *card C* **if** *C* ⊆ {*0*..<*card A*} **for** *C*
    by (*meson bij-betw-imp-inj-on bij-betw-subset card-image that*)
  from *bij* **have** *bij-betw* (*image f*) (*Pow* {*0*..<*card A*}) (*Pow A*)
    by (*rule bij-betw-Pow*)
  then have *inj-on* (*image f*) (*Pow* {*0*..<*card A*})
    by (*rule bij-betw-imp-inj-on*)
  moreover have {*K*. *K* ⊆ {*0*..<*card A*} ∧ *card K* = *k*} ⊆ *Pow* {*0*..<*card A*}
    by *auto*
  ultimately have *inj-on* (*image f*) {*K*. *K* ⊆ {*0*..<*card A*} ∧ *card K* = *k*}
    by (*rule inj-on-subset*)
  then have *card* {*K*. *K* ⊆ {*0*..<*card A*} ∧ *card K* = *k*} =
      card (*image f* ‘ {*K*. *K* ⊆ {*0*..<*card A*} ∧ *card K* = *k*}) (**is** - = *card ?C*)
    by (*simp add*: *card-image*)
  also have *?C* = {*K*. *K* ⊆ *f* ‘ {*0*..<*card A*} ∧ *card K* = *k*}
    by (*auto elim*!: *subset-imageE*)
  also have *f* ‘ {*0*..<*card A*} = *A*
    by (*meson bij bij-betw-def*)
  finally show *?thesis*
    by (*simp add*: *binomial-def*)
qed

Recursive characterization

lemma *binomial-n-0* [*simp*]: *n choose 0* = *1*
proof −
  have {*K* ∈ *Pow* {*0*..<*n*}. *card K* = *0*} = {{}}
    by (*auto dest*: *finite-subset*)

then show *?thesis*
by (*simp add*: *binomial-def*)
qed

lemma *binomial-0-Suc* [*simp*]: *0 choose Suc k = 0*
by (*simp add*: *binomial-def*)

lemma *binomial-Suc-Suc* [*simp*]: *Suc n choose Suc k = (n choose k) + (n choose Suc k)*
proof −
let *?P = λn k. {K. K ⊆ {0..<n} ∧ card K = k}*
let *?Q = ?P (Suc n) (Suc k)*
have *inj*: *inj-on (insert n) (?P n k)*
by *rule* (*auto*; *metis atLeastLessThan-iff insert-iff less-irrefl subsetCE*)
have *disjoint*: *insert n ‘ ?P n k ∩ ?P n (Suc k) = {}*
by *auto*
have *?Q = {K∈?Q. n ∈ K} ∪ {K∈?Q. n ∉ K}*
by *auto*
also have *{K∈?Q. n ∈ K} = insert n ‘ ?P n k* (**is** *?A = ?B*)
proof (*rule set-eqI*)
fix *K*
have *K-finite*: *finite K* **if** *K ⊆ insert n {0..<n}*
using *that* **by** (*rule finite-subset*) *simp-all*
have *Suc-card-K*: *Suc (card K − Suc 0) = card K* **if** *n ∈ K*
and *finite K*
proof −
from ‹*n ∈ K*› **obtain** *L* **where** *K = insert n L* **and** *n ∉ L*
by (*blast elim*: *Set.set-insert*)
with *that* **show** *?thesis* **by** (*simp add*: *card.insert-remove*)
qed
show *K ∈ ?A ⟷ K ∈ ?B*
by (*subst in-image-insert-iff*)
(*auto simp add*: *card.insert-remove subset-eq-atLeast0-lessThan-finite*
Diff-subset-conv K-finite Suc-card-K)
qed
also have *{K∈?Q. n ∉ K} = ?P n (Suc k)*
by (*auto simp add*: *atLeast0-lessThan-Suc*)
finally show *?thesis* **using** *inj disjoint*
by (*simp add*: *binomial-def card-Un-disjoint card-image*)
qed

lemma *binomial-eq-0*: *n < k ⟹ n choose k = 0*
by (*auto simp add*: *binomial-def dest*: *subset-eq-atLeast0-lessThan-card*)

lemma *zero-less-binomial*: *k ≤ n ⟹ n choose k > 0*
by (*induct n k rule*: *diff-induct*) *simp-all*

lemma *binomial-eq-0-iff* [*simp*]: *n choose k = 0 ⟷ n < k*
by (*metis binomial-eq-0 less-numeral-extra*(*3*) *not-less zero-less-binomial*)

lemma *zero-less-binomial-iff* [*simp*]: *n choose k* > *0* ⟷ *k* ≤ *n*
by (*metis binomial-eq-0-iff not-less0 not-less zero-less-binomial*)

lemma *binomial-n-n* [*simp*]: *n choose n* = *1*
by (*induct n*) (*simp-all add*: *binomial-eq-0*)

lemma *binomial-Suc-n* [*simp*]: *Suc n choose n* = *Suc n*
by (*induct n*) *simp-all*

lemma *binomial-1* [*simp*]: *n choose Suc 0* = *n*
by (*induct n*) *simp-all*

lemma *choose-one*: *n choose 1* = *n* **for** *n* :: *nat*
by *simp*

lemma *choose-reduce-nat*:
0 < *n* ⟹ *0* < *k* ⟹
n choose k = ((*n* − *1*) *choose* (*k* − *1*)) + ((*n* − *1*) *choose k*)
using *binomial-Suc-Suc* [*of n* − *1 k* − *1*] **by** *simp*

lemma *Suc-times-binomial-eq*: *Suc n* ∗ (*n choose k*) = (*Suc n choose Suc k*) ∗ *Suc k*
proof (*induction n arbitrary*: *k*)
case *0*
then show *?case*
by *auto*
next
case (*Suc n*)
show *?case*
proof (*cases k*)
case (*Suc k′*)
then show *?thesis*
using *Suc.IH*
by (*auto simp add*: *add-mult-distrib add-mult-distrib2 le-Suc-eq binomial-eq-0*)
qed *auto*
qed

lemma *binomial-le-pow2*: *n choose k* ≤ *2*^*n*
proof (*induction n arbitrary*: *k*)
case *0*
then show *?case*
using *le-less less-le-trans* **by** *fastforce*
next
case (*Suc n*)
show *?case*
proof (*cases k*)
case (*Suc k′*)
then show *?thesis*

using *Suc.IH* **by** (*simp add*: *add-le-mono mult-2*)
qed *auto*
qed

The absorption property.

lemma *Suc-times-binomial*: *Suc k* ∗ (*Suc n choose Suc k*) = *Suc n* ∗ (*n choose k*)
using *Suc-times-binomial-eq* **by** *auto*

This is the well-known version of absorption, but it's harder to use because of the need to reason about division.

lemma *binomial-Suc-Suc-eq-times*: (*Suc n choose Suc k*) = (*Suc n* ∗ (*n choose k*)) *div Suc k*
by (*simp add*: *Suc-times-binomial-eq del*: *mult-Suc mult-Suc-right*)

Another version of absorption, with -1 instead of *Suc*.

lemma *times-binomial-minus1-eq*: *0* < *k* ⟹ *k* ∗ (*n choose k*) = *n* ∗ ((*n* − *1*) *choose* (*k* − *1*))
using *Suc-times-binomial-eq* [**where** *n* = *n* − *1* **and** *k* = *k* − *1*]
by (*auto split*: *nat-diff-split*)

93.2 The binomial theorem (courtesy of Tobias Nipkow):

Avigad's version, generalized to any commutative ring

theorem (**in** *comm-semiring-1*) *binomial-ring*:
(*a* + *b* :: *'a*)^*n* = (∑ *k*≤*n*. (*of-nat* (*n choose k*)) ∗ *a*^*k* ∗ *b*^(*n*−*k*))
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have *decomp*: {*0*..*n*+*1*} = {*0*} ∪ {*n* + *1*} ∪ {*1*..*n*} **and** *decomp2*: {*0*..*n*} = {*0*} ∪ {*1*..*n*}
by *auto*
have (*a* + *b*)^(*n*+*1*) = (*a* + *b*) ∗ (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n* − *k*))
using *Suc.hyps* **by** *simp*
also have ... = *a* ∗ (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n*−*k*)) +
b ∗ (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n*−*k*))
by (*rule distrib-right*)
also have ... = (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^(*k*+*1*) ∗ *b*^(*n*−*k*)) +
(∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n* − *k* + *1*))
by (*auto simp add*: *sum-distrib-left ac-simps*)
also have ... = (∑ *k*≤*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n* + *1* − *k*)) +
(∑ *k*=*1*..*n*+*1*. *of-nat* (*n choose* (*k* − *1*)) ∗ *a*^*k* ∗ *b*^(*n* + *1* − *k*))
by (*simp add*: *atMost-atLeast0 sum.shift-bounds-cl-Suc-ivl Suc-diff-le field-simps del*: *sum.cl-ivl-Suc*)
also have ... = *b*^(*n* + *1*) +
(∑ *k*=*1*..*n*. *of-nat* (*n choose k*) ∗ *a*^*k* ∗ *b*^(*n* + *1* − *k*)) + (*a*^(*n* + *1*) +

```
    (∑k=1..n. of-nat (n choose (k − 1)) * a^k * b^(n + 1 − k)))
    using sum.nat-ivl-Suc' [of 1 n λk. of-nat (n choose (k−1)) * a ^ k * b ^ (n
+ 1 − k)]
  by (simp add: sum.atLeast-Suc-atMost atMost-atLeast0)
 also have ... = a^(n + 1) + b^(n + 1) +
    (∑k=1..n. of-nat (n + 1 choose k) * a^k * b^(n + 1 − k))
  by (auto simp add: field-simps sum.distrib [symmetric] choose-reduce-nat)
 also have ... = (∑k≤n+1. of-nat (n + 1 choose k) * a^k * b^(n + 1 − k))
  using decomp by (simp add: atMost-atLeast0 field-simps)
 finally show ?case
  by simp
qed
```

Original version for the naturals.

```
corollary binomial: (a + b :: nat)^n = (∑k≤n. (of-nat (n choose k)) * a^k *
b^(n − k))
 using binomial-ring [of int a int b n]
 by (simp only: of-nat-add [symmetric] of-nat-mult [symmetric] of-nat-power [symmetric]
    of-nat-sum [symmetric] of-nat-eq-iff of-nat-id)

lemma binomial-fact-lemma: k ≤ n ⟹ fact k * fact (n − k) * (n choose k) =
fact n
proof (induct n arbitrary: k rule: nat-less-induct)
 fix n k
 assume H: ∀m<n. ∀x≤m. fact x * fact (m − x) * (m choose x) = fact m
 assume kn: k ≤ n
 let ?ths = fact k * fact (n − k) * (n choose k) = fact n
 consider n = 0 ∨ k = 0 ∨ n = k | m h where n = Suc m k = Suc h h < m
  using kn by atomize-elim presburger
 then show fact k * fact (n − k) * (n choose k) = fact n
 proof cases
  case 1
  with kn show ?thesis by auto
 next
  case 2
  note n = ‹n = Suc m›
  note k = ‹k = Suc h›
  note hm = ‹h < m›
  have mn: m < n
   using n by arith
  have hm': h ≤ m
   using hm by arith
  have km: k ≤ m
   using hm k n kn by arith
  have m − h = Suc (m − Suc h)
   using  k km hm by arith
  with km k have fact (m − h) = (m − h) * fact (m − k)
   by simp
  with n k have fact k * fact (n − k) * (n choose k) =
```

```
      k * (fact h * fact (m - h) * (m choose h)) +
      (m - h) * (fact k * fact (m - k) * (m choose k))
    by (simp add: field-simps)
  also have ... = (k + (m - h)) * fact m
    using H[rule-format, OF mn hm'] H[rule-format, OF mn km]
    by (simp add: field-simps)
  finally show ?thesis
    using k n km by simp
 qed
qed

lemma binomial-fact':
 assumes k ≤ n
 shows n choose k = fact n div (fact k * fact (n - k))
 using binomial-fact-lemma [OF assms]
 by (metis fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left)

lemma binomial-fact:
 assumes kn: k ≤ n
 shows (of-nat (n choose k) :: 'a::field-char-0) = fact n / (fact k * fact (n - k))
 using binomial-fact-lemma[OF kn]
 by (metis (mono-tags, lifting) fact-nonzero mult-eq-0-iff nonzero-mult-div-cancel-left
of-nat-fact of-nat-mult)

lemma fact-binomial:
 assumes k ≤ n
 shows fact k * of-nat (n choose k) = (fact n / fact (n - k) :: 'a::field-char-0)
 unfolding binomial-fact [OF assms] by (simp add: field-simps)

lemma binomial-fact-pow: (n choose s) * fact s ≤ n^s
proof (cases s ≤ n)
 case True
 then show ?thesis
  by (smt (verit) binomial-fact-lemma mult.assoc mult.commute fact-div-fact-le-pow
fact-nonzero nonzero-mult-div-cancel-right)
qed (simp add: binomial-eq-0)

lemma choose-two: n choose 2 = n * (n - 1) div 2
proof (cases n ≥ 2)
 case False
 then have n = 0 ∨ n = 1
  by auto
 then show ?thesis by auto
next
 case True
 define m where m = n - 2
 with True have n = m + 2
  by simp
 then have fact n = n * (n - 1) * fact (n - 2)
```

by (*simp add*: *fact-prod-Suc atLeast0-lessThan-Suc algebra-simps*)
with *True* **show** *?thesis*
by (*simp add*: *binomial-fact'*)
qed

lemma *choose-row-sum*: $(\sum k{\le}n.\ n\ choose\ k) = 2\hat{}n$
using *binomial* [*of 1 1 n*] **by** (*simp add*: *numeral-2-eq-2*)

lemma *sum-choose-lower*: $(\sum k{\le}n.\ (r{+}k)\ choose\ k) = Suc\ (r{+}n)\ choose\ n$
by (*induct n*) *auto*

lemma *sum-choose-upper*: $(\sum k{\le}n.\ k\ choose\ m) = Suc\ n\ choose\ Suc\ m$
by (*induct n*) *auto*

lemma *choose-alternating-sum*:
$n > 0 \Longrightarrow (\sum i{\le}n.\ (-1)\hat{}i * of\text{-}nat\ (n\ choose\ i)) = (0 :: 'a{::}comm\text{-}ring\text{-}1)$
using *binomial-ring*[*of* $-1 :: 'a\ 1\ n$]
by (*simp add*: *atLeast0AtMost mult-of-nat-commute zero-power*)

lemma *choose-even-sum*:
assumes $n > 0$
shows $2 * (\sum i{\le}n.\ if\ even\ i\ then\ of\text{-}nat\ (n\ choose\ i)\ else\ 0) = (2\ \hat{}\ n :: 'a{::}comm\text{-}ring\text{-}1)$
proof −
have $2\ \hat{}\ n = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i)) + (\sum i{\le}n.\ (-1)\ \hat{}\ i * of\text{-}nat\ (n\ choose\ i) :: 'a)$
using *choose-row-sum*[*of n*]
by (*simp add*: *choose-alternating-sum assms atLeast0AtMost of-nat-sum*[*symmetric*])
also have $\ldots = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) + (-1)\ \hat{}\ i * of\text{-}nat\ (n\ choose\ i))$
by (*simp add*: *sum.distrib*)
also have $\ldots = 2 * (\sum i{\le}n.\ if\ even\ i\ then\ of\text{-}nat\ (n\ choose\ i)\ else\ 0)$
by (*subst sum-distrib-left*, *intro sum.cong*) *simp-all*
finally show *?thesis* **..**
qed

lemma *choose-odd-sum*:
assumes $n > 0$
shows $2 * (\sum i{\le}n.\ if\ odd\ i\ then\ of\text{-}nat\ (n\ choose\ i)\ else\ 0) = (2\ \hat{}\ n :: 'a{::}comm\text{-}ring\text{-}1)$
proof −
have $2\ \hat{}\ n = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i)) - (\sum i{\le}n.\ (-1)\ \hat{}\ i * of\text{-}nat\ (n\ choose\ i) :: 'a)$
using *choose-row-sum*[*of n*]
by (*simp add*: *choose-alternating-sum assms atLeast0AtMost of-nat-sum*[*symmetric*])
also have $\ldots = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) - (-1)\ \hat{}\ i * of\text{-}nat\ (n\ choose\ i))$
by (*simp add*: *sum-subtractf*)
also have $\ldots = 2 * (\sum i{\le}n.\ if\ odd\ i\ then\ of\text{-}nat\ (n\ choose\ i)\ else\ 0)$
by (*subst sum-distrib-left*, *intro sum.cong*) *simp-all*
finally show *?thesis* **..**

qed

NW diagonal sum property

lemma *sum-choose-diagonal*:
assumes $m \leq n$
shows $(\sum k{\leq}m.\ (n - k)\ choose\ (m - k)) = Suc\ n\ choose\ m$
proof −
have $(\sum k{\leq}m.\ (n{-}k)\ choose\ (m - k)) = (\sum k{\leq}m.\ (n - m + k)\ choose\ k)$
using *sum.atLeastAtMost-rev* [*of* $\lambda k.\ (n - k)\ choose\ (m - k)\ 0\ m$] *assms*
by (*simp add*: *atMost-atLeast0*)
also have $\ldots = Suc\ (n - m + m)\ choose\ m$
by (*rule sum-choose-lower*)
also have $\ldots = Suc\ n\ choose\ m$
using *assms* **by** *simp*
finally show *?thesis* **.**
qed

93.3 Generalized binomial coefficients

definition *gbinomial* :: $'a{::}\{semidom\text{-}divide,semiring\text{-}char\text{-}0\} \Rightarrow nat \Rightarrow 'a$ (**infix** ‹*gchoose*› *64*)
where *gbinomial-prod-rev*: $a\ gchoose\ k = (\prod i{=}0..{<}k.\ a - of\text{-}nat\ i)\ div\ fact\ k$

lemma *gbinomial-0* [*simp*]:
$a\ gchoose\ 0 = 1$
$0\ gchoose\ (Suc\ k) = 0$
by (*simp-all add*: *gbinomial-prod-rev prod.atLeast0-lessThan-Suc-shift del*: *prod.op-ivl-Suc*)

lemma *gbinomial-Suc*: $a\ gchoose\ (Suc\ k) = prod\ (\lambda i.\ a - of\text{-}nat\ i)\ \{0..k\}\ div\ fact\ (Suc\ k)$
by (*simp add*: *gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)

lemma *gbinomial-1* [*simp*]: $a\ gchoose\ 1 = a$
by (*simp add*: *gbinomial-prod-rev lessThan-Suc*)

lemma *gbinomial-Suc0* [*simp*]: $a\ gchoose\ Suc\ 0 = a$
by (*simp add*: *gbinomial-prod-rev lessThan-Suc*)

lemma *gbinomial-0-left*: $0\ gchoose\ k = (if\ k = 0\ then\ 1\ else\ 0)$
by (*cases k*) *simp-all*

lemma *gbinomial-mult-fact*: $fact\ k * (a\ gchoose\ k) = (\prod i = 0..{<}k.\ a - of\text{-}nat\ i)$
for $a :: 'a{::}field\text{-}char\text{-}0$
by (*simp-all add*: *gbinomial-prod-rev field-simps*)

lemma *gbinomial-mult-fact'*: $(a\ gchoose\ k) * fact\ k = (\prod i = 0..{<}k.\ a - of\text{-}nat\ i)$
for $a :: 'a{::}field\text{-}char\text{-}0$
using *gbinomial-mult-fact* [*of k a*] **by** (*simp add*: *ac-simps*)

lemma *gbinomial-pochhammer*: *a gchoose k = (− 1) ^ k * pochhammer (− a) k / fact k*
for *a :: 'a::field-char-0*
proof (*cases k*)
case (*Suc k'*)
then have *a gchoose k = pochhammer (a − of-nat k') (Suc k') / ((1 + of-nat k') * fact k')*
by (*simp add*: *gbinomial-prod-rev pochhammer-prod-rev atLeastLessThanSuc-atLeastAtMost prod.atLeast-Suc-atMost-Suc-shift of-nat-diff flip*: *power-mult-distrib prod.cl-ivl-Suc*)
then show *?thesis*
by (*simp add*: *pochhammer-minus Suc*)
qed *auto*

lemma *gbinomial-pochhammer'*: *a gchoose k = pochhammer (a − of-nat k + 1) k / fact k*
for *a :: 'a::field-char-0*
proof −
have *a gchoose k = ((−1)^k * (−1)^k) * pochhammer (a − of-nat k + 1) k / fact k*
by (*simp add*: *gbinomial-pochhammer pochhammer-minus mult-ac*)
also have *(−1 :: 'a)^k * (−1)^k = 1*
by (*subst power-add* [*symmetric*]) *simp*
finally show *?thesis*
by *simp*
qed

lemma *gbinomial-binomial*: *n gchoose k = n choose k*
proof (*cases k ≤ n*)
case *False*
then have *n < k*
by (*simp add*: *not-le*)
then have *0 ∈ ((−) n) ` {0..<k}*
by *auto*
then have *prod ((−) n) {0..<k} = 0*
by (*auto intro*: *prod-zero*)
with ‹*n < k*› **show** *?thesis*
by (*simp add*: *binomial-eq-0 gbinomial-prod-rev prod-zero*)
next
case *True*
from *True* **have** *: *prod ((−) n) {0..<k} = ∏ {Suc (n − k)..n}*
by (*intro prod.reindex-bij-witness*[*of* - *λi. n − i λi. n − i*]) *auto*
from *True* **have** *n choose k = fact n div (fact k * fact (n − k))*
by (*rule binomial-fact'*)
with * **show** *?thesis*
by (*simp add*: *gbinomial-prod-rev mult.commute* [*of fact k*] *div-mult2-eq fact-div-fact*)
qed

lemma *of-nat-gbinomial*: *of-nat (n gchoose k) = (of-nat n gchoose k :: 'a::field-char-0)*
proof (*cases k ≤ n*)

case *False*
then show *?thesis*
by (*simp add*: *not-le gbinomial-binomial binomial-eq-0 gbinomial-prod-rev*)
next
case *True*
define *m* **where** *m* = *n* − *k*
with *True* **have** *n*: *n* = *m* + *k*
by *arith*
from *n* **have** *fact n* = ((∏ *i* = *0*..<*m* + *k*. *of-nat* (*m* + *k* − *i*)):: *'a*)
by (*simp add*: *fact-prod-rev*)
also have ... = ((∏ *i*∈{*0*..<*k*} ∪ {*k*..<*m* + *k*}. *of-nat* (*m* + *k* − *i*)) :: *'a*)
by (*simp add*: *ivl-disj-un*)
finally have *fact n* = (*fact m* ∗ (∏ *i* = *0*..<*k*. *of-nat m* + *of-nat k* − *of-nat i*)
:: *'a*)
using *prod.shift-bounds-nat-ivl* [*of* λ*i*. *of-nat* (*m* + *k* − *i*) :: *'a 0 k m*]
by (*simp add*: *fact-prod-rev* [*of m*] *prod.union-disjoint of-nat-diff*)
then have *fact n* / *fact* (*n* − *k*) = ((∏ *i* = *0*..<*k*. *of-nat n* − *of-nat i*) :: *'a*)
by (*simp add*: *n*)
with *True* **have** *fact k* ∗ *of-nat* (*n gchoose k*) = (*fact k* ∗ (*of-nat n gchoose k*) ::
'a)
by (*simp only*: *gbinomial-mult-fact* [*of k of-nat n*] *gbinomial-binomial* [*of n k*]
fact-binomial)
then show *?thesis*
by *simp*
qed

lemma *binomial-gbinomial*: *of-nat* (*n choose k*) = (*of-nat n gchoose k* :: *'a*::*field-char-0*)
using *gbinomial-binomial of-nat-gbinomial* **by** *auto*

setup
‹*Sign.add-const-constraint* (***const-name*** ‹*gbinomial*›, *SOME* ***typ*** ‹*'a*::*field-char-0*
⇒ *nat* ⇒ *'a*›)›

lemma *gbinomial-mult-1*:
fixes *a* :: *'a*::*field-char-0*
shows *a* ∗ (*a gchoose k*) = *of-nat k* ∗ (*a gchoose k*) + *of-nat* (*Suc k*) ∗ (*a gchoose*
(*Suc k*))
(**is** *?l* = *?r*)
proof −
have *?r* = ((− *1*) ^*k* ∗ *pochhammer* (− *a*) *k* / *fact k*) ∗ (*of-nat k* − (− *a* +
of-nat k))
unfolding *gbinomial-pochhammer pochhammer-Suc right-diff-distrib power-Suc*
by (*auto simp add*: *field-simps simp del*: *of-nat-Suc*)
also have ... = *?l*
by (*simp add*: *field-simps gbinomial-pochhammer*)
finally show *?thesis* **..**
qed

lemma *gbinomial-mult-1'*:

(*a gchoose k*) * *a* = *of-nat k* * (*a gchoose k*) + *of-nat* (*Suc k*) * (*a gchoose* (*Suc k*))
for *a* :: *'a*::*field-char-0*
by (*simp add*: *mult.commute gbinomial-mult-1*)

lemma *gbinomial-Suc-Suc*: (*a* + *1*) *gchoose* (*Suc k*) = (*a gchoose k*) + (*a gchoose* (*Suc k*))
for *a* :: *'a*::*field-char-0*
proof (*cases k*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc h*)
have *eq0*: (∏ *i*∈{*1*..*k*}. (*a* + *1*) − *of-nat i*) = (∏ *i*∈{*0*..*h*}. *a* − *of-nat i*)
proof (*rule prod.reindex-cong*)
show {*1*..*k*} = *Suc* ' {*0*..*h*}
using *Suc* **by** (*auto simp add*: *image-Suc-atMost*)
qed *auto*
have *fact* (*Suc k*) * ((*a gchoose k*) + (*a gchoose* (*Suc k*))) =
(*a gchoose Suc h*) * (*fact* (*Suc* (*Suc h*))) +
(*a gchoose Suc* (*Suc h*)) * (*fact* (*Suc* (*Suc h*)))
by (*simp add*: *Suc field-simps del*: *fact-Suc*)
also have ... =
(*a gchoose Suc h*) * *of-nat* (*Suc* (*Suc h*) * *fact* (*Suc h*)) + (∏ *i*=*0*..*Suc h*. *a* − *of-nat i*)
by (*metis atLeastLessThanSuc-atLeastAtMost fact-Suc gbinomial-mult-fact mult.commute of-nat-fact of-nat-mult*)
also have ... =
(*fact* (*Suc h*) * (*a gchoose Suc h*)) * *of-nat* (*Suc* (*Suc h*)) + (∏ *i*=*0*..*Suc h*. *a* − *of-nat i*)
by (*simp only*: *fact-Suc mult.commute mult.left-commute of-nat-fact of-nat-id of-nat-mult*)
also have ... =
of-nat (*Suc* (*Suc h*)) * (∏ *i*=*0*..*h*. *a* − *of-nat i*) + (∏ *i*=*0*..*Suc h*. *a* − *of-nat i*)
unfolding *gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost* **by** *auto*
also have ... =
(∏ *i*=*0*..*Suc h*. *a* − *of-nat i*) + (*of-nat h* * (∏ *i*=*0*..*h*. *a* − *of-nat i*) + *2* * (∏ *i*=*0*..*h*. *a* − *of-nat i*))
by (*simp add*: *field-simps*)
also have ... =
((*a gchoose Suc h*) * (*fact* (*Suc h*)) * *of-nat* (*Suc k*)) + (∏ *i*∈{*0*..*Suc h*}. *a* − *of-nat i*)
unfolding *gbinomial-mult-fact'*
by (*simp add*: *comm-semiring-class.distrib field-simps Suc atLeastLessThanSuc-atLeastAtMost*)
also have ... = (∏ *i*∈{*0*..*h*}. *a* − *of-nat i*) * (*a* + *1*)
unfolding *gbinomial-mult-fact' atLeast0-atMost-Suc*
by (*simp add*: *field-simps Suc atLeastLessThanSuc-atLeastAtMost*)
also have ... = (∏ *i*∈{*0*..*k*}. (*a* + *1*) − *of-nat i*)
using *eq0*

by (*simp add*: *Suc prod.atLeast0-atMost-Suc-shift del*: *prod.cl-ivl-Suc*)
also have ... = (*fact* (*Suc k*)) * ((*a* + *1*) *gchoose* (*Suc k*))
by (*simp only*: *gbinomial-mult-fact atLeastLessThanSuc-atLeastAtMost*)
finally show *?thesis*
using *fact-nonzero* [*of Suc k*] **by** *auto*
qed

lemma *gbinomial-reduce-nat*: *0* < *k* ⟹ *a gchoose k* = (*a*−*1 gchoose k*−*1*) + (*a*−*1 gchoose k*)
for *a* :: *'a*::*field-char-0*
by (*metis Suc-pred' diff-add-cancel gbinomial-Suc-Suc*)

lemma *gchoose-row-sum-weighted*:
(∑ *k* = *0*..*m*. (*r gchoose k*) * (*r*/*2* − *of-nat k*)) = *of-nat*(*Suc m*) / *2* * (*r gchoose* (*Suc m*))
for *r* :: *'a*::*field-char-0*
by (*induct m*) (*simp-all add*: *field-simps distrib gbinomial-mult-1*)

lemma *binomial-symmetric*:
assumes *kn*: *k* ≤ *n*
shows *n choose k* = *n choose* (*n* − *k*)
proof −
have *kn'*: *n* − *k* ≤ *n*
using *kn* **by** *arith*
from *binomial-fact-lemma*[*OF kn*] *binomial-fact-lemma*[*OF kn'*]
have *fact k* * *fact* (*n* − *k*) * (*n choose k*) = *fact* (*n* − *k*) * *fact* (*n* − (*n* − *k*)) * (*n choose* (*n* − *k*))
by *simp*
then show *?thesis*
using *kn* **by** *simp*
qed

lemma *choose-rising-sum*:
(∑ *j*≤*m*. ((*n* + *j*) *choose n*)) = ((*n* + *m* + *1*) *choose* (*n* + *1*))
(∑ *j*≤*m*. ((*n* + *j*) *choose n*)) = ((*n* + *m* + *1*) *choose m*)
proof −
show (∑ *j*≤*m*. ((*n* + *j*) *choose n*)) = ((*n* + *m* + *1*) *choose* (*n* + *1*))
by (*induct m*) *simp-all*
also have ... = (*n* + *m* + *1*) *choose m*
by (*subst binomial-symmetric*) *simp-all*
finally show (∑ *j*≤*m*. ((*n* + *j*) *choose n*)) = (*n* + *m* + *1*) *choose m* .
qed

lemma *choose-linear-sum*: (∑ *i*≤*n*. *i* * (*n choose i*)) = *n* * *2* ^ (*n* − *1*)
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)

```
  have (∑ i≤n. i * (n choose i)) = (∑ i≤Suc m. i * (Suc m choose i))
    by (simp add: Suc)
  also have ... = Suc m * 2 ^ m
   unfolding sum.atMost-Suc-shift Suc-times-binomial sum-distrib-left[symmetric]
    by (simp add: choose-row-sum)
  finally show ?thesis
    using Suc by simp
qed

lemma choose-alternating-linear-sum:
  assumes n ≠ 1
  shows (∑ i≤n. (-1)^i * of-nat i * of-nat (n choose i) :: 'a::comm-ring-1) = 0
proof (cases n)
  case 0
  then show ?thesis by simp
next
  case (Suc m)
  with assms have m > 0
    by simp
  have (∑ i≤n. (-1) ^ i * of-nat i * of-nat (n choose i) :: 'a) =
      (∑ i≤Suc m. (-1) ^ i * of-nat i * of-nat (Suc m choose i))
    by (simp add: Suc)
  also have ... = (∑ i≤m. (-1) ^ (Suc i) * of-nat (Suc i * (Suc m choose Suc
i)))
   by (simp only: sum.atMost-Suc-shift sum-distrib-left[symmetric] mult-ac of-nat-mult)
simp
  also have ... = - of-nat (Suc m) * (∑ i≤m. (-1) ^ i * of-nat (m choose i))
   proof (subst sum-distrib-left, rule sum.cong[OF refl], subst Suc-times-binomial)
    qed (simp add: algebra-simps)
  also have (∑ i≤m. (-1 :: 'a) ^ i * of-nat ((m choose i))) = 0
    using choose-alternating-sum[OF ‹m > 0›] by simp
  finally show ?thesis
    by simp
qed

lemma vandermonde: (∑ k≤r. (m choose k) * (n choose (r - k))) = (m + n)
choose r
proof (induct n arbitrary: r)
  case 0
  have (∑ k≤r. (m choose k) * (0 choose (r - k))) = (∑ k≤r. if k = r then (m
choose k) else 0)
    by (intro sum.cong) simp-all
  also have ... = m choose r
    by simp
  finally show ?case
    by simp
next
  case (Suc n r)
  show ?case
```

by (*cases r*) (*simp-all add*: *Suc* [*symmetric*] *algebra-simps sum.distrib Suc-diff-le*)
qed

lemma *choose-square-sum*: $(\sum k{\le}n.\ (n\ choose\ k)\hat{}2) = ((2{*}n)\ choose\ n)$
using *vandermonde*[*of n n n*]
by (*simp add*: *power2-eq-square mult-2 binomial-symmetric* [*symmetric*])

lemma *pochhammer-binomial-sum*:
fixes *a b* :: *'a*::*comm-ring-1*
shows *pochhammer* $(a + b)\ n = (\sum k{\le}n.\ of\text{-}nat\ (n\ choose\ k) * pochhammer\ a\ k * pochhammer\ b\ (n - k))$
proof (*induction n arbitrary*: *a b*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n a b*)
have $(\sum k{\le}Suc\ n.\ of\text{-}nat\ (Suc\ n\ choose\ k) * pochhammer\ a\ k * pochhammer\ b\ (Suc\ n - k)) =$
$(\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ (Suc\ i) * pochhammer\ b\ (n - i)) +$
$((\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ Suc\ i) * pochhammer\ a\ (Suc\ i) * pochhammer\ b\ (n - i)) +$
$pochhammer\ b\ (Suc\ n))$
by (*subst sum.atMost-Suc-shift*) (*simp add*: *ring-distribs sum.distrib*)
also have $(\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ (Suc\ i) * pochhammer\ b\ (n - i)) =$
$a * pochhammer\ ((a + 1) + b)\ n$
by (*subst Suc*) (*simp add*: *sum-distrib-left pochhammer-rec mult-ac*)
also have $(\sum i{\le}n.\ of\text{-}nat(n\ choose\ Suc\ i) * pochhammer\ a\ (Suc\ i) * pochhammer\ b\ (n{-}i)) + pochhammer\ b\ (Suc\ n)$
$= of\text{-}nat\ (n\ choose\ 0) * pochhammer\ a\ 0 * pochhammer\ b\ (Suc\ n - 0)$
$+ (\sum i = Suc\ 0..Suc\ n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ i * pochhammer\ b\ (Suc\ n - i))$
unfolding *sum.shift-bounds-cl-Suc-ivl* **by** (*simp add*: *atLeast0AtMost*)
also have $\ldots = (\sum i{=}0..Suc\ n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ i * pochhammer\ b\ (Suc\ n - i))$
by (*simp add*: *sum.atLeast-Suc-atMost*)
also have $\ldots = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ i * pochhammer\ b\ (Suc\ n - i))$
using *Suc* **by** (*intro sum.mono-neutral-right*) (*auto simp*: *not-le binomial-eq-0*)
also have $\ldots = (\sum i{\le}n.\ of\text{-}nat\ (n\ choose\ i) * pochhammer\ a\ i * pochhammer\ b\ (Suc\ (n - i)))$
by (*simp add*: *Suc-diff-le*)
also have $\ldots = b * pochhammer\ (a + (b + 1))\ n$
by (*subst Suc*) (*simp add*: *sum-distrib-left mult-ac pochhammer-rec*)
also have $a * pochhammer\ ((a + 1) + b)\ n + b * pochhammer\ (a + (b + 1))\ n =$
$pochhammer\ (a + b)\ (Suc\ n)$
by (*simp add*: *pochhammer-rec algebra-simps*)

finally show *?case* **..**
qed

Contributed by Manuel Eberl, generalised by LCP. Alternative definition of the binomial coefficient as $\prod i{<}k.\ (n - i)\ /\ (k - i)$.

lemma *gbinomial-altdef-of-nat*: *a gchoose k* = (∏ *i* = *0*..<*k*. (*a* − *of-nat i*) / *of-nat* (*k* − *i*) :: *'a*)
for *k* :: *nat* **and** *a* :: *'a*::*field-char-0*
by (*simp add*: *prod-dividef gbinomial-prod-rev fact-prod-rev*)

lemma *gbinomial-ge-n-over-k-pow-k*:
fixes *k* :: *nat*
and *a* :: *'a*::*linordered-field*
assumes *of-nat k* ≤ *a*
shows (*a* / *of-nat k* :: *'a*) ^ *k* ≤ *a gchoose k*
proof −
have *x*: *0* ≤ *a*
using *assms of-nat-0-le-iff order-trans* **by** *blast*
have (*a* / *of-nat k* :: *'a*) ^ *k* = (∏ *i* = *0*..<*k*. *a* / *of-nat k* :: *'a*)
by *simp*
also have ... ≤ *a gchoose k*
proof −
have ⋀*i*. *i* < *k* ⟹ *0* ≤ *a* / *of-nat k*
by (*simp add*: *x zero-le-divide-iff*)
moreover have *a* / *of-nat k* ≤ (*a* − *of-nat i*) / *of-nat* (*k* − *i*) **if** *i* < *k* **for** *i*
proof −
from *assms* **have** *a* ∗ *of-nat i* ≥ *of-nat* (*i* ∗ *k*)
by (*metis mult.commute mult-le-cancel-right of-nat-less-0-iff of-nat-mult*)
then have *a* ∗ *of-nat k* − *a* ∗ *of-nat i* ≤ *a* ∗ *of-nat k* − *of-nat* (*i* ∗ *k*)
by *arith*
then have *a* ∗ *of-nat* (*k* − *i*) ≤ (*a* − *of-nat i*) ∗ *of-nat k*
using ‹*i* < *k*› **by** (*simp add*: *algebra-simps zero-less-mult-iff of-nat-diff*)
then have *a* ∗ *of-nat* (*k* − *i*) ≤ (*a* − *of-nat i*) ∗ (*of-nat k* :: *'a*)
by *blast*
with *assms* **show** *?thesis*
using ‹*i* < *k*› **by** (*simp add*: *field-simps*)
qed
ultimately show *?thesis*
unfolding *gbinomial-altdef-of-nat*
by (*intro prod-mono*) *auto*
qed
finally show *?thesis* **.**
qed

lemma *gbinomial-negated-upper*: (*a gchoose k*) = (−*1*) ^ *k* ∗ ((*of-nat k* − *a* − *1*) *gchoose k*)
by (*simp add*: *gbinomial-pochhammer pochhammer-minus algebra-simps*)

lemma *gbinomial-minus*: ((−*a*) *gchoose k*) = (−*1*) ^ *k* ∗ ((*a* + *of-nat k* − *1*)

gchoose k)
by (*metis add.commute diff-minus-eq-add gbinomial-negated-upper*)

lemma *Suc-times-gbinomial*: *of-nat* (*Suc k*) * ((*a* + *1*) *gchoose* (*Suc k*)) = (*a* + *1*) * (*a gchoose k*)
proof (*cases k*)
case *0*
then show *?thesis* **by** *simp*
next
case *Suc*
have ((*a* + *1*) *gchoose* (*Suc k*)) = (∏ *i* = *0*..*k*. *a* + (*1* − *of-nat i*)) / *fact* (*Suc k*)
by (*simp add*: *add-diff-eq gbinomial-Suc*)
also have (∏ *i* = *0*..*k*. *a* + (*1* − *of-nat i*)) = (*a* + *1*) * (∏ *i* = *0*..*k*−*1*. *a* − *of-nat i*)
by (*simp add*: *Suc prod.atLeast0-atMost-Suc-shift del*: *prod.cl-ivl-Suc*)
also have ... / *fact* (*Suc k*) = (*a* + *1*) / *of-nat* (*Suc k*) * (*a gchoose k*)
by (*simp-all add*: *Suc gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost*)
finally show *?thesis*
using *of-nat-neq-0* **by** *auto*
qed

lemma *gbinomial-factors*: ((*a* + *1*) *gchoose* (*Suc k*)) = (*a* + *1*) / *of-nat* (*Suc k*) * (*a gchoose k*)
by (*metis Suc-times-gbinomial nonzero-mult-div-cancel-left of-nat-neq-0 times-divide-eq-left*)

lemma *gbinomial-rec*: ((*a* + *1*) *gchoose* (*Suc k*)) = (*a gchoose k*) * ((*a* + *1*) / *of-nat* (*Suc k*))
by (*simp add*: *gbinomial-factors mult.commute plus-1-eq-Suc*)

lemma *gbinomial-of-nat-symmetric*: *k* ≤ *n* ⟹ (*of-nat n*) *gchoose k* = (*of-nat n*) *gchoose* (*n* − *k*)
using *binomial-symmetric*[*of k n*] **by** (*simp add*: *binomial-gbinomial* [*symmetric*])

The absorption identity (equation 5.5 [3, p. 157]):

$$\binom{r}{k} = \frac{r}{k}\binom{r-1}{k-1}, \quad \text{integer } k \neq 0.$$

lemma *gbinomial-absorption'*: *k* > *0* ⟹ *a gchoose k* = (*a* / *of-nat k*) * (*a* − *1 gchoose* (*k* − *1*))
using *gbinomial-rec*[*of a* − *1 k* − *1*]
by (*simp-all add*: *gbinomial-rec field-simps del*: *of-nat-Suc*)

The absorption identity is written in the following form to avoid division by k (the lower index) and therefore remove the $k \neq 0$ restriction [3, p. 157]:

$$k\binom{r}{k} = r\binom{r-1}{k-1}, \quad \text{integer } k.$$

lemma *gbinomial-absorption*: *of-nat (Suc k) * (a gchoose Suc k) = a * ((a - 1) gchoose k)*
by (*metis Suc-times-gbinomial diff-eq-eq*)

The absorption identity for natural number binomial coefficients:

lemma *binomial-absorption*: *Suc k * (n choose Suc k) = n * ((n - 1) choose k)*
using *times-binomial-minus1-eq* **by** *fastforce*

The absorption companion identity for natural number coefficients, following the proof by GKP [3, p. 157]:

lemma *binomial-absorb-comp*: *(n - k) * (n choose k) = n * ((n - 1) choose k)*
(**is** *?lhs = ?rhs*)
proof (*cases n ≤ k*)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have *?rhs = Suc ((n - 1) - k) * (n choose Suc ((n - 1) - k))*
using *binomial-symmetric*[*of k n - 1*] *binomial-absorption*[*of (n - 1) - k n*]
by *simp*
also have *Suc ((n - 1) - k) = n - k*
using *False* **by** *simp*
also have *n choose ... = n choose k*
using *False* **by** (*intro binomial-symmetric* [*symmetric*]) *simp-all*
finally show *?thesis* **..**
qed

The generalised absorption companion identity:

lemma *gbinomial-absorb-comp*: *(a - of-nat k) * (a gchoose k) = a * ((a - 1) gchoose k)*
using *pochhammer-absorb-comp*[*of a k*] **by** (*simp add*: *gbinomial-pochhammer*)

lemma *gbinomial-addition-formula*:
a gchoose (Suc k) = ((a - 1) gchoose (Suc k)) + ((a - 1) gchoose k)
using *gbinomial-Suc-Suc*[*of a - 1 k*] **by** (*simp add*: *algebra-simps*)

lemma *binomial-addition-formula*:
0 < n ⟹ n choose (Suc k) = ((n - 1) choose (Suc k)) + ((n - 1) choose k)
by (*metis Suc-diff-1 add.commute binomial-Suc-Suc*)

Equation 5.9 of the reference material [3, p. 159] is a useful summation formula, operating on both indices:

$$\sum_{k \le n} \binom{r+k}{k} = \binom{r+n+1}{n}, \quad \text{integer } n.$$

lemma *gbinomial-parallel-sum*: *(∑k≤n. (a + of-nat k) gchoose k) = (a + of-nat n + 1) gchoose n*

proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
then show *?case*
using *gbinomial-Suc-Suc*[*of* (*a* + *of-nat m* + *1*) *m*]
by (*simp add*: *add-ac*)
qed

93.4 Summation on the upper index

Another summation formula is equation 5.10 of the reference material [3, p. 160], aptly named *summation on the upper index*:

$$\sum_{0 \le k \le n} \binom{k}{m} = \binom{n+1}{m+1}, \quad \text{integers } m, n \ge 0.$$

lemma *gbinomial-sum-up-index*:
(∑*j* = *0*..*n*. (*of-nat j gchoose k*) :: *'a*::*field-char-0*) = (*of-nat n* + *1*) *gchoose* (*k* + *1*)
proof (*induct n*)
case *0*
show *?case*
using *gbinomial-Suc-Suc*[*of 0 k*]
by (*cases k*) *auto*
next
case (*Suc n*)
then show *?case*
using *gbinomial-Suc-Suc*[*of of-nat* (*Suc n*) :: *'a k*]
by (*simp add*: *add-ac*)
qed

lemma *gbinomial-index-swap*:
((−*1*) ^ *k*) ∗ ((− (*of-nat n*) − *1*) *gchoose k*) = ((−*1*) ^ *n*) ∗ ((− (*of-nat k*) − *1*) *gchoose n*)
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = (*of-nat* (*k* + *n*) *gchoose k*)
by (*simp add*: *gbinomial-negated-upper* [*of* − *of-nat n* − *1*])
also have ... = (*of-nat* (*k* + *n*) *gchoose n*)
by (*subst gbinomial-of-nat-symmetric*; *simp*)
also have ... = *?rhs*
by (*subst gbinomial-negated-upper*; *simp*)
finally show *?thesis* **.**
qed

lemma *gbinomial-sum-lower-neg*: (∑*k*≤*m*. (*a gchoose k*) ∗ (− *1*) ^ *k*) = (− *1*) ^ *m* ∗ (*a* − *1 gchoose m*)

```
  (is ?lhs = ?rhs)
proof −
  have ?lhs = (∑ k≤m. −(a + 1) + of-nat k gchoose k)
    by (simp add: gbinomial-negated-upper [of a] power-mult-distrib diff-add-eq
mult.commute)
  also have ... = − a + of-nat m gchoose m
    by (simp add: gbinomial-parallel-sum)
  also have ... = ?rhs
    by (simp add: gbinomial-negated-upper [of a−1] power-mult-distrib)
  finally show ?thesis .
qed

lemma gbinomial-partial-row-sum:
  (∑ k≤m. (a gchoose k) * ((a / 2) − of-nat k)) = ((of-nat m + 1)/2) * (a gchoose
(m + 1))
proof (induct m)
  case 0
  then show ?case by simp
next
  case (Suc mm)
  then have (∑ k≤Suc mm. (a gchoose k) * (a / 2 − of-nat k)) =
      (a − of-nat (Suc mm)) * (a gchoose Suc mm) / 2
    by (simp add: field-simps)
  also have ... = a * (a − 1 gchoose Suc mm) / 2
    by (subst gbinomial-absorb-comp) (rule refl)
  also have ... = (of-nat (Suc mm) + 1) / 2 * (a gchoose (Suc mm + 1))
    by (subst gbinomial-absorption [symmetric]) simp
  finally show ?case .
qed

lemma sum-bounds-lt-plus1: (∑ k<mm. f (Suc k)) = (∑ k=1..mm. f k)
  by (induct mm) simp-all

lemma gbinomial-partial-sum-poly:
  (∑ k≤m. (of-nat m + a gchoose k) * x^k * y^(m−k)) =
    (∑ k≤m. (−a gchoose k) * (−x)^k * (x + y)^(m−k))
  (is ?lhs m = ?rhs m)
proof (induction m)
  case 0
  then show ?case by simp
next
  case (Suc m)
  define G where G i k = (of-nat i + a gchoose k) * x^k * y^(i − k) for i k
  define S where S = ?lhs
  have SG-def: S = (λi. (∑ k≤i. (G i k)))
    unfolding S-def G-def ..

  have S (Suc m) = G (Suc m) 0 + (∑ k=Suc 0..Suc m. G (Suc m) k)
   using SG-def by (simp add: sum.atLeast-Suc-atMost atLeast0AtMost [symmetric])
```

also have $(\sum k{=}Suc\ 0..Suc\ m.\ G\ (Suc\ m)\ k) = (\sum k{=}0..m.\ G\ (Suc\ m)\ (Suc\ k))$
by (*subst sum.shift-bounds-cl-Suc-ivl*) *simp*
also have $\ldots = (\sum k{=}0..m.\ ((of\text{-}nat\ m + a\ gchoose\ (Suc\ k)) + (of\text{-}nat\ m + a\ gchoose\ k)) * x\hat{}(Suc\ k) * y\hat{}(m - k))$
unfolding *G-def* **by** (*subst gbinomial-addition-formula*) *simp*
also have $\ldots = (\sum k{=}0..m.\ (of\text{-}nat\ m + a\ gchoose\ (Suc\ k)) * x\hat{}(Suc\ k) * y\hat{}(m - k)) + (\sum k{=}0..m.\ (of\text{-}nat\ m + a\ gchoose\ k) * x\hat{}(Suc\ k) * y\hat{}(m - k))$
by (*subst sum.distrib* [*symmetric*]) (*simp add*: *algebra-simps*)
also have $(\sum k{=}0..m.\ (of\text{-}nat\ m + a\ gchoose\ (Suc\ k)) * x\hat{}(Suc\ k) * y\hat{}(m - k)) = (\sum k{<}Suc\ m.\ (of\text{-}nat\ m + a\ gchoose\ (Suc\ k)) * x\hat{}(Suc\ k) * y\hat{}(m - k))$
by (*simp only*: *atLeast0AtMost lessThan-Suc-atMost*)
also have $\ldots = (\sum k{<}m.\ (of\text{-}nat\ m + a\ gchoose\ Suc\ k) * x\hat{}(Suc\ k) * y\hat{}(m{-}k)) + (of\text{-}nat\ m + a\ gchoose\ (Suc\ m)) * x\hat{}(Suc\ m)$
(**is** - = *?A* + *?B*)
by (*subst sum.lessThan-Suc*) *simp*
also have *?A* $= (\sum k{=}1..m.\ (of\text{-}nat\ m + a\ gchoose\ k) * x\hat{}k * y\hat{}(m - k + 1))$
proof (*subst sum-bounds-lt-plus1* [*symmetric*], *intro sum.cong*[*OF refl*], *clarify*)
fix *k*
assume $k < m$
then have $m - k = m - Suc\ k + 1$
by *linarith*
then show $(of\text{-}nat\ m + a\ gchoose\ Suc\ k) * x\ \hat{}\ Suc\ k * y\ \hat{}\ (m - k) = (of\text{-}nat\ m + a\ gchoose\ Suc\ k) * x\ \hat{}\ Suc\ k * y\ \hat{}\ (m - Suc\ k + 1)$
by (*simp only*:)
qed
also have $\ldots$ + *?B* $= y * (\sum k{=}1..m.\ (G\ m\ k)) + (of\text{-}nat\ m + a\ gchoose\ (Suc\ m)) * x\hat{}(Suc\ m)$
unfolding *G-def* **by** (*simp add*: *sum-distrib-left algebra-simps*)
also have $(\sum k{=}0..m.\ (of\text{-}nat\ m + a\ gchoose\ k) * x\hat{}(Suc\ k) * y\hat{}(m - k)) = x * (S\ m)$
unfolding *S-def* **by** (*simp add*: *sum-distrib-left atLeast0AtMost algebra-simps*)
also have $(G\ (Suc\ m)\ 0) = y * (G\ m\ 0)$
by (*simp add*: *G-def*)
finally have $S\ (Suc\ m) = y * (G\ m\ 0 + (\sum k{=}1..m.\ (G\ m\ k))) + (of\text{-}nat\ m + a\ gchoose\ (Suc\ m)) * x\hat{}(Suc\ m) + x * (S\ m)$
by (*simp add*: *ring-distribs*)
also have $G\ m\ 0 + (\sum k{=}1..m.\ (G\ m\ k)) = S\ m$
by (*simp add*: *SG-def atLeast0AtMost flip*: *sum.atLeast-Suc-atMost*)
finally have $S\ (Suc\ m) = (x + y) * (S\ m) + (of\text{-}nat\ m + a\ gchoose\ (Suc\ m)) * x\hat{}(Suc\ m)$
by (*simp add*: *algebra-simps*)
also have $(of\text{-}nat\ m + a\ gchoose\ (Suc\ m)) = (-1)\ \hat{}\ (Suc\ m) * (-\ a\ gchoose\ (Suc\ m))$
by (*subst gbinomial-negated-upper*) *simp*

also have $(-1) \hat{} Suc\ m * (-\ a\ gchoose\ Suc\ m) * x \hat{} Suc\ m = (-\ a\ gchoose\ (Suc\ m)) * (-x) \hat{} Suc\ m$
by (*simp add*: *power-minus*[*of x*])
also have $(x + y) * S\ m + \ldots = (x + y) * ?rhs\ m + (-\ a\ gchoose\ (Suc\ m)) * (-\ x) \hat{} Suc\ m$
unfolding *S-def* **by** (*simp add*: *Suc.IH*)
also have $(x + y) * ?rhs\ m = (\sum n{\leq}m.\ ((-\ a\ gchoose\ n) * (-\ x) \hat{}\ n * (x + y) \hat{}\ (Suc\ m - n)))$
by (*auto simp*: *Suc-diff-le sum-distrib-left intro*!: *sum.cong*)
also have $\ldots + (-a\ gchoose\ (Suc\ m)) * (-x) \hat{} Suc\ m =$
$(\sum n{\leq}Suc\ m.\ (-\ a\ gchoose\ n) * (-\ x) \hat{}\ n * (x + y) \hat{}\ (Suc\ m - n))$
by *simp*
finally show *?case*
by (*simp only*: *S-def*)
qed

lemma *gbinomial-partial-sum-poly-xpos*:
$(\sum k{\leq}m.\ (of\text{-}nat\ m + a\ gchoose\ k) * x \hat{} k * y \hat{} (m-k)) =$
$(\sum k{\leq}m.\ (of\text{-}nat\ k + a - 1\ gchoose\ k) * x \hat{} k * (x + y) \hat{} (m-k))$ (**is** *?lhs* = *?rhs*)
proof −
have $?lhs = (\sum k{\leq}m.\ (-\ a\ gchoose\ k) * (-\ x) \hat{}\ k * (x + y) \hat{}\ (m - k))$
by (*simp add*: *gbinomial-partial-sum-poly*)
also have $\ldots = (\sum k{\leq}m.\ (-1) \hat{}\ k * (of\text{-}nat\ k - - a - 1\ gchoose\ k) * (-\ x) \hat{}\ k * (x + y) \hat{}\ (m - k))$
by (*metis* (*no-types*, *opaque-lifting*) *gbinomial-negated-upper*)
also have $\ldots = ?rhs$
by (*auto simp flip*: *power-mult-distrib intro*: *sum.cong*)
finally show *?thesis* **.**
qed

lemma *binomial-r-part-sum*: $(\sum k{\leq}m.\ (2 * m + 1\ choose\ k)) = 2 \hat{}\ (2 * m)$
proof −
have $2 * 2 \hat{} (2{*}m) = (\sum k = 0..(2 * m + 1).\ (2 * m + 1\ choose\ k))$
using *choose-row-sum*[**where** *n*=*2 * m + 1*] **by** (*simp add*: *atMost-atLeast0*)
also have $(\sum k = 0..(2 * m + 1).\ (2 * m + 1\ choose\ k)) =$
$(\sum k = 0..m.\ (2 * m + 1\ choose\ k)) +$
$(\sum k = m{+}1..2{*}m{+}1.\ (2 * m + 1\ choose\ k))$
using *sum.ub-add-nat*[*of 0 m* λ*k. 2 * m + 1 choose k m+1*]
by (*simp add*: *mult-2*)
also have $(\sum k = m{+}1..2{*}m{+}1.\ (2 * m + 1\ choose\ k)) =$
$(\sum k = 0..m.\ (2 * m + 1\ choose\ (k + (m + 1))))$
by (*subst sum.shift-bounds-cl-nat-ivl* [*symmetric*]) (*simp add*: *mult-2*)
also have $\ldots = (\sum k = 0..m.\ (2 * m + 1\ choose\ (m - k)))$
by (*intro sum.cong*[*OF refl*], *subst binomial-symmetric*) *simp-all*
also have $\ldots = (\sum k = 0..m.\ (2 * m + 1\ choose\ k))$
using *sum.atLeastAtMost-rev* [*of* λ*k. 2 * m + 1 choose* (*m − k*) *0 m*]
by *simp*
also have $\ldots + \ldots = 2 * \ldots$

by *simp*
finally show *?thesis*
by (*simp add*: *atLeast0AtMost*)
qed

lemma *gbinomial-r-part-sum*: $(\sum k{\leq}m.\ (2 * (of\text{-}nat\ m) + 1\ gchoose\ k)) = 2\ \hat{}\ (2 * m)$
(**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = *of-nat* $(\sum k{\leq}m.\ (2 * m + 1)\ choose\ k)$
by (*simp add*: *binomial-gbinomial add-ac*)
also have ... = *of-nat* $(2\ \hat{}\ (2 * m))$
using *binomial-r-part-sum* **by** *presburger*
finally show *?thesis* **by** *simp*
qed

lemma *gbinomial-sum-nat-pow2*:
$(\sum k{\leq}m.\ (of\text{-}nat\ (m + k)\ gchoose\ k :: 'a{::}field\text{-}char\text{-}0)\ /\ 2\ \hat{}\ k) = 2\ \hat{}\ m$
(**is** *?lhs* = *?rhs*)
proof −
have $2\ \hat{}\ m * 2\ \hat{}\ m = (2\ \hat{}\ (2{*}m) :: 'a)$
by (*induct m*) *simp-all*
also have ... = $(\sum k{\leq}m.\ (2 * (of\text{-}nat\ m) + 1\ gchoose\ k))$
using *gbinomial-r-part-sum* **..**
also have ... = $(\sum k{\leq}m.\ (of\text{-}nat\ (m + k)\ gchoose\ k) * 2\ \hat{}\ (m - k))$
using *gbinomial-partial-sum-poly-xpos*[**where** *x=1* **and** *y=1* **and** *a=of-nat m + 1* **and** *m=m*]
by (*simp add*: *add-ac*)
also have ... = $2\ \hat{}\ m * (\sum k{\leq}m.\ (of\text{-}nat\ (m + k)\ gchoose\ k)\ /\ 2\ \hat{}\ k)$
by (*simp add*: *sum-distrib-left algebra-simps power-diff*)
finally show *?thesis*
by (*subst* (*asm*) *mult-left-cancel*) *simp-all*
qed

lemma *gbinomial-trinomial-revision*:
assumes $k \leq m$
shows $(a\ gchoose\ m) * (of\text{-}nat\ m\ gchoose\ k) = (a\ gchoose\ k) * (a - of\text{-}nat\ k\ gchoose\ (m - k))$
proof −
have $(a\ gchoose\ m) * (of\text{-}nat\ m\ gchoose\ k) = (a\ gchoose\ m) * fact\ m\ /\ (fact\ k * fact\ (m - k))$
using *assms* **by** (*simp add*: *binomial-gbinomial* [*symmetric*] *binomial-fact*)
also have ... = $(a\ gchoose\ k) * (a - of\text{-}nat\ k\ gchoose\ (m - k))$
using *assms* **by** (*simp add*: *gbinomial-pochhammer power-diff pochhammer-product*)
finally show *?thesis* **.**
qed

Versions of the theorems above for the natural-number version of "choose"

lemma *binomial-altdef-of-nat*:

$k \leq n \Longrightarrow$ *of-nat* (*n choose k*) = ($\prod i = 0..<k$. *of-nat* ($n - i$) / *of-nat* ($k - i$) :: *'a*)
for *n k* :: *nat* **and** *x* :: *'a*::*field-char-0*
by (*simp add*: *gbinomial-altdef-of-nat binomial-gbinomial of-nat-diff*)

lemma *binomial-ge-n-over-k-pow-k*: $k \leq n \Longrightarrow$ (*of-nat n* / *of-nat k* :: *'a*) ^ *k* ≤ *of-nat* (*n choose k*)
for *k n* :: *nat* **and** *x* :: *'a*::*linordered-field*
by (*simp add*: *binomial-gbinomial gbinomial-ge-n-over-k-pow-k*)

lemma *binomial-le-pow*:
assumes $r \leq n$
shows *n choose r* ≤ *n* ^ *r*
proof −
have *n choose r* ≤ *fact n div fact* ($n - r$)
using *assms* **by** (*subst binomial-fact-lemma*[*symmetric*]) *auto*
with *fact-div-fact-le-pow* [*OF assms*] **show** *?thesis*
by *auto*
qed

lemma *choose-dvd*:
assumes $k \leq n$ **shows** *fact k* ∗ *fact* ($n - k$) *dvd* (*fact n*)
by (*metis assms binomial-fact-lemma dvd-def of-nat-fact of-nat-mult*)

lemma *fact-fact-dvd-fact*:
fact k ∗ *fact n dvd* (*fact* ($k + n$))
by (*metis add.commute add-diff-cancel-left' choose-dvd le-add2*)

lemma *choose-mult-lemma*:
((*m* + *r* + *k*) *choose* (*m* + *k*)) ∗ ((*m* + *k*) *choose k*) = ((*m* + *r* + *k*) *choose k*) ∗ ((*m* + *r*) *choose m*)
(**is** *?lhs* = -)
proof −
have *?lhs* =
fact (*m* + *r* + *k*) *div* (*fact* (*m* + *k*) ∗ *fact* (*m* + *r* − *m*)) ∗ (*fact* (*m* + *k*) *div* (*fact k* ∗ *fact m*))
by (*simp add*: *binomial-fact'*)
also have ... = *fact* (*m* + *r* + *k*) ∗ *fact* (*m* + *k*) *div*
(*fact* (*m* + *k*) ∗ *fact* (*m* + *r* − *m*) ∗ (*fact k* ∗ *fact m*))
by (*metis add-implies-diff add-le-mono1 choose-dvd diff-cancel2 div-mult-div-if-dvd le-add1 le-add2*)
also have ... = *fact* (*m* + *r* + *k*) *div* (*fact r* ∗ (*fact k* ∗ *fact m*))
by (*auto simp*: *algebra-simps fact-fact-dvd-fact*)
also have ... = (*fact* (*m* + *r* + *k*) ∗ *fact* (*m* + *r*)) *div* (*fact r* ∗ (*fact k* ∗ *fact m*) ∗ *fact* (*m* + *r*))
by *simp*
also have ... =
(*fact* (*m* + *r* + *k*) *div* (*fact k* ∗ *fact* (*m* + *r*)) ∗ (*fact* (*m* + *r*) *div* (*fact r* ∗ *fact m*)))

by (*auto simp*: *div-mult-div-if-dvd fact-fact-dvd-fact algebra-simps*)
finally show *?thesis*
by (*simp add*: *binomial-fact' mult.commute*)
qed

The "Subset of a Subset" identity.

lemma *choose-mult*:
$k \leq m \Longrightarrow m \leq n \Longrightarrow (n\ choose\ m) * (m\ choose\ k) = (n\ choose\ k) * ((n - k)\ choose\ (m - k))$
using *choose-mult-lemma* [*of m−k n−m k*] **by** *simp*

lemma *of-nat-binomial-eq-mult-binomial-Suc*:
assumes $k \leq n$
shows $(of\text{-}nat :: (nat \Rightarrow ('a :: field\text{-}char\text{-}0))) (n\ choose\ k) = of\text{-}nat\ (n + 1 - k) / of\text{-}nat\ (n + 1) * of\text{-}nat\ (Suc\ n\ choose\ k)$
proof (*cases k*)
case *0* **then show** *?thesis*
using *of-nat-neq-0* **by** *auto*
next
case (*Suc l*)
have $of\text{-}nat\ (n + 1) * (\prod i{=}0..{<}k.\ of\text{-}nat\ (n - i)) = (of\text{-}nat :: (nat \Rightarrow 'a))\ (n + 1 - k) * (\prod i{=}0..{<}k.\ of\text{-}nat\ (Suc\ n - i))$
using *prod.atLeast0-lessThan-Suc* [**where** *?'a* = *'a*, *symmetric*, *of* $\lambda i.\ of\text{-}nat\ (Suc\ n - i)\ k$]
by (*simp add*: *ac-simps prod.atLeast0-lessThan-Suc-shift del*: *prod.op-ivl-Suc*)
also have ... = $(of\text{-}nat :: (nat \Rightarrow 'a))\ (Suc\ n - k) * (\prod i{=}0..{<}k.\ of\text{-}nat\ (Suc\ n - i))$
by (*simp add*: *Suc atLeast0-atMost-Suc atLeastLessThanSuc-atLeastAtMost*)
also have ... = $(of\text{-}nat :: (nat \Rightarrow 'a))\ (n + 1 - k) * (\prod i{=}0..{<}k.\ of\text{-}nat\ (Suc\ n - i))$
by (*simp only*: *Suc-eq-plus1*)
finally have $(\prod i{=}0..{<}k.\ of\text{-}nat\ (n - i)) = (of\text{-}nat :: (nat \Rightarrow 'a))\ (n + 1 - k) / of\text{-}nat\ (n + 1) * (\prod i{=}0..{<}k.\ of\text{-}nat\ (Suc\ n - i))$
using *of-nat-neq-0* **by** (*auto simp*: *mult.commute divide-simps*)
with *assms* **show** *?thesis*
by (*simp add*: *binomial-altdef-of-nat prod-dividef*)
qed

93.5 More on Binomial Coefficients

The number of nat lists of length m summing to N is $N + m - 1\ choose\ N$:

lemma *card-length-sum-list-rec*:
assumes $m \geq 1$
shows $card\ \{l{::}nat\ list.\ length\ l = m \wedge sum\text{-}list\ l = N\} =$
$card\ \{l.\ length\ l = (m - 1) \wedge sum\text{-}list\ l = N\} +$
$card\ \{l.\ length\ l = m \wedge sum\text{-}list\ l + 1 = N\}$
(**is** *card ?C* = *card ?A* + *card ?B*)

proof −
let *?A′* = {*l*. *length l* = *m* ∧ *sum-list l* = *N* ∧ *hd l* = *0*}
let *?B′* = {*l*. *length l* = *m* ∧ *sum-list l* = *N* ∧ *hd l* ≠ *0*}
let *?f* = λ*l*. *0* # *l*
let *?g* = λ*l*. (*hd l* + *1*) # *tl l*
have *1*: *xs* ≠ [] ⟹ *x* = *hd xs* ⟹ *x* # *tl xs* = *xs* **for** *x* :: *nat* **and** *xs*
by *simp*
have *2*: *xs* ≠ [] ⟹ *sum-list*(*tl xs*) = *sum-list xs* − *hd xs* **for** *xs* :: *nat list*
by (*auto simp add*: *neq-Nil-conv*)
have *f*: *bij-betw ?f ?A ?A′*
by (*rule bij-betw-byWitness*[**where** *f′* = *tl*]) (*use assms* **in** ‹*auto simp*: *2 1 simp flip*: *length-0-conv*›)
have *3*: *xs* ≠ [] ⟹ *hd xs* + (*sum-list xs* − *hd xs*) = *sum-list xs* **for** *xs* :: *nat list*
by (*metis 1 sum-list-simps*(*2*) *2*)
have *g*: *bij-betw ?g ?B ?B′*
apply (*rule bij-betw-byWitness*[**where** *f′* = λ*l*. (*hd l* − *1*) # *tl l*])
using *assms*
by (*auto simp*: *2 simp flip*: *length-0-conv intro*!: *3*)
have *fin*: *finite* {*xs*. *size xs* = *M* ∧ *set xs* ⊆ {*0*..<*N*}} **for** *M N* :: *nat*
using *finite-lists-length-eq*[*OF finite-atLeastLessThan*] *conj-commute* **by** *auto*
have *fin-A*: *finite ?A* **using** *fin*[*of* - *N+1*]
by (*intro finite-subset*[**where** *?A* = *?A* **and** *?B* = {*xs*. *size xs* = *m* − *1* ∧ *set xs* ⊆ {*0*..<*N+1*}}])
(*auto simp*: *member-le-sum-list less-Suc-eq-le*)
have *fin-B*: *finite ?B*
by (*intro finite-subset*[**where** *?A* = *?B* **and** *?B* = {*xs*. *size xs* = *m* ∧ *set xs* ⊆ {*0*..<*N*}}])
(*auto simp*: *member-le-sum-list less-Suc-eq-le fin*)
have *disj*: *?A′* ∩ *?B′* = {} **by** *blast*
have *?C* = *?A′* ∪ *?B′*
by *auto*
then have *card ?C* = *card*(*?A′* ∪ *?B′*)
by *simp*
also have ... = *card ?A* + *card ?B*
using *card-Un-disjoint*[*OF* - - *disj*] *bij-betw-finite*[*OF f*] *bij-betw-finite*[*OF g*]
bij-betw-same-card[*OF f*] *bij-betw-same-card*[*OF g*] *fin-A fin-B*
by *presburger*
finally show *?thesis* **.**
qed

lemma *card-length-sum-list*: *card* {*l*::*nat list*. *size l* = *m* ∧ *sum-list l* = *N*} = (*N* + *m* − *1*) *choose N*
— by Holden Lee, tidied by Tobias Nipkow
proof (*cases m*)
case *0*
then show *?thesis*
by (*cases N*) (*auto cong*: *conj-cong*)
next
case (*Suc m′*)

```
  have m: m ≥ 1
    by (simp add: Suc)
  then show ?thesis
  proof (induct N + m − 1 arbitrary: N m)
    case 0 — In the base case, the only solution is [0].
    have [simp]: {l::nat list. length l = Suc 0 ∧ (∀ n∈set l. n = 0)} = {[0]}
      by (auto simp: length-Suc-conv)
    have m = 1 ∧ N = 0
      using 0 by linarith
    then show ?case
      by simp
  next
    case (Suc k)
    have c1: card {l::nat list. size l = (m − 1) ∧ sum-list l =  N} = (N + (m −
1) − 1) choose N
    proof (cases m = 1)
      case True
      with Suc.hyps have N ≥ 1
        by auto
      with True show ?thesis
        by (simp add: binomial-eq-0)
    next
      case False
      then show ?thesis
        using Suc by fastforce
    qed
    from Suc have c2: card {l::nat list. size l = m ∧ sum-list l + 1 = N} =
      (if N > 0 then ((N − 1) + m − 1) choose (N − 1) else 0)
    proof −
      have ∗: n > 0 ⟹ Suc m = n ⟷ m = n − 1 for m n
        by arith
      from Suc have N > 0 ⟹
        card {l::nat list. size l = m ∧ sum-list l + 1 = N} =
          ((N − 1) + m − 1) choose (N − 1)
        by (simp add: ∗)
      then show ?thesis
        by auto
    qed
    from Suc.prems have (card {l::nat list. size l = (m − 1) ∧ sum-list l = N} +
          card {l::nat list. size l = m ∧ sum-list l + 1 = N}) = (N + m − 1)
choose N
      by (auto simp: c1 c2 choose-reduce-nat[of N + m − 1 N] simp del: One-nat-def)
    then show ?case
      using card-length-sum-list-rec[OF Suc.prems] by auto
  qed
qed

lemma card-disjoint-shuffles:
  assumes set xs ∩ set ys = {}
```

shows *card (shuffles xs ys) = (length xs + length ys) choose length xs*
using *assms*
proof (*induction xs ys rule: shuffles.induct*)
case (*3 x xs y ys*)
have *shuffles (x # xs) (y # ys) = (#) x ' shuffles xs (y # ys) ∪ (#) y ' shuffles (x # xs) ys*
by (*rule shuffles.simps*)
also have *card ... = card ((#) x ' shuffles xs (y # ys)) + card ((#) y ' shuffles (x # xs) ys)*
by (*rule card-Un-disjoint*) (*use 3* **in** *auto*)
also have *card ((#) x ' shuffles xs (y # ys)) = card (shuffles xs (y # ys))*
by (*rule card-image*) *auto*
also have *... = (length xs + length (y # ys)) choose length xs*
using *3.prems* **by** (*intro 3.IH*) *auto*
also have *card ((#) y ' shuffles (x # xs) ys) = card (shuffles (x # xs) ys)*
by (*rule card-image*) *auto*
also have *... = (length (x # xs) + length ys) choose length (x # xs)*
using *3.prems* **by** (*intro 3.IH*) *auto*
also have *(length xs + length (y # ys) choose length xs) + ... =*
(length (x # xs) + length (y # ys)) choose length (x # xs) **by** *simp*
finally show *?case* **.**
qed *auto*

lemma *Suc-times-binomial-add: Suc a * (Suc (a + b) choose Suc a) = Suc b * (Suc (a + b) choose a)*
— by Lukas Bulwahn
proof −
have *dvd: Suc a * (fact a * fact b) dvd fact (Suc (a + b))* **for** *a b*
using *fact-fact-dvd-fact[of Suc a b]*
by (*metis add.commute add-Suc-right fact-Suc id-apply mult.assoc of-nat-eq-id*)
have *Suc a * (fact (Suc (a + b)) div (Suc a * fact a * fact b)) =*
*Suc a * fact (Suc (a + b)) div (Suc a * (fact a * fact b))*
by (*metis mult.assoc div-mult-swap dvd*)
also have *... = Suc b * fact (Suc (a + b)) div (Suc b * (fact a * fact b))*
by (*simp only: div-mult-mult1*)
also have *... = Suc b * (fact (Suc (a + b)) div (Suc b * (fact a * fact b)))*
by (*metis add.commute div-mult-swap dvd mult.commute*)
finally show *?thesis*
by (*metis Suc-diff-le Suc-eq-plus1 Suc-times-binomial add.commute binomial-absorb-comp diff-add-inverse le-add1*)
qed

93.6 Inclusion-exclusion principle

Ported from HOL Light by lcp

lemma *Inter-over-Union:*
⋂ {⋃ (ℱ x) |x. x ∈ S} = ⋃ {⋂ (G ' S) |G. ∀x∈S. G x ∈ ℱ x}
proof −
have *⋀x. ∀s∈S. ∃X ∈ ℱ s. x ∈ X ⟹ ∃G. (∀x∈S. G x ∈ ℱ x) ∧ (∀s∈S. x ∈*

G s)
by *metis*
then show *?thesis*
by (*auto simp flip*: *all-simps ex-simps*)
qed

lemma *subset-insert-lemma*:
$\{T.\ T \subseteq (insert\ a\ S) \wedge P\ T\} = \{T.\ T \subseteq S \wedge P\ T\} \cup \{insert\ a\ T\ |T.\ T \subseteq S \wedge P(insert\ a\ T)\}$ (**is** *?L=?R*)
proof
show *?L* $\subseteq$ *?R*
by (*smt* (*verit*) *UnI1 UnI2 insert-Diff mem-Collect-eq subsetI subset-insert-iff*)
qed *blast*

Versions for additive real functions, where the additivity applies only to some specific subsets (e.g. cardinality of finite sets, measurable sets with bounded measure. (From HOL Light)

locale *Incl-Excl* =
fixes $P :: 'a\ set \Rightarrow bool$ **and** $f :: 'a\ set \Rightarrow 'b::ring\text{-}1$
assumes *disj-add*: $[\![P\ S;\ P\ T;\ disjnt\ S\ T]\!] \Longrightarrow f(S \cup T) = f\ S + f\ T$
and *empty*: $P\{\}$
and *Int*: $[\![P\ S;\ P\ T]\!] \Longrightarrow P(S \cap T)$
and *Un*: $[\![P\ S;\ P\ T]\!] \Longrightarrow P(S \cup T)$
and *Diff*: $[\![P\ S;\ P\ T]\!] \Longrightarrow P(S - T)$

begin

lemma *f-empty* [*simp*]: $f\{\} = 0$
using *disj-add empty* **by** *fastforce*

lemma *f-Un-Int*: $[\![P\ S;\ P\ T]\!] \Longrightarrow f(S \cup T) + f(S \cap T) = f\ S + f\ T$
by (*smt* (*verit*, *ccfv-threshold*) *Groups.add-ac*(*2*) *Incl-Excl.Diff Incl-Excl.Int Incl-Excl-axioms Int-Diff-Un Int-Diff-disjoint Int-absorb Un-Diff Un-Int-eq*(*2*) *disj-add disjnt-def group-cancel.add2 sup-bot.right-neutral*)

lemma *restricted-indexed*:
assumes *finite A* **and** *X*: $\bigwedge a.\ a \in A \Longrightarrow P(X\ a)$
shows $f(\bigcup (X\ `\ A)) = (\sum B \mid B \subseteq A \wedge B \neq \{\}.\ (-\ 1)\ \widehat{}\ (card\ B + 1) * f\ (\bigcap (X\ `\ B)))$
proof −
have $[\![finite\ A;\ card\ A = n;\ \forall a \in A.\ P\ (X\ a)]\!] \Longrightarrow f(\bigcup (X\ `\ A)) = (\sum B \mid B \subseteq A \wedge B \neq \{\}.\ (-\ 1)\ \widehat{}\ (card\ B + 1) * f\ (\bigcap\ (X\ `\ B)))$ **for** *n X* **and** $A :: 'c\ set$
proof (*induction n arbitrary*: *A X rule*: *less-induct*)
case (*less n0 A0 X*)
show *?case*
proof (*cases n0=0*)
case *True*
with *less* **show** *?thesis*

by *fastforce*
next
case *False*
with *less.prems* **obtain** *A n a* **where** ∗: *n0 = Suc n A0 = insert a A a ∉ A card A = n finite A*
by (*metis card-Suc-eq-finite not0-implies-Suc*)
with *less* **have** *P (X a)* **by** *blast*
have *APX*: ∀ *a* ∈ *A*. *P (X a)*
by (*simp add*: ∗ *less.prems*)
have *PUXA*: *P* (⋃ (*X ‘ A*))
using ‹*finite A*› *APX*
by (*induction*) (*auto simp*: *empty Un*)
have *f* (⋃ (*X ‘ A0*)) = *f* (*X a* ∪ ⋃ (*X ‘ A*))
by (*simp add*: ∗)
also have ... = *f* (*X a*) + *f* (⋃ (*X ‘ A*)) − *f* (*X a* ∩ ⋃ (*X ‘ A*))
using *f-Un-Int add-diff-cancel PUXA* ‹*P* (*X a*)› **by** *metis*
also have ... = *f* (*X a*) − (∑ *B* | *B* ⊆ *A* ∧ *B* ≠ {}. (− *1*) ^ *card B* ∗ *f* (⋂ (*X ‘ B*))) +
(∑ *B* | *B* ⊆ *A* ∧ *B* ≠ {}. (− *1*) ^ *card B* ∗ *f* (*X a* ∩ ⋂ (*X ‘ B*)))
proof −
have *1*: *f* (⋃*i*∈*A*. *X a* ∩ *X i*) = (∑ *B* | *B* ⊆ *A* ∧ *B* ≠ {}. (− *1*) ^ (*card B* + *1*) ∗ *f* (⋂*b*∈*B*. *X a* ∩ *X b*))
using *less.IH* [*of n A* λ*i*. *X a* ∩ *X i*] *APX Int* ‹*P* (*X a*)› **by** (*simp add*: ∗)
have *2*: *X a* ∩ ⋃ (*X ‘ A*) = (⋃*i*∈*A*. *X a* ∩ *X i*)
by *auto*
have *3*: *f* (⋃ (*X ‘ A*)) = (∑ *B* | *B* ⊆ *A* ∧ *B* ≠ {}. (− *1*) ^ (*card B* + *1*) ∗ *f* (⋂ (*X ‘ B*)))
using *less.IH* [*of n A X*] *APX Int* ‹*P* (*X a*)› **by** (*simp add*: ∗)
show *?thesis*
unfolding *3 2 1*
by (*simp add*: *sum-negf*)
qed
also have ... = (∑ *B* | *B* ⊆ *A0* ∧ *B* ≠ {}. (− *1*) ^ (*card B* + *1*) ∗ *f* (⋂ (*X ‘ B*)))
proof −
have *F*: {*insert a B* |*B*. *B* ⊆ *A*} = *insert a ‘ Pow A* ∧ {*B*. *B* ⊆ *A* ∧ *B* ≠ {}} = *Pow A* − {{}}
by *auto*
have *G*: (∑ *B*∈*Pow A*. (− *1*) ^ *card* (*insert a B*) ∗ *f* (*X a* ∩ ⋂ (*X ‘ B*))) = (∑ *B*∈*Pow A*. − ((− *1*) ^ *card B* ∗ *f* (*X a* ∩ ⋂ (*X ‘ B*))))
proof (*rule sum.cong* [*OF refl*])
fix *B*
assume *B*: *B* ∈ *Pow A*
then have *finite B*
using ‹*finite A*› *finite-subset* **by** *auto*
show (− *1*) ^ *card* (*insert a B*) ∗ *f* (*X a* ∩ ⋂ (*X ‘ B*)) = − ((− *1*) ^ *card B* ∗ *f* (*X a* ∩ ⋂ (*X ‘ B*)))
using *B* ∗ **by** (*auto simp add*: *card-insert-if* ‹*finite B*›)
qed

```
      have disj: {B. B ⊆ A ∧ B ≠ {}} ∩ {insert a B |B. B ⊆ A} = {}
        using ∗ by blast
      have inj: inj-on (insert a) (Pow A)
        using ∗ inj-on-def by fastforce
      show ?thesis
       apply (simp add: ∗ subset-insert-lemma sum.union-disjoint disj sum-negf)
        apply (simp add: F G sum-negf sum.reindex [OF inj] o-def sum-diff ∗)
        done
    qed
    finally show ?thesis .
  qed
 qed
 then show ?thesis
   by (meson assms)
qed

lemma restricted:
 assumes finite A ⋀a. a ∈ A ⟹ P a
 shows  f(⋃ A) = (∑ B | B ⊆ A ∧ B ≠ {}. (− 1) ^(card B + 1) ∗ f (⋂ B))
 using restricted-indexed [of A λx. x] assms by auto

end
```

93.7 Versions for unrestrictedly additive functions

```
lemma Incl-Excl-UN:
 fixes f :: 'a set ⇒ 'b::ring-1
 assumes ⋀S T. disjnt S T ⟹ f(S ∪ T) = f S + f T finite A
 shows f(⋃(G ‘ A)) = (∑ B | B ⊆ A ∧ B ≠ {}. (−1) ^(card B + 1) ∗ f (⋂
(G ‘ B)))
proof −
 interpret Incl-Excl λx. True f
   by (simp add: Incl-Excl.intro assms(1))
 show ?thesis
   using restricted-indexed assms by blast
qed

lemma Incl-Excl-Union:
 fixes f :: 'a set ⇒ 'b::ring-1
 assumes ⋀S T. disjnt S T ⟹ f(S ∪ T) = f S + f T finite A
 shows f(⋃ A) = (∑ B | B ⊆ A ∧ B ≠ {}. (− 1) ^(card B + 1) ∗ f (⋂ B))
 using Incl-Excl-UN[of f A λX. X] assms by simp
```

The famous inclusion-exclusion formula for the cardinality of a union

```
lemma int-card-UNION:
 assumes finite A ⋀K. K ∈ A ⟹ finite K
 shows int (card (⋃A)) = (∑ I | I ⊆ A ∧ I ≠ {}. (− 1) ^(card I + 1) ∗ int
(card (⋂I)))
proof −
```

interpret *Incl-Excl finite int o card*
proof qed (*auto simp add*: *card-Un-disjnt*)
show *?thesis*
using *restricted assms* **by** *auto*
qed

A more conventional form

lemma *inclusion-exclusion*:
assumes *finite A* $\bigwedge K.\ K \in A \Longrightarrow$ *finite K*
shows *int*(*card*($\bigcup$ *A*)) =
($\sum n{=}1..card\ A.\ (-1)\ \hat{}\ (Suc\ n) * (\sum B \mid B \subseteq A \wedge card\ B = n.\ int\ (card\ (\bigcap\ B))))$ (**is** -=*?R*)
proof −
have *fin*: *finite* $\{I.\ I \subseteq A \wedge I \neq \{\}\}$
by (*simp add*: *assms*)
have $\bigwedge k.\ [\![Suc\ 0 \leq k;\ k \leq card\ A]\!] \Longrightarrow \exists B{\subseteq}A.\ B \neq \{\} \wedge k = card\ B$
by (*metis* (*mono-tags*, *lifting*) *Suc-le-D Zero-neq-Suc card-eq-0-iff obtain-subset-with-card-n*)
with ‹*finite A*› *finite-subset*
have *card-eq*: *card* ‘ $\{I.\ I \subseteq A \wedge I \neq \{\}\} = \{1..card\ A\}$
using *not-less-eq-eq card-mono* **by** (*fastforce simp*: *image-iff*)
have *int*(*card*($\bigcup$ *A*))
$= (\sum y = 1..card\ A.\ \sum I{\in}\{x.\ x \subseteq A \wedge x \neq \{\} \wedge card\ x = y\}.\ -\ ((-\ 1)\ \hat{}\ y * int\ (card\ (\bigcap\ I))))$
by (*simp add*: *int-card-UNION assms sum.image-gen* [*OF fin*, **where** *g*=*card*] *card-eq*)
also have ... = *?R*
proof −
have $\{B.\ B \subseteq A \wedge B \neq \{\} \wedge card\ B = k\} = \{B.\ B \subseteq A \wedge card\ B = k\}$
if *Suc 0* ≤ *k* **and** *k* ≤ *card A* **for** *k*
using *that* **by** *auto*
then show *?thesis*
by (*clarsimp simp add*: *sum-negf simp flip*: *sum-distrib-left*)
qed
finally show *?thesis* **.**
qed

lemma *card-UNION*:
assumes *finite A* **and** $\bigwedge K.\ K \in A \Longrightarrow$ *finite K*
shows *card* $(\bigcup A) = nat\ (\sum I \mid I \subseteq A \wedge I \neq \{\}.\ (-\ 1)\ \hat{}\ (card\ I + 1) * int\ (card\ (\bigcap I)))$
by (*simp only*: *flip*: *int-card-UNION* [*OF assms*])

lemma *card-UNION-nonneg*:
assumes *finite A* **and** $\bigwedge K.\ K \in A \Longrightarrow$ *finite K*
shows $(\sum I \mid I \subseteq A \wedge I \neq \{\}.\ (-\ 1)\ \hat{}\ (card\ I + 1) * int\ (card\ (\bigcap I))) \geq 0$
using *int-card-UNION* [*OF assms*] **by** *presburger*

93.8 General "Moebius inversion" inclusion-exclusion principle

This "symmetric" form is from Ira Gessel: "Symmetric Inclusion-Exclusion"

lemma *sum-Un-eq*:
  $[\![S \cap T = \{\};\ S \cup T = U;\ \mathit{finite}\ U]\!]$
    $\Longrightarrow (\mathit{sum}\ f\ S + \mathit{sum}\ f\ T = \mathit{sum}\ f\ U)$
  by (*metis finite-Un sum.union-disjoint*)

lemma *card-adjust-lemma*: $[\![\mathit{inj\text{-}on}\ f\ S;\ x = y + \mathit{card}\ (f\ `\ S)]\!] \Longrightarrow x = y + \mathit{card}\ S$
  by (*simp add*: *card-image*)

lemma *card-subsets-step*:
  assumes *finite S* $x \notin S$ $U \subseteq S$
  shows *card* $\{T.\ T \subseteq (\mathit{insert}\ x\ S) \wedge U \subseteq T \wedge \mathit{odd}(\mathit{card}\ T)\}$
      $= \mathit{card}\ \{T.\ T \subseteq S \wedge U \subseteq T \wedge \mathit{odd}(\mathit{card}\ T)\} + \mathit{card}\ \{T.\ T \subseteq S \wedge U \subseteq T \wedge \mathit{even}(\mathit{card}\ T)\} \wedge$
        $\mathit{card}\ \{T.\ T \subseteq (\mathit{insert}\ x\ S) \wedge U \subseteq T \wedge \mathit{even}(\mathit{card}\ T)\}$
      $= \mathit{card}\ \{T.\ T \subseteq S \wedge U \subseteq T \wedge \mathit{even}(\mathit{card}\ T)\} + \mathit{card}\ \{T.\ T \subseteq S \wedge U \subseteq T \wedge \mathit{odd}(\mathit{card}\ T)\}$
proof −
  have *inj*: *inj-on* (*insert x*) $\{T.\ T \subseteq S \wedge P\ T\}$ **for** *P*
    using *assms* **by** (*auto simp*: *inj-on-def*)
  have [*simp*]: *finite* $\{T.\ T \subseteq S \wedge P\ T\}$ *finite* (*insert x* ` $\{T.\ T \subseteq S \wedge P\ T\}$) **for** *P*
    using ‹*finite S*› **by** *auto*
  have [*simp*]: *disjnt* $\{T.\ T \subseteq S \wedge P\ T\}$ (*insert x* ` $\{T.\ T \subseteq S \wedge Q\ T\}$) **for** *P Q*
    using *assms* **by** (*auto simp*: *disjnt-iff*)
  have *eq*: $\{T.\ T \subseteq S \wedge U \subseteq T \wedge P\ T\} \cup \mathit{insert}\ x\ `\ \{T.\ T \subseteq S \wedge U \subseteq T \wedge Q\ T\}$
      $= \{T.\ T \subseteq \mathit{insert}\ x\ S \wedge U \subseteq T \wedge P\ T\}$ (**is** *?L* = *?R*)
    if $\bigwedge A.\ A \subseteq S \Longrightarrow Q\ (\mathit{insert}\ x\ A) \longleftrightarrow P\ A$ $\bigwedge A.\ \neg\ Q\ A \longleftrightarrow P\ A$ **for** *P Q*
  proof
    show *?L* ⊆ *?R*
      by (*clarsimp simp*: *image-iff subset-iff*) (*meson subsetI that*)
    show *?R* ⊆ *?L*
      using ‹$U \subseteq S$›
     by (*clarsimp simp*: *image-iff*) (*smt* (*verit*) *insert-iff mk-disjoint-insert subset-iff that*)
  qed
  have [*simp*]: $\bigwedge A.\ A \subseteq S \Longrightarrow \mathit{even}\ (\mathit{card}\ (\mathit{insert}\ x\ A)) \longleftrightarrow \mathit{odd}\ (\mathit{card}\ A)$
    by (*metis* ‹*finite S*› ‹$x \notin S$› *card-insert-disjoint even-Suc finite-subset subsetD*)
  show *?thesis*
    by (*intro conjI card-adjust-lemma* [*OF inj*]; *simp add*: *eq flip*: *card-Un-disjnt*)
qed

lemma *card-subsupersets-even-odd*:
  assumes *finite S* $U \subset S$
  shows *card* $\{T.\ T \subseteq S \wedge U \subseteq T \wedge \mathit{even}(\mathit{card}\ T)\}$

```
    = card {T. T ⊆ S ∧ U ⊆ T ∧ odd(card T)}
  using assms
proof (induction card S arbitrary: S rule: less-induct)
  case (less S)
  then obtain x where x ∉ U x ∈ S
    by blast
  then have U: U ⊆ S − {x}
    using less.prems(2) by blast
  let ?V = S − {x}
  show ?case
    using card-subsets-step [of ?V x U] less.prems U
    by (simp add: insert-absorb ‹x ∈ S›)
qed

lemma sum-alternating-cancels:
  assumes finite S card {x. x ∈ S ∧ even(f x)} = card {x. x ∈ S ∧ odd(f x)}
  shows (∑ x∈S. (−1) ^ f x) = (0::'b::ring-1)
proof −
  have (∑ x∈S. (−1) ^ f x)
    = (∑ x | x ∈ S ∧ even (f x). (−1) ^ f x) + (∑ x | x ∈ S ∧ odd (f x). (−1)
^ f x)
    by (rule sum-Un-eq [symmetric]; force simp: ‹finite S›)
  also have ... = (0::'b::ring-1)
    by (simp add: minus-one-power-iff assms cong: conj-cong)
  finally show ?thesis .
qed

lemma inclusion-exclusion-symmetric:
  fixes f :: 'a set ⇒ 'b::ring-1
  assumes §: ⋀S. finite S ⟹ g S = (∑ T ∈ Pow S. (−1) ^ card T * f T)
    and finite S
  shows f S = (∑ T ∈ Pow S. (−1) ^ card T * g T)
proof −
  have (−1) ^ card T * g T = (−1) ^ card T * (∑ U | U ⊆ S ∧ U ⊆ T. (−1)
^ card U * f U)
    if T ⊆ S for T
  proof −
    have [simp]: {U. U ⊆ S ∧ U ⊆ T} = Pow T
      using that by auto
    show ?thesis
      using that by (simp add: ‹finite S› finite-subset §)
  qed
  then have (∑ T ∈ Pow S. (−1) ^ card T * g T)
    = (∑ T∈Pow S. (−1) ^ card T * (∑ U | U ∈ {U. U ⊆ S} ∧ U ⊆ T. (−1)
^ card U * f U))
    by simp
 also have ... = (∑ U∈Pow S. (∑ T | T ⊆ S ∧ U ⊆ T. (−1) ^ card T) * (−1)
^ card U * f U)
    unfolding sum-distrib-left
```

by (*subst sum.swap-restrict*; *simp add*: ‹*finite S*› *algebra-simps sum-distrib-right Pow-def*)
also have ... = ($\sum$ *U*∈*Pow S*. *if U*=*S then f S else 0*)
proof −
have [*simp*]: {*T*. *T* ⊆ *S* ∧ *S* ⊆ *T*} = {*S*}
by *auto*
show *?thesis*
apply (*rule sum.cong* [*OF refl*])
by (*simp add*: *sum-alternating-cancels card-subsupersets-even-odd* ‹*finite S*› *flip*: *power-add*)
qed
also have ... = *f S*
by (*simp add*: ‹*finite S*›)
finally show *?thesis*
by *presburger*
qed

The more typical non-symmetric version.

lemma *inclusion-exclusion-mobius*:
fixes *f* :: *'a set* ⇒ *'b*::*ring-1*
assumes §: ⋀*S*. *finite S* ⟹ *g S* = *sum f* (*Pow S*) **and** *finite S*
shows *f S* = ($\sum$ *T* ∈ *Pow S*. (−1) ^ (*card S* − *card T*) ∗ *g T*) (**is** - = *?rhs*)
proof −
have (− *1*) ^ *card S* ∗ *f S* = ($\sum$ *T*∈*Pow S*. (− *1*) ^ *card T* ∗ *g T*)
by (*rule inclusion-exclusion-symmetric*; *simp add*: *assms flip*: *power-add mult.assoc*)
then have ((− *1*) ^ *card S* ∗ (− *1*) ^ *card S*) ∗ *f S* = ((− *1*) ^ *card S*) ∗ ($\sum$ *T*∈*Pow S*. (− *1*) ^ *card T* ∗ *g T*)
by (*simp add*: *mult-ac*)
then have *f S* = ($\sum$ *T*∈*Pow S*. (− *1*) ^ (*card S* + *card T*) ∗ *g T*)
by (*simp add*: *sum-distrib-left flip*: *power-add mult.assoc*)
also have ... = *?rhs*
by (*simp add*: ‹*finite S*› *card-mono neg-one-power-add-eq-neg-one-power-diff*)
finally show *?thesis* **.**
qed

93.9 Executable code

lemma *gbinomial-code* [*code*]:
a gchoose k =
(*if k* = *0 then 1*
else fold-atLeastAtMost-nat (λ*k acc*. (*a* − *of-nat k*) ∗ *acc*) *0* (*k* − *1*) *1* / *fact k*)
by (*cases k*) (*simp-all add*: *gbinomial-prod-rev atLeastLessThanSuc-atLeastAtMost flip*: *prod-atLeastAtMost-code*)

lemma *binomial-code* [*code*]:
n choose k =
(*if k* > *n then 0*
else if 2 ∗ *k* > *n then n choose* (*n* − *k*)

else (*fold-atLeastAtMost-nat* (∗) (*n* − *k* + *1*) *n 1 div fact k*))
proof −
{
assume $k \leq n$
then have (*fact n* :: *nat*) = *fact* (*n*−*k*) ∗ $\prod$ {*n*−*k*+*1*..*n*}
by (*metis Suc-eq-plus1 diff-le-self fact-eq-fact-times*)
}
then show *?thesis* **by** (*auto simp*: *binomial-fact′ mult-ac prod-atLeastAtMost-code*)
qed

end

theory *Alethe-Rare-Nary-Ops*
imports *HOL.List*
begin

datatype *′a cvc-ListVar* = *ListVar ′a list*
datatype (*′a*,*′b*) *cvc-ListOp* = *ListOp ′a* ⇒ *′b* ⇒ *′b ′a*

fun *cvc-isListOp* **where**
cvc-isListOp (*ListOp op neutralElement*) =
((∀ *x*. *op x neutralElement* = *x*) ∧ (∀ *x*. *op neutralElement x* = *x*))

named-theorems *cvc-ListOp-neutral* ‹*neutral elements for cvc-ListOps*›

lemma [*cvc-ListOp-neutral*]:
shows *cvc-ListOp-neutral-and*: *cvc-isListOp* (*ListOp* (∧) *True*)
and *cvc-ListOp-neutral-or*: *cvc-isListOp* (*ListOp* (∨) *False*)
and *cvc-ListOp-neutral-plus*: *cvc-isListOp* (*ListOp* (+) (*0*::*′a*::*monoid-add*))
and *cvc-ListOp-neutral-plus-int*: *cvc-isListOp* (*ListOp* (+) (*0*::*int*))
and *cvc-ListOp-neutral-mult*: *cvc-isListOp* (*ListOp* (∗) (*1*::*int*))
and *cvc-ListOp-neutral-append*: *cvc-isListOp* (*ListOp* (@) [])
by (*simp-all*)

fun *cvc-nary-op-fold* :: (*′a* ⇒ *′a* ⇒ *′a*) ⇒ *′a list* ⇒ *′a* **where**
cvc-nary-op-fold-Nil: *cvc-nary-op-fold op* [*x*] = *x* |
cvc-nary-op-fold-Cons: *cvc-nary-op-fold op* (*x*#*xs*) = (*op x* (*cvc-nary-op-fold op xs*))

fun *cvc-bin-op-fold* :: (*′a* ⇒ *′b* ⇒ *′b*) ⇒ *′a list* ⇒ *′b* ⇒ *′b* **where**
cvc-bin-op-fold-Nil: *cvc-bin-op-fold op* [] *y* = *y* |
cvc-bin-op-fold-Cons: *cvc-bin-op-fold op* (*x*#*xs*) *y* = (*op x* (*cvc-bin-op-fold op xs y*))

fun *cvc-bin-op* :: ('*a* ⇒ '*b* ⇒ '*b*) ⇒ '*a cvc-ListVar* ⇒ '*b* ⇒ '*b* **where**
cvc-bin-op op (*ListVar xs*) *y* = *cvc-bin-op-fold op xs y*
definition *cvc-list-left* **where** *cvc-list-left op lv y* = *cvc-bin-op op lv y*

fun *cvc-bin-op2* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ '*a* ⇒ '*a cvc-ListVar* ⇒ '*a* **where**
cvc-bin-op2 op y (*ListVar xs*) = (*if xs* = [] *then y else op y* (*cvc-nary-op-fold op xs*))
definition *cvc-list-right* **where** *cvc-list-right op y lv* = *cvc-bin-op2 op y lv*

fun *cvc-bin-op3* **where**
cvc-bin-op3 op (*ListVar* []) (*ListVar* []) *neutral* = *neutral* |
cvc-bin-op3 op (*ListVar xs*) (*ListVar* []) *neutral* = *cvc-nary-op-fold op xs* |
cvc-bin-op3 op (*ListVar xs*) (*ListVar ys*) *neutral* = *cvc-bin-op-fold op xs* (*cvc-nary-op-fold op ys*)
definition *cvc-list-both* **where** *cvc-list-both op neutral lv1 lv2* = *cvc-bin-op3 op lv1 lv2 neutral*

fun *cvc-bin-op3'* **where**
cvc-bin-op3' op (*ListVar xs*) (*ListVar* []) = *cvc-nary-op-fold op xs* |
cvc-bin-op3' op (*ListVar xs*) (*ListVar ys*) = *cvc-bin-op-fold op xs* (*cvc-nary-op-fold op ys*)
definition *cvc-list-both'* **where** *cvc-list-both' op lv1 lv2* = *cvc-bin-op3' op lv1 lv2*

fun *cvc-pairwise-op* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a cvc-ListVar* ⇒ '*a* ⇒ *bool* ⇒ *bool* **where**
cvc-pairwise-op op (*ListVar* []) *y neutral* = *neutral* |
cvc-pairwise-op op (*ListVar* [*x*]) *y neutral* = (*op x y*) |
cvc-pairwise-op op (*ListVar* (*x*#*xs*)) *y neutral* = ((*op x y*) ∧ *cvc-pairwise-op op* (*ListVar xs*) *y neutral*)
definition *cvc-pairwise-list-left* **where** *cvc-pairwise-list-left op lv y* = *cvc-pairwise-op op lv y*

fun *cvc-pairwise-op2* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*a cvc-ListVar* ⇒ *bool* ⇒ *bool* **where**
cvc-pairwise-op2 op x (*ListVar* []) *neutral* = *neutral* |
cvc-pairwise-op2 op x (*ListVar* [*y*]) *neutral* = (*op x y*) |
cvc-pairwise-op2 op x (*ListVar* (*y*#*ys*)) *neutral* = ((*op x y*) ∧ *cvc-pairwise-op op* (*ListVar ys*) *y neutral*)
definition *cvc-pairwise-list-right* **where** *cvc-pairwise-list-right op lv y* = *cvc-pairwise-op2 op lv y*

fun *cvc-pairwise-fold* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*a list* ⇒ *bool* **where**

cvc-pairwise-fold-Nil: *cvc-pairwise-fold op x* [] = *True* |
cvc-pairwise-fold-Cons: *cvc-pairwise-fold op x* (*y*#*ys*) = ((*op x y*) ∧ (*cvc-pairwise-fold op x ys*))

fun *cvc-pairwise-fold2* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
cvc-pairwise-fold2-Nil: *cvc-pairwise-fold2 op* [] = *True* |
cvc-pairwise-fold2-Cons: *cvc-pairwise-fold2 op* (*x*#*xs*) = (*cvc-pairwise-fold op x xs* ∧ *cvc-pairwise-fold2 op xs*)

lemma *cvc-nary-op-fold-transfer-h1*:
assumes *1* ≤ *n cvc-isListOp* (*ListOp op neutral*)
shows ∀ *xs*. *n* = *length xs* ⟶ *cvc-nary-op-fold op xs* = *foldr op xs neutral*
apply (*induct-tac rule*: *nat-induct-at-least*[*of 1*])
subgoal
using *assms*(*1*) **by** *simp*
subgoal
proof(*rule allI*,*rule impI*)
fix *xs*::'*a list*
assume *a0*: *1* = *length xs*
obtain *x* **where** *t0*: [*x*] = *xs*
by (*metis One-nat-def Suc-length-conv a0 length-0-conv*)
have *cvc-nary-op-fold op xs* = *x*
using *t0 cvc-nary-op-fold-Nil* **by** *force*
moreover have *foldr op xs neutral* = (*op x neutral*)
using *t0* **by** *force*
ultimately show *cvc-nary-op-fold op xs* = *foldr op xs neutral*
using *assms*(*2*) **by** *auto*
qed
subgoal for *n*
proof(*rule allI*, *rule impI*)
fix *xs*::'*a list*
assume *a0*: *1* ≤ *n*
assume *a1*: ∀ *xs*. *n* = *length xs* ⟶ *cvc-nary-op-fold op xs* = *foldr op xs neutral*
assume *a2*: *Suc n* = *length xs*
then obtain *x xss* **where** *t0*: *xs* = *x*#*xss*
by (*meson Suc-length-conv*)
show *cvc-nary-op-fold op xs* = *foldr op xs neutral*
proof(*cases xss* = [])
assume *a00*: *xss* = []
then show *cvc-nary-op-fold op xs* = *foldr op xs neutral*
using *a0 a00 a2 t0* **by** *auto*
next
assume *a01*: *xss* ≠ []
then have *cvc-nary-op-fold op xs* = *op x* (*cvc-nary-op-fold op xss*)
using *cvc-nary-op-fold.elims*(*1*) *t0*
by (*metis list.inject list.simps*(*3*))
then have *cvc-nary-op-fold op xs* = *op x* (*foldr op xss neutral*)
using *a1 a2 t0* **by** *fastforce*
then show *cvc-nary-op-fold op xs* = (*foldr op xs neutral*)

```
      by (simp add: t0)
    qed
  qed
done

lemma cvc-nary-op-fold-transfer:
  assumes 1 ≤ length xs cvc-isListOp (ListOp op neutral)
  shows cvc-nary-op-fold op xs = foldr op xs neutral
  by (meson assms(1) assms(2) cvc-nary-op-fold-transfer-h1)

lemma cvc-bin-op-fold-transfer: cvc-bin-op-fold op xs y = foldr op xs y
  apply (induction xs)
  by auto



lemma cvc-list-left-transfer:
  shows cvc-list-left op (ListVar xs) y = foldr op xs y
  by (simp add: cvc-list-left-def cvc-bin-op-fold-transfer)



lemma cvc-list-right-transfer-neutral1:
  assumes cvc-isListOp (ListOp op neutral)
  shows cvc-list-right op y (ListVar xs) = op y (foldr op xs neutral)
  using assms
  unfolding cvc-list-right-def
  apply (cases xs)
  by (simp-all add: cvc-nary-op-fold-transfer)

lemma cvc-list-right-transfer-neutral2:
  assumes cvc-isListOp (ListOp op neutral)
  shows cvc-list-right op y (ListVar xs) = (foldr op (y#xs) neutral)
  using assms
  unfolding cvc-list-right-def
  apply (cases xs)
  by (simp-all add: cvc-nary-op-fold-transfer)

lemma cvc-list-right-transfer:
  cvc-list-right op y (ListVar (xs @ [xn])) = op y (foldr op xs xn)
  apply (induction xs arbitrary: y)
  unfolding cvc-list-right-def
  apply simp
  subgoal for x1 xss yshows
    apply (cases xss)
    by simp-all
  done

lemma cvc-list-right-transfer-2:
```

assumes $xs \neq []$
shows *cvc-list-right op y* (*ListVar xs*) = (*foldr op* (*y # butlast xs*) (*last xs*))
apply (*cases rev xs*)
using *assms* **apply** *force*
by (*simp add*: *cvc-list-right-transfer*)

named-theorems *cvc-list-right-transfer-op* ‹*cvc-list-right-transfer instantiated with operator*›

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (∧) *y* (*ListVar xs*) = (∧) *y* (*foldr* (∧) *xs True*)
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (∨) *y* (*ListVar xs*) = (∨) *y* (*foldr* (∨) *xs False*)
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (+) *y* (*ListVar xs*) = (+) *y* (*foldr* (+) *xs* (*0*::*int*))
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (+) *y* (*ListVar xs*) = (+) *y* (*foldr* (+) *xs* (*0*::*'a*::*monoid-add*))
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (∗) *y* (*ListVar xs*) = (∗) *y* (*foldr* (∗) *xs* (*1*::*int*))
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (∗) *y* (*ListVar xs*) = (∗) *y* (*foldr* (∗) *xs* (*1*::*'a*::*monoid-mult*))
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma [*cvc-list-right-transfer-op*]:
cvc-list-right (@) *y* (*ListVar xs*) = (@) *y* (*foldr* (@) *xs* [])
by (*simp add*: *cvc-list-right-transfer-neutral1*)

lemma *cvc-list-both-transfer*:
assumes *cvc-isListOp* (*ListOp op neutral*)
shows *cvc-list-both op neutral* (*ListVar ys*) (*ListVar xs*) = *foldr op ys* (*foldr op xs neutral*)
using *assms*
unfolding *cvc-list-both-def*
apply(*cases* ‹(*op*,(*ListVar ys*),(*ListVar xs*),*neutral*)› *rule*: *cvc-bin-op3.cases*)
by (*simp-all add*: *cvc-bin-op-fold-transfer cvc-nary-op-fold-transfer*)

lemma *cvc-list-both-transfer'*:
assumes *xs* ≠ [] *ys* ≠ []
shows *cvc-list-both' op* (*ListVar ys*) (*ListVar xs*) = *foldr op ys* (*foldr op* (*butlast xs*) (*last xs*))
apply (*cases xs*)
apply (*case-tac*[!] *ys*)
apply *simp-all*
using *assms*
unfolding *cvc-list-both'-def*
apply *simp-all*
subgoal for *x xss y yss*
apply (*induction yss*)
apply *simp-all*
apply (*metis append-butlast-last-id assms*(*1*) *cvc-bin-op2.simps cvc-list-right-def cvc-list-right-transfer cvc-nary-op-fold.elims list.inject*)
by (*metis cvc-nary-op-fold-Cons cvc-bin-op2.simps cvc-bin-op-fold-transfer cvc-list-right-def cvc-list-right-transfer list.exhaust snoc-eq-iff-butlast*)
done

named-theorems *cvc-list-both-transfer-op* ‹*cvc-list-both-transfer instantiated with operator*›

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (∧) *True* (*ListVar ys*) (*ListVar xs*) = *foldr* (∧) *ys* (*foldr* (∧) *xs True*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (∨) *False* (*ListVar ys*) (*ListVar xs*) = *foldr* (∨) *ys* (*foldr* (∨) *xs False*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (+) (*0*::'*a*::{*linordered-field*}) (*ListVar ys*) (*ListVar xs*) = *foldr* (+) *ys* (*foldr* (+) *xs 0*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (∗) (*1*::'*a*::{*linordered-field*}) (*ListVar ys*) (*ListVar xs*) = *foldr* (∗) *ys* (*foldr* (∗) *xs 1*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (+) (*0*::*int*) (*ListVar ys*) (*ListVar xs*) = *foldr* (+) *ys* (*foldr* (+) *xs 0*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:

cvc-list-both (∗) (*1*::*int*) (*ListVar ys*) (*ListVar xs*) = *foldr* (∗) *ys* (*foldr* (∗) *xs 1*)
by (*simp add*: *cvc-list-both-transfer*)

lemma [*cvc-list-both-transfer-op*]:
cvc-list-both (@) [] (*ListVar ys*) (*ListVar xs*) = *foldr* (@) *ys* (*foldr* (@) *xs* [])
by (*simp add*: *cvc-list-both-transfer*)

lemma *cvc-list-left-Nil*: *cvc-list-left op* (*ListVar* []) *y* = *y*
unfolding *cvc-list-left-def*
by *simp*

lemma *cvc-list-right-Nil*: *cvc-list-right op y* (*ListVar* []) = *y*
unfolding *cvc-list-right-def*
by *simp*

lemma *cvc-list-both-Singleton*: *cvc-list-both op neutral* (*ListVar* [*x*]) (*ListVar* []) = *x*
cvc-list-both op neutral (*ListVar* []) (*ListVar* [*x*]) = *x*
unfolding *cvc-list-both-def*
by *auto*

lemma *cvc-list-left-Cons*: *cvc-list-left op* (*ListVar* (*x*#*xs*)) *y*
= *op x* (*cvc-list-left op* (*ListVar xs*) *y*)
unfolding *cvc-list-left-def*
apply (*induction xs*)
by *auto*

lemma *cvc-list-right-Cons*: *cvc-list-right op y* (*ListVar* (*x*#*xs*))
= *op y* (*cvc-list-right op x* (*ListVar xs*))
unfolding *cvc-list-right-def*
apply (*induction xs*)
by *auto*

lemma *cvc-list-both-Cons-0*: *cvc-list-both op neutral* (*ListVar* (*x*#*xs*)) (*ListVar* [])
= *cvc-list-right op x* (*ListVar xs*)
by (*metis cvc-nary-op-fold-Cons cvc-nary-op-fold-Nil cvc-bin-op2.simps cvc-bin-op3.simps*(*2*) *cvc-list-both-def cvc-list-right-def list.exhaust*)

lemma *cvc-list-both-Cons-1*: *cvc-list-both op neutral* (*ListVar* []) (*ListVar* (*y*#*ys*))
= *cvc-list-right op y* (*ListVar ys*)
by (*metis cvc-bin-op-fold-Nil cvc-bin-op3.simps*(*2*) *cvc-bin-op3.simps*(*4*) *cvc-list-both-Cons-0 cvc-list-both-def*)

lemma *cvc-list-both-Cons*: *cvc-list-both op neutral* (*ListVar* (*x*#*xs*)) (*ListVar* (*y*#*ys*))
= *op x* (*cvc-list-both op neutral* (*ListVar xs*) (*ListVar* (*y*#*ys*)))
unfolding *cvc-list-both-def*
apply (*induction xs*)
by *auto*

lemma *bool-and-flatten-test-help*:
shows *foldr* (∧) *xss* ((*b* ∧ *foldr* (∧) *yss True*) ∧ *foldr* (∧) *zss True*) = *foldr* (∧) *xss* (*foldr* (∧) *zss* (*b* ∧ *foldr* (∧) *yss True*))
apply (*induction xss*)
apply *simp-all*
apply (*induct-tac*[!] *yss*)
apply *simp-all*
apply (*induct-tac*[!] *zss*)
apply *simp-all*
apply *blast*
by *blast*

lemma *bool-and-flatten-test*:
fixes *xs ys zs* :: *bool cvc-ListVar*
shows (*cvc-list-left* (∧) *xs* (*cvc-list-right* (∧) (*cvc-list-right* (∧) *b ys*) *zs*))
= (*cvc-list-left* (∧) *xs* (*cvc-list-left* (∧) *zs* (*cvc-list-right* (∧) *b ys*)))
apply (*cases xs*)
apply (*cases ys*)
apply (*cases zs*)
subgoal for *xss yss zss*
apply *simp*
unfolding *cvc-list-left-transfer*
unfolding *cvc-list-right-transfer-op*
using *bool-and-flatten-test-help* **by** *auto*
done

lemma *bool-and-flatten-test-reconstruction*: *a* ∧ (*c* ∧ (*b* ∧ *d*)) ⟶ (*a* ∧ (*c* ∧ (*b* ∧ *d*)))
using *bool-and-flatten-test*[*of ListVar* [*a*,*c*] *b ListVar* [*d*] *ListVar* []]
unfolding *cvc-list-right-Nil cvc-list-left-Nil*
unfolding *cvc-list-right-Cons cvc-list-left-Cons*
unfolding *cvc-list-right-Nil cvc-list-left-Nil*
by *simp*

lemma *foldr-or-neutral* [*simp*]: *foldr* (∨) *xs True* = *True*
apply (*induction xs*)
by *auto*

lemma *foldr-and-neutral* [*simp*]: *foldr* (∧) *xs False* = *False*
apply (*induction xs*)
by *auto*

lemmas *cvc-rewrites-fold* = *append.right-neutral append-Nil*
append.assoc append.right-neutral fold-append-concat-rev foldr-conv-fold
append-eq-append-conv concat-append append.left-neutral append-Nil2

end

theory *Alethe-Boolean-Rewrites-Lemmas*
imports *Alethe-Rare-Nary-Ops*
begin

lemma *bool-or-taut-lemma*:
$w \vee a \vee foldr\ (\vee)\ yss\ (\neg\ w)$
$a \vee foldr\ (\vee)\ xss\ (w \vee aa \vee foldr\ (\vee)\ yss\ (\neg\ w))$
$foldr\ (\vee)\ xss\ (w \vee foldr\ (\vee)\ yss\ (w \longrightarrow a \vee foldr\ (\vee)\ zss\ False))$
apply *force*
apply (*metis* (*mono-tags*, *opaque-lifting*) *foldr-or-neutral*)
by (*metis*(*full-types*) *foldr-or-neutral*)

lemma *bool-and-conf-lemma*:
$\neg\ foldr\ (\wedge)\ xss\ (w \wedge aa \wedge foldr\ (\wedge)\ yss\ (\neg\ w))$
$\neg\ foldr\ (\wedge)\ xss\ (w \wedge foldr\ (\wedge)\ yss\ (\neg\ w \wedge a \wedge foldr\ (\wedge)\ zss\ True))$
apply (*metis* (*mono-tags*, *opaque-lifting*) *foldr-and-neutral*)
by (*metis*(*full-types*) *foldr-and-neutral*)

lemma *bool-and-conf2-lemma*:
$\neg\ foldr\ (\wedge)\ xss\ ((\neg\ w) \wedge aa \wedge foldr\ (\wedge)\ yss\ w)$
$\neg\ foldr\ (\wedge)\ xss\ ((\neg\ w) \wedge foldr\ (\wedge)\ yss\ (w \wedge a \wedge foldr\ (\wedge)\ zss\ True))$
apply (*metis* (*mono-tags*, *opaque-lifting*) *foldr-and-neutral*)
by (*metis*(*full-types*) *foldr-and-neutral*)

lemma *bool-and-dup-lemma*:
$(b \wedge a \wedge foldr\ (\wedge)\ yss\ b) = (b \wedge a \wedge foldr\ (\wedge)\ yss\ True)$
$(a \wedge foldr\ (\wedge)\ xss\ (b \wedge aa \wedge foldr\ (\wedge)\ yss\ b)) = (a \wedge foldr\ (\wedge)\ xss\ (b \wedge aa \wedge foldr\ (\wedge)\ yss\ True))$
$foldr\ (\wedge)\ xss\ (b \wedge foldr\ (\wedge)\ yss\ (b \wedge a \wedge foldr\ (\wedge)\ zss\ True)) = foldr\ (\wedge)\ xss\ (b \wedge foldr\ (\wedge)\ yss\ (a \wedge foldr\ (\wedge)\ zss\ True))$
apply *auto*[*1*]
apply (*metis*(*full-types*))
by (*metis*(*full-types*))

lemma *bool-and-flatten-lemma*:
$(b \wedge a \wedge foldr\ (\wedge)\ yss\ True \wedge aa \wedge foldr\ (\wedge)\ zss\ True) = (b \wedge a \wedge foldr\ (\wedge)\ yss\ (aa \wedge foldr\ (\wedge)\ zss\ True))$
by (*metis*(*full-types*) *foldr-and-neutral*)

lemma *bool-or-dup-lemma*:
$(b \vee a \vee foldr\ (\vee)\ yss\ b) = (b \vee a \vee foldr\ (\vee)\ yss\ False)$
$(a \vee foldr\ (\vee)\ xss\ (b \vee aa \vee foldr\ (\vee)\ yss\ b)) = (a \vee foldr\ (\vee)\ xss\ (b \vee aa \vee foldr\ (\vee)\ yss\ False))$
$foldr\ (\vee)\ xss\ (b \vee foldr\ (\vee)\ yss\ (b \vee a \vee foldr\ (\vee)\ zss\ False)) = foldr\ (\vee)\ xss\ (b \vee foldr\ (\vee)\ yss\ (a \vee foldr\ (\vee)\ zss\ False))$
apply *auto*[*1*]
apply (*metis*(*full-types*))

by (*metis*(*full-types*))

lemma *bool-or-flatten-lemma*:
(*b* ∨ *a* ∨ *foldr* (∨) *yss False* ∨ *aa* ∨ *foldr* (∨) *zss False*) = (*b* ∨ *a* ∨ *foldr* (∨) *yss* (*aa* ∨ *foldr* (∨) *zss False*))
(*a* ∨ *foldr* (∨) *xss* (*b* ∨ *aa* ∨ *foldr* (∨) *yss False* ∨ *aaa* ∨ *foldr* (∨) *zss False*)) =
(*a* ∨ *foldr* (∨) *xss* (*b* ∨ *aa* ∨ *foldr* (∨) *yss* (*aaa* ∨ *foldr* (∨) *zss False*)))
apply (*metis*(*full-types*) *foldr-or-neutral*)
by (*metis*(*full-types*) *foldr-or-neutral*)

lemma *bool-or-and-distrib-lemma*:
shows *zs* = *ListVar zss* ⟹
ys = *ListVar yss* ⟹
(*y1* ∧ *y2* ∧ *foldr* (∧) *yss True* ∨ *z1* ∨ *foldr* (∨) *zss False*) =
((*y1* ∨ *z1* ∨ *foldr* (∨) *zss False*) ∧ (*y2* ∧ *foldr* (∧) *yss True* ∨ *z1* ∨ *foldr* (∨) *zss False*))
using *disj-conj-distribR* **by** *simp*

end

theory *Alethe-Boolean-Rewrites*
imports *Alethe-Boolean-Rewrites-Lemmas*
begin

named-theorems *rewrite-bool-double-not-elim* ‹*automatically-generated*›

lemma [*rewrite-bool-double-not-elim*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (¬ ¬ *t*) = *t*
by *auto*

named-theorems *rewrite-bool-not-true* ‹*automatically-generated*›

lemma [*rewrite-bool-not-true*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* = *False*) ⟹ (¬ *t*) = *True*
by *auto*

named-theorems *rewrite-bool-not-false* ‹*automatically-generated*›

lemma [*rewrite-bool-not-false*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* = *True*) ⟹ (¬ *t*) = *False*
by *auto*

named-theorems *rewrite-bool-eq-true* ‹*automatically-generated*›

lemma [*rewrite-bool-eq-true*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* = *True*) = *t*
by *auto*

named-theorems *rewrite-bool-eq-false* ‹*automatically-generated*›

lemma [*rewrite-bool-eq-false*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* = *False*) = (¬ *t*)
by *auto*

named-theorems *rewrite-bool-eq-nrefl* ‹*automatically-generated*›

lemma [*rewrite-bool-eq-nrefl*]:
fixes *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ (*x* = (¬ *x*)) = *False*
by *auto*

named-theorems *rewrite-bool-impl-false1* ‹*automatically-generated*›

lemma [*rewrite-bool-impl-false1*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* ⟶ *False*) = (¬ *t*)
by *auto*

named-theorems *rewrite-bool-impl-false2* ‹*automatically-generated*›

lemma [*rewrite-bool-impl-false2*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*False* ⟶ *t*) = *True*
by *auto*

named-theorems *rewrite-bool-impl-true1* ‹*automatically-generated*›

lemma [*rewrite-bool-impl-true1*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* ⟶ *True*) = *True*
by *auto*

named-theorems *rewrite-bool-impl-true2* ‹*automatically-generated*›

lemma [*rewrite-bool-impl-true2*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*True* ⟶ *t*) = *t*
by *auto*

named-theorems *rewrite-bool-impl-elim* ‹*automatically-generated*›

lemma [*rewrite-bool-impl-elim*]:
fixes *t*::*bool* **and** *s*::*bool*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*t* ⟶ *s*) = (¬ *t* ∨ *s*)
by *auto*

named-theorems *rewrite-bool-dual-impl-eq* ‹*automatically-generated*›

lemma [*rewrite-bool-dual-impl-eq*]:
fixes *t*::*bool* **and** *s*::*bool*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ ((*t* ⟶ *s*) ∧ (*s* ⟶ *t*)) = (*t* = *s*)
by *auto*

named-theorems *rewrite-bool-and-conf* ‹*automatically-generated*›

lemma [*rewrite-bool-and-conf*]:
fixes *xs*::*bool cvc-ListVar* **and** *w*::*bool* **and** *ys*::*bool cvc-ListVar* **and** *zs*::*bool cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined xs w ys zs*) ⟹ *cvc-list-left* (∧) *xs*
(*w* ∧
cvc-list-left (∧) *ys* (*cvc-list-right* (∧) (¬ *w*) *zs*)) =
False
apply (*cases zs*)
apply (*cases ys*)
apply (*cases xs*)
subgoal for *zss yss xss*
apply (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op cvc-list-both-transfer-op*)
apply (*induction zss*)
apply *simp-all*
apply (*induction yss*)
apply *simp-all*
apply (*induction xss*)
apply *simp-all*
by (*simp-all add*: *bool-and-conf-lemma*)
done

named-theorems *rewrite-bool-and-conf2* ‹*automatically-generated*›

lemma [*rewrite-bool-and-conf2*]:
fixes *xs*::*bool cvc-ListVar* **and** *w*::*bool* **and** *ys*::*bool cvc-ListVar* **and** *zs*::*bool cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined xs w ys zs*) ⟹ *cvc-list-left* (∧) *xs* ((¬ *w*) ∧ *cvc-list-left* (∧) *ys* (*cvc-list-right* (∧) *w zs*)) = *False*
apply (*cases zs*)
apply (*cases ys*)
apply (*cases xs*)
subgoal for *zss yss xss*
apply (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op cvc-list-both-transfer-op*)
apply (*induction zss*)
apply *simp-all*
apply (*induction yss*)
apply *simp-all*
apply (*induction xss*)
apply *simp-all*
apply (*simp-all add*: *bool-and-conf2-lemma*)
by *force*
done

named-theorems *rewrite-bool-or-taut* ‹*automatically-generated*›

lemma [*rewrite-bool-or-taut*]:
fixes *xs*::*bool cvc-ListVar* **and** *w*::*bool* **and** *ys*::*bool cvc-ListVar* **and** *zs*::*bool cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined xs w ys zs*) ⟹ *cvc-list-left* (∨) *xs* (*w* ∨ *cvc-list-left* (∨) *ys* (*cvc-list-right* (∨) (¬ *w*) *zs*)) = *True*
apply (*cases zs*)
apply (*cases ys*)
apply (*cases xs*)
subgoal for *zss yss xss*
apply (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op cvc-list-both-transfer-op*)
apply (*induction zss*)
apply *simp-all*
apply (*induction yss*)
apply *simp-all*
apply (*induction xss*)
apply *simp-all*
by (*simp-all add*: *bool-or-taut-lemma*)
done

named-theorems *rewrite-bool-or-taut2* ‹*automatically-generated*›

lemma [*rewrite-bool-or-taut2*]:
fixes *xs*::*bool cvc-ListVar* **and** *w*::*bool* **and** *ys*::*bool cvc-ListVar* **and** *zs*::*bool cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined xs w ys zs*) ⟹ *cvc-list-left* (∨) *xs* ((¬ *w*) ∨ *cvc-list-left* (∨) *ys* (*cvc-list-right* (∨) *w zs*)) = *True*
apply (*cases zs*)
apply (*cases ys*)
apply (*cases xs*)
subgoal for *zss yss xss*
by (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op cvc-list-both-transfer-op*)
done

named-theorems *rewrite-bool-or-de-morgan* ‹*automatically-generated*›

lemma [*rewrite-bool-or-de-morgan*]:
fixes *zs* :: *bool cvc-ListVar* **and** *y* :: *bool* **and** *x* :: *bool*
shows *NO-MATCH cvc-a* (*undefined x y zs*) ⟹ (¬ (*x* ∨ *cvc-list-right* (∨) *y zs*)) = (¬ *x* ∧ ¬ *cvc-list-right* (∨) *y zs*)
apply *simp ?*
done

named-theorems *rewrite-bool-implies-de-morgan* ‹*automatically-generated*›

lemma [*rewrite-bool-implies-de-morgan*]:
fixes *y* :: *bool* **and** *x* :: *bool*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (¬ (*x* ⟶ *y*)) = (*x* ∧ ¬ *y*)
apply *simp ?*
done

named-theorems *rewrite-bool-and-de-morgan* ‹*automatically-generated*›

lemma [*rewrite-bool-and-de-morgan*]:
fixes *zs* :: *bool cvc-ListVar* **and** *y* :: *bool* **and** *x* :: *bool*
shows *NO-MATCH cvc-a* (*undefined x y zs*) ⟹ (¬ (*x* ∧ *cvc-list-right* (∧) *y zs*)) = (¬ *x* ∨ ¬ *cvc-list-right* (∧) *y zs*)
apply *simp ?*
done

named-theorems *rewrite-bool-or-and-distrib* ‹*automatically-generated*›

lemma [*rewrite-bool-or-and-distrib*]:
fixes *zs* :: *bool cvc-ListVar* **and** *ys* :: *bool cvc-ListVar* **and** *y2* :: *bool* **and** *y1* :: *bool* **and** *z1* :: *bool*
shows *NO-MATCH cvc-a* (*undefined y1 y2 ys z1 zs*) ⟹
((*y1* ∧ *cvc-list-right* (∧) *y2 ys*) ∨ (*cvc-list-right* (∨) *z1 zs*)) =
((*y1* ∨ *cvc-list-right* (∨) *z1 zs*) ∧
((*cvc-list-right* (∧) *y2 ys*) ∨ *cvc-list-right* (∨) *z1 zs*))
apply (*cases zs*)
apply (*cases ys*)
subgoal for *zss yss*
apply (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op cvc-list-both-transfer-op*)
by (*simp-all add*: *bool-or-and-distrib-lemma*)
done

named-theorems *rewrite-bool-implies-or-distrib* ‹*automatically-generated*›

lemma [*rewrite-bool-implies-or-distrib*]:
fixes *y1* :: *bool* **and** *y2* :: *bool* **and** *ys* :: *bool cvc-ListVar* **and** *z* :: *bool*
shows *NO-MATCH cvc-a* (*undefined y1 y2 ys z*) ⟹ (*y1* ∨ *cvc-list-right* (∨) *y2 ys* ⟶ *z*) =
((*y1* ⟶ *z*) ∧
(*cvc-list-right* (∨) *y2 ys* ⟶ *z*))
apply *simp ?*
done

named-theorems *rewrite-bool-xor-refl* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-refl*]:
fixes *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ (*x* ≠ *x*) = *False*
by *auto*

named-theorems *rewrite-bool-xor-nrefl* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-nrefl*]:
fixes *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ (*x* ≠ (¬ *x*)) = *True*
by *auto*

named-theorems *rewrite-bool-xor-false* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-false*]:
fixes *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ (*x* ≠ *False*) = *x*

by *auto*

named-theorems *rewrite-bool-xor-true* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-true*]:
fixes *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ (*x* ≠ *True*) = (¬ *x*)
by *auto*

named-theorems *rewrite-bool-xor-comm* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-comm*]:
fixes *x*::*bool* **and** *y*::*bool*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (*x* ≠ *y*) = (¬(*y* = *x*))
by *auto*

named-theorems *rewrite-bool-xor-elim* ‹*automatically-generated*›

lemma [*rewrite-bool-xor-elim*]:
fixes *x*::*bool* **and** *y*::*bool*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (*x* ≠ *y*) = ((¬*x*) = *y*)
by *auto*

named-theorems *rewrite-bool-not-xor-elim* ‹*automatically-generated*›

lemma [*rewrite-bool-not-xor-elim*]:
fixes *y* :: *bool* **and** *x* :: *bool*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (¬ (*x* ≠ *y*)) = (*x* = *y*)
apply *simp ?*
done

named-theorems *rewrite-bool-not-eq-elim1* ‹*automatically-generated*›

lemma [*rewrite-bool-not-eq-elim1*]:
fixes *y* :: *bool* **and** *x* :: *bool*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (*x* ≠ *y*) = ((¬ *x*) = *y*)
apply *simp ?*
apply *auto ?*
done

named-theorems *rewrite-bool-not-eq-elim2* ‹*automatically-generated*›

lemma [*rewrite-bool-not-eq-elim2*]:

fixes $y :: bool$ **and** $x :: bool$
shows *NO-MATCH cvc-a* $(undefined\ x\ y) \Longrightarrow (x \neq y) = (x = (\neg\ y))$
apply *simp ?*
apply *auto ?*
done

named-theorems *rewrite-ite-neg-branch* ‹*automatically-generated*›

lemma [*rewrite-ite-neg-branch*]:
fixes c::*bool* **and** x::*bool* **and** y::*bool*
shows *NO-MATCH cvc-a* $(undefined\ c\ x\ y) \Longrightarrow (\neg\ y) = x \longrightarrow (if\ c\ then\ x\ else\ y) = (c = x)$
by *auto*

named-theorems *rewrite-ite-then-true* ‹*automatically-generated*›

lemma [*rewrite-ite-then-true*]:
fixes c::*bool* **and** x::*bool*
shows *NO-MATCH cvc-a* $(undefined\ c\ x) \Longrightarrow (if\ c\ then\ True\ else\ x) = (c \vee x)$
by *auto*

named-theorems *rewrite-ite-else-false* ‹*automatically-generated*›

lemma [*rewrite-ite-else-false*]:
fixes c::*bool* **and** x::*bool*
shows *NO-MATCH cvc-a* $(undefined\ c\ x) \Longrightarrow (if\ c\ then\ x\ else\ False) = (c \wedge x)$
by *auto*

named-theorems *rewrite-ite-then-false* ‹*automatically-generated*›

lemma [*rewrite-ite-then-false*]:
fixes c::*bool* **and** x::*bool*
shows *NO-MATCH cvc-a* $(undefined\ c\ x) \Longrightarrow (if\ c\ then\ False\ else\ x) = (\neg\ c \wedge x)$
by *auto*

named-theorems *rewrite-ite-else-true* ‹*automatically-generated*›

lemma [*rewrite-ite-else-true*]:
fixes c::*bool* **and** x::*bool*
shows *NO-MATCH cvc-a* $(undefined\ c\ x) \Longrightarrow (if\ c\ then\ x\ else\ True) = (\neg\ c \vee x)$
by *auto*

named-theorems *rewrite-ite-then-lookahead-self* ‹*automatically-generated*›

lemma [*rewrite-ite-then-lookahead-self*]:
fixes *c*::*bool* **and** *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x*) ⟹ (*if c then c else x*) = (*if c then True else x*)
by *auto*

named-theorems *rewrite-ite-else-lookahead-self* ‹*automatically-generated*›

lemma [*rewrite-ite-else-lookahead-self*]:
fixes *c*::*bool* **and** *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x*) ⟹ (*if c then x else c*) = (*if c then x else False*)
by *auto*

named-theorems *rewrite-ite-then-lookahead-not-self* ‹*automatically-generated*›

lemma [*rewrite-ite-then-lookahead-not-self*]:
fixes *c*::*bool* **and** *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x*) ⟹ (*if c then* ¬*c else x*) = (*if c then False else x*)
by *auto*

named-theorems *rewrite-ite-else-lookahead-not-self* ‹*automatically-generated*›

lemma [*rewrite-ite-else-lookahead-not-self*]:
fixes *c*::*bool* **and** *x*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x*) ⟹ (*if c then x else* ¬*c*) = (*if c then x else True*)
by *auto*

named-theorems *rewrite-ite-expand* ‹*automatically-generated*›

lemma [*rewrite-ite-expand*]:
fixes *c*::*bool* **and** *x*::*bool* **and** *y*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x y*) ⟹ (*if c then x else y*) = ((¬*c* ∨ *x*) ∧ (*c* ∨ *y*))
by *auto*

named-theorems *rewrite-bool-not-ite-elim* ‹*automatically-generated*›

lemma [*rewrite-bool-not-ite-elim*]:

fixes *c*::*bool* **and** *x*::*bool* **and** *y*::*bool*
shows *NO-MATCH cvc-a* (*undefined c x y*) ⟹ (¬(*if c then x else y*)) = (*if c then* ¬*x else* ¬*y*)
by *auto*

end

theory *Alethe-Builtin-Rewrites*
imports *Alethe-Rare-Nary-Ops*
begin

named-theorems *rewrite-ite-true-cond* ‹*automatically-generated*›

lemma [*rewrite-ite-true-cond*]:
fixes *x*::'*a*::*type* **and** *y*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (*if True then x else y*) = *x*
by *auto*

named-theorems *rewrite-ite-false-cond* ‹*automatically-generated*›

lemma [*rewrite-ite-false-cond*]:
fixes *x*::'*a*::*type* **and** *y*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined x y*) ⟹ (*if False then x else y*) = *y*
by *auto*

named-theorems *rewrite-ite-not-cond* ‹*automatically-generated*›

lemma [*rewrite-ite-not-cond*]:
fixes *c*::*bool* **and** *x*::'*a*::*type* **and** *y*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x y*) ⟹ (*if* ¬ *c then x else y*) = (*if c then y else x*)
by *auto*

named-theorems *rewrite-ite-eq-branch* ‹*automatically-generated*›

lemma [*rewrite-ite-eq-branch*]:
fixes *c*::*bool* **and** *x*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x*) ⟹ (*if c then x else x*) = *x*
by *auto*

named-theorems *rewrite-ite-then-lookahead* ‹*automatically-generated*›

lemma [*rewrite-ite-then-lookahead*]:
fixes *c*::*bool* **and** *x*::*'a*::*type* **and** *y*::*'a*::*type* **and** *z*::*'a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x y z*) ⟹ (*if c then if c then x else y else z*) = (*if c then x else z*)
by *auto*

named-theorems *rewrite-ite-else-lookahead* ‹*automatically-generated*›

lemma [*rewrite-ite-else-lookahead*]:
fixes *c*::*bool* **and** *x*::*'a*::*type* **and** *y*::*'a*::*type* **and** *z*::*'a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x y z*) ⟹ (*if c then x else if c then y else z*) = (*if c then x else z*)
by *auto*

named-theorems *rewrite-ite-then-neg-lookahead* ‹*automatically-generated*›

lemma [*rewrite-ite-then-neg-lookahead*]:
fixes *c*::*bool* **and** *x*::*'a*::*type* **and** *y*::*'a*::*type* **and** *z*::*'a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x y z*) ⟹ (*if c then if* ¬ *c then x else y else z*) = (*if c then y else z*)
by *auto*

named-theorems *rewrite-ite-else-neg-lookahead* ‹*automatically-generated*›

lemma [*rewrite-ite-else-neg-lookahead*]:
fixes *c*::*bool* **and** *x*::*'a*::*type* **and** *y*::*'a*::*type* **and** *z*::*'a*::*type*
shows *NO-MATCH cvc-a* (*undefined c x y z*) ⟹ (*if c then x else if* ¬ *c then y else z*) = (*if c then x else y*)
by *auto*

end

theory *Alethe-Arith-Rewrites-Lemmas*
imports *Alethe-Rare-Nary-Ops*
begin

lemma *arith-plus-zero-lemma*:
shows *foldr* (+) *xs* (*0*::*int*) + (*a* + *foldr* (+) *ys 0*) = *foldr* (+) *xs* (*a* + *foldr* (+) *ys 0*)
apply (*induction xs*)
by *simp-all*

lemma *arith-mul-one-lemma*:
shows *a* = (*0*::*int*) ∨ *foldr* (∗) *ts* (*1*::*int*) ∗ (*aa* ∗ *foldr* (∗) *ss* (*1*::*int*)) = *foldr*

(∗) *ts* (*aa* ∗ *foldr* (∗) *ss* (*1*::*int*))
apply (*induction ts*)
apply *simp-all*
by *blast*

lemma *arith-mul-zero-lemma*:
shows
a = (*0*::*int*) ∨ *foldr* (∗) *ts* (*0*::*int*) = (*0*::*int*)
foldr (∗) *ts* (*0*::*int*) = (*0*::*int*) ∨ *a* = (*0*::*int*) ∨ *foldr* (∗) *ss* (*1*::*int*) = (*0*::*int*)
apply (*metis arith-mul-one-lemma mult-zero-left mult-zero-right*)
by (*metis arith-mul-one-lemma times-int-code*(*1*) *times-int-code*(*2*))

lemma *arith-plus-flatten-lemma*:
shows
foldr (+) *xss* (*w* + (*aa* + *foldr* (+) *yss* (*0*::*int*))) = *foldr* (+) *xss w* + (*aa* + *foldr* (+) *yss* (*0*::*int*))
foldr (+) *xss* (*w* + *foldr* (+) *yss* (*0*::*int*)) = *foldr* (+) *xss w* + *foldr* (+) *yss* (*0*::*int*)
apply (*metis* (*no-types*, *opaque-lifting*) *add.assoc add-cancel-right-right arith-mul-one-lemma arith-plus-zero-lemma foldr.simps*(*1*) *mult-zero-left ring-class.ring-distribs*(*1*))
by (*metis ab-semigroup-add-class.add-ac*(*1*) *add.right-neutral arith-plus-zero-lemma foldr.simps*(*1*) *id-apply*)

lemma *arith-mult-flatten-lemma*:
shows
a = (*0*::*int*) ∨ *foldr* (∗) *xss* (*w* ∗ (*aa* ∗ *foldr* (∗) *yss* (*1*::*int*))) = *foldr* (∗) *xss w* ∗ (*aa* ∗ *foldr* (∗) *yss* (*1*::*int*))
a = (*0*::*int*) ∨ *foldr* (∗) *zss* (*1*::*int*) = (*0*::*int*) ∨ *foldr* (∗) *xss* (*w* ∗ *foldr* (∗) *yss* (*1*::*int*)) = *foldr* (∗) *xss w* ∗ *foldr* (∗) *yss* (*1*::*int*)
apply (*case-tac* [!] *a* = *0*)
apply *simp-all*
apply (*metis* (*no-types*, *opaque-lifting*) *arith-mul-one-lemma foldr.simps*(*1*) *id-apply mult.assoc mult.right-neutral*)
apply (*case-tac* [!] *foldr* (∗) *zss* (*1*::*int*) = (*0*::*int*))
apply *simp-all*
proof −
have ∀ *i is ia*. *foldr* (∗) *is* (*1*::*int*) ∗ *ia* = *foldr* (∗) *is ia* ∨ (*0*::*int*) = *i*
by (*metis arith-mul-one-lemma foldr.simps*(*1*) *id-apply mult.right-neutral*)
then show *foldr* (∗) *xss* (*w* ∗ *foldr* (∗) *yss 1*) = *foldr* (∗) *xss w* ∗ *foldr* (∗) *yss 1*
by (*metis* (*no-types*) *mult.left-commute mult.right-neutral*)
qed

lemma *arith-mult-dist-lemma*:
shows
x ∗ (*y* + *z*) = *x* ∗ *y* + *x* ∗ *z*
x ∗ (*y* + *z* + (*a* + *foldr* (+) *ws* (*0*::*int*))) = *x* ∗ *y* + *x* ∗ (*z* + (*a* + *foldr* (+) *ws* (*0*::*int*)))
by (*simp-all add*: *distrib-left*)

lemma *arith-plus-cancel1-lemma*:
shows
foldr (+) *ts x* − *x* = *foldr* (+) *ts* (*0*::*int*)
foldr (+) *ts x* + (*a* + *foldr* (+) *ss* (*0*::*int*)) − *x* = *foldr* (+) *ts* (*a* + *foldr* (+) *ss* (*0*::*int*))
foldr (+) *ts x* + *foldr* (+) *ss* (*0*::*int*) − *x* = *foldr* (+) *ts* (*foldr* (+) *ss* (*0*::*int*))
apply (*induction ts*)
by *simp-all*

lemma *arith-plus-cancel2-lemma*:
shows
foldr (+) *ts* (− *x*) + *x* = *foldr* (+) *ts* (*0*::*int*)
foldr (+) *ts* (− *x*) + (*a* + *foldr* (+) *ss* (*0*::*int*)) + *x* = *foldr* (+) *ts* (*a* + *foldr* (+) *ss* (*0*::*int*))
foldr (+) *ts* (− *x*) + *foldr* (+) *ss* (*0*::*int*) + *x* = *foldr* (+) *ts* (*foldr* (+) *ss* (*0*::*int*))
apply (*induction ts*)
by *simp-all*

end

theory *Alethe-Arith-Rewrites*
imports *Alethe-Arith-Rewrites-Lemmas*
begin

named-theorems *rewrite-arith-elim-gt* ‹*automatically-generated*›

lemma [*rewrite-arith-elim-gt*]:
fixes *t*::*'a*::*linorder* **and** *s*::*'a*::*linorder*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*s* < *t*) = (¬ *t* ≤ *s*)
by *auto*

named-theorems *rewrite-arith-elim-lt* ‹*automatically-generated*›

lemma [*rewrite-arith-elim-lt*]:
fixes *t*::*'a*::*linorder* **and** *s*::*'a*::*linorder*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*t* < *s*) = (¬ *s* ≤ *t*)
by *auto*

named-theorems *rewrite-arith-elim-int-gt* ‹*automatically-generated*›

lemma [*rewrite-arith-elim-int-gt*]:
fixes *t*::*int* **and** *s*::*int*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*t* > *s*) = (*t* ≥ (*s* + *1*))
by *auto*

named-theorems *rewrite-arith-elim-int-lt* ‹*automatically-generated*›

lemma [*rewrite-arith-elim-int-lt*]:
fixes t::*int* **and** s::*int*
shows *NO-MATCH cvc-a* (*undefined t s*) $\Longrightarrow$ $(t < s) = (s \geq (t + 1))$
by *auto*

named-theorems *rewrite-arith-elim-leq* ‹*automatically-generated*›

lemma [*rewrite-arith-elim-leq*]:
fixes t::$'a$::*linorder* **and** s::$'a$::*linorder*
shows *NO-MATCH cvc-a* (*undefined t s*) $\Longrightarrow$ $(t \leq s) = (t \leq s)$
by *auto*

named-theorems *rewrite-arith-leq-norm* ‹*automatically-generated*›

lemma [*rewrite-arith-leq-norm*]:
fixes t::*int* **and** s::*int*
shows *NO-MATCH cvc-a* (*undefined t s*) $\Longrightarrow$ $(t \leq s) = (\neg\ s + (1\text{::}int) \leq t)$
by *auto*

named-theorems *rewrite-arith-geq-tighten* ‹*automatically-generated*›

lemma [*rewrite-arith-geq-tighten*]:
fixes t::*int* **and** s::*int*
shows *NO-MATCH cvc-a* (*undefined t s*) $\Longrightarrow$ $(\neg\ s \leq t) = (t + (1\text{::}int) \leq s)$
by *auto*

named-theorems *rewrite-arith-geq-norm1-int* ‹*automatically-generated*›

lemma [*rewrite-arith-geq-norm1-int*]:
fixes t::*int* **and** s::*int*
shows *NO-MATCH cvc-a* (*undefined t s*) $\Longrightarrow$ $(s \leq t) = ((0\text{::}int) \leq t - s)$
by *auto*

named-theorems *rewrite-arith-eq-elim-int* ‹*automatically-generated*›

lemma [*rewrite-arith-eq-elim-int*]:
fixes t::*int* **and** s::*int*
shows *NO-MATCH cvc-a* (*undefined t s*)
$\Longrightarrow$ $((t = s) = (t \geq s \wedge t \leq s))$
by *auto*

named-theorems *rewrite-arith-mod-over-mod* ‹*automatically-generated*›

lemma [*rewrite-arith-mod-over-mod*]:
fixes *c*::*int* **and** *ts*::*int cvc-ListVar* **and** *r*::*int* **and** *ss*::*int cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined c ts r ss*)
⟹ ¬(*c*=*0*)
⟹ *SMT.z3mod* (*cvc-list-left* (+) *ts* (*cvc-list-right* (+) (*SMT.z3mod r c*) *ss*)) *c*
= *SMT.z3mod* (*cvc-list-left* (+) *ts* (*cvc-list-right* (+) *r ss*)) *c*
apply (*cases ts*)
apply (*cases ss*)
subgoal for *ts*′ *ss*′
unfolding *SMT.z3mod-def*
apply *simp-all*
apply (*simp add*: *cvc-list-left-transfer cvc-list-right-transfer-op*(*3*))
apply (*induction ts*′ *arbitrary*: *ts*)
apply *simp-all*
apply (*induction ss*′ *arbitrary*: *ss*)
apply *simp-all*
apply (*meson mod-add-left-eq*)
by (*metis* (*no-types*, *lifting*) *mod-add-right-eq*)
done

named-theorems *rewrite-arith-divisible-elim* ‹*automatically-generated*›

lemma [*rewrite-arith-divisible-elim*]:
fixes *n*::*int* **and** *t*::*int*
shows *NO-MATCH cvc-a* (*undefined n t*)
⟹ ¬(*n* = *0*) ⟹ ((*n dvd t*) = (*SMT.z3mod t n* = *0*))
apply (*simp add*: *SMT.z3mod-def*)
by *auto*

named-theorems *rewrite-arith-geq-ite-lift* ‹›

lemma [*rewrite-arith-geq-ite-lift*]:
fixes *C*::*bool* **and** *t*::′*a*::*linordered-idom* **and** *r*::′*a*::*linordered-idom* **and** *s*::′*a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined C t s r*)
⟹ (((*if C then t else s*) ≥ *r*) = (*if C then* (*t* ≥ *r*) *else* (*s* ≥ *r*)))
by *auto*

named-theorems *rewrite-arith-leq-ite-lift* ‹›

lemma [*rewrite-arith-leq-ite-lift*]:
fixes *C*::*bool* **and** *t*::′*a*::*linordered-idom* **and** *r*::′*a*::*linordered-idom* **and** *s*::′*a*::*linordered-idom*

shows *NO-MATCH cvc-a (undefined C t s r)*
$\Longrightarrow$ (((*if C then t else s*) $\leq$ *r*) = (*if C then* (*t* $\leq$ *r*) *else* (*s* $\leq$ *r*)))
by *auto*

named-theorems *rewrite-arith-min-lt1* ‹›

lemma [*rewrite-arith-min-lt1*]:
fixes *t*::'*a*::*linordered-idom* **and** *s*::'*a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined t s*)
$\Longrightarrow$ ((*if* (*t* < *s*) *then t else s*) $\leq$ *t*) = *True*
by *auto*

named-theorems *rewrite-arith-min-lt2* ‹›

lemma [*rewrite-arith-min-lt2*]:
fixes *t*::'*a*::*linordered-idom* **and** *s*::'*a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined t s*)
$\Longrightarrow$ ((*if* (*t* < *s*) *then t else s*) $\leq$ *s*) = *True*
by *auto*

named-theorems *rewrite-arith-max-geq1* ‹›

lemma [*rewrite-arith-max-geq1*]:
fixes *t*::'*a*::*linordered-idom* **and** *s*::'*a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined t s*)
$\Longrightarrow$ ((*if* (*t* $\geq$ *s*) *then t else s*) $\geq$ *t*) = *True*
by *auto*

named-theorems *rewrite-arith-max-geq2* ‹›

lemma [*rewrite-arith-max-geq2*]:
fixes *t*::'*a*::*linordered-idom* **and** *s*::'*a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined t s*)
$\Longrightarrow$ ((*if* (*t* $\geq$ *s*) *then t else s*) $\geq$ *s*) = *True*
by *auto*

end

theory *Alethe-UF-Rewrites*
imports *Alethe-Rare-Nary-Ops*
begin

named-theorems *rewrite-eq-refl* ‹*automatically-generated*›

lemma [*rewrite-eq-refl*]:
fixes *t*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹ (*t* = *t*) = *True*
by *auto*

named-theorems *rewrite-eq-symm* ‹*automatically-generated*›

lemma [*rewrite-eq-symm*]:
fixes *t*::'*a*::*type* **and** *s*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*t* = *s*) = (*s* = *t*)
by *auto*

named-theorems *rewrite-eq-cond-deq* ‹*automatically-generated*›

lemma [*rewrite-eq-cond-deq*]:
fixes *t*::'*a*::*type* **and** *s*::'*a*::*type* **and** *r*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined t s r*) ⟹ ((*s* = *r*) = *False*) ⟹ ((*t* = *s*) = (*t* = *r*)) = ((¬(*t* = *s*)) ∧ (¬(*t* = *r*)))
by *auto*

named-theorems *rewrite-eq-ite-lift* ‹*automatically-generated*›

lemma [*rewrite-eq-ite-lift*]:
fixes *C*::*bool* **and** *t*::'*a*::*type* **and** *s*::'*a*::*type* **and** *r*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined C t s r*) ⟹ ((*if C then t else s*) = *r*) = (*if C then* (*t* = *r*) *else* (*s* = *r*))
by *auto*

named-theorems *rewrite-distinct-binary-elim* ‹*automatically-generated*›

lemma [*rewrite-distinct-binary-elim*]:
fixes *t*::'*a*::*type* **and** *s*::'*a*::*type*
shows *NO-MATCH cvc-a* (*undefined t s*) ⟹ (*t* ≠ *s*) = (¬ (*t* = *s*))
by *auto*

end

theory *Alethe-cvc5-Rewrites*
imports *Alethe-Rare-Nary-Ops*
begin

named-theorems *rewrite-ite-eq* ‹*cvc5 specific*›

lemma [*rewrite-ite-eq*]:
fixes *C*::*bool* **and** *t1*::*'a*::*type* **and** *t2*::*'a*::*type*
shows *NO-MATCH cvc-a* (*undefined C t1 t2*) ⟹
(*if C then* ((*if C then t1 else t2*) = *t1*) *else* ((*if C then t1 else t2*) = *t2*)) = *True*
by *simp*

named-theorems *rewrite-or-not-refl* ‹*added in postprocessing*›

lemma [*rewrite-or-not-refl*]:
fixes *t*::*'a* **and** *xs*::*bool cvc-ListVar*
shows *NO-MATCH cvc-a* (*undefined t xs*) ⟹
((*cvc-list-right* (∨) (¬(*t* = *t*)) *xs*)) = (*cvc-list-both* (∨) *False xs* (*ListVar* []))
unfolding *cvc-list-both-def cvc-list-right-def*
apply (*cases xs*)
subgoal for *xs'*
apply (*induction xs'*)
apply *simp-all*
done
done

named-theorems *rewrite-distinct-binary-elim* ‹*added in postprocessing*›

lemma [*rewrite-distinct-binary-elim*]:
fixes *t*::*bool*
shows *NO-MATCH cvc-a* (*undefined t*) ⟹
¬(*t* = *t*) = *False*
by *simp*

end

theory *Alethe-Rare-Interface*
imports *Alethe-Boolean-Rewrites Alethe-Builtin-Rewrites Alethe-Arith-Rewrites Alethe-UF-Rewrites Alethe-cvc5-Rewrites*
keywords *cvc5-rare* :: *thy-decl*
abbrevs *cvc5-rare* =
begin

93.10 Parse arguments

ML ‹
(∗*alethe proofs can contain rare-rewrites. The arguments may use rare−list to express lists.*∗)
fun alethe-term-parser (*SMTLIB.Sym rare−list*, []) = (
(∗*If there are no elements in the list we cannot know the type at this point*∗)
SOME(*Const*(***const-name*** ‹*ListVar*›, *dummyT* −−> *dummyT*) $ *Const*(***const-name*** ‹*List.Nil*›, *dummyT*)))
| *alethe-term-parser* (*SMTLIB.Sym rare−list*, *ts*) = (

```
  let
    val new-type = fastype-of (hd ts)
  in
     SOME(Const(const-name ‹ListVar›,  Type(type-name ‹List.list›,[new-type])
--> Type(type-name ‹cvc-ListVar›,[new-type]))
    $ (HOLogic.mk-list new-type ts))
  end)
| alethe-term-parser - = NONE

val - = Theory.setup (Context.theory-map (
  SMTLIB-Proof.add-term-parser alethe-term-parser)
)›
```

93.11 Replay `rare_rules`

ML-file ‹*Tools/SMT/alethe/rare-rewrites/cvc5-rare.ML*›

93.12 Register `rare_rules`

cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-elim-gt*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-elim-lt*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-elim-int-gt*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-elim-int-lt*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-elim-leq*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-leq-norm*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-geq-tighten*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-geq-norm1-int*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-eq-elim-int*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-mod-over-mod*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-divisible-elim*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-geq-ite-lift*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-leq-ite-lift*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-min-lt1*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-min-lt2*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-max-geq1*
cvc5-rare *Alethe-Arith-Rewrites.rewrite-arith-max-geq2*

cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-double-not-elim*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-true*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-false*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-eq-true*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-eq-false*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-eq-nrefl*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-impl-false1*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-impl-false2*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-impl-true1*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-impl-true2*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-impl-elim*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-dual-impl-eq*

cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-and-conf*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-and-conf2*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-or-taut*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-or-taut2*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-or-de-morgan*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-implies-de-morgan*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-and-de-morgan*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-or-and-distrib*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-implies-or-distrib*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-refl*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-nrefl*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-false*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-true*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-comm*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-xor-elim*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-xor-elim*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-eq-elim1*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-eq-elim2*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-neg-branch*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-then-true*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-else-false*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-then-false*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-else-true*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-then-lookahead-self*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-else-lookahead-self*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-then-lookahead-not-self*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-else-lookahead-not-self*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-ite-expand*
cvc5-rare *Alethe-Boolean-Rewrites.rewrite-bool-not-ite-elim*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-true-cond*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-false-cond*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-not-cond*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-eq-branch*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-then-lookahead*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-else-lookahead*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-then-neg-lookahead*
cvc5-rare *Alethe-Builtin-Rewrites.rewrite-ite-else-neg-lookahead*

cvc5-rare *Alethe-UF-Rewrites.rewrite-eq-refl*
cvc5-rare *Alethe-UF-Rewrites.rewrite-eq-symm*
cvc5-rare *Alethe-UF-Rewrites.rewrite-eq-cond-deq*
cvc5-rare *Alethe-UF-Rewrites.rewrite-eq-ite-lift*
cvc5-rare *Alethe-UF-Rewrites.rewrite-distinct-binary-elim*

cvc5-rare *Alethe-cvc5-Rewrites.rewrite-ite-eq*
cvc5-rare *Alethe-cvc5-Rewrites.rewrite-or-not-refl*

ML-file ‹*Tools/SMT/alethe/rare-rewrites/alethe-replay-rare-simplify-methods.ML*›

end
theory *SMT-CVC* — More Setup for CVC that should be in HOL eventually
  imports *HOL.SMT HOL.Alethe-Rare-Interface*
begin

ML ‹

```
fun power - - [t1] =
    let
      val mk = Term.list-comb o pair @{term pow-2}
    in SOME (int.pow2, 1, [t1], mk) end
| power - - - = NONE

val setup-builtins =
  SMT-Builtin.add-builtin-fun SMTLIB-Interface.smtlibC
    ((int.pow2, Term.dest-Const (Const ‹SMT.pow-2›) |> snd), power)

val - = Theory.setup (Context.theory-map (
  setup-builtins
))
```

›

ML ‹

```
if is-none (SMT-Builtin.dest-builtin-fun @{context}
  (int.pow2, @{typ int ⇒ int})
    [@{term 2::int}])
then error fail to recognize int.pow2 else ()
```

›

end

94 Main HOL

Classical Higher-order Logic – only "Main", excluding real and complex numbers etc.

theory *Main*
  imports
    Predicate-Compile
    Quickcheck-Narrowing
    Mirabelle
    Extraction
    Nunchaku
    BNF-Greatest-Fixpoint
    Filter

Conditionally-Complete-Lattices
Binomial
GCD
SMT-CVC
begin

94.1 Namespace cleanup

hide-const (**open**)
czero cinfinite cfinite csum cone ctwo Csum cprod cexp image2 image2p vimage2p Gr Grp collect
fsts snds setl setr convol pick-middlep fstOp sndOp csquare relImage relInvImage Succ Shift
shift proj id-bnf

hide-fact (**open**) *id-bnf-def type-definition-id-bnf-UNIV*

94.2 Syntax cleanup

no-notation
ordLeq2 (**infix** ‹<=o› *50*) **and**
ordLeq3 (**infix** ‹≤o› *50*) **and**
ordLess2 (**infix** ‹<o› *50*) **and**
ordIso2 (**infix** ‹=o› *50*) **and**
card-of (‹(‹*open-block notation*=‹*mixfix card-of*››|-|)›) **and**
BNF-Cardinal-Arithmetic.csum (**infixr** ‹+c› *65*) **and**
BNF-Cardinal-Arithmetic.cprod (**infixr** ‹*c› *80*) **and**
BNF-Cardinal-Arithmetic.cexp (**infixr** ‹^c› *90*) **and**
BNF-Def.convol (‹(‹*indent*=*1 notation*=‹*mixfix convol*››⟨-,/ -⟩)›)

bundle *cardinal-syntax*
begin

notation
ordLeq2 (**infix** ‹<=o› *50*) **and**
ordLeq3 (**infix** ‹≤o› *50*) **and**
ordLess2 (**infix** ‹<o› *50*) **and**
ordIso2 (**infix** ‹=o› *50*) **and**
card-of (‹(‹*open-block notation*=‹*mixfix card-of*››|-|)›) **and**
BNF-Cardinal-Arithmetic.csum (**infixr** ‹+c› *65*) **and**
BNF-Cardinal-Arithmetic.cprod (**infixr** ‹*c› *80*) **and**
BNF-Cardinal-Arithmetic.cexp (**infixr** ‹^c› *90*)

alias *cinfinite* = *BNF-Cardinal-Arithmetic.cinfinite*
alias *czero* = *BNF-Cardinal-Arithmetic.czero*
alias *cone* = *BNF-Cardinal-Arithmetic.cone*
alias *ctwo* = *BNF-Cardinal-Arithmetic.ctwo*

end

94.3 Lattice syntax

bundle *lattice-syntax*
begin

notation
bot (‹⊥›) **and**
top (‹⊤›) **and**
inf (**infixl** ‹⊓› *70*) **and**
sup (**infixl** ‹⊔› *65*) **and**
Inf (‹(‹*open-block notation*=‹*prefix* ⨅››⨅ -)› [*900*] *900*) **and**
Sup (‹(‹*open-block notation*=‹*prefix* ⨆››⨆ -)› [*900*] *900*)

syntax
-INF1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3 notation*=‹*binder* ⨅››⨅ -./ -)› [*0*, *10*] *10*)
-INF :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3 notation*=‹*binder* ⨅››⨅ -∈-./ -)› [*0*, *0*, *10*] *10*)
-SUP1 :: *pttrns* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3 notation*=‹*binder* ⨆››⨆ -./ -)› [*0*, *10*] *10*)
-SUP :: *pttrn* ⇒ *'a set* ⇒ *'b* ⇒ *'b* (‹(‹*indent*=*3 notation*=‹*binder* ⨆››⨆ -∈-./ -)› [*0*, *0*, *10*] *10*)

end

unbundle *no lattice-syntax*

end

95 Archimedean Fields, Floor and Ceiling Functions

theory *Archimedean-Field*
imports *Main*
begin

lemma *cInf-abs-ge*:
fixes *S* :: *'a*::{*linordered-idom*,*conditionally-complete-linorder*} *set*
assumes *S* ≠ {}
and *bdd*: ⋀*x*. *x*∈*S* ⟹ |*x*| ≤ *a*
shows |*Inf S*| ≤ *a*
proof −
have *Sup* (*uminus* ‘ *S*) = − (*Inf S*)
proof (*rule antisym*)
have ⋀*x*. *x* ∈ *S* ⟹ *bdd-above* (*uminus* ‘ *S*)
using *bdd* **by** (*force simp*: *abs-le-iff bdd-above-def*)
then show − (*Inf S*) ≤ *Sup* (*uminus* ‘ *S*)
by (*meson cInf-greatest* [*OF* ‹*S* ≠ {}›] *cSUP-upper minus-le-iff*)
next

have $*$: $\bigwedge x.\ x \in S \Longrightarrow Inf\ S \leq x$
by (*meson abs-le-iff bdd bdd-below-def cInf-lower minus-le-iff*)
show *Sup* (*uminus* ' *S*) $\leq -$ *Inf S*
using ‹$S \neq \{\}$› **by** (*force intro*: $*$ *cSup-least*)
qed
with *cSup-abs-le* [*of uminus* ' *S*] *assms* **show** *?thesis*
by *fastforce*
qed

lemma *cSup-asclose*:
fixes *S* :: *'a*::{*linordered-idom*,*conditionally-complete-linorder*} *set*
assumes *S*: $S \neq \{\}$
and *b*: $\forall x{\in}S.\ |x - l| \leq e$
shows $|Sup\ S - l| \leq e$
proof −
have $*$: $|x - l| \leq e \longleftrightarrow l - e \leq x \wedge x \leq l + e$ **for** *x l e* :: *'a*
by *arith*
have *bdd-above S*
using *b* **by** (*auto intro*!: *bdd-aboveI*[*of* - *l* + *e*])
with *S b* **show** *?thesis*
unfolding $*$ **by** (*auto intro*!: *cSup-upper2 cSup-least*)
qed

lemma *cInf-asclose*:
fixes *S* :: *'a*::{*linordered-idom*,*conditionally-complete-linorder*} *set*
assumes *S*: $S \neq \{\}$
and *b*: $\forall x{\in}S.\ |x - l| \leq e$
shows $|Inf\ S - l| \leq e$
proof −
have $*$: $|x - l| \leq e \longleftrightarrow l - e \leq x \wedge x \leq l + e$ **for** *x l e* :: *'a*
by *arith*
have *bdd-below S*
using *b* **by** (*auto intro*!: *bdd-belowI*[*of* - *l* − *e*])
with *S b* **show** *?thesis*
unfolding $*$ **by** (*auto intro*!: *cInf-lower2 cInf-greatest*)
qed

95.1 Class of Archimedean fields

Archimedean fields have no infinite elements.

class *archimedean-field* = *linordered-field* +
assumes *ex-le-of-int*: $\exists z.\ x \leq of\text{-}int\ z$

lemma *ex-less-of-int*: $\exists z.\ x < of\text{-}int\ z$
for *x* :: *'a*::*archimedean-field*
proof −
from *ex-le-of-int* **obtain** *z* **where** $x \leq of\text{-}int\ z$ **..**
then have $x < of\text{-}int\ (z + 1)$ **by** *simp*
then show *?thesis* **..**

qed

lemma *ex-of-int-less*: $\exists\, z.\ \mathit{of\text{-}int}\ z < x$
for $x :: 'a$::*archimedean-field*
proof −
from *ex-less-of-int* **obtain** z **where** $-\ x < \mathit{of\text{-}int}\ z$ **..**
then have $\mathit{of\text{-}int}\ (-\ z) < x$ **by** *simp*
then show *?thesis* **..**
qed

lemma *reals-Archimedean2*: $\exists\, n.\ x < \mathit{of\text{-}nat}\ n$
for $x :: 'a$::*archimedean-field*
proof −
obtain z **where** $x < \mathit{of\text{-}int}\ z$
using *ex-less-of-int* **..**
also have $\ldots \leq \mathit{of\text{-}int}\ (\mathit{int}\ (\mathit{nat}\ z))$
by *simp*
also have $\ldots = \mathit{of\text{-}nat}\ (\mathit{nat}\ z)$
by (*simp only*: *of-int-of-nat-eq*)
finally show *?thesis* **..**
qed

lemma *real-arch-simple*: $\exists\, n.\ x \leq \mathit{of\text{-}nat}\ n$
for $x :: 'a$::*archimedean-field*
proof −
obtain n **where** $x < \mathit{of\text{-}nat}\ n$
using *reals-Archimedean2* **..**
then have $x \leq \mathit{of\text{-}nat}\ n$
by *simp*
then show *?thesis* **..**
qed

Archimedean fields have no infinitesimal elements.

lemma *reals-Archimedean*:
fixes $x :: 'a$::*archimedean-field*
assumes $0 < x$
shows $\exists\, n.\ \mathit{inverse}\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ n)) < x$
proof −
from ‹$0 < x$› **have** $0 < \mathit{inverse}\ x$
by (*rule positive-imp-inverse-positive*)
obtain n **where** $\mathit{inverse}\ x < \mathit{of\text{-}nat}\ n$
using *reals-Archimedean2* **..**
then obtain m **where** $\mathit{inverse}\ x < \mathit{of\text{-}nat}\ (\mathit{Suc}\ m)$
using ‹$0 < \mathit{inverse}\ x$› **by** (*cases n*) (*simp-all del*: *of-nat-Suc*)
then have $\mathit{inverse}\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ m)) < \mathit{inverse}\ (\mathit{inverse}\ x)$
using ‹$0 < \mathit{inverse}\ x$› **by** (*rule less-imp-inverse-less*)
then have $\mathit{inverse}\ (\mathit{of\text{-}nat}\ (\mathit{Suc}\ m)) < x$
using ‹$0 < x$› **by** (*simp add*: *nonzero-inverse-inverse-eq*)
then show *?thesis* **..**

qed

lemma *ex-inverse-of-nat-less*:
fixes x :: *'a*::*archimedean-field*
assumes $0 < x$
shows $\exists\, n{>}0.$ *inverse* (*of-nat n*) $< x$
using *reals-Archimedean* [*OF* ‹$0 < x$›] **by** *auto*

lemma *ex-less-of-nat-mult*:
fixes x :: *'a*::*archimedean-field*
assumes $0 < x$
shows $\exists\, n.\ y <$ *of-nat n* $* x$
proof −
obtain n **where** y / $x <$ *of-nat n*
using *reals-Archimedean2* **..**
with ‹$0 < x$› **have** $y <$ *of-nat n* $* x$
by (*simp add*: *pos-divide-less-eq*)
then show *?thesis* **..**
qed

95.2 Existence and uniqueness of floor function

lemma *exists-least-lemma*:
assumes ¬ *P 0* **and** $\exists\, n.$ *P n*
shows $\exists\, n.$ ¬ *P n* ∧ *P* (*Suc n*)
proof −
from ‹$\exists\, n.$ *P n*› **have** *P* (*Least P*)
by (*rule LeastI-ex*)
with ‹¬ *P 0*› **obtain** n **where** *Least P* = *Suc n*
by (*cases Least P*) *auto*
then have $n <$ *Least P*
by *simp*
then have ¬ *P n*
by (*rule not-less-Least*)
then have ¬ *P n* ∧ *P* (*Suc n*)
using ‹*P* (*Least P*)› ‹*Least P* = *Suc n*› **by** *simp*
then show *?thesis* **..**
qed

lemma *floor-exists*:
fixes x :: *'a*::*archimedean-field*
shows $\exists\, z.$ *of-int z* $\leq x \wedge x <$ *of-int* ($z + 1$)
proof (*cases* $0 \leq x$)
case *True*
then have ¬ $x <$ *of-nat 0*
by *simp*
then have $\exists\, n.$ ¬ $x <$ *of-nat n* ∧ $x <$ *of-nat* (*Suc n*)
using *reals-Archimedean2* **by** (*rule exists-least-lemma*)
then obtain n **where** ¬ $x <$ *of-nat n* ∧ $x <$ *of-nat* (*Suc n*) **..**

```
  then have of-int (int n) ≤ x ∧ x < of-int (int n + 1)
    by simp
  then show ?thesis ..
next
  case False
  then have ¬ - x ≤ of-nat 0
    by simp
  then have ∃ n. ¬ - x ≤ of-nat n ∧ - x ≤ of-nat (Suc n)
    using real-arch-simple by (rule exists-least-lemma)
  then obtain n where ¬ - x ≤ of-nat n ∧ - x ≤ of-nat (Suc n) ..
  then have of-int (- int n - 1) ≤ x ∧ x < of-int (- int n - 1 + 1)
    by simp
  then show ?thesis ..
qed

lemma floor-exists1: ∃!z. of-int z ≤ x ∧ x < of-int (z + 1)
  for x :: 'a::archimedean-field
proof (rule ex-ex1I)
  show ∃ z. of-int z ≤ x ∧ x < of-int (z + 1)
    by (rule floor-exists)
next
  fix y z
  assume of-int y ≤ x ∧ x < of-int (y + 1)
    and of-int z ≤ x ∧ x < of-int (z + 1)
  with le-less-trans [of of-int y x of-int (z + 1)]
       le-less-trans [of of-int z x of-int (y + 1)] show y = z
    by (simp del: of-int-add)
qed
```

95.3 Floor function

```
class floor-ceiling = archimedean-field +
  fixes floor :: 'a ⇒ int  (‹(‹open-block notation=‹mixfix floor››⌊-⌋)›)
  assumes floor-correct: of-int ⌊x⌋ ≤ x ∧ x < of-int (⌊x⌋ + 1)

lemma floor-unique: of-int z ≤ x ⟹ x < of-int z + 1 ⟹ ⌊x⌋ = z
  using floor-correct [of x] floor-exists1 [of x] by auto

lemma floor-eq-iff: ⌊x⌋ = a ⟷ of-int a ≤ x ∧ x < of-int a + 1
  using floor-correct floor-unique by auto

lemma of-int-floor-le [simp]: of-int ⌊x⌋ ≤ x
  using floor-correct ..

lemma le-floor-iff: z ≤ ⌊x⌋ ⟷ of-int z ≤ x
proof
  assume z ≤ ⌊x⌋
  then have (of-int z :: 'a) ≤ of-int ⌊x⌋ by simp
  also have of-int ⌊x⌋ ≤ x by (rule of-int-floor-le)
```

finally show *of-int z ≤ x* **.**
next
assume *of-int z ≤ x*
also have *x < of-int (⌊x⌋ + 1)* **using** *floor-correct* **..**
finally show *z ≤ ⌊x⌋* **by** (*simp del*: *of-int-add*)
qed

lemma *floor-less-iff*: *⌊x⌋ < z ⟷ x < of-int z*
by (*simp add*: *not-le* [*symmetric*] *le-floor-iff*)

lemma *less-floor-iff*: *z < ⌊x⌋ ⟷ of-int z + 1 ≤ x*
using *le-floor-iff* [*of z + 1 x*] **by** *auto*

lemma *floor-le-iff*: *⌊x⌋ ≤ z ⟷ x < of-int z + 1*
by (*simp add*: *not-less* [*symmetric*] *less-floor-iff*)

lemma *floor-split*[*linarith-split*]: *P ⌊t⌋ ⟷ (∀ i. of-int i ≤ t ∧ t < of-int i + 1 ⟶ P i)*
by (*metis floor-correct floor-unique less-floor-iff not-le order-refl*)

lemma *floor-eq-imp-diff-1*: *⌊x⌋ = ⌊y⌋ ⟹ |x−y| < 1*
unfolding *floor-eq-iff* **by** *linarith*

lemma *floor-mono*:
assumes *x ≤ y*
shows *⌊x⌋ ≤ ⌊y⌋*
using *assms le-floor-iff of-int-floor-le order.trans* **by** *blast*

lemma *floor-less-cancel*: *⌊x⌋ < ⌊y⌋ ⟹ x < y*
by (*auto simp add*: *not-le* [*symmetric*] *floor-mono*)

lemma *floor-of-int* [*simp*]: *⌊of-int z⌋ = z*
by (*rule floor-unique*) *simp-all*

lemma *floor-of-nat* [*simp*]: *⌊of-nat n⌋ = int n*
using *floor-of-int* [*of of-nat n*] **by** *simp*

lemma *le-floor-add*: *⌊x⌋ + ⌊y⌋ ≤ ⌊x + y⌋*
by (*simp only*: *le-floor-iff of-int-add add-mono of-int-floor-le*)

Floor with numerals.

lemma *floor-zero* [*simp*]: *⌊0⌋ = 0*
using *floor-of-int* [*of 0*] **by** *simp*

lemma *floor-one* [*simp*]: *⌊1⌋ = 1*
using *floor-of-int* [*of 1*] **by** *simp*

lemma *floor-numeral* [*simp*]: *⌊numeral v⌋ = numeral v*
using *floor-of-int* [*of numeral v*] **by** *simp*

lemma *floor-neg-numeral* [*simp*]: $\lfloor -\ numeral\ v \rfloor = -\ numeral\ v$
using *floor-of-int* [*of* $-\ numeral\ v$] **by** *simp*

lemma *zero-le-floor* [*simp*]: $0 \leq \lfloor x \rfloor \longleftrightarrow 0 \leq x$
by (*simp add*: *le-floor-iff*)

lemma *one-le-floor* [*simp*]: $1 \leq \lfloor x \rfloor \longleftrightarrow 1 \leq x$
by (*simp add*: *le-floor-iff*)

lemma *numeral-le-floor* [*simp*]: $numeral\ v \leq \lfloor x \rfloor \longleftrightarrow numeral\ v \leq x$
by (*simp add*: *le-floor-iff*)

lemma *neg-numeral-le-floor* [*simp*]: $-\ numeral\ v \leq \lfloor x \rfloor \longleftrightarrow -\ numeral\ v \leq x$
by (*simp add*: *le-floor-iff*)

lemma *zero-less-floor* [*simp*]: $0 < \lfloor x \rfloor \longleftrightarrow 1 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *one-less-floor* [*simp*]: $1 < \lfloor x \rfloor \longleftrightarrow 2 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *numeral-less-floor* [*simp*]: $numeral\ v < \lfloor x \rfloor \longleftrightarrow numeral\ v + 1 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *neg-numeral-less-floor* [*simp*]: $-\ numeral\ v < \lfloor x \rfloor \longleftrightarrow -\ numeral\ v + 1 \leq x$
by (*simp add*: *less-floor-iff*)

lemma *floor-le-zero* [*simp*]: $\lfloor x \rfloor \leq 0 \longleftrightarrow x < 1$
by (*simp add*: *floor-le-iff*)

lemma *floor-le-one* [*simp*]: $\lfloor x \rfloor \leq 1 \longleftrightarrow x < 2$
by (*simp add*: *floor-le-iff*)

lemma *floor-le-numeral* [*simp*]: $\lfloor x \rfloor \leq numeral\ v \longleftrightarrow x < numeral\ v + 1$
by (*simp add*: *floor-le-iff*)

lemma *floor-le-neg-numeral* [*simp*]: $\lfloor x \rfloor \leq -\ numeral\ v \longleftrightarrow x < -\ numeral\ v + 1$
by (*simp add*: *floor-le-iff*)

lemma *floor-less-zero* [*simp*]: $\lfloor x \rfloor < 0 \longleftrightarrow x < 0$
by (*simp add*: *floor-less-iff*)

lemma *floor-less-one* [*simp*]: $\lfloor x \rfloor < 1 \longleftrightarrow x < 1$
by (*simp add*: *floor-less-iff*)

lemma *floor-less-numeral* [*simp*]: $\lfloor x \rfloor < numeral\ v \longleftrightarrow x < numeral\ v$

by (*simp add*: *floor-less-iff*)

lemma *floor-less-neg-numeral* [*simp*]: $\lfloor x \rfloor < - \ numeral\ v \longleftrightarrow x < - \ numeral\ v$
by (*simp add*: *floor-less-iff*)

lemma *le-mult-floor-Ints*:
assumes $0 \leq a$ $a \in Ints$
shows *of-int* $(\lfloor a \rfloor * \lfloor b \rfloor) \leq (of\text{-}int \lfloor a * b \rfloor :: 'a :: linordered\text{-}idom)$
by (*metis Ints-cases assms floor-less-iff floor-of-int linorder-not-less mult-left-mono of-int-floor-le of-int-less-iff of-int-mult*)

Addition and subtraction of integers.

lemma *floor-add-int*: $\lfloor x \rfloor + z = \lfloor x + of\text{-}int\ z \rfloor$
using *floor-correct* [*of x*] **by** (*simp add*: *floor-unique*[*symmetric*])

lemma *int-add-floor*: $z + \lfloor x \rfloor = \lfloor of\text{-}int\ z + x \rfloor$
using *floor-correct* [*of x*] **by** (*simp add*: *floor-unique*[*symmetric*])

lemma *one-add-floor*: $\lfloor x \rfloor + 1 = \lfloor x + 1 \rfloor$
using *floor-add-int* [*of x 1*] **by** *simp*

lemma *floor-diff-of-int* [*simp*]: $\lfloor x - of\text{-}int\ z \rfloor = \lfloor x \rfloor - z$
using *floor-add-int* [*of x − z*] **by** (*simp add*: *algebra-simps*)

lemma *floor-uminus-of-int* [*simp*]: $\lfloor - (of\text{-}int\ z) \rfloor = - z$
by (*metis floor-diff-of-int* [*of 0*] *diff-0 floor-zero*)

lemma *floor-diff-numeral* [*simp*]: $\lfloor x - numeral\ v \rfloor = \lfloor x \rfloor - numeral\ v$
using *floor-diff-of-int* [*of x numeral v*] **by** *simp*

lemma *floor-diff-one* [*simp*]: $\lfloor x - 1 \rfloor = \lfloor x \rfloor - 1$
using *floor-diff-of-int* [*of x 1*] **by** *simp*

lemma *le-mult-floor*:
assumes $0 \leq a$ **and** $0 \leq b$
shows $\lfloor a \rfloor * \lfloor b \rfloor \leq \lfloor a * b \rfloor$
proof −
have *of-int* $\lfloor a \rfloor \leq a$ **and** *of-int* $\lfloor b \rfloor \leq b$
by (*auto intro*: *of-int-floor-le*)
then have *of-int* $(\lfloor a \rfloor * \lfloor b \rfloor) \leq a * b$
using *assms* **by** (*auto intro*!: *mult-mono*)
also have $a * b <$ *of-int* $(\lfloor a * b \rfloor + 1)$
using *floor-correct*[*of a * b*] **by** *auto*
finally show *?thesis*
unfolding *of-int-less-iff* **by** *simp*
qed

lemma *floor-divide-of-int-eq*: $\lfloor of\text{-}int\ k\ /\ of\text{-}int\ l \rfloor = k\ div\ l$
for $k\ l :: int$

proof (*cases* $l = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have $*$: $\lfloor$*of-int* $(k$ *mod* $l)$ / *of-int* $l :: 'a\rfloor = 0$
proof (*cases* $l > 0$)
case *True*
then show *?thesis*
by (*auto intro*: *floor-unique*)
next
case *False*
obtain r **where** $r = -\ l$
by *blast*
then have *l*: $l = -\ r$
by *simp*
with ‹$l \neq 0$› *False* **have** $r > 0$
by *simp*
with l **show** *?thesis*
using *pos-mod-bound* [*of r*]
by (*auto simp add*: *zmod-zminus2-eq-if less-le field-simps intro*: *floor-unique*)
qed
have (*of-int* $k :: 'a$) = *of-int* $(k$ *div* $l * l + k$ *mod* $l)$
by *simp*
also have ... = (*of-int* $(k$ *div* $l)$ + *of-int* $(k$ *mod* $l)$ / *of-int* $l) *$ *of-int* l
using *False* **by** (*simp only*: *of-int-add*) (*simp add*: *field-simps*)
finally have (*of-int* k / *of-int* $l :: 'a$) = ... / *of-int* l
by *simp*
then have (*of-int* k / *of-int* $l :: 'a$) = *of-int* $(k$ *div* $l)$ + *of-int* $(k$ *mod* $l)$ / *of-int* l
using *False* **by** (*simp only*:) (*simp add*: *field-simps*)
then have $\lfloor$*of-int* k / *of-int* $l :: 'a\rfloor$ = $\lfloor$*of-int* $(k$ *div* $l)$ + *of-int* $(k$ *mod* $l)$ / *of-int* $l :: 'a\rfloor$
by *simp*
then have $\lfloor$*of-int* k / *of-int* $l :: 'a\rfloor$ = $\lfloor$*of-int* $(k$ *mod* $l)$ / *of-int* l + *of-int* $(k$ *div* $l) :: 'a\rfloor$
by (*simp add*: *ac-simps*)
then have $\lfloor$*of-int* k / *of-int* $l :: 'a\rfloor$ = $\lfloor$*of-int* $(k$ *mod* $l)$ / *of-int* $l :: 'a\rfloor + k$ *div* l
by (*simp add*: *floor-add-int*)
with $*$ **show** *?thesis*
by *simp*
qed

lemma *floor-divide-of-nat-eq*: $\lfloor$*of-nat* m / *of-nat* $n\rfloor$ = *of-nat* $(m$ *div* $n)$
for m $n ::$ *nat*
by (*metis floor-divide-of-int-eq of-int-of-nat-eq linordered-euclidean-semiring-class.of-nat-div*)

lemma *floor-divide-lower*:
fixes $q :: 'a$::*floor-ceiling*
shows $q > 0 \Longrightarrow$ *of-int* $\lfloor p$ / $q\rfloor * q \leq p$

using *of-int-floor-le pos-le-divide-eq* **by** *blast*

lemma *floor-divide-upper*:
fixes q :: *'a*::*floor-ceiling*
shows $q > 0 \Longrightarrow p < (\textit{of-int}\ \lfloor p\ /\ q \rfloor + 1) * q$
by (*meson floor-eq-iff pos-divide-less-eq*)

95.4 Ceiling function

definition *ceiling* :: *'a*::*floor-ceiling* ⇒ *int* (‹(‹*open-block notation*=‹*mixfix ceiling*››⌈-⌉)›)
where $\lceil x \rceil = - \lfloor - x \rfloor$

lemma *ceiling-correct*: *of-int* $\lceil x \rceil - 1 < x \wedge x \leq$ *of-int* $\lceil x \rceil$
unfolding *ceiling-def* **using** *floor-correct* [*of* $- x$]
by (*simp add*: *le-minus-iff*)

lemma *ceiling-unique*: *of-int* $z - 1 < x \Longrightarrow x \leq$ *of-int* $z \Longrightarrow \lceil x \rceil = z$
unfolding *ceiling-def* **using** *floor-unique* [*of* $- z - x$] **by** *simp*

lemma *ceiling-eq-iff*: $\lceil x \rceil = a \longleftrightarrow$ *of-int* $a - 1 < x \wedge x \leq$ *of-int* a
using *ceiling-correct ceiling-unique* **by** *auto*

lemma *le-of-int-ceiling* [*simp*]: $x \leq$ *of-int* $\lceil x \rceil$
using *ceiling-correct* **..**

lemma *ceiling-le-iff*: $\lceil x \rceil \leq z \longleftrightarrow x \leq$ *of-int* z
unfolding *ceiling-def* **using** *le-floor-iff* [*of* $- z - x$] **by** *auto*

lemma *less-ceiling-iff*: $z <$ *of-int* $\lceil x \rceil \longleftrightarrow$ *of-int* $z < x$
by (*simp add*: *not-le* [*symmetric*] *ceiling-le-iff*)

lemma *ceiling-less-iff*: $\lceil x \rceil < z \longleftrightarrow x \leq$ *of-int* $z - 1$
using *ceiling-le-iff* [*of x z* $- 1$] **by** *simp*

lemma *le-ceiling-iff*: $z \leq \lceil x \rceil \longleftrightarrow$ *of-int* $z - 1 < x$
by (*simp add*: *not-less* [*symmetric*] *ceiling-less-iff*)

lemma *ceiling-mono*: $x \geq y \Longrightarrow \lceil x \rceil \geq \lceil y \rceil$
unfolding *ceiling-def* **by** (*simp add*: *floor-mono*)

lemma *ceiling-less-cancel*: $\lceil x \rceil < \lceil y \rceil \Longrightarrow x < y$
by (*auto simp add*: *not-le* [*symmetric*] *ceiling-mono*)

lemma *ceiling-of-int* [*simp*]: $\lceil$*of-int* $z \rceil = z$
by (*rule ceiling-unique*) *simp-all*

lemma *ceiling-of-nat* [*simp*]: $\lceil$*of-nat* $n \rceil =$ *int* n
using *ceiling-of-int* [*of of-nat n*] **by** *simp*

lemma *ceiling-add-le*: $\lceil x + y \rceil \leq \lceil x \rceil + \lceil y \rceil$
by (*simp only*: *ceiling-le-iff of-int-add add-mono le-of-int-ceiling*)

lemma *mult-ceiling-le*:
assumes $0 \leq a$ **and** $0 \leq b$
shows $\lceil a * b \rceil \leq \lceil a \rceil * \lceil b \rceil$
by (*metis assms ceiling-le-iff ceiling-mono le-of-int-ceiling mult-mono of-int-mult*)

lemma *mult-ceiling-le-Ints*:
assumes $0 \leq a$ $a \in$ *Ints*
shows (*of-int* $\lceil a * b \rceil :: 'a ::$ *linordered-idom*) $\leq$ *of-int*$(\lceil a \rceil * \lceil b \rceil)$
by (*metis Ints-cases assms ceiling-le-iff ceiling-of-int le-of-int-ceiling mult-left-mono of-int-le-iff of-int-mult*)

lemma *finite-int-segment*:
fixes $a :: 'a$::*floor-ceiling*
shows *finite* $\{x \in \mathbb{Z}.\ a \leq x \wedge x \leq b\}$
proof −
have *finite* {*ceiling a*..*floor b*}
by *simp*
moreover have $\{x \in \mathbb{Z}.\ a \leq x \wedge x \leq b\} =$ *of-int* ' {*ceiling a*..*floor b*}
by (*auto simp*: *le-floor-iff ceiling-le-iff elim*!: *Ints-cases*)
ultimately show *?thesis*
by *simp*
qed

corollary *finite-abs-int-segment*:
fixes $a :: 'a$::*floor-ceiling*
shows *finite* $\{k \in \mathbb{Z}.\ |k| \leq a\}$
using *finite-int-segment* [*of* $-a$ a] **by** (*auto simp add*: *abs-le-iff conj-commute minus-le-iff*)

95.4.1 Ceiling with numerals.

lemma *ceiling-zero* [*simp*]: $\lceil 0 \rceil = 0$
using *ceiling-of-int* [*of 0*] **by** *simp*

lemma *ceiling-one* [*simp*]: $\lceil 1 \rceil = 1$
using *ceiling-of-int* [*of 1*] **by** *simp*

lemma *ceiling-numeral* [*simp*]: $\lceil$*numeral v*$\rceil =$ *numeral v*
using *ceiling-of-int* [*of numeral v*] **by** *simp*

lemma *ceiling-neg-numeral* [*simp*]: $\lceil -$ *numeral v*$\rceil = -$ *numeral v*
using *ceiling-of-int* [*of* $-$ *numeral v*] **by** *simp*

lemma *ceiling-le-zero* [*simp*]: $\lceil x \rceil \leq 0 \longleftrightarrow x \leq 0$
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-one* [*simp*]: $\lceil x \rceil \leq 1 \longleftrightarrow x \leq 1$
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-numeral* [*simp*]: $\lceil x \rceil \leq numeral\ v \longleftrightarrow x \leq numeral\ v$
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-le-neg-numeral* [*simp*]: $\lceil x \rceil \leq - numeral\ v \longleftrightarrow x \leq - numeral\ v$
by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-less-zero* [*simp*]: $\lceil x \rceil < 0 \longleftrightarrow x \leq -1$
by (*simp add*: *ceiling-less-iff*)

lemma *ceiling-less-one* [*simp*]: $\lceil x \rceil < 1 \longleftrightarrow x \leq 0$
by (*simp add*: *ceiling-less-iff*)

lemma *ceiling-less-numeral* [*simp*]: $\lceil x \rceil < numeral\ v \longleftrightarrow x \leq numeral\ v - 1$
by (*simp add*: *ceiling-less-iff*)

lemma *ceiling-less-neg-numeral* [*simp*]: $\lceil x \rceil < - numeral\ v \longleftrightarrow x \leq - numeral\ v - 1$
by (*simp add*: *ceiling-less-iff*)

lemma *zero-le-ceiling* [*simp*]: $0 \leq \lceil x \rceil \longleftrightarrow -1 < x$
by (*simp add*: *le-ceiling-iff*)

lemma *one-le-ceiling* [*simp*]: $1 \leq \lceil x \rceil \longleftrightarrow 0 < x$
by (*simp add*: *le-ceiling-iff*)

lemma *numeral-le-ceiling* [*simp*]: $numeral\ v \leq \lceil x \rceil \longleftrightarrow numeral\ v - 1 < x$
by (*simp add*: *le-ceiling-iff*)

lemma *neg-numeral-le-ceiling* [*simp*]: $- numeral\ v \leq \lceil x \rceil \longleftrightarrow - numeral\ v - 1 < x$
by (*simp add*: *le-ceiling-iff*)

lemma *zero-less-ceiling* [*simp*]: $0 < \lceil x \rceil \longleftrightarrow 0 < x$
by (*simp add*: *less-ceiling-iff*)

lemma *one-less-ceiling* [*simp*]: $1 < \lceil x \rceil \longleftrightarrow 1 < x$
by (*simp add*: *less-ceiling-iff*)

lemma *numeral-less-ceiling* [*simp*]: $numeral\ v < \lceil x \rceil \longleftrightarrow numeral\ v < x$
by (*simp add*: *less-ceiling-iff*)

lemma *neg-numeral-less-ceiling* [*simp*]: $- numeral\ v < \lceil x \rceil \longleftrightarrow - numeral\ v < x$
by (*simp add*: *less-ceiling-iff*)

lemma *ceiling-altdef*: $\lceil x \rceil = (if\ x = of\text{-}int\ \lfloor x \rfloor\ then\ \lfloor x \rfloor\ else\ \lfloor x \rfloor + 1)$

by (*intro ceiling-unique*; *simp*, *linarith?*)

lemma *floor-le-ceiling* [*simp*]: $\lfloor x \rfloor \leq \lceil x \rceil$
by (*simp add*: *ceiling-altdef*)

95.4.2 Addition and subtraction of integers.

lemma *ceiling-add-of-int* [*simp*]: $\lceil x + \textit{of-int } z \rceil = \lceil x \rceil + z$
using *ceiling-correct* [*of x*] **by** (*simp add*: *ceiling-def*)

lemma *ceiling-add-numeral* [*simp*]: $\lceil x + \textit{numeral } v \rceil = \lceil x \rceil + \textit{numeral } v$
using *ceiling-add-of-int* [*of x numeral v*] **by** *simp*

lemma *ceiling-add-one* [*simp*]: $\lceil x + 1 \rceil = \lceil x \rceil + 1$
using *ceiling-add-of-int* [*of x 1*] **by** *simp*

lemma *ceiling-diff-of-int* [*simp*]: $\lceil x - \textit{of-int } z \rceil = \lceil x \rceil - z$
using *ceiling-add-of-int* [*of x* $-$ *z*] **by** (*simp add*: *algebra-simps*)

lemma *ceiling-diff-numeral* [*simp*]: $\lceil x - \textit{numeral } v \rceil = \lceil x \rceil - \textit{numeral } v$
using *ceiling-diff-of-int* [*of x numeral v*] **by** *simp*

lemma *ceiling-diff-one* [*simp*]: $\lceil x - 1 \rceil = \lceil x \rceil - 1$
using *ceiling-diff-of-int* [*of x 1*] **by** *simp*

lemma *ceiling-split*[*linarith-split*]: $P\ \lceil t \rceil \longleftrightarrow (\forall i.\ \textit{of-int } i - 1 < t \wedge t \leq \textit{of-int } i \longrightarrow P\ i)$
by (*auto simp add*: *ceiling-unique ceiling-correct*)

lemma *ceiling-eq-imp-diff-1*: $\lceil x \rceil = \lceil y \rceil \Longrightarrow |x-y| < 1$
unfolding *ceiling-eq-iff* **by** *linarith*

lemma *ceiling-diff-floor-le-1*: $\lceil x \rceil - \lfloor x \rfloor \leq 1$
by (*simp add*: *ceiling-altdef*)

lemma *floor-eq-ceiling-imp-diff-2*: $\lfloor x \rfloor = \lceil y \rceil \Longrightarrow |x-y| < 2$
unfolding *floor-eq-iff* **by** *linarith*

lemma *nat-approx-posE*:
fixes *e*:: *'a*::{*archimedean-field*,*floor-ceiling*}
assumes $0 < e$
obtains *n* :: *nat* **where** $1 \ / \ \textit{of-nat}(\textit{Suc } n) < e$
proof
have $(1::'a) \ / \ \textit{of-nat } (\textit{Suc } (\textit{nat } \lceil 1/e \rceil)) < 1 \ / \ \textit{of-int } (\lceil 1/e \rceil)$
proof (*rule divide-strict-left-mono*)
show $(\textit{of-int } \lceil 1 \ / \ e \rceil::'a) < \textit{of-nat } (\textit{Suc } (\textit{nat } \lceil 1 \ / \ e \rceil))$
using *assms* **by** (*simp add*: *field-simps*)
show $(0::'a) < \textit{of-nat } (\textit{Suc } (\textit{nat } \lceil 1 \ / \ e \rceil)) * \textit{of-int } \lceil 1 \ / \ e \rceil$
using *assms* **by** (*auto simp*: *zero-less-mult-iff pos-add-strict*)

qed *auto*
also have *1 / of-int* (⌈*1/e*⌉) ≤ *1 / (1/e)*
by (*rule divide-left-mono*) (*auto simp*: ‹*0 < e*› *ceiling-correct*)
also have ... = *e* **by** *simp*
finally show *1 / of-nat* (*Suc* (*nat* ⌈*1 / e*⌉)) < *e*
by *metis*
qed

lemma *ceiling-divide-upper*:
fixes *q* :: *'a*::*floor-ceiling*
shows *q > 0* ⟹ *p* ≤ *of-int* (*ceiling* (*p / q*)) * *q*
by (*meson divide-le-eq le-of-int-ceiling*)

lemma *ceiling-divide-lower*:
fixes *q* :: *'a*::*floor-ceiling*
shows *q > 0* ⟹ (*of-int* ⌈*p / q*⌉ − *1*) * *q* < *p*
by (*meson ceiling-eq-iff pos-less-divide-eq*)

95.5 Negation

lemma *floor-minus*: ⌊− *x*⌋ = − ⌈*x*⌉
unfolding *ceiling-def* **by** *simp*

lemma *ceiling-minus*: ⌈− *x*⌉ = − ⌊*x*⌋
unfolding *ceiling-def* **by** *simp*

95.6 Natural numbers

lemma *of-nat-floor*: *r*≥*0* ⟹ *of-nat* (*nat* ⌊*r*⌋) ≤ *r*
by *simp*

lemma *of-nat-ceiling*: *of-nat* (*nat* ⌈*r*⌉) ≥ *r*
by (*cases r*≥*0*) *auto*

lemma *of-nat-int-floor* [*simp*]: *x*≥*0* ⟹ *of-nat* (*nat*⌊*x*⌋) = *of-int* ⌊*x*⌋
by *auto*

lemma *of-nat-int-ceiling* [*simp*]: *x*≥*0* ⟹ *of-nat* (*nat* ⌈*x*⌉) = *of-int* ⌈*x*⌉
by *auto*

95.7 Frac Function

definition *frac* :: *'a* ⇒ *'a*::*floor-ceiling*
where *frac x* ≡ *x* − *of-int* ⌊*x*⌋

lemma *frac-lt-1*: *frac x < 1*
by (*simp add*: *frac-def*) *linarith*

lemma *frac-eq-0-iff* [*simp*]: *frac x = 0* ⟷ *x* ∈ ℤ
by (*simp add*: *frac-def*) (*metis Ints-cases Ints-of-int floor-of-int*)

lemma *frac-ge-0* [*simp*]: *frac x* ≥ *0*
unfolding *frac-def* **by** *linarith*

lemma *frac-gt-0-iff* [*simp*]: *frac x* > *0* ⟷ *x* ∉ ℤ
by (*metis frac-eq-0-iff frac-ge-0 le-less less-irrefl*)

lemma *frac-of-int* [*simp*]: *frac* (*of-int z*) = *0*
by (*simp add*: *frac-def*)

lemma *frac-frac* [*simp*]: *frac* (*frac x*) = *frac x*
by (*simp add*: *frac-def*)

lemma *floor-add*: ⌊*x* + *y*⌋ = (*if frac x* + *frac y* < *1 then* ⌊*x*⌋ + ⌊*y*⌋ *else* (⌊*x*⌋ + ⌊*y*⌋) + *1*)
proof −
have *x* + *y* < *1* + (*of-int* ⌊*x*⌋ + *of-int* ⌊*y*⌋) ⟹ ⌊*x* + *y*⌋ = ⌊*x*⌋ + ⌊*y*⌋
by (*metis add.commute floor-unique le-floor-add le-floor-iff of-int-add*)
moreover
have ¬ *x* + *y* < *1* + (*of-int* ⌊*x*⌋ + *of-int* ⌊*y*⌋) ⟹ ⌊*x* + *y*⌋ = *1* + (⌊*x*⌋ + ⌊*y*⌋)
apply (*simp add*: *floor-eq-iff*)
apply (*auto simp add*: *algebra-simps*)
apply *linarith*
done
ultimately show *?thesis* **by** (*auto simp add*: *frac-def algebra-simps*)
qed

lemma *floor-add2*[*simp*]: *x* ∈ ℤ ∨ *y* ∈ ℤ ⟹ ⌊*x* + *y*⌋ = ⌊*x*⌋ + ⌊*y*⌋
by (*metis add.commute add.left-neutral frac-lt-1 floor-add frac-eq-0-iff*)

lemma *frac-add*:
frac (*x* + *y*) = (*if frac x* + *frac y* < *1 then frac x* + *frac y else* (*frac x* + *frac y*) − *1*)
by (*simp add*: *frac-def floor-add*)

lemma *frac-unique-iff*: *frac x* = *a* ⟷ *x* − *a* ∈ ℤ ∧ *0* ≤ *a* ∧ *a* < *1*
for *x* :: *'a*::*floor-ceiling*
by (*auto simp*: *Ints-def frac-def algebra-simps floor-unique*; *linarith*)

lemma *frac-eq*: *frac x* = *x* ⟷ *0* ≤ *x* ∧ *x* < *1*
by (*simp add*: *frac-unique-iff*)

lemma *frac-eq-id* [*simp*]: *x* ∈ {*0*..<*1*} ⟹ *frac x* = *x*
by (*simp add*: *frac-eq*)

lemma *frac-in-Ints-iff* [*simp*]: *frac x* ∈ ℤ ⟷ *x* ∈ ℤ
by (*metis frac-eq-0-iff frac-frac*)

lemma *frac-neg*: *frac* (− *x*) = (*if x* ∈ ℤ *then 0 else 1* − *frac x*)

for $x :: 'a$::*floor-ceiling*
unfolding *frac-unique-iff* **using** *frac-lt-1* [*of* x]
apply (*simp add*: *frac-def*)
by (*metis Ints-of-int floor-eq-iff nless-le*)

lemma *frac-1-eq*: *frac* $(x+1) =$ *frac* x
by (*simp add*: *frac-def*)

95.8 Fractional part arithmetic

Many thanks to Stepan Holub

lemma *frac-non-zero*: *frac* $x \neq 0 \Longrightarrow$ *frac* $(-x) =$ $1 -$ *frac* x
using *frac-eq-0-iff frac-neg* **by** *metis*

lemma *frac-add-simps* [*simp*]:
frac (*frac* $a +$ b) = *frac* $(a + b)$
frac $(a +$ *frac* $b) =$ *frac* $(a + b)$
by (*simp-all add*: *frac-add*)

lemma *frac-neg-frac*: *frac* $(-$ *frac* $x) =$ *frac* $(-x)$
unfolding *frac-neg frac-frac* **by** *force*

lemma *frac-diff-simp*: *frac* $(y -$ *frac* $x) =$ *frac* $(y - x)$
unfolding *diff-conv-add-uminus frac-add frac-neg-frac*..

lemma *frac-diff*: *frac* $(a - b) =$ *frac* (*frac* $a + (-$ *frac* $b))$
unfolding *frac-add-simps*(*1*)
unfolding *ab-group-add-class.ab-diff-conv-add-uminus*[*symmetric*] *frac-diff-simp*..

lemma *frac-diff-pos*: *frac* $x \leq$ *frac* $y \Longrightarrow$ *frac* $(y - x) =$ *frac* $y -$ *frac* x
unfolding *diff-conv-add-uminus frac-add frac-neg*
using *frac-lt-1* **by** *force*

lemma *frac-diff-neg*: **assumes** *frac* $y <$ *frac* x
shows *frac* $(y - x) =$ *frac* $y + 1 -$ *frac* x
proof−
have $x \notin \mathbb{Z}$
unfolding *frac-gt-0-iff*[*symmetric*]
using *assms frac-ge-0*[*of y*] **by** *order*
have *frac* $y + (1 + -$ *frac* $x) < 1$
using *frac-lt-1*[*of x*] *assms* **by** *fastforce*
show *?thesis*
unfolding *diff-conv-add-uminus frac-add frac-neg*
if-not-P[*OF* ‹$x \notin \mathbb{Z}$›] *if-P*[*OF* ‹*frac* $y + (1 + -$ *frac* $x) < 1$›]
by *simp*
qed

lemma *frac-diff-eq*: **assumes** *frac* $y =$ *frac* x
shows *frac* $(y - x) = 0$

by (*simp add*: *assms frac-diff-pos*)

lemma *frac-diff-zero*: **assumes** *frac* $(x - y) = 0$
shows *frac* $x =$ *frac* y
using *frac-add-simps*(*1*)[*of* $x - y$ y, *symmetric*]
unfolding *assms add.group-left-neutral diff-add-cancel*.

lemma *frac-neg-eq-iff*: *frac* $(-x) =$ *frac* $(-y) \longleftrightarrow$ *frac* $x =$ *frac* y
using *add.inverse-inverse frac-neg-frac* **by** *metis*

lemma *frac-eqE*:
assumes *frac* $x =$ *frac* y
obtains n **where** $x = y +$ *of-int* n
by (*rule that*[*of floor* $x -$ *floor* y]) (*use assms* **in** ‹*auto simp*: *frac-def*›)

lemma *frac-add-of-int-right* [*simp*]: *frac* $(x +$ *of-int* $n) =$ *frac* x
by (*auto simp*: *frac-def*)

lemma *frac-add-of-int-left* [*simp*]: *frac* (*of-int* $n + x) =$ *frac* x
by (*auto simp*: *frac-def*)

lemma *frac-add-int-right*: $y \in \mathbb{Z} \Longrightarrow$ *frac* $(x + y) =$ *frac* x
by (*elim Ints-cases*) *auto*

lemma *frac-add-int-left*: $x \in \mathbb{Z} \Longrightarrow$ *frac* $(x + y) =$ *frac* y
by (*elim Ints-cases*) *auto*

lemma *bij-betw-frac*: *bij-betw frac* $\{x..<x+1\}$ $\{0..<1\}$
unfolding *bij-betw-def*
proof
show *inj-on frac* $\{x..<x+1\}$
proof
fix a b **assume** $*$: $a \in \{x..<x+1\}$ $b \in \{x..<x+1\}$ *frac* $a =$ *frac* b
then obtain n **where** n: $a = b +$ *of-int* n
by (*elim frac-eqE*)
have $|$*of-int* $n| = |a - b|$
using n **by** (*simp add*: *algebra-simps*)
also have $\ldots < 1$
using $*$ **by** *auto*
finally have $n = 0$
by (*simp flip*: *of-int-abs*)
with n **show** $a = b$
by *auto*
qed
next
show *frac* ‘ $\{x..<x + 1\} = \{0..<1\}$
proof (*intro equalityI subsetI*)
fix $t :: 'a$ **assume** t: $t \in \{0..<1\}$
have $t =$ *frac* (*if* $t \geq$ *frac* x *then* $x + t -$ *frac* x *else* $x + t -$ *frac* $x + 1$)

using *frac-eq*[*of t*] *t* **by** (*auto simp*: *frac-def*)
moreover have (*if t* ≥ *frac x then x* + *t* − *frac x else x* + *t* − *frac x* + *1*) ∈ {*x*..<*x*+*1*}
using *frac-lt-1*[*of x*] *frac-ge-0*[*of x*] *t* **by** (*auto simp del*: *frac-ge-0*)
ultimately show *t* ∈ *frac* ' {*x*..<*x* + *1*}
by *blast*
qed (*auto intro*: *frac-lt-1*)
qed

95.9 Rounding to the nearest integer

definition *round* :: *'a*::*floor-ceiling* ⇒ *int*
where *round x* = ⌊*x* + *1/2*⌋

lemma *round-eq-imp-diff-1*: *round x* = *round y* ⟹ |*x*−*y*| < *1*
unfolding *round-def*
using *floor-eq-imp-diff-1* **by** *fastforce*

lemma *of-int-round-ge*: *of-int* (*round x*) ≥ *x* − *1/2*
and *of-int-round-le*: *of-int* (*round x*) ≤ *x* + *1/2*
and *of-int-round-abs-le*: |*of-int* (*round x*) − *x*| ≤ *1/2*
and *of-int-round-gt*: *of-int* (*round x*) > *x* − *1/2*
proof −
from *floor-correct*[*of x* + *1/2*] **have** *x* + *1/2* < *of-int* (*round x*) + *1*
by (*simp add*: *round-def*)
from *add-strict-right-mono*[*OF this*, *of* −*1*] **show** *A*: *of-int* (*round x*) > *x* − *1/2*
by *simp*
then show *of-int* (*round x*) ≥ *x* − *1/2*
by *simp*
from *floor-correct*[*of x* + *1/2*] **show** *of-int* (*round x*) ≤ *x* + *1/2*
by (*simp add*: *round-def*)
with *A* **show** |*of-int* (*round x*) − *x*| ≤ *1/2*
by *linarith*
qed

lemma *round-of-int* [*simp*]: *round* (*of-int n*) = *n*
unfolding *round-def* **by** (*subst floor-eq-iff*) *force*

lemma *round-0* [*simp*]: *round 0* = *0*
using *round-of-int*[*of 0*] **by** *simp*

lemma *round-1* [*simp*]: *round 1* = *1*
using *round-of-int*[*of 1*] **by** *simp*

lemma *round-numeral* [*simp*]: *round* (*numeral n*) = *numeral n*
using *round-of-int*[*of numeral n*] **by** *simp*

lemma *round-neg-numeral* [*simp*]: *round* (−*numeral n*) = −*numeral n*
using *round-of-int*[*of* −*numeral n*] **by** *simp*

lemma *round-of-nat* [*simp*]: *round* (*of-nat n*) = *of-nat n*
using *round-of-int*[*of int n*] **by** *simp*

lemma *round-mono*: $x \leq y \Longrightarrow$ *round x* $\leq$ *round y*
unfolding *round-def* **by** (*intro floor-mono*) *simp*

lemma *round-unique*: *of-int y* $>$ *x* $-$ *1/2* $\Longrightarrow$ *of-int y* $\leq$ *x* $+$ *1/2* $\Longrightarrow$ *round x* = *y*
unfolding *round-def*
proof (*rule floor-unique*)
assume *x* $-$ *1* / *2* $<$ *of-int y*
from *add-strict-left-mono*[*OF this*, *of 1*] **show** *x* $+$ *1* / *2* $<$ *of-int y* $+$ *1*
by *simp*
qed

lemma *round-unique'*: $|$*x* $-$ *of-int n*$|$ $<$ *1/2* $\Longrightarrow$ *round x* = *n*
by (*subst* (*asm*) *abs-less-iff*, *rule round-unique*) (*simp-all add*: *field-simps*)

lemma *round-altdef*: *round x* = (*if frac x* $\geq$ *1/2 then* $\lceil$*x*$\rceil$ *else* $\lfloor$*x*$\rfloor$)
by (*cases frac x* $\geq$ *1/2*)
(*rule round-unique*, ((*simp add*: *frac-def field-simps ceiling-altdef*; *linarith*)+)[*2*])+

lemma *floor-le-round*: $\lfloor$*x*$\rfloor$ $\leq$ *round x*
unfolding *round-def* **by** (*intro floor-mono*) *simp*

lemma *ceiling-ge-round*: $\lceil$*x*$\rceil$ $\geq$ *round x*
unfolding *round-altdef* **by** *simp*

lemma *round-diff-minimal*: $|$*z* $-$ *of-int* (*round z*)$|$ $\leq$ $|$*z* $-$ *of-int m*$|$
for *z* :: *'a*::*floor-ceiling*
proof (*cases of-int m* $\geq$ *z*)
case *True*
then have $|$*z* $-$ *of-int* (*round z*)$|$ $\leq$ $|$*of-int* $\lceil$*z*$\rceil$ $-$ *z*$|$
unfolding *round-altdef* **by** (*simp add*: *field-simps ceiling-altdef frac-def*) *linarith*
also have *of-int* $\lceil$*z*$\rceil$ $-$ *z* $\geq$ *0*
by *linarith*
with *True* **have** $|$*of-int* $\lceil$*z*$\rceil$ $-$ *z*$|$ $\leq$ $|$*z* $-$ *of-int m*$|$
by (*simp add*: *ceiling-le-iff*)
finally show *?thesis* **.**
next
case *False*
then have $|$*z* $-$ *of-int* (*round z*)$|$ $\leq$ $|$*of-int* $\lfloor$*z*$\rfloor$ $-$ *z*$|$
unfolding *round-altdef* **by** (*simp add*: *field-simps ceiling-altdef frac-def*) *linarith*
also have *z* $-$ *of-int* $\lfloor$*z*$\rfloor$ $\geq$ *0*
by *linarith*
with *False* **have** $|$*of-int* $\lfloor$*z*$\rfloor$ $-$ *z*$|$ $\leq$ $|$*z* $-$ *of-int m*$|$
by (*simp add*: *le-floor-iff*)
finally show *?thesis* **.**

qed

bundle *floor-ceiling-syntax*
begin
notation *floor* (‹(‹*open-block notation*=‹*mixfix floor*››⌊-⌋)›)
and *ceiling* (‹(‹*open-block notation*=‹*mixfix ceiling*››⌈-⌉)›)
end

end

96 Rational numbers

theory *Rat*
imports *Archimedean-Field*
begin

96.1 Rational numbers as quotient

96.1.1 Construction of the type of rational numbers

definition *ratrel* :: (*int* × *int*) ⇒ (*int* × *int*) ⇒ *bool*
where *ratrel* = (λ*x y*. *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*)

lemma *ratrel-iff* [*simp*]: *ratrel x y* ⟷ *snd x* ≠ *0* ∧ *snd y* ≠ *0* ∧ *fst x* ∗ *snd y* = *fst y* ∗ *snd x*
by (*simp add*: *ratrel-def*)

lemma *exists-ratrel-refl*: ∃ *x*. *ratrel x x*
by (*auto intro*!: *one-neq-zero*)

lemma *symp-ratrel*: *symp ratrel*
by (*simp add*: *ratrel-def symp-def*)

lemma *transp-ratrel*: *transp ratrel*
proof (*rule transpI*, *unfold split-paired-all*)
fix *a b a′ b′ a″ b″* :: *int*
assume ∗: *ratrel* (*a*, *b*) (*a′*, *b′*)
assume ∗∗: *ratrel* (*a′*, *b′*) (*a″*, *b″*)
have *b′* ∗ (*a* ∗ *b″*) = *b″* ∗ (*a* ∗ *b′*) **by** *simp*
also from ∗ **have** *a* ∗ *b′* = *a′* ∗ *b* **by** *auto*
also have *b″* ∗ (*a′* ∗ *b*) = *b* ∗ (*a′* ∗ *b″*) **by** *simp*
also from ∗∗ **have** *a′* ∗ *b″* = *a″* ∗ *b′* **by** *auto*
also have *b* ∗ (*a″* ∗ *b′*) = *b′* ∗ (*a″* ∗ *b*) **by** *simp*
finally have *b′* ∗ (*a* ∗ *b″*) = *b′* ∗ (*a″* ∗ *b*) **.**
moreover from ∗∗ **have** *b′* ≠ *0* **by** *auto*
ultimately have *a* ∗ *b″* = *a″* ∗ *b* **by** *simp*
with ∗ ∗∗ **show** *ratrel* (*a*, *b*) (*a″*, *b″*) **by** *auto*
qed

lemma *part-equivp-ratrel*: *part-equivp ratrel*
by (*rule part-equivpI* [*OF exists-ratrel-refl symp-ratrel transp-ratrel*])

quotient-type *rat* = *int* × *int* / *partial*: *ratrel*
morphisms *Rep-Rat Abs-Rat*
by (*rule part-equivp-ratrel*)

lemma *Domainp-cr-rat* [*transfer-domain-rule*]: *Domainp pcr-rat* = (λ*x*. *snd x* ≠ *0*)
by (*simp add*: *rat.domain-eq*)

96.1.2 Representation and basic operations

lift-definition *Fract* :: *int* ⇒ *int* ⇒ *rat*
is λ*a b*. *if b* = *0 then* (*0*, *1*) *else* (*a*, *b*)
by *simp*

lemma *eq-rat*:
⋀*a b c d*. *b* ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* = *Fract c d* ⟷ *a* ∗ *d* = *c* ∗ *b*
⋀*a*. *Fract a 0* = *Fract 0 1*
⋀*a c*. *Fract 0 a* = *Fract 0 c*
by (*transfer*, *simp*)+

lemma *Rat-cases* [*case-names Fract*, *cases type*: *rat*]:
assumes *that*: ⋀*a b*. *q* = *Fract a b* ⟹ *b* > *0* ⟹ *coprime a b* ⟹ *C*
shows *C*
proof −
obtain *a b* :: *int* **where** *q*: *q* = *Fract a b* **and** *b*: *b* ≠ *0*
by *transfer simp*
let *?a* = *a div gcd a b*
let *?b* = *b div gcd a b*
from *b* **have** *?b* ∗ *gcd a b* = *b*
by *simp*
with *b* **have** *?b* ≠ *0*
by *fastforce*
with *q b* **have** *q2*: *q* = *Fract ?a ?b*
by (*simp add*: *eq-rat dvd-div-mult mult.commute* [*of a*])
from *b* **have** *coprime*: *coprime ?a ?b*
by (*auto intro*: *div-gcd-coprime*)
show *C*
proof (*cases b* > *0*)
case *True*
then have *?b* > *0*
by (*simp add*: *nonneg1-imp-zdiv-pos-iff*)
from *q2 this coprime* **show** *C* **by** (*rule that*)
next
case *False*
have *q* = *Fract* (− *?a*) (− *?b*)
unfolding *q2* **by** *transfer simp*

```
    moreover from False b have - ?b > 0
      by (simp add: pos-imp-zdiv-neg-iff)
    moreover from coprime have coprime (- ?a) (- ?b)
      by simp
    ultimately show C
      by (rule that)
  qed
qed

lemma Rat-induct [case-names Fract, induct type: rat]:
  assumes ⋀a b. b > 0 ⟹ coprime a b ⟹ P (Fract a b)
  shows P q
  using assms by (cases q) simp

instantiation rat :: field
begin

lift-definition zero-rat :: rat is (0, 1)
  by simp

lift-definition one-rat :: rat is (1, 1)
  by simp

lemma Zero-rat-def: 0 = Fract 0 1
  by transfer simp

lemma One-rat-def: 1 = Fract 1 1
  by transfer simp

lift-definition plus-rat :: rat ⇒ rat ⇒ rat
  is λx y. (fst x * snd y + fst y * snd x, snd x * snd y)
  by (auto simp: distrib-right) (simp add: ac-simps)

lemma add-rat [simp]:
  assumes b ≠ 0 and d ≠ 0
  shows Fract a b + Fract c d = Fract (a * d + c * b) (b * d)
  using assms by transfer simp

lift-definition uminus-rat :: rat ⇒ rat is λx. (- fst x, snd x)
  by simp

lemma minus-rat [simp]: - Fract a b = Fract (- a) b
  by transfer simp

lemma minus-rat-cancel [simp]: Fract (- a) (- b) = Fract a b
  by (cases b = 0) (simp-all add: eq-rat)

definition diff-rat-def: q - r = q + - r for q r :: rat
```

lemma *diff-rat* [*simp*]:
b ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* − *Fract c d* = *Fract* (*a* ∗ *d* − *c* ∗ *b*) (*b* ∗ *d*)
by (*simp add*: *diff-rat-def*)

lift-definition *times-rat* :: *rat* ⇒ *rat* ⇒ *rat*
is λ*x y*. (*fst x* ∗ *fst y*, *snd x* ∗ *snd y*)
by (*simp add*: *ac-simps*)

lemma *mult-rat* [*simp*]: *Fract a b* ∗ *Fract c d* = *Fract* (*a* ∗ *c*) (*b* ∗ *d*)
by *transfer simp*

lemma *mult-rat-cancel*: *c* ≠ *0* ⟹ *Fract* (*c* ∗ *a*) (*c* ∗ *b*) = *Fract a b*
by *transfer simp*

lift-definition *inverse-rat* :: *rat* ⇒ *rat*
is λ*x*. *if fst x* = *0 then* (*0*, *1*) *else* (*snd x*, *fst x*)
by (*auto simp add*: *mult.commute*)

lemma *inverse-rat* [*simp*]: *inverse* (*Fract a b*) = *Fract b a*
by *transfer simp*

definition *divide-rat-def*: *q div r* = *q* ∗ *inverse r* **for** *q r* :: *rat*

lemma *divide-rat* [*simp*]: *Fract a b div Fract c d* = *Fract* (*a* ∗ *d*) (*b* ∗ *c*)
by (*simp add*: *divide-rat-def*)

instance
proof
fix *q r s* :: *rat*
show (*q* ∗ *r*) ∗ *s* = *q* ∗ (*r* ∗ *s*)
by *transfer simp*
show *q* ∗ *r* = *r* ∗ *q*
by *transfer simp*
show *1* ∗ *q* = *q*
by *transfer simp*
show (*q* + *r*) + *s* = *q* + (*r* + *s*)
by *transfer* (*simp add*: *algebra-simps*)
show *q* + *r* = *r* + *q*
by *transfer simp*
show *0* + *q* = *q*
by *transfer simp*
show − *q* + *q* = *0*
by *transfer simp*
show *q* − *r* = *q* + − *r*
by (*fact diff-rat-def*)
show (*q* + *r*) ∗ *s* = *q* ∗ *s* + *r* ∗ *s*
by *transfer* (*simp add*: *algebra-simps*)
show (*0*::*rat*) ≠ *1*
by *transfer simp*

show *inverse q ∗ q = 1* **if** *q ≠ 0*
using *that* **by** *transfer simp*
show *q div r = q ∗ inverse r*
by (*fact divide-rat-def*)
show *inverse 0 = (0::rat)*
by *transfer simp*
qed

end

lemma *div-add-self1-no-field* [*simp*]:
assumes *NO-MATCH* (*x* :: *'b* :: *field*) *b* (*b* :: *'a* :: *euclidean-semiring-cancel*) *≠ 0*
shows (*b + a*) *div b = a div b + 1*
using *assms*(*2*) **by** (*fact div-add-self1*)

lemma *div-add-self2-no-field* [*simp*]:
assumes *NO-MATCH* (*x* :: *'b* :: *field*) *b* (*b* :: *'a* :: *euclidean-semiring-cancel*) *≠ 0*
shows (*a + b*) *div b = a div b + 1*
using *assms*(*2*) **by** (*fact div-add-self2*)

lemma *of-nat-rat*: *of-nat k = Fract* (*of-nat k*) *1*
by (*induct k*) (*simp-all add*: *Zero-rat-def One-rat-def*)

lemma *of-int-rat*: *of-int k = Fract k 1*
by (*cases k rule*: *int-diff-cases*) (*simp add*: *of-nat-rat*)

lemma *Fract-of-nat-eq*: *Fract* (*of-nat k*) *1 = of-nat k*
by (*rule of-nat-rat* [*symmetric*])

lemma *Fract-of-int-eq*: *Fract k 1 = of-int k*
by (*rule of-int-rat* [*symmetric*])

lemma *rat-number-collapse*:
Fract 0 k = 0
Fract 1 1 = 1
Fract (*numeral w*) *1 = numeral w*
Fract (*− numeral w*) *1 = − numeral w*
Fract (*− 1*) *1 = − 1*
Fract k 0 = 0
using *Fract-of-int-eq* [*of numeral w*]
and *Fract-of-int-eq* [*of − numeral w*]
by (*simp-all add*: *Zero-rat-def One-rat-def eq-rat*)

lemma *rat-number-expand*:
0 = Fract 0 1
1 = Fract 1 1

numeral k = Fract (numeral k) 1
− 1 = Fract (− 1) 1
− numeral k = Fract (− numeral k) 1
by (*simp-all add*: *rat-number-collapse*)

lemma *Rat-cases-nonzero* [*case-names Fract 0*]:
assumes *Fract*: $\bigwedge a\ b.\ q = Fract\ a\ b \Longrightarrow b > 0 \Longrightarrow a \neq 0 \Longrightarrow coprime\ a\ b \Longrightarrow C$
and *0*: $q = 0 \Longrightarrow C$
shows *C*
proof (*cases q = 0*)
case *True*
then show *C* **using** *0* **by** *auto*
next
case *False*
then obtain *a b* **where** ∗: *q = Fract a b b > 0 coprime a b*
by (*cases q*) *auto*
with *False* **have** *0 ≠ Fract a b*
by *simp*
with ‹*b > 0*› **have** *a ≠ 0*
by (*simp add*: *Zero-rat-def eq-rat*)
with *Fract* ∗ **show** *C* **by** *blast*
qed

96.1.3 Function *normalize*

lemma *Fract-coprime*: *Fract (a div gcd a b) (b div gcd a b) = Fract a b*
proof (*cases b = 0*)
case *True*
then show *?thesis*
by (*simp add*: *eq-rat*)
next
case *False*
moreover have *b div gcd a b ∗ gcd a b = b*
by (*rule dvd-div-mult-self*) *simp*
ultimately have *b div gcd a b ∗ gcd a b ≠ 0*
by *simp*
then have *b div gcd a b ≠ 0*
by *fastforce*
with *False* **show** *?thesis*
by (*simp add*: *eq-rat dvd-div-mult mult.commute* [*of a*])
qed

definition *normalize* :: *int × int ⇒ int × int*
where *normalize p =*
(*if snd p > 0 then* (*let a = gcd* (*fst p*) (*snd p*) *in* (*fst p div a*, *snd p div a*))
else if snd p = 0 then (*0*, *1*)
else (*let a = − gcd* (*fst p*) (*snd p*) *in* (*fst p div a*, *snd p div a*)))

lemma *normalize-crossproduct*:
assumes $q \neq 0$ $s \neq 0$
assumes *normalize* $(p, q) =$ *normalize* (r, s)
shows $p * s = r * q$
proof −
have $*$: $p * s = q * r$
if $p * gcd\ r\ s = sgn\ (q * s) * r * gcd\ p\ q$ **and** $q * gcd\ r\ s = sgn\ (q * s) * s * gcd\ p\ q$
proof −
from *that* **have** $(p * gcd\ r\ s) * (sgn\ (q * s) * s * gcd\ p\ q) = (q * gcd\ r\ s) * (sgn\ (q * s) * r * gcd\ p\ q)$
by *simp*
with *assms* **show** *?thesis*
by (*auto simp add*: *ac-simps sgn-mult sgn-0-0*)
qed
from *assms* **show** *?thesis*
by (*auto simp*: *normalize-def Let-def dvd-div-div-eq-mult mult.commute sgn-mult split*: *if-splits intro*: $*$)
qed

lemma *normalize-eq*: *normalize* $(a, b) = (p, q) \Longrightarrow$ *Fract* $p\ q =$ *Fract* $a\ b$
by (*auto simp*: *normalize-def Let-def Fract-coprime dvd-div-neg rat-number-collapse split*: *if-split-asm*)

lemma *normalize-denom-pos*: *normalize* $r = (p, q) \Longrightarrow q > 0$
by (*auto simp*: *normalize-def Let-def dvd-div-neg pos-imp-zdiv-neg-iff nonneg1-imp-zdiv-pos-iff split*: *if-split-asm*)

lemma *normalize-coprime*: *normalize* $r = (p, q) \Longrightarrow$ *coprime* $p\ q$
by (*auto simp*: *normalize-def Let-def dvd-div-neg div-gcd-coprime split*: *if-split-asm*)

lemma *normalize-stable* [*simp*]: $q > 0 \Longrightarrow$ *coprime* $p\ q \Longrightarrow$ *normalize* $(p, q) = (p, q)$
by (*simp add*: *normalize-def*)

lemma *normalize-denom-zero* [*simp*]: *normalize* $(p, 0) = (0, 1)$
by (*simp add*: *normalize-def*)

lemma *normalize-negative* [*simp*]: $q < 0 \Longrightarrow$ *normalize* $(p, q) =$ *normalize* $(-p, -q)$
by (*simp add*: *normalize-def Let-def dvd-div-neg dvd-neg-div*)

Decompose a fraction into normalized, i.e. coprime numerator and denominator:

definition *quotient-of* :: *rat* $\Rightarrow$ *int* $\times$ *int*
where *quotient-of* $x =$
(*THE pair*. $x =$ *Fract* (*fst pair*) (*snd pair*) $\wedge$ *snd pair* $> 0 \wedge$ *coprime* (*fst pair*) (*snd pair*))

lemma *quotient-of-unique*: $\exists!p.\ r = Fract\ (fst\ p)\ (snd\ p) \wedge snd\ p > 0 \wedge coprime\ (fst\ p)\ (snd\ p)$
proof (*cases r*)
  case (*Fract a b*)
  then have $r = Fract\ (fst\ (a,\ b))\ (snd\ (a,\ b)) \wedge$
    $snd\ (a,\ b) > 0 \wedge coprime\ (fst\ (a,\ b))\ (snd\ (a,\ b))$
    by *auto*
  then show *?thesis*
  proof (*rule ex1I*)
    fix *p*
    assume *r*: $r = Fract\ (fst\ p)\ (snd\ p) \wedge snd\ p > 0 \wedge coprime\ (fst\ p)\ (snd\ p)$
    obtain *c d* **where** *p*: $p = (c,\ d)$ **by** (*cases p*)
    with *r* **have** *Fract'*: $r = Fract\ c\ d\ d > 0\ coprime\ c\ d$
      by *simp-all*
    have $(c,\ d) = (a,\ b)$
    proof (*cases* $a = 0$)
      case *True*
      with *Fract Fract'* **show** *?thesis*
        by (*simp add*: *eq-rat*)
    next
      case *False*
      with *Fract Fract'* **have** $*$: $c * b = a * d$ **and** $c \neq 0$
        by (*auto simp add*: *eq-rat*)
      then have $c * b > 0 \longleftrightarrow a * d > 0$
        by *auto*
      with ‹$b > 0$› ‹$d > 0$› **have** $a > 0 \longleftrightarrow c > 0$
        by (*simp add*: *zero-less-mult-iff*)
      with ‹$a \neq 0$› ‹$c \neq 0$› **have** *sgn*: $sgn\ a = sgn\ c$
        by (*auto simp add*: *not-less*)
      from ‹*coprime a b*› ‹*coprime c d*› **have** $|a| * |d| = |c| * |b| \longleftrightarrow |a| = |c| \wedge |d| = |b|$
        by (*simp add*: *coprime-crossproduct-int*)
      with ‹$b > 0$› ‹$d > 0$› **have** $|a| * d = |c| * b \longleftrightarrow |a| = |c| \wedge d = b$
        by *simp*
      then have $a * sgn\ a * d = c * sgn\ c * b \longleftrightarrow a * sgn\ a = c * sgn\ c \wedge d = b$
        by (*simp add*: *abs-sgn*)
      with $sgn * $ **show** *?thesis*
        by (*auto simp add*: *sgn-0-0*)
    qed
    with *p* **show** $p = (a,\ b)$
      by *simp*
  qed
qed

lemma *quotient-of-Fract* [*code*]: *quotient-of* $(Fract\ a\ b) = normalize\ (a,\ b)$
proof −
  have $Fract\ a\ b = Fract\ (fst\ (normalize\ (a,\ b)))\ (snd\ (normalize\ (a,\ b)))$ (**is** *?Fract*)
    by (*rule sym*) (*auto intro*: *normalize-eq*)

moreover have *0 < snd (normalize (a, b))* (**is** *?denom-pos*)
by (*cases normalize (a, b)*) (*rule normalize-denom-pos*, *simp*)
moreover have *coprime (fst (normalize (a, b))) (snd (normalize (a, b)))* (**is** *?coprime*)
by (*rule normalize-coprime*) *simp*
ultimately have *?Fract ∧ ?denom-pos ∧ ?coprime* **by** *blast*
then have (*THE p. Fract a b = Fract (fst p) (snd p) ∧ 0 < snd p ∧*
coprime (fst p) (snd p)) = normalize (a, b)
by (*rule the1-equality* [*OF quotient-of-unique*])
then show *?thesis* **by** (*simp add*: *quotient-of-def*)
qed

lemma *quotient-of-number* [*simp*]:
quotient-of 0 = (0, 1)
quotient-of 1 = (1, 1)
quotient-of (numeral k) = (numeral k, 1)
quotient-of (− 1) = (− 1, 1)
quotient-of (− numeral k) = (− numeral k, 1)
by (*simp-all add*: *rat-number-expand quotient-of-Fract*)

lemma *quotient-of-eq*: *quotient-of (Fract a b) = (p, q) ⟹ Fract p q = Fract a b*
by (*simp add*: *quotient-of-Fract normalize-eq*)

lemma *quotient-of-denom-pos*: *quotient-of r = (p, q) ⟹ q > 0*
by (*cases r*) (*simp add*: *quotient-of-Fract normalize-denom-pos*)

lemma *quotient-of-denom-pos'*: *snd (quotient-of r) > 0*
using *quotient-of-denom-pos* [*of r*] **by** (*simp add*: *prod-eq-iff*)

lemma *quotient-of-coprime*: *quotient-of r = (p, q) ⟹ coprime p q*
by (*cases r*) (*simp add*: *quotient-of-Fract normalize-coprime*)

lemma *quotient-of-inject*:
assumes *quotient-of a = quotient-of b*
shows *a = b*
proof −
obtain *p q r s* **where** *a*: *a = Fract p q* **and** *b*: *b = Fract r s* **and** *q > 0* **and** *s > 0*
by (*cases a*, *cases b*)
with *assms* **show** *?thesis*
by (*simp add*: *eq-rat quotient-of-Fract normalize-crossproduct*)
qed

lemma *quotient-of-inject-eq*: *quotient-of a = quotient-of b ⟷ a = b*
by (*auto simp add*: *quotient-of-inject*)

96.1.4 Various

lemma *Fract-of-int-quotient*: *Fract k l = of-int k / of-int l*

by (*simp add*: *Fract-of-int-eq* [*symmetric*])

lemma *Fract-add-one*: $n \neq 0 \Longrightarrow$ *Fract* $(m + n)$ n = *Fract* m $n + 1$
by (*simp add*: *rat-number-expand*)

lemma *quotient-of-div*:
assumes r: *quotient-of* $r = (n,d)$
shows r = *of-int* n / *of-int* d
proof −
from *theI'*[*OF quotient-of-unique*[*of* r], *unfolded* r[*unfolded quotient-of-def*]]
have r = *Fract* n d **by** *simp*
then show *?thesis* **using** *Fract-of-int-quotient*
by *simp*
qed

lemma *Fract-quotient-of* [*simp*]: *Fract* (*fst* (*quotient-of* r)) (*snd* (*quotient-of* r))
= r
using *Fract-of-int-quotient quotient-of-div* **by** *auto*

96.1.5 The ordered field of rational numbers

lift-definition *positive* :: *rat* ⇒ *bool*
is $\lambda x.\ 0 <$ *fst* $x *$ *snd* x
proof *clarsimp*
fix a b c d :: *int*
assume $b \neq 0$ **and** $d \neq 0$ **and** $a * d = c * b$
then have $a * d * b * d = c * b * b * d$
by *simp*
then have $a * b * d^2 = c * d * b^2$
unfolding *power2-eq-square* **by** (*simp add*: *ac-simps*)
then have $0 < a * b * d^2 \longleftrightarrow 0 < c * d * b^2$
by *simp*
then show $0 < a * b \longleftrightarrow 0 < c * d$
using ‹$b \neq 0$› **and** ‹$d \neq 0$›
by (*simp add*: *zero-less-mult-iff*)
qed

lemma *positive-zero*: ¬ *positive* 0
by *transfer simp*

lemma *positive-add*: *positive* $x \Longrightarrow$ *positive* $y \Longrightarrow$ *positive* $(x + y)$
apply *transfer*
by (*metis add-neg-neg fst-eqD mult-less-0-iff pos-add-strict snd-eqD zero-less-mult-iff*)

lemma *positive-mult*: *positive* $x \Longrightarrow$ *positive* $y \Longrightarrow$ *positive* $(x * y)$
apply *transfer*
by (*metis mult-less-0-iff split-pairs zero-less-mult-iff*)

lemma *positive-minus*: ¬ *positive* $x \Longrightarrow x \neq 0 \Longrightarrow$ *positive* $(- x)$

by *transfer* (*auto simp*: *neq-iff zero-less-mult-iff mult-less-0-iff*)

instantiation *rat* :: *linordered-field*
begin

definition $x < y \longleftrightarrow$ *positive* $(y - x)$

definition $x \leq y \longleftrightarrow x < y \vee x = y$ **for** $x\ y$ :: *rat*

definition $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$ **for** a :: *rat*

definition *sgn* $a = (\mathit{if}\ a = 0\ \mathit{then}\ 0\ \mathit{else\ if}\ 0 < a\ \mathit{then}\ 1\ \mathit{else} - 1)$ **for** a :: *rat*

instance
proof
fix $a\ b\ c$ :: *rat*
show $|a| = (\mathit{if}\ a < 0\ \mathit{then} - a\ \mathit{else}\ a)$
by (*rule abs-rat-def*)
show $a < b \longleftrightarrow a \leq b \wedge \neg\ b \leq a$
unfolding *less-eq-rat-def less-rat-def*
using *positive-add positive-zero* **by** *force*
show $a \leq a$
unfolding *less-eq-rat-def* **by** *simp*
show $a \leq b \Longrightarrow b \leq c \Longrightarrow a \leq c$
unfolding *less-eq-rat-def less-rat-def*
using *positive-add* **by** *fastforce*
show $a \leq b \Longrightarrow b \leq a \Longrightarrow a = b$
unfolding *less-eq-rat-def less-rat-def*
using *positive-add positive-zero* **by** *fastforce*
show $a \leq b \Longrightarrow c + a \leq c + b$
unfolding *less-eq-rat-def less-rat-def* **by** *auto*
show *sgn* $a = (\mathit{if}\ a = 0\ \mathit{then}\ 0\ \mathit{else\ if}\ 0 < a\ \mathit{then}\ 1\ \mathit{else} - 1)$
by (*rule sgn-rat-def*)
show $a \leq b \vee b \leq a$
unfolding *less-eq-rat-def less-rat-def*
by (*auto dest*!: *positive-minus*)
show $a < b \Longrightarrow 0 < c \Longrightarrow c * a < c * b$
unfolding *less-rat-def*
by (*metis diff-zero positive-mult right-diff-distrib'*)
qed

end

instantiation *rat* :: *distrib-lattice*
begin

definition (*inf* :: *rat* ⇒ *rat* ⇒ *rat*) = *min*

definition (*sup* :: *rat* ⇒ *rat* ⇒ *rat*) = *max*

instance
by *standard* (*auto simp add*: *inf-rat-def sup-rat-def max-min-distrib2*)

end

lemma *positive-rat*: *positive* (*Fract a b*) ⟷ *0* < *a* * *b*
by *transfer simp*

lemma *less-rat* [*simp*]:
b ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* < *Fract c d* ⟷ (*a* * *d*) * (*b* * *d*) < (*c* * *b*) * (*b* * *d*)
by (*simp add*: *less-rat-def positive-rat algebra-simps*)

lemma *le-rat* [*simp*]:
b ≠ *0* ⟹ *d* ≠ *0* ⟹ *Fract a b* ≤ *Fract c d* ⟷ (*a* * *d*) * (*b* * *d*) ≤ (*c* * *b*) * (*b* * *d*)
by (*simp add*: *le-less eq-rat*)

lemma *abs-rat* [*simp*, *code*]: |*Fract a b*| = *Fract* |*a*| |*b*|
by (*auto simp add*: *abs-rat-def zabs-def Zero-rat-def not-less le-less eq-rat zero-less-mult-iff*)

lemma *sgn-rat* [*simp*, *code*]: *sgn* (*Fract a b*) = *of-int* (*sgn a* * *sgn b*)
unfolding *Fract-of-int-eq*
by (*auto simp*: *zsgn-def sgn-rat-def Zero-rat-def eq-rat*)
(*auto simp*: *rat-number-collapse not-less le-less zero-less-mult-iff*)

lemma *Rat-induct-pos* [*case-names Fract*, *induct type*: *rat*]:
assumes *step*: ⋀*a b*. *0* < *b* ⟹ *P* (*Fract a b*)
shows *P q*
proof (*cases q*)
case (*Fract a b*)
have *step'*: *P* (*Fract a b*) **if** *b*: *b* < *0* **for** *a b* :: *int*
proof −
from *b* **have** *0* < − *b*
by *simp*
then have *P* (*Fract* (− *a*) (− *b*))
by (*rule step*)
then show *P* (*Fract a b*)
by (*simp add*: *order-less-imp-not-eq* [*OF b*])
qed
from *Fract* **show** *P q*
by (*auto simp add*: *linorder-neq-iff step step'*)
qed

lemma *zero-less-Fract-iff*: *0* < *b* ⟹ *0* < *Fract a b* ⟷ *0* < *a*
by (*simp add*: *Zero-rat-def zero-less-mult-iff*)

lemma *Fract-less-zero-iff*: *0* < *b* ⟹ *Fract a b* < *0* ⟷ *a* < *0*

by (*simp add*: *Zero-rat-def mult-less-0-iff*)

lemma *zero-le-Fract-iff*: $0 < b \Longrightarrow 0 \leq Fract\ a\ b \longleftrightarrow 0 \leq a$
by (*simp add*: *Zero-rat-def zero-le-mult-iff*)

lemma *Fract-le-zero-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 0 \longleftrightarrow a \leq 0$
by (*simp add*: *Zero-rat-def mult-le-0-iff*)

lemma *one-less-Fract-iff*: $0 < b \Longrightarrow 1 < Fract\ a\ b \longleftrightarrow b < a$
by (*simp add*: *One-rat-def mult-less-cancel-right-disj*)

lemma *Fract-less-one-iff*: $0 < b \Longrightarrow Fract\ a\ b < 1 \longleftrightarrow a < b$
by (*simp add*: *One-rat-def mult-less-cancel-right-disj*)

lemma *one-le-Fract-iff*: $0 < b \Longrightarrow 1 \leq Fract\ a\ b \longleftrightarrow b \leq a$
by (*simp add*: *One-rat-def mult-le-cancel-right*)

lemma *Fract-le-one-iff*: $0 < b \Longrightarrow Fract\ a\ b \leq 1 \longleftrightarrow a \leq b$
by (*simp add*: *One-rat-def mult-le-cancel-right*)

96.1.6 Rationals are an Archimedean field

lemma *rat-floor-lemma*: *of-int* (*a div b*) ≤ *Fract a b* ∧ *Fract a b* < *of-int* (*a div b* + *1*)
proof −
have *Fract a b* = *of-int* (*a div b*) + *Fract* (*a mod b*) *b*
by (*cases b* = *0*) (*simp*, *simp add*: *of-int-rat*)
moreover have *0* ≤ *Fract* (*a mod b*) *b* ∧ *Fract* (*a mod b*) *b* < *1*
unfolding *Fract-of-int-quotient*
by (*rule linorder-cases* [*of b 0*]) (*simp-all add*: *divide-nonpos-neg*)
ultimately show *?thesis* **by** *simp*
qed

instance *rat* :: *archimedean-field*
proof
show ∃ *z*. *r* ≤ *of-int z* **for** *r* :: *rat*
proof (*induct r*)
case (*Fract a b*)
have *Fract a b* ≤ *of-int* (*a div b* + *1*)
using *rat-floor-lemma* [*of a b*] **by** *simp*
then show ∃ *z*. *Fract a b* ≤ *of-int z* **..**
qed
qed

instantiation *rat* :: *floor-ceiling*
begin

definition *floor-rat* :: *rat* ⇒ *int*
where⌊*x*⌋ = (*THE z*. *of-int z* ≤ *x* ∧ *x* < *of-int* (*z* + *1*)) **for** *x* :: *rat*

instance
proof
show *of-int* $\lfloor x \rfloor \leq x \wedge x <$ *of-int* $(\lfloor x \rfloor + 1)$ **for** $x ::$ *rat*
unfolding *floor-rat-def* **using** *floor-exists1* **by** (*rule theI'*)
qed

end

lemma *floor-Fract* [*simp*]: $\lfloor$*Fract a b*$\rfloor$ = *a div b*
by (*simp add*: *Fract-of-int-quotient floor-divide-of-int-eq*)

96.2 Linear arithmetic setup

declaration ‹
K (*Lin-Arith.add-inj-thms* @{*thms of-int-le-iff* [*THEN iffD2*] *of-int-eq-iff* [*THEN iffD2*]}
(∗ *not needed because x < (y::int) can be rewritten as x + 1 <= y*: *of-int-less-iff RS iffD2* ∗)
#> *Lin-Arith.add-inj-const* (***const-name*** ‹*of-nat*›, ***typ*** ‹*nat* ⇒ *rat*›)
#> *Lin-Arith.add-inj-const* (***const-name*** ‹*of-int*›, ***typ*** ‹*int* ⇒ *rat*›))
›

96.3 Embedding from Rationals to other Fields

context *field-char-0*
begin

lift-definition *of-rat* :: *rat* ⇒ *'a*
is λ*x*. *of-int* (*fst x*) / *of-int* (*snd x*)
by (*auto simp*: *nonzero-divide-eq-eq nonzero-eq-divide-eq*) (*simp only*: *of-int-mult* [*symmetric*])

end

lemma *of-rat-rat*: *b* ≠ *0* ⟹ *of-rat* (*Fract a b*) = *of-int a* / *of-int b*
by *transfer simp*

lemma *of-rat-0* [*simp*]: *of-rat 0* = *0*
by *transfer simp*

lemma *of-rat-1* [*simp*]: *of-rat 1* = *1*
by *transfer simp*

lemma *of-rat-add*: *of-rat* (*a* + *b*) = *of-rat a* + *of-rat b*
by *transfer* (*simp add*: *add-frac-eq*)

lemma *of-rat-minus*: *of-rat* (− *a*) = − *of-rat a*
by *transfer simp*

lemma *of-rat-neg-one* [*simp*]: *of-rat* (− *1*) = − *1*
by (*simp add*: *of-rat-minus*)

lemma *of-rat-diff*: *of-rat* (*a* − *b*) = *of-rat a* − *of-rat b*
using *of-rat-add* [*of a* − *b*] **by** (*simp add*: *of-rat-minus*)

lemma *of-rat-mult*: *of-rat* (*a* ∗ *b*) = *of-rat a* ∗ *of-rat b*
by *transfer* (*simp add*: *divide-inverse nonzero-inverse-mult-distrib ac-simps*)

lemma *of-rat-sum*: *of-rat* (∑ *a*∈*A*. *f a*) = (∑ *a*∈*A*. *of-rat* (*f a*))
by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *of-rat-add*)

lemma *of-rat-prod*: *of-rat* (∏ *a*∈*A*. *f a*) = (∏ *a*∈*A*. *of-rat* (*f a*))
by (*induct rule*: *infinite-finite-induct*) (*auto simp*: *of-rat-mult*)

lemma *nonzero-of-rat-inverse*: *a* ≠ *0* ⟹ *of-rat* (*inverse a*) = *inverse* (*of-rat a*)
by (*rule inverse-unique* [*symmetric*]) (*simp add*: *of-rat-mult* [*symmetric*])

lemma *of-rat-inverse*: (*of-rat* (*inverse a*) :: *'a*::*field-char-0*) = *inverse* (*of-rat a*)
by (*cases a* = *0*) (*simp-all add*: *nonzero-of-rat-inverse*)

lemma *nonzero-of-rat-divide*: *b* ≠ *0* ⟹ *of-rat* (*a* / *b*) = *of-rat a* / *of-rat b*
by (*simp add*: *divide-inverse of-rat-mult nonzero-of-rat-inverse*)

lemma *of-rat-divide*: (*of-rat* (*a* / *b*) :: *'a*::*field-char-0*) = *of-rat a* / *of-rat b*
by (*cases b* = *0*) (*simp-all add*: *nonzero-of-rat-divide*)

lemma *of-rat-power*: (*of-rat* (*a* ^ *n*) :: *'a*::*field-char-0*) = *of-rat a* ^ *n*
by (*induct n*) (*simp-all add*: *of-rat-mult*)

lemma *of-rat-eq-iff* [*simp*]: *of-rat a* = *of-rat b* ⟷ *a* = *b*
apply *transfer*
apply (*simp add*: *nonzero-divide-eq-eq nonzero-eq-divide-eq flip*: *of-int-mult*)
done

lemma *of-rat-eq-0-iff* [*simp*]: *of-rat a* = *0* ⟷ *a* = *0*
using *of-rat-eq-iff* [*of - 0*] **by** *simp*

lemma *zero-eq-of-rat-iff* [*simp*]: *0* = *of-rat a* ⟷ *0* = *a*
by *simp*

lemma *of-rat-eq-1-iff* [*simp*]: *of-rat a* = *1* ⟷ *a* = *1*
using *of-rat-eq-iff* [*of - 1*] **by** *simp*

lemma *one-eq-of-rat-iff* [*simp*]: *1* = *of-rat a* ⟷ *1* = *a*
by *simp*

lemma *of-rat-less*: (*of-rat r* :: *'a*::*linordered-field*) < *of-rat s* ⟷ *r* < *s*
proof (*induct r*, *induct s*)

fix *a b c d* :: *int*
assume *not-zero*: *b* > *0 d* > *0*
then have *b* ∗ *d* > *0* **by** *simp*
have *of-int-divide-less-eq*:
(*of-int a* :: *'a*) / *of-int b* < *of-int c* / *of-int d* ⟷
(*of-int a* :: *'a*) ∗ *of-int d* < *of-int c* ∗ *of-int b*
using *not-zero* **by** (*simp add*: *pos-less-divide-eq pos-divide-less-eq*)
show (*of-rat* (*Fract a b*) :: *'a*::*linordered-field*) < *of-rat* (*Fract c d*) ⟷
Fract a b < *Fract c d*
using *not-zero* ‹*b* ∗ *d* > *0*›
by (*simp add*: *of-rat-rat of-int-divide-less-eq of-int-mult* [*symmetric*] *del*: *of-int-mult*)
qed

lemma *of-rat-less-eq*: (*of-rat r* :: *'a*::*linordered-field*) ≤ *of-rat s* ⟷ *r* ≤ *s*
unfolding *le-less* **by** (*auto simp add*: *of-rat-less*)

lemma *of-rat-le-0-iff* [*simp*]: (*of-rat r* :: *'a*::*linordered-field*) ≤ *0* ⟷ *r* ≤ *0*
using *of-rat-less-eq* [*of r 0*, **where** *'a* = *'a*] **by** *simp*

lemma *zero-le-of-rat-iff* [*simp*]: *0* ≤ (*of-rat r* :: *'a*::*linordered-field*) ⟷ *0* ≤ *r*
using *of-rat-less-eq* [*of 0 r*, **where** *'a* = *'a*] **by** *simp*

lemma *of-rat-le-1-iff* [*simp*]: (*of-rat r* :: *'a*::*linordered-field*) ≤ *1* ⟷ *r* ≤ *1*
using *of-rat-less-eq* [*of r 1*] **by** *simp*

lemma *one-le-of-rat-iff* [*simp*]: *1* ≤ (*of-rat r* :: *'a*::*linordered-field*) ⟷ *1* ≤ *r*
using *of-rat-less-eq* [*of 1 r*] **by** *simp*

lemma *of-rat-less-0-iff* [*simp*]: (*of-rat r* :: *'a*::*linordered-field*) < *0* ⟷ *r* < *0*
using *of-rat-less* [*of r 0*, **where** *'a* = *'a*] **by** *simp*

lemma *zero-less-of-rat-iff* [*simp*]: *0* < (*of-rat r* :: *'a*::*linordered-field*) ⟷ *0* < *r*
using *of-rat-less* [*of 0 r*, **where** *'a* = *'a*] **by** *simp*

lemma *of-rat-less-1-iff* [*simp*]: (*of-rat r* :: *'a*::*linordered-field*) < *1* ⟷ *r* < *1*
using *of-rat-less* [*of r 1*] **by** *simp*

lemma *one-less-of-rat-iff* [*simp*]: *1* < (*of-rat r* :: *'a*::*linordered-field*) ⟷ *1* < *r*
using *of-rat-less* [*of 1 r*] **by** *simp*

lemma *of-rat-eq-id* [*simp*]: *of-rat* = *id*
proof
show *of-rat a* = *id a* **for** *a*
by (*induct a*) (*simp add*: *of-rat-rat Fract-of-int-eq* [*symmetric*])
qed

lemma *abs-of-rat* [*simp*]:
|*of-rat r*| = (*of-rat* |*r*| :: *'a* :: *linordered-field*)
by (*cases r* ≥ *0*) (*simp-all add*: *not-le of-rat-minus*)

Collapse nested embeddings.

lemma *of-rat-of-nat-eq* [*simp*]: *of-rat* (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *of-rat-add*)

lemma *of-rat-of-int-eq* [*simp*]: *of-rat* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) (*simp add*: *of-rat-diff*)

lemma *of-rat-numeral-eq* [*simp*]: *of-rat* (*numeral w*) = *numeral w*
using *of-rat-of-int-eq* [*of numeral w*] **by** *simp*

lemma *of-rat-neg-numeral-eq* [*simp*]: *of-rat* (− *numeral w*) = − *numeral w*
using *of-rat-of-int-eq* [*of* − *numeral w*] **by** *simp*

lemma *of-rat-floor* [*simp*]:
⌊*of-rat r*⌋ = ⌊*r*⌋
by (*cases r*) (*simp add*: *Fract-of-int-quotient of-rat-divide floor-divide-of-int-eq*)

lemma *of-rat-ceiling* [*simp*]:
⌈*of-rat r*⌉ = ⌈*r*⌉
using *of-rat-floor* [*of* − *r*] **by** (*simp add*: *of-rat-minus ceiling-def*)

lemmas *zero-rat* = *Zero-rat-def*
lemmas *one-rat* = *One-rat-def*

abbreviation *rat-of-nat* :: *nat* ⇒ *rat*
where *rat-of-nat* ≡ *of-nat*

abbreviation *rat-of-int* :: *int* ⇒ *rat*
where *rat-of-int* ≡ *of-int*

96.4 The Set of Rational Numbers

context *field-char-0*
begin

definition *Rats* :: *'a set* (‹ℚ›)
where ℚ = *range of-rat*

end

lemma *Rats-cases* [*cases set*: *Rats*]:
assumes *q* ∈ ℚ
obtains (*of-rat*) *r* **where** *q* = *of-rat r*
proof −
from ‹*q* ∈ ℚ› **have** *q* ∈ *range of-rat*
by (*simp only*: *Rats-def*)
then obtain *r* **where** *q* = *of-rat r* **..**
then show *thesis* **..**
qed

lemma *Rats-cases'*:
assumes $(x :: 'a :: \textit{field-char-0}) \in \mathbb{Q}$
obtains *a b* **where** $b > 0$ *coprime a b* $x = \textit{of-int } a \mathbin{/} \textit{of-int } b$
proof −
from *assms* **obtain** *r* **where** $x = \textit{of-rat } r$
by (*auto simp*: *Rats-def*)
obtain *a b* **where** *quot*: *quotient-of r* = (*a*,*b*) **by** *force*
have $b > 0$ **using** *quotient-of-denom-pos*[*OF quot*] **.**
moreover have *coprime a b* **using** *quotient-of-coprime*[*OF quot*] **.**
moreover have $x = \textit{of-int } a \mathbin{/} \textit{of-int } b$ **unfolding** ‹$x = \textit{of-rat } r$›
quotient-of-div[*OF quot*] **by** (*simp add*: *of-rat-divide*)
ultimately show *?thesis* **using** *that* **by** *blast*
qed

lemma *Rats-of-rat* [*simp*]: *of-rat r* ∈ ℚ
by (*simp add*: *Rats-def*)

lemma *Rats-of-int* [*simp*]: *of-int z* ∈ ℚ
by (*subst of-rat-of-int-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Ints-subset-Rats*: ℤ ⊆ ℚ
using *Ints-cases Rats-of-int* **by** *blast*

lemma *Rats-of-nat* [*simp*]: *of-nat n* ∈ ℚ
by (*subst of-rat-of-nat-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Nats-subset-Rats*: ℕ ⊆ ℚ
using *Ints-subset-Rats Nats-subset-Ints* **by** *blast*

lemma *Rats-number-of* [*simp*]: *numeral w* ∈ ℚ
by (*subst of-rat-numeral-eq* [*symmetric*]) (*rule Rats-of-rat*)

lemma *Rats-0* [*simp*]: *0* ∈ ℚ
unfolding *Rats-def* **by** (*rule range-eqI*) (*rule of-rat-0* [*symmetric*])

lemma *Rats-1* [*simp*]: *1* ∈ ℚ
unfolding *Rats-def* **by** (*rule range-eqI*) (*rule of-rat-1* [*symmetric*])

lemma *Rats-add* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a + b \in \mathbb{Q}$
by (*metis Rats-cases Rats-of-rat of-rat-add*)

lemma *Rats-minus-iff* [*simp*]: $- a \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q}$
by (*metis Rats-cases Rats-of-rat add.inverse-inverse of-rat-minus*)

lemma *Rats-diff* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a - b \in \mathbb{Q}$
by (*metis Rats-add Rats-minus-iff diff-conv-add-uminus*)

lemma *Rats-mult* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a * b \in \mathbb{Q}$

by (*metis Rats-cases Rats-of-rat of-rat-mult*)

lemma *Rats-inverse* [*simp*]: $a \in \mathbb{Q} \Longrightarrow \mathit{inverse}\ a \in \mathbb{Q}$
for $a :: 'a$::*field-char-0*
by (*metis Rats-cases Rats-of-rat of-rat-inverse*)

lemma *Rats-divide* [*simp*]: $a \in \mathbb{Q} \Longrightarrow b \in \mathbb{Q} \Longrightarrow a\ /\ b \in \mathbb{Q}$
for $a\ b :: 'a$::*field-char-0*
by (*simp add*: *divide-inverse*)

lemma *Rats-power* [*simp*]: $a \in \mathbb{Q} \Longrightarrow a \mathbin{\hat{}} n \in \mathbb{Q}$
for $a :: 'a$::*field-char-0*
by (*metis Rats-cases Rats-of-rat of-rat-power*)

lemma *Rats-sum* [*intro*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Q}) \Longrightarrow \mathit{sum}\ f\ A \in \mathbb{Q}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *Rats-prod* [*intro*]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in \mathbb{Q}) \Longrightarrow \mathit{prod}\ f\ A \in \mathbb{Q}$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *Rats-induct* [*case-names of-rat*, *induct set*: *Rats*]: $q \in \mathbb{Q} \Longrightarrow (\bigwedge r.\ P\ (\mathit{of\text{-}rat}\ r)) \Longrightarrow P\ q$
by (*rule Rats-cases*) *auto*

lemma *Rats-infinite*: $\neg\ \mathit{finite}\ \mathbb{Q}$
by (*auto dest*!: *finite-imageD simp*: *inj-on-def infinite-UNIV-char-0 Rats-def*)

lemma *Rats-add-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \Longrightarrow a+b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
by (*metis Rats-add Rats-diff add-diff-cancel add-diff-cancel-left'*)

lemma *Rats-diff-iff*: $a \in \mathbb{Q} \vee b \in \mathbb{Q} \Longrightarrow a-b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
by (*metis Rats-add-iff diff-add-cancel*)

lemma *Rats-mult-iff*: $a \in \mathbb{Q}-\{0\} \vee b \in \mathbb{Q}-\{0\} \Longrightarrow a*b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
by (*metis Diff-iff Rats-divide Rats-mult insertI1 mult.commute nonzero-divide-eq-eq*)

lemma *Rats-inverse-iff* [*simp*]: $\mathit{inverse}\ a \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q}$
using *Rats-inverse* **by** *force*

lemma *Rats-divide-iff*: $a \in \mathbb{Q}-\{0\} \vee b \in \mathbb{Q}-\{0\} \Longrightarrow a/b \in \mathbb{Q} \longleftrightarrow a \in \mathbb{Q} \wedge b \in \mathbb{Q}$
by (*metis Rats-divide Rats-mult-iff divide-eq-0-iff divide-inverse nonzero-mult-div-cancel-right*)

96.5 Implementation of rational numbers as pairs of integers

Formal constructor

definition $\mathit{Frct} :: \mathit{int} \times \mathit{int} \Rightarrow \mathit{rat}$
where [*simp*]: $\mathit{Frct}\ p = \mathit{Fract}\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)$

lemma [*code abstype*]: *Frct* (*quotient-of q*) = *q*
by (*cases q*) (*auto intro*: *quotient-of-eq*)

Numerals

declare *quotient-of-Fract* [*code abstract*]

definition *of-int* :: *int* ⇒ *rat*
where [*code-abbrev*]: *of-int* = *Int.of-int*

hide-const (**open**) *of-int*

lemma *quotient-of-int* [*code abstract*]: *quotient-of* (*Rat.of-int a*) = (*a*, *1*)
by (*simp add*: *of-int-def of-int-rat quotient-of-Fract*)

lemma *quotient-of-rat-of-int* [*simp*]: *quotient-of* (*rat-of-int i*) = (*i*, *1*)
using *Rat.of-int-def quotient-of-int* **by** *force*

lemma *quotient-of-rat-of-nat* [*simp*]: *quotient-of* (*rat-of-nat i*) = (*int i*, *1*)
by (*metis of-int-of-nat-eq quotient-of-rat-of-int*)

lemma [*code-unfold*]: *numeral k* = *Rat.of-int* (*numeral k*)
by (*simp add*: *Rat.of-int-def*)

lemma [*code-unfold*]: − *numeral k* = *Rat.of-int* (− *numeral k*)
by (*simp add*: *Rat.of-int-def*)

lemma *Frct-code-post* [*code-post*]:
Frct (*0*, *a*) = *0*
Frct (*a*, *0*) = *0*
Frct (*1*, *1*) = *1*
Frct (*numeral k*, *1*) = *numeral k*
Frct (*1*, *numeral k*) = *1* / *numeral k*
Frct (*numeral k*, *numeral l*) = *numeral k* / *numeral l*
Frct (− *a*, *b*) = − *Frct* (*a*, *b*)
Frct (*a*, − *b*) = − *Frct* (*a*, *b*)
− (− *Frct q*) = *Frct q*
by (*simp-all add*: *Fract-of-int-quotient*)

Operations

lemma *rat-zero-code* [*code abstract*]: *quotient-of 0* = (*0*, *1*)
by (*simp add*: *Zero-rat-def quotient-of-Fract normalize-def*)

lemma *rat-one-code* [*code abstract*]: *quotient-of 1* = (*1*, *1*)
by (*simp add*: *One-rat-def quotient-of-Fract normalize-def*)

lemma *rat-plus-code* [*code abstract*]:
quotient-of (*p* + *q*) = (*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d* + *b* ∗ *c*, *c* ∗ *d*))
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract*)

lemma *rat-uminus-code* [*code abstract*]:
quotient-of (− *p*) = (*let* (*a*, *b*) = *quotient-of p in* (− *a*, *b*))
by (*cases p*) (*simp add*: *quotient-of-Fract*)

lemma *rat-minus-code* [*code abstract*]:
quotient-of (*p* − *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d* − *b* ∗ *c*, *c* ∗ *d*))
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract*)

lemma *rat-times-code* [*code abstract*]:
quotient-of (*p* ∗ *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *b*, *c* ∗ *d*))
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract*)

lemma *rat-inverse-code* [*code abstract*]:
quotient-of (*inverse p*) =
(*let* (*a*, *b*) = *quotient-of p*
in if a = *0 then* (*0*, *1*) *else* (*sgn a* ∗ *b*, |*a*|))
proof (*cases p*)
case (*Fract a b*)
then show *?thesis*
by (*cases 0*::*int a rule*: *linorder-cases*) (*simp-all add*: *quotient-of-Fract ac-simps*)
qed

lemma *rat-divide-code* [*code abstract*]:
quotient-of (*p* / *q*) =
(*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q*
in normalize (*a* ∗ *d*, *c* ∗ *b*))
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract*)

lemma *rat-abs-code* [*code abstract*]:
quotient-of |*p*| = (*let* (*a*, *b*) = *quotient-of p in* (|*a*|, *b*))
by (*cases p*) (*simp add*: *quotient-of-Fract*)

lemma *rat-sgn-code* [*code abstract*]: *quotient-of* (*sgn p*) = (*sgn* (*fst* (*quotient-of p*)), *1*)
proof (*cases p*)
case (*Fract a b*)
then show *?thesis*
by (*cases 0*::*int a rule*: *linorder-cases*) (*simp-all add*: *quotient-of-Fract*)
qed

lemma *rat-floor-code* [*code*]: ⌊*p*⌋ = (*let* (*a*, *b*) = *quotient-of p in a div b*)
by (*cases p*) (*simp add*: *quotient-of-Fract floor-Fract*)

instantiation *rat* :: *equal*

begin

definition [*code*]: *HOL.equal a b* ⟷ *quotient-of a = quotient-of b*

instance
by *standard* (*simp add*: *equal-rat-def quotient-of-inject-eq*)

lemma *rat-eq-refl* [*code nbe*]: *HOL.equal* (*r*::*rat*) *r* ⟷ *True*
by (*rule equal-refl*)

end

lemma *rat-less-eq-code* [*code*]:
p ≤ *q* ⟷ (*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q in a* ∗ *d* ≤ *c* ∗ *b*)
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract mult.commute*)

lemma *rat-less-code* [*code*]:
p < *q* ⟷ (*let* (*a*, *c*) = *quotient-of p*; (*b*, *d*) = *quotient-of q in a* ∗ *d* < *c* ∗ *b*)
by (*cases p*, *cases q*) (*simp add*: *quotient-of-Fract mult.commute*)

lemma [*code*]: *of-rat p* = (*let* (*a*, *b*) = *quotient-of p in of-int a* / *of-int b*)
by (*cases p*) (*simp add*: *quotient-of-Fract of-rat-rat*)

Quickcheck

context
includes *term-syntax*
begin

definition
valterm-fract :: *int* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
int × (*unit* ⇒ *Code-Evaluation.term*) ⇒
rat × (*unit* ⇒ *Code-Evaluation.term*)
where [*code-unfold*]: *valterm-fract k l* = *Code-Evaluation.valtermify Fract* {·} *k* {·} *l*

end

instantiation *rat* :: *random*
begin

context
includes *state-combinator-syntax*
begin

definition
Quickcheck-Random.random i =
Quickcheck-Random.random i ○→ (λ*num*. *Random.range i* ○→ (λ*denom*. *Pair*
(*let j* = *int-of-integer* (*integer-of-natural* (*denom* + *1*))
in valterm-fract num (*j*, λ*u*. *Code-Evaluation.term-of j*))))

instance ..

end

end

instantiation *rat* :: *exhaustive*
begin

definition
exhaustive-rat f d =
Quickcheck-Exhaustive.exhaustive
(λ*l*. *Quickcheck-Exhaustive.exhaustive*
(λ*k*. *f* (*Fract k* (*int-of-integer* (*integer-of-natural l*) + *1*))) *d*) *d*

instance ..

end

instantiation *rat* :: *full-exhaustive*
begin

definition
full-exhaustive-rat f d =
Quickcheck-Exhaustive.full-exhaustive
(λ(*l*, -). *Quickcheck-Exhaustive.full-exhaustive*
(λ*k*. *f*
(*let j* = *int-of-integer* (*integer-of-natural l*) + *1*
in valterm-fract k (*j*, λ-. *Code-Evaluation.term-of j*))) *d*) *d*

instance ..

end

instance *rat* :: *partial-term-of* **..**

lemma [*code*]:
partial-term-of (*ty* :: *rat itself*) (*Quickcheck-Narrowing.Narrowing-constructor 0* [*l*, *k*]) ≡
Code-Evaluation.App
(*Code-Evaluation.Const* (*STR ''Rat.Frct''*)
(*Typerep.Typerep* (*STR ''fun''*)
[*Typerep.Typerep* (*STR ''Product-Type.prod''*)
[*Typerep.Typerep* (*STR ''Int.int''*) [], *Typerep.Typerep* (*STR ''Int.int''*) []],
Typerep.Typerep (*STR ''Rat.rat''*) []]))
(*Code-Evaluation.App*
(*Code-Evaluation.App*
(*Code-Evaluation.Const* (*STR ''Product-Type.Pair''*)

(*Typerep.Typerep* (*STR ''fun''*)
[*Typerep.Typerep* (*STR ''Int.int''*) [],
Typerep.Typerep (*STR ''fun''*)
[*Typerep.Typerep* (*STR ''Int.int''*) [],
Typerep.Typerep (*STR ''Product-Type.prod''*)
[*Typerep.Typerep* (*STR ''Int.int''*) [], *Typerep.Typerep* (*STR ''Int.int''*)
[]]]]))
(*partial-term-of* (*TYPE*(*int*)) *l*)) (*partial-term-of* (*TYPE*(*int*)) *k*))
partial-term-of (*ty* :: *rat itself*) (*Quickcheck-Narrowing.Narrowing-variable p tt*)
≡
Code-Evaluation.Free (*STR ''-''*) (*Typerep.Typerep* (*STR ''Rat.rat''*) [])
by (*rule partial-term-of-anything*)+

instantiation *rat* :: *narrowing*
begin

definition
narrowing =
Quickcheck-Narrowing.apply
(*Quickcheck-Narrowing.apply*
(*Quickcheck-Narrowing.cons* (λ*nom denom*. *Fract nom denom*)) *narrowing*)
narrowing

instance ..

end

96.6 Setup for Nitpick

declaration ‹
Nitpick-HOL.register-frac-type ***type-name*** ‹*rat*›
[(***const-name*** ‹*Abs-Rat*›, ***const-name*** ‹*Nitpick.Abs-Frac*›),
(***const-name*** ‹*zero-rat-inst.zero-rat*›, ***const-name*** ‹*Nitpick.zero-frac*›),
(***const-name*** ‹*one-rat-inst.one-rat*›, ***const-name*** ‹*Nitpick.one-frac*›),
(***const-name*** ‹*plus-rat-inst.plus-rat*›, ***const-name*** ‹*Nitpick.plus-frac*›),
(***const-name*** ‹*times-rat-inst.times-rat*›, ***const-name*** ‹*Nitpick.times-frac*›),
(***const-name*** ‹*uminus-rat-inst.uminus-rat*›, ***const-name*** ‹*Nitpick.uminus-frac*›),
(***const-name*** ‹*inverse-rat-inst.inverse-rat*›, ***const-name*** ‹*Nitpick.inverse-frac*›),
(***const-name*** ‹*ord-rat-inst.less-rat*›, ***const-name*** ‹*Nitpick.less-frac*›),
(***const-name*** ‹*ord-rat-inst.less-eq-rat*›, ***const-name*** ‹*Nitpick.less-eq-frac*›),
(***const-name*** ‹*field-char-0-class.of-rat*›, ***const-name*** ‹*Nitpick.of-frac*›)]
›

lemmas [*nitpick-unfold*] =
inverse-rat-inst.inverse-rat
one-rat-inst.one-rat ord-rat-inst.less-rat
ord-rat-inst.less-eq-rat plus-rat-inst.plus-rat times-rat-inst.times-rat
uminus-rat-inst.uminus-rat zero-rat-inst.zero-rat

96.7 Float syntax

syntax *-Float :: float-const ⇒ 'a* (‹(‹*open-block notation=‹literal number››-*)›)

parse-translation ‹
let
fun mk-frac str =
let
val {mant = i, exp = n} = Lexicon.read-float str;
val exp = Syntax.const ***const-syntax*** ‹*Power.power*›;
val ten = Numeral.mk-number-syntax 10;
val exp10 = if n = 1 then ten else exp $ ten $ Numeral.mk-number-syntax n;
in Syntax.const ***const-syntax*** ‹*Fields.inverse-divide*› *$ Numeral.mk-number-syntax i $ exp10 end;*

*fun float-tr [(c as Const (****syntax-const*** ‹*-constrain*›*, -)) $ t $ u] = c $ float-tr [t] $ u*
| float-tr [t as Const (str, -)] = mk-frac str
| float-tr ts = raise TERM (float-tr, ts);
*in [(****syntax-const*** ‹*-Float*›*, K float-tr)] end*
›

Test:

lemma *123.456 = −111.111 + 200 + 30 + 4 + 5/10 + 6/100 + (7/1000::rat)*
by *simp*

96.8 Hiding implementation details

hide-const (**open**) *normalize positive*

lifting-update *rat.lifting*
lifting-forget *rat.lifting*

end

97 Development of the Reals using Cauchy Sequences

theory *Real*
imports *Rat*
begin

This theory contains a formalization of the real numbers as equivalence classes of Cauchy sequences of rationals. See the AFP entry *Dedekind-Real* for an alternative construction using Dedekind cuts.

97.1 Preliminary lemmas

Useful in convergence arguments

lemma *inverse-of-nat-le*:
fixes *n*::*nat* **shows** ⟦*n* ≤ *m*; *n*≠*0*⟧ ⟹ *1* / *of-nat m* ≤ (*1*::'*a*::*linordered-field*) / *of-nat n*
by (*simp add*: *frac-le*)

lemma *add-diff-add*: (*a* + *c*) − (*b* + *d*) = (*a* − *b*) + (*c* − *d*)
for *a b c d* :: '*a*::*ab-group-add*
by *simp*

lemma *minus-diff-minus*: − *a* − − *b* = − (*a* − *b*)
for *a b* :: '*a*::*ab-group-add*
by *simp*

lemma *mult-diff-mult*: (*x* ∗ *y* − *a* ∗ *b*) = *x* ∗ (*y* − *b*) + (*x* − *a*) ∗ *b*
for *x y a b* :: '*a*::*ring*
by (*simp add*: *algebra-simps*)

lemma *inverse-diff-inverse*:
fixes *a b* :: '*a*::*division-ring*
assumes *a* ≠ *0* **and** *b* ≠ *0*
shows *inverse a* − *inverse b* = − (*inverse a* ∗ (*a* − *b*) ∗ *inverse b*)
using *assms* **by** (*simp add*: *algebra-simps*)

lemma *obtain-pos-sum*:
fixes *r* :: *rat* **assumes** *r*: *0* < *r*
obtains *s t* **where** *0* < *s* **and** *0* < *t* **and** *r* = *s* + *t*
proof
from *r* **show** *0* < *r*/*2* **by** *simp*
from *r* **show** *0* < *r*/*2* **by** *simp*
show *r* = *r*/*2* + *r*/*2* **by** *simp*
qed

97.2 Sequences that converge to zero

definition *vanishes* :: (*nat* ⇒ *rat*) ⇒ *bool*
where *vanishes X* ⟷ (∀ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*)

lemma *vanishesI*: (⋀*r*. *0* < *r* ⟹ ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*) ⟹ *vanishes X*
unfolding *vanishes-def* **by** *simp*

lemma *vanishesD*: *vanishes X* ⟹ *0* < *r* ⟹ ∃ *k*. ∀ *n*≥*k*. |*X n*| < *r*
unfolding *vanishes-def* **by** *simp*

lemma *vanishes-const* [*simp*]: *vanishes* (λ*n*. *c*) ⟷ *c* = *0*
by (*metis abs-eq-0-iff less-imp-neq order-eq-refl vanishes-def zero-less-abs-iff*)

lemma *vanishes-minus*: *vanishes X* $\Longrightarrow$ *vanishes* ($\lambda n.\ -\ X\ n$)
unfolding *vanishes-def* **by** *simp*

lemma *vanishes-add*:
assumes *X*: *vanishes X*
and *Y*: *vanishes Y*
shows *vanishes* ($\lambda n.\ X\ n\ +\ Y\ n$)
proof (*rule vanishesI*)
fix *r* :: *rat*
assume *0* < *r*
then obtain *s t* **where** *s*: *0* < *s* **and** *t*: *0* < *t* **and** *r*: *r* = *s* + *t*
by (*rule obtain-pos-sum*)
obtain *i* **where** *i*: $\forall\, n{\geq}i.\ |X\ n| < s$
using *vanishesD* [*OF X s*] **..**
obtain *j* **where** *j*: $\forall\, n{\geq}j.\ |Y\ n| < t$
using *vanishesD* [*OF Y t*] **..**
have $\forall\, n{\geq}max\ i\ j.\ |X\ n\ +\ Y\ n| < r$
proof *clarsimp*
fix *n*
assume *n*: *i* ≤ *n j* ≤ *n*
have $|X\ n\ +\ Y\ n| \leq |X\ n| + |Y\ n|$
by (*rule abs-triangle-ineq*)
also have ... < *s* + *t*
by (*simp add*: *add-strict-mono i j n*)
finally show $|X\ n\ +\ Y\ n| < r$
by (*simp only*: *r*)
qed
then show $\exists\, k.\ \forall\, n{\geq}k.\ |X\ n\ +\ Y\ n| < r$ **..**
qed

lemma *vanishes-diff*:
assumes *vanishes X vanishes Y*
shows *vanishes* ($\lambda n.\ X\ n\ -\ Y\ n$)
unfolding *diff-conv-add-uminus* **by** (*intro vanishes-add vanishes-minus assms*)

lemma *vanishes-mult-bounded*:
assumes *X*: $\exists\, a{>}0.\ \forall\, n.\ |X\ n| < a$
assumes *Y*: *vanishes* ($\lambda n.\ Y\ n$)
shows *vanishes* ($\lambda n.\ X\ n\ *\ Y\ n$)
proof (*rule vanishesI*)
fix *r* :: *rat*
assume *r*: *0* < *r*
obtain *a* **where** *a*: *0* < *a* $\forall\, n.\ |X\ n| < a$
using *X* **by** *blast*
then obtain *b* **where** *b*: *0* < *b r* = *a* * *b*
by (*metis divide-pos-pos less-numeral-extra*(*3*) *nonzero-mult-div-cancel-left r*
times-divide-eq-right)
obtain *k* **where** *k*: $\forall\, n{\geq}k.\ |Y\ n| < b$
using *vanishesD* [*OF Y b*(*1*)] **..**

have $\forall\, n{\geq}k.\ |X\ n * Y\ n| < r$
by (*simp add*: *b*(*2*) *abs-mult mult-strict-mono'* *a k*)
then show $\exists\, k.\ \forall\, n{\geq}k.\ |X\ n * Y\ n| < r$ **..**
qed

97.3 Cauchy sequences

definition *cauchy* :: $(nat \Rightarrow rat) \Rightarrow bool$
where $cauchy\ X \longleftrightarrow (\forall\, r{>}0.\ \exists\, k.\ \forall\, m{\geq}k.\ \forall\, n{\geq}k.\ |X\ m - X\ n| < r)$

lemma *cauchyI*: $(\bigwedge r.\ 0 < r \Longrightarrow \exists\, k.\ \forall\, m{\geq}k.\ \forall\, n{\geq}k.\ |X\ m - X\ n| < r) \Longrightarrow cauchy\ X$
unfolding *cauchy-def* **by** *simp*

lemma *cauchyD*: $cauchy\ X \Longrightarrow 0 < r \Longrightarrow \exists\, k.\ \forall\, m{\geq}k.\ \forall\, n{\geq}k.\ |X\ m - X\ n| < r$
unfolding *cauchy-def* **by** *simp*

lemma *cauchy-const* [*simp*]: $cauchy\ (\lambda n.\ x)$
unfolding *cauchy-def* **by** *simp*

lemma *cauchy-add* [*simp*]:
assumes *X*: *cauchy X* **and** *Y*: *cauchy Y*
shows $cauchy\ (\lambda n.\ X\ n + Y\ n)$
proof (*rule cauchyI*)
fix *r* :: *rat*
assume $0 < r$
then obtain *s t* **where** *s*: $0 < s$ **and** *t*: $0 < t$ **and** *r*: $r = s + t$
by (*rule obtain-pos-sum*)
obtain *i* **where** *i*: $\forall\, m{\geq}i.\ \forall\, n{\geq}i.\ |X\ m - X\ n| < s$
using *cauchyD* [*OF X s*] **..**
obtain *j* **where** *j*: $\forall\, m{\geq}j.\ \forall\, n{\geq}j.\ |Y\ m - Y\ n| < t$
using *cauchyD* [*OF Y t*] **..**
have $\forall\, m{\geq}max\ i\ j.\ \forall\, n{\geq}max\ i\ j.\ |(X\ m + Y\ m) - (X\ n + Y\ n)| < r$
proof *clarsimp*
fix *m n*
assume $*$: $i \leq m\ j \leq m\ i \leq n\ j \leq n$
have $|(X\ m + Y\ m) - (X\ n + Y\ n)| \leq |X\ m - X\ n| + |Y\ m - Y\ n|$
unfolding *add-diff-add* **by** (*rule abs-triangle-ineq*)
also have $\ldots < s + t$
by (*rule add-strict-mono*) (*simp-all add*: *i j* $*$)
finally show $|(X\ m + Y\ m) - (X\ n + Y\ n)| < r$ **by** (*simp only*: *r*)
qed
then show $\exists\, k.\ \forall\, m{\geq}k.\ \forall\, n{\geq}k.\ |(X\ m + Y\ m) - (X\ n + Y\ n)| < r$ **..**
qed

lemma *cauchy-minus* [*simp*]:
assumes *X*: *cauchy X*
shows $cauchy\ (\lambda n.\ -\ X\ n)$
using *assms* **unfolding** *cauchy-def*

unfolding *minus-diff-minus abs-minus-cancel* **.**

lemma *cauchy-diff* [*simp*]:
assumes *cauchy X cauchy Y*
shows *cauchy* (λ*n*. *X n* − *Y n*)
using *assms* **unfolding** *diff-conv-add-uminus* **by** (*simp del*: *add-uminus-conv-diff*)

lemma *cauchy-imp-bounded*:
assumes *cauchy X*
shows ∃ *b*>*0*. ∀ *n*. |*X n*| < *b*
proof −
obtain *k* **where** *k*: ∀ *m*≥*k*. ∀ *n*≥*k*. |*X m* − *X n*| < *1*
using *cauchyD* [*OF assms zero-less-one*] **..**
show ∃ *b*>*0*. ∀ *n*. |*X n*| < *b*
proof (*intro exI conjI allI*)
have *0* ≤ |*X 0*| **by** *simp*
also have |*X 0*| ≤ *Max* (*abs* ‘ *X* ‘ {*..k*}) **by** *simp*
finally have *0* ≤ *Max* (*abs* ‘ *X* ‘ {*..k*}) **.**
then show *0* < *Max* (*abs* ‘ *X* ‘ {*..k*}) + *1* **by** *simp*
next
fix *n* :: *nat*
show |*X n*| < *Max* (*abs* ‘ *X* ‘ {*..k*}) + *1*
proof (*rule linorder-le-cases*)
assume *n* ≤ *k*
then have |*X n*| ≤ *Max* (*abs* ‘ *X* ‘ {*..k*}) **by** *simp*
then show |*X n*| < *Max* (*abs* ‘ *X* ‘ {*..k*}) + *1* **by** *simp*
next
assume *k* ≤ *n*
have |*X n*| = |*X k* + (*X n* − *X k*)| **by** *simp*
also have |*X k* + (*X n* − *X k*)| ≤ |*X k*| + |*X n* − *X k*|
by (*rule abs-triangle-ineq*)
also have ... < *Max* (*abs* ‘ *X* ‘ {*..k*}) + *1*
by (*rule add-le-less-mono*) (*simp-all add*: *k* ‹*k* ≤ *n*›)
finally show |*X n*| < *Max* (*abs* ‘ *X* ‘ {*..k*}) + *1* **.**
qed
qed
qed

lemma *cauchy-mult* [*simp*]:
assumes *X*: *cauchy X* **and** *Y*: *cauchy Y*
shows *cauchy* (λ*n*. *X n* ∗ *Y n*)
proof (*rule cauchyI*)
fix *r* :: *rat* **assume** *0* < *r*
then obtain *u v* **where** *u*: *0* < *u* **and** *v*: *0* < *v* **and** *r* = *u* + *v*
by (*rule obtain-pos-sum*)
obtain *a* **where** *a*: *0* < *a* ∀ *n*. |*X n*| < *a*
using *cauchy-imp-bounded* [*OF X*] **by** *blast*
obtain *b* **where** *b*: *0* < *b* ∀ *n*. |*Y n*| < *b*
using *cauchy-imp-bounded* [*OF Y*] **by** *blast*

obtain s t **where** s: $0 < s$ **and** t: $0 < t$ **and** r: $r = a * t + s * b$
proof
show $0 < v/b$ **using** v $b(1)$ **by** *simp*
show $0 < u/a$ **using** u $a(1)$ **by** *simp*
show $r = a * (u/a) + (v/b) * b$
using $a(1)$ $b(1)$ ‹$r = u + v$› **by** *simp*
qed
obtain i **where** i: $\forall m{\geq}i.\ \forall n{\geq}i.\ |X\ m - X\ n| < s$
using *cauchyD* [*OF* X s] **..**
obtain j **where** j: $\forall m{\geq}j.\ \forall n{\geq}j.\ |Y\ m - Y\ n| < t$
using *cauchyD* [*OF* Y t] **..**
have $\forall m{\geq}max\ i\ j.\ \forall n{\geq}max\ i\ j.\ |X\ m * Y\ m - X\ n * Y\ n| < r$
proof *clarsimp*
fix m n
assume $*$: $i \leq m\ j \leq m\ i \leq n\ j \leq n$
have $|X\ m * Y\ m - X\ n * Y\ n| = |X\ m * (Y\ m - Y\ n) + (X\ m - X\ n) * Y\ n|$
unfolding *mult-diff-mult* **..**
also have $\ldots \leq |X\ m * (Y\ m - Y\ n)| + |(X\ m - X\ n) * Y\ n|$
by (*rule abs-triangle-ineq*)
also have $\ldots = |X\ m| * |Y\ m - Y\ n| + |X\ m - X\ n| * |Y\ n|$
unfolding *abs-mult* **..**
also have $\ldots < a * t + s * b$
by (*simp-all add*: *add-strict-mono mult-strict-mono'* a b i j $*$)
finally show $|X\ m * Y\ m - X\ n * Y\ n| < r$
by (*simp only*: r)
qed
then show $\exists k.\ \forall m{\geq}k.\ \forall n{\geq}k.\ |X\ m * Y\ m - X\ n * Y\ n| < r$ **..**
qed

lemma *cauchy-not-vanishes-cases*:
assumes X: *cauchy* X
assumes nz: $\neg$ *vanishes* X
shows $\exists b{>}0.\ \exists k.\ (\forall n{\geq}k.\ b < -\ X\ n) \vee (\forall n{\geq}k.\ b < X\ n)$
proof −
obtain r **where** $0 < r$ **and** r: $\forall k.\ \exists n{\geq}k.\ r \leq |X\ n|$
using nz **unfolding** *vanishes-def* **by** (*auto simp add*: *not-less*)
obtain s t **where** s: $0 < s$ **and** t: $0 < t$ **and** $r = s + t$
using ‹$0 < r$› **by** (*rule obtain-pos-sum*)
obtain i **where** i: $\forall m{\geq}i.\ \forall n{\geq}i.\ |X\ m - X\ n| < s$
using *cauchyD* [*OF* X s] **..**
obtain k **where** $i \leq k$ **and** $r \leq |X\ k|$
using r **by** *blast*
have k: $\forall n{\geq}k.\ |X\ n - X\ k| < s$
using i ‹$i \leq k$› **by** *auto*
have $X\ k \leq -\ r \vee r \leq X\ k$
using ‹$r \leq |X\ k|$› **by** *auto*
then have $(\forall n{\geq}k.\ t < -\ X\ n) \vee (\forall n{\geq}k.\ t < X\ n)$
unfolding ‹$r = s + t$› **using** k **by** *auto*

```
  then have ∃k. (∀n≥k. t < - X n) ∨ (∀n≥k. t < X n) ..
  then show ∃t>0. ∃k. (∀n≥k. t < - X n) ∨ (∀n≥k. t < X n)
    using t by auto
qed

lemma cauchy-not-vanishes:
  assumes X: cauchy X
    and nz: ¬ vanishes X
  shows ∃b>0. ∃k. ∀n≥k. b < |X n|
  using cauchy-not-vanishes-cases [OF assms]
  by (elim ex-forward conj-forward asm-rl) auto

lemma cauchy-inverse [simp]:
  assumes X: cauchy X
    and nz: ¬ vanishes X
  shows cauchy (λn. inverse (X n))
proof (rule cauchyI)
  fix r :: rat
  assume 0 < r
  obtain b i where b: 0 < b and i: ∀n≥i. b < |X n|
    using cauchy-not-vanishes [OF X nz] by blast
  from b i have nz: ∀n≥i. X n ≠ 0 by auto
  obtain s where s: 0 < s and r: r = inverse b * s * inverse b
  proof
    show 0 < b * r * b by (simp add: ‹0 < r› b)
  qed (use b in simp)
  obtain j where j: ∀m≥j. ∀n≥j. |X m - X n| < s
    using cauchyD [OF X s] ..
  have ∀m≥max i j. ∀n≥max i j. |inverse (X m) - inverse (X n)| < r
  proof clarsimp
    fix m n
    assume *: i ≤ m j ≤ m i ≤ n j ≤ n
    have |inverse (X m) - inverse (X n)| = inverse |X m| * |X m - X n| * inverse
|X n|
      by (simp add: inverse-diff-inverse nz * abs-mult)
    also have ... < inverse b * s * inverse b
      by (simp add: mult-strict-mono less-imp-inverse-less i j b * s)
    finally show |inverse (X m) - inverse (X n)| < r by (simp only: r)
  qed
  then show ∃k. ∀m≥k. ∀n≥k. |inverse (X m) - inverse (X n)| < r ..
qed

lemma vanishes-diff-inverse:
  assumes X: cauchy X ¬ vanishes X
    and Y: cauchy Y ¬ vanishes Y
    and XY: vanishes (λn. X n - Y n)
  shows vanishes (λn. inverse (X n) - inverse (Y n))
proof (rule vanishesI)
  fix r :: rat
```

assume r: $0 < r$
obtain a i **where** a: $0 < a$ **and** i: $\forall n{\geq}i.\ a < |X\ n|$
using *cauchy-not-vanishes* [*OF X*] **by** *blast*
obtain b j **where** b: $0 < b$ **and** j: $\forall n{\geq}j.\ b < |Y\ n|$
using *cauchy-not-vanishes* [*OF Y*] **by** *blast*
obtain s **where** s: $0 < s$ **and** *inverse a* $*$ s $*$ *inverse b* $= r$
proof
show $0 < a * r * b$
using a r b **by** *simp*
qed (*use a r b* **in** *simp*)
obtain k **where** k: $\forall n{\geq}k.\ |X\ n - Y\ n| < s$
using *vanishesD* [*OF XY s*] **..**
have $\forall n{\geq}max\ (max\ i\ j)\ k.\ |inverse\ (X\ n) - inverse\ (Y\ n)| < r$
proof *clarsimp*
fix n
assume n: $i \leq n\ j \leq n\ k \leq n$
with i j a b **have** $X\ n \neq 0$ **and** $Y\ n \neq 0$
by *auto*
then have $|inverse\ (X\ n) - inverse\ (Y\ n)| = inverse\ |X\ n| * |X\ n - Y\ n| * inverse\ |Y\ n|$
by (*simp add*: *inverse-diff-inverse abs-mult*)
also have $\ldots < inverse\ a * s * inverse\ b$
by (*intro mult-strict-mono' less-imp-inverse-less*) (*simp-all add*: a b i j k n)
also note ‹*inverse a* $*$ s $*$ *inverse b* $= r$›
finally show $|inverse\ (X\ n) - inverse\ (Y\ n)| < r$ **.**
qed
then show $\exists k.\ \forall n{\geq}k.\ |inverse\ (X\ n) - inverse\ (Y\ n)| < r$ **..**
qed

97.4 Equivalence relation on Cauchy sequences

definition *realrel* :: $(nat \Rightarrow rat) \Rightarrow (nat \Rightarrow rat) \Rightarrow bool$
where *realrel* $= (\lambda X\ Y.\ cauchy\ X \wedge cauchy\ Y \wedge vanishes\ (\lambda n.\ X\ n - Y\ n))$

lemma *realrelI* [*intro?*]: *cauchy X* $\Longrightarrow$ *cauchy Y* $\Longrightarrow$ *vanishes* $(\lambda n.\ X\ n - Y\ n)$ $\Longrightarrow$ *realrel X Y*
by (*simp add*: *realrel-def*)

lemma *realrel-refl*: *cauchy X* $\Longrightarrow$ *realrel X X*
by (*simp add*: *realrel-def*)

lemma *symp-realrel*: *symp realrel*
by (*simp add*: *abs-minus-commute realrel-def symp-def vanishes-def*)

lemma *transp-realrel*: *transp realrel*
unfolding *realrel-def*
by (*rule transpI*) (*force simp add*: *dest*: *vanishes-add*)

lemma *part-equivp-realrel*: *part-equivp realrel*

by (*blast intro*: *part-equivpI symp-realrel transp-realrel realrel-refl cauchy-const*)

97.5 The field of real numbers

quotient-type *real* = *nat* ⇒ *rat* / *partial*: *realrel*
morphisms *rep-real Real*
by (*rule part-equivp-realrel*)

lemma *cr-real-eq*: *pcr-real* = (λ*x y*. *cauchy x* ∧ *Real x* = *y*)
unfolding *real.pcr-cr-eq cr-real-def realrel-def* **by** *auto*

lemma *Real-induct* [*induct type*: *real*]:
assumes ⋀*X*. *cauchy X* ⟹ *P* (*Real X*)
shows *P x*
proof (*induct x*)
case (*1 X*)
then have *cauchy X* **by** (*simp add*: *realrel-def*)
then show *P* (*Real X*) **by** (*rule assms*)
qed

lemma *eq-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* = *Real Y* ⟷ *vanishes* (λ*n*. *X n* − *Y n*)
using *real.rel-eq-transfer*
unfolding *real.pcr-cr-eq cr-real-def rel-fun-def realrel-def* **by** *simp*

lemma *Domainp-pcr-real* [*transfer-domain-rule*]: *Domainp pcr-real* = *cauchy*
by (*simp add*: *real.domain-eq realrel-def*)

instantiation *real* :: *field*
begin

lift-definition *zero-real* :: *real* **is** λ*n*. *0*
by (*simp add*: *realrel-refl*)

lift-definition *one-real* :: *real* **is** λ*n*. *1*
by (*simp add*: *realrel-refl*)

lift-definition *plus-real* :: *real* ⇒ *real* ⇒ *real* **is** λ*X Y n*. *X n* + *Y n*
unfolding *realrel-def add-diff-add*
by (*simp only*: *cauchy-add vanishes-add simp-thms*)

lift-definition *uminus-real* :: *real* ⇒ *real* **is** λ*X n*. − *X n*
unfolding *realrel-def minus-diff-minus*
by (*simp only*: *cauchy-minus vanishes-minus simp-thms*)

lift-definition *times-real* :: *real* ⇒ *real* ⇒ *real* **is** λ*X Y n*. *X n* ∗ *Y n*
proof −
fix *f1 f2 f3 f4*
have ⟦*cauchy f1*; *cauchy f4*; *vanishes* (λ*n*. *f1 n* − *f2 n*); *vanishes* (λ*n*. *f3 n* − *f4*

n)⟧
⟹ *vanishes* (λ*n*. *f1 n* ∗ (*f3 n* − *f4 n*) + *f4 n* ∗ (*f1 n* − *f2 n*))
by (*simp add*: *vanishes-add vanishes-mult-bounded cauchy-imp-bounded*)
then show ⟦*realrel f1 f2*; *realrel f3 f4*⟧ ⟹ *realrel* (λ*n*. *f1 n* ∗ *f3 n*) (λ*n*. *f2 n* ∗ *f4 n*)
by (*simp add*: *mult.commute realrel-def mult-diff-mult*)
qed

lift-definition *inverse-real* :: *real* ⇒ *real*
is λ*X*. *if vanishes X then* (λ*n*. *0*) *else* (λ*n*. *inverse* (*X n*))
proof −
fix *X Y*
assume *realrel X Y*
then have *X*: *cauchy X* **and** *Y*: *cauchy Y* **and** *XY*: *vanishes* (λ*n*. *X n* − *Y n*)
by (*simp-all add*: *realrel-def*)
have *vanishes X* ⟷ *vanishes Y*
proof
assume *vanishes X*
from *vanishes-diff* [*OF this XY*] **show** *vanishes Y*
by *simp*
next
assume *vanishes Y*
from *vanishes-add* [*OF this XY*] **show** *vanishes X*
by *simp*
qed
then show *?thesis X Y*
by (*simp add*: *vanishes-diff-inverse X Y XY realrel-def*)
qed

definition *x* − *y* = *x* + − *y* **for** *x y* :: *real*

definition *x div y* = *x* ∗ *inverse y* **for** *x y* :: *real*

lemma *add-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* + *Real Y* = *Real* (λ*n*. *X n* + *Y n*)
using *plus-real.transfer* **by** (*simp add*: *cr-real-eq rel-fun-def*)

lemma *minus-Real*: *cauchy X* ⟹ − *Real X* = *Real* (λ*n*. − *X n*)
using *uminus-real.transfer* **by** (*simp add*: *cr-real-eq rel-fun-def*)

lemma *diff-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* − *Real Y* = *Real* (λ*n*. *X n* − *Y n*)
by (*simp add*: *minus-Real add-Real minus-real-def*)

lemma *mult-Real*: *cauchy X* ⟹ *cauchy Y* ⟹ *Real X* ∗ *Real Y* = *Real* (λ*n*. *X n* ∗ *Y n*)
using *times-real.transfer* **by** (*simp add*: *cr-real-eq rel-fun-def*)

lemma *inverse-Real*:

cauchy X $\Longrightarrow$ *inverse* (*Real X*) = (*if vanishes X then 0 else Real* (λ*n*. *inverse* (*X n*)))
using *inverse-real.transfer zero-real.transfer*
unfolding *cr-real-eq rel-fun-def* **by** (*simp split*: *if-split-asm*, *metis*)

instance
proof
fix *a b c* :: *real*
show *a* + *b* = *b* + *a*
by *transfer* (*simp add*: *ac-simps realrel-def*)
show (*a* + *b*) + *c* = *a* + (*b* + *c*)
by *transfer* (*simp add*: *ac-simps realrel-def*)
show *0* + *a* = *a*
by *transfer* (*simp add*: *realrel-def*)
show − *a* + *a* = *0*
by *transfer* (*simp add*: *realrel-def*)
show *a* − *b* = *a* + − *b*
by (*rule minus-real-def*)
show (*a* ∗ *b*) ∗ *c* = *a* ∗ (*b* ∗ *c*)
by *transfer* (*simp add*: *ac-simps realrel-def*)
show *a* ∗ *b* = *b* ∗ *a*
by *transfer* (*simp add*: *ac-simps realrel-def*)
show *1* ∗ *a* = *a*
by *transfer* (*simp add*: *ac-simps realrel-def*)
show (*a* + *b*) ∗ *c* = *a* ∗ *c* + *b* ∗ *c*
by *transfer* (*simp add*: *distrib-right realrel-def*)
show (*0*::*real*) ≠ (*1*::*real*)
by *transfer* (*simp add*: *realrel-def*)
have *vanishes* (λ*n*. *inverse* (*X n*) ∗ *X n* − *1*) **if** *X*: *cauchy X* ¬ *vanishes X* **for** *X*
proof (*rule vanishesI*)
fix *r*::*rat*
assume *0* < *r*
obtain *b k* **where** *b*>*0* ∀ *n*≥*k*. *b* < |*X n*|
using *X cauchy-not-vanishes* **by** *blast*
then show ∃ *k*. ∀ *n*≥*k*. |*inverse* (*X n*) ∗ *X n* − *1*| < *r*
using ‹*0* < *r*› **by** *force*
qed
then show *a* ≠ *0* $\Longrightarrow$ *inverse a* ∗ *a* = *1*
by *transfer* (*simp add*: *realrel-def*)
show *a div b* = *a* ∗ *inverse b*
by (*rule divide-real-def*)
show *inverse* (*0*::*real*) = *0*
by *transfer* (*simp add*: *realrel-def*)
qed

end

97.6 Positive reals

lift-definition *positive* :: *real* ⇒ *bool*
is λ*X*. ∃ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *r* < *X n*
proof −
have *1*: ∃ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *r* < *Y n*
if ∗: *realrel X Y* **and** ∗∗: ∃ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *r* < *X n* **for** *X Y*
proof −
from ∗ **have** *XY*: *vanishes* (λ*n*. *X n* − *Y n*)
by (*simp-all add*: *realrel-def*)
from ∗∗ **obtain** *r i* **where** *0* < *r* **and** *i*: ∀ *n*≥*i*. *r* < *X n*
by *blast*
obtain *s t* **where** *s*: *0* < *s* **and** *t*: *0* < *t* **and** *r*: *r* = *s* + *t*
using ‹*0* < *r*› **by** (*rule obtain-pos-sum*)
obtain *j* **where** *j*: ∀ *n*≥*j*. |*X n* − *Y n*| < *s*
using *vanishesD* [*OF XY s*] **..**
have ∀ *n*≥*max i j*. *t* < *Y n*
using *i j r* **by** *fastforce*
then show *?thesis* **using** *t* **by** *blast*
qed
fix *X Y* **assume** *realrel X Y*
then have *realrel X Y* **and** *realrel Y X*
using *symp-realrel* **by** (*auto simp*: *symp-def*)
then show *?thesis X Y*
by (*safe elim*!: *1*)
qed

lemma *positive-Real*: *cauchy X* ⟹ *positive* (*Real X*) ⟷ (∃ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *r* < *X n*)
using *positive.transfer* **by** (*simp add*: *cr-real-eq rel-fun-def*)

lemma *positive-zero*: ¬ *positive 0*
by *transfer auto*

lemma *positive-add*:
assumes *positive x positive y* **shows** *positive* (*x* + *y*)
proof −
have ∗: ⟦∀ *n*≥*i*. *a* < *x n*; ∀ *n*≥*j*. *b* < *y n*; *0* < *a*; *0* < *b*; *n* ≥ *max i j*⟧
⟹ *a*+*b* < *x n* + *y n* **for** *x y* **and** *a b*::*rat* **and** *i j n*::*nat*
by (*simp add*: *add-strict-mono*)
show *?thesis*
using *assms*
by *transfer* (*blast intro*: ∗ *pos-add-strict*)
qed

lemma *positive-mult*:
assumes *positive x positive y* **shows** *positive* (*x* ∗ *y*)
proof −
have ∗: ⟦∀ *n*≥*i*. *a* < *x n*; ∀ *n*≥*j*. *b* < *y n*; *0* < *a*; *0* < *b*; *n* ≥ *max i j*⟧
⟹ *a*∗*b* < *x n* ∗ *y n* **for** *x y* **and** *a b*::*rat* **and** *i j n*::*nat*

```
    by (simp add: mult-strict-mono')
  show ?thesis
    using assms
    by transfer (blast intro: * mult-pos-pos)
qed

lemma positive-minus: ¬ positive x ⟹ x ≠ 0 ⟹ positive (− x)
  apply transfer
  apply (simp add: realrel-def)
  apply (blast dest: cauchy-not-vanishes-cases)
  done

instantiation real :: linordered-field
begin

definition x < y ⟷ positive (y − x)

definition x ≤ y ⟷ x < y ∨ x = y for x y :: real

definition |a| = (if a < 0 then − a else a) for a :: real

definition sgn a = (if a = 0 then 0 else if 0 < a then 1 else − 1) for a :: real

instance
proof
  fix a b c :: real
  show |a| = (if a < 0 then − a else a)
    by (rule abs-real-def)
  show a < b ⟷ a ≤ b ∧ ¬ b ≤ a
      a ≤ b ⟹ b ≤ c ⟹ a ≤ c  a ≤ a
      a ≤ b ⟹ b ≤ a ⟹ a = b
      a ≤ b ⟹ c + a ≤ c + b
    unfolding less-eq-real-def less-real-def
    by (force simp add: positive-zero dest: positive-add)+
  show sgn a = (if a = 0 then 0 else if 0 < a then 1 else − 1)
    by (rule sgn-real-def)
  show a ≤ b ∨ b ≤ a
    by (auto dest!: positive-minus simp: less-eq-real-def less-real-def)
  show a < b ⟹ 0 < c ⟹ c * a < c * b
    unfolding less-real-def
    by (force simp add: algebra-simps dest: positive-mult)
qed

end

instantiation real :: distrib-lattice
begin

definition (inf :: real ⇒ real ⇒ real) = min
```

definition (*sup* :: *real* ⇒ *real* ⇒ *real*) = *max*

instance
by *standard* (*auto simp add*: *inf-real-def sup-real-def max-min-distrib2*)

end

lemma *of-nat-Real*: *of-nat x* = *Real* (λ*n*. *of-nat x*)
by (*induct x*) (*simp-all add*: *zero-real-def one-real-def add-Real*)

lemma *of-int-Real*: *of-int x* = *Real* (λ*n*. *of-int x*)
by (*cases x rule*: *int-diff-cases*) (*simp add*: *of-nat-Real diff-Real*)

lemma *of-rat-Real*: *of-rat x* = *Real* (λ*n*. *x*)
proof (*induct x*)
case (*Fract a b*)
then show *?case*
apply (*simp add*: *Fract-of-int-quotient of-rat-divide*)
apply (*simp add*: *of-int-Real divide-inverse inverse-Real mult-Real*)
done
qed

instance *real* :: *archimedean-field*
proof
show ∃*z*. *x* ≤ *of-int z* **for** *x* :: *real*
proof (*induct x*)
case (*1 X*)
then obtain *b* **where** *0* < *b* **and** *b*: ⋀*n*. |*X n*| < *b*
by (*blast dest*: *cauchy-imp-bounded*)
then have *Real X* < *of-int* (⌈*b*⌉ + *1*)
using *1*
apply (*simp add*: *of-int-Real less-real-def diff-Real positive-Real*)
apply (*rule-tac x=1* **in** *exI*)
apply (*simp add*: *algebra-simps*)
by (*metis abs-ge-self le-less-trans le-of-int-ceiling less-le*)
then show *?case*
using *less-eq-real-def* **by** *blast*
qed
qed

instantiation *real* :: *floor-ceiling*
begin

definition [*code drop*]: ⌊*x*::*real*⌋ = (*THE z*. *of-int z* ≤ *x* ∧ *x* < *of-int* (*z* + *1*))

instance
proof
show *of-int* ⌊*x*⌋ ≤ *x* ∧ *x* < *of-int* (⌊*x*⌋ + *1*) **for** *x* :: *real*

unfolding *floor-real-def* **using** *floor-exists1* **by** (*rule theI'*)
qed

end

97.7 Completeness

lemma *not-positive-Real*:
assumes *cauchy X* **shows** ¬ *positive* (*Real X*) ⟷ (∀ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *X n* ≤ *r*) (**is** *?lhs* = *?rhs*)
unfolding *positive-Real* [*OF assms*]
proof (*intro iffI allI notI impI*)
show ∃ *k*. ∀ *n*≥*k*. *X n* ≤ *r* **if** *r*: ¬ (∃ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *r* < *X n*) **and** *0* < *r* **for** *r*
proof −
obtain *s t* **where** *s* > *0* *t* > *0* *r* = *s*+*t*
using ‹*r* > *0*› *obtain-pos-sum* **by** *blast*
obtain *k* **where** *k*: ⋀*m n*. ⟦*m*≥*k*; *n*≥*k*⟧ ⟹ |*X m* − *X n*| < *t*
using *cauchyD* [*OF assms* ‹*t* > *0*›] **by** *blast*
obtain *n* **where** *n* ≥ *k* *X n* ≤ *s*
by (*meson r* ‹*0* < *s*› *not-less*)
then have *X l* ≤ *r* **if** *l* ≥ *n* **for** *l*
using *k* [*OF* ‹*n* ≥ *k*›, *of l*] *that* ‹*r* = *s*+*t*› **by** *linarith*
then show *?thesis*
by *blast*
qed
qed (*meson le-cases not-le*)

lemma *le-Real*:
assumes *cauchy X cauchy Y*
shows *Real X* ≤ *Real Y* = (∀ *r*>*0*. ∃ *k*. ∀ *n*≥*k*. *X n* ≤ *Y n* + *r*)
unfolding *not-less* [*symmetric*, **where** *'a*=*real*] *less-real-def*
apply (*simp add*: *diff-Real not-positive-Real assms*)
apply (*simp add*: *diff-le-eq ac-simps*)
done

lemma *le-RealI*:
assumes *Y*: *cauchy Y*
shows ∀ *n*. *x* ≤ *of-rat* (*Y n*) ⟹ *x* ≤ *Real Y*
proof (*induct x*)
fix *X*
assume *X*: *cauchy X* **and** ∀ *n*. *Real X* ≤ *of-rat* (*Y n*)
then have *le*: ⋀*m r*. *0* < *r* ⟹ ∃ *k*. ∀ *n*≥*k*. *X n* ≤ *Y m* + *r*
by (*simp add*: *of-rat-Real le-Real*)
then have ∃ *k*. ∀ *n*≥*k*. *X n* ≤ *Y n* + *r* **if** *0* < *r* **for** *r* :: *rat*
proof −
from *that* **obtain** *s t* **where** *s*: *0* < *s* **and** *t*: *0* < *t* **and** *r*: *r* = *s* + *t*
by (*rule obtain-pos-sum*)
obtain *i* **where** *i*: ∀ *m*≥*i*. ∀ *n*≥*i*. |*Y m* − *Y n*| < *s*
using *cauchyD* [*OF Y s*] **..**

```
    obtain j where j: ∀n≥j. X n ≤ Y i + t
      using le [OF t] ..
    have ∀n≥max i j. X n ≤ Y n + r
    proof clarsimp
      fix n
      assume n: i ≤ n j ≤ n
      have X n ≤ Y i + t
        using n j by simp
      moreover have |Y i − Y n| < s
        using n i by simp
      ultimately show X n ≤ Y n + r
        unfolding r by simp
    qed
    then show ?thesis ..
  qed
  then show Real X ≤ Real Y
    by (simp add: of-rat-Real le-Real X Y)
qed

lemma Real-leI:
  assumes X: cauchy X
  assumes le: ∀n. of-rat (X n) ≤ y
  shows Real X ≤ y
proof −
  have − y ≤ − Real X
    by (simp add: minus-Real X le-RealI of-rat-minus le)
  then show ?thesis by simp
qed

lemma less-RealD:
  assumes cauchy Y
  shows x < Real Y ⟹ ∃n. x < of-rat (Y n)
  by (meson Real-leI assms leD leI)

lemma of-nat-less-two-power [simp]: of-nat n < (2::'a::linordered-idom) ^ n
  by auto

lemma complete-real:
  fixes S :: real set
  assumes ∃x. x ∈ S and ∃z. ∀x∈S. x ≤ z
  shows ∃y. (∀x∈S. x ≤ y) ∧ (∀z. (∀x∈S. x ≤ z) ⟶ y ≤ z)
proof −
  obtain x where x: x ∈ S using assms(1) ..
  obtain z where z: ∀x∈S. x ≤ z using assms(2) ..

  define P where P x ⟷ (∀y∈S. y ≤ of-rat x) for x
  obtain a where a: ¬ P a
  proof
    have of-int ⌊x − 1⌋ ≤ x − 1 by (rule of-int-floor-le)
```

```
    also have x − 1 < x by simp
    finally have of-int ⌊x − 1⌋ < x .
    then have ¬ x ≤ of-int ⌊x − 1⌋ by (simp only: not-le)
    then show ¬ P (of-int ⌊x − 1⌋)
      unfolding P-def of-rat-of-int-eq using x by blast
  qed
  obtain b where b: P b
  proof
    show P (of-int ⌈z⌉)
      unfolding P-def of-rat-of-int-eq
      using ceiling-le-iff ceiling-mono z by blast
  qed

  define avg where avg x y = x/2 + y/2 for x y :: rat
  define bisect where bisect = (λ(x, y). if P (avg x y) then (x, avg x y) else (avg
x y, y))
  define A where A n = fst ((bisect ^^ n) (a, b)) for n
  define B where B n = snd ((bisect ^^ n) (a, b)) for n
  define C where C n = avg (A n) (B n) for n
  have A-0 [simp]: A 0 = a unfolding A-def by simp
  have B-0 [simp]: B 0 = b unfolding B-def by simp
  have A-Suc [simp]: ⋀n. A (Suc n) = (if P (C n) then A n else C n)
    unfolding A-def B-def C-def bisect-def split-def by simp
  have B-Suc [simp]: ⋀n. B (Suc n) = (if P (C n) then C n else B n)
    unfolding A-def B-def C-def bisect-def split-def by simp

  have width: B n − A n = (b − a) / 2^n for n
  proof (induct n)
    case (Suc n)
    then show ?case
      by (simp add: C-def eq-divide-eq avg-def algebra-simps)
  qed simp
  have twos: ∃ n. y / 2 ^ n < r if 0 < r for y r :: rat
  proof −
    obtain n where y / r < rat-of-nat n
      using ‹0 < r› reals-Archimedean2 by blast
    then have ∃ n. y < r * 2 ^ n
      by (metis divide-less-eq less-trans mult.commute of-nat-less-two-power that)
    then show ?thesis
      by (simp add: field-split-simps)
  qed
  have PA: ¬ P (A n) for n
    by (induct n) (simp-all add: a)
  have PB: P (B n) for n
    by (induct n) (simp-all add: b)
  have ab: a < b
    using a b unfolding P-def
    by (meson leI less-le-trans of-rat-less)
  have AB: A n < B n for n
```

```
    by (induct n) (simp-all add: ab C-def avg-def)
  have  A i ≤ A j ∧  B j ≤ B i if i < j for i j
    using that
  proof (induction rule: less-Suc-induct)
    case (1 i)
    then show ?case
      apply (clarsimp simp add: C-def avg-def add-divide-distrib [symmetric])
      apply (rule AB [THEN less-imp-le])
      done
  qed simp
  then have A-mono: A i ≤ A j and B-mono: B j ≤ B i if i ≤ j for i j
    by (metis eq-refl le-neq-implies-less that)+
  have cauchy-lemma: cauchy X if ∗: ⋀n i. i≥n ⟹ A n ≤ X i ∧ X i ≤ B n for
X
  proof (rule cauchyI)
    fix r::rat
    assume 0 < r
    then obtain k where k: (b − a) / 2 ^ k < r
      using twos by blast
    have |X m − X n| < r if m≥k n≥k for m n
    proof −
      have |X m − X n| ≤ B k − A k
        by (simp add: ∗ abs-rat-def diff-mono that)
      also have ... < r
        by (simp add: k width)
      finally show ?thesis .
    qed
    then show ∃ k. ∀ m≥k. ∀ n≥k. |X m − X n| < r
      by blast
  qed
  have cauchy A
   by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order
less-le-trans)
  have cauchy B
   by (rule cauchy-lemma) (meson AB A-mono B-mono dual-order.strict-implies-order
le-less-trans)
  have ∀ x∈S. x ≤ Real B
  proof
    fix x
    assume x ∈ S
    then show x ≤ Real B
      using PB P-def ‹cauchy B› le-RealI by force
  qed
  moreover have ∀ z. (∀ x∈S. x ≤ z) ⟶ Real A ≤ z
    by (meson PA Real-leI P-def ‹cauchy A› le-cases order.trans)
  moreover have vanishes (λn. (b − a) / 2 ^ n)
  proof (rule vanishesI)
    fix r :: rat
    assume 0 < r
```

```
    then obtain k where k: |b − a| / 2 ^ k < r
      using twos by blast
    have ∀ n≥k. |(b − a) / 2 ^ n| < r
    proof clarify
      fix n
      assume n: k ≤ n
      have |(b − a) / 2 ^ n| = |b − a| / 2 ^ n
        by simp
      also have ... ≤ |b − a| / 2 ^ k
        using n by (simp add: divide-left-mono)
      also note k
      finally show |(b − a) / 2 ^ n| < r .
    qed
    then show ∃ k. ∀ n≥k. |(b − a) / 2 ^ n| < r ..
  qed
  then have Real B = Real A
    by (simp add: eq-Real ‹cauchy A› ‹cauchy B› width)
  ultimately show ∃ y. (∀ x∈S. x ≤ y) ∧ (∀ z. (∀ x∈S. x ≤ z) ⟶ y ≤ z)
    by force
qed

instantiation real :: linear-continuum
begin
```

97.8 Supremum of a set of reals

```
definition Sup X = (LEAST z::real. ∀ x∈X. x ≤ z)
definition Inf X = − Sup (uminus ‘ X) for X :: real set

instance
proof
  show Sup-upper: x ≤ Sup X
    if x ∈ X bdd-above X
    for x :: real and X :: real set
  proof −
    from that obtain s where s: ∀ y∈X. y ≤ s ⋀z. ∀ y∈X. y ≤ z ⟹ s ≤ z
      using complete-real[of X] unfolding bdd-above-def by blast
    then show ?thesis
      unfolding Sup-real-def by (rule LeastI2-order) (auto simp: that)
  qed
  show Sup-least: Sup X ≤ z
    if X ≠ {} and z: ⋀x. x ∈ X ⟹ x ≤ z
    for z :: real and X :: real set
  proof −
    from that obtain s where s: ∀ y∈X. y ≤ s ⋀z. ∀ y∈X. y ≤ z ⟹ s ≤ z
      using complete-real [of X] by blast
    then have Sup X = s
      unfolding Sup-real-def by (best intro: Least-equality)
    also from s z have ... ≤ z
```

by *blast*
finally show *?thesis* **.**
qed
show $Inf\ X \leq x$ **if** $x \in X$ *bdd-below X*
for x :: *real* **and** X :: *real set*
using *Sup-upper* [*of* $-x$ *uminus* ' X] **by** (*auto simp*: *Inf-real-def that*)
show $z \leq Inf\ X$ **if** $X \neq \{\}$ $\bigwedge x.\ x \in X \Longrightarrow z \leq x$
for z :: *real* **and** X :: *real set*
using *Sup-least* [*of uminus* ' $X - z$] **by** (*force simp*: *Inf-real-def that*)
show $\exists\, a\ b$::*real*. $a \neq b$
using *zero-neq-one* **by** *blast*
qed

end

97.9 Hiding implementation details

hide-const (**open**) *vanishes cauchy positive Real*

declare *Real-induct* [*induct del*]
declare *Abs-real-induct* [*induct del*]
declare *Abs-real-cases* [*cases del*]

lifting-update *real.lifting*
lifting-forget *real.lifting*

97.10 Embedding numbers into the Reals

abbreviation *real-of-nat* :: *nat* ⇒ *real*
where *real-of-nat* ≡ *of-nat*

abbreviation *real* :: *nat* ⇒ *real*
where *real* ≡ *of-nat*

abbreviation *real-of-int* :: *int* ⇒ *real*
where *real-of-int* ≡ *of-int*

abbreviation *real-of-rat* :: *rat* ⇒ *real*
where *real-of-rat* ≡ *of-rat*

declare [[*coercion-enabled*]]

declare [[*coercion of-nat* :: *nat* ⇒ *int*]]
declare [[*coercion of-nat* :: *nat* ⇒ *real*]]
declare [[*coercion of-int* :: *int* ⇒ *real*]]

declare [[*coercion-map map*]]
declare [[*coercion-map* λ*f g h x. g* (*h* (*f x*))]]

declare [[*coercion-map* $\lambda f\ g\ (x,y).\ (f\ x,\ g\ y)$]]

declare *of-int-eq-0-iff* [*algebra*, *presburger*]
declare *of-int-eq-1-iff* [*algebra*, *presburger*]
declare *of-int-eq-iff* [*algebra*, *presburger*]
declare *of-int-less-0-iff* [*algebra*, *presburger*]
declare *of-int-less-1-iff* [*algebra*, *presburger*]
declare *of-int-less-iff* [*algebra*, *presburger*]
declare *of-int-le-0-iff* [*algebra*, *presburger*]
declare *of-int-le-1-iff* [*algebra*, *presburger*]
declare *of-int-le-iff* [*algebra*, *presburger*]
declare *of-int-0-less-iff* [*algebra*, *presburger*]
declare *of-int-0-le-iff* [*algebra*, *presburger*]
declare *of-int-1-less-iff* [*algebra*, *presburger*]
declare *of-int-1-le-iff* [*algebra*, *presburger*]

lemma *int-less-real-le*: $n < m \longleftrightarrow$ *real-of-int* $n + 1 \leq$ *real-of-int* m
  by (*metis floor-of-int less-floor-iff*)

lemma *int-le-real-less*: $n \leq m \longleftrightarrow$ *real-of-int* $n <$ *real-of-int* $m + 1$
  by (*meson int-less-real-le not-le*)

lemma (**in** *field-char-0*) *of-int-div-aux*:
  (*of-int* x) / (*of-int* d) =
    of-int (x *div* d) + (*of-int* (x *mod* d)) / (*of-int* d)
proof −
  have $x = (x\ div\ d) * d + x\ mod\ d$
    by *auto*
  then have *of-int* x = *of-int* (x *div* d) $*$ *of-int* d + *of-int*(x *mod* d)
    by (*metis local.of-int-add local.of-int-mult*)
  then show *?thesis*
    by (*simp add*: *divide-simps*)
qed

lemma *real-of-int-div*:
  d *dvd* $n \Longrightarrow$ *real-of-int* (n *div* d) = *real-of-int* n / *real-of-int* d **for** $d\ n$ :: *int*
  by *auto*

lemma *real-of-int-div2*: $0 \leq$ *real-of-int* n / *real-of-int* x − *real-of-int* (n *div* x)
proof (*cases* $x = 0$)
  case *False*
  then show *?thesis*
    by (*metis diff-ge-0-iff-ge floor-divide-of-int-eq of-int-floor-le*)
qed *simp*

lemma *real-of-int-div3*: *real-of-int* n / *real-of-int* x − *real-of-int* (n *div* x) ≤ 1
  by (*metis add.commute diff-le-eq floor-divide-of-int-eq less-floor-iff linorder-not-le nle-le*)

lemma *real-of-int-div4*: *real-of-int* (*n div x*) ≤ *real-of-int n* / *real-of-int x*
using *real-of-int-div2* [*of n x*] **by** *simp*

97.11 Embedding the Naturals into the Reals

lemma (**in** *field-char-0*) *of-nat-of-nat-div-aux*:
of-nat x / *of-nat d* = *of-nat* (*x div d*) + *of-nat* (*x mod d*) / *of-nat d*
by (*metis add-divide-distrib diff-add-cancel of-nat-div*)

lemma(**in** *field-char-0*) *of-nat-of-nat-div*: *d dvd n* ⟹ *of-nat*(*n div d*) = *of-nat n* / *of-nat d*
by *auto*

lemma (**in** *linordered-field*) *of-nat-div-le-of-nat*: *of-nat* (*n div x*) ≤ *of-nat n* / *of-nat x*
by (*metis le-add-same-cancel1 of-nat-0-le-iff of-nat-of-nat-div-aux zero-le-divide-iff*)

lemma *real-of-card*: *real* (*card A*) = *sum* (λ*x*. *1*) *A*
by *simp*

lemma *nat-less-real-le*: *n* < *m* ⟷ *real n* + *1* ≤ *real m*
by (*metis less-iff-succ-less-eq of-nat-1 of-nat-add of-nat-le-iff*)

lemma *nat-le-real-less*: *n* ≤ *m* ⟷ *real n* < *real m* + *1*
by (*meson nat-less-real-le not-le*)

lemma *real-of-nat-div*: *d dvd n* ⟹ *real*(*n div d*) = *real n* / *real d*
by *auto*

lemma *real-binomial-eq-mult-binomial-Suc*:
assumes *k* ≤ *n*
shows *real*(*n choose k*) = (*n* + *1* − *k*) / (*n* + *1*) ∗ (*Suc n choose k*)
using *assms*
by (*simp add*: *of-nat-binomial-eq-mult-binomial-Suc* [*of k n*] *add.commute*)

97.12 The Archimedean Property of the Reals

Not actually the reals any more!

lemma *real-arch-inverse*:
fixes *e*::'*a*::*archimedean-field*
shows *0* < *e* ⟷ (∃ *n*::*nat*. *n* ≠ *0* ∧ *0* < *inverse* (*real n*) ∧ *inverse* (*of-nat n*) < *e*)
using *reals-Archimedean*[*of e*] *less-trans*[*of 0 1* / *of-nat n e* **for** *n*::*nat*]
by (*auto simp add*: *field-simps cong*: *conj-cong simp del*: *of-nat-Suc*)

lemma *reals-Archimedean3*:
fixes *x*::'*a*::*archimedean-field*
shows *0* < *x* ⟹ ∀ *y*. ∃ *n*. *y* < *of-nat n* ∗ *x*

by (*auto intro*: *ex-less-of-nat-mult*)

lemma *real-archimedian-rdiv-eq-0*:
fixes *x*::'*a*::*archimedean-field*
assumes $x \geq 0$ **and** $\bigwedge m$::*nat*. $m > 0 \Longrightarrow$ *of-nat* $m * x \leq c$
shows $x = 0$
by (*metis* (*no-types*, *opaque-lifting*) *reals-Archimedean3 order.order-iff-strict le0 le-less-trans not-le assms*)

lemma *inverse-Suc*: *inverse* (*of-nat* (*Suc n*)) > (*0*::'*a*::*archimedean-field*)
using *of-nat-0-less-iff positive-imp-inverse-positive zero-less-Suc* **by** *blast*

lemma *Archimedean-eventually-inverse*:
fixes ε::'*a*::*archimedean-field* **shows** ($\forall_F$ *n in sequentially*. *inverse* (*of-nat* (*Suc n*)) < ε) $\longleftrightarrow$ *0* < ε
(**is** *?lhs*=*?rhs*)
proof
assume *?lhs*
then show *?rhs*
unfolding *eventually-at-top-dense*
by (*metis* (*no-types*, *lifting*) *gt-ex inverse-Suc nat.distinct*(*1*) *real-arch-inverse*)
next
assume *?rhs*
then obtain *N* **where** *inverse* (*of-nat* (*Suc N*)) < ε
using *reals-Archimedean* **by** *blast*
then have *inverse* (*of-nat* (*Suc n*)) < ε **if** $n \geq N$ **for** *n*
using *that Suc-le-mono inverse-Suc inverse-less-imp-less*
by (*meson inverse-positive-iff-positive linorder-not-less of-nat-less-iff order-le-less-trans*)
then show *?lhs*
unfolding *eventually-sequentially* **by** *blast*
qed

On the relationship between two different ways of converting to 0

lemma *Inter-eq-Inter-inverse-Suc*:
assumes $\bigwedge r'\ r.\ r' < r \Longrightarrow A\ r' \subseteq A\ r$
shows $\bigcap\ (A\ `\ \{0<..\}) = (\bigcap n.\ A(inverse(Suc\ n)))$
proof
have $x \in A\ \varepsilon$
if $\forall n.\ x \in A\ (inverse\ (Suc\ n))$ **and** $\varepsilon > 0$ **for** x **and** ε :: *real*
by (*meson assms in-mono reals-Archimedean that*)
then show $(\bigcap n.\ A(inverse(Suc\ n))) \subseteq (\bigcap \varepsilon \in \{0<..\}.\ A\ \varepsilon)$
by *auto*
qed (*use inverse-Suc* **in** *fastforce*)

97.13 Rationals

lemma *Rats-abs-iff*[*simp*]:
$|(x::real)| \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
by(*simp add*: *abs-real-def split*: *if-splits*)

lemma *Rats-eq-int-div-int*: $\mathbb{Q} = \{real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j \mid i\ j.\ j \neq 0\}$ (**is** - = *?S*)
proof
  show $\mathbb{Q} \subseteq$ *?S*
  proof
    fix *x* :: *real*
    assume $x \in \mathbb{Q}$
    then obtain *r* **where** $x = of\text{-}rat\ r$
      unfolding *Rats-def* **..**
    have $of\text{-}rat\ r \in$ *?S*
      by (*cases r*) (*auto simp add*: *of-rat-rat*)
    then show $x \in$ *?S*
      using ‹$x = of\text{-}rat\ r$› **by** *simp*
  qed
next
  show *?S* $\subseteq \mathbb{Q}$
  proof (*auto simp*: *Rats-def*)
    fix *i j* :: *int*
    assume $j \neq 0$
    then have $real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j = of\text{-}rat\ (Fract\ i\ j)$
      by (*simp add*: *of-rat-rat*)
    then show $real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j \in range\ of\text{-}rat$
      by *blast*
  qed
qed

lemma *Rats-eq-int-div-nat*: $\mathbb{Q} = \{\ real\text{-}of\text{-}int\ i\ /\ real\ n \mid i\ n.\ n \neq 0\}$
proof (*auto simp*: *Rats-eq-int-div-int*)
  fix *i j* :: *int*
  assume $j \neq 0$
  show $\exists (i'::int)\ (n::nat).\ real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j = real\text{-}of\text{-}int\ i'\ /\ real\ n \wedge 0 < n$
  proof (*cases* $j > 0$)
    case *True*
    then have $real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j = real\text{-}of\text{-}int\ i\ /\ real\ (nat\ j) \wedge 0 < nat\ j$
      by *simp*
    then show *?thesis* **by** *blast*
  next
    case *False*
    with ‹$j \neq 0$›
    have $real\text{-}of\text{-}int\ i\ /\ real\text{-}of\text{-}int\ j = real\text{-}of\text{-}int\ (-\ i)\ /\ real\ (nat\ (-\ j)) \wedge 0 < nat\ (-\ j)$
      by *simp*
    then show *?thesis* **by** *blast*
  qed
next
  fix *i* :: *int* **and** *n* :: *nat*
  assume $0 < n$

then have *real-of-int i / real n = real-of-int i / real-of-int*(*int n*) ∧ *int n ≠ 0*
by *simp*
then show ∃ *i' j. real-of-int i / real n = real-of-int i' / real-of-int j* ∧ *j ≠ 0*
by *blast*
qed

lemma *Rats-abs-nat-div-natE*:
assumes *x* ∈ ℚ
obtains *m n* :: *nat* **where** *n ≠ 0* **and** |*x*| = *real m / real n* **and** *coprime m n*
proof −
from ‹*x* ∈ ℚ› **obtain** *i* :: *int* **and** *n* :: *nat* **where** *n ≠ 0* **and** *x = real-of-int i / real n*
by (*auto simp add*: *Rats-eq-int-div-nat*)
then have |*x*| = *real* (*nat* |*i*|) / *real n* **by** *simp*
then obtain *m* :: *nat* **where** *x-rat*: |*x*| = *real m / real n* **by** *blast*
let *?gcd = gcd m n*
from ‹*n ≠ 0*› **have** *gcd*: *?gcd ≠ 0* **by** *simp*
let *?k = m div ?gcd*
let *?l = n div ?gcd*
let *?gcd' = gcd ?k ?l*
have *?gcd dvd m* **..**
then have *gcd-k*: *?gcd* ∗ *?k = m*
by (*rule dvd-mult-div-cancel*)
have *?gcd dvd n* **..**
then have *gcd-l*: *?gcd* ∗ *?l = n*
by (*rule dvd-mult-div-cancel*)
from ‹*n ≠ 0*› **and** *gcd-l* **have** *?gcd* ∗ *?l ≠ 0* **by** *simp*
then have *?l ≠ 0* **by** (*blast dest*!: *mult-not-zero*)
moreover
have |*x*| = *real ?k / real ?l*
proof −
from *gcd* **have** *real ?k / real ?l = real* (*?gcd* ∗ *?k*) / *real* (*?gcd* ∗ *?l*)
by (*simp add*: *real-of-nat-div*)
also from *gcd-k* **and** *gcd-l* **have** ... = *real m / real n* **by** *simp*
also from *x-rat* **have** ... = |*x*| **..**
finally show *?thesis* **..**
qed
moreover
have *?gcd' = 1*
using ‹*n ≠ 0*› *coprime-iff-gcd-eq-1 div-gcd-coprime* **by** *blast*
then have *coprime ?k ?l*
by (*simp only*: *coprime-iff-gcd-eq-1*)
ultimately show *?thesis* **..**
qed

97.14 Density of the Rational Reals in the Reals

This density proof is due to Stefan Richter and was ported by TN. The original source is *Real Analysis* by H.L. Royden. It employs the Archimedean

property of the reals.

lemma *Rats-dense-in-real*:
fixes x :: *real*
assumes $x < y$
shows $\exists\, r{\in}\mathbb{Q}.\ x < r \wedge r < y$
proof −
from ‹$x < y$› **have** $0 < y - x$ **by** *simp*
with *reals-Archimedean* **obtain** q :: *nat* **where** q: *inverse* (*real* q) $< y - x$ **and** $0 < q$
by *blast*
define p **where** $p = \lceil y * \mathit{real}\ q \rceil - 1$
define r **where** $r =$ *of-int* p / *real* q
from q **have** $x < y -$ *inverse* (*real* q)
by *simp*
also from ‹$0 < q$› **have** $y -$ *inverse* (*real* q) $\leq r$
by (*simp add*: *r-def p-def le-divide-eq left-diff-distrib*)
finally have $x < r$ **.**
moreover from ‹$0 < q$› **have** $r < y$
by (*simp add*: *r-def p-def divide-less-eq diff-less-eq less-ceiling-iff* [*symmetric*])
moreover have $r \in \mathbb{Q}$
by (*simp add*: *r-def*)
ultimately show *?thesis* **by** *blast*
qed

lemma *of-rat-dense*:
fixes $x\ y$ :: *real*
assumes $x < y$
shows $\exists\, q$ :: *rat*. $x <$ *of-rat* $q \wedge$ *of-rat* $q < y$
using *Rats-dense-in-real* [*OF* ‹$x < y$›]
by (*auto elim*: *Rats-cases*)

lemma *le-iff-forall-rat-less-imp*:
fixes $x\ y$ :: *real*
shows $x \leq y \longleftrightarrow (\forall\, q \in \mathbb{Q}.\ y < q \longrightarrow x < q)$
by (*meson Rats-dense-in-real less-asym less-le-trans not-less*)

97.15 Numerals and Arithmetic

declaration ‹
K (*Lin-Arith.add-inj-const* (***const-name*** ‹*of-nat*›, ***typ*** ‹*nat* ⇒ *real*›)
#> *Lin-Arith.add-inj-const* (***const-name*** ‹*of-int*›, ***typ*** ‹*int* ⇒ *real*›))
›

97.16 Simprules combining $x + y$ and 0

lemma *real-add-minus-iff* [*simp*]: $x + - a = 0 \longleftrightarrow x = a$
for $x\ a$ :: *real*
by *arith*

lemma *real-add-less-0-iff*: $x + y < 0 \longleftrightarrow y < -x$
for $x\ y$ :: *real*
by *auto*

lemma *real-0-less-add-iff*: $0 < x + y \longleftrightarrow -x < y$
for $x\ y$ :: *real*
by *auto*

lemma *real-add-le-0-iff*: $x + y \leq 0 \longleftrightarrow y \leq -x$
for $x\ y$ :: *real*
by *auto*

lemma *real-0-le-add-iff*: $0 \leq x + y \longleftrightarrow -x \leq y$
for $x\ y$ :: *real*
by *auto*

lemma *mult-ge1-I*: $[\![x \geq 1;\ y \geq 1]\!] \Longrightarrow x * y \geq (1::real)$
using *mult-mono* **by** *fastforce*

97.17 Lemmas about powers

lemma *two-realpow-ge-one*: $(1::real) \leq 2 \hat{} n$
by *simp*

declare *sum-squares-eq-zero-iff* [*simp*] *sum-power2-eq-zero-iff* [*simp*]

lemma *real-minus-mult-self-le* [*simp*]: $-(u * u) \leq x * x$
for $u\ x$ :: *real*
by (*rule order-trans* [**where** $y = 0$]) *auto*

lemma *realpow-square-minus-le* [*simp*]: $-u^2 \leq x^2$
for $u\ x$ :: *real*
by (*auto simp add*: *power2-eq-square*)

The geometric sum formula for natural numbers

lemma
fixes $a\ k$ :: *nat*
assumes $a > 1$
shows *geometric-sum-nat-aux*: $(a - 1) * (\sum i{<}k.\ a \hat{} i) = a \hat{} k - 1$
and *geometric-sum-nat-dvd*: $a - 1$ *dvd* $a \hat{} k - 1$
and *geometric-sum-nat*: $(\sum i{<}k.\ a \hat{} i) = (a \hat{} k - 1)\ div\ (a - 1)$
proof −
have $(real\ a - 1) * (\sum i{<}k.\ real\ a \hat{} i) = real\ a \hat{} k - 1$
by (*simp add*: *power-diff-1-eq*)
also have $(real\ a - 1) * (\sum i{<}k.\ real\ a \hat{} i) = real\ ((a - 1) * (\sum i{<}k.\ a \hat{} i))$
using *assms* **by** *simp*
also have $real\ a \hat{} k - 1 = real\ (a \hat{} k - 1)$
using *assms* **by** *auto*
finally show $*$: $(a - 1) * (\sum i{<}k.\ a \hat{} i) = a \hat{} k - 1$

by (*subst* (*asm*) *of-nat-eq-iff*)
show *a − 1 dvd a ^ k − 1*
by (*metis ∗ dvd-triv-left*)
from *assms* **show** $(\sum i{<}k.\ a\ \hat{}\ i) = (a\ \hat{}\ k - 1)\ div\ (a - 1)$
by (*metis ∗ div-mult-self1-is-m zero-less-diff*)
qed

97.18 Density of the Reals

lemma *field-lbound-gt-zero*: $0 < d1 \Longrightarrow 0 < d2 \Longrightarrow \exists e.\ 0 < e \wedge e < d1 \wedge e < d2$
for *d1 d2* :: *'a*::*linordered-field*
by (*rule exI* [**where** *x* = *min d1 d2 / 2*]) (*simp add*: *min-def*)

lemma *field-less-half-sum*: $x < y \Longrightarrow x < (x + y) / 2$
for *x y* :: *'a*::*linordered-field*
by *auto*

lemma *field-sum-of-halves*: $x / 2 + x / 2 = x$
for *x* :: *'a*::*linordered-field*
by *simp*

97.19 Archimedean properties and useful consequences

Bernoulli's inequality

proposition *Bernoulli-inequality*:
fixes *x* :: *'a* :: *linordered-field*
assumes $-1 \leq x$
shows $1 + of\text{-}nat\ n * x \leq (1 + x)\ \hat{}\ n$
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have $1 + of\text{-}nat\ (Suc\ n) * x \leq 1 + of\text{-}nat(Suc\ n) * x + of\text{-}nat\ n * x\hat{}2$
by *simp*
also have ... = $(1 + x) * (1 + of\text{-}nat\ n * x)$
by (*auto simp*: *power2-eq-square algebra-simps*)
also have ... $\leq (1 + x)\ \hat{}\ Suc\ n$
using *Suc.hyps assms mult-left-mono* **by** *fastforce*
finally show *?case* **.**
qed

corollary *Bernoulli-inequality-even*:
fixes *x* :: *'a* :: *linordered-field*
assumes *even n*
shows $1 + of\text{-}nat\ n * x \leq (1 + x)\ \hat{}\ n$
proof (*cases* $-1 \leq x \vee n{=}0$)
case *True*

then show *?thesis*
by (*auto simp*: *Bernoulli-inequality*)
next
case *False*
then have *of-nat n* $\geq$ (*1*::*'a*)
by *simp*
with *False* **have** *of-nat n* $*$ *x* ≤ -1
by (*meson le-minus-one-simps*(*1*) *mult-le-cancel-right2 nle-le order.order-iff-strict order.trans*)
then have *1* + *of-nat n* $*$ *x* $\leq$ *0*
by *auto*
also have ... $\leq$ (*1* + *x*) ^ *n*
using *assms zero-le-even-power* **by** *blast*
finally show *?thesis* **.**
qed

corollary *arch-pow*:
fixes *x* :: *'a* :: *archimedean-field*
assumes *x*: *1* < *x*
shows $\exists$ *n*. *y* < *x*^*n*
proof −
from *x* **have** *x0*: *x* − *1* > *0*
by *arith*
from *reals-Archimedean3*[*OF x0*, *rule-format*, *of y*]
obtain *n* :: *nat* **where** *n*: *y* < *of-nat n* $*$ (*x* − *1*) **by** *metis*
from *x0* **have** *x00*: *x*− *1* ≥ -1 **by** *arith*
from *Bernoulli-inequality*[*OF x00*, *of n*] *n*
have *y* < *x*^*n* **by** *auto*
then show *?thesis* **by** *metis*
qed

corollary *arch-pow-inv*:
fixes *x y* :: *'a* :: *archimedean-field*
assumes *y*: *y* > *0* **and** *x1*: *x* < *1*
shows $\exists$ *n*. *x*^*n* < *y*
proof (*cases x* > *0*)
case *True*
with *x1* **have** *ix*: *1* < *1*/*x* **by** (*simp add*: *field-simps*)
from *arch-pow*[*OF ix*, *of 1*/*y*]
obtain *n* **where** *n*: *1*/*y* < (*1*/*x*)^*n* **by** *blast*
then show *?thesis* **using** *y* ‹*x* > *0*›
by (*auto simp add*: *field-simps*)
next
case *False*
with *y x1* **show** *?thesis*
by (*metis less-le-trans not-less power-one-right*)
qed

lemma *forall-pos-mono*:

$(\bigwedge d\ e :: 'a :: archimedean\text{-}field.\ d < e \Longrightarrow P\ d \Longrightarrow P\ e) \Longrightarrow$
$(\bigwedge n{::}nat.\ n \neq 0 \Longrightarrow P\ (inverse\ (of\text{-}nat\ n))) \Longrightarrow (\bigwedge e.\ 0 < e \Longrightarrow P\ e)$
by (*metis real-arch-inverse*)

lemma *forall-pos-mono-1*:
$(\bigwedge d\ e :: 'a :: archimedean\text{-}field.\ d < e \Longrightarrow P\ d \Longrightarrow P\ e) \Longrightarrow$
$(\bigwedge n.\ P\ (inverse\ (of\text{-}nat\ (Suc\ n)))) \Longrightarrow 0 < e \Longrightarrow P\ e$
using *reals-Archimedean* **by** *blast*

lemma *Archimedean-eventually-pow*:
fixes $x :: 'a :: archimedean\text{-}field$
assumes $1 < x$
shows $\forall_F\ n\ in\ sequentially.\ b < x\ \hat{}\ n$
proof −
obtain N **where** $\bigwedge n.\ n \geq N \Longrightarrow b < x\ \hat{}\ n$
by (*metis assms le-less order-less-trans power-strict-increasing-iff arch-pow*)
then show *?thesis*
using *eventually-sequentially* **by** *blast*
qed

lemma *Archimedean-eventually-pow-inverse*:
fixes $x :: 'a :: archimedean\text{-}field$
assumes $|x| < 1\ \varepsilon > 0$
shows $\forall_F\ n\ in\ sequentially.\ |x\hat{}n| < \varepsilon$
proof (*cases* $x = 0$)
case *True*
then show *?thesis*
by (*simp add*: *assms eventually-at-top-dense zero-power*)
next
case *False*
then have $\forall_F\ n\ in\ sequentially.\ inverse\ \varepsilon < inverse\ |x|\ \hat{}\ n$
by (*simp add*: *Archimedean-eventually-pow assms*(*1*) *one-less-inverse*)
then show *?thesis*
by *eventually-elim* (*metis* ‹$\varepsilon > 0$› *inverse-less-imp-less power-abs power-inverse*)
qed

lemma *power-tends-to-zero-sequentially*:
fixes $b :: 'a :: archimedean\text{-}field$
assumes $c > 0\ 0 < b\ b < 1$
shows $\forall_F\ n\ in\ sequentially.\ b\ \hat{}\ n < c$
proof −
have $b\ \hat{}\ n < b\ \hat{}\ m$ **if** $m < n$ **for** $m\ n$
by (*simp add*: *assms that*)
then show *?thesis*
using *arch-pow-inv* [*of c b*] *assms*
by (*metis* (*no-types*, *lifting*) *order.strict-trans eventually-at-top-dense*)
qed

97.20 Floor and Ceiling Functions from the Reals to the Integers

lemma *real-of-nat-less-numeral-iff* [*simp*]: *real n* < *numeral w* ⟷ *n* < *numeral w*
for *n* :: *nat*
by (*metis of-nat-less-iff of-nat-numeral*)

lemma *numeral-less-real-of-nat-iff* [*simp*]: *numeral w* < *real n* ⟷ *numeral w* < *n*
for *n* :: *nat*
by (*metis of-nat-less-iff of-nat-numeral*)

lemma *numeral-le-real-of-nat-iff* [*simp*]: *numeral n* ≤ *real m* ⟷ *numeral n* ≤ *m*
for *m* :: *nat*
by (*metis not-le real-of-nat-less-numeral-iff*)

lemma *of-int-floor-cancel* [*simp*]: *of-int* ⌊*x*⌋ = *x* ⟷ (∃ *n*::*int*. *x* = *of-int n*)
by (*metis floor-of-int*)

lemma *of-int-floor* [*simp*]: *a* ∈ ℤ ⟹ *of-int* (*floor a*) = *a*
by (*metis Ints-cases of-int-floor-cancel*)

lemma *floor-frac* [*simp*]: ⌊*frac r*⌋ = *0*
by (*simp add*: *frac-def*)

lemma *frac-1* [*simp*]: *frac 1* = *0*
by (*simp add*: *frac-def*)

lemma *frac-in-Rats-iff* [*simp*]:
fixes *r*::*'a*::{*floor-ceiling*,*field-char-0*}
shows *frac r* ∈ ℚ ⟷ *r* ∈ ℚ
by (*metis Rats-add Rats-diff Rats-of-int diff-add-cancel frac-def*)

lemma *floor-eq*: *real-of-int n* < *x* ⟹ *x* < *real-of-int n* + *1* ⟹ ⌊*x*⌋ = *n*
by *linarith*

lemma *floor-eq2*: *real-of-int n* ≤ *x* ⟹ *x* < *real-of-int n* + *1* ⟹ ⌊*x*⌋ = *n*
by (*fact floor-unique*)

lemma *floor-eq3*: *real n* < *x* ⟹ *x* < *real* (*Suc n*) ⟹ *nat* ⌊*x*⌋ = *n*
by *linarith*

lemma *floor-eq4*: *real n* ≤ *x* ⟹ *x* < *real* (*Suc n*) ⟹ *nat* ⌊*x*⌋ = *n*
by *linarith*

lemma *real-of-int-floor-ge-diff-one* [*simp*]: *r* − *1* ≤ *real-of-int* ⌊*r*⌋
by *linarith*

lemma *real-of-int-floor-gt-diff-one* [*simp*]: *r* − *1* < *real-of-int* ⌊*r*⌋

by *linarith*

lemma *real-of-int-floor-add-one-ge* [*simp*]: $r \leq$ *real-of-int* $\lfloor r \rfloor + 1$
by *linarith*

lemma *real-of-int-floor-add-one-gt* [*simp*]: $r <$ *real-of-int* $\lfloor r \rfloor + 1$
by *linarith*

lemma *floor-divide-real-eq-div*:
assumes $0 \leq b$
shows $\lfloor a$ / *real-of-int* $b \rfloor = \lfloor a \rfloor$ *div* b
proof (*cases* $b = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
with *assms* **have** *b*: $b > 0$ **by** *simp*
have $j = i$ *div* b
if *real-of-int* $i \leq a$ $a <$ *real-of-int* $i + 1$
real-of-int $j *$ *real-of-int* $b \leq a$ $a <$ *real-of-int* $b +$ *real-of-int* $j *$ *real-of-int* b
for i j :: *int*
proof −
from *that* **have** $i < b + j * b$
by (*metis le-less-trans of-int-add of-int-less-iff of-int-mult*)
moreover have $j * b < 1 + i$
proof −
have *real-of-int* $(j * b) <$ *real-of-int* $i + 1$
using ‹$a < 1 +$ *real-of-int* i› ‹*real-of-int* $j *$ *real-of-int* $b \leq a$› **by** *force*
then show $j * b < 1 + i$ **by** *linarith*
qed
ultimately have $(j - i$ *div* $b) * b \leq i$ *mod* b i *mod* $b < ((j - i$ *div* $b) + 1) * b$
by (*auto simp*: *field-simps*)
then have $(j - i$ *div* $b) * b < 1 * b$ $0 * b < ((j - i$ *div* $b) + 1) * b$
using *pos-mod-bound* [*OF b*, *of i*] *pos-mod-sign* [*OF b*, *of i*]
by *linarith*+
then show *?thesis* **using** *b* **unfolding** *mult-less-cancel-right* **by** *auto*
qed
with *b* **show** *?thesis* **by** (*auto split*: *floor-split simp*: *field-simps*)
qed

lemma *floor-one-divide-eq-div-numeral* [*simp*]:
$\lfloor 1$ / *numeral b*::*real* $\rfloor = 1$ *div numeral* b
by (*metis floor-divide-of-int-eq of-int-1 of-int-numeral*)

lemma *floor-minus-one-divide-eq-div-numeral* [*simp*]:
$\lfloor -$ (*1* / *numeral b*)::*real* $\rfloor = -1$ *div numeral* b
by (*metis* (*mono-tags*, *opaque-lifting*) *div-minus-right minus-divide-right*
floor-divide-of-int-eq of-int-neg-numeral of-int-1)

lemma *floor-divide-eq-div-numeral* [*simp*]:
  $\lfloor$*numeral a / numeral b*::*real*$\rfloor$ = *numeral a div numeral b*
by (*metis floor-divide-of-int-eq of-int-numeral*)

lemma *floor-minus-divide-eq-div-numeral* [*simp*]:
  $\lfloor -$ (*numeral a / numeral b*)::*real*$\rfloor$ = $-$ *numeral a div numeral b*
by (*metis divide-minus-left floor-divide-of-int-eq of-int-neg-numeral of-int-numeral*)

lemma *of-int-ceiling-cancel* [*simp*]: *of-int* $\lceil x \rceil$ = $x \longleftrightarrow$ ($\exists$ *n*::*int*. *x* = *of-int n*)
  using *ceiling-of-int* **by** *metis*

lemma *of-int-ceiling* [*simp*]: $a \in \mathbb{Z} \Longrightarrow$ *of-int* (*ceiling a*) = *a*
  by (*metis Ints-cases of-int-ceiling-cancel*)

lemma *ceiling-eq*: *of-int n* < *x* $\Longrightarrow$ *x* $\leq$ *of-int n* + *1* $\Longrightarrow$ $\lceil x \rceil$ = *n* + *1*
  by (*simp add*: *ceiling-unique*)

lemma *of-int-ceiling-diff-one-le* [*simp*]: *of-int* $\lceil r \rceil$ $-$ *1* $\leq$ *r*
  by *linarith*

lemma *of-int-ceiling-le-add-one* [*simp*]: *of-int* $\lceil r \rceil$ $\leq$ *r* + *1*
  by *linarith*

lemma *ceiling-le*: *x* $\leq$ *of-int a* $\Longrightarrow$ $\lceil x \rceil$ $\leq$ *a*
  by (*simp add*: *ceiling-le-iff*)

lemma *ceiling-divide-eq-div*: $\lceil$*of-int a / of-int b*$\rceil$ = $-$ ($-$ *a div b*)
  by (*metis ceiling-def floor-divide-of-int-eq minus-divide-left of-int-minus*)

lemma *ceiling-divide-eq-div-numeral* [*simp*]:
  $\lceil$*numeral a / numeral b* :: *real*$\rceil$ = $-$ ($-$ *numeral a div numeral b*)
  using *ceiling-divide-eq-div*[*of numeral a numeral b*] **by** *simp*

lemma *ceiling-minus-divide-eq-div-numeral* [*simp*]:
  $\lceil -$ (*numeral a / numeral b* :: *real*)$\rceil$ = $-$ (*numeral a div numeral b*)
  using *ceiling-divide-eq-div*[*of* $-$ *numeral a numeral b*] **by** *simp*

The following lemmas are remnants of the erstwhile functions natfloor and natceiling.

lemma *nat-floor-neg*: *x* $\leq$ *0* $\Longrightarrow$ *nat* $\lfloor x \rfloor$ = *0*
  for *x* :: *real*
  by *linarith*

lemma *le-nat-floor*: *real x* $\leq$ *a* $\Longrightarrow$ *x* $\leq$ *nat* $\lfloor a \rfloor$
  by *linarith*

lemma *le-mult-nat-floor*: *nat* $\lfloor a \rfloor$ * *nat* $\lfloor b \rfloor$ $\leq$ *nat* $\lfloor a * b \rfloor$
  by (*cases* 0 $\leq$ *a* $\wedge$ 0 $\leq$ *b*)
    (*auto simp add*: *nat-mult-distrib*[*symmetric*] *nat-mono le-mult-floor*)

lemma *nat-ceiling-le-eq* [*simp*]: $nat\ \lceil x \rceil \leq a \longleftrightarrow x \leq real\ a$
by *linarith*

lemma *real-nat-ceiling-ge*: $x \leq real\ (nat\ \lceil x \rceil)$
by *linarith*

lemma *Rats-no-top-le*: $\exists\, q \in \mathbb{Q}.\ x \leq q$
for $x :: real$
by (*auto intro*!: *bexI*[*of - of-nat* (*nat* $\lceil x \rceil$)]) *linarith*

lemma *Rats-no-bot-less*: $\exists\, q \in \mathbb{Q}.\ q < x$ **for** $x :: real$
by (*auto intro*!: *bexI*[*of - of-int* ($\lfloor x \rfloor - 1$)]) *linarith*

lemma *floor-ceiling-diff-le*: $0 \leq r \Longrightarrow nat\lfloor real\ k - r \rfloor \leq k - nat\lceil r \rceil$
by *linarith*

lemma *floor-ceiling-diff-le'*: $nat\lfloor r - real\ k \rfloor \leq nat\lceil r \rceil - k$
by *linarith*

lemma *ceiling-floor-diff-ge*: $nat\lceil r - real\ k \rceil \geq nat\lfloor r \rfloor - k$
by *linarith*

lemma *ceiling-floor-diff-ge'*: $r \leq k \Longrightarrow nat\lceil r - real\ k \rceil \leq k - nat\lfloor r \rfloor$
by *linarith*

97.21 Exponentiation with floor

lemma *floor-power*:
assumes $x = of\text{-}int\ \lfloor x \rfloor$
shows $\lfloor x \mathbin{\hat{}} n \rfloor = \lfloor x \rfloor \mathbin{\hat{}} n$
proof −
have $x \mathbin{\hat{}} n = of\text{-}int\ (\lfloor x \rfloor \mathbin{\hat{}} n)$
using *assms* **by** (*induct n arbitrary*: x) *simp-all*
then show *?thesis* **by** (*metis floor-of-int*)
qed

lemma *floor-numeral-power* [*simp*]: $\lfloor numeral\ x \mathbin{\hat{}} n \rfloor = numeral\ x \mathbin{\hat{}} n$
by (*metis floor-of-int of-int-numeral of-int-power*)

lemma *ceiling-numeral-power* [*simp*]: $\lceil numeral\ x \mathbin{\hat{}} n \rceil = numeral\ x \mathbin{\hat{}} n$
by (*metis ceiling-of-int of-int-numeral of-int-power*)

97.22 Implementation of rational real numbers

Formal constructor

definition *Ratreal* :: $rat \Rightarrow real$
where [*code-abbrev*, *simp*]: $Ratreal = real\text{-}of\text{-}rat$

code-datatype *Ratreal*

Quasi-Numerals

lemma [*code-abbrev*]:
real-of-rat (numeral k) = numeral k
real-of-rat (− numeral k) = − numeral k
real-of-rat (rat-of-int a) = real-of-int a
by *simp-all*

lemma [*code-post*]:
real-of-rat 0 = 0
real-of-rat 1 = 1
real-of-rat (− 1) = − 1
real-of-rat (1 / numeral k) = 1 / numeral k
real-of-rat (numeral k / numeral l) = numeral k / numeral l
real-of-rat (− (1 / numeral k)) = − (1 / numeral k)
real-of-rat (− (numeral k / numeral l)) = − (numeral k / numeral l)
by (*simp-all add*: *of-rat-divide of-rat-minus*)

Operations

lemma *zero-real-code* [*code*]: *0 = Ratreal 0*
by *simp*

lemma *one-real-code* [*code*]: *1 = Ratreal 1*
by *simp*

instantiation *real* :: *equal*
begin

definition *HOL.equal x y ⟷ x − y = 0* **for** *x* :: *real*

instance by *standard* (*simp add*: *equal-real-def*)

lemma *real-equal-code* [*code*]: *HOL.equal* (*Ratreal x*) (*Ratreal y*) ⟷ *HOL.equal x y*
by (*simp add*: *equal-real-def equal*)

lemma [*code nbe*]: *HOL.equal x x ⟷ True*
for *x* :: *real*
by (*rule equal-refl*)

end

lemma *real-less-eq-code* [*code*]: *Ratreal x ≤ Ratreal y ⟷ x ≤ y*
by (*simp add*: *of-rat-less-eq*)

lemma *real-less-code* [*code*]: *Ratreal x < Ratreal y ⟷ x < y*
by (*simp add*: *of-rat-less*)

lemma *real-plus-code* [*code*]: *Ratreal x + Ratreal y = Ratreal (x + y)*
by (*simp add*: *of-rat-add*)

lemma *real-times-code* [*code*]: *Ratreal x * Ratreal y = Ratreal (x * y)*
by (*simp add*: *of-rat-mult*)

lemma *real-uminus-code* [*code*]: *- Ratreal x = Ratreal (- x)*
by (*simp add*: *of-rat-minus*)

lemma *real-minus-code* [*code*]: *Ratreal x - Ratreal y = Ratreal (x - y)*
by (*simp add*: *of-rat-diff*)

lemma *real-inverse-code* [*code*]: *inverse (Ratreal x) = Ratreal (inverse x)*
by (*simp add*: *of-rat-inverse*)

lemma *real-divide-code* [*code*]: *Ratreal x / Ratreal y = Ratreal (x / y)*
by (*simp add*: *of-rat-divide*)

lemma *real-floor-code* [*code*]: *⌊Ratreal x⌋ = ⌊x⌋*
by (*metis Ratreal-def floor-le-iff floor-unique le-floor-iff of-int-floor-le of-rat-of-int-eq real-less-eq-code*)

Quickcheck

context
includes *term-syntax*
begin

definition
valterm-ratreal :: rat × (unit ⇒ Code-Evaluation.term) ⇒ real × (unit ⇒ Code-Evaluation.term)
where [*code-unfold*]: *valterm-ratreal k = Code-Evaluation.valtermify Ratreal {·} k*

end

instantiation *real :: random*
begin

context
includes *state-combinator-syntax*
begin

definition
Quickcheck-Random.random i = Quickcheck-Random.random i ∘→ (λr. Pair (valterm-ratreal r))

instance ..

end

end

instantiation *real* :: *exhaustive*
begin

definition
exhaustive-real f d = *Quickcheck-Exhaustive.exhaustive* (λ*r*. *f* (*Ratreal r*)) *d*

instance ..

end

instantiation *real* :: *full-exhaustive*
begin

definition
full-exhaustive-real f d = *Quickcheck-Exhaustive.full-exhaustive* (λ*r*. *f* (*valterm-ratreal r*)) *d*

instance ..

end

instantiation *real* :: *narrowing*
begin

definition
narrowing-real = *Quickcheck-Narrowing.apply* (*Quickcheck-Narrowing.cons Ratreal*) *narrowing*

instance ..

end

97.23 Setup for Nitpick

declaration ‹
Nitpick-HOL.register-frac-type ***type-name*** ‹*real*›
[(***const-name*** ‹*zero-real-inst.zero-real*›, ***const-name*** ‹*Nitpick.zero-frac*›),
(***const-name*** ‹*one-real-inst.one-real*›, ***const-name*** ‹*Nitpick.one-frac*›),
(***const-name*** ‹*plus-real-inst.plus-real*›, ***const-name*** ‹*Nitpick.plus-frac*›),
(***const-name*** ‹*times-real-inst.times-real*›, ***const-name*** ‹*Nitpick.times-frac*›),
(***const-name*** ‹*uminus-real-inst.uminus-real*›, ***const-name*** ‹*Nitpick.uminus-frac*›),
(***const-name*** ‹*inverse-real-inst.inverse-real*›, ***const-name*** ‹*Nitpick.inverse-frac*›),
(***const-name*** ‹*ord-real-inst.less-real*›, ***const-name*** ‹*Nitpick.less-frac*›),
(***const-name*** ‹*ord-real-inst.less-eq-real*›, ***const-name*** ‹*Nitpick.less-eq-frac*›)]
›

lemmas [*nitpick-unfold*] = *inverse-real-inst.inverse-real one-real-inst.one-real ord-real-inst.less-real ord-real-inst.less-eq-real plus-real-inst.plus-real times-real-inst.times-real uminus-real-inst.uminus-real zero-real-inst.zero-real*

97.24 Setup for SMT

ML-file ‹*Tools/SMT/smt-real.ML*›
ML-file ‹*Tools/SMT/z3/z3-real.ML*›

lemma [*z3-rule*]:
0 + x = x
x + 0 = x
*0 * x = 0*
*1 * x = x*
*−x = −1 * x*
x + y = y + x
for *x y* :: *real*
by *auto*

lemma [*smt-arith-multiplication*]:
fixes *A B* :: *real* **and** *p n* :: *real*
assumes *A ≤ B 0 < n p > 0*
shows *(A / n) * p ≤ (B / n) * p*
using *assms* **by** (*auto simp*: *field-simps*)

lemma [*smt-arith-multiplication*]:
fixes *A B* :: *real* **and** *p n* :: *real*
assumes *A < B 0 < n p > 0*
shows *(A / n) * p < (B / n) * p*
using *assms* **by** (*auto simp*: *field-simps*)

lemma [*smt-arith-multiplication*]:
fixes *A B* :: *real* **and** *p n* :: *int*
assumes *A ≤ B 0 < n p > 0*
shows *(A / n) * p ≤ (B / n) * p*
using *assms* **by** (*auto simp*: *field-simps*)

lemma [*smt-arith-multiplication*]:
fixes *A B* :: *real* **and** *p n* :: *int*
assumes *A < B 0 < n p > 0*
shows *(A / n) * p < (B / n) * p*
using *assms* **by** (*auto simp*: *field-simps*)

lemmas [*smt-arith-multiplication*] =
alethe-le-mono-div[*THEN mult-left-mono*, *unfolded int-distrib*, *of - -* ‹*nat* (*floor* (*- :: real*))› ‹*nat* (*floor* (*- :: real*))›]
div-le-mono[*THEN mult-left-mono*, *unfolded int-distrib*, *of - -* ‹*nat* (*floor* (*- :: real*))› ‹*nat* (*floor* (*- :: real*))›]

alethe-le-mono-div-int[*THEN mult-left-mono*, *unfolded int-distrib*, *of* - - ‹*floor* (- :: *real*)› ‹*floor* (- :: *real*)›]
zdiv-mono1[*THEN mult-left-mono*, *unfolded int-distrib*, *of* - - ‹*floor* (- :: *real*)› ‹*floor* (- :: *real*)›]
arg-cong[*of* - - ‹λa :: *real*. *a* / *real* (*n*::*nat*) * *real* (*p*::*nat*)› **for** *n p* :: *nat*]
arg-cong[*of* - - ‹λa :: *real*. *a* / *real-of-int n* * *real-of-int p*› **for** *n p* :: *int*]
arg-cong[*of* - - ‹λa :: *real*. *a* / *n* * *p*› **for** *n p* :: *real*]

lemmas [*smt-arith-simplify*,*arith-simp-cvc5*,*arith-mult-poly-norm-cvc5*] =
floor-one floor-numeral div-by-1 times-divide-eq-right
nonzero-mult-div-cancel-left division-ring-divide-zero div-0
divide-minus-left zero-less-divide-iff

lemma [*smt-arith-simplify*,*arith-simp-cvc5*,*arith-mult-poly-norm-cvc5*]:
‹*NO-MATCH 0* (*b* :: *real*) $\Longrightarrow$ *NO-MATCH 0* (*a* :: *real*) $\Longrightarrow$ $a < b \longleftrightarrow b - a > 0$›
‹*NO-MATCH 0 b* $\Longrightarrow$ *NO-MATCH 0 a* $\Longrightarrow$ $a \leq b \longleftrightarrow b - a \geq 0$›
‹*NO-MATCH 0 b* $\Longrightarrow$ *NO-MATCH 0 a* $\Longrightarrow$ $a = b \longleftrightarrow b - a = 0$›
by *auto*

lemma [*cvc5-normalized-input*]:
‹*a div 0* $\equiv$ *0*›
‹*a div 1* $\equiv$ *a*›
‹*0 div a* $\equiv$ *0*› **for** *a* :: ‹'*a* :: *divide-trivial*›
by *auto*

lemmas [*smt-arith-simplify*] = *diff-divide-distrib add-divide-distrib*

97.25 Setup for Argo

ML-file ‹*Tools/Argo/argo-real.ML*›

end

98 Topological Spaces

theory *Topological-Spaces*
imports *Main*
begin

named-theorems *continuous-intros structural introduction rules for continuity*

98.1 Topological space

class *open* =
fixes *open* :: '*a set* $\Rightarrow$ *bool*

context — Suppress difficult-to-implement class parameter
begin

private definition *unimplemented-open* :: ‹'*a*::*open set* ⇒ *bool*›
where [*code drop*]: ‹*unimplemented-open* = *open*›

lemma [*code*]:
‹*open* = *unimplemented-open*›
by (*simp add*: *unimplemented-open-def*)

end

class *topological-space* = *open* +
assumes *open-UNIV* [*simp*, *intro*]: *open UNIV*
assumes *open-Int* [*intro*]: *open S* ⟹ *open T* ⟹ *open* (*S* ∩ *T*)
assumes *open-Union* [*intro*]: ∀ *S*∈*K*. *open S* ⟹ *open* (⋃ *K*)
begin

definition *closed* :: '*a set* ⇒ *bool*
where *closed S* ⟷ *open* (− *S*)

lemma *open-empty* [*continuous-intros*, *intro*, *simp*]: *open* {}
using *open-Union* [*of* {}] **by** *simp*

lemma *open-Un* [*continuous-intros*, *intro*]: *open S* ⟹ *open T* ⟹ *open* (*S* ∪ *T*)
using *open-Union* [*of* {*S*, *T*}] **by** *simp*

lemma *open-UN* [*continuous-intros*, *intro*]: ∀ *x*∈*A*. *open* (*B x*) ⟹ *open* (⋃ *x*∈*A*. *B x*)
using *open-Union* [*of B* ‘ *A*] **by** *simp*

lemma *open-Inter* [*continuous-intros*, *intro*]: *finite S* ⟹ ∀ *T*∈*S*. *open T* ⟹ *open* (⋂ *S*)
by (*induction set*: *finite*) *auto*

lemma *open-INT* [*continuous-intros*, *intro*]: *finite A* ⟹ ∀ *x*∈*A*. *open* (*B x*) ⟹ *open* (⋂ *x*∈*A*. *B x*)
using *open-Inter* [*of B* ‘ *A*] **by** *simp*

lemma *openI*:
assumes ⋀*x*. *x* ∈ *S* ⟹ ∃ *T*. *open T* ∧ *x* ∈ *T* ∧ *T* ⊆ *S*
shows *open S*
proof −
have *open* (⋃ {*T*. *open T* ∧ *T* ⊆ *S*}) **by** *auto*
moreover have ⋃ {*T*. *open T* ∧ *T* ⊆ *S*} = *S* **by** (*auto dest*!: *assms*)
ultimately show *open S* **by** *simp*
qed

lemma *open-subopen*: *open S* ⟷ (∀ *x*∈*S*. ∃ *T*. *open T* ∧ *x* ∈ *T* ∧ *T* ⊆ *S*)
by (*auto intro*: *openI*)

lemma *closed-empty* [*continuous-intros*, *intro*, *simp*]: *closed* {}
unfolding *closed-def* **by** *simp*

lemma *closed-Un* [*continuous-intros*, *intro*]: *closed S* ⟹ *closed T* ⟹ *closed* (*S* ∪ *T*)
unfolding *closed-def* **by** *auto*

lemma *closed-UNIV* [*continuous-intros*, *intro*, *simp*]: *closed UNIV*
unfolding *closed-def* **by** *simp*

lemma *closed-Int* [*continuous-intros*, *intro*]: *closed S* ⟹ *closed T* ⟹ *closed* (*S* ∩ *T*)
unfolding *closed-def* **by** *auto*

lemma *closed-INT* [*continuous-intros*, *intro*]: ∀ *x*∈*A*. *closed* (*B x*) ⟹ *closed* (⋂ *x*∈*A*. *B x*)
unfolding *closed-def* **by** *auto*

lemma *closed-Inter* [*continuous-intros*, *intro*]: ∀ *S*∈*K*. *closed S* ⟹ *closed* (⋂ *K*)
unfolding *closed-def uminus-Inf* **by** *auto*

lemma *closed-Union* [*continuous-intros*, *intro*]: *finite S* ⟹ ∀ *T*∈*S*. *closed T* ⟹ *closed* (⋃ *S*)
by (*induct set*: *finite*) *auto*

lemma *closed-UN* [*continuous-intros*, *intro*]:
finite A ⟹ ∀ *x*∈*A*. *closed* (*B x*) ⟹ *closed* (⋃ *x*∈*A*. *B x*)
using *closed-Union* [*of B ' A*] **by** *simp*

lemma *open-closed*: *open S* ⟷ *closed* (− *S*)
by (*simp add*: *closed-def*)

lemma *closed-open*: *closed S* ⟷ *open* (− *S*)
by (*rule closed-def*)

lemma *open-Diff* [*continuous-intros*, *intro*]: *open S* ⟹ *closed T* ⟹ *open* (*S* − *T*)
by (*simp add*: *closed-open Diff-eq open-Int*)

lemma *closed-Diff* [*continuous-intros*, *intro*]: *closed S* ⟹ *open T* ⟹ *closed* (*S* − *T*)
by (*simp add*: *open-closed Diff-eq closed-Int*)

lemma *open-Compl* [*continuous-intros*, *intro*]: *closed S* ⟹ *open* (− *S*)
by (*simp add*: *closed-open*)

lemma *closed-Compl* [*continuous-intros*, *intro*]: *open S* ⟹ *closed* (− *S*)
by (*simp add*: *open-closed*)

lemma *open-Collect-neg*: *closed* $\{x.\ P\ x\} \Longrightarrow$ *open* $\{x.\ \neg\ P\ x\}$
unfolding *Collect-neg-eq* **by** (*rule open-Compl*)

lemma *open-Collect-conj*:
assumes *open* $\{x.\ P\ x\}$ *open* $\{x.\ Q\ x\}$
shows *open* $\{x.\ P\ x \wedge Q\ x\}$
using *open-Int*[*OF assms*] **by** (*simp add*: *Int-def*)

lemma *open-Collect-disj*:
assumes *open* $\{x.\ P\ x\}$ *open* $\{x.\ Q\ x\}$
shows *open* $\{x.\ P\ x \vee Q\ x\}$
using *open-Un*[*OF assms*] **by** (*simp add*: *Un-def*)

lemma *open-Collect-ex*: $(\bigwedge i.$ *open* $\{x.\ P\ i\ x\}) \Longrightarrow$ *open* $\{x.\ \exists\, i.\ P\ i\ x\}$
using *open-UN*[*of UNIV* $\lambda i.$ $\{x.\ P\ i\ x\}$] **unfolding** *Collect-ex-eq* **by** *simp*

lemma *open-Collect-imp*: *closed* $\{x.\ P\ x\} \Longrightarrow$ *open* $\{x.\ Q\ x\} \Longrightarrow$ *open* $\{x.\ P\ x \longrightarrow Q\ x\}$
unfolding *imp-conv-disj* **by** (*intro open-Collect-disj open-Collect-neg*)

lemma *open-Collect-const*: *open* $\{x.\ P\}$
by (*cases P*) *auto*

lemma *closed-Collect-neg*: *open* $\{x.\ P\ x\} \Longrightarrow$ *closed* $\{x.\ \neg\ P\ x\}$
unfolding *Collect-neg-eq* **by** (*rule closed-Compl*)

lemma *closed-Collect-conj*:
assumes *closed* $\{x.\ P\ x\}$ *closed* $\{x.\ Q\ x\}$
shows *closed* $\{x.\ P\ x \wedge Q\ x\}$
using *closed-Int*[*OF assms*] **by** (*simp add*: *Int-def*)

lemma *closed-Collect-disj*:
assumes *closed* $\{x.\ P\ x\}$ *closed* $\{x.\ Q\ x\}$
shows *closed* $\{x.\ P\ x \vee Q\ x\}$
using *closed-Un*[*OF assms*] **by** (*simp add*: *Un-def*)

lemma *closed-Collect-all*: $(\bigwedge i.$ *closed* $\{x.\ P\ i\ x\}) \Longrightarrow$ *closed* $\{x.\ \forall\, i.\ P\ i\ x\}$
using *closed-INT*[*of UNIV* $\lambda i.$ $\{x.\ P\ i\ x\}$] **by** (*simp add*: *Collect-all-eq*)

lemma *closed-Collect-imp*: *open* $\{x.\ P\ x\} \Longrightarrow$ *closed* $\{x.\ Q\ x\} \Longrightarrow$ *closed* $\{x.\ P\ x \longrightarrow Q\ x\}$
unfolding *imp-conv-disj* **by** (*intro closed-Collect-disj closed-Collect-neg*)

lemma *closed-Collect-const*: *closed* $\{x.\ P\}$
by (*cases P*) *auto*

end

98.2 Hausdorff and other separation properties

class *t0-space* = *topological-space* +
  assumes *t0-space*: $x \neq y \Longrightarrow \exists\, U.\ open\ U \wedge \neg\ (x \in U \longleftrightarrow y \in U)$

class *t1-space* = *topological-space* +
  assumes *t1-space*: $x \neq y \Longrightarrow \exists\, U.\ open\ U \wedge x \in U \wedge y \notin U$

instance *t1-space* $\subseteq$ *t0-space*
  by *standard* (*fast dest*: *t1-space*)

context *t1-space* **begin**

lemma *separation-t1*: $x \neq y \longleftrightarrow (\exists\, U.\ open\ U \wedge x \in U \wedge y \notin U)$
  using *t1-space*[*of x y*] **by** *blast*

lemma *closed-singleton* [*iff*]: *closed* $\{a\}$
proof −
  let *?T* = $\bigcup\{S.\ open\ S \wedge a \notin S\}$
  have *open ?T*
    by (*simp add*: *open-Union*)
  also have *?T* = − $\{a\}$
    by (*auto simp add*: *set-eq-iff separation-t1*)
  finally show *closed* $\{a\}$
    by (*simp only*: *closed-def*)
qed

lemma *closed-insert* [*continuous-intros*, *simp*]:
  assumes *closed S*
  shows *closed* (*insert a S*)
proof −
  from *closed-singleton assms* **have** *closed* ($\{a\} \cup S$)
    by (*rule closed-Un*)
  then show *closed* (*insert a S*)
    by *simp*
qed

lemma *finite-imp-closed*: *finite S* $\Longrightarrow$ *closed S*
  by (*induct pred*: *finite*) *simp-all*

end

T2 spaces are also known as Hausdorff spaces.

class *t2-space* = *topological-space* +
  assumes *hausdorff*: $x \neq y \Longrightarrow \exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$

instance *t2-space* $\subseteq$ *t1-space*
  by *standard* (*fast dest*: *hausdorff*)

lemma (**in** *t2-space*) *separation-t2*: $x \neq y \longleftrightarrow (\exists\, U\ V.\ \mathit{open}\ U \wedge \mathit{open}\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\})$
using *hausdorff* [*of x y*] **by** *blast*

lemma (**in** *t0-space*) *separation-t0*: $x \neq y \longleftrightarrow (\exists\, U.\ \mathit{open}\ U \wedge \neg\ (x \in U \longleftrightarrow y \in U))$
using *t0-space* [*of x y*] **by** *blast*

A classical separation axiom for topological space, the T3 axiom – also called regularity: if a point is not in a closed set, then there are open sets separating them.

class *t3-space* = *t2-space* +
assumes *t3-space*: $\mathit{closed}\ S \Longrightarrow y \notin S \Longrightarrow \exists\, U\ V.\ \mathit{open}\ U \wedge \mathit{open}\ V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\}$

A classical separation axiom for topological space, the T4 axiom – also called normality: if two closed sets are disjoint, then there are open sets separating them.

class *t4-space* = *t2-space* +
assumes *t4-space*: $\mathit{closed}\ S \Longrightarrow \mathit{closed}\ T \Longrightarrow S \cap T = \{\} \Longrightarrow \exists\, U\ V.\ \mathit{open}\ U \wedge \mathit{open}\ V \wedge S \subseteq U \wedge T \subseteq V \wedge U \cap V = \{\}$

T4 is stronger than T3, and weaker than metric.

instance *t4-space* $\subseteq$ *t3-space*
proof
fix *S* **and** *y*::*'a* **assume** *closed S y* $\notin$ *S*
then show $\exists\, U\ V.\ \mathit{open}\ U \wedge \mathit{open}\ V \wedge y \in U \wedge S \subseteq V \wedge U \cap V = \{\}$
using *t4-space*[*of* {*y*} *S*] **by** *auto*
qed

A perfect space is a topological space with no isolated points.

class *perfect-space* = *topological-space* +
assumes *not-open-singleton*: $\neg\ \mathit{open}\ \{x\}$

lemma (**in** *perfect-space*) *UNIV-not-singleton*: $\mathit{UNIV} \neq \{x\}$
for *x*::*'a*
by (*metis* (*no-types*) *open-UNIV not-open-singleton*)

98.3 Generators for topologies

inductive *generate-topology* :: *'a set set* $\Rightarrow$ *'a set* $\Rightarrow$ *bool* **for** *S* :: *'a set set*
where
UNIV: *generate-topology S UNIV*
| *Int*: *generate-topology S* $(a \cap b)$ **if** *generate-topology S a* **and** *generate-topology S b*
| *UN*: *generate-topology S* $(\bigcup K)$ **if** $(\bigwedge k.\ k \in K \Longrightarrow \mathit{generate\text{-}topology}\ S\ k)$
| *Basis*: *generate-topology S s* **if** $s \in S$

hide-fact (**open**) *UNIV Int UN Basis*

lemma *generate-topology-Union*:
(⋀*k*. *k* ∈ *I* ⟹ *generate-topology S* (*K k*)) ⟹ *generate-topology S* (⋃*k*∈*I*. *K k*)
using *generate-topology.UN* [*of K ' I*] **by** *auto*

lemma *topological-space-generate-topology*: *class.topological-space* (*generate-topology S*)
by *standard* (*auto intro*: *generate-topology.intros*)

98.4 Order topologies

class *order-topology* = *order* + *open* +
assumes *open-generated-order*: *open* = *generate-topology* (*range* (λ*a*. {..< *a*}) ∪ *range* (λ*a*. {*a* <..}))
begin

subclass *topological-space*
unfolding *open-generated-order*
by (*rule topological-space-generate-topology*)

lemma *open-greaterThan* [*continuous-intros*, *simp*]: *open* {*a* <..}
unfolding *open-generated-order* **by** (*auto intro*: *generate-topology.Basis*)

lemma *open-lessThan* [*continuous-intros*, *simp*]: *open* {..< *a*}
unfolding *open-generated-order* **by** (*auto intro*: *generate-topology.Basis*)

lemma *open-greaterThanLessThan* [*continuous-intros*, *simp*]: *open* {*a* <..< *b*}
unfolding *greaterThanLessThan-eq* **by** (*simp add*: *open-Int*)

end

class *linorder-topology* = *linorder* + *order-topology*

lemma *closed-atMost* [*continuous-intros*, *simp*]: *closed* {..*a*}
for *a* :: *'a*::*linorder-topology*
by (*simp add*: *closed-open*)

lemma *closed-atLeast* [*continuous-intros*, *simp*]: *closed* {*a*..}
for *a* :: *'a*::*linorder-topology*
by (*simp add*: *closed-open*)

lemma *closed-atLeastAtMost* [*continuous-intros*, *simp*]: *closed* {*a*..*b*}
for *a b* :: *'a*::*linorder-topology*
proof −
have {*a* .. *b*} = {*a* ..} ∩ {.. *b*}
by *auto*
then show *?thesis*
by (*simp add*: *closed-Int*)

qed

lemma (**in** *order*) *less-separate*:
assumes $x < y$
shows $\exists\, a\ b.\ x \in \{..< a\} \wedge y \in \{b <..\} \wedge \{..< a\} \cap \{b <..\} = \{\}$
proof (*cases* $\exists\, z.\ x < z \wedge z < y$)
case *True*
then obtain z **where** $x < z \wedge z < y$ **..**
then have $x \in \{..< z\} \wedge y \in \{z <..\} \wedge \{z <..\} \cap \{..< z\} = \{\}$
by *auto*
then show *?thesis* **by** *blast*
next
case *False*
with ‹$x < y$› **have** $x \in \{..< y\}\ y \in \{x <..\}\ \{x <..\} \cap \{..< y\} = \{\}$
by *auto*
then show *?thesis* **by** *blast*
qed

instance *linorder-topology* $\subseteq$ *t2-space*
proof
fix $x\ y :: 'a$
show $x \neq y \Longrightarrow \exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$
using *less-separate* [*of x y*] *less-separate* [*of y x*]
by (*elim neqE*; *metis open-lessThan open-greaterThan Int-commute*)
qed

lemma (**in** *linorder-topology*) *open-right*:
assumes *open* $S\ x \in S$
and *gt-ex*: $x < y$
shows $\exists\, b{>}x.\ \{x\ ..< b\} \subseteq S$
using *assms* **unfolding** *open-generated-order*
proof *induct*
case *UNIV*
then show *?case* **by** *blast*
next
case (*Int A B*)
then obtain $a\ b$ **where** $a > x\ \{x\ ..< a\} \subseteq A\ \ b > x\ \{x\ ..< b\} \subseteq B$
by *auto*
then show *?case*
by (*auto intro*!: *exI*[*of - min a b*])
next
case *UN*
then show *?case* **by** *blast*
next
case *Basis*
then show *?case*
by (*fastforce intro*: *exI*[*of - y*] *gt-ex*)
qed

lemma (**in** *linorder-topology*) *open-left*:
assumes *open S x* $\in$ *S*
and *lt-ex*: *y* < *x*
shows $\exists$ *b*<*x*. {*b* <.. *x*} $\subseteq$ *S*
using *assms* **unfolding** *open-generated-order*
proof *induction*
case *UNIV*
then show *?case* **by** *blast*
next
case (*Int A B*)
then obtain *a b* **where** *a* < *x* {*a* <.. *x*} $\subseteq$ *A* *b* < *x* {*b* <.. *x*} $\subseteq$ *B*
by *auto*
then show *?case*
by (*auto intro*!: *exI*[*of* - *max a b*])
next
case *UN*
then show *?case* **by** *blast*
next
case *Basis*
then show *?case*
by (*fastforce intro*: *exI*[*of* - *y*] *lt-ex*)
qed

lemma *filterlim-atLeastAtMost-at-bot-at-top*:
fixes *f g* :: *'a* $\Rightarrow$ *'b* :: *linorder-topology*
assumes *filterlim f at-bot F filterlim g at-top F*
assumes [*simp*]: $\bigwedge$*a b*. *finite* {*a*..*b*::*'b*}
shows *filterlim* (λ*x*. {*f x*..*g x*}) *finite-sets-at-top F*
unfolding *filterlim-finite-subsets-at-top*
proof *safe*
fix *X* :: *'b set*
assume *X*: *finite X*
from *X* **obtain** *lb* **where** *lb*: $\bigwedge$*x*. *x* $\in$ *X* $\Longrightarrow$ *lb* $\leq$ *x*
by (*metis finite-has-minimal2 nle-le*)
from *X* **obtain** *ub* **where** *ub*: $\bigwedge$*x*. *x* $\in$ *X* $\Longrightarrow$ *x* $\leq$ *ub*
by (*metis all-not-in-conv finite-has-maximal nle-le*)
have *eventually* (λ*x*. *f x* $\leq$ *lb*) *F eventually* (λ*x*. *g x* $\geq$ *ub*) *F*
using *assms* **by** (*simp-all add*: *filterlim-at-bot filterlim-at-top*)
thus *eventually* (λ*x*. *finite* {*f x*..*g x*} $\wedge$ *X* $\subseteq$ {*f x*..*g x*} $\wedge$ {*f x*..*g x*} $\subseteq$ *UNIV*) *F*
proof *eventually-elim*
case (*elim x*)
have *X* $\subseteq$ {*f x*..*g x*}
proof
fix *y* **assume** *y* $\in$ *X*
thus *y* $\in$ {*f x*..*g x*}
using *lb*[*of y*] *ub*[*of y*] *elim* **by** *auto*
qed
thus *?case*
by *auto*

qed
qed

98.5 Setup some topologies

98.5.1 Boolean is an order topology

class *discrete-topology* = *topological-space* +
assumes *open-discrete*: $\bigwedge A.$ *open* A

instance *discrete-topology* < *t2-space*
proof
fix x y :: $'a$
assume $x \neq y$
then show $\exists\, U\ V.$ *open* $U \wedge$ *open* $V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$
by (*intro exI*[*of* - {-}]) (*auto intro*!: *open-discrete*)
qed

instantiation *bool* :: *linorder-topology*
begin

definition *open-bool* :: *bool set* ⇒ *bool*
where *open-bool* = *generate-topology* (*range* (λ*a*. {..< *a*}) ∪ *range* (λ*a*. {*a* <..}))

instance
by *standard* (*rule open-bool-def*)

end

instance *bool* :: *discrete-topology*
proof
fix A :: *bool set*
have ∗: {*False* <..} = {*True*} {..< *True*} = {*False*}
by *auto*
have A = *UNIV* ∨ A = {} ∨ A = {*False* <..} ∨ A = {..< *True*}
using *subset-UNIV*[*of* A] **unfolding** *UNIV-bool* ∗ **by** *blast*
then show *open* A
by *auto*
qed

instantiation *nat* :: *linorder-topology*
begin

definition *open-nat* :: *nat set* ⇒ *bool*
where *open-nat* = *generate-topology* (*range* (λ*a*. {..< *a*}) ∪ *range* (λ*a*. {*a* <..}))

instance
by *standard* (*rule open-nat-def*)

end

instance *nat* :: *discrete-topology*
proof
  fix *A* :: *nat set*
  have *open* {*n*} **for** *n* :: *nat*
  proof (*cases n*)
    case *0*
    moreover have {*0*} = {..<*1*::*nat*}
      by *auto*
    ultimately show *?thesis*
      by *auto*
  next
    case (*Suc n'*)
    then have {*n*} = {..<*Suc n*} ∩ {*n'* <..}
      by *auto*
    with *Suc* **show** *?thesis*
      by (*auto intro*: *open-lessThan open-greaterThan*)
  qed
  then have *open* (⋃ *a*∈*A*. {*a*})
    by (*intro open-UN*) *auto*
  then show *open A*
    by *simp*
qed

instantiation *int* :: *linorder-topology*
begin

definition *open-int* :: *int set* ⇒ *bool*
  where *open-int* = *generate-topology* (*range* (λ*a*. {..< *a*}) ∪ *range* (λ*a*. {*a* <..}))

instance
  by *standard* (*rule open-int-def*)

end

instance *int* :: *discrete-topology*
proof
  fix *A* :: *int set*
  have {..<*i* + *1*} ∩ {*i*−*1* <..} = {*i*} **for** *i* :: *int*
    by *auto*
  then have *open* {*i*} **for** *i* :: *int*
    using *open-Int*[*OF open-lessThan*[*of i* + *1*] *open-greaterThan*[*of i* − *1*]] **by** *auto*
  then have *open* (⋃ *a*∈*A*. {*a*})
    by (*intro open-UN*) *auto*
  then show *open A*
    by *simp*
qed

98.5.2 Topological filters

definition (**in** *topological-space*) *nhds* :: *'a* ⇒ *'a filter*
where *nhds a* = (*INF S*∈{*S*. *open S* ∧ *a* ∈ *S*}. *principal S*)

definition (**in** *topological-space*) *at-within* :: *'a* ⇒ *'a set* ⇒ *'a filter*
(‹*at* (-)/ *within* (-)› [*1000*, *60*] *60*)
where *at a within s* = *inf* (*nhds a*) (*principal* (*s* − {*a*}))

abbreviation (**in** *topological-space*) *at* :: *'a* ⇒ *'a filter* (‹*at*›)
where *at x* ≡ *at x within* (*CONST UNIV*)

abbreviation (**in** *order-topology*) *at-right* :: *'a* ⇒ *'a filter*
where *at-right x* ≡ *at x within* {*x* <..}

abbreviation (**in** *order-topology*) *at-left* :: *'a* ⇒ *'a filter*
where *at-left x* ≡ *at x within* {..< *x*}

lemma (**in** *topological-space*) *nhds-generated-topology*:
open = *generate-topology T* ⟹ *nhds x* = (*INF S*∈{*S*∈*T*. *x* ∈ *S*}. *principal S*)
unfolding *nhds-def*
proof (*safe intro*!: *antisym INF-greatest*)
fix *S*
assume *generate-topology T S x* ∈ *S*
then show (*INF S*∈{*S* ∈ *T*. *x* ∈ *S*}. *principal S*) ≤ *principal S*
by *induct*
(*auto intro*: *INF-lower order-trans simp*: *inf-principal*[*symmetric*] *simp del*: *inf-principal*)
qed (*auto intro*!: *INF-lower intro*: *generate-topology.intros*)

lemma (**in** *topological-space*) *eventually-nhds*:
eventually P (*nhds a*) ⟷ (∃ *S*. *open S* ∧ *a* ∈ *S* ∧ (∀ *x*∈*S*. *P x*))
unfolding *nhds-def* **by** (*subst eventually-INF-base*) (*auto simp*: *eventually-principal*)

lemma *eventually-eventually*:
eventually (λ*y*. *eventually P* (*nhds y*)) (*nhds x*) = *eventually P* (*nhds x*)
by (*auto simp*: *eventually-nhds*)

lemma (**in** *topological-space*) *eventually-nhds-in-open*:
open s ⟹ *x* ∈ *s* ⟹ *eventually* (λ*y*. *y* ∈ *s*) (*nhds x*)
by (*subst eventually-nhds*) *blast*

lemma (**in** *topological-space*) *eventually-nhds-x-imp-x*: *eventually P* (*nhds x*) ⟹ *P x*
by (*subst* (*asm*) *eventually-nhds*) *blast*

lemma (**in** *topological-space*) *nhds-neq-bot* [*simp*]: *nhds a* ≠ *bot*
by (*simp add*: *trivial-limit-def eventually-nhds*)

lemma (**in** *t1-space*) *t1-space-nhds*: *x* ≠ *y* ⟹ (∀ *F x in nhds x*. *x* ≠ *y*)

by (*drule t1-space*) (*auto simp*: *eventually-nhds*)

lemma (**in** *topological-space*) *nhds-discrete-open*: *open* $\{x\} \Longrightarrow$ *nhds* x = *principal* $\{x\}$
by (*auto simp*: *nhds-def intro*!: *antisym INF-greatest INF-lower2*[*of* $\{x\}$])

lemma (**in** *discrete-topology*) *nhds-discrete*: *nhds* x = *principal* $\{x\}$
by (*simp add*: *nhds-discrete-open open-discrete*)

lemma (**in** *discrete-topology*) *at-discrete*: *at* x *within* S = *bot*
unfolding *at-within-def nhds-discrete* **by** *simp*

lemma (**in** *discrete-topology*) *tendsto-discrete*:
filterlim (f :: $'b \Rightarrow 'a$) (*nhds* y) $F \longleftrightarrow$ *eventually* ($\lambda x.\ f\ x = y$) F
by (*auto simp*: *nhds-discrete filterlim-principal*)

lemma (**in** *topological-space*) *at-within-eq*:
at x *within* s = (*INF* $S \in \{S.\ open\ S \wedge x \in S\}$. *principal* ($S \cap s - \{x\}$))
unfolding *nhds-def at-within-def*
by (*subst INF-inf-const2*[*symmetric*]) (*auto simp*: *Diff-Int-distrib*)

lemma (**in** *topological-space*) *eventually-at-filter*:
eventually P (*at* a *within* s) $\longleftrightarrow$ *eventually* ($\lambda x.\ x \neq a \longrightarrow x \in s \longrightarrow P\ x$) (*nhds* a)
by (*simp add*: *at-within-def eventually-inf-principal imp-conjL*[*symmetric*] *conj-commute*)

lemma (**in** *topological-space*) *at-le*: $s \subseteq t \Longrightarrow$ *at* x *within* $s \leq$ *at* x *within* t
unfolding *at-within-def* **by** (*intro inf-mono*) *auto*

lemma (**in** *topological-space*) *eventually-at-topological*:
eventually P (*at* a *within* s) $\longleftrightarrow$ ($\exists S.\ open\ S \wedge a \in S \wedge (\forall x \in S.\ x \neq a \longrightarrow x \in s \longrightarrow P\ x)$)
by (*simp add*: *eventually-nhds eventually-at-filter*)

lemma *eventually-nhds-conv-at*:
eventually P (*nhds* x) $\longleftrightarrow$ *eventually* P (*at* x) $\wedge$ $P\ x$
unfolding *eventually-at-topological eventually-nhds* **by** *fast*

lemma *eventually-at-in-open*:
assumes *open* A $x \in A$
shows *eventually* ($\lambda y.\ y \in A - \{x\}$) (*at* x)
using *assms eventually-at-topological* **by** *blast*

lemma *eventually-at-in-open'*:
assumes *open* A $x \in A$
shows *eventually* ($\lambda y.\ y \in A$) (*at* x)
using *assms eventually-at-topological* **by** *blast*

lemma (**in** *topological-space*) *at-within-open*: $a \in S \Longrightarrow$ *open* $S \Longrightarrow$ *at* a *within* S

$= at\ a$
unfolding *filter-eq-iff eventually-at-topological* **by** (*metis open-Int Int-iff UNIV-I*)

lemma (**in** *topological-space*) *at-within-open-NO-MATCH*:
$a \in s \Longrightarrow open\ s \Longrightarrow NO\text{-}MATCH\ UNIV\ s \Longrightarrow at\ a\ within\ s = at\ a$
by (*simp only*: *at-within-open*)

lemma (**in** *topological-space*) *at-within-open-subset*:
$a \in S \Longrightarrow open\ S \Longrightarrow S \subseteq T \Longrightarrow at\ a\ within\ T = at\ a$
by (*metis at-le at-within-open dual-order.antisym subset-UNIV*)

lemma (**in** *topological-space*) *at-within-nhd*:
assumes $x \in S$ *open* S $T \cap S - \{x\} = U \cap S - \{x\}$
shows *at* x *within* $T = at\ x\ within\ U$
unfolding *filter-eq-iff eventually-at-filter*
proof (*intro allI eventually-subst*)
have *eventually* $(\lambda x.\ x \in S)$ (*nhds* x)
using ‹$x \in S$› ‹*open* S› **by** (*auto simp*: *eventually-nhds*)
then show $\forall_F\ n$ *in nhds* x. $(n \neq x \longrightarrow n \in T \longrightarrow P\ n) = (n \neq x \longrightarrow n \in U \longrightarrow P\ n)$ **for** P
by *eventually-elim* (*insert* ‹$T \cap S - \{x\} = U \cap S - \{x\}$›, *blast*)
qed

lemma (**in** *topological-space*) *at-within-empty* [*simp*]: *at a within* $\{\} = bot$
unfolding *at-within-def* **by** *simp*

lemma (**in** *topological-space*) *at-within-union*:
at x *within* $(S \cup T) = sup\ (at\ x\ within\ S)\ (at\ x\ within\ T)$
unfolding *filter-eq-iff eventually-sup eventually-at-filter*
by (*auto elim*!: *eventually-rev-mp*)

lemma (**in** *topological-space*) *at-eq-bot-iff*: *at a* $= bot \longleftrightarrow open\ \{a\}$
unfolding *trivial-limit-def eventually-at-topological*
by (*metis UNIV-I empty-iff is-singletonE is-singletonI' singleton-iff*)

lemma (**in** *t1-space*) *eventually-neq-at-within*:
eventually $(\lambda w.\ w \neq x)$ (*at z within A*)
by (*smt* (*verit*, *ccfv-threshold*) *eventually-True eventually-at-topological separation-t1*)

lemma (**in** *perfect-space*) *at-neq-bot* [*simp*]: *at a* $\neq bot$
by (*simp add*: *at-eq-bot-iff not-open-singleton*)

lemma (**in** *order-topology*) *nhds-order*:
nhds $x = inf\ (INF\ a{\in}\{x <..\}.\ principal\ \{..< a\})\ (INF\ a{\in}\{..< x\}.\ principal\ \{a <..\})$
proof −
have *1*: $\{S \in range\ lessThan \cup range\ greaterThan.\ x \in S\} =$
$(\lambda a.\ \{..< a\})\ `\ \{x <..\} \cup (\lambda a.\ \{a <..\})\ `\ \{..< x\}$

by *auto*
show *?thesis*
by (*simp only*: *nhds-generated-topology*[*OF open-generated-order*] *INF-union 1 INF-image comp-def*)
qed

lemma (**in** *topological-space*) *filterlim-at-within-If*:
assumes *filterlim f G* (*at x within* (*A* ∩ {*x*. *P x*}))
and *filterlim g G* (*at x within* (*A* ∩ {*x*. ¬*P x*}))
shows *filterlim* (λ*x*. *if P x then f x else g x*) *G* (*at x within A*)
proof (*rule filterlim-If*)
note *assms*(*1*)
also have *at x within* (*A* ∩ {*x*. *P x*}) = *inf* (*nhds x*) (*principal* (*A* ∩ *Collect P* − {*x*}))
by (*simp add*: *at-within-def*)
also have *A* ∩ *Collect P* − {*x*} = (*A* − {*x*}) ∩ *Collect P*
by *blast*
also have *inf* (*nhds x*) (*principal* ...) = *inf* (*at x within A*) (*principal* (*Collect P*))
by (*simp add*: *at-within-def inf-assoc*)
finally show *filterlim f G* (*inf* (*at x within A*) (*principal* (*Collect P*))) **.**
next
note *assms*(*2*)
also have *at x within* (*A* ∩ {*x*. ¬ *P x*}) = *inf* (*nhds x*) (*principal* (*A* ∩ {*x*. ¬ *P x*} − {*x*}))
by (*simp add*: *at-within-def*)
also have *A* ∩ {*x*. ¬ *P x*} − {*x*} = (*A* − {*x*}) ∩ {*x*. ¬ *P x*}
by *blast*
also have *inf* (*nhds x*) (*principal* ...) = *inf* (*at x within A*) (*principal* {*x*. ¬ *P x*})
by (*simp add*: *at-within-def inf-assoc*)
finally show *filterlim g G* (*inf* (*at x within A*) (*principal* {*x*. ¬ *P x*})) **.**
qed

lemma (**in** *topological-space*) *filterlim-at-If*:
assumes *filterlim f G* (*at x within* {*x*. *P x*})
and *filterlim g G* (*at x within* {*x*. ¬*P x*})
shows *filterlim* (λ*x*. *if P x then f x else g x*) *G* (*at x*)
using *assms* **by** (*intro filterlim-at-within-If*) *simp-all*
lemma (**in** *linorder-topology*) *at-within-order*:
assumes *UNIV* ≠ {*x*}
shows *at x within s* =
inf (*INF a*∈{*x* <..}. *principal* ({..< *a*} ∩ *s* − {*x*}))
(*INF a*∈{..< *x*}. *principal* ({*a* <..} ∩ *s* − {*x*}))
proof (*cases* {*x* <..} = {} {..< *x*} = {} *rule*: *case-split* [*case-product case-split*])
case *True-True*
have *UNIV* = {..< *x*} ∪ {*x*} ∪ {*x* <..}
by *auto*
with *assms True-True* **show** *?thesis*

by *auto*
qed (*auto simp del*: *inf-principal simp*: *at-within-def nhds-order Int-Diff*
inf-principal[*symmetric*] *INF-inf-const2 inf-sup-aci*[**where** *'a='a filter*])

lemma (**in** *linorder-topology*) *at-left-eq*:
$y < x \Longrightarrow$ *at-left* $x =$ (*INF* $a \in \{..< x\}$. *principal* $\{a <..< x\}$)
by (*subst at-within-order*)
(*auto simp*: *greaterThan-Int-greaterThan greaterThanLessThan-eq*[*symmetric*]
min.absorb2 INF-constant
intro!: *INF-lower2 inf-absorb2*)

lemma (**in** *linorder-topology*) *eventually-at-left*:
$y < x \Longrightarrow$ *eventually* P (*at-left* x) $\longleftrightarrow$ ($\exists\, b<x.\ \forall\, y>b.\ y < x \longrightarrow P\ y$)
unfolding *at-left-eq*
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal Ball-def*)

lemma (**in** *linorder-topology*) *at-right-eq*:
$x < y \Longrightarrow$ *at-right* $x =$ (*INF* $a \in \{x <..\}$. *principal* $\{x <..< a\}$)
by (*subst at-within-order*)
(*auto simp*: *lessThan-Int-lessThan greaterThanLessThan-eq*[*symmetric*] *max.absorb2*
INF-constant Int-commute
intro!: *INF-lower2 inf-absorb1*)

lemma (**in** *linorder-topology*) *eventually-at-right*:
$x < y \Longrightarrow$ *eventually* P (*at-right* x) $\longleftrightarrow$ ($\exists\, b>x.\ \forall\, y>x.\ y < b \longrightarrow P\ y$)
unfolding *at-right-eq*
by (*subst eventually-INF-base*) (*auto simp*: *eventually-principal Ball-def*)

lemma *eventually-at-right-less*: $\forall_F$ *y in at-right* (x::*'a*::{*linorder-topology*, *no-top*}).
$x < y$
using *gt-ex*[*of x*] *eventually-at-right*[*of x*] **by** *auto*

lemma *trivial-limit-at-right-top*: *at-right* (*top*::-::{*order-top*,*linorder-topology*}) =
bot
by (*auto simp*: *filter-eq-iff eventually-at-topological*)

lemma *trivial-limit-at-left-bot*: *at-left* (*bot*::-::{*order-bot*,*linorder-topology*}) = *bot*
by (*auto simp*: *filter-eq-iff eventually-at-topological*)

lemma *trivial-limit-at-left-real* [*simp*]: ¬ *trivial-limit* (*at-left x*)
for x :: *'a*::{*no-bot*,*dense-order*,*linorder-topology*}
using *lt-ex* [*of x*]
by *safe* (*auto simp add*: *trivial-limit-def eventually-at-left dest*: *dense*)

lemma *trivial-limit-at-right-real* [*simp*]: ¬ *trivial-limit* (*at-right x*)
for x :: *'a*::{*no-top*,*dense-order*,*linorder-topology*}
using *gt-ex*[*of x*]
by *safe* (*auto simp add*: *trivial-limit-def eventually-at-right dest*: *dense*)

lemma (**in** *linorder-topology*) *at-eq-sup-left-right*: *at x = sup* (*at-left x*) (*at-right x*)
by (*auto simp*: *eventually-at-filter filter-eq-iff eventually-sup elim*: *eventually-elim2 eventually-mono*)

lemma (**in** *linorder-topology*) *eventually-at-split*:
eventually P (*at x*) ⟷ *eventually P* (*at-left x*) ∧ *eventually P* (*at-right x*)
by (*subst at-eq-sup-left-right*) (*simp add*: *eventually-sup*)

lemma (**in** *order-topology*) *eventually-at-leftI*:
assumes ⋀*x*. *x* ∈ {*a*<..<*b*} ⟹ *P x a* < *b*
shows *eventually P* (*at-left b*)
using *assms* **unfolding** *eventually-at-topological* **by** (*intro exI*[*of* - {*a*<..}]) *auto*

lemma (**in** *order-topology*) *eventually-at-rightI*:
assumes ⋀*x*. *x* ∈ {*a*<..<*b*} ⟹ *P x a* < *b*
shows *eventually P* (*at-right a*)
using *assms* **unfolding** *eventually-at-topological* **by** (*intro exI*[*of* - {..<*b*}]) *auto*

lemma *eventually-filtercomap-nhds*:
eventually P (*filtercomap f* (*nhds x*)) ⟷ (∃ *S*. *open S* ∧ *x* ∈ *S* ∧ (∀ *x*. *f x* ∈ *S* ⟶ *P x*))
unfolding *eventually-filtercomap eventually-nhds* **by** *auto*

lemma *eventually-filtercomap-at-topological*:
eventually P (*filtercomap f* (*at A within B*)) ⟷
(∃ *S*. *open S* ∧ *A* ∈ *S* ∧ (∀ *x*. *f x* ∈ *S* ∩ *B* − {*A*} ⟶ *P x*)) (**is** *?lhs* = *?rhs*)
unfolding *at-within-def filtercomap-inf eventually-inf-principal filtercomap-principal*

eventually-filtercomap-nhds eventually-principal **by** *blast*

lemma *eventually-at-right-field*:
eventually P (*at-right x*) ⟷ (∃ *b*>*x*. ∀ *y*>*x*. *y* < *b* ⟶ *P y*)
for *x* :: *'a*::{*linordered-field*, *linorder-topology*}
using *linordered-field-no-ub*[*rule-format*, *of x*]
by (*auto simp*: *eventually-at-right*)

lemma *eventually-at-left-field*:
eventually P (*at-left x*) ⟷ (∃ *b*<*x*. ∀ *y*>*b*. *y* < *x* ⟶ *P y*)
for *x* :: *'a*::{*linordered-field*, *linorder-topology*}
using *linordered-field-no-lb*[*rule-format*, *of x*]
by (*auto simp*: *eventually-at-left*)

lemma *filtermap-nhds-eq-imp-filtermap-at-eq*:
assumes *filtermap f* (*nhds z*) = *nhds* (*f z*)
assumes *eventually* (λ*x*. *f x* = *f z* ⟶ *x* = *z*) (*at z*)
shows *filtermap f* (*at z*) = *at* (*f z*)
proof (*rule filter-eqI*)
fix *P* :: *'a* ⇒ *bool*

have *eventually P (filtermap f (at z)) ⟷ (∀_F x in nhds z. x ≠ z ⟶ P (f x))*
by (*simp add*: *eventually-filtermap eventually-at-filter*)
also have *... ⟷ (∀_F x in nhds z. f x ≠ f z ⟶ P (f x))*
by (*rule eventually-cong* [*OF assms(2)*[*unfolded eventually-at-filter*]]) *auto*
also have *... ⟷ (∀_F x in filtermap f (nhds z). x ≠ f z ⟶ P x)*
by (*simp add*: *eventually-filtermap*)
also have *filtermap f (nhds z) = nhds (f z)*
by (*rule assms*)
also have *(∀_F x in nhds (f z). x ≠ f z ⟶ P x) ⟷ (∀_F x in at (f z). P x)*
by (*simp add*: *eventually-at-filter*)
finally show *eventually P (filtermap f (at z)) = eventually P (at (f z))* **.**
qed

98.5.3 Tendsto

abbreviation (**in** *topological-space*)
tendsto :: *('b ⇒ 'a) ⇒ 'a ⇒ 'b filter ⇒ bool* (**infixr** ‹⟶› *55*)
where *(f ⟶ l) F ≡ filterlim f (nhds l) F*

definition (**in** *t2-space*) *Lim* :: *'f filter ⇒ ('f ⇒ 'a) ⇒ 'a*
where *Lim A f = (THE l. (f ⟶ l) A)*

lemma (**in** *topological-space*) *tendsto-eq-rhs*: *(f ⟶ x) F ⟹ x = y ⟹ (f ⟶ y) F*
by *simp*

named-theorems *tendsto-intros introduction rules for tendsto*
setup ‹
Global-Theory.add-thms-dynamic (***binding*** ‹*tendsto-eq-intros*›,
fn context =>
Named-Theorems.get (*Context.proof-of context*) ***named-theorems*** ‹*tendsto-intros*›
|> map-filter (*try* (*fn thm => @{thm tendsto-eq-rhs} OF* [*thm*])))
›

context *topological-space* **begin**

lemma *tendsto-def*:
(f ⟶ l) F ⟷ (∀ S. open S ⟶ l ∈ S ⟶ eventually (λx. f x ∈ S) F)
unfolding *nhds-def filterlim-INF filterlim-principal* **by** *auto*

lemma *tendsto-cong*: *(f ⟶ c) F ⟷ (g ⟶ c) F* **if** *eventually (λx. f x = g x) F*
by (*rule filterlim-cong* [*OF refl refl that*])

lemma *tendsto-mono*: *F ≤ F' ⟹ (f ⟶ l) F' ⟹ (f ⟶ l) F*
unfolding *tendsto-def le-filter-def* **by** *fast*

lemma *tendsto-ident-at* [*tendsto-intros*, *simp*, *intro*]: *((λx. x) ⟶ a) (at a within s)*

by (*auto simp*: *tendsto-def eventually-at-topological*)

lemma *tendsto-const* [*tendsto-intros*, *simp*, *intro*]: $((\lambda x.\ k) \longrightarrow k)\ F$
by (*simp add*: *tendsto-def*)

lemma *filterlim-at*:
$(LIM\ x\ F.\ f\ x :> at\ b\ within\ s) \longleftrightarrow eventually\ (\lambda x.\ f\ x \in s \wedge f\ x \neq b)\ F \wedge (f \longrightarrow b)\ F$
by (*simp add*: *at-within-def filterlim-inf filterlim-principal conj-commute*)

lemma (**in** −)
assumes *filterlim f* (*nhds L*) *F*
shows *tendsto-imp-filterlim-at-right*:
eventually $(\lambda x.\ f\ x > L)\ F \Longrightarrow$ *filterlim f* (*at-right L*) *F*
and *tendsto-imp-filterlim-at-left*:
eventually $(\lambda x.\ f\ x < L)\ F \Longrightarrow$ *filterlim f* (*at-left L*) *F*
using *assms* **by** (*auto simp*: *filterlim-at elim*: *eventually-mono*)

lemma *filterlim-at-withinI*:
assumes *filterlim f* (*nhds c*) *F*
assumes *eventually* $(\lambda x.\ f\ x \in A - \{c\})\ F$
shows *filterlim f* (*at c within A*) *F*
using *assms* **by** (*simp add*: *filterlim-at*)

lemma *filterlim-atI*:
assumes *filterlim f* (*nhds c*) *F*
assumes *eventually* $(\lambda x.\ f\ x \neq c)\ F$
shows *filterlim f* (*at c*) *F*
using *assms* **by** (*intro filterlim-at-withinI*) *simp-all*

lemma *topological-tendstoI*:
$(\bigwedge S.\ open\ S \Longrightarrow l \in S \Longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F) \Longrightarrow (f \longrightarrow l)\ F$
by (*auto simp*: *tendsto-def*)

lemma *topological-tendstoD*:
$(f \longrightarrow l)\ F \Longrightarrow open\ S \Longrightarrow l \in S \Longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ F$
by (*auto simp*: *tendsto-def*)

lemma *tendsto-bot* [*simp*]: $(f \longrightarrow a)\ bot$
by (*simp add*: *tendsto-def*)

lemma *tendsto-eventually*: $eventually\ (\lambda x.\ f\ x = l)\ net \Longrightarrow ((\lambda x.\ f\ x) \longrightarrow l)\ net$
by (*rule topological-tendstoI*) (*auto elim*: *eventually-mono*)

lemma *tendsto-principal-singleton*[*simp*]:
shows $(f \longrightarrow f\ x)\ (principal\ \{x\})$
unfolding *tendsto-def eventually-principal* **by** *simp*

end

lemma (**in** *topological-space*) *filterlim-within-subset*:
filterlim f l (*at x within S*) $\Longrightarrow$ $T \subseteq S \Longrightarrow$ *filterlim f l* (*at x within T*)
by (*blast intro*: *filterlim-mono at-le*)

lemmas *tendsto-within-subset* = *filterlim-within-subset*

lemma (**in** *order-topology*) *order-tendsto-iff*:
($f \longrightarrow x$) *F* $\longleftrightarrow$ ($\forall l<x$. *eventually* (λx. $l < f\ x$) *F*) $\wedge$ ($\forall u>x$. *eventually* (λx. $f\ x < u$) *F*)
by (*auto simp*: *nhds-order filterlim-inf filterlim-INF filterlim-principal*)

lemma (**in** *order-topology*) *order-tendstoI*:
($\bigwedge a$. $a < y \Longrightarrow$ *eventually* (λx. $a < f\ x$) *F*) $\Longrightarrow$ ($\bigwedge a$. $y < a \Longrightarrow$ *eventually* (λx. $f\ x < a$) *F*) $\Longrightarrow$
($f \longrightarrow y$) *F*
by (*auto simp*: *order-tendsto-iff*)

lemma (**in** *order-topology*) *order-tendstoD*:
assumes ($f \longrightarrow y$) *F*
shows $a < y \Longrightarrow$ *eventually* (λx. $a < f\ x$) *F*
and $y < a \Longrightarrow$ *eventually* (λx. $f\ x < a$) *F*
using *assms* **by** (*auto simp*: *order-tendsto-iff*)

lemma (**in** *linorder-topology*) *tendsto-max*[*tendsto-intros*]:
assumes *X*: ($X \longrightarrow x$) *net*
and *Y*: ($Y \longrightarrow y$) *net*
shows ((λx. *max* ($X\ x$) ($Y\ x$)) $\longrightarrow$ *max x y*) *net*
proof (*rule order-tendstoI*)
fix *a*
assume $a <$ *max x y*
then show *eventually* (λx. $a <$ *max* ($X\ x$) ($Y\ x$)) *net*
using *order-tendstoD*(*1*)[*OF X*, *of a*] *order-tendstoD*(*1*)[*OF Y*, *of a*]
by (*auto simp*: *less-max-iff-disj elim*: *eventually-mono*)
next
fix *a*
assume *max x y* $< a$
then show *eventually* (λx. *max* ($X\ x$) ($Y\ x$) $< a$) *net*
using *order-tendstoD*(*2*)[*OF X*, *of a*] *order-tendstoD*(*2*)[*OF Y*, *of a*]
by (*auto simp*: *eventually-conj-iff*)
qed

lemma (**in** *linorder-topology*) *tendsto-min*[*tendsto-intros*]:
assumes *X*: ($X \longrightarrow x$) *net*
and *Y*: ($Y \longrightarrow y$) *net*
shows ((λx. *min* ($X\ x$) ($Y\ x$)) $\longrightarrow$ *min x y*) *net*
proof (*rule order-tendstoI*)
fix *a*

assume $a < min\ x\ y$
then show *eventually* $(\lambda x.\ a < min\ (X\ x)\ (Y\ x))$ *net*
using *order-tendstoD(1)*[*OF X*, *of a*] *order-tendstoD(1)*[*OF Y*, *of a*]
by (*auto simp*: *eventually-conj-iff*)
next
fix *a*
assume $min\ x\ y < a$
then show *eventually* $(\lambda x.\ min\ (X\ x)\ (Y\ x) < a)$ *net*
using *order-tendstoD(2)*[*OF X*, *of a*] *order-tendstoD(2)*[*OF Y*, *of a*]
by (*auto simp*: *min-less-iff-disj elim*: *eventually-mono*)
qed

lemma (**in** *order-topology*)
assumes $a < b$
shows *at-within-Icc-at-right*: *at a within* $\{a..b\} =$ *at-right a*
and *at-within-Icc-at-left*: *at b within* $\{a..b\} =$ *at-left b*
using *order-tendstoD(2)*[*OF tendsto-ident-at assms*, *of* $\{a<..\}$]
using *order-tendstoD(1)*[*OF tendsto-ident-at assms*, *of* $\{..<b\}$]
by (*auto intro*!: *order-class.order-antisym filter-leI*
simp: *eventually-at-filter less-le*
elim: *eventually-elim2*)

lemma (**in** *order-topology*)
shows *at-within-Ici-at-right*: *at a within* $\{a..\} =$ *at-right a*
and *at-within-Iic-at-left*: *at a within* $\{..a\} =$ *at-left a*
using *order-tendstoD(2)*[*OF tendsto-ident-at* [**where** $s = \{a<..\}$]]
using *order-tendstoD(1)*[*OF tendsto-ident-at*[**where** $s = \{..<a\}$]]
by (*auto intro*!: *order-class.order-antisym filter-leI*
simp: *eventually-at-filter less-le*
elim: *eventually-elim2*)

lemma (**in** *order-topology*) *at-within-Icc-at*: $a < x \Longrightarrow x < b \Longrightarrow$ *at x within* $\{a..b\} =$ *at x*
by (*rule at-within-open-subset*[**where** $S=\{a<..<b\}$]) *auto*

lemma (**in** *t2-space*) *tendsto-unique*:
assumes $F \neq bot$
and $(f \longrightarrow a)\ F$
and $(f \longrightarrow b)\ F$
shows $a = b$
proof (*rule ccontr*)
assume $a \neq b$
obtain *U V* **where** *open U open V* $a \in U\ b \in V\ U \cap V = \{\}$
using *hausdorff* [*OF* ‹$a \neq b$›] **by** *fast*
have *eventually* $(\lambda x.\ f\ x \in U)\ F$
using ‹$(f \longrightarrow a)\ F$› ‹*open U*› ‹$a \in U$› **by** (*rule topological-tendstoD*)
moreover
have *eventually* $(\lambda x.\ f\ x \in V)\ F$
using ‹$(f \longrightarrow b)\ F$› ‹*open V*› ‹$b \in V$› **by** (*rule topological-tendstoD*)

ultimately
have *eventually* (λx. *False*) *F*
proof *eventually-elim*
case (*elim x*)
then have *f x* ∈ *U* ∩ *V* **by** *simp*
with ‹*U* ∩ *V* = {}› **show** *?case* **by** *simp*
qed
with ‹¬ *trivial-limit F*› **show** *False*
by (*simp add*: *trivial-limit-def*)
qed

lemma (**in** *t2-space*) *tendsto-const-iff*:
fixes *a b* :: *'a*
assumes ¬ *trivial-limit F*
shows ((λx. *a*) ⟶ *b*) *F* ⟷ *a* = *b*
by (*auto intro*!: *tendsto-unique* [*OF assms tendsto-const*])

lemma (**in** *t2-space*) *tendsto-unique'*:
assumes *F* ≠ *bot*
shows ∃≤1*l*. (*f* ⟶ *l*) *F*
using *Uniq-def assms local.tendsto-unique* **by** *fastforce*

lemma *Lim-in-closed-set*:
assumes *closed S eventually* (λx. *f*(*x*) ∈ *S*) *F F* ≠ *bot* (*f* ⟶ *l*) *F*
shows *l* ∈ *S*
proof (*rule ccontr*)
assume *l* ∉ *S*
with ‹*closed S*› **have** *open* (− *S*) *l* ∈ − *S*
by (*simp-all add*: *open-Compl*)
with *assms*(*4*) **have** *eventually* (λx. *f x* ∈ − *S*) *F*
by (*rule topological-tendstoD*)
with *assms*(*2*) **have** *eventually* (λx. *False*) *F*
by (*rule eventually-elim2*) *simp*
with *assms*(*3*) **show** *False*
by (*simp add*: *eventually-False*)
qed

lemma (**in** *t3-space*) *nhds-closed*:
assumes *x* ∈ *A* **and** *open A*
shows ∃*A'*. *x* ∈ *A'* ∧ *closed A'* ∧ *A'* ⊆ *A* ∧ *eventually* (λy. *y* ∈ *A'*) (*nhds x*)
proof −
from *assms* **have** ∃*U V*. *open U* ∧ *open V* ∧ *x* ∈ *U* ∧ − *A* ⊆ *V* ∧ *U* ∩ *V* = {}
by (*intro t3-space*) *auto*
then obtain *U V* **where** *UV*: *open U open V x* ∈ *U* −*A* ⊆ *V U* ∩ *V* = {}
by *auto*
have *eventually* (λy. *y* ∈ *U*) (*nhds x*)
using ‹*open U*› **and** ‹*x* ∈ *U*› **by** (*intro eventually-nhds-in-open*)
hence *eventually* (λy. *y* ∈ −*V*) (*nhds x*)

by *eventually-elim* (*use UV* **in** *auto*)
with *UV* **show** *?thesis* **by** (*intro exI*[*of* - − *V*]) *auto*
qed

lemma (**in** *order-topology*) *increasing-tendsto*:
assumes *bdd*: *eventually* (λ*n*. *f n* ≤ *l*) *F*
and *en*: ⋀*x*. *x* < *l* ⟹ *eventually* (λ*n*. *x* < *f n*) *F*
shows (*f* ⟶ *l*) *F*
using *assms* **by** (*intro order-tendstoI*) (*auto elim*!: *eventually-mono*)

lemma (**in** *order-topology*) *decreasing-tendsto*:
assumes *bdd*: *eventually* (λ*n*. *l* ≤ *f n*) *F*
and *en*: ⋀*x*. *l* < *x* ⟹ *eventually* (λ*n*. *f n* < *x*) *F*
shows (*f* ⟶ *l*) *F*
using *assms* **by** (*intro order-tendstoI*) (*auto elim*!: *eventually-mono*)

lemma (**in** *order-topology*) *tendsto-sandwich*:
assumes *ev*: *eventually* (λ*n*. *f n* ≤ *g n*) *net eventually* (λ*n*. *g n* ≤ *h n*) *net*
assumes *lim*: (*f* ⟶ *c*) *net* (*h* ⟶ *c*) *net*
shows (*g* ⟶ *c*) *net*
proof (*rule order-tendstoI*)
fix *a*
show *a* < *c* ⟹ *eventually* (λ*x*. *a* < *g x*) *net*
using *order-tendstoD*[*OF lim*(*1*), *of a*] *ev* **by** (*auto elim*: *eventually-elim2*)
next
fix *a*
show *c* < *a* ⟹ *eventually* (λ*x*. *g x* < *a*) *net*
using *order-tendstoD*[*OF lim*(*2*), *of a*] *ev* **by** (*auto elim*: *eventually-elim2*)
qed

lemma (**in** *t1-space*) *limit-frequently-eq*:
assumes *F* ≠ *bot*
and *frequently* (λ*x*. *f x* = *c*) *F*
and (*f* ⟶ *d*) *F*
shows *d* = *c*
proof (*rule ccontr*)
assume *d* ≠ *c*
from *t1-space*[*OF this*] **obtain** *U* **where** *open U d* ∈ *U c* ∉ *U*
by *blast*
with *assms* **have** *eventually* (λ*x*. *f x* ∈ *U*) *F*
unfolding *tendsto-def* **by** *blast*
then have *eventually* (λ*x*. *f x* ≠ *c*) *F*
by *eventually-elim* (*insert* ‹*c* ∉ *U*›, *blast*)
with *assms*(*2*) **show** *False*
unfolding *frequently-def* **by** *contradiction*
qed

lemma (**in** *t1-space*) *tendsto-imp-eventually-ne*:
assumes (*f* ⟶ *c*) *F c* ≠ *c*′

shows *eventually* $(\lambda z.\ f\ z \neq c')\ F$
proof (*cases F=bot*)
case *True*
thus *?thesis* **by** *auto*
next
case *False*
show *?thesis*
proof (*rule ccontr*)
assume ¬ *eventually* $(\lambda z.\ f\ z \neq c')\ F$
then have *frequently* $(\lambda z.\ f\ z = c')\ F$
by (*simp add*: *frequently-def*)
from *limit-frequently-eq*[*OF False this* ‹$(f \longrightarrow c)\ F$›] **and** ‹$c \neq c'$› **show** *False*
by *contradiction*
qed
qed

lemma (**in** *linorder-topology*) *tendsto-le*:
assumes *F*: ¬ *trivial-limit F*
and *x*: $(f \longrightarrow x)\ F$
and *y*: $(g \longrightarrow y)\ F$
and *ev*: *eventually* $(\lambda x.\ g\ x \leq f\ x)\ F$
shows $y \leq x$
proof (*rule ccontr*)
assume ¬ $y \leq x$
with *less-separate*[*of x y*] **obtain** *a b* **where** *xy*: $x < a\ b < y\ \{..<a\} \cap \{b<..\} = \{\}$
by (*auto simp*: *not-le*)
then have *eventually* $(\lambda x.\ f\ x < a)\ F$ *eventually* $(\lambda x.\ b < g\ x)\ F$
using *x y* **by** (*auto intro*: *order-tendstoD*)
with *ev* **have** *eventually* $(\lambda x.\ False)\ F$
by *eventually-elim* (*insert xy*, *fastforce*)
with *F* **show** *False*
by (*simp add*: *eventually-False*)
qed

lemma (**in** *linorder-topology*) *tendsto-lowerbound*:
assumes *x*: $(f \longrightarrow x)\ F$
and *ev*: *eventually* $(\lambda i.\ a \leq f\ i)\ F$
and *F*: ¬ *trivial-limit F*
shows $a \leq x$
using *F x tendsto-const ev* **by** (*rule tendsto-le*)

lemma (**in** *linorder-topology*) *tendsto-upperbound*:
assumes *x*: $(f \longrightarrow x)\ F$
and *ev*: *eventually* $(\lambda i.\ a \geq f\ i)\ F$
and *F*: ¬ *trivial-limit F*
shows $a \geq x$
by (*rule tendsto-le* [*OF F tendsto-const x ev*])

lemma *filterlim-at-within-not-equal*:
fixes $f::'a \Rightarrow 'b::t2\text{-}space$
assumes *filterlim f* (*at a within s*) *F*
shows *eventually* ($\lambda w.\ f\ w \in s \wedge f\ w \neq b$) *F*
proof (*cases a=b*)
case *True*
then show *?thesis* **using** *assms* **by** (*simp add*: *filterlim-at*)
next
case *False*
from *hausdorff*[*OF this*] **obtain** *U V* **where** *UV*:*open U open V a* $\in$ *U b* $\in$ *V* $U \cap V = \{\}$
by *auto*
have (*f* $\longrightarrow$ *a*) *F* **using** *assms filterlim-at* **by** *auto*
then have $\forall_F\ x\ in\ F.\ f\ x \in U$ **using** *UV* **unfolding** *tendsto-def* **by** *auto*
moreover have $\forall_F\ x\ in\ F.\ f\ x \in s \wedge f\ x \neq a$ **using** *assms filterlim-at* **by** *auto*
ultimately show *?thesis*
apply *eventually-elim*
using *UV* **by** *auto*
qed

98.5.4 Rules about *Lim*

lemma *tendsto-Lim*: $\neg$ *trivial-limit net* $\Longrightarrow$ (*f* $\longrightarrow$ *l*) *net* $\Longrightarrow$ *Lim net f* = *l*
unfolding *Lim-def* **using** *tendsto-unique* [*of net f*] **by** *auto*

lemma *Lim-ident-at*: $\neg$ *trivial-limit* (*at x within s*) $\Longrightarrow$ *Lim* (*at x within s*) ($\lambda x.\ x$) = *x*
by (*simp add*: *tendsto-Lim*)

lemma *Lim-cong*:
assumes $\forall_F\ x\ in\ F.\ f\ x = g\ x$ *F* = *G*
shows *Lim F f* = *Lim F g*
unfolding *t2-space-class.Lim-def* **using** *tendsto-cong assms* **by** *fastforce*

lemma *eventually-Lim-ident-at*:
($\forall_F\ y\ in\ at\ x\ within\ X.\ P$ (*Lim* (*at x within X*) ($\lambda x.\ x$)) *y*) $\longleftrightarrow$
($\forall_F\ y\ in\ at\ x\ within\ X.\ P\ x\ y$) **for** $x::'a::t2\text{-}space$
by (*cases at x within X* = *bot*) (*auto simp*: *Lim-ident-at*)

lemma *filterlim-at-bot-at-right*:
fixes *f* :: $'a::linorder\text{-}topology \Rightarrow 'b::linorder$
assumes *mono*: $\bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow x \leq y \Longrightarrow f\ x \leq f\ y$
and *bij*: $\bigwedge x.\ P\ x \Longrightarrow f\ (g\ x) = x$ $\bigwedge x.\ P\ x \Longrightarrow Q\ (g\ x)$
and *Q*: *eventually Q* (*at-right a*)
and *bound*: $\bigwedge b.\ Q\ b \Longrightarrow a < b$
and *P*: *eventually P at-bot*
shows *filterlim f at-bot* (*at-right a*)
proof −

```
  from P obtain x where x: ⋀y. y ≤ x ⟹ P y
    unfolding eventually-at-bot-linorder by auto
  show ?thesis
  proof (intro filterlim-at-bot-le[THEN iffD2] allI impI)
    fix z
    assume z ≤ x
    with x have P z by auto
    have eventually (λx. x ≤ g z) (at-right a)
      using bound[OF bij(2)[OF ‹P z›]]
      unfolding eventually-at-right[OF bound[OF bij(2)[OF ‹P z›]]]
      by (auto intro!: exI[of - g z])
    with Q show eventually (λx. f x ≤ z) (at-right a)
      by eventually-elim (metis bij ‹P z› mono)
  qed
qed

lemma filterlim-at-top-at-left:
  fixes f :: 'a::linorder-topology ⇒ 'b::linorder
  assumes mono: ⋀x y. Q x ⟹ Q y ⟹ x ≤ y ⟹ f x ≤ f y
    and bij: ⋀x. P x ⟹ f (g x) = x ⋀x. P x ⟹ Q (g x)
    and Q: eventually Q (at-left a)
    and bound: ⋀b. Q b ⟹ b < a
    and P: eventually P at-top
  shows filterlim f at-top (at-left a)
proof −
  from P obtain x where x: ⋀y. x ≤ y ⟹ P y
    unfolding eventually-at-top-linorder by auto
  show ?thesis
  proof (intro filterlim-at-top-ge[THEN iffD2] allI impI)
    fix z
    assume x ≤ z
    with x have P z by auto
    have eventually (λx. g z ≤ x) (at-left a)
      using bound[OF bij(2)[OF ‹P z›]]
      unfolding eventually-at-left[OF bound[OF bij(2)[OF ‹P z›]]]
      by (auto intro!: exI[of - g z])
    with Q show eventually (λx. z ≤ f x) (at-left a)
      by eventually-elim (metis bij ‹P z› mono)
  qed
qed

lemma filterlim-split-at:
  filterlim f F (at-left x) ⟹ filterlim f F (at-right x) ⟹
    filterlim f F (at x)
  for x :: 'a::linorder-topology
  by (subst at-eq-sup-left-right) (rule filterlim-sup)

lemma filterlim-at-split:
  filterlim f F (at x) ⟷ filterlim f F (at-left x) ∧ filterlim f F (at-right x)
```

for $x :: 'a$::*linorder-topology*
by (*subst at-eq-sup-left-right*) (*simp add*: *filterlim-def filtermap-sup*)

lemma *eventually-nhds-top*:
fixes $P :: 'a :: \{order\text{-}top, linorder\text{-}topology\} \Rightarrow bool$
and $b :: 'a$
assumes $b < top$
shows $eventually\ P\ (nhds\ top) \longleftrightarrow (\exists\, b{<}top.\ (\forall\, z.\ b < z \longrightarrow P\ z))$
unfolding *eventually-nhds*
proof *safe*
fix $S :: 'a\ set$
assume $open\ S\ top \in S$
note *open-left*[*OF this* ‹$b < top$›]
moreover assume $\forall\, s{\in}S.\ P\ s$
ultimately show $\exists\, b{<}top.\ \forall\, z{>}b.\ P\ z$
by (*auto simp*: *subset-eq Ball-def*)
next
fix b
assume $b < top\ \forall\, z{>}b.\ P\ z$
then show $\exists\, S.\ open\ S \wedge top \in S \wedge (\forall\, xa{\in}S.\ P\ xa)$
by (*intro exI*[*of* - {b <..}]) *auto*
qed

lemma *tendsto-at-within-iff-tendsto-nhds*:
$(g \longrightarrow g\ l)\ (at\ l\ within\ S) \longleftrightarrow (g \longrightarrow g\ l)\ (inf\ (nhds\ l)\ (principal\ S))$
unfolding *tendsto-def eventually-at-filter eventually-inf-principal*
by (*intro ext all-cong imp-cong*) (*auto elim*!: *eventually-mono*)

98.6 Limits on sequences

abbreviation (**in** *topological-space*)
$LIMSEQ :: [nat \Rightarrow 'a, 'a] \Rightarrow bool$ (‹(‹*notation*=‹*infix LIMSEQ*››(-)/ $\longrightarrow$ (-))› [60, 60] 60)
where $X \longrightarrow L \equiv (X \longrightarrow L)\ sequentially$

abbreviation (**in** *t2-space*) $lim :: (nat \Rightarrow 'a) \Rightarrow 'a$
where $lim\ X \equiv Lim\ sequentially\ X$

definition (**in** *topological-space*) $convergent :: (nat \Rightarrow 'a) \Rightarrow bool$
where $convergent\ X = (\exists\, L.\ X \longrightarrow L)$

lemma *lim-def*: $lim\ X = (THE\ L.\ X \longrightarrow L)$
unfolding *Lim-def* **..**

lemma *lim-explicit*:
$f \longrightarrow f0 \longleftrightarrow (\forall\, S.\ open\ S \longrightarrow f0 \in S \longrightarrow (\exists\, N.\ \forall\, n{\geq}N.\ f\ n \in S))$
unfolding *tendsto-def eventually-sequentially* **by** *auto*

lemma *closed-sequentially*:

assumes *closed S* **and** $\bigwedge n.\ f\ n \in S$ **and** $f \longrightarrow l$
shows $l \in S$
by (*metis Lim-in-closed-set assms eventually-sequentially trivial-limit-sequentially*)

98.7 Monotone sequences and subsequences

Definition of monotonicity. The use of disjunction here complicates proofs considerably. One alternative is to add a Boolean argument to indicate the direction. Another is to develop the notions of increasing and decreasing first.

definition *monoseq* :: $(nat \Rightarrow 'a{::}order) \Rightarrow bool$
where $monoseq\ X \longleftrightarrow (\forall m.\ \forall n{\geq}m.\ X\ m \leq X\ n) \vee (\forall m.\ \forall n{\geq}m.\ X\ n \leq X\ m)$

abbreviation *incseq* :: $(nat \Rightarrow 'a{::}order) \Rightarrow bool$
where $incseq\ X \equiv mono\ X$

lemma *incseq-def*: $incseq\ X \longleftrightarrow (\forall m.\ \forall n{\geq}m.\ X\ n \geq X\ m)$
unfolding *mono-def* **..**

abbreviation *decseq* :: $(nat \Rightarrow 'a{::}order) \Rightarrow bool$
where $decseq\ X \equiv antimono\ X$

lemma *decseq-def*: $decseq\ X \longleftrightarrow (\forall m.\ \forall n{\geq}m.\ X\ n \leq X\ m)$
unfolding *antimono-def* **..**

98.7.1 Definition of subsequence.

lemma *strict-mono-leD*: $strict\text{-}mono\ r \Longrightarrow m \leq n \Longrightarrow r\ m \leq r\ n$
by (*erule* (*1*) *monoD* [*OF strict-mono-mono*])

lemma *strict-mono-id*: *strict-mono id*
by (*simp add*: *strict-mono-def*)

lemma *incseq-SucI*: $(\bigwedge n.\ X\ n \leq X\ (Suc\ n)) \Longrightarrow incseq\ X$
by (*simp add*: *mono-iff-le-Suc*)

lemma *incseqD*: $incseq\ f \Longrightarrow i \leq j \Longrightarrow f\ i \leq f\ j$
by (*auto simp*: *incseq-def*)

lemma *incseq-SucD*: $incseq\ A \Longrightarrow A\ i \leq A\ (Suc\ i)$
using *incseqD*[*of A i Suc i*] **by** *auto*

lemma *incseq-Suc-iff*: $incseq\ f \longleftrightarrow (\forall n.\ f\ n \leq f\ (Suc\ n))$
by (*auto intro*: *incseq-SucI dest*: *incseq-SucD*)

lemma *incseq-const*[*simp*, *intro*]: *incseq* $(\lambda x.\ k)$
unfolding *incseq-def* **by** *auto*

lemma *decseq-SucI*: $(\bigwedge n.\ X\ (Suc\ n) \leq X\ n) \Longrightarrow decseq\ X$

by (*simp add*: *antimono-iff-le-Suc*)

lemma *decseqD*: *decseq f* $\Longrightarrow$ $i \leq j \Longrightarrow f\ j \leq f\ i$
by (*auto simp*: *decseq-def*)

lemma *decseq-SucD*: *decseq A* $\Longrightarrow$ $A\ (Suc\ i) \leq A\ i$
using *decseqD*[*of A i Suc i*] **by** *auto*

lemma *decseq-Suc-iff*: *decseq f* $\longleftrightarrow$ $(\forall n.\ f\ (Suc\ n) \leq f\ n)$
by (*auto intro*: *decseq-SucI dest*: *decseq-SucD*)

lemma *decseq-const*[*simp*, *intro*]: *decseq* $(\lambda x.\ k)$
unfolding *decseq-def* **by** *auto*

lemma *monoseq-iff*: *monoseq X* $\longleftrightarrow$ *incseq X* $\vee$ *decseq X*
unfolding *monoseq-def incseq-def decseq-def* **..**

lemma *monoseq-Suc*: *monoseq X* $\longleftrightarrow$ $(\forall n.\ X\ n \leq X\ (Suc\ n)) \vee (\forall n.\ X\ (Suc\ n) \leq X\ n)$
unfolding *monoseq-iff incseq-Suc-iff decseq-Suc-iff* **..**

lemma *monoI1*: $\forall m.\ \forall n \geq m.\ X\ m \leq X\ n \Longrightarrow$ *monoseq X*
by (*simp add*: *monoseq-def*)

lemma *monoI2*: $\forall m.\ \forall n \geq m.\ X\ n \leq X\ m \Longrightarrow$ *monoseq X*
by (*simp add*: *monoseq-def*)

lemma *mono-SucI1*: $\forall n.\ X\ n \leq X\ (Suc\ n) \Longrightarrow$ *monoseq X*
by (*simp add*: *monoseq-Suc*)

lemma *mono-SucI2*: $\forall n.\ X\ (Suc\ n) \leq X\ n \Longrightarrow$ *monoseq X*
by (*simp add*: *monoseq-Suc*)

lemma *monoseq-minus*:
fixes *a* :: *nat* $\Rightarrow$ *'a*::*ordered-ab-group-add*
assumes *monoseq a*
shows *monoseq* $(\lambda\ n.\ -\ a\ n)$
proof (*cases* $\forall m.\ \forall n \geq m.\ a\ m \leq a\ n$)
case *True*
then have $\forall m.\ \forall n \geq m.\ -\ a\ n \leq -\ a\ m$ **by** *auto*
then show *?thesis* **by** (*rule monoI2*)
next
case *False*
then have $\forall m.\ \forall n \geq m.\ -\ a\ m \leq -\ a\ n$
using ‹*monoseq a*›[*unfolded monoseq-def*] **by** *auto*
then show *?thesis* **by** (*rule monoI1*)
qed

98.7.2 Subsequence (alternative definition, (e.g. Hoskins)

For any sequence, there is a monotonic subsequence.

lemma *seq-monosub*:
fixes $s :: nat \Rightarrow 'a::linorder$
shows $\exists f.\ strict\text{-}mono\ f \wedge monoseq\ (\lambda n.\ (s\ (f\ n)))$
proof (*cases* $\forall n.\ \exists p{>}n.\ \forall m{\geq}p.\ s\ m \leq s\ p$)
case *True*
then have $\exists f.\ \forall n.\ (\forall m{\geq}f\ n.\ s\ m \leq s\ (f\ n)) \wedge f\ n < f\ (Suc\ n)$
by (*intro dependent-nat-choice*) (*auto simp*: *conj-commute*)
then obtain $f :: nat \Rightarrow nat$
where *f*: *strict-mono f* **and** *mono*: $\bigwedge n\ m.\ f\ n \leq m \Longrightarrow s\ m \leq s\ (f\ n)$
by (*auto simp*: *strict-mono-Suc-iff*)
then have *incseq f*
unfolding *strict-mono-Suc-iff incseq-Suc-iff* **by** (*auto intro*: *less-imp-le*)
then have *monoseq* $(\lambda n.\ s\ (f\ n))$
by (*auto simp add*: *incseq-def intro*!: *mono monoI2*)
with *f* **show** *?thesis*
by *auto*
next
case *False*
then obtain *N* **where** *N*: $p > N \Longrightarrow \exists m{>}p.\ s\ p < s\ m$ **for** *p*
by (*force simp*: *not-le le-less*)
have $\exists f.\ \forall n.\ N < f\ n \wedge f\ n < f\ (Suc\ n) \wedge s\ (f\ n) \leq s\ (f\ (Suc\ n))$
proof (*intro dependent-nat-choice*)
fix *x*
assume $N < x$ **with** *N*[*of x*]
show $\exists y{>}N.\ x < y \wedge s\ x \leq s\ y$
by (*auto intro*: *less-trans*)
qed *auto*
then show *?thesis*
by (*auto simp*: *monoseq-iff incseq-Suc-iff strict-mono-Suc-iff*)
qed

lemma *seq-suble*:
assumes *sf*: *strict-mono* $(f :: nat \Rightarrow nat)$
shows $n \leq f\ n$
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
with *sf* [*unfolded strict-mono-Suc-iff*, *rule-format*, *of n*] **have** $n < f\ (Suc\ n)$
by *arith*
then show *?case* **by** *arith*
qed

lemma *eventually-subseq*:
strict-mono r $\Longrightarrow$ *eventually P sequentially* $\Longrightarrow$ *eventually* $(\lambda n.\ P\ (r\ n))$ *sequen-*

tially
unfolding *eventually-sequentially* **by** (*metis seq-suble le-trans*)

lemma *not-eventually-sequentiallyD*:
assumes ¬ *eventually P sequentially*
shows ∃ *r*::*nat*⇒*nat*. *strict-mono r* ∧ (∀ *n*. ¬ *P* (*r n*))
proof −
from *assms* **have** ∀ *n*. ∃ *m*≥*n*. ¬ *P m*
unfolding *eventually-sequentially* **by** (*simp add*: *not-less*)
then obtain *r* **where** ⋀*n*. *r n* ≥ *n* ⋀*n*. ¬ *P* (*r n*)
by (*auto simp*: *choice-iff*)
then show *?thesis*
by (*auto intro*!: *exI*[*of* - λ*n*. *r* (((*Suc* ∘ *r*) ^^ *Suc n*) *0*)]
simp: *less-eq-Suc-le strict-mono-Suc-iff*)
qed

lemma *sequentially-offset*:
assumes *eventually* (λ*i*. *P i*) *sequentially*
shows *eventually* (λ*i*. *P* (*i* + *k*)) *sequentially*
using *assms* **by** (*rule eventually-sequentially-seg* [*THEN iffD2*])

lemma *seq-offset-neg*:
(*f* ⟶ *l*) *sequentially* ⟹ ((λ*i*. *f*(*i* − *k*)) ⟶ *l*) *sequentially*
apply (*erule filterlim-compose*)
apply (*simp add*: *filterlim-def le-sequentially eventually-filtermap eventually-sequentially*, *arith*)
done

lemma *filterlim-subseq*: *strict-mono f* ⟹ *filterlim f sequentially sequentially*
unfolding *filterlim-iff* **by** (*metis eventually-subseq*)

lemma *strict-mono-o*: *strict-mono r* ⟹ *strict-mono s* ⟹ *strict-mono* (*r* ∘ *s*)
unfolding *strict-mono-def* **by** *simp*

lemma *strict-mono-compose*: *strict-mono r* ⟹ *strict-mono s* ⟹ *strict-mono* (λ*x*. *r* (*s x*))
using *strict-mono-o*[*of r s*] **by** (*simp add*: *o-def*)

lemma *incseq-imp-monoseq*: *incseq X* ⟹ *monoseq X*
by (*simp add*: *incseq-def monoseq-def*)

lemma *decseq-imp-monoseq*: *decseq X* ⟹ *monoseq X*
by (*simp add*: *decseq-def monoseq-def*)

lemma *decseq-eq-incseq*: *decseq X* = *incseq* (λ*n*. − *X n*)
for *X* :: *nat* ⇒ *'a*::*ordered-ab-group-add*
by (*simp add*: *decseq-def incseq-def*)

lemma *INT-decseq-offset*:

assumes *decseq F*
shows $(\bigcap i.\ F\ i) = (\bigcap i{\in}\{n..\}.\ F\ i)$
proof *safe*
fix *x i*
assume *x*: $x \in (\bigcap i{\in}\{n..\}.\ F\ i)$
show $x \in F\ i$
proof *cases*
from *x* **have** $x \in F\ n$ **by** *auto*
also assume $i \leq n$ **with** ‹*decseq F*› **have** $F\ n \subseteq F\ i$
unfolding *decseq-def* **by** *simp*
finally show *?thesis* **.**
qed (*insert x*, *simp*)
qed *auto*

lemma *LIMSEQ-const-iff*: $(\lambda n.\ k) \longrightarrow l \longleftrightarrow k = l$
for *k l* :: *'a*::*t2-space*
using *trivial-limit-sequentially* **by** (*rule tendsto-const-iff*)

lemma *LIMSEQ-SUP*: *incseq X* $\Longrightarrow X \longrightarrow$ (*SUP i. X i* :: *'a*::{*complete-linorder*,*linorder-topology*})
by (*intro increasing-tendsto*)
(*auto simp*: *SUP-upper less-SUP-iff incseq-def eventually-sequentially intro*: *less-le-trans*)

lemma *LIMSEQ-INF*: *decseq X* $\Longrightarrow X \longrightarrow$ (*INF i. X i* :: *'a*::{*complete-linorder*,*linorder-topology*})
by (*intro decreasing-tendsto*)
(*auto simp*: *INF-lower INF-less-iff decseq-def eventually-sequentially intro*: *le-less-trans*)

lemma *LIMSEQ-ignore-initial-segment*: $f \longrightarrow a \Longrightarrow (\lambda n.\ f\ (n + k)) \longrightarrow a$
unfolding *tendsto-def* **by** (*subst eventually-sequentially-seg*[**where** *k*=*k*])

lemma *LIMSEQ-offset*: $(\lambda n.\ f\ (n + k)) \longrightarrow a \Longrightarrow f \longrightarrow a$
unfolding *tendsto-def*
by (*subst* (*asm*) *eventually-sequentially-seg*[**where** *k*=*k*])

lemma *LIMSEQ-Suc*: $f \longrightarrow l \Longrightarrow (\lambda n.\ f\ (Suc\ n)) \longrightarrow l$
by (*drule LIMSEQ-ignore-initial-segment* [**where** *k*=*Suc 0*]) *simp*

lemma *LIMSEQ-imp-Suc*: $(\lambda n.\ f\ (Suc\ n)) \longrightarrow l \Longrightarrow f \longrightarrow l$
by (*rule LIMSEQ-offset* [**where** *k*=*Suc 0*]) *simp*

lemma *LIMSEQ-lessThan-iff-atMost*:
shows $(\lambda n.\ f\ \{..<n\}) \longrightarrow x \longleftrightarrow (\lambda n.\ f\ \{..n\}) \longrightarrow x$
apply (*subst filterlim-sequentially-Suc* [*symmetric*])
apply (*simp only*: *lessThan-Suc-atMost*)
done

lemma (**in** *t2-space*) *LIMSEQ-Uniq*: $\exists_{\leq 1} l.\ X \longrightarrow l$
by (*simp add*: *tendsto-unique'*)

lemma (**in** *t2-space*) *LIMSEQ-unique*: $X \longrightarrow a \Longrightarrow X \longrightarrow b \Longrightarrow a = b$
using *trivial-limit-sequentially* **by** (*rule tendsto-unique*)

lemma *LIMSEQ-le-const*: $X \longrightarrow x \Longrightarrow \exists N.\ \forall n{\geq}N.\ a \leq X\ n \Longrightarrow a \leq x$
for $a\ x :: 'a{::}$*linorder-topology*
by (*simp add*: *eventually-at-top-linorder tendsto-lowerbound*)

lemma *LIMSEQ-le*: $X \longrightarrow x \Longrightarrow Y \longrightarrow y \Longrightarrow \exists N.\ \forall n{\geq}N.\ X\ n \leq Y\ n$
$\Longrightarrow x \leq y$
for $x\ y :: 'a{::}$*linorder-topology*
using *tendsto-le*[*of sequentially Y y X x*] **by** (*simp add*: *eventually-sequentially*)

lemma *LIMSEQ-le-const2*: $X \longrightarrow x \Longrightarrow \exists N.\ \forall n{\geq}N.\ X\ n \leq a \Longrightarrow x \leq a$
for $a\ x :: 'a{::}$*linorder-topology*
by (*rule LIMSEQ-le*[*of X x* $\lambda n.\ a$]) *auto*

lemma *Lim-bounded*: $f \longrightarrow l \Longrightarrow \forall n{\geq}M.\ f\ n \leq C \Longrightarrow l \leq C$
for $l :: 'a{::}$*linorder-topology*
by (*intro LIMSEQ-le-const2*) *auto*

lemma *Lim-bounded2*:
fixes $f :: nat \Rightarrow 'a{::}$*linorder-topology*
assumes *lim*:$f \longrightarrow l$ **and** *ge*: $\forall n{\geq}N.\ f\ n \geq C$
shows $l \geq C$
using *ge*
by (*intro tendsto-le*[*OF trivial-limit-sequentially lim tendsto-const*])
(*auto simp*: *eventually-sequentially*)

lemma *lim-mono*:
fixes $X\ Y :: nat \Rightarrow 'a{::}$*linorder-topology*
assumes $\bigwedge n.\ N \leq n \Longrightarrow X\ n \leq Y\ n$
and $X \longrightarrow x$
and $Y \longrightarrow y$
shows $x \leq y$
using *assms*(*1*) **by** (*intro LIMSEQ-le*[*OF assms*(*2*,*3*)]) *auto*

lemma *Sup-lim*:
fixes $a :: 'a{::}\{$*complete-linorder*,*linorder-topology*$\}$
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $a \leq Sup\ s$
by (*metis Lim-bounded assms complete-lattice-class.Sup-upper*)

lemma *Inf-lim*:
fixes $a :: 'a{::}\{$*complete-linorder*,*linorder-topology*$\}$
assumes $\bigwedge n.\ b\ n \in s$
and $b \longrightarrow a$
shows $Inf\ s \leq a$

by (*metis Lim-bounded2 assms complete-lattice-class.Inf-lower*)

lemma *SUP-Lim*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder,linorder\text{-}topology\}$
assumes *inc*: *incseq X*
and *l*: $X \longrightarrow l$
shows $(SUP\ n.\ X\ n) = l$
using *LIMSEQ-SUP*[*OF inc*] *tendsto-unique*[*OF trivial-limit-sequentially l*]
by *simp*

lemma *INF-Lim*:
fixes $X :: nat \Rightarrow 'a::\{complete\text{-}linorder,linorder\text{-}topology\}$
assumes *dec*: *decseq X*
and *l*: $X \longrightarrow l$
shows $(INF\ n.\ X\ n) = l$
using *LIMSEQ-INF*[*OF dec*] *tendsto-unique*[*OF trivial-limit-sequentially l*]
by *simp*

lemma *convergentD*: *convergent X* $\Longrightarrow \exists L.\ X \longrightarrow L$
by (*simp add*: *convergent-def*)

lemma *convergentI*: $X \longrightarrow L \Longrightarrow$ *convergent X*
by (*auto simp add*: *convergent-def*)

lemma *convergent-LIMSEQ-iff*: *convergent X* $\longleftrightarrow X \longrightarrow lim\ X$
by (*auto intro*: *theI LIMSEQ-unique simp add*: *convergent-def lim-def*)

lemma *convergent-const*: *convergent* $(\lambda n.\ c)$
by (*rule convergentI*) (*rule tendsto-const*)

lemma *monoseq-le*:
monoseq a $\Longrightarrow$ *a* $\longrightarrow$ *x* $\Longrightarrow$
$(\forall n.\ a\ n \leq x) \wedge (\forall m.\ \forall n{\geq}m.\ a\ m \leq a\ n) \vee$
$(\forall n.\ x \leq a\ n) \wedge (\forall m.\ \forall n{\geq}m.\ a\ n \leq a\ m)$
for $x :: 'a::linorder\text{-}topology$
by (*metis LIMSEQ-le-const LIMSEQ-le-const2 decseq-def incseq-def monoseq-iff*)

lemma *LIMSEQ-subseq-LIMSEQ*: $X \longrightarrow L \Longrightarrow$ *strict-mono f* $\Longrightarrow$ $(X \circ f) \longrightarrow L$
unfolding *comp-def* **by** (*rule filterlim-compose* [*of X*, *OF - filterlim-subseq*])

lemma *convergent-subseq-convergent*: *convergent X* $\Longrightarrow$ *strict-mono f* $\Longrightarrow$ *convergent* $(X \circ f)$
by (*auto simp*: *convergent-def intro*: *LIMSEQ-subseq-LIMSEQ*)

lemma *limI*: $X \longrightarrow L \Longrightarrow lim\ X = L$
by (*rule tendsto-Lim*) (*rule trivial-limit-sequentially*)

lemma *lim-le*: *convergent f* $\Longrightarrow (\bigwedge n.\ f\ n \leq x) \Longrightarrow lim\ f \leq x$

for $x :: 'a$::*linorder-topology*
using *LIMSEQ-le-const2*[*of f lim f x*] **by** (*simp add*: *convergent-LIMSEQ-iff*)

lemma *lim-const* [*simp*]: *lim* ($\lambda m.\ a$) = a
by (*simp add*: *limI*)

98.7.3 Increasing and Decreasing Series

lemma *incseq-le*: *incseq* $X \Longrightarrow X \longrightarrow L \Longrightarrow X\ n \leq L$
for $L :: 'a$::*linorder-topology*
by (*metis incseq-def LIMSEQ-le-const*)

lemma *decseq-ge*: *decseq* $X \Longrightarrow X \longrightarrow L \Longrightarrow L \leq X\ n$
for $L :: 'a$::*linorder-topology*
by (*metis decseq-def LIMSEQ-le-const2*)

98.8 First countable topologies

class *first-countable-topology* = *topological-space* +
assumes *first-countable-basis*:
$\exists A$::*nat* $\Rightarrow$ $'a$ *set*. ($\forall i.\ x \in A\ i \wedge$ *open* ($A\ i$)) $\wedge$ ($\forall S.$ *open* $S \wedge x \in S \longrightarrow (\exists i.\ A\ i \subseteq S)$)

lemma (**in** *first-countable-topology*) *countable-basis-at-decseq*:
obtains $A :: nat \Rightarrow 'a\ set$ **where**
$\bigwedge i.$ *open* ($A\ i$) $\bigwedge i.\ x \in (A\ i)$
$\bigwedge S.$ *open* $S \Longrightarrow x \in S \Longrightarrow$ *eventually* ($\lambda i.\ A\ i \subseteq S$) *sequentially*
proof *atomize-elim*
from *first-countable-basis*[*of x*] **obtain** $A :: nat \Rightarrow 'a\ set$
where *nhds*: $\bigwedge i.$ *open* ($A\ i$) $\bigwedge i.\ x \in A\ i$
and *incl*: $\bigwedge S.$ *open* $S \Longrightarrow x \in S \Longrightarrow \exists i.\ A\ i \subseteq S$
by *auto*
define F **where** $F\ n = (\bigcap i \leq n.\ A\ i)$ **for** n
show $\exists A.\ (\forall i.$ *open* ($A\ i$)) $\wedge$ ($\forall i.\ x \in A\ i$) $\wedge$
($\forall S.$ *open* $S \longrightarrow x \in S \longrightarrow$ *eventually* ($\lambda i.\ A\ i \subseteq S$) *sequentially*)
proof (*safe intro*!: *exI*[*of* - F])
fix i
show *open* ($F\ i$)
using *nhds*(*1*) **by** (*auto simp*: *F-def*)
show $x \in F\ i$
using *nhds*(*2*) **by** (*auto simp*: *F-def*)
next
fix S
assume *open* S $x \in S$
from *incl*[*OF this*] **obtain** i **where** $F\ i \subseteq S$
unfolding *F-def* **by** *auto*
moreover have $\bigwedge j.\ i \leq j \Longrightarrow F\ j \subseteq F\ i$
by (*simp add*: *Inf-superset-mono F-def image-mono*)
ultimately show *eventually* ($\lambda i.\ F\ i \subseteq S$) *sequentially*
by (*auto simp*: *eventually-sequentially*)

```
  qed
qed

lemma (in first-countable-topology) nhds-countable:
  obtains X :: nat ⇒ 'a set
  where decseq X ⋀n. open (X n) ⋀n. x ∈ X n nhds x = (INF n. principal (X
n))
proof −
  from first-countable-basis obtain A :: nat ⇒ 'a set
    where ∗: ⋀n. x ∈ A n ⋀n. open (A n) ⋀S. open S ⟹ x ∈ S ⟹ ∃ i. A i ⊆
S
    by metis
  show thesis
  proof
    show decseq (λn. ⋂ i≤n. A i)
      by (simp add: antimono-iff-le-Suc atMost-Suc)
    show x ∈ (⋂ i≤n. A i) ⋀n. open (⋂ i≤n. A i) for n
      using ∗ by auto
    with ∗ show nhds x = (INF n. principal (⋂ i≤n. A i))
      unfolding nhds-def
      apply (intro INF-eq)
       apply fastforce
      apply blast
      done
  qed
qed

lemma (in first-countable-topology) countable-basis:
  obtains A :: nat ⇒ 'a set where
    ⋀i. open (A i) ⋀i. x ∈ A i
    ⋀F. (∀ n. F n ∈ A n) ⟹ F ——⟶ x
proof atomize-elim
  obtain A :: nat ⇒ 'a set where ∗:
    ⋀i. open (A i)
    ⋀i. x ∈ A i
    ⋀S. open S ⟹ x ∈ S ⟹ eventually (λi. A i ⊆ S) sequentially
    by (rule countable-basis-at-decseq) blast
  have eventually (λn. F n ∈ S) sequentially
    if ∀ n. F n ∈ A n open S x ∈ S for F S
    using ∗(3)[of S] that by (auto elim: eventually-mono simp: subset-eq)
  with ∗ show ∃ A. (∀ i. open (A i)) ∧ (∀ i. x ∈ A i) ∧ (∀ F. (∀ n. F n ∈ A n)
⟶ F ——⟶ x)
    by (intro exI[of - A]) (auto simp: tendsto-def)
qed

lemma (in first-countable-topology) sequentially-imp-eventually-nhds-within:
  assumes ∀ f. (∀ n. f n ∈ s) ∧ f ——⟶ a ⟶ eventually (λn. P (f n)) sequentially
  shows eventually P (inf (nhds a) (principal s))
proof (rule ccontr)
```

obtain $A :: nat \Rightarrow 'a\ set$ **where** $*$:
$\bigwedge i.\ open\ (A\ i)$
$\bigwedge i.\ a \in A\ i$
$\bigwedge F.\ \forall n.\ F\ n \in A\ n \Longrightarrow F \longrightarrow a$
by (*rule countable-basis*) *blast*
assume $\neg$ *?thesis*
with $*$ **have** $\exists F.\ \forall n.\ F\ n \in s \wedge F\ n \in A\ n \wedge \neg\ P\ (F\ n)$
unfolding *eventually-inf-principal eventually-nhds*
by (*intro choice*) *fastforce*
then obtain F **where** F: $\forall n.\ F\ n \in s$ **and** $\forall n.\ F\ n \in A\ n$ **and** F': $\forall n.\ \neg\ P\ (F\ n)$
by *blast*
with $*$ **have** $F \longrightarrow a$
by *auto*
then have *eventually* $(\lambda n.\ P\ (F\ n))$ *sequentially*
using *assms F* **by** *simp*
then show *False*
by (*simp add*: F')
qed

lemma (**in** *first-countable-topology*) *eventually-nhds-within-iff-sequentially*:
eventually P (*inf* (*nhds a*) (*principal s*)) $\longleftrightarrow$
$(\forall f.\ (\forall n.\ f\ n \in s) \wedge f \longrightarrow a \longrightarrow$ *eventually* $(\lambda n.\ P\ (f\ n))$ *sequentially*)
proof (*safe intro*!: *sequentially-imp-eventually-nhds-within*)
assume *eventually P* (*inf* (*nhds a*) (*principal s*))
then obtain *S* **where** *open S a* $\in S\ \forall x{\in}S.\ x \in s \longrightarrow P\ x$
by (*auto simp*: *eventually-inf-principal eventually-nhds*)
moreover
fix *f*
assume $\forall n.\ f\ n \in s\ f \longrightarrow a$
ultimately show *eventually* $(\lambda n.\ P\ (f\ n))$ *sequentially*
by (*auto dest*!: *topological-tendstoD elim*: *eventually-mono*)
qed

lemma (**in** *first-countable-topology*) *eventually-nhds-iff-sequentially*:
eventually P (*nhds a*) $\longleftrightarrow$ $(\forall f.\ f \longrightarrow a \longrightarrow$ *eventually* $(\lambda n.\ P\ (f\ n))$ *sequentially*)
using *eventually-nhds-within-iff-sequentially*[*of P a UNIV*] **by** *simp*

lemma *Inf-as-limit*:
fixes *A*::'*a*::{*linorder-topology*, *first-countable-topology*, *complete-linorder*} *set*
assumes $A \neq \{\}$
shows $\exists u.\ (\forall n.\ u\ n \in A) \wedge u \longrightarrow Inf\ A$
proof (*cases Inf A* $\in A$)
case *True*
show *?thesis*
by (*rule exI*[*of* - $\lambda n.$ *Inf A*], *auto simp add*: *True*)
next

case *False*
obtain y **where** $y \in A$ **using** *assms* **by** *auto*
then have $Inf\ A < y$ **using** *False Inf-lower less-le* **by** *auto*
obtain $F :: nat \Rightarrow 'a\ set$ **where** F: $\bigwedge i.\ open\ (F\ i)\ \bigwedge i.\ Inf\ A \in F\ i$
$\bigwedge u.\ (\forall n.\ u\ n \in F\ n) \Longrightarrow u \longrightarrow Inf\ A$
by (*metis first-countable-topology-class.countable-basis*)
define u **where** $u = (\lambda n.\ SOME\ z.\ z \in F\ n \wedge z \in A)$
have $\exists z.\ z \in U \wedge z \in A$ **if** $Inf\ A \in U\ open\ U$ **for** U
proof −
obtain b **where** $b > Inf\ A$ $\{Inf\ A\ ..<b\} \subseteq U$
using *open-right*[*OF* ‹*open U*› ‹*Inf A* ∈ *U*› ‹*Inf A* < *y*›] **by** *auto*
obtain z **where** $z < b\ z \in A$
using ‹*Inf A* < *b*› *Inf-less-iff* **by** *auto*
then have $z \in \{Inf\ A\ ..<b\}$
by (*simp add*: *Inf-lower*)
then show *?thesis* **using** ‹*z* ∈ *A*› ‹{*Inf A* ..<*b*} ⊆ *U*› **by** *auto*
qed
then have $*$: $u\ n \in F\ n \wedge u\ n \in A$ **for** n
using ‹*Inf A* ∈ *F n*› ‹*open* (*F n*)› **unfolding** *u-def* **by** (*metis* (*no-types*, *lifting*) *someI-ex*)
then have $u \longrightarrow Inf\ A$ **using** $F(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*
qed

lemma *tendsto-at-iff-sequentially*:
$(f \longrightarrow a)\ (at\ x\ within\ s) \longleftrightarrow (\forall X.\ (\forall i.\ X\ i \in s - \{x\}) \longrightarrow X \longrightarrow x \longrightarrow ((f \circ X) \longrightarrow a))$
for $f :: 'a$::*first-countable-topology* ⇒ -
unfolding *filterlim-def*[*of* - *nhds a*] *le-filter-def eventually-filtermap*
at-within-def eventually-nhds-within-iff-sequentially comp-def
by *metis*

lemma *approx-from-above-dense-linorder*:
fixes x::$'a$::{*dense-linorder*, *linorder-topology*, *first-countable-topology*}
assumes $x < y$
shows $\exists u.\ (\forall n.\ u\ n > x) \wedge (u \longrightarrow x)$
proof −
obtain $A :: nat \Rightarrow 'a\ set$ **where** A: $\bigwedge i.\ open\ (A\ i)\ \bigwedge i.\ x \in A\ i$
$\bigwedge F.\ (\forall n.\ F\ n \in A\ n) \Longrightarrow F \longrightarrow x$
by (*metis first-countable-topology-class.countable-basis*)
define u **where** $u = (\lambda n.\ SOME\ z.\ z \in A\ n \wedge z > x)$
have $\exists z.\ z \in U \wedge x < z$ **if** $x \in U\ open\ U$ **for** U
using *open-right*[*OF* ‹*open U*› ‹*x* ∈ *U*› ‹*x* < *y*›]
by (*meson atLeastLessThan-iff dense less-imp-le subset-eq*)
then have $*$: $u\ n \in A\ n \wedge x < u\ n$ **for** n
using ‹*x* ∈ *A n*› ‹*open* (*A n*)› **unfolding** *u-def* **by** (*metis* (*no-types*, *lifting*) *someI-ex*)
then have $u \longrightarrow x$ **using** $A(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*

qed

lemma *approx-from-below-dense-linorder*:
fixes x::'a::{*dense-linorder*, *linorder-topology*, *first-countable-topology*}
assumes $x > y$
shows $\exists\, u.\ (\forall\, n.\ u\ n < x) \wedge (u \longrightarrow x)$
proof −
obtain $A :: nat \Rightarrow 'a\ set$ **where** A: $\bigwedge i.\ open\ (A\ i)\ \bigwedge i.\ x \in A\ i$
$\bigwedge F.\ (\forall\, n.\ F\ n \in A\ n) \Longrightarrow F \longrightarrow x$
by (*metis first-countable-topology-class.countable-basis*)
define u **where** $u = (\lambda n.\ SOME\ z.\ z \in A\ n \wedge z < x)$
have $\exists\, z.\ z \in U \wedge z < x$ **if** $x \in U$ *open* U **for** U
using *open-left*[*OF* ‹*open* U› ‹$x \in U$› ‹$x > y$›]
by (*meson dense greaterThanAtMost-iff less-imp-le subset-eq*)
then have $*$: $u\ n \in A\ n \wedge u\ n < x$ **for** n
using ‹$x \in A\ n$› ‹*open* $(A\ n)$› **unfolding** *u-def* **by** (*metis* (*no-types*, *lifting*) *someI-ex*)
then have $u \longrightarrow x$ **using** $A(3)$ **by** *simp*
then show *?thesis* **using** $*$ **by** *auto*
qed

98.9 Function limit at a point

abbreviation *LIM* :: ('a::*topological-space* ⇒ 'b::*topological-space*) ⇒ 'a ⇒ 'b ⇒ *bool*
(‹(‹*notation*=‹*infix LIM*››(-)/ −(-)/→ (-))› [60, 0, 60] 60)
where f −a→ $L \equiv (f \longrightarrow L)\ (at\ a)$

lemma *tendsto-within-open*: $a \in S \Longrightarrow open\ S \Longrightarrow (f \longrightarrow l)\ (at\ a\ within\ S) \longleftrightarrow (f$ −a→ $l)$
by (*simp add*: *tendsto-def at-within-open*[**where** $S = S$])

lemma *tendsto-within-open-NO-MATCH*:
$a \in S \Longrightarrow NO\text{-}MATCH\ UNIV\ S \Longrightarrow open\ S \Longrightarrow (f \longrightarrow l)(at\ a\ within\ S) \longleftrightarrow (f \longrightarrow l)(at\ a)$
for f :: 'a::*topological-space* ⇒ 'b::*topological-space*
using *tendsto-within-open* **by** *blast*

lemma *LIM-const-not-eq*[*tendsto-intros*]: $k \neq L \Longrightarrow \neg\ (\lambda x.\ k)$ −a→ L
for a :: 'a::*perfect-space* **and** $k\ L$ :: 'b::*t2-space*
by (*simp add*: *tendsto-const-iff*)

lemmas *LIM-not-zero* = *LIM-const-not-eq* [**where** $L = 0$]

lemma *LIM-const-eq*: $(\lambda x.\ k)$ −a→ $L \Longrightarrow k = L$
for a :: 'a::*perfect-space* **and** $k\ L$ :: 'b::*t2-space*
by (*simp add*: *tendsto-const-iff*)

lemma *LIM-unique*: f −a→ $L \Longrightarrow f$ −a→ $M \Longrightarrow L = M$

for $a :: 'a$::*perfect-space* **and** $L\ M :: 'b$::*t2-space*
using *at-neq-bot* **by** (*rule tendsto-unique*)

lemma *LIM-Uniq*: $\exists_{\leq 1} L::'b::t2\text{-}space.\ f\ -a\to\ L$
for $a :: 'a$::*perfect-space*
by (*auto simp add*: *Uniq-def LIM-unique*)

Limits are equal for functions equal except at limit point.

lemma *LIM-equal*: $\forall x.\ x \neq a \longrightarrow f\ x = g\ x \Longrightarrow (f\ -a\to\ l) \longleftrightarrow (g\ -a\to\ l)$
by (*simp add*: *tendsto-def eventually-at-topological*)

lemma *LIM-cong*: $a = b \Longrightarrow (\bigwedge x.\ x \neq b \Longrightarrow f\ x = g\ x) \Longrightarrow l = m \Longrightarrow (f\ -a\to\ l) \longleftrightarrow (g\ -b\to\ m)$
by (*simp add*: *LIM-equal*)

lemma *tendsto-cong-limit*: $(f \longrightarrow l)\ F \Longrightarrow k = l \Longrightarrow (f \longrightarrow k)\ F$
by *simp*

lemma *tendsto-at-iff-tendsto-nhds*: $g\ -l\to\ g\ l \longleftrightarrow (g \longrightarrow g\ l)\ (nhds\ l)$
unfolding *tendsto-def eventually-at-filter*
by (*intro ext all-cong imp-cong*) (*auto elim*!: *eventually-mono*)

lemma *tendsto-compose*: $g\ -l\to\ g\ l \Longrightarrow (f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ g\ (f\ x)) \longrightarrow g\ l)\ F$
unfolding *tendsto-at-iff-tendsto-nhds* **by** (*rule filterlim-compose*[*of g*])

lemma *tendsto-compose-eventually*:
$g\ -l\to\ m \Longrightarrow (f \longrightarrow l)\ F \Longrightarrow eventually\ (\lambda x.\ f\ x \neq l)\ F \Longrightarrow ((\lambda x.\ g\ (f\ x)) \longrightarrow m)\ F$
by (*rule filterlim-compose*[*of g - at l*]) (*auto simp add*: *filterlim-at*)

lemma *LIM-compose-eventually*:
assumes $f\ -a\to\ b$
and $g\ -b\to\ c$
and $eventually\ (\lambda x.\ f\ x \neq b)\ (at\ a)$
shows $(\lambda x.\ g\ (f\ x))\ -a\to\ c$
using *assms*(*2*,*1*,*3*) **by** (*rule tendsto-compose-eventually*)

lemma *tendsto-compose-filtermap*: $((g \circ f) \longrightarrow T)\ F \longleftrightarrow (g \longrightarrow T)\ (filtermap\ f\ F)$
by (*simp add*: *filterlim-def filtermap-filtermap comp-def*)

lemma *tendsto-compose-at*:
assumes *f*: $(f \longrightarrow y)\ F$ **and** *g*: $(g \longrightarrow z)\ (at\ y)$ **and** *fg*: $eventually\ (\lambda w.\ f\ w = y \longrightarrow g\ y = z)\ F$
shows $((g \circ f) \longrightarrow z)\ F$
proof −
have $(\forall_F\ a\ in\ F.\ f\ a \neq y) \vee g\ y = z$
using *fg* **by** *force*

moreover have $(g \longrightarrow z)$ $(\mathit{filtermap}\ f\ F) \vee \neg\ (\forall_F\ a\ \mathit{in}\ F.\ f\ a \neq y)$
by (*metis* (*no-types*) *filterlim-atI filterlim-def tendsto-mono f g*)
ultimately show *?thesis*
by (*metis* (*no-types*) *f filterlim-compose filterlim-filtermap g tendsto-at-iff-tendsto-nhds tendsto-compose-filtermap*)
qed

lemma *tendsto-nhds-iff*: $(f \longrightarrow (c :: 'a :: t1\text{-}space))$ $(nhds\ x) \longleftrightarrow f\ -x\rightarrow\ c \wedge f\ x = c$
proof *safe*
assume *lim*: $(f \longrightarrow c)$ $(nhds\ x)$
show $f\ x = c$
proof (*rule ccontr*)
assume $f\ x \neq c$
hence $c \neq f\ x$
by *auto*
then obtain A **where** A: *open* A $c \in A$ $f\ x \notin A$
by (*subst* (*asm*) *separation-t1*) *auto*
with *lim* **obtain** B **where** *open* B $x \in B$ $\bigwedge x.\ x \in B \Longrightarrow f\ x \in A$
unfolding *tendsto-def eventually-nhds* **by** *metis*
with ‹$f\ x \notin A$› **show** *False*
by *blast*
qed
show $(f \longrightarrow c)$ $(at\ x)$
using *lim* **by** (*rule filterlim-mono*) (*auto simp*: *at-within-def*)
next
assume $f\ -x\rightarrow\ f\ x$ $c = f\ x$
thus $(f \longrightarrow f\ x)$ $(nhds\ x)$
unfolding *tendsto-def eventually-at-filter* **by** (*fast elim*: *eventually-mono*)
qed

98.9.1 Relation of *LIM* and *LIMSEQ*

lemma (**in** *first-countable-topology*) *sequentially-imp-eventually-within*:
$(\forall f.\ (\forall n.\ f\ n \in s \wedge f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \mathit{eventually}\ (\lambda n.\ P\ (f\ n))\ \mathit{sequentially}) \Longrightarrow$
eventually P $(at\ a\ within\ s)$
unfolding *at-within-def*
by (*intro sequentially-imp-eventually-nhds-within*) *auto*

lemma (**in** *first-countable-topology*) *sequentially-imp-eventually-at*:
$(\forall f.\ (\forall n.\ f\ n \neq a) \wedge f \longrightarrow a \longrightarrow \mathit{eventually}\ (\lambda n.\ P\ (f\ n))\ \mathit{sequentially}) \Longrightarrow$ *eventually* P $(at\ a)$
using *sequentially-imp-eventually-within* [**where** *s*=*UNIV*] **by** *simp*

lemma *LIMSEQ-SEQ-conv*:
$(\forall S.\ (\forall n.\ S\ n \neq a) \wedge S \longrightarrow a \longrightarrow (\lambda n.\ X\ (S\ n)) \longrightarrow L) \longleftrightarrow X\ -a\rightarrow\ L$ (**is** *?lhs*=*?rhs*)
for $a :: 'a$::*first-countable-topology* **and** $L :: 'b$::*topological-space*

proof
assume *?lhs* **then show** *?rhs*
by (*simp add*: *sequentially-imp-eventually-within tendsto-def*)
next
assume *?rhs* **then show** *?lhs*
using *tendsto-compose-eventually eventuallyI* **by** *blast*
qed

lemma *sequentially-imp-eventually-at-left*:
fixes a :: $'a$::{*linorder-topology*,*first-countable-topology*}
assumes *b*[*simp*]: $b < a$
and $*$: $\bigwedge f.\ (\bigwedge n.\ b < f\ n) \Longrightarrow (\bigwedge n.\ f\ n < a) \Longrightarrow$ *incseq* $f \Longrightarrow f \longrightarrow a \Longrightarrow$ *eventually* $(\lambda n.\ P\ (f\ n))$ *sequentially*
shows *eventually* P (*at-left* a)
proof (*safe intro*!: *sequentially-imp-eventually-within*)
fix X
assume X: $\forall\, n.\ X\ n \in \{..< a\} \wedge X\ n \neq a\ X \longrightarrow a$
show *eventually* $(\lambda n.\ P\ (X\ n))$ *sequentially*
proof (*rule ccontr*)
assume *neg*: $\neg$ *?thesis*
have $\exists\, s.\ \forall\, n.\ (\neg\ P\ (X\ (s\ n)) \wedge b < X\ (s\ n)) \wedge (X\ (s\ n) \leq X\ (s\ (Suc\ n)) \wedge Suc\ (s\ n) \leq s\ (Suc\ n))$
(**is** $\exists\, s.$ *?P s*)
proof (*rule dependent-nat-choice*)
have $\neg$ *eventually* $(\lambda n.\ b < X\ n \longrightarrow P\ (X\ n))$ *sequentially*
by (*intro not-eventually-impI neg order-tendstoD*(*1*) [*OF X*(*2*) *b*])
then show $\exists\, x.\ \neg\ P\ (X\ x) \wedge b < X\ x$
by (*auto dest*!: *not-eventuallyD*)
next
fix $x\ n$
have $\neg$ *eventually* $(\lambda n.\ Suc\ x \leq n \longrightarrow b < X\ n \longrightarrow X\ x < X\ n \longrightarrow P\ (X\ n))$ *sequentially*
using X
by (*intro not-eventually-impI order-tendstoD*(*1*)[*OF X*(*2*)] *eventually-ge-at-top neg*) *auto*
then show $\exists\, n.\ (\neg\ P\ (X\ n) \wedge b < X\ n) \wedge (X\ x \leq X\ n \wedge Suc\ x \leq n)$
by (*auto dest*!: *not-eventuallyD*)
qed
then obtain s **where** *?P s* **..**
with X **have** $b < X\ (s\ n)$
and $X\ (s\ n) < a$
and *incseq* $(\lambda n.\ X\ (s\ n))$
and $(\lambda n.\ X\ (s\ n)) \longrightarrow a$
and $\neg\ P\ (X\ (s\ n))$
for n
by (*auto simp*: *strict-mono-Suc-iff Suc-le-eq incseq-Suc-iff*
intro!: *LIMSEQ-subseq-LIMSEQ*[*OF* ‹$X \longrightarrow a$›, *unfolded comp-def*])
from $*$[*OF this*(*1*,*2*,*3*,*4*)] *this*(*5*) **show** *False*
by *auto*

qed
qed

lemma *tendsto-at-left-sequentially*:
fixes *a b* :: *'b*::{*linorder-topology*,*first-countable-topology*}
assumes $b < a$
assumes ∗: $\bigwedge S.\ (\bigwedge n.\ S\ n < a) \Longrightarrow (\bigwedge n.\ b < S\ n) \Longrightarrow$ *incseq S* $\Longrightarrow S \longrightarrow a \Longrightarrow$
$(\lambda n.\ X\ (S\ n)) \longrightarrow L$
shows $(X \longrightarrow L)$ (*at-left a*)
using *assms* **by** (*simp add*: *tendsto-def* [**where** *l*=*L*] *sequentially-imp-eventually-at-left*)

lemma *sequentially-imp-eventually-at-right*:
fixes *a b* :: *'a*::{*linorder-topology*,*first-countable-topology*}
assumes *b*[*simp*]: $a < b$
assumes ∗: $\bigwedge f.\ (\bigwedge n.\ a < f\ n) \Longrightarrow (\bigwedge n.\ f\ n < b) \Longrightarrow$ *decseq f* $\Longrightarrow f \longrightarrow a \Longrightarrow$
eventually $(\lambda n.\ P\ (f\ n))$ *sequentially*
shows *eventually P* (*at-right a*)
proof (*safe intro*!: *sequentially-imp-eventually-within*)
fix *X*
assume *X*: $\forall n.\ X\ n \in \{a <..\} \wedge X\ n \neq a\ X \longrightarrow a$
show *eventually* $(\lambda n.\ P\ (X\ n))$ *sequentially*
proof (*rule ccontr*)
assume *neg*: ¬ *?thesis*
have $\exists s.\ \forall n.\ (\neg\ P\ (X\ (s\ n)) \wedge X\ (s\ n) < b) \wedge (X\ (s\ (Suc\ n)) \leq X\ (s\ n) \wedge Suc\ (s\ n) \leq s\ (Suc\ n))$
(**is** $\exists s.$ *?P s*)
proof (*rule dependent-nat-choice*)
have ¬ *eventually* $(\lambda n.\ X\ n < b \longrightarrow P\ (X\ n))$ *sequentially*
by (*intro not-eventually-impI neg order-tendstoD*(*2*) [*OF X*(*2*) *b*])
then show $\exists x.\ \neg\ P\ (X\ x) \wedge X\ x < b$
by (*auto dest*!: *not-eventuallyD*)
next
fix *x n*
have ¬ *eventually* $(\lambda n.\ Suc\ x \leq n \longrightarrow X\ n < b \longrightarrow X\ n < X\ x \longrightarrow P\ (X\ n))$ *sequentially*
using *X*
by (*intro not-eventually-impI order-tendstoD*(*2*)[*OF X*(*2*)] *eventually-ge-at-top neg*) *auto*
then show $\exists n.\ (\neg\ P\ (X\ n) \wedge X\ n < b) \wedge (X\ n \leq X\ x \wedge Suc\ x \leq n)$
by (*auto dest*!: *not-eventuallyD*)
qed
then obtain *s* **where** *?P s* **..**
with *X* **have** $a < X\ (s\ n)$
and $X\ (s\ n) < b$
and *decseq* $(\lambda n.\ X\ (s\ n))$
and $(\lambda n.\ X\ (s\ n)) \longrightarrow a$
and ¬ $P\ (X\ (s\ n))$

for *n*
by (*auto simp*: *strict-mono-Suc-iff Suc-le-eq decseq-Suc-iff*
intro!: *LIMSEQ-subseq-LIMSEQ*[*OF* ‹*X* ⟶ *a*›, *unfolded comp-def*])
from ∗[*OF this*(*1*,*2*,*3*,*4*)] *this*(*5*) **show** *False*
by *auto*
qed
qed

lemma *tendsto-at-right-sequentially*:
fixes *a* :: - :: {*linorder-topology*, *first-countable-topology*}
assumes *a* < *b*
and ∗: ⋀*S*. (⋀*n*. *a* < *S n*) ⟹ (⋀*n*. *S n* < *b*) ⟹ *decseq S* ⟹ *S* ⟶ *a* ⟹
(λ*n*. *X* (*S n*)) ⟶ *L*
shows (*X* ⟶ *L*) (*at-right a*)
using *assms* **by** (*simp add*: *tendsto-def* [**where** *l*=*L*] *sequentially-imp-eventually-at-right*)

98.10 Continuity

98.10.1 Continuity on a set

definition *continuous-on* :: *'a set* ⇒ (*'a*::*topological-space* ⇒ *'b*::*topological-space*) ⇒ *bool*
where *continuous-on s f* ⟷ (∀ *x*∈*s*. (*f* ⟶ *f x*) (*at x within s*))

lemma *continuous-on-cong* [*cong*]:
s = *t* ⟹ (⋀*x*. *x* ∈ *t* ⟹ *f x* = *g x*) ⟹ *continuous-on s f* ⟷ *continuous-on t g*
unfolding *continuous-on-def*
by (*intro ball-cong filterlim-cong*) (*auto simp*: *eventually-at-filter*)

lemma *continuous-on-cong-simp*:
s = *t* ⟹ (⋀*x*. *x* ∈ *t* =*simp*=> *f x* = *g x*) ⟹ *continuous-on s f* ⟷ *continuous-on t g*
unfolding *simp-implies-def* **by** (*rule continuous-on-cong*)

lemma *continuous-on-topological*:
continuous-on s f ⟷
(∀ *x*∈*s*. ∀ *B*. *open B* ⟶ *f x* ∈ *B* ⟶ (∃ *A*. *open A* ∧ *x* ∈ *A* ∧ (∀ *y*∈*s*. *y* ∈ *A* ⟶ *f y* ∈ *B*)))
unfolding *continuous-on-def tendsto-def eventually-at-topological* **by** *metis*

lemma *continuous-on-open-invariant*:
continuous-on s f ⟷ (∀ *B*. *open B* ⟶ (∃ *A*. *open A* ∧ *A* ∩ *s* = *f* −‘ *B* ∩ *s*))
proof *safe*
fix *B* :: *'b set*
assume *continuous-on s f open B*
then have ∀ *x*∈*f* −‘ *B* ∩ *s*. (∃ *A*. *open A* ∧ *x* ∈ *A* ∧ *s* ∩ *A* ⊆ *f* −‘ *B*)
by (*auto simp*: *continuous-on-topological subset-eq Ball-def imp-conjL*)
then obtain *A* **where** ∀ *x*∈*f* −‘ *B* ∩ *s*. *open* (*A x*) ∧ *x* ∈ *A x* ∧ *s* ∩ *A x* ⊆ *f*

$-`B$
unfolding *bchoice-iff* **..**
then show $\exists A.\ open\ A \wedge A \cap s = f -` B \cap s$
by (*intro exI*[*of* - $\bigcup x{\in}f -` B \cap s.\ A\ x$]) *auto*
next
assume *B*: $\forall B.\ open\ B \longrightarrow (\exists A.\ open\ A \wedge A \cap s = f -` B \cap s)$
show *continuous-on s f*
unfolding *continuous-on-topological*
proof *safe*
fix *x B*
assume $x \in s$ *open B f* $x \in B$
with *B* **obtain** *A* **where** *A*: *open A* $A \cap s = f -` B \cap s$
by *auto*
with ‹$x \in s$› ‹$f\ x \in B$› **show** $\exists A.\ open\ A \wedge x \in A \wedge (\forall y{\in}s.\ y \in A \longrightarrow f\ y \in B)$
by (*intro exI*[*of* - *A*]) *auto*
qed
qed

lemma *continuous-on-open-vimage*:
open s $\Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ $(\forall B.\ open\ B \longrightarrow open\ (f -` B \cap s))$
unfolding *continuous-on-open-invariant*
by (*metis open-Int Int-absorb Int-commute*[*of s*] *Int-assoc*[*of* - - *s*])

corollary *continuous-imp-open-vimage*:
assumes *continuous-on s f open s open B f* $-` B \subseteq s$
shows *open* $(f -` B)$
by (*metis assms continuous-on-open-vimage le-iff-inf*)

corollary *open-vimage*[*continuous-intros*]:
assumes *open s*
and *continuous-on UNIV f*
shows *open* $(f -` s)$
using *assms* **by** (*simp add*: *continuous-on-open-vimage* [*OF open-UNIV*])

lemma *continuous-on-closed-invariant*:
continuous-on s f $\longleftrightarrow$ $(\forall B.\ closed\ B \longrightarrow (\exists A.\ closed\ A \wedge A \cap s = f -` B \cap s))$
proof −
have $*$: $(\bigwedge A.\ P\ A \longleftrightarrow Q\ (-\ A)) \Longrightarrow (\forall A.\ P\ A) \longleftrightarrow (\forall A.\ Q\ A)$
for *P Q* :: *'b set* ⇒ *bool*
by (*metis double-compl*)
show *?thesis*
unfolding *continuous-on-open-invariant*
by (*intro* $*$) (*auto simp*: *open-closed*[*symmetric*])
qed

lemma *continuous-on-closed-vimage*:
closed s $\Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ $(\forall B.\ closed\ B \longrightarrow closed\ (f -` B \cap s))$
unfolding *continuous-on-closed-invariant*

by (*metis closed-Int Int-absorb Int-commute*[*of s*] *Int-assoc*[*of - - s*])

corollary *closed-vimage-Int*[*continuous-intros*]:
assumes *closed s*
and *continuous-on t f*
and *t*: *closed t*
shows *closed* (*f* −‘ *s* ∩ *t*)
using *assms* **by** (*simp add*: *continuous-on-closed-vimage* [*OF t*])

corollary *closed-vimage*[*continuous-intros*]:
assumes *closed s*
and *continuous-on UNIV f*
shows *closed* (*f* −‘ *s*)
using *closed-vimage-Int* [*OF assms*] **by** *simp*

lemma *continuous-on-empty* [*simp*]: *continuous-on* {} *f*
by (*simp add*: *continuous-on-def*)

lemma *continuous-on-sing* [*simp*]: *continuous-on* {*x*} *f*
by (*simp add*: *continuous-on-def at-within-def*)

lemma *continuous-on-open-Union*:
(⋀*s*. *s* ∈ *S* ⟹ *open s*) ⟹ (⋀*s*. *s* ∈ *S* ⟹ *continuous-on s f*) ⟹ *continuous-on* (⋃ *S*) *f*
unfolding *continuous-on-def*
by *safe* (*metis open-Union at-within-open UnionI*)

lemma *continuous-on-open-UN*:
(⋀*s*. *s* ∈ *S* ⟹ *open* (*A s*)) ⟹ (⋀*s*. *s* ∈ *S* ⟹ *continuous-on* (*A s*) *f*) ⟹
continuous-on (⋃ *s*∈*S*. *A s*) *f*
by (*rule continuous-on-open-Union*) *auto*

lemma *continuous-on-open-Un*:
open s ⟹ *open t* ⟹ *continuous-on s f* ⟹ *continuous-on t f* ⟹ *continuous-on* (*s* ∪ *t*) *f*
using *continuous-on-open-Union* [*of* {*s*,*t*}] **by** *auto*

lemma *continuous-on-closed-Un*:
closed s ⟹ *closed t* ⟹ *continuous-on s f* ⟹ *continuous-on t f* ⟹ *continuous-on* (*s* ∪ *t*) *f*
by (*auto simp add*: *continuous-on-closed-vimage closed-Un Int-Un-distrib*)

lemma *continuous-on-closed-Union*:
assumes *finite I*
⋀*i*. *i* ∈ *I* ⟹ *closed* (*U i*)
⋀*i*. *i* ∈ *I* ⟹ *continuous-on* (*U i*) *f*
shows *continuous-on* (⋃ *i* ∈ *I*. *U i*) *f*
using *assms*
by (*induction I*) (*auto intro*!: *continuous-on-closed-Un*)

lemma *continuous-on-If*:
assumes *closed*: *closed s closed t*
and *cont*: *continuous-on s f continuous-on t g*
and *P*: $\bigwedge x.\ x \in s \Longrightarrow \neg\ P\ x \Longrightarrow f\ x = g\ x\ \bigwedge x.\ x \in t \Longrightarrow P\ x \Longrightarrow f\ x = g\ x$
shows *continuous-on* $(s \cup t)$ $(\lambda x.$ *if P x then f x else g x*$)$
(**is** *continuous-on - ?h*)
proof−
from *P* **have** $\forall\, x{\in}s.\ f\ x =$ *?h x* $\forall\, x{\in}t.\ g\ x =$ *?h x*
by *auto*
with *cont* **have** *continuous-on s ?h continuous-on t ?h*
by *simp-all*
with *closed* **show** *?thesis*
by (*rule continuous-on-closed-Un*)
qed

lemma *continuous-on-cases*:
closed s $\Longrightarrow$ *closed t* $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$ *continuous-on t g* $\Longrightarrow$
$\forall\, x.\ (x{\in}s \wedge \neg\ P\ x) \vee (x \in t \wedge P\ x) \longrightarrow f\ x = g\ x \Longrightarrow$
continuous-on $(s \cup t)$ $(\lambda x.$ *if P x then f x else g x*$)$
by (*rule continuous-on-If*) *auto*

lemma *continuous-on-id*[*continuous-intros*,*simp*]: *continuous-on s* $(\lambda x.\ x)$
unfolding *continuous-on-def* **by** *fast*

lemma *continuous-on-id'*[*continuous-intros*,*simp*]: *continuous-on s id*
unfolding *continuous-on-def id-def* **by** *fast*

lemma *continuous-on-const*[*continuous-intros*,*simp*]: *continuous-on s* $(\lambda x.\ c)$
unfolding *continuous-on-def* **by** *auto*

lemma *continuous-on-subset*: *continuous-on s f* $\Longrightarrow$ $t \subseteq s \Longrightarrow$ *continuous-on t f*
unfolding *continuous-on-def*
by (*metis subset-eq tendsto-within-subset*)

lemma *continuous-on-compose*[*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on* $(f\ `\ s)$ *g* $\Longrightarrow$ *continuous-on s* $(g \circ f)$
unfolding *continuous-on-topological* **by** *simp metis*

lemma *continuous-on-compose2*:
continuous-on t g $\Longrightarrow$ *continuous-on s f* $\Longrightarrow$ $f\ `\ s \subseteq t \Longrightarrow$ *continuous-on s* $(\lambda x.$ *g* $(f\ x))$
using *continuous-on-compose*[*of s f g*] *continuous-on-subset* **by** (*force simp add*: *comp-def*)

lemma *continuous-on-generate-topology*:
assumes $*$: *open* $=$ *generate-topology X*
and $**$: $\bigwedge B.\ B \in X \Longrightarrow \exists\, C.\ open\ C \wedge C \cap A = f\ -`\ B \cap A$
shows *continuous-on A f*

```
  unfolding continuous-on-open-invariant
proof safe
  fix B :: 'a set
  assume open B
  then show ∃C. open C ∧ C ∩ A = f -` B ∩ A
    unfolding *
  proof induct
    case (UN K)
    then obtain C where ⋀k. k ∈ K ⟹ open (C k) ⋀k. k ∈ K ⟹ C k ∩ A =
f -` k ∩ A
      by metis
    then show ?case
      by (intro exI[of - ⋃k∈K. C k]) blast
  qed (auto intro: **)
qed

lemma continuous-onI-mono:
  fixes f :: 'a::linorder-topology ⇒ 'b::{dense-order,linorder-topology}
  assumes open (f`A)
    and mono: ⋀x y. x ∈ A ⟹ y ∈ A ⟹ x ≤ y ⟹ f x ≤ f y
  shows continuous-on A f
proof (rule continuous-on-generate-topology[OF open-generated-order], safe)
  have monoD: ⋀x y. x ∈ A ⟹ y ∈ A ⟹ f x < f y ⟹ x < y
    by (auto simp: not-le[symmetric] mono)
  have ∃x. x ∈ A ∧ f x < b ∧ a < x if a: a ∈ A and fa: f a < b for a b
  proof -
    obtain y where f a < y {f a ..< y} ⊆ f`A
      using open-right[OF ‹open (f`A)›, of f a b] a fa
      by auto
    obtain z where z: f a < z z < min b y
      using dense[of f a min b y] ‹f a < y› ‹f a < b› by auto
    then obtain c where z = f c c ∈ A
      using ‹{f a ..< y} ⊆ f`A›[THEN subsetD, of z] by (auto simp: less-imp-le)
    with a z show ?thesis
      by (auto intro!: exI[of - c] simp: monoD)
  qed
  then show ∃C. open C ∧ C ∩ A = f -` {..<b} ∩ A for b
    by (intro exI[of - (⋃x∈{x∈A. f x < b}. {..< x})])
       (auto intro: le-less-trans[OF mono] less-imp-le)

  have ∃x. x ∈ A ∧ b < f x ∧ x < a if a: a ∈ A and fa: b < f a for a b
  proof -
    note a fa
    moreover
    obtain y where y < f a {y <.. f a} ⊆ f`A
      using open-left[OF ‹open (f`A)›, of f a b]  a fa
      by auto
    then obtain z where z: max b y < z z < f a
      using dense[of max b y f a] ‹y < f a› ‹b < f a› by auto
```

then obtain c **where** $z = f\ c\ c \in A$
using ‹$\{y <.. f\ a\} \subseteq f`A$›[*THEN subsetD*, *of z*] **by** (*auto simp*: *less-imp-le*)
with $a\ z$ **show** *?thesis*
by (*auto intro*!: *exI*[*of* - c] *simp*: *monoD*)
qed
then show $\exists C.$ *open* $C \wedge C \cap A = f -` \{b <..\} \cap A$ **for** b
by (*intro exI*[*of* - $(\bigcup x{\in}\{x{\in}A.\ b < f\ x\}.\ \{x <..\})$])
(*auto intro*: *less-le-trans*[*OF* - *mono*] *less-imp-le*)
qed

lemma *continuous-on-IccI*:
⟦$(f \longrightarrow f\ a)$ (*at-right* a);
$(f \longrightarrow f\ b)$ (*at-left* b);
$(\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow f \mathrel{-x\to} f\ x);\ a < b$⟧ ⟹
continuous-on $\{a\ ..\ b\}\ f$
for a::$'a$::*linorder-topology*
using *at-within-open*[*of* - $\{a<..<b\}$]
by (*auto simp*: *continuous-on-def at-within-Icc-at-right at-within-Icc-at-left le-less*
at-within-Icc-at)

lemma
fixes $a\ b$::$'a$::*linorder-topology*
assumes *continuous-on* $\{a\ ..\ b\}\ f\ a < b$
shows *continuous-on-Icc-at-rightD*: $(f \longrightarrow f\ a)$ (*at-right* a)
and *continuous-on-Icc-at-leftD*: $(f \longrightarrow f\ b)$ (*at-left* b)
using *assms*
by (*auto simp*: *at-within-Icc-at-right at-within-Icc-at-left continuous-on-def*
dest: *bspec*[**where** $x{=}a$] *bspec*[**where** $x{=}b$])

lemma *continuous-on-discrete* [*simp*]:
continuous-on A $(f :: 'a :: discrete\text{-}topology \Rightarrow -)$
by (*auto simp*: *continuous-on-def at-discrete*)

lemma *continuous-on-of-nat* [*continuous-intros*]:
assumes *continuous-on* $A\ f$
shows *continuous-on* A $(\lambda n.\ of\text{-}nat\ (f\ n))$
using *continuous-on-compose*[*OF assms continuous-on-discrete*[*of* - *of-nat*]]
by (*simp add*: *o-def*)

lemma *continuous-on-of-int* [*continuous-intros*]:
assumes *continuous-on* $A\ f$
shows *continuous-on* A $(\lambda n.\ of\text{-}int\ (f\ n))$
using *continuous-on-compose*[*OF assms continuous-on-discrete*[*of* - *of-int*]]
by (*simp add*: *o-def*)

98.10.2 Continuity at a point

definition *continuous* :: $'a$::*t2-space filter* ⇒ ($'a$ ⇒ $'b$::*topological-space*) ⇒ *bool*
where *continuous* $F\ f \longleftrightarrow (f \longrightarrow f\ (Lim\ F\ (\lambda x.\ x)))\ F$

lemma *continuous-bot*[*continuous-intros*, *simp*]: *continuous bot f*
unfolding *continuous-def* **by** *auto*

lemma *continuous-trivial-limit*: *trivial-limit net* $\Longrightarrow$ *continuous net f*
by *simp*

lemma *continuous-within*: *continuous* (*at x within s*) *f* $\longleftrightarrow$ (*f* $\longrightarrow$ *f x*) (*at x within s*)
by (*cases trivial-limit* (*at x within s*)) (*auto simp add*: *Lim-ident-at continuous-def*)

lemma *continuous-within-topological*:
continuous (*at x within s*) *f* $\longleftrightarrow$
(∀ *B*. *open B* $\longrightarrow$ *f x* ∈ *B* $\longrightarrow$ (∃ *A*. *open A* ∧ *x* ∈ *A* ∧ (∀ *y*∈*s*. *y* ∈ *A* $\longrightarrow$ *f y* ∈ *B*)))
unfolding *continuous-within tendsto-def eventually-at-topological* **by** *metis*

lemma *continuous-within-compose*[*continuous-intros*]:
continuous (*at x within s*) *f* $\Longrightarrow$ *continuous* (*at* (*f x*) *within f* ‘ *s*) *g* $\Longrightarrow$
continuous (*at x within s*) (*g* ∘ *f*)
by (*simp add*: *continuous-within-topological*) *metis*

lemma *continuous-within-compose2*:
continuous (*at x within s*) *f* $\Longrightarrow$ *continuous* (*at* (*f x*) *within f* ‘ *s*) *g* $\Longrightarrow$
continuous (*at x within s*) (λ*x*. *g* (*f x*))
using *continuous-within-compose*[*of x s f g*] **by** (*simp add*: *comp-def*)

lemma *continuous-at*: *continuous* (*at x*) *f* $\longleftrightarrow$ *f* −*x*→ *f x*
using *continuous-within*[*of x UNIV f*] **by** *simp*

lemma *continuous-ident*[*continuous-intros*, *simp*]: *continuous* (*at x within S*) (λ*x*. *x*)
unfolding *continuous-within* **by** (*rule tendsto-ident-at*)

lemma *continuous-id*[*continuous-intros*, *simp*]: *continuous* (*at x within S*) *id*
by (*simp add*: *id-def*)

lemma *continuous-const*[*continuous-intros*, *simp*]: *continuous F* (λ*x*. *c*)
unfolding *continuous-def* **by** (*rule tendsto-const*)

lemma *continuous-on-eq-continuous-within*:
continuous-on s f $\longleftrightarrow$ (∀ *x*∈*s*. *continuous* (*at x within s*) *f*)
unfolding *continuous-on-def continuous-within* **..**

lemma *continuous-discrete* [*simp*]:
continuous (*at x within A*) (*f* :: *'a* :: *discrete-topology* ⇒ -)
by (*auto simp*: *continuous-def at-discrete*)

lemma
fixes *f*::*'a*::*linorder-topology* ⇒ *'b*::*topological-space*
assumes *continuous-on* {*a*..*b*} *f* *a*<*b*
shows *continuous-on-at-left*: *continuous* (*at-left b*) *f*
and *continuous-on-at-right*: *continuous* (*at-right a*) *f*
using *assms continuous-on-Icc-at-leftD continuous-on-Icc-at-rightD continuous-within*

by *blast*+

Continuity in terms of open preimages.

lemma *continuous-at-open*:
continuous (*at x*) *f* ⟷ (∀ *t*. *open t* ∧ *f x* ∈ *t* ⟶ (∃ *S*. *open S* ∧ *x* ∈ *S* ∧ (∀ *x'* ∈ *S*. (*f x'*) ∈ *t*)))
by (*metis UNIV-I continuous-within-topological*)

lemma *continuous-imp-tendsto*:
assumes *continuous* (*at x0*) *f* **and** *x* ⟶⟶ *x0*
shows (*f* ∘ *x*) ⟶⟶ (*f x0*)
proof (*rule topological-tendstoI*)
fix *S*
assume *open S f x0* ∈ *S*
then obtain *T* **where** *T-def*: *open T x0* ∈ *T* ∀ *x*∈*T*. *f x* ∈ *S*
using *assms continuous-at-open* **by** *metis*
then have *eventually* (λ*n*. *x n* ∈ *T*) *sequentially*
using *assms T-def* **by** (*auto simp*: *tendsto-def*)
then show *eventually* (λ*n*. (*f* ∘ *x*) *n* ∈ *S*) *sequentially*
using *T-def* **by** (*auto elim*!: *eventually-mono*)
qed

abbreviation *isCont* :: (*'a*::*t2-space* ⇒ *'b*::*topological-space*) ⇒ *'a* ⇒ *bool*
where *isCont f a* ≡ *continuous* (*at a*) *f*

lemma *isCont-def*: *isCont f a* ⟷ *f* −*a*→ *f a*
by (*rule continuous-at*)

lemma *isContD*: *isCont f x* ⟹ *f* −*x*→ *f x*
by (*simp add*: *isCont-def*)

lemma *isCont-cong*:
assumes *eventually* (λ*x*. *f x* = *g x*) (*nhds x*)
shows *isCont f x* ⟷ *isCont g x*
proof −
from *assms* **have** [*simp*]: *f x* = *g x*
by (*rule eventually-nhds-x-imp-x*)
from *assms* **have** *eventually* (λ*x*. *f x* = *g x*) (*at x*)
by (*auto simp*: *eventually-at-filter elim*!: *eventually-mono*)
with *assms* **have** *isCont f x* ⟷ *isCont g x* **unfolding** *isCont-def*
by (*intro filterlim-cong*) (*auto elim*!: *eventually-mono*)
with *assms* **show** *?thesis* **by** *simp*

qed

lemma *continuous-at-imp-continuous-at-within*: *isCont f x* $\Longrightarrow$ *continuous* (*at x within s*) *f*
by (*auto intro*: *tendsto-mono at-le simp*: *continuous-at continuous-within*)

lemma *continuous-on-eq-continuous-at*: *open s* $\Longrightarrow$ *continuous-on s f* $\longleftrightarrow$ ($\forall x{\in}s$. *isCont f x*)
by (*simp add*: *continuous-on-def continuous-at at-within-open*[*of - s*])

lemma *continuous-within-open*: $a \in A$ $\Longrightarrow$ *open A* $\Longrightarrow$ *continuous* (*at a within A*) *f* $\longleftrightarrow$ *isCont f a*
by (*simp add*: *at-within-open-NO-MATCH*)

lemma *continuous-at-imp-continuous-on*: $\forall x{\in}s$. *isCont f x* $\Longrightarrow$ *continuous-on s f*
by (*auto intro*: *continuous-at-imp-continuous-at-within simp*: *continuous-on-eq-continuous-within*)

lemma *isCont-o2*: *isCont f a* $\Longrightarrow$ *isCont g* (*f a*) $\Longrightarrow$ *isCont* ($\lambda x.$ *g* (*f x*)) *a*
unfolding *isCont-def* **by** (*rule tendsto-compose*)

lemma *continuous-at-compose*[*continuous-intros*]: *isCont f a* $\Longrightarrow$ *isCont g* (*f a*) $\Longrightarrow$ *isCont* (*g* $\circ$ *f*) *a*
unfolding *o-def* **by** (*rule isCont-o2*)

lemma *isCont-tendsto-compose*: *isCont g l* $\Longrightarrow$ (*f* $\longrightarrow$ *l*) *F* $\Longrightarrow$ ((λ*x.* *g* (*f x*)) $\longrightarrow$ *g l*) *F*
unfolding *isCont-def* **by** (*rule tendsto-compose*)

lemma *continuous-on-tendsto-compose*:
assumes *f-cont*: *continuous-on s f*
and *g*: (*g* $\longrightarrow$ *l*) *F*
and *l*: *l* $\in$ *s*
and *ev*: $\forall_F x$ *in F*. *g x* $\in$ *s*
shows ((λ*x.* *f* (*g x*)) $\longrightarrow$ *f l*) *F*
proof −
from *f-cont l* **have** *f*: (*f* $\longrightarrow$ *f l*) (*at l within s*)
by (*simp add*: *continuous-on-def*)
have *i*: ((λ*x.* *if g x* = *l then f l else f* (*g x*)) $\longrightarrow$ *f l*) *F*
by (*rule filterlim-If*)
(*auto intro*!: *filterlim-compose*[*OF f*] *eventually-conj tendsto-mono*[*OF - g*]
simp: *filterlim-at eventually-inf-principal eventually-mono*[*OF ev*])
show *?thesis*
by (*rule filterlim-cong*[*THEN iffD1*[*OF - i*]]) *auto*
qed

lemma *continuous-within-compose3*:
isCont g (*f x*) $\Longrightarrow$ *continuous* (*at x within s*) *f* $\Longrightarrow$ *continuous* (*at x within s*) (λ*x.* *g* (*f x*))
using *continuous-at-imp-continuous-at-within continuous-within-compose2* **by**

blast

lemma *at-within-isCont-imp-nhds*:
fixes f:: $'a$:: {*t2-space*,*perfect-space*} $\Rightarrow$ $'b$:: *t2-space*
assumes $\forall_F\ w$ *in at* z. $f\ w = g\ w$ *isCont* $f\ z$ *isCont* $g\ z$
shows $\forall_F\ w$ *in nhds* z. $f\ w = g\ w$
proof −
have $g \longrightarrow z \rightarrow f\ z$
using *assms isContD tendsto-cong* **by** *blast*
moreover have $g \longrightarrow z \rightarrow g\ z$ **using** ‹*isCont g z*› **using** *isCont-def* **by** *blast*
ultimately have $f\ z{=}g\ z$ **using** *LIM-unique* **by** *auto*
moreover have $\forall_F\ x$ *in nhds* z. $x \neq z \longrightarrow f\ x = g\ x$
using *assms* **unfolding** *eventually-at-filter* **by** *auto*
ultimately show *?thesis*
by (*auto elim:eventually-mono*)
qed

lemma *filtermap-nhds-open-map′*:
assumes *cont*: *isCont f a*
and *open A a* $\in$ *A*
and *open-map*: $\bigwedge S$. *open S* $\Longrightarrow$ $S \subseteq A \Longrightarrow$ *open* (*f* ‘ *S*)
shows *filtermap f* (*nhds a*) = *nhds* (*f a*)
unfolding *filter-eq-iff*
proof *safe*
fix *P*
assume *eventually P* (*filtermap f* (*nhds a*))
then obtain *S* **where** *S*: *open S a* $\in$ *S* $\forall x{\in}S$. *P* (*f x*)
by (*auto simp*: *eventually-filtermap eventually-nhds*)
show *eventually P* (*nhds* (*f a*))
unfolding *eventually-nhds*
proof (*rule exI* [*of* - *f* ‘ (*A* $\cap$ *S*)], *safe*)
show *open* (*f* ‘ (*A* $\cap$ *S*))
using *S* **by** (*intro open-Int assms*) *auto*
show *f a* $\in$ *f* ‘ (*A* $\cap$ *S*)
using *assms S* **by** *auto*
show *P* (*f x*) **if** *x* $\in$ *A x* $\in$ *S* **for** *x*
using *S that* **by** *auto*
qed
qed (*metis filterlim-iff tendsto-at-iff-tendsto-nhds isCont-def eventually-filtermap cont*)

lemma *filtermap-nhds-open-map*:
assumes *cont*: *isCont f a*
and *open-map*: $\bigwedge S$. *open S* $\Longrightarrow$ *open* (*f*‘*S*)
shows *filtermap f* (*nhds a*) = *nhds* (*f a*)
using *cont filtermap-nhds-open-map′ open-map* **by** *blast*

lemma *continuous-at-split*:
continuous (*at x*) *f* $\longleftrightarrow$ *continuous* (*at-left x*) *f* $\wedge$ *continuous* (*at-right x*) *f*

for $x :: 'a$::*linorder-topology*
by (*simp add*: *continuous-within filterlim-at-split*)

lemma *continuous-on-max* [*continuous-intros*]:
fixes $f\ g :: 'a$::*topological-space* $\Rightarrow$ $'b$::*linorder-topology*
shows *continuous-on A f* $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ *continuous-on A* ($\lambda x.$ *max* (*f x*) (*g x*))
by (*auto simp*: *continuous-on-def intro*!: *tendsto-max*)

lemma *continuous-on-min* [*continuous-intros*]:
fixes $f\ g :: 'a$::*topological-space* $\Rightarrow$ $'b$::*linorder-topology*
shows *continuous-on A f* $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ *continuous-on A* ($\lambda x.$ *min* (*f x*) (*g x*))
by (*auto simp*: *continuous-on-def intro*!: *tendsto-min*)

lemma *continuous-max* [*continuous-intros*]:
fixes $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*linorder-topology*
shows $[\![$*continuous F f*; *continuous F g*$]\!] \Longrightarrow$ *continuous F* ($\lambda x.$ (*max* (*f x*) (*g x*)))
by (*simp add*: *tendsto-max continuous-def*)

lemma *continuous-min* [*continuous-intros*]:
fixes $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*linorder-topology*
shows $[\![$*continuous F f*; *continuous F g*$]\!] \Longrightarrow$ *continuous F* ($\lambda x.$ (*min* (*f x*) (*g x*)))
by (*simp add*: *tendsto-min continuous-def*)

The following open/closed Collect lemmas are ported from Sébastien Gouëzel's *Ergodic-Theory*.

lemma *open-Collect-neq*:
fixes $f\ g :: 'a$::*topological-space* $\Rightarrow$ $'b$::*t2-space*
assumes *f*: *continuous-on UNIV f* **and** *g*: *continuous-on UNIV g*
shows *open* $\{x.\ f\ x \neq g\ x\}$
proof (*rule openI*)
fix *t*
assume $t \in \{x.\ f\ x \neq g\ x\}$
then obtain *U V* **where** $*$: *open U open V f t* $\in$ *U g t* $\in$ *V U* $\cap$ *V* $= \{\}$
by (*auto simp add*: *separation-t2*)
with *open-vimage*[*OF* ‹*open U*› *f*] *open-vimage*[*OF* ‹*open V*› *g*]
show $\exists$ *T*. *open T* $\wedge$ *t* $\in$ *T* $\wedge$ *T* $\subseteq \{x.\ f\ x \neq g\ x\}$
by (*intro exI*[*of - f* $-$‘ *U* $\cap$ *g* $-$‘ *V*]) *auto*
qed

lemma *closed-Collect-eq*:
fixes $f\ g :: 'a$::*topological-space* $\Rightarrow$ $'b$::*t2-space*
assumes *f*: *continuous-on UNIV f* **and** *g*: *continuous-on UNIV g*
shows *closed* $\{x.\ f\ x = g\ x\}$
using *open-Collect-neq*[*OF f g*] **by** (*simp add*: *closed-def Collect-neg-eq*)

lemma *open-Collect-less*:

fixes *f g* :: *'a*::*topological-space* ⇒ *'b*::*linorder-topology*
assumes *f*: *continuous-on UNIV f* **and** *g*: *continuous-on UNIV g*
shows *open* {*x*. *f x* < *g x*}
proof (*rule openI*)
fix *t*
assume *t*: *t* ∈ {*x*. *f x* < *g x*}
show ∃ *T*. *open T* ∧ *t* ∈ *T* ∧ *T* ⊆ {*x*. *f x* < *g x*}
proof (*cases* ∃ *z*. *f t* < *z* ∧ *z* < *g t*)
case *True*
then obtain *z* **where** *f t* < *z* ∧ *z* < *g t* **by** *blast*
then show *?thesis*
using *open-vimage*[*OF - f*, *of* {..< *z*}] *open-vimage*[*OF - g*, *of* {*z* <..}]
by (*intro exI*[*of - f* −‘ {..<*z*} ∩ *g* −‘ {*z*<..}]) *auto*
next
case *False*
then have ∗: {*g t* ..} = {*f t* <..} {..< *g t*} = {.. *f t*}
using *t* **by** (*auto intro*: *leI*)
show *?thesis*
using *open-vimage*[*OF - f*, *of* {..< *g t*}] *open-vimage*[*OF - g*, *of* {*f t* <..}] *t*
apply (*intro exI*[*of - f* −‘ {..< *g t*} ∩ *g* −‘ {*f t*<..}])
apply (*simp add*: *open-Int*)
apply (*auto simp add*: ∗)
done
qed
qed

lemma *closed-Collect-le*:
fixes *f g* :: *'a* :: *topological-space* ⇒ *'b*::*linorder-topology*
assumes *f*: *continuous-on UNIV f*
and *g*: *continuous-on UNIV g*
shows *closed* {*x*. *f x* ≤ *g x*}
using *open-Collect-less* [*OF g f*]
by (*simp add*: *closed-def Collect-neg-eq*[*symmetric*] *not-le*)

98.10.3 Open-cover compactness

context *topological-space*
begin

definition *compact* :: *'a set* ⇒ *bool* **where**
compact-eq-Heine-Borel:
compact S ⟷ (∀ *C*. (∀ *c*∈*C*. *open c*) ∧ *S* ⊆ ⋃ *C* ⟶ (∃ *D*⊆*C*. *finite D* ∧ *S* ⊆ ⋃ *D*))

lemma *compactI*:
assumes ⋀*C*. ∀ *t*∈*C*. *open t* ⟹ *s* ⊆ ⋃ *C* ⟹ ∃ *C'*. *C'* ⊆ *C* ∧ *finite C'* ∧ *s* ⊆ ⋃ *C'*
shows *compact s*
unfolding *compact-eq-Heine-Borel* **using** *assms* **by** *metis*

lemma *compact-empty*[*simp*]: *compact* {}
by (*auto intro*!: *compactI*)

lemma *compactE*:
assumes *compact S* $S \subseteq \bigcup \mathcal{T}$ $\bigwedge B.\ B \in \mathcal{T} \Longrightarrow$ *open B*
obtains $\mathcal{T}'$ **where** $\mathcal{T}' \subseteq \mathcal{T}$ *finite* $\mathcal{T}'$ $S \subseteq \bigcup \mathcal{T}'$
by (*meson assms compact-eq-Heine-Borel*)

lemma *compactE-image*:
assumes *compact S*
and *opn*: $\bigwedge T.\ T \in C \Longrightarrow$ *open* (*f T*)
and *S*: $S \subseteq (\bigcup c \in C.\ f\ c)$
obtains C' **where** $C' \subseteq C$ **and** *finite* C' **and** $S \subseteq (\bigcup c \in C'.\ f\ c)$
apply (*rule compactE*[*OF* ‹*compact S*› *S*])
using *opn* **apply** *force*
by (*metis finite-subset-image*)

lemma *compact-Int-closed* [*intro*]:
assumes *compact S*
and *closed T*
shows *compact* ($S \cap T$)
proof (*rule compactI*)
fix *C*
assume *C*: $\forall\, c \in C.$ *open c*
assume *cover*: $S \cap T \subseteq \bigcup C$
from *C* ‹*closed T*› **have** $\forall\, c \in C \cup \{-\ T\}.$ *open c*
by *auto*
moreover from *cover* **have** $S \subseteq \bigcup (C \cup \{-\ T\})$
by *auto*
ultimately have $\exists\, D \subseteq C \cup \{-\ T\}.$ *finite D* $\wedge$ $S \subseteq \bigcup D$
using ‹*compact S*› **unfolding** *compact-eq-Heine-Borel* **by** *auto*
then obtain *D* **where** $D \subseteq C \cup \{-\ T\}$ $\wedge$ *finite D* $\wedge$ $S \subseteq \bigcup D$ **..**
then show $\exists\, D \subseteq C.$ *finite D* $\wedge$ $S \cap T \subseteq \bigcup D$
by (*intro exI*[*of* - *D* − {−*T*}]) *auto*
qed

lemma *compact-diff*: ⟦*compact S*; *open T*⟧ $\Longrightarrow$ *compact*($S - T$)
by (*simp add*: *Diff-eq compact-Int-closed open-closed*)

lemma *inj-setminus*: *inj-on uminus* (*A*::'*a set set*)
by (*auto simp*: *inj-on-def*)

98.11 Finite intersection property

lemma *compact-fip*:
compact U ⟷
($\forall A.$ ($\forall\, a \in A.$ *closed a*) ⟶ ($\forall\, B \subseteq A.$ *finite B* ⟶ $U \cap \bigcap B \neq \{\}$) ⟶ $U \cap \bigcap A \neq \{\}$)

(**is** - $\longleftrightarrow$ *?R*)
proof (*safe intro*!: *compact-eq-Heine-Borel*[*THEN iffD2*])
fix *A*
assume *compact U*
assume *A*: $\forall\, a{\in}A.$ *closed a* $U \cap \bigcap A = \{\}$
assume *fin*: $\forall\, B \subseteq A.$ *finite B* $\longrightarrow$ $U \cap \bigcap B \neq \{\}$
from *A* **have** $(\forall\, a{\in}$*uminus*‘*A*. *open a*$) \wedge U \subseteq \bigcup$(*uminus*‘*A*)
by *auto*
with ‹*compact U*› **obtain** *B* **where** $B \subseteq A$ *finite* (*uminus*‘*B*) $U \subseteq \bigcup$(*uminus*‘*B*)
unfolding *compact-eq-Heine-Borel* **by** (*metis subset-image-iff*)
with *fin*[*THEN spec*, *of B*] **show** *False*
by (*auto dest*: *finite-imageD intro*: *inj-setminus*)
next
fix *A*
assume *?R*
assume $\forall\, a{\in}A.$ *open a* $U \subseteq \bigcup A$
then have $U \cap \bigcap$(*uminus*‘*A*) $= \{\}$ $\forall\, a{\in}$*uminus*‘*A*. *closed a*
by *auto*
with ‹*?R*› **obtain** *B* **where** $B \subseteq A$ *finite* (*uminus*‘*B*) $U \cap \bigcap$(*uminus*‘*B*) $= \{\}$
by (*metis subset-image-iff*)
then show $\exists\, T{\subseteq}A.$ *finite* $T \wedge U \subseteq \bigcup T$
by (*auto intro*!: *exI*[*of* - *B*] *inj-setminus dest*: *finite-imageD*)
qed

lemma *compact-imp-fip*:
assumes *compact S*
and $\bigwedge T.$ $T \in F \Longrightarrow$ *closed T*
and $\bigwedge F'.$ *finite* $F' \Longrightarrow F' \subseteq F \Longrightarrow S \cap (\bigcap F') \neq \{\}$
shows $S \cap (\bigcap F) \neq \{\}$
using *assms* **unfolding** *compact-fip* **by** *auto*

lemma *compact-imp-fip-image*:
assumes *compact s*
and *P*: $\bigwedge i.$ $i \in I \Longrightarrow$ *closed* (*f i*)
and *Q*: $\bigwedge I'.$ *finite* $I' \Longrightarrow I' \subseteq I \Longrightarrow (s \cap (\bigcap i{\in}I'.\ f\ i) \neq \{\})$
shows $s \cap (\bigcap i{\in}I.\ f\ i) \neq \{\}$
proof −
from *P* **have** $\forall\, i \in f\ `\ I.$ *closed i*
by *blast*
moreover have $\forall\, A.$ *finite* $A \wedge A \subseteq f\ `\ I \longrightarrow (s \cap (\bigcap A) \neq \{\})$
by (*metis Q finite-subset-image*)
ultimately show $s \cap (\bigcap (f\ `\ I)) \neq \{\}$
by (*metis* ‹*compact s*› *compact-imp-fip*)
qed

end

lemma (**in** *t2-space*) *compact-imp-closed*:
assumes *compact s*

shows *closed s*
unfolding *closed-def*
proof (*rule openI*)
fix *y*
assume $y \in - s$
let *?C* = $\bigcup x{\in}s.$ {*u. open u* $\wedge$ $x \in u$ $\wedge$ *eventually* ($\lambda y.\ y \notin u$) (*nhds y*)}
have $s \subseteq \bigcup$ *?C*
proof
fix *x*
assume $x \in s$
with ‹$y \in - s$› **have** $x \neq y$ **by** *clarsimp*
then have $\exists u\ v.$ *open u* $\wedge$ *open v* $\wedge$ $x \in u$ $\wedge$ $y \in v$ $\wedge$ $u \cap v = \{\}$
by (*rule hausdorff*)
with ‹$x \in s$› **show** $x \in \bigcup$ *?C*
unfolding *eventually-nhds* **by** *auto*
qed
then obtain *D* **where** $D \subseteq$ *?C* **and** *finite D* **and** $s \subseteq \bigcup D$
by (*rule compactE* [*OF* ‹*compact s*›]) *auto*
from ‹$D \subseteq$ *?C*› **have** $\forall x{\in}D.$ *eventually* ($\lambda y.\ y \notin x$) (*nhds y*)
by *auto*
with ‹*finite D*› **have** *eventually* ($\lambda y.\ y \notin \bigcup D$) (*nhds y*)
by (*simp add*: *eventually-ball-finite*)
with ‹$s \subseteq \bigcup D$› **have** *eventually* ($\lambda y.\ y \notin s$) (*nhds y*)
by (*auto elim*!: *eventually-mono*)
then show $\exists t.$ *open t* $\wedge$ $y \in t$ $\wedge$ $t \subseteq - s$
by (*simp add*: *eventually-nhds subset-eq*)
qed

lemma *compact-continuous-image*:
assumes *f*: *continuous-on s f*
and *s*: *compact s*
shows *compact* (*f* ‘ *s*)
proof (*rule compactI*)
fix *C*
assume $\forall\, c{\in}C.$ *open c* **and** *cover*: *f*‘*s* $\subseteq \bigcup C$
with *f* **have** $\forall\, c{\in}C.\ \exists A.$ *open A* $\wedge$ $A \cap s = f -$‘ $c \cap s$
unfolding *continuous-on-open-invariant* **by** *blast*
then obtain *A* **where** *A*: $\forall\, c{\in}C.$ *open* (*A c*) $\wedge$ $A\ c \cap s = f -$‘ $c \cap s$
unfolding *bchoice-iff* **..**
with *cover* **have** $\bigwedge c.\ c \in C \Longrightarrow$ *open* (*A c*) $s \subseteq (\bigcup c{\in}C.\ A\ c)$
by (*fastforce simp add*: *subset-eq set-eq-iff*)+
from *compactE-image*[*OF s this*] **obtain** *D* **where** $D \subseteq C$ *finite D* $s \subseteq (\bigcup c{\in}D.$ *A c*) **.**
with *A* **show** $\exists\, D \subseteq C.$ *finite D* $\wedge$ *f*‘*s* $\subseteq \bigcup D$
by (*intro exI*[*of - D*]) (*fastforce simp add*: *subset-eq set-eq-iff*)+
qed

lemma *continuous-on-inv*:
fixes *f* :: *'a*::*topological-space* $\Rightarrow$ *'b*::*t2-space*

assumes *continuous-on s f*
and *compact s*
and $\forall x{\in}s.\ g\ (f\ x) = x$
shows *continuous-on* $(f\ `\ s)\ g$
unfolding *continuous-on-topological*
proof (*clarsimp simp add*: *assms*(*3*))
fix $x :: 'a$ **and** $B :: 'a\ set$
assume $x \in s$ **and** *open B* **and** $x \in B$
have *1*: $\forall x{\in}s.\ f\ x \in f\ `\ (s - B) \longleftrightarrow x \in s - B$
using *assms*(*3*) **by** (*auto*, *metis*)
have *continuous-on* $(s - B)\ f$
using ‹*continuous-on s f*› *Diff-subset*
by (*rule continuous-on-subset*)
moreover have *compact* $(s - B)$
using ‹*open B*› **and** ‹*compact s*›
unfolding *Diff-eq* **by** (*intro compact-Int-closed closed-Compl*)
ultimately have *compact* $(f\ `\ (s - B))$
by (*rule compact-continuous-image*)
then have *closed* $(f\ `\ (s - B))$
by (*rule compact-imp-closed*)
then have *open* $(-\ f\ `\ (s - B))$
by (*rule open-Compl*)
moreover have $f\ x \in -\ f\ `\ (s - B)$
using ‹$x \in s$› **and** ‹$x \in B$› **by** (*simp add*: *1*)
moreover have $\forall y{\in}s.\ f\ y \in -\ f\ `\ (s - B) \longrightarrow y \in B$
by (*simp add*: *1*)
ultimately show $\exists A.\ open\ A \wedge f\ x \in A \wedge (\forall y{\in}s.\ f\ y \in A \longrightarrow y \in B)$
by *fast*
qed

lemma *continuous-on-inv-into*:
fixes $f :: 'a{::}topological\text{-}space \Rightarrow 'b{::}t2\text{-}space$
assumes *s*: *continuous-on s f compact s*
and *f*: *inj-on f s*
shows *continuous-on* $(f\ `\ s)$ (*the-inv-into s f*)
by (*rule continuous-on-inv*[*OF s*]) (*auto simp*: *the-inv-into-f-f*[*OF f*])

lemma (**in** *linorder-topology*) *compact-attains-sup*:
assumes *compact S* $S \neq \{\}$
shows $\exists s{\in}S.\ \forall t{\in}S.\ t \leq s$
proof (*rule classical*)
assume $\neg\ (\exists s{\in}S.\ \forall t{\in}S.\ t \leq s)$
then obtain *t* **where** *t*: $\forall s{\in}S.\ t\ s \in S$ **and** $\forall s{\in}S.\ s < t\ s$
by (*metis not-le*)
then have $\bigwedge s.\ s{\in}S \Longrightarrow open\ \{..<\ t\ s\}\ S \subseteq (\bigcup s{\in}S.\ \{..<\ t\ s\})$
by *auto*
with ‹*compact S*› **obtain** *C* **where** $C \subseteq S$ *finite C* **and** *C*: $S \subseteq (\bigcup s{\in}C.\ \{..<\ t\ s\})$
by (*metis compactE-image*)

with ‹$S \neq \{\}$› **have** *Max*: *Max* (t‘C) $\in$ t‘C **and** $\forall\, s{\in}t$‘$C.\ s \leq$ *Max* (t‘C)
by (*auto intro*!: *Max-in*)
with C **have** $S \subseteq \{..<$ *Max* (t‘C)$\}$
by (*auto intro*: *less-le-trans simp*: *subset-eq*)
with t *Max* ‹$C \subseteq S$› **show** *?thesis*
by *fastforce*
qed

lemma (**in** *linorder-topology*) *compact-attains-inf*:
assumes *compact* S $S \neq \{\}$
shows $\exists\, s{\in}S.\ \forall\, t{\in}S.\ s \leq t$
proof (*rule classical*)
assume $\neg\ (\exists\, s{\in}S.\ \forall\, t{\in}S.\ s \leq t)$
then obtain t **where** t: $\forall\, s{\in}S.\ t\ s \in S$ **and** $\forall\, s{\in}S.\ t\ s < s$
by (*metis not-le*)
then have $\bigwedge s.\ s{\in}S \Longrightarrow$ *open* $\{t\ s <..\}$ $S \subseteq (\bigcup s{\in}S.\ \{t\ s <..\})$
by *auto*
with ‹*compact* S› **obtain** C **where** $C \subseteq S$ *finite* C **and** C: $S \subseteq (\bigcup s{\in}C.\ \{t\ s <..\})$
by (*metis compactE-image*)
with ‹$S \neq \{\}$› **have** *Min*: *Min* (t‘C) $\in$ t‘C **and** $\forall\, s{\in}t$‘$C.$ *Min* (t‘C) $\leq s$
by (*auto intro*!: *Min-in*)
with C **have** $S \subseteq \{$*Min* (t‘C) $<..\}$
by (*auto intro*: *le-less-trans simp*: *subset-eq*)
with t *Min* ‹$C \subseteq S$› **show** *?thesis*
by *fastforce*
qed

lemma *continuous-attains-sup*:
fixes f :: *'a*::*topological-space* $\Rightarrow$ *'b*::*linorder-topology*
shows *compact* $s \Longrightarrow s \neq \{\} \Longrightarrow$ *continuous-on* s $f \Longrightarrow (\exists\, x{\in}s.\ \forall\, y{\in}s.\ \ f\ y \leq f\ x)$
using *compact-attains-sup*[*of* f ‘ s] *compact-continuous-image*[*of* s f] **by** *auto*

lemma *continuous-attains-inf*:
fixes f :: *'a*::*topological-space* $\Rightarrow$ *'b*::*linorder-topology*
shows *compact* $s \Longrightarrow s \neq \{\} \Longrightarrow$ *continuous-on* s $f \Longrightarrow (\exists\, x{\in}s.\ \forall\, y{\in}s.\ f\ x \leq f\ y)$
using *compact-attains-inf*[*of* f ‘ s] *compact-continuous-image*[*of* s f] **by** *auto*

98.12 Connectedness

context *topological-space*
begin

definition *connected* $S \longleftrightarrow$
$\neg\ (\exists\, A\ B.$ *open* $A \wedge$ *open* $B \wedge S \subseteq A \cup B \wedge A \cap B \cap S = \{\} \wedge A \cap S \neq \{\} \wedge B \cap S \neq \{\})$

lemma *connectedI*:

($\bigwedge A\ B$. *open* $A \Longrightarrow$ *open* $B \Longrightarrow A \cap U \neq \{\} \Longrightarrow B \cap U \neq \{\} \Longrightarrow A \cap B \cap U = \{\} \Longrightarrow U \subseteq A \cup B \Longrightarrow$ *False*)
$\Longrightarrow$ *connected* U
by (*auto simp*: *connected-def*)

lemma *connected-empty* [*simp*]: *connected* $\{\}$
by (*auto intro*!: *connectedI*)

lemma *connected-sing* [*simp*]: *connected* $\{x\}$
by (*auto intro*!: *connectedI*)

lemma *connectedD*:
connected $A \Longrightarrow$ *open* $U \Longrightarrow$ *open* $V \Longrightarrow U \cap V \cap A = \{\} \Longrightarrow A \subseteq U \cup V \Longrightarrow U \cap A = \{\} \vee V \cap A = \{\}$
by (*auto simp*: *connected-def*)

end

lemma *connected-closed*:
connected $s \longleftrightarrow$
$\neg$ ($\exists A\ B$. *closed* $A \wedge$ *closed* $B \wedge s \subseteq A \cup B \wedge A \cap B \cap s = \{\} \wedge A \cap s \neq \{\} \wedge B \cap s \neq \{\}$)
apply (*simp add*: *connected-def del*: *ex-simps*, *safe*)
apply (*drule-tac* $x{=}-A$ **in** *spec*)
apply (*drule-tac* $x{=}-B$ **in** *spec*)
apply (*fastforce simp add*: *closed-def* [*symmetric*])
apply (*drule-tac* $x{=}-A$ **in** *spec*)
apply (*drule-tac* $x{=}-B$ **in** *spec*)
apply (*fastforce simp add*: *open-closed* [*symmetric*])
done

lemma *connected-closedD*:
$[\![$*connected* s; $A \cap B \cap s = \{\}$; $s \subseteq A \cup B$; *closed* A; *closed* $B]\!] \Longrightarrow A \cap s = \{\} \vee B \cap s = \{\}$
by (*simp add*: *connected-closed*)

lemma *connected-Union*:
assumes *cs*: $\bigwedge s.\ s \in S \Longrightarrow$ *connected* s
and *ne*: $\bigcap S \neq \{\}$
shows *connected*($\bigcup S$)
proof (*rule connectedI*)
fix $A\ B$
assume A: *open* A **and** B: *open* B **and** *Alap*: $A \cap \bigcup S \neq \{\}$ **and** *Blap*: $B \cap \bigcup S \neq \{\}$
and *disj*: $A \cap B \cap \bigcup S = \{\}$ **and** *cover*: $\bigcup S \subseteq A \cup B$
have *disjs*:$\bigwedge s.\ s \in S \Longrightarrow A \cap B \cap s = \{\}$
using *disj* **by** *auto*
obtain *sa* **where** *sa*: $sa \in S$ $A \cap sa \neq \{\}$
using *Alap* **by** *auto*

obtain *sb* **where** *sb*: $sb \in S\ B \cap sb \neq \{\}$
using *Blap* **by** *auto*
obtain *x* **where** *x*: $\bigwedge s.\ s \in S \Longrightarrow x \in s$
using *ne* **by** *auto*
then have $x \in \bigcup S$
using ‹*sa* ∈ *S*› **by** *blast*
then have $x \in A \vee x \in B$
using *cover* **by** *auto*
then show *False*
using *cs* [*unfolded connected-def*]
by (*metis A B IntI Sup-upper sa sb disjs x cover empty-iff subset-trans*)
qed

lemma *connected-Un*: *connected s* $\Longrightarrow$ *connected t* $\Longrightarrow$ $s \cap t \neq \{\}$ $\Longrightarrow$ *connected* $(s \cup t)$
using *connected-Union* [*of* {*s*,*t*}] **by** *auto*

lemma *connected-diff-open-from-closed*:
assumes *st*: $s \subseteq t$
and *tu*: $t \subseteq u$
and *s*: *open s*
and *t*: *closed t*
and *u*: *connected u*
and *ts*: *connected* $(t - s)$
shows *connected*$(u - s)$
proof (*rule connectedI*)
fix *A B*
assume *AB*: *open A open B* $A \cap (u - s) \neq \{\}$ $B \cap (u - s) \neq \{\}$
and *disj*: $A \cap B \cap (u - s) = \{\}$
and *cover*: $u - s \subseteq A \cup B$
then consider $A \cap (t - s) = \{\} \mid B \cap (t - s) = \{\}$
using *st ts tu connectedD* [*of t−s A B*] **by** *auto*
then show *False*
proof *cases*
case *1*
then have $(A - t) \cap (B \cup s) \cap u = \{\}$
using *disj st* **by** *auto*
moreover have $u \subseteq (A - t) \cup (B \cup s)$
using *1 cover* **by** *auto*
ultimately show *False*
using *connectedD* [*of u A − t B ∪ s*] *AB s t 1 u* **by** *auto*
next
case *2*
then have $(A \cup s) \cap (B - t) \cap u = \{\}$
using *disj st* **by** *auto*
moreover have $u \subseteq (A \cup s) \cup (B - t)$
using *2 cover* **by** *auto*
ultimately show *False*
using *connectedD* [*of u A ∪ s B − t*] *AB s t 2 u* **by** *auto*

```
  qed
qed

lemma connected-iff-const:
  fixes S :: 'a::topological-space set
  shows connected S ⟷ (∀ P::'a ⇒ bool. continuous-on S P ⟶ (∃ c. ∀ s∈S. P s
= c))
proof safe
  fix P :: 'a ⇒ bool
  assume connected S continuous-on S P
  then have ⋀b. ∃ A. open A ∧ A ∩ S = P -` {b} ∩ S
    unfolding continuous-on-open-invariant by (simp add: open-discrete)
  from this[of True] this[of False]
  obtain t f where open t open f and ∗: f ∩ S = P -` {False} ∩ S t ∩ S = P
-` {True} ∩ S
    by meson
  then have t ∩ S = {} ∨ f ∩ S = {}
    by (intro connectedD[OF ‹connected S›]) auto
  then show ∃ c. ∀ s∈S. P s = c
  proof (rule disjE)
    assume t ∩ S = {}
    then show ?thesis
      unfolding ∗ by (intro exI[of - False]) auto
  next
    assume f ∩ S = {}
    then show ?thesis
      unfolding ∗ by (intro exI[of - True]) auto
  qed
next
  assume P: ∀ P::'a ⇒ bool. continuous-on S P ⟶ (∃ c. ∀ s∈S. P s = c)
  show connected S
  proof (rule connectedI)
    fix A B
    assume ∗: open A open B A ∩ S ≠ {} B ∩ S ≠ {} A ∩ B ∩ S = {} S ⊆ A ∪
B
    have continuous-on S (λx. x ∈ A)
      unfolding continuous-on-open-invariant
    proof safe
      fix C :: bool set
      have C = UNIV ∨ C = {True} ∨ C = {False} ∨ C = {}
        using subset-UNIV[of C] unfolding UNIV-bool by auto
      with ∗ show ∃ T. open T ∧ T ∩ S = (λx. x ∈ A) -` C ∩ S
        by (intro exI[of - (if True ∈ C then A else {}) ∪ (if False ∈ C then B else
{})]) auto
    qed
    from P[rule-format, OF this] obtain c where ⋀s. s ∈ S ⟹ (s ∈ A) = c
      by blast
    with ∗ show False
      by (cases c) auto
```

qed
qed

lemma *connectedD-const*: *connected S* $\Longrightarrow$ *continuous-on S P* $\Longrightarrow$ $\exists\, c.\ \forall\, s{\in}S.\ P\ s = c$
for *P* :: *'a::topological-space* $\Rightarrow$ *bool*
by (*auto simp*: *connected-iff-const*)

lemma *connectedI-const*:
$(\bigwedge P{::}'a{::}$*topological-space* $\Rightarrow$ *bool. continuous-on S P* $\Longrightarrow$ $\exists\, c.\ \forall\, s{\in}S.\ P\ s = c)$
$\Longrightarrow$ *connected S*
by (*auto simp*: *connected-iff-const*)

lemma *connected-local-const*:
assumes *connected A a* $\in$ *A b* $\in$ *A*
and $*$: $\forall\, a{\in}A.$ *eventually* $(\lambda b.\ f\ a = f\ b)$ (*at a within A*)
shows *f a = f b*
proof −
obtain *S* **where** *S*: $\bigwedge a.\ a \in A \Longrightarrow a \in S\ a$ $\bigwedge a.\ a \in A \Longrightarrow$ *open* (*S a*)
$\bigwedge a\ x.\ a \in A \Longrightarrow x \in S\ a \Longrightarrow x \in A \Longrightarrow f\ a = f\ x$
using $*$ **unfolding** *eventually-at-topological* **by** *metis*
let *?P* = $\bigcup b{\in}\{b{\in}A.\ f\ a = f\ b\}.\ S\ b$ **and** *?N* = $\bigcup b{\in}\{b{\in}A.\ f\ a \neq f\ b\}.\ S\ b$
have *?P* $\cap$ *A* = {} $\vee$ *?N* $\cap$ *A* = {}
using ‹*connected A*› *S* ‹*a*∈*A*›
by (*intro connectedD*) (*auto*, *metis*)
then show *f a = f b*
proof
assume *?N* $\cap$ *A* = {}
then have $\forall\, x{\in}A.\ f\ a = f\ x$
using *S*(*1*) **by** *auto*
with ‹*b*∈*A*› **show** *?thesis* **by** *auto*
next
assume *?P* $\cap$ *A* = {} **then show** *?thesis*
using ‹*a* ∈ *A*› *S*(*1*)[*of a*] **by** *auto*
qed
qed

lemma (**in** *linorder-topology*) *connectedD-interval*:
assumes *connected U*
and *xy*: *x* ∈ *U y* ∈ *U*
and *x* ≤ *z z* ≤ *y*
shows *z* ∈ *U*
proof −
have *eq*: {..<*z*} ∪ {*z*<..} = − {*z*}
by *auto*
have ¬ *connected U* **if** *z* ∉ *U x* < *z z* < *y*
using *xy that*
apply (*simp only*: *connected-def simp-thms*)
apply (*rule-tac exI*[*of* - {..< *z*}])

```
    apply (rule-tac exI[of - {z <..}])
    apply (auto simp add: eq)
    done
  with assms show z ∈ U
    by (metis less-le)
qed

lemma (in linorder-topology) not-in-connected-cases:
  assumes conn: connected S
  assumes nbdd: x ∉ S
  assumes ne: S ≠ {}
  obtains bdd-above S ⋀y. y ∈ S ⟹ x ≥ y | bdd-below S ⋀y. y ∈ S ⟹ x ≤ y
proof −
  obtain s where s ∈ S using ne by blast
  {
    assume s ≤ x
    have False if x ≤ y y ∈ S for y
      using connectedD-interval[OF conn ‹s ∈ S› ‹y ∈ S› ‹s ≤ x› ‹x ≤ y›] ‹x ∉ S›
      by simp
    then have wit: y ∈ S ⟹ x ≥ y for y
      using le-cases by blast
    then have bdd-above S
      by (rule local.bdd-aboveI)
    note this wit
  } moreover {
    assume x ≤ s
    have False if x ≥ y y ∈ S for y
      using connectedD-interval[OF conn ‹y ∈ S› ‹s ∈ S› ‹x ≥ y› ‹s ≥ x› ] ‹x ∉
S›
      by simp
    then have wit: y ∈ S ⟹ x ≤ y for y
      using le-cases by blast
    then have bdd-below S
      by (rule bdd-belowI)
    note this wit
  } ultimately show ?thesis
    by (meson le-cases that)
qed

lemma connected-continuous-image:
  assumes ∗: continuous-on s f
    and connected s
  shows connected (f ‘ s)
proof (rule connectedI-const)
  fix P :: 'b ⇒ bool
  assume continuous-on (f ‘ s) P
  then have continuous-on s (P ∘ f)
    by (rule continuous-on-compose[OF ∗])
  from connectedD-const[OF ‹connected s› this] show ∃ c. ∀ s∈f ‘ s. P s = c
```

by *auto*
qed

lemma *connected-Un-UN*:
assumes *connected A* $\bigwedge X.\ X \in B \Longrightarrow$ *connected X* $\bigwedge X.\ X \in B \Longrightarrow A \cap X \neq \{\}$
shows *connected* $(A \cup \bigcup B)$
proof (*rule connectedI-const*)
fix $f :: 'a \Rightarrow bool$
assume f: *continuous-on* $(A \cup \bigcup B)$ f
have *connected A continuous-on A f*
by (*auto intro*: *assms continuous-on-subset*[*OF f*(*1*)])
from *connectedD-const*[*OF this*] **obtain** c **where** c: $\bigwedge x.\ x \in A \Longrightarrow f\ x = c$
by *metis*
have $f\ x = c$ **if** $x \in X\ X \in B$ **for** $x\ X$
proof −
have *connected X continuous-on X f*
using *that* **by** (*auto intro*: *assms continuous-on-subset*[*OF f*])
from *connectedD-const*[*OF this*] **obtain** c' **where** c': $\bigwedge x.\ x \in X \Longrightarrow f\ x = c'$
by *metis*
from *assms*(*3*) **and** *that* **obtain** y **where** $y \in A \cap X$
by *auto*
with *c*[*of y*] *c'*[*of y*] *c'*[*of x*] *that* **show** *?thesis*
by *auto*
qed
with c **show** $\exists c.\ \forall x{\in}A \cup \bigcup\ B.\ f\ x = c$
by (*intro exI*[*of - c*]) *auto*
qed

99 Linear Continuum Topologies

class *linear-continuum-topology* = *linorder-topology* + *linear-continuum*
begin

lemma *Inf-notin-open*:
assumes *A*: *open A*
and *bnd*: $\forall a{\in}A.\ x < a$
shows *Inf A* $\notin A$
proof
assume *Inf A* $\in A$
then obtain b **where** $b <$ *Inf A* $\{b <..$ *Inf A*$\} \subseteq A$
using *open-left*[*of A Inf A x*] *assms* **by** *auto*
with *dense*[*of b Inf A*] **obtain** c **where** $c <$ *Inf A* $c \in A$
by (*auto simp*: *subset-eq*)
then show *False*
using *cInf-lower*[*OF* ‹$c \in A$›] *bnd*
by (*metis not-le less-imp-le bdd-belowI*)
qed

lemma *Sup-notin-open*:
assumes *A*: *open A*
and *bnd*: $\forall\, a{\in}A.\ a < x$
shows *Sup A* $\notin$ *A*
proof
assume *Sup A* $\in$ *A*
with *assms* **obtain** *b* **where** *Sup A* $< b$ $\{$*Sup A* ..$<$ *b*$\}$ $\subseteq$ *A*
using *open-right*[*of A Sup A x*] **by** *auto*
with *dense*[*of Sup A b*] **obtain** *c* **where** *Sup A* $< c$ *c* $\in$ *A*
by (*auto simp*: *subset-eq*)
then show *False*
using *cSup-upper*[*OF* ‹*c* $\in$ *A*›] *bnd*
by (*metis less-imp-le not-le bdd-aboveI*)
qed

end

instance *linear-continuum-topology* $\subseteq$ *perfect-space*
proof
fix *x* :: *'a*
obtain *y* **where** $x < y \vee y < x$
using *ex-gt-or-lt* [*of x*] **..**
with *Inf-notin-open*[*of* $\{x\}$ *y*] *Sup-notin-open*[*of* $\{x\}$ *y*] **show** $\neg$ *open* $\{x\}$
by *auto*
qed

lemma *connectedI-interval*:
fixes *U* :: *'a* :: *linear-continuum-topology set*
assumes $*$: $\bigwedge x\ y\ z.\ x \in U \Longrightarrow y \in U \Longrightarrow x \leq z \Longrightarrow z \leq y \Longrightarrow z \in U$
shows *connected U*
proof (*rule connectedI*)
{
fix *A B*
assume *open A open B A* $\cap$ *B* $\cap$ *U* $= \{\}$ *U* $\subseteq$ *A* $\cup$ *B*
fix *x y*
assume $x < y$ $x \in A$ $y \in B$ $x \in U$ $y \in U$

let *?z* = *Inf* (*B* $\cap$ $\{x$ <..$\}$)

have $x \leq$ *?z* *?z* $\leq y$
using ‹$y \in B$› ‹$x < y$› **by** (*auto intro*: *cInf-lower cInf-greatest*)
with ‹$x \in U$› ‹$y \in U$› **have** *?z* $\in$ *U*
by (*rule* $*$)
moreover have *?z* $\notin$ *B* $\cap$ $\{x$ <..$\}$
using ‹*open B*› **by** (*intro Inf-notin-open*) *auto*
ultimately have *?z* $\in$ *A*
using ‹$x \leq$ *?z*› ‹*A* $\cap$ *B* $\cap$ *U* $= \{\}$› ‹$x \in A$› ‹*U* $\subseteq$ *A* $\cup$ *B*› **by** *auto*
have $\exists\, b{\in}B.\ b \in A \wedge b \in U$ **if** *?z* $< y$
proof $-$

obtain *a* **where** *?z* < *a* {*?z* ..< *a*} ⊆ *A*
using *open-right*[*OF* ‹*open A*› ‹*?z* ∈ *A*› ‹*?z* < *y*›] **by** *auto*
moreover obtain *b* **where** *b* ∈ *B x* < *b b* < *min a y*
using *cInf-less-iff*[*of B* ∩ {*x* <..} *min a y*] ‹*?z* < *a*› ‹*?z* < *y*› ‹*x* < *y*› ‹*y* ∈ *B*›
by *auto*
moreover have *?z* ≤ *b*
using ‹*b* ∈ *B*› ‹*x* < *b*›
by (*intro cInf-lower*) *auto*
moreover have *b* ∈ *U*
using ‹*x* ≤ *?z*› ‹*?z* ≤ *b*› ‹*b* < *min a y*›
by (*intro* ∗[*OF* ‹*x* ∈ *U*› ‹*y* ∈ *U*›]) (*auto simp*: *less-imp-le*)
ultimately show *?thesis*
by (*intro bexI*[*of* - *b*]) *auto*
qed
then have *False*
using ‹*?z* ≤ *y*› ‹*?z* ∈ *A*› ‹*y* ∈ *B*› ‹*y* ∈ *U*› ‹*A* ∩ *B* ∩ *U* = {}›
unfolding *le-less* **by** *blast*
}
note *not-disjoint* = *this*

fix *A B* **assume** *AB*: *open A open B U* ⊆ *A* ∪ *B A* ∩ *B* ∩ *U* = {}
moreover assume *A* ∩ *U* ≠ {} **then obtain** *x* **where** *x*: *x* ∈ *U x* ∈ *A* **by** *auto*
moreover assume *B* ∩ *U* ≠ {} **then obtain** *y* **where** *y*: *y* ∈ *U y* ∈ *B* **by** *auto*
moreover note *not-disjoint*[*of B A y x*] *not-disjoint*[*of A B x y*]
ultimately show *False*
by (*cases x y rule*: *linorder-cases*) *auto*
qed

lemma *connected-iff-interval*: *connected U* ⟷ (∀ *x*∈*U*. ∀ *y*∈*U*. ∀ *z*. *x* ≤ *z* ⟶ *z* ≤ *y* ⟶ *z* ∈ *U*)
for *U* :: *'a*::*linear-continuum-topology set*
by (*auto intro*: *connectedI-interval dest*: *connectedD-interval*)

lemma *connected-UNIV*[*simp*]: *connected* (*UNIV*::*'a*::*linear-continuum-topology set*)
by (*simp add*: *connected-iff-interval*)

lemma *connected-Ioi*[*simp*]: *connected* {*a*<..}
for *a* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Ici*[*simp*]: *connected* {*a*..}
for *a* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Iio*[*simp*]: *connected* {..<*a*}
for *a* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Iic*[*simp*]: *connected* {..*a*}
for *a* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Ioo*[*simp*]: *connected* {*a*<..<*b*}
for *a b* :: *'a*::*linear-continuum-topology*
unfolding *connected-iff-interval* **by** *auto*

lemma *connected-Ioc*[*simp*]: *connected* {*a*<..*b*}
for *a b* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Ico*[*simp*]: *connected* {*a*..<*b*}
for *a b* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-Icc*[*simp*]: *connected* {*a*..*b*}
for *a b* :: *'a*::*linear-continuum-topology*
by (*auto simp*: *connected-iff-interval*)

lemma *connected-contains-Ioo*:
fixes *A* :: *'a* :: *linorder-topology set*
assumes *connected A a* ∈ *A b* ∈ *A* **shows** {*a* <..< *b*} ⊆ *A*
using *connectedD-interval*[*OF assms*] **by** (*simp add*: *subset-eq Ball-def less-imp-le*)

lemma *connected-contains-Icc*:
fixes *A* :: *'a*::*linorder-topology set*
assumes *connected A a* ∈ *A b* ∈ *A*
shows {*a*..*b*} ⊆ *A*
proof
fix *x* **assume** *x* ∈ {*a*..*b*}
then have *x* = *a* ∨ *x* = *b* ∨ *x* ∈ {*a*<..<*b*}
by *auto*
then show *x* ∈ *A*
using *assms connected-contains-Ioo*[*of A a b*] **by** *auto*
qed

lemma *continuous-on-neq-split*:
fixes *f* :: *'a*::*linear-continuum-topology* ⇒ *'b*::*linorder-topology*
assumes ∀ *x*∈*S*. *f x* ≠ *y* **and** *f*: *continuous-on S f* **and** *S*: *connected S*
shows (∀ *x*∈*S*. *f x* > *y*) ∨ (∀ *x*∈*S*. *f x* < *y*)
using *connectedD-interval assms connected-continuous-image* [*OF f S*]
using *linorder-not-less* **by** *blast*

99.1 Intermediate Value Theorem

lemma *IVT'*:
fixes *f* :: *'a*::*linear-continuum-topology* ⇒ *'b*::*linorder-topology*
assumes *y*: *f a* ≤ *y y* ≤ *f b a* ≤ *b*

```
    and ∗: continuous-on {a .. b} f
  shows ∃x. a ≤ x ∧ x ≤ b ∧ f x = y
proof −
  have connected {a..b}
    unfolding connected-iff-interval by auto
  from connected-continuous-image[OF ∗ this, THEN connectedD-interval, of f a
f b y] y
  show ?thesis
    by (auto simp add: atLeastAtMost-def atLeast-def atMost-def)
qed

lemma IVT2':
  fixes f :: 'a :: linear-continuum-topology ⇒ 'b :: linorder-topology
  assumes y: f b ≤ y y ≤ f a a ≤ b
    and ∗: continuous-on {a .. b} f
  shows ∃x. a ≤ x ∧ x ≤ b ∧ f x = y
proof −
  have connected {a..b}
    unfolding connected-iff-interval by auto
  from connected-continuous-image[OF ∗ this, THEN connectedD-interval, of f b
f a y] y
  show ?thesis
    by (auto simp add: atLeastAtMost-def atLeast-def atMost-def)
qed

lemma IVT:
  fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
  shows f a ≤ y ⟹ y ≤ f b ⟹ a ≤ b ⟹ (∀x. a ≤ x ∧ x ≤ b ⟶ isCont f x)
⟹
    ∃x. a ≤ x ∧ x ≤ b ∧ f x = y
  by (rule IVT') (auto intro: continuous-at-imp-continuous-on)

lemma IVT2:
  fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
  shows f b ≤ y ⟹ y ≤ f a ⟹ a ≤ b ⟹ (∀x. a ≤ x ∧ x ≤ b ⟶ isCont f x)
⟹
    ∃x. a ≤ x ∧ x ≤ b ∧ f x = y
  by (rule IVT2') (auto intro: continuous-at-imp-continuous-on)

lemma continuous-inj-imp-mono:
  fixes f :: 'a::linear-continuum-topology ⇒ 'b::linorder-topology
  assumes x: a < x x < b
    and cont: continuous-on {a..b} f
    and inj: inj-on f {a..b}
  shows (f a < f x ∧ f x < f b) ∨ (f b < f x ∧ f x < f a)
proof −
  note I = inj-on-eq-iff[OF inj]
  {
    assume f x < f a f x < f b
```

```
    then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f x < f s
      using IVT'[of f x min (f a) (f b) b] IVT2'[of f x min (f a) (f b) a] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
    with x I have False by auto
  }
  moreover
  {
    assume f a < f x f b < f x
    then obtain s t where x ≤ s s ≤ b a ≤ t t ≤ x f s = f t f s < f x
      using IVT'[of f a max (f a) (f b) x] IVT2'[of f b max (f a) (f b) x] x
      by (auto simp: continuous-on-subset[OF cont] less-imp-le)
    with x I have False by auto
  }
  ultimately show ?thesis
    using I[of a x] I[of x b] x less-trans[OF x]
    by (auto simp add: le-less less-imp-neq neq-iff)
qed

lemma continuous-at-Sup-mono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes mono f
    and cont: continuous (at-left (Sup S)) f
    and S: S ≠ {} bdd-above S
  shows f (Sup S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Sup S)) (at-left (Sup S))
    using cont unfolding continuous-within .
  show f (Sup S) ≤ (SUP s∈S. f s)
  proof cases
    assume Sup S ∈ S
    then show ?thesis
      by (rule cSUP-upper) (auto intro: bdd-above-image-mono S ‹mono f›)
  next
    assume Sup S ∉ S
    from ‹S ≠ {}› obtain s where s ∈ S
      by auto
    with ‹Sup S ∉ S› S have s < Sup S
      unfolding less-le by (blast intro: cSup-upper)
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with order-tendstoD(1)[OF f, of SUP s∈S. f s] obtain b where b < Sup S
        and ∗: ⋀y. b < y ⟹ y < Sup S ⟹ (SUP s∈S. f s) < f y
        by (auto simp: not-le eventually-at-left[OF ‹s < Sup S›])
      with ‹S ≠ {}› obtain c where c ∈ S b < c
        using less-cSupD[of S b] by auto
      with ‹Sup S ∉ S› S have c < Sup S
        unfolding less-le by (blast intro: cSup-upper)
```

```
    from *[OF ‹b < c› ‹c < Sup S›] cSUP-upper[OF ‹c ∈ S› bdd-above-image-mono[of
f]]
     show False
       by (auto simp: assms)
    qed
  qed
qed (intro cSUP-least ‹mono f›[THEN monoD] cSup-upper S)

lemma continuous-at-Sup-antimono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes antimono f
    and cont: continuous (at-left (Sup S)) f
    and S: S ≠ {} bdd-above S
  shows f (Sup S) = (INF s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Sup S)) (at-left (Sup S))
    using cont unfolding continuous-within .
  show (INF s∈S. f s) ≤ f (Sup S)
  proof cases
    assume Sup S ∈ S
    then show ?thesis
     by (intro cINF-lower) (auto intro: bdd-below-image-antimono S ‹antimono f›)
  next
    assume Sup S ∉ S
    from ‹S ≠ {}› obtain s where s ∈ S
      by auto
    with ‹Sup S ∉ S› S have s < Sup S
      unfolding less-le by (blast intro: cSup-upper)
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with order-tendstoD(2)[OF f, of INF s∈S. f s] obtain b where b < Sup S
        and *: ⋀y. b < y ⟹ y < Sup S ⟹ f y < (INF s∈S. f s)
        by (auto simp: not-le eventually-at-left[OF ‹s < Sup S›])
      with ‹S ≠ {}› obtain c where c ∈ S b < c
        using less-cSupD[of S b] by auto
      with ‹Sup S ∉ S› S have c < Sup S
        unfolding less-le by (blast intro: cSup-upper)
      from *[OF ‹b < c› ‹c < Sup S›] cINF-lower[OF bdd-below-image-antimono,
of f S c] ‹c ∈ S›
      show False
        by (auto simp: assms)
    qed
  qed
qed (intro cINF-greatest ‹antimono f›[THEN antimonoD] cSup-upper S)

lemma continuous-at-Inf-mono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
```

```
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes mono f
    and cont: continuous (at-right (Inf S)) f
    and S: S ≠ {} bdd-below S
  shows f (Inf S) = (INF s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Inf S)) (at-right (Inf S))
    using cont unfolding continuous-within .
  show (INF s∈S. f s) ≤ f (Inf S)
  proof cases
    assume Inf S ∈ S
    then show ?thesis
      by (rule cINF-lower[rotated]) (auto intro: bdd-below-image-mono S ‹mono f›)
  next
    assume Inf S ∉ S
    from ‹S ≠ {}› obtain s where s ∈ S
      by auto
    with ‹Inf S ∉ S› S have Inf S < s
      unfolding less-le by (blast intro: cInf-lower)
    show ?thesis
    proof (rule ccontr)
      assume ¬ ?thesis
      with order-tendstoD(2)[OF f, of INF s∈S. f s] obtain b where Inf S < b
        and *: ⋀y. Inf S < y ⟹ y < b ⟹ f y < (INF s∈S. f s)
        by (auto simp: not-le eventually-at-right[OF ‹Inf S < s›])
      with ‹S ≠ {}› obtain c where c ∈ S c < b
        using cInf-lessD[of S b] by auto
      with ‹Inf S ∉ S› S have Inf S < c
        unfolding less-le by (blast intro: cInf-lower)
      from *[OF ‹Inf S < c› ‹c < b›] cINF-lower[OF bdd-below-image-mono[of f]
‹c ∈ S›]
      show False
        by (auto simp: assms)
    qed
  qed
qed (intro cINF-greatest ‹mono f›[THEN monoD] cInf-lower ‹bdd-below S› ‹S ≠
{}›)

lemma continuous-at-Inf-antimono:
  fixes f :: 'a::{linorder-topology,conditionally-complete-linorder} ⇒
    'b::{linorder-topology,conditionally-complete-linorder}
  assumes antimono f
    and cont: continuous (at-right (Inf S)) f
    and S: S ≠ {} bdd-below S
  shows f (Inf S) = (SUP s∈S. f s)
proof (rule antisym)
  have f: (f ⟶ f (Inf S)) (at-right (Inf S))
    using cont unfolding continuous-within .
  show f (Inf S) ≤ (SUP s∈S. f s)
```

proof *cases*
assume *Inf S ∈ S*
then show *?thesis*
by (*rule cSUP-upper*) (*auto intro*: *bdd-above-image-antimono S ‹antimono f›*)
next
assume *Inf S ∉ S*
from ‹*S ≠ {}*› **obtain** *s* **where** *s ∈ S*
by *auto*
with ‹*Inf S ∉ S*› *S* **have** *Inf S < s*
unfolding *less-le* **by** (*blast intro*: *cInf-lower*)
show *?thesis*
proof (*rule ccontr*)
assume ¬ *?thesis*
with *order-tendstoD*(*1*)[*OF f*, *of SUP s∈S*. *f s*] **obtain** *b* **where** *Inf S < b*
and ∗: ⋀*y*. *Inf S < y ⟹ y < b ⟹* (*SUP s∈S*. *f s*) *< f y*
by (*auto simp*: *not-le eventually-at-right*[*OF* ‹*Inf S < s*›])
with ‹*S ≠ {}*› **obtain** *c* **where** *c ∈ S c < b*
using *cInf-lessD*[*of S b*] **by** *auto*
with ‹*Inf S ∉ S*› *S* **have** *Inf S < c*
unfolding *less-le* **by** (*blast intro*: *cInf-lower*)
from ∗[*OF* ‹*Inf S < c*› ‹*c < b*›] *cSUP-upper*[*OF* ‹*c ∈ S*› *bdd-above-image-antimono*[*of f*]]
show *False*
by (*auto simp*: *assms*)
qed
qed
qed (*intro cSUP-least* ‹*antimono f*›[*THEN antimonoD*] *cInf-lower S*)

99.2 Uniform spaces

class *uniformity* =
fixes *uniformity* :: (*'a* × *'a*) *filter*
begin

abbreviation *uniformity-on* :: *'a set* ⇒ (*'a* × *'a*) *filter*
where *uniformity-on s* ≡ *inf uniformity* (*principal* (*s*×*s*))

end

context — Suppress difficult-to-implement class parameter
begin

private definition *unimplemented-uniformity* :: ‹(*'a*::*uniformity* × *'a*) *filter*›
where [*code drop*]: ‹*unimplemented-uniformity* = *uniformity*›

lemma [*code*]:
‹*uniformity* = *unimplemented-uniformity*›
by (*simp add*: *unimplemented-uniformity-def*)

end

class *open-uniformity* = *open* + *uniformity* +
  assumes *open-uniformity*:
    $\bigwedge U$. *open* $U \longleftrightarrow (\forall\, x{\in}U$. *eventually* $(\lambda(x', y).\ x' = x \longrightarrow y \in U)$ *uniformity*)
begin

subclass *topological-space*
 by *standard* (*force elim*: *eventually-mono eventually-elim2 simp*: *split-beta'* *open-uniformity*)+

end

class *uniform-space* = *open-uniformity* +
  assumes *uniformity-refl*: *eventually E uniformity* $\Longrightarrow$ *E* (x, x)
   and *uniformity-sym*: *eventually E uniformity* $\Longrightarrow$ *eventually* $(\lambda(x, y).\ E\ (y, x))$ *uniformity*
   and *uniformity-trans*:
     eventually E uniformity $\Longrightarrow$
      $\exists\, D$. *eventually D uniformity* $\wedge\ (\forall\, x\ y\ z.\ D\ (x, y) \longrightarrow D\ (y, z) \longrightarrow E\ (x, z))$
begin

lemma *uniformity-bot*: *uniformity* $\neq$ *bot*
  using *uniformity-refl* **by** *auto*

lemma *uniformity-trans'*:
  eventually E uniformity $\Longrightarrow$
   eventually $(\lambda((x, y), (y', z)).\ y = y' \longrightarrow E\ (x, z))$ $(\mathit{uniformity} \times_F \mathit{uniformity})$
  by (*drule uniformity-trans*) (*auto simp add*: *eventually-prod-same*)

lemma *uniformity-transE*:
  assumes *eventually E uniformity*
  obtains *D* **where** *eventually D uniformity* $\bigwedge x\ y\ z.\ D\ (x, y) \Longrightarrow D\ (y, z) \Longrightarrow E\ (x, z)$
  using *uniformity-trans* [*OF assms*] **by** *auto*

lemma *eventually-nhds-uniformity*:
  eventually P (*nhds x*) $\longleftrightarrow$ *eventually* $(\lambda(x', y).\ x' = x \longrightarrow P\ y)$ *uniformity*
  (**is** - $\longleftrightarrow$ *?N P x*)
  unfolding *eventually-nhds*
proof *safe*
  assume $*$: *?N P x*
  have *?N* (*?N P*) *x* **if** *?N P x* **for** *x*
  proof −
    from *that* **obtain** *D* **where** *ev*: *eventually D uniformity*
      and *D*: *D* $(a, b) \Longrightarrow D\ (b, c) \Longrightarrow$ *case* (a, c) *of* $(x', y) \Rightarrow x' = x \longrightarrow P\ y$ **for** *a b c*
      by (*rule uniformity-transE*) *simp*
    from *ev* **show** *?thesis*
      by *eventually-elim* (*insert ev D*, *force elim*: *eventually-mono split*: *prod.split*)

qed
then have *open* $\{x.\ ?N\ P\ x\}$
by (*simp add*: *open-uniformity*)
then show $\exists S.\ open\ S \wedge x \in S \wedge (\forall x{\in}S.\ P\ x)$
by (*intro exI*[*of* - $\{x.\ ?N\ P\ x\}$]) (*auto dest*: *uniformity-refl simp*: $*$)
qed (*force simp add*: *open-uniformity elim*: *eventually-mono*)

99.2.1 Totally bounded sets

definition *totally-bounded* :: *'a set* ⇒ *bool*
where *totally-bounded S* ⟷
$(\forall E.\ eventually\ E\ uniformity \longrightarrow (\exists X.\ finite\ X \wedge (\forall s{\in}S.\ \exists x{\in}X.\ E\ (x,\ s))))$

lemma *totally-bounded-empty*[*iff*]: *totally-bounded* {}
by (*auto simp add*: *totally-bounded-def*)

lemma *totally-bounded-subset*: *totally-bounded S* ⟹ $T \subseteq S$ ⟹ *totally-bounded T*
by (*fastforce simp add*: *totally-bounded-def*)

lemma *totally-bounded-Union*[*intro*]:
assumes *M*: *finite M* $\bigwedge S.\ S \in M$ ⟹ *totally-bounded S*
shows *totally-bounded* $(\bigcup M)$
unfolding *totally-bounded-def*
proof *safe*
fix *E*
assume *eventually E uniformity*
with *M* **obtain** *X* **where** $\forall S{\in}M.\ finite\ (X\ S) \wedge (\forall s{\in}S.\ \exists x{\in}X\ S.\ E\ (x,\ s))$
by (*metis totally-bounded-def*)
with ‹*finite M*› **show** $\exists X.\ finite\ X \wedge (\forall s{\in}\bigcup M.\ \exists x{\in}X.\ E\ (x,\ s))$
by (*intro exI*[*of* - $\bigcup S{\in}M.\ X\ S$]) *force*
qed

99.2.2 Cauchy filter

definition *cauchy-filter* :: *'a filter* ⇒ *bool*
where *cauchy-filter F* ⟷ $F \times_F F \leq uniformity$

definition *Cauchy* :: (*nat* ⇒ *'a*) ⇒ *bool*
where *Cauchy-uniform*: *Cauchy X* = *cauchy-filter* (*filtermap X sequentially*)

lemma *Cauchy-uniform-iff*:
Cauchy X ⟷ $(\forall P.\ eventually\ P\ uniformity \longrightarrow (\exists N.\ \forall n{\geq}N.\ \forall m{\geq}N.\ P\ (X\ n,\ X\ m)))$
unfolding *Cauchy-uniform cauchy-filter-def le-filter-def eventually-prod-same eventually-filtermap eventually-sequentially*
proof *safe*
let *?U* = $\lambda P.$ *eventually P uniformity*
{
fix *P*

assume *?U P* $\forall P.$ *?U P* $\longrightarrow (\exists Q.\ (\exists N.\ \forall n{\geq}N.\ Q\ (X\ n)) \wedge (\forall x\ y.\ Q\ x \longrightarrow Q\ y \longrightarrow P\ (x,\ y)))$
then obtain *Q N* **where** $\bigwedge n.\ n \geq N \Longrightarrow Q\ (X\ n)\ \bigwedge x\ y.\ Q\ x \Longrightarrow Q\ y \Longrightarrow P\ (x,\ y)$
by *metis*
then show $\exists N.\ \forall n{\geq}N.\ \forall m{\geq}N.\ P\ (X\ n,\ X\ m)$
by *blast*
next
fix *P*
assume *?U P* **and** *P*: $\forall P.$ *?U P* $\longrightarrow (\exists N.\ \forall n{\geq}N.\ \forall m{\geq}N.\ P\ (X\ n,\ X\ m))$
then obtain *Q* **where** *?U Q* **and** *Q*: $\bigwedge x\ y\ z.\ Q\ (x,\ y) \Longrightarrow Q\ (y,\ z) \Longrightarrow P\ (x,\ z)$
by (*auto elim*: *uniformity-transE*)
then have *?U* $(\lambda x.\ Q\ x \wedge (\lambda(x,\ y).\ Q\ (y,\ x))\ x)$
unfolding *eventually-conj-iff* **by** (*simp add*: *uniformity-sym*)
from *P*[*rule-format*, *OF this*]
obtain *N* **where** *N*: $\bigwedge n\ m.\ n \geq N \Longrightarrow m \geq N \Longrightarrow Q\ (X\ n,\ X\ m) \wedge Q\ (X\ m,\ X\ n)$
by *auto*
show $\exists Q.\ (\exists N.\ \forall n{\geq}N.\ Q\ (X\ n)) \wedge (\forall x\ y.\ Q\ x \longrightarrow Q\ y \longrightarrow P\ (x,\ y))$
proof (*safe intro*!: *exI*[*of* - $\lambda x.\ \forall n{\geq}N.\ Q\ (x,\ X\ n) \wedge Q\ (X\ n,\ x)$] *exI*[*of* - *N*] *N*)
fix *x y*
assume $\forall n{\geq}N.\ Q\ (x,\ X\ n) \wedge Q\ (X\ n,\ x)\ \forall n{\geq}N.\ Q\ (y,\ X\ n) \wedge Q\ (X\ n,\ y)$
then have *Q* $(x,\ X\ N)$ *Q* $(X\ N,\ y)$ **by** *auto*
then show *P* $(x,\ y)$
by (*rule Q*)
qed
}
qed

lemma *nhds-imp-cauchy-filter*:
assumes $*$: $F \leq$ *nhds x*
shows *cauchy-filter F*
proof −
have $F \times_F F \leq$ *nhds x* $\times_F$ *nhds x*
by (*intro prod-filter-mono* $*$)
also have $\ldots \leq$ *uniformity*
unfolding *le-filter-def eventually-nhds-uniformity eventually-prod-same*
proof *safe*
fix *P*
assume *eventually P uniformity*
then obtain *Ql* **where** *ev*: *eventually Ql uniformity*
and *Ql* $(x,\ y) \Longrightarrow$ *Ql* $(y,\ z) \Longrightarrow P\ (x,\ z)$ **for** *x y z*
by (*rule uniformity-transE*) *simp*
with *ev*[*THEN uniformity-sym*]
show $\exists Q.$ *eventually* $(\lambda(x',\ y).\ x' = x \longrightarrow Q\ y)$ *uniformity* $\wedge$
$(\forall x\ y.\ Q\ x \longrightarrow Q\ y \longrightarrow P\ (x,\ y))$
by (*rule-tac exI*[*of* - $\lambda y.$ *Ql* $(y,\ x) \wedge$ *Ql* $(x,\ y)$]) (*fastforce elim*: *eventually-elim2*)

qed
finally show *?thesis*
by (*simp add*: *cauchy-filter-def*)
qed

lemma *LIMSEQ-imp-Cauchy*: $X \longrightarrow x \Longrightarrow$ *Cauchy X*
unfolding *Cauchy-uniform filterlim-def* **by** (*intro nhds-imp-cauchy-filter*)

lemma *Cauchy-subseq-Cauchy*:
assumes *Cauchy X strict-mono f*
shows *Cauchy* $(X \circ f)$
unfolding *Cauchy-uniform comp-def filtermap-filtermap*[*symmetric*] *cauchy-filter-def*
by (*rule order-trans*[*OF - ‹Cauchy X›*[*unfolded Cauchy-uniform cauchy-filter-def*]])
(*intro prod-filter-mono filtermap-mono filterlim-subseq*[*OF ‹strict-mono f›*, *unfolded filterlim-def*])

lemma *convergent-Cauchy*: *convergent X* $\Longrightarrow$ *Cauchy X*
unfolding *convergent-def* **by** (*erule exE*, *erule LIMSEQ-imp-Cauchy*)

definition *complete* :: *'a set* $\Rightarrow$ *bool*
where *complete-uniform*: *complete S* $\longleftrightarrow$
$(\forall F \leq$ *principal S*. $F \neq$ *bot* $\longrightarrow$ *cauchy-filter F* $\longrightarrow (\exists x{\in}S.\ F \leq$ *nhds x*$))$

lemma (**in** *uniform-space*) *cauchy-filter-complete-converges*:
assumes *cauchy-filter F complete A* $F \leq$ *principal A* $F \neq$ *bot*
shows $\exists c.\ F \leq$ *nhds c*
using *assms* **unfolding** *complete-uniform* **by** *blast*

end

99.2.3 Uniformly continuous functions

definition *uniformly-continuous-on* :: *'a set* $\Rightarrow$ (*'a*::*uniform-space* $\Rightarrow$ *'b*::*uniform-space*) $\Rightarrow$ *bool*
where *uniformly-continuous-on-uniformity*: *uniformly-continuous-on s f* $\longleftrightarrow$
(*LIM* (x, y) (*uniformity-on s*). $(f\ x, f\ y)$:> *uniformity*)

lemma *uniformly-continuous-onD*:
uniformly-continuous-on s f $\Longrightarrow$ *eventually E uniformity* $\Longrightarrow$
eventually $(\lambda(x, y).\ x \in s \longrightarrow y \in s \longrightarrow E\ (f\ x, f\ y))$ *uniformity*
by (*simp add*: *uniformly-continuous-on-uniformity filterlim-iff eventually-inf-principal split-beta' mem-Times-iff imp-conjL*)

lemma *uniformly-continuous-on-const*[*continuous-intros*]: *uniformly-continuous-on s* $(\lambda x.\ c)$
by (*auto simp*: *uniformly-continuous-on-uniformity filterlim-iff uniformity-refl*)

lemma *uniformly-continuous-on-id*[*continuous-intros*]: *uniformly-continuous-on s* $(\lambda x.\ x)$

by (*auto simp*: *uniformly-continuous-on-uniformity filterlim-def*)

lemma *uniformly-continuous-on-compose*:
uniformly-continuous-on s g $\Longrightarrow$ *uniformly-continuous-on* (*g*'*s*) *f* $\Longrightarrow$
uniformly-continuous-on s ($\lambda x.$ *f* (*g x*))
using *filterlim-compose*[*of* $\lambda(x, y).$ (*f x*, *f y*) *uniformity*
uniformity-on (*g*'*s*) $\lambda(x, y).$ (*g x*, *g y*) *uniformity-on s*]
by (*simp add*: *split-beta*′ *uniformly-continuous-on-uniformity*
filterlim-inf filterlim-principal eventually-inf-principal mem-Times-iff)

lemma *uniformly-continuous-imp-continuous*:
assumes *f*: *uniformly-continuous-on s f*
shows *continuous-on s f*
by (*auto simp*: *filterlim-iff eventually-at-filter eventually-nhds-uniformity continuous-on-def*
elim: *eventually-mono dest*!: *uniformly-continuous-onD*[*OF f*])

100 Product Topology

100.1 Product is a topological space

instantiation *prod* :: (*topological-space*, *topological-space*) *topological-space*
begin

definition *open-prod-def* [*code drop*]:
open (*S* :: ('*a* × '*b*) *set*) $\longleftrightarrow$
($\forall x{\in}S.$ $\exists A\ B.$ *open A* $\wedge$ *open B* $\wedge$ $x \in A \times B \wedge A \times B \subseteq S$)

lemma *open-prod-elim*:
assumes *open S* **and** $x \in S$
obtains *A B* **where** *open A* **and** *open B* **and** $x \in A \times B$ **and** $A \times B \subseteq S$
using *assms* **unfolding** *open-prod-def* **by** *fast*

lemma *open-prod-intro*:
assumes $\bigwedge x.$ $x \in S \Longrightarrow \exists A\ B.$ *open A* $\wedge$ *open B* $\wedge$ $x \in A \times B \wedge A \times B \subseteq S$
shows *open S*
using *assms* **unfolding** *open-prod-def* **by** *fast*

instance
proof
show *open* (*UNIV* :: ('*a* × '*b*) *set*)
unfolding *open-prod-def* **by** *auto*
next
fix *S T* :: ('*a* × '*b*) *set*
assume *open S open T*
show *open* (*S* ∩ *T*)
proof (*rule open-prod-intro*)
fix *x*
assume *x*: $x \in S \cap T$

```
    from x have x ∈ S by simp
    obtain Sa Sb where A: open Sa open Sb x ∈ Sa × Sb Sa × Sb ⊆ S
      using ‹open S› and ‹x ∈ S› by (rule open-prod-elim)
    from x have x ∈ T by simp
    obtain Ta Tb where B: open Ta open Tb x ∈ Ta × Tb Ta × Tb ⊆ T
      using ‹open T› and ‹x ∈ T› by (rule open-prod-elim)
    let ?A = Sa ∩ Ta and ?B = Sb ∩ Tb
    have open ?A ∧ open ?B ∧ x ∈ ?A × ?B ∧ ?A × ?B ⊆ S ∩ T
      using A B by (auto simp add: open-Int)
    then show ∃ A B. open A ∧ open B ∧ x ∈ A × B ∧ A × B ⊆ S ∩ T
      by fast
  qed
next
  fix K :: ('a × 'b) set set
  assume ∀ S∈K. open S
  then show open (⋃ K)
    unfolding open-prod-def by fast
qed

end

lemma open-Times: open S ⟹ open T ⟹ open (S × T)
  unfolding open-prod-def by auto

lemma fst-vimage-eq-Times: fst -` S = S × UNIV
  by auto

lemma snd-vimage-eq-Times: snd -` S = UNIV × S
  by auto

lemma open-vimage-fst: open S ⟹ open (fst -` S)
  by (simp add: fst-vimage-eq-Times open-Times)

lemma open-vimage-snd: open S ⟹ open (snd -` S)
  by (simp add: snd-vimage-eq-Times open-Times)

lemma closed-vimage-fst: closed S ⟹ closed (fst -` S)
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-fst)

lemma closed-vimage-snd: closed S ⟹ closed (snd -` S)
  unfolding closed-open vimage-Compl [symmetric]
  by (rule open-vimage-snd)

lemma closed-Times: closed S ⟹ closed T ⟹ closed (S × T)
proof -
  have S × T = (fst -` S) ∩ (snd -` T)
    by auto
  then show closed S ⟹ closed T ⟹ closed (S × T)
```

by (*simp add*: *closed-vimage-fst closed-vimage-snd closed-Int*)
qed

lemma *subset-fst-imageI*: $A \times B \subseteq S \Longrightarrow y \in B \Longrightarrow A \subseteq fst\ `\ S$
unfolding *image-def subset-eq* **by** *force*

lemma *subset-snd-imageI*: $A \times B \subseteq S \Longrightarrow x \in A \Longrightarrow B \subseteq snd\ `\ S$
unfolding *image-def subset-eq* **by** *force*

lemma *open-image-fst*:
assumes *open S*
shows *open* (*fst* ` *S*)
proof (*rule openI*)
fix *x*
assume $x \in fst\ `\ S$
then obtain *y* **where** $(x, y) \in S$
by *auto*
then obtain *A B* **where** *open A open B* $x \in A$ $y \in B$ $A \times B \subseteq S$
using ‹*open S*› **unfolding** *open-prod-def* **by** *auto*
from ‹$A \times B \subseteq S$› ‹$y \in B$› **have** $A \subseteq fst\ `\ S$
by (*rule subset-fst-imageI*)
with ‹*open A*› ‹$x \in A$› **have** $open\ A \wedge x \in A \wedge A \subseteq fst\ `\ S$
by *simp*
then show $\exists T.\ open\ T \wedge x \in T \wedge T \subseteq fst\ `\ S$ **..**
qed

lemma *open-image-snd*:
assumes *open S*
shows *open* (*snd* ` *S*)
proof (*rule openI*)
fix *y*
assume $y \in snd\ `\ S$
then obtain *x* **where** $(x, y) \in S$
by *auto*
then obtain *A B* **where** *open A open B* $x \in A$ $y \in B$ $A \times B \subseteq S$
using ‹*open S*› **unfolding** *open-prod-def* **by** *auto*
from ‹$A \times B \subseteq S$› ‹$x \in A$› **have** $B \subseteq snd\ `\ S$
by (*rule subset-snd-imageI*)
with ‹*open B*› ‹$y \in B$› **have** $open\ B \wedge y \in B \wedge B \subseteq snd\ `\ S$
by *simp*
then show $\exists T.\ open\ T \wedge y \in T \wedge T \subseteq snd\ `\ S$ **..**
qed

lemma *nhds-prod*: *nhds* $(a, b) = nhds\ a \times_F nhds\ b$
unfolding *nhds-def*
proof (*subst prod-filter-INF*, *auto intro*!: *antisym INF-greatest simp*: *principal-prod-principal*)
fix *S T*
assume *open S* $a \in S$ *open T* $b \in T$
then show (*INF* $x \in \{S.\ open\ S \wedge (a, b) \in S\}.$ *principal x*) $\leq$ *principal* ($S \times$

T)
by (*intro INF-lower*) (*auto intro*!: *open-Times*)
next
fix *S'*
assume *open S'* (*a*, *b*) ∈ *S'*
then obtain *S T* **where** *open S a* ∈ *S open T b* ∈ *T S* × *T* ⊆ *S'*
by (*auto elim*: *open-prod-elim*)
then show (*INF x* ∈ {*S*. *open S* ∧ *a* ∈ *S*}. *INF y* ∈ {*S*. *open S* ∧ *b* ∈ *S*}.
principal (*x* × *y*)) ≤ *principal S'*
by (*auto intro*!: *INF-lower2*)
qed

100.1.1 Continuity of operations

lemma *tendsto-fst* [*tendsto-intros*]:
assumes (*f* ⟶ *a*) *F*
shows ((λ*x*. *fst* (*f x*)) ⟶ *fst a*) *F*
proof (*rule topological-tendstoI*)
fix *S*
assume *open S* **and** *fst a* ∈ *S*
then have *open* (*fst* −‘ *S*) **and** *a* ∈ *fst* −‘ *S*
by (*simp-all add*: *open-vimage-fst*)
with *assms* **have** *eventually* (λ*x*. *f x* ∈ *fst* −‘ *S*) *F*
by (*rule topological-tendstoD*)
then show *eventually* (λ*x*. *fst* (*f x*) ∈ *S*) *F*
by *simp*
qed

lemma *tendsto-snd* [*tendsto-intros*]:
assumes (*f* ⟶ *a*) *F*
shows ((λ*x*. *snd* (*f x*)) ⟶ *snd a*) *F*
proof (*rule topological-tendstoI*)
fix *S*
assume *open S* **and** *snd a* ∈ *S*
then have *open* (*snd* −‘ *S*) **and** *a* ∈ *snd* −‘ *S*
by (*simp-all add*: *open-vimage-snd*)
with *assms* **have** *eventually* (λ*x*. *f x* ∈ *snd* −‘ *S*) *F*
by (*rule topological-tendstoD*)
then show *eventually* (λ*x*. *snd* (*f x*) ∈ *S*) *F*
by *simp*
qed

lemma *tendsto-Pair* [*tendsto-intros*]:
assumes (*f* ⟶ *a*) *F* **and** (*g* ⟶ *b*) *F*
shows ((λ*x*. (*f x*, *g x*)) ⟶ (*a*, *b*)) *F*
unfolding *nhds-prod* **using** *assms* **by** (*rule filterlim-Pair*)

lemma *continuous-fst*[*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *fst* (*f x*))

unfolding *continuous-def* **by** (*rule tendsto-fst*)

lemma *continuous-snd*[*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *snd* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-snd*)

lemma *continuous-Pair*[*continuous-intros*]:
continuous F f ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. (*f x*, *g x*))
unfolding *continuous-def* **by** (*rule tendsto-Pair*)

lemma *continuous-on-fst*[*continuous-intros*]:
continuous-on s f ⟹ *continuous-on s* (λ*x*. *fst* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-fst*)

lemma *continuous-on-snd*[*continuous-intros*]:
continuous-on s f ⟹ *continuous-on s* (λ*x*. *snd* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-snd*)

lemma *continuous-on-Pair*[*continuous-intros*]:
continuous-on s f ⟹ *continuous-on s g* ⟹ *continuous-on s* (λ*x*. (*f x*, *g x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-Pair*)

lemma *continuous-on-swap*[*continuous-intros*]: *continuous-on A prod.swap*
by (*simp add*: *prod.swap-def continuous-on-fst continuous-on-snd*
continuous-on-Pair continuous-on-id)

lemma *continuous-on-swap-args*:
assumes *continuous-on* (*A*×*B*) (λ(*x*,*y*). *d x y*)
shows *continuous-on* (*B*×*A*) (λ(*x*,*y*). *d y x*)
proof −
have (λ(*x*,*y*). *d y x*) = (λ(*x*,*y*). *d x y*) ∘ *prod.swap*
by *force*
then show *?thesis*
by (*metis assms continuous-on-compose continuous-on-swap product-swap*)
qed

lemma *isCont-fst* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *fst* (*f x*)) *a*
by (*fact continuous-fst*)

lemma *isCont-snd* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *snd* (*f x*)) *a*
by (*fact continuous-snd*)

lemma *isCont-Pair* [*simp*]: ⟦*isCont f a*; *isCont g a*⟧ ⟹ *isCont* (λ*x*. (*f x*, *g x*)) *a*
by (*fact continuous-Pair*)

lemma *continuous-on-compose-Pair*:
assumes *f*: *continuous-on* (*Sigma A B*) (λ(*a*, *b*). *f a b*)
assumes *g*: *continuous-on C g*
assumes *h*: *continuous-on C h*

assumes *subset*: $\bigwedge c.\ c \in C \Longrightarrow g\ c \in A\ \bigwedge c.\ c \in C \Longrightarrow h\ c \in B\ (g\ c)$
shows *continuous-on C* $(\lambda c.\ f\ (g\ c)\ (h\ c))$
using *continuous-on-compose2*[*OF f continuous-on-Pair*[*OF g h*]] *subset*
by *auto*

100.1.2 Connectedness of products

proposition *connected-Times*:
assumes *S*: *connected S* **and** *T*: *connected T*
shows *connected* $(S \times T)$
proof (*rule connectedI-const*)
fix *P*::$'a \times 'b \Rightarrow bool$
assume *P*[*THEN continuous-on-compose2*, *continuous-intros*]: *continuous-on* $(S \times T)$ *P*
have *continuous-on S* $(\lambda s.\ P\ (s,\ t))$ **if** $t \in T$ **for** *t*
by (*auto intro*!: *continuous-intros that*)
from *connectedD-const*[*OF S this*]
obtain *c1* **where** *c1*: $\bigwedge s\ t.\ t \in T \Longrightarrow s \in S \Longrightarrow P\ (s,\ t) = c1\ t$
by *metis*
moreover
have *continuous-on T* $(\lambda t.\ P\ (s,\ t))$ **if** $s \in S$ **for** *s*
by (*auto intro*!: *continuous-intros that*)
from *connectedD-const*[*OF T this*]
obtain *c2* **where** $\bigwedge s\ t.\ t \in T \Longrightarrow s \in S \Longrightarrow P\ (s,\ t) = c2\ s$
by *metis*
ultimately show $\exists c.\ \forall s{\in}S \times T.\ P\ s = c$
by *auto*
qed

corollary *connected-Times-eq* [*simp*]:
connected $(S \times T) \longleftrightarrow S = \{\} \vee T = \{\} \vee$ *connected S* $\wedge$ *connected T* (**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
show *?rhs*
proof *cases*
assume $S \neq \{\} \wedge T \neq \{\}$
moreover
have *connected* (*fst* ' $(S \times T)$) *connected* (*snd* ' $(S \times T)$)
using *continuous-on-fst continuous-on-snd continuous-on-id*
by (*blast intro*: *connected-continuous-image* [*OF - L*])+
ultimately show *?thesis*
by *auto*
qed *auto*
qed (*auto simp*: *connected-Times*)

100.1.3 Separation axioms

instance *prod* :: (*t0-space*, *t0-space*) *t0-space*
proof

fix $x\ y :: 'a \times 'b$
assume $x \neq y$
then have $fst\ x \neq fst\ y \vee snd\ x \neq snd\ y$
by (*simp add*: *prod-eq-iff*)
then show $\exists\, U.\ open\ U \wedge (x \in U) \neq (y \in U)$
by (*fast dest*: *t0-space elim*: *open-vimage-fst open-vimage-snd*)
qed

instance *prod* :: (*t1-space*, *t1-space*) *t1-space*
proof
fix $x\ y :: 'a \times 'b$
assume $x \neq y$
then have $fst\ x \neq fst\ y \vee snd\ x \neq snd\ y$
by (*simp add*: *prod-eq-iff*)
then show $\exists\, U.\ open\ U \wedge x \in U \wedge y \notin U$
by (*fast dest*: *t1-space elim*: *open-vimage-fst open-vimage-snd*)
qed

instance *prod* :: (*t2-space*, *t2-space*) *t2-space*
proof
fix $x\ y :: 'a \times 'b$
assume $x \neq y$
then have $fst\ x \neq fst\ y \vee snd\ x \neq snd\ y$
by (*simp add*: *prod-eq-iff*)
then show $\exists\, U\ V.\ open\ U \wedge open\ V \wedge x \in U \wedge y \in V \wedge U \cap V = \{\}$
by (*fast dest*: *hausdorff elim*: *open-vimage-fst open-vimage-snd*)
qed

lemma *isCont-swap*[*continuous-intros*]: *isCont prod.swap a*
using *continuous-on-eq-continuous-within continuous-on-swap* **by** *blast*

lemma *open-diagonal-complement*:
open $\{(x,y)\ |x\ y.\ x \neq (y::('a::t2\text{-}space))\}$
proof −
have *open* $\{(x,\ y).\ x \neq (y::'a)\}$
unfolding *split-def* **by** (*intro open-Collect-neq continuous-intros*)
also have $\{(x,\ y).\ x \neq (y::'a)\} = \{(x,\ y)\ |x\ y.\ x \neq (y::'a)\}$
by *auto*
finally show *?thesis* **.**
qed

lemma *closed-diagonal*:
closed $\{y.\ \exists\ x::('a::t2\text{-}space).\ y = (x,x)\}$
proof −
have $\{y.\ \exists\ x::'a.\ y = (x,x)\} = UNIV - \{(x,y)\ |\ x\ y.\ x \neq y\}$ **by** *auto*
then show *?thesis* **using** *open-diagonal-complement closed-Diff* **by** *auto*
qed

lemma *open-superdiagonal*:

open {(*x*,*y*) | *x y*. *x* > (*y*::′*a*::{*linorder-topology*})}
proof −
have *open* {(*x*, *y*). *x* > (*y*::′*a*)}
unfolding *split-def* **by** (*intro open-Collect-less continuous-intros*)
also have {(*x*, *y*). *x* > (*y*::′*a*)} = {(*x*, *y*) |*x y*. *x* > (*y*::′*a*)}
by *auto*
finally show *?thesis* **.**
qed

lemma *closed-subdiagonal*:
closed {(*x*,*y*) | *x y*. *x* ≤ (*y*::′*a*::{*linorder-topology*})}
proof −
have {(*x*,*y*) | *x y*. *x* ≤ (*y*::′*a*)} = *UNIV* − {(*x*,*y*) | *x y*. *x* > (*y*::′*a*)} **by** *auto*
then show *?thesis* **using** *open-superdiagonal closed-Diff* **by** *auto*
qed

lemma *open-subdiagonal*:
open {(*x*,*y*) | *x y*. *x* < (*y*::′*a*::{*linorder-topology*})}
proof −
have *open* {(*x*, *y*). *x* < (*y*::′*a*)}
unfolding *split-def* **by** (*intro open-Collect-less continuous-intros*)
also have {(*x*, *y*). *x* < (*y*::′*a*)} = {(*x*, *y*) |*x y*. *x* < (*y*::′*a*)}
by *auto*
finally show *?thesis* **.**
qed

lemma *closed-superdiagonal*:
closed {(*x*,*y*) | *x y*. *x* ≥ (*y*::(′*a*::{*linorder-topology*}))}
proof −
have {(*x*,*y*) | *x y*. *x* ≥ (*y*::′*a*)} = *UNIV* − {(*x*,*y*) | *x y*. *x* < *y*} **by** *auto*
then show *?thesis* **using** *open-subdiagonal closed-Diff* **by** *auto*
qed

end

theory *Hull*
imports *Main*
begin

100.2 A generic notion of the convex, affine, conic hull, or closed "hull".

definition *hull* :: (′*a set* ⇒ *bool*) ⇒ ′*a set* ⇒ ′*a set* (**infixl** ‹*hull*› *75*)
where *S hull s* = ⋂{*t*. *S t* ∧ *s* ⊆ *t*}

lemma *hull-same*: *S s* ⟹ *S hull s* = *s*
unfolding *hull-def* **by** *auto*

lemma *hull-in*: $(\bigwedge T.\ Ball\ T\ S \Longrightarrow S\ (\bigcap T)) \Longrightarrow S\ (S\ hull\ s)$
unfolding *hull-def Ball-def* **by** *auto*

lemma *hull-eq*: $(\bigwedge T.\ Ball\ T\ S \Longrightarrow S\ (\bigcap T)) \Longrightarrow (S\ hull\ s) = s \longleftrightarrow S\ s$
using *hull-same*[*of S s*] *hull-in*[*of S s*] **by** *metis*

lemma *hull-hull* [*simp*]: $S\ hull\ (S\ hull\ s) = S\ hull\ s$
unfolding *hull-def* **by** *blast*

lemma *hull-subset*[*intro*]: $s \subseteq (S\ hull\ s)$
unfolding *hull-def* **by** *blast*

lemma *hull-mono*: $s \subseteq t \Longrightarrow (S\ hull\ s) \subseteq (S\ hull\ t)$
unfolding *hull-def* **by** *blast*

lemma *hull-antimono*: $\forall x.\ S\ x \longrightarrow T\ x \Longrightarrow (T\ hull\ s) \subseteq (S\ hull\ s)$
unfolding *hull-def* **by** *blast*

lemma *hull-minimal*: $s \subseteq t \Longrightarrow S\ t \Longrightarrow (S\ hull\ s) \subseteq t$
unfolding *hull-def* **by** *blast*

lemma *subset-hull*: $S\ t \Longrightarrow S\ hull\ s \subseteq t \longleftrightarrow s \subseteq t$
unfolding *hull-def* **by** *blast*

lemma *hull-UNIV* [*simp*]: $S\ hull\ UNIV = UNIV$
unfolding *hull-def* **by** *auto*

lemma *hull-unique*: $s \subseteq t \Longrightarrow S\ t \Longrightarrow (\bigwedge t'.\ s \subseteq t' \Longrightarrow S\ t' \Longrightarrow t \subseteq t') \Longrightarrow (S\ hull\ s = t)$
unfolding *hull-def* **by** *auto*

lemma *hull-induct*: $[\![a \in Q\ hull\ S;\ \bigwedge x.\ x \in S \Longrightarrow P\ x;\ Q\ \{x.\ P\ x\}]\!] \Longrightarrow P\ a$
using *hull-minimal*[*of S* $\{x.\ P\ x\}$ *Q*]
by (*auto simp add*: *subset-eq*)

lemma *hull-inc*: $x \in S \Longrightarrow x \in P\ hull\ S$
by (*metis hull-subset subset-eq*)

lemma *hull-Un-subset*: $(S\ hull\ s) \cup (S\ hull\ t) \subseteq (S\ hull\ (s \cup t))$
unfolding *Un-subset-iff* **by** (*metis hull-mono Un-upper1 Un-upper2*)

lemma *hull-Un*:
assumes *T*: $\bigwedge T.\ Ball\ T\ S \Longrightarrow S\ (\bigcap T)$
shows $S\ hull\ (s \cup t) = S\ hull\ (S\ hull\ s \cup S\ hull\ t)$
apply (*rule equalityI*)
apply (*meson hull-mono hull-subset sup.mono*)
by (*metis hull-Un-subset hull-hull hull-mono*)

lemma *hull-Un-left*: $P\ hull\ (S \cup T) = P\ hull\ (P\ hull\ S \cup T)$

apply (*rule equalityI*)
apply (*simp add*: *Un-commute hull-mono hull-subset sup.coboundedI2*)
by (*metis Un-subset-iff hull-hull hull-mono hull-subset*)

lemma *hull-Un-right*: *P hull* (*S* ∪ *T*) = *P hull* (*S* ∪ *P hull T*)
by (*metis hull-Un-left sup.commute*)

lemma *hull-insert*:
P hull (*insert a S*) = *P hull* (*insert a* (*P hull S*))
by (*metis hull-Un-right insert-is-Un*)

lemma *hull-redundant-eq*: *a* ∈ (*S hull s*) ⟷ *S hull* (*insert a s*) = *S hull s*
unfolding *hull-def* **by** *blast*

lemma *hull-redundant*: *a* ∈ (*S hull s*) ⟹ *S hull* (*insert a s*) = *S hull s*
by (*metis hull-redundant-eq*)

end

101 Modules

Bases of a linear algebra based on modules (i.e. vector spaces of rings).

theory *Modules*
imports *Hull*
begin

101.1 Locale for additive functions

locale *additive* =
fixes *f* :: *'a*::*ab-group-add* ⇒ *'b*::*ab-group-add*
assumes *add*: *f* (*x* + *y*) = *f x* + *f y*
begin

lemma *zero*: *f 0* = *0*
proof −
have *f 0* = *f* (*0* + *0*) **by** *simp*
also have ... = *f 0* + *f 0* **by** (*rule add*)
finally show *f 0* = *0* **by** *simp*
qed

lemma *minus*: *f* (− *x*) = − *f x*
proof −
have *f* (− *x*) + *f x* = *f* (− *x* + *x*) **by** (*rule add* [*symmetric*])
also have ... = − *f x* + *f x* **by** (*simp add*: *zero*)
finally show *f* (− *x*) = − *f x* **by** (*rule add-right-imp-eq*)
qed

lemma *diff*: *f* (*x* − *y*) = *f x* − *f y*

using *add* [*of* $x - y$] **by** (*simp add*: *minus*)

lemma *sum*: $f\ (sum\ g\ A) = (\sum x{\in}A.\ f\ (g\ x))$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *zero add*)

end

Modules form the central spaces in linear algebra. They are a generalization from vector spaces by replacing the scalar field by a scalar ring.

locale *module* =
fixes *scale* :: *'a*::*comm-ring-1* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*s*› *75*)
assumes *scale-right-distrib* [*algebra-simps*, *algebra-split-simps*]:
$a *s (x + y) = a *s x + a *s y$
and *scale-left-distrib* [*algebra-simps*, *algebra-split-simps*]:
$(a + b) *s x = a *s x + b *s x$
and *scale-scale* [*simp*]: $a *s (b *s x) = (a * b) *s x$
and *scale-one* [*simp*]: $1 *s x = x$
begin

lemma *scale-left-commute*: $a *s (b *s x) = b *s (a *s x)$
by (*simp add*: *mult.commute*)

lemma *scale-zero-left* [*simp*]: $0 *s x = 0$
and *scale-minus-left* [*simp*]: $(- a) *s x = - (a *s x)$
and *scale-left-diff-distrib* [*algebra-simps*, *algebra-split-simps*]:
$(a - b) *s x = a *s x - b *s x$
and *scale-sum-left*: $(sum\ f\ A) *s x = (\sum a{\in}A.\ (f\ a) *s x)$
proof −
interpret *s*: *additive* $\lambda a.\ a *s x$
by *standard* (*rule scale-left-distrib*)
show $0 *s x = 0$ **by** (*rule s.zero*)
show $(- a) *s x = - (a *s x)$ **by** (*rule s.minus*)
show $(a - b) *s x = a *s x - b *s x$ **by** (*rule s.diff*)
show $(sum\ f\ A) *s x = (\sum a{\in}A.\ (f\ a) *s x)$ **by** (*rule s.sum*)
qed

lemma *scale-zero-right* [*simp*]: $a *s 0 = 0$
and *scale-minus-right* [*simp*]: $a *s (- x) = - (a *s x)$
and *scale-right-diff-distrib* [*algebra-simps*, *algebra-split-simps*]:
$a *s (x - y) = a *s x - a *s y$
and *scale-sum-right*: $a *s (sum\ f\ A) = (\sum x{\in}A.\ a *s (f\ x))$
proof −
interpret *s*: *additive* $\lambda x.\ a *s x$
by *standard* (*rule scale-right-distrib*)
show $a *s 0 = 0$ **by** (*rule s.zero*)
show $a *s (- x) = - (a *s x)$ **by** (*rule s.minus*)
show $a *s (x - y) = a *s x - a *s y$ **by** (*rule s.diff*)
show $a *s (sum\ f\ A) = (\sum x{\in}A.\ a *s (f\ x))$ **by** (*rule s.sum*)
qed

lemma *sum-constant-scale*: $(\sum x{\in}A.\ y) = \mathit{scale}\ (\mathit{of\text{-}nat}\ (\mathit{card}\ A))\ y$
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *algebra-simps*)

end

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*divide*›, *SOME* ***typ*** ‹$'a \Rightarrow 'a \Rightarrow 'a$›)›

context *module*
begin

lemma [*field-simps*, *field-split-simps*]:
shows *scale-left-distrib-NO-MATCH*: *NO-MATCH* $(x\ \mathit{div}\ y)\ c \Longrightarrow (a + b) *s x = a *s x + b *s x$
and *scale-right-distrib-NO-MATCH*: *NO-MATCH* $(x\ \mathit{div}\ y)\ a \Longrightarrow a *s (x + y) = a *s x + a *s y$
and *scale-left-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ \mathit{div}\ y)\ c \Longrightarrow (a - b) *s x = a *s x - b *s x$
and *scale-right-diff-distrib-NO-MATCH*: *NO-MATCH* $(x\ \mathit{div}\ y)\ a \Longrightarrow a *s (x - y) = a *s x - a *s y$
by (*rule scale-left-distrib scale-right-distrib scale-left-diff-distrib scale-right-diff-distrib*)+

end

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*divide*›, *SOME* ***typ*** ‹$'a{::}\mathit{divide} \Rightarrow 'a \Rightarrow 'a$›)›

102 Subspace

context *module*
begin

definition *subspace* :: $'b\ \mathit{set} \Rightarrow \mathit{bool}$
where $\mathit{subspace}\ S \longleftrightarrow 0 \in S \wedge (\forall x{\in}S.\ \forall y{\in}S.\ x + y \in S) \wedge (\forall c.\ \forall x{\in}S.\ c *s x \in S)$

lemma *subspaceI*:
$0 \in S \Longrightarrow (\bigwedge x\ y.\ x \in S \Longrightarrow y \in S \Longrightarrow x + y \in S) \Longrightarrow (\bigwedge c\ x.\ x \in S \Longrightarrow c *s x \in S) \Longrightarrow \mathit{subspace}\ S$
by (*auto simp*: *subspace-def*)

lemma *subspace-UNIV*[*simp*]: *subspace UNIV*
by (*simp add*: *subspace-def*)

lemma *subspace-single-0*[*simp*]: *subspace* $\{0\}$
by (*simp add*: *subspace-def*)

lemma *subspace-0*: $\mathit{subspace}\ S \Longrightarrow 0 \in S$

by (*metis subspace-def*)

lemma *subspace-add*: *subspace* $S \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow x + y \in S$
by (*metis subspace-def*)

lemma *subspace-scale*: *subspace* $S \Longrightarrow x \in S \Longrightarrow c *s x \in S$
by (*metis subspace-def*)

lemma *subspace-neg*: *subspace* $S \Longrightarrow x \in S \Longrightarrow - x \in S$
by (*metis scale-minus-left scale-one subspace-scale*)

lemma *subspace-diff*: *subspace* $S \Longrightarrow x \in S \Longrightarrow y \in S \Longrightarrow x - y \in S$
by (*metis diff-conv-add-uminus subspace-add subspace-neg*)

lemma *subspace-sum*: *subspace* $A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \in A) \Longrightarrow sum\ f\ B \in A$
by (*induct B rule*: *infinite-finite-induct*) (*auto simp add*: *subspace-add subspace-0*)

lemma *subspace-Int*: $(\bigwedge i.\ i \in I \Longrightarrow subspace\ (s\ i)) \Longrightarrow subspace\ (\bigcap i{\in}I.\ s\ i)$
by (*auto simp*: *subspace-def*)

lemma *subspace-Inter*: $\forall s \in f.\ subspace\ s \Longrightarrow subspace\ (\bigcap f)$
unfolding *subspace-def* **by** *auto*

lemma *subspace-inter*: *subspace* $A \Longrightarrow$ *subspace* $B \Longrightarrow$ *subspace* $(A \cap B)$
by (*simp add*: *subspace-def*)

103 Span: subspace generated by a set

definition *span* :: *'b set* ⇒ *'b set*
where *span-explicit*: *span* $b = \{(\sum a{\in}t.\ r\ a *s\ \ a) \mid t\ r.\ finite\ t \wedge t \subseteq b\}$

lemma *span-explicit'*:
span $b = \{(\sum v \mid f\ v \neq 0.\ f\ v *s\ v) \mid f.\ finite\ \{v.\ f\ v \neq 0\} \wedge (\forall v.\ f\ v \neq 0 \longrightarrow v \in b)\}$
unfolding *span-explicit*
proof *safe*
fix *t r* **assume** *finite t t* ⊆ *b*
then show $\exists f.\ (\sum a{\in}t.\ r\ a *s\ a) = (\sum v \mid f\ v \neq 0.\ f\ v *s\ v) \wedge finite\ \{v.\ f\ v \neq 0\} \wedge (\forall v.\ f\ v \neq 0 \longrightarrow v \in b)$
by (*intro exI*[*of* - $\lambda v.$ *if v* ∈ *t then r v else 0*]) (*auto intro*!: *sum.mono-neutral-cong-right*)
next
fix *f* :: *'b* ⇒ *'a* **assume** *finite* $\{v.\ f\ v \neq 0\}$ $(\forall v.\ f\ v \neq 0 \longrightarrow v \in b)$
then show $\exists t\ r.\ (\sum v \mid f\ v \neq 0.\ f\ v *s\ v) = (\sum a{\in}t.\ r\ a *s\ a) \wedge finite\ t \wedge t \subseteq b$
by (*intro exI*[*of* - $\{v.\ f\ v \neq 0\}$] *exI*[*of* - *f*]) *auto*
qed

lemma *span-alt*:
span $B = \{(\sum x \mid f\ x \neq 0.\ f\ x *s\ x) \mid f.\ \{x.\ f\ x \neq 0\} \subseteq B \wedge finite\ \{x.\ f\ x \neq 0\}\}$
unfolding *span-explicit'* **by** *auto*

lemma *span-finite*:
assumes *fS*: *finite S*
shows *span S* = *range* (λ*u*. ∑ *v*∈*S*. *u v* ∗*s v*)
unfolding *span-explicit*
proof *safe*
fix *t r* **assume** *t* ⊆ *S* **then show** (∑ *a*∈*t*. *r a* ∗*s a*) ∈ *range* (λ*u*. ∑ *v*∈*S*. *u v* ∗*s v*)
by (*intro image-eqI*[*of* - - λ*a*. *if a* ∈ *t then r a else 0*])
(*auto simp*: *if-distrib*[*of* λ*r*. *r* ∗*s a* **for** *a*] *sum.If-cases fS Int-absorb1*)
next
show ∃ *t r*. (∑ *v*∈*S*. *u v* ∗*s v*) = (∑ *a*∈*t*. *r a* ∗*s a*) ∧ *finite t* ∧ *t* ⊆ *S* **for** *u*
by (*intro exI*[*of* - *u*] *exI*[*of* - *S*]) (*auto intro*: *fS*)
qed

lemma *span-induct-alt* [*consumes 1*, *case-names base step*, *induct set*: *span*]:
assumes *x*: *x* ∈ *span S*
assumes *h0*: *h 0* **and** *hS*: ⋀*c x y*. *x* ∈ *S* ⟹ *h y* ⟹ *h* (*c* ∗*s x* + *y*)
shows *h x*
using *x* **unfolding** *span-explicit*
proof *safe*
fix *t r* **assume** *finite t t* ⊆ *S* **then show** *h* (∑ *a*∈*t*. *r a* ∗*s a*)
by (*induction t*) (*auto intro*!: *hS h0*)
qed

lemma *span-mono*: *A* ⊆ *B* ⟹ *span A* ⊆ *span B*
by (*auto simp*: *span-explicit*)

lemma *span-base*: *a* ∈ *S* ⟹ *a* ∈ *span S*
by (*auto simp*: *span-explicit intro*!: *exI*[*of* - {*a*}] *exI*[*of* - λ-. *1*])

lemma *span-superset*: *S* ⊆ *span S*
by (*auto simp*: *span-base*)

lemma *span-zero*: *0* ∈ *span S*
by (*auto simp*: *span-explicit intro*!: *exI*[*of* - {}])

lemma *span-UNIV*[*simp*]: *span UNIV* = *UNIV*
by (*auto intro*: *span-base*)

lemma *span-add*: *x* ∈ *span S* ⟹ *y* ∈ *span S* ⟹ *x* + *y* ∈ *span S*
unfolding *span-explicit*
proof *safe*
fix *tx ty rx ry* **assume** ∗: *finite tx finite ty tx* ⊆ *S ty* ⊆ *S*
have [*simp*]: (*tx* ∪ *ty*) ∩ *tx* = *tx* (*tx* ∪ *ty*) ∩ *ty* = *ty*
by *auto*
show ∃ *t r*. (∑ *a*∈*tx*. *rx a* ∗*s a*) + (∑ *a*∈*ty*. *ry a* ∗*s a*) = (∑ *a*∈*t*. *r a* ∗*s a*) ∧ *finite t* ∧ *t* ⊆ *S*
apply (*intro exI*[*of* - *tx* ∪ *ty*])

apply (*intro exI*[*of* - $\lambda a.$ (*if* $a \in tx$ *then* $rx\ a$ *else* 0) + (*if* $a \in ty$ *then* $ry\ a$ *else* 0)])
apply (*auto simp*: $*$ *scale-left-distrib sum.distrib if-distrib*[*of* $\lambda r.\ r *s a$ **for** a] *sum.If-cases*)
done
qed

lemma *span-scale*: $x \in span\ S \Longrightarrow c *s x \in span\ S$
unfolding *span-explicit*
proof *safe*
fix $t\ r$ **assume** $*$: *finite* t $t \subseteq S$
show $\exists t'\ r'.\ c *s (\sum a \in t.\ r\ a *s a) = (\sum a \in t'.\ r'\ a *s a) \wedge$ *finite* $t' \wedge t' \subseteq S$
by (*intro exI*[*of* - t] *exI*[*of* - $\lambda a.\ c * r\ a$]) (*auto simp*: $*$ *scale-sum-right*)
qed

lemma *subspace-span* [*iff*]: *subspace* (*span* S)
by (*auto simp*: *subspace-def span-zero span-add span-scale*)

lemma *span-neg*: $x \in span\ S \Longrightarrow - x \in span\ S$
by (*metis subspace-neg subspace-span*)

lemma *span-diff*: $x \in span\ S \Longrightarrow y \in span\ S \Longrightarrow x - y \in span\ S$
by (*metis subspace-span subspace-diff*)

lemma *span-sum*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in span\ S) \Longrightarrow sum\ f\ A \in span\ S$
by (*rule subspace-sum*, *rule subspace-span*)

lemma *span-minimal*: $S \subseteq T \Longrightarrow subspace\ T \Longrightarrow span\ S \subseteq T$
by (*auto simp*: *span-explicit intro*!: *subspace-sum subspace-scale*)

lemma *span-def*: *span* S = *subspace hull* S
by (*intro hull-unique*[*symmetric*] *span-superset subspace-span span-minimal*)

lemma *span-unique*:
$S \subseteq T \Longrightarrow subspace\ T \Longrightarrow (\bigwedge T'.\ S \subseteq T' \Longrightarrow subspace\ T' \Longrightarrow T \subseteq T') \Longrightarrow span\ S = T$
unfolding *span-def* **by** (*rule hull-unique*)

lemma *span-subspace-induct*[*consumes 2*]:
assumes x: $x \in span\ S$
and P: *subspace* P
and SP: $\bigwedge x.\ x \in S \Longrightarrow x \in P$
shows $x \in P$
proof −
from SP **have** SP': $S \subseteq P$
by (*simp add*: *subset-eq*)
from x *hull-minimal*[**where** S=*subspace*, *OF* SP' P, *unfolded span-def*[*symmetric*]]
show $x \in P$
by (*metis subset-eq*)

qed

lemma (**in** *module*) *span-induct*[*consumes 1*, *case-names base step*, *induct set*: *span*]:
  assumes x: $x \in span\ S$
    and P: *subspace* (*Collect* P)
    and SP: $\bigwedge x.\ x \in S \Longrightarrow P\ x$
  shows $P\ x$
  using P SP *span-subspace-induct* x **by** *fastforce*

lemma *span-empty*[*simp*]: $span\ \{\} = \{0\}$
  by (*rule span-unique*) (*auto simp add*: *subspace-def*)

lemma *span-subspace*: $A \subseteq B \Longrightarrow B \subseteq span\ A \Longrightarrow subspace\ B \Longrightarrow span\ A = B$
  by (*metis order-antisym span-def hull-minimal*)

lemma *span-span*: $span\ (span\ A) = span\ A$
  unfolding *span-def hull-hull* **..**

lemma *span-add-eq*: **assumes** x: $x \in span\ S$ **shows** $x + y \in span\ S \longleftrightarrow y \in span\ S$
proof
  assume $*$: $x + y \in span\ S$
  have $(x + y) - x \in span\ S$ **using** $*$ x **by** (*rule span-diff*)
  then show $y \in span\ S$ **by** *simp*
qed (*intro span-add* x)

lemma *span-add-eq2*: **assumes** y: $y \in span\ S$ **shows** $x + y \in span\ S \longleftrightarrow x \in span\ S$
  using *span-add-eq*[*of* y S x] y **by** (*auto simp*: *ac-simps*)

lemma *span-singleton*: $span\ \{x\} = range\ (\lambda k.\ k *s x)$
  by (*auto simp*: *span-finite*)

lemma *span-Un*: $span\ (S \cup T) = \{x + y \mid x\ y.\ x \in span\ S \wedge y \in span\ T\}$
proof *safe*
  fix x **assume** $x \in span\ (S \cup T)$
  then obtain t r **where** t: *finite* t $t \subseteq S \cup T$ **and** x: $x = (\sum a \in t.\ r\ a *s a)$
    by (*auto simp*: *span-explicit*)
  moreover have $t \cap S \cup (t - S) = t$ **by** *auto*
  ultimately show $\exists xa\ y.\ x = xa + y \wedge xa \in span\ S \wedge y \in span\ T$
    unfolding x
    apply (*rule-tac exI*[*of* - $\sum a \in t \cap S.\ r\ a *s a$])
    apply (*rule-tac exI*[*of* - $\sum a \in t - S.\ r\ a *s a$])
    apply (*subst sum.union-inter-neutral*[*symmetric*])
    apply (*auto intro*!: *span-sum span-scale intro*: *span-base*)
    done
next

fix x y **assume**$x \in span\ S\ y \in span\ T$ **then show** $x + y \in span\ (S \cup T)$
using *span-mono*[*of* $S\ S \cup T$] *span-mono*[*of* $T\ S \cup T$]
by (*auto intro*!: *span-add*)
qed

lemma *span-insert*: $span\ (insert\ a\ S) = \{x.\ \exists k.\ (x - k *s a) \in span\ S\}$
proof −
have $span\ (\{a\} \cup S) = \{x.\ \exists k.\ (x - k *s a) \in span\ S\}$
unfolding *span-Un span-singleton*
apply (*auto simp add*: *set-eq-iff*)
subgoal for $y\ k$ **by** (*auto intro*!: *exI*[*of* - k])
subgoal for $y\ k$ **by** (*rule exI*[*of* - $k *s a$], *rule exI*[*of* - $y - k *s a$]) *auto*
done
then show *?thesis* **by** *simp*
qed

lemma *span-breakdown*:
assumes bS: $b \in S$
and aS: $a \in span\ S$
shows $\exists k.\ a - k *s b \in span\ (S - \{b\})$
using *assms span-insert* [*of* $b\ S - \{b\}$]
by (*simp add*: *insert-absorb*)

lemma *span-breakdown-eq*: $x \in span\ (insert\ a\ S) \longleftrightarrow (\exists k.\ x - k *s a \in span\ S)$
by (*simp add*: *span-insert*)

lemmas *span-clauses* = *span-base span-zero span-add span-scale*

lemma *span-eq-iff*[*simp*]: $span\ s = s \longleftrightarrow subspace\ s$
unfolding *span-def* **by** (*rule hull-eq*) (*rule subspace-Inter*)

lemma *span-eq*: $span\ S = span\ T \longleftrightarrow S \subseteq span\ T \wedge T \subseteq span\ S$
by (*metis span-minimal span-subspace span-superset subspace-span*)

lemma *eq-span-insert-eq*:
assumes $(x - y) \in span\ S$
shows $span(insert\ x\ S) = span(insert\ y\ S)$
proof −
have $*$: $span(insert\ x\ S) \subseteq span(insert\ y\ S)$ **if** $(x - y) \in span\ S$ **for** $x\ y$
proof −
have *1*: $(r *s x - r *s y) \in span\ S$ **for** r
by (*metis scale-right-diff-distrib span-scale that*)
have *2*: $(z - k *s y) - k *s (x - y) = z - k *s x$ **for** $z\ k$
by (*simp add*: *scale-right-diff-distrib*)
show *?thesis*
apply (*clarsimp simp add*: *span-breakdown-eq*)
by (*metis 1 2 diff-add-cancel scale-right-diff-distrib span-add-eq*)
qed
show *?thesis*

apply (*intro subset-antisym* ∗ *assms*)
using *assms subspace-neg subspace-span minus-diff-eq* **by** *force*
qed

104 Dependent and independent sets

definition *dependent* :: *'b set* ⇒ *bool*
where *dependent-explicit*: *dependent s* ⟷ (∃ *t u*. *finite t* ∧ *t* ⊆ *s* ∧ (∑ *v*∈*t*. *u v* ∗*s v*) = *0* ∧ (∃ *v*∈*t*. *u v* ≠ *0*))

abbreviation *independent s* ≡ ¬ *dependent s*

lemma *dependent-mono*: *dependent B* ⟹ *B* ⊆ *A* ⟹ *dependent A*
by (*auto simp add*: *dependent-explicit*)

lemma *independent-mono*: *independent A* ⟹ *B* ⊆ *A* ⟹ *independent B*
by (*auto intro*: *dependent-mono*)

lemma *dependent-zero*: *0* ∈ *A* ⟹ *dependent A*
by (*auto simp*: *dependent-explicit intro*!: *exI*[*of* - λ*i*. *1*] *exI*[*of* - {*0*}])

lemma *independent-empty*[*intro*]: *independent* {}
by (*simp add*: *dependent-explicit*)

lemma *independent-explicit-module*:
independent s ⟷ (∀ *t u v*. *finite t* ⟶ *t* ⊆ *s* ⟶ (∑ *v*∈*t*. *u v* ∗*s v*) = *0* ⟶ *v* ∈ *t* ⟶ *u v* = *0*)
unfolding *dependent-explicit* **by** *auto*

lemma *independentD*: *independent s* ⟹ *finite t* ⟹ *t* ⊆ *s* ⟹ (∑ *v*∈*t*. *u v* ∗*s v*) = *0* ⟹ *v* ∈ *t* ⟹ *u v* = *0*
by (*simp add*: *independent-explicit-module*)

lemma *independent-Union-directed*:
assumes *directed*: ⋀*c d*. *c* ∈ *C* ⟹ *d* ∈ *C* ⟹ *c* ⊆ *d* ∨ *d* ⊆ *c*
assumes *indep*: ⋀*c*. *c* ∈ *C* ⟹ *independent c*
shows *independent* (⋃ *C*)
proof
assume *dependent* (⋃ *C*)
then obtain *u v S* **where** *S*: *finite S S* ⊆ ⋃ *C v* ∈ *S u v* ≠ *0* (∑ *v*∈*S*. *u v* ∗*s v*) = *0*
by (*auto simp*: *dependent-explicit*)

have *S* ≠ {}
using ‹*v* ∈ *S*› **by** *auto*
have ∃ *c*∈*C*. *S* ⊆ *c*
using ‹*finite S*› ‹*S* ≠ {}› ‹*S* ⊆ ⋃ *C*›
proof (*induction rule*: *finite-ne-induct*)
case (*insert i I*)

```
    then obtain c d where cd: c ∈ C d ∈ C and iI: I ⊆ c i ∈ d
      by blast
    from directed[OF cd] cd have c ∪ d ∈ C
      by (auto simp: sup.absorb1 sup.absorb2)
    with iI show ?case
      by (intro bexI[of - c ∪ d]) auto
  qed auto
  then obtain c where c ∈ C S ⊆ c
    by auto
  have dependent c
    unfolding dependent-explicit
    by (intro exI[of - S] exI[of - u] bexI[of - v] conjI) fact+
  with indep[OF ‹c ∈ C›] show False
    by auto
qed

lemma dependent-finite:
  assumes finite S
  shows dependent S ⟷ (∃ u. (∃ v ∈ S. u v ≠ 0) ∧ (∑ v∈S. u v *s v) = 0)
    (is ?lhs = ?rhs)
proof
  assume ?lhs
  then obtain T u v
    where finite T T ⊆ S v∈T u v ≠ 0 (∑ v∈T. u v *s v) = 0
    by (force simp: dependent-explicit)
  with assms show ?rhs
    apply (rule-tac x=λv. if v ∈ T then u v else 0 in exI)
    apply (auto simp: sum.mono-neutral-right)
    done
next
  assume ?rhs  with assms show ?lhs
    by (fastforce simp add: dependent-explicit)
qed

lemma dependent-alt:
  dependent B ⟷
    (∃ X. finite {x. X x ≠ 0} ∧ {x. X x ≠ 0} ⊆ B ∧ (∑ x|X x ≠ 0. X x *s x) =
0 ∧ (∃ x. X x ≠ 0))
  unfolding dependent-explicit
  apply safe
  subgoal for S u v
    apply (intro exI[of - λx. if x ∈ S then u x else 0])
    apply (subst sum.mono-neutral-cong-left[where T=S])
    apply (auto intro!: sum.mono-neutral-cong-right cong: rev-conj-cong)
    done
  apply auto
  done

lemma independent-alt:
```

independent B $\longleftrightarrow$
(∀ *X*. *finite* {*x*. *X x* ≠ *0*} $\longrightarrow$ {*x*. *X x* ≠ *0*} ⊆ *B* $\longrightarrow$ (∑ *x*|*X x* ≠ *0*. *X x* ∗*s* *x*) = *0* $\longrightarrow$ (∀ *x*. *X x* = *0*))
unfolding *dependent-alt* **by** *auto*

lemma *independentD-alt*:
independent B ⟹ *finite* {*x*. *X x* ≠ *0*} ⟹ {*x*. *X x* ≠ *0*} ⊆ *B* ⟹ (∑ *x*|*X x* ≠ *0*. *X x* ∗*s* *x*) = *0* ⟹ *X x* = *0*
unfolding *independent-alt* **by** *blast*

lemma *independentD-unique*:
assumes *B*: *independent B*
and *X*: *finite* {*x*. *X x* ≠ *0*} {*x*. *X x* ≠ *0*} ⊆ *B*
and *Y*: *finite* {*x*. *Y x* ≠ *0*} {*x*. *Y x* ≠ *0*} ⊆ *B*
and (∑ *x* | *X x* ≠ *0*. *X x* ∗*s* *x*) = (∑ *x*| *Y x* ≠ *0*. *Y x* ∗*s* *x*)
shows *X* = *Y*
proof −
have *X x* − *Y x* = *0* **for** *x*
using *B*
proof (*rule independentD-alt*)
have {*x*. *X x* − *Y x* ≠ *0*} ⊆ {*x*. *X x* ≠ *0*} ∪ {*x*. *Y x* ≠ *0*}
by *auto*
then show *finite* {*x*. *X x* − *Y x* ≠ *0*} {*x*. *X x* − *Y x* ≠ *0*} ⊆ *B*
using *X Y* **by** (*auto dest*: *finite-subset*)
then have (∑ *x* | *X x* − *Y x* ≠ *0*. (*X x* − *Y x*) ∗*s* *x*) = (∑ *v*∈{*S*. *X S* ≠ *0*} ∪ {*S*. *Y S* ≠ *0*}. (*X v* − *Y v*) ∗*s* *v*)
using *X Y* **by** (*intro sum.mono-neutral-cong-left*) *auto*
also have … = (∑ *v*∈{*S*. *X S* ≠ *0*} ∪ {*S*. *Y S* ≠ *0*}. *X v* ∗*s* *v*) − (∑ *v*∈{*S*. *X S* ≠ *0*} ∪ {*S*. *Y S* ≠ *0*}. *Y v* ∗*s* *v*)
by (*simp add*: *scale-left-diff-distrib sum-subtractf assms*)
also have (∑ *v*∈{*S*. *X S* ≠ *0*} ∪ {*S*. *Y S* ≠ *0*}. *X v* ∗*s* *v*) = (∑ *v*∈{*S*. *X S* ≠ *0*}. *X v* ∗*s* *v*)
using *X Y* **by** (*intro sum.mono-neutral-cong-right*) *auto*
also have (∑ *v*∈{*S*. *X S* ≠ *0*} ∪ {*S*. *Y S* ≠ *0*}. *Y v* ∗*s* *v*) = (∑ *v*∈{*S*. *Y S* ≠ *0*}. *Y v* ∗*s* *v*)
using *X Y* **by** (*intro sum.mono-neutral-cong-right*) *auto*
finally show (∑ *x* | *X x* − *Y x* ≠ *0*. (*X x* − *Y x*) ∗*s* *x*) = *0*
using *assms* **by** *simp*
qed
then show *?thesis*
by *auto*
qed

105 Representation of a vector on a specific basis

definition *representation* :: *'b set* ⇒ *'b* ⇒ *'b* ⇒ *'a*
where *representation basis v* =
(*if independent basis* ∧ *v* ∈ *span basis then*
SOME f. (∀ *v*. *f v* ≠ *0* $\longrightarrow$ *v* ∈ *basis*) ∧ *finite* {*v*. *f v* ≠ *0*} ∧ (∑ *v*∈{*v*. *f v* ≠

```
0}. f v *s v) = v
    else (λb. 0))

lemma unique-representation:
  assumes basis: independent basis
    and in-basis: ⋀v. f v ≠ 0 ⟹ v ∈ basis ⋀v. g v ≠ 0 ⟹ v ∈ basis
    and [simp]: finite {v. f v ≠ 0} finite {v. g v ≠ 0}
    and eq: (∑ v∈{v. f v ≠ 0}. f v *s v) = (∑ v∈{v. g v ≠ 0}. g v *s v)
  shows f = g
proof (rule ext, rule ccontr)
  fix v assume ne: f v ≠ g v
  have dependent basis
    unfolding dependent-explicit
  proof (intro exI conjI)
    have *: {v. f v - g v ≠ 0} ⊆ {v. f v ≠ 0} ∪ {v. g v ≠ 0}
      by auto
    show finite {v. f v - g v ≠ 0}
      by (rule finite-subset[OF *]) simp
    show ∃ v∈{v. f v - g v ≠ 0}. f v - g v ≠ 0
      by (rule bexI[of - v]) (auto simp: ne)
    have (∑ v | f v - g v ≠ 0. (f v - g v) *s v) =
        (∑ v∈{v. f v ≠ 0} ∪ {v. g v ≠ 0}. (f v - g v) *s v)
      by (intro sum.mono-neutral-cong-left *) auto
    also have ... =
        (∑ v∈{v. f v ≠ 0} ∪ {v. g v ≠ 0}. f v *s v) - (∑ v∈{v. f v ≠ 0} ∪ {v. g
v ≠ 0}. g v *s v)
      by (simp add: algebra-simps sum-subtractf)
    also have ... = (∑ v | f v ≠ 0. f v *s v) - (∑ v | g v ≠ 0. g v *s v)
      by (intro arg-cong2[where f= (-)] sum.mono-neutral-cong-right) auto
    finally show (∑ v | f v - g v ≠ 0. (f v - g v) *s v) = 0
      by (simp add: eq)
    show {v. f v - g v ≠ 0} ⊆ basis
      using in-basis * by auto
  qed
  with basis show False by auto
qed

lemma
  shows representation-ne-zero: ⋀b. representation basis v b ≠ 0 ⟹ b ∈ basis
    and finite-representation: finite {b. representation basis v b ≠ 0}
    and sum-nonzero-representation-eq:
      independent basis ⟹ v ∈ span basis ⟹ (∑ b | representation basis v b ≠ 0.
representation basis v b *s b) = v
proof -
  { assume basis: independent basis and v: v ∈ span basis
    define p where p f ⟷
      (∀ v. f v ≠ 0 ⟶ v ∈ basis) ∧ finite {v. f v ≠ 0} ∧ (∑ v∈{v. f v ≠ 0}. f v
*s v) = v for f
    obtain t r where *: finite t t ⊆ basis (∑ b∈t. r b *s b) = v
```

```
    using ‹v ∈ span basis› by (auto simp: span-explicit)
  define f where f b = (if b ∈ t then r b else 0) for b
  have p f
    using ∗ by (auto simp: p-def f-def intro!: sum.mono-neutral-cong-left)
  have ∗: representation basis v = Eps p by (simp add: p-def[abs-def] representation-def basis v)
  from someI[of p f, OF ‹p f›] have p (representation basis v)
    unfolding ∗ . }
 note ∗ = this

 show representation basis v b ≠ 0 ⟹ b ∈ basis for b
  using ∗ by (cases independent basis ∧ v ∈ span basis) (auto simp: representation-def)

 show finite {b. representation basis v b ≠ 0}
  using ∗ by (cases independent basis ∧ v ∈ span basis) (auto simp: representation-def)

 show independent basis ⟹ v ∈ span basis ⟹ (∑ b | representation basis v b ≠ 0. representation basis v b ∗s b) = v
  using ∗ by auto
qed

lemma sum-representation-eq:
 (∑ b∈B. representation basis v b ∗s b) = v
 if independent basis v ∈ span basis finite B basis ⊆ B
proof −
 have (∑ b∈B. representation basis v b ∗s b) =
    (∑ b | representation basis v b ≠ 0. representation basis v b ∗s b)
  apply (rule sum.mono-neutral-cong)
    apply (rule finite-representation)
   apply fact
  subgoal for b
   using that representation-ne-zero[of basis v b]
   by auto
  subgoal by auto
  subgoal by simp
  done
 also have ... = v
  by (rule sum-nonzero-representation-eq; fact)
 finally show ?thesis .
qed

lemma representation-eqI:
 assumes basis: independent basis and b: v ∈ span basis
  and ne-zero: ⋀b. f b ≠ 0 ⟹ b ∈ basis
  and finite: finite {b. f b ≠ 0}
  and eq: (∑ b | f b ≠ 0. f b ∗s b) = v
 shows representation basis v = f
```

by (*rule unique-representation*[*OF basis*])
(*auto simp*: *representation-ne-zero finite-representation*
sum-nonzero-representation-eq[*OF basis b*] *ne-zero finite eq*)

lemma *representation-basis*:
assumes *basis*: *independent basis* **and** *b*: $b \in$ *basis*
shows *representation basis b* = (λv. *if v* = *b then 1 else 0*)
proof (*rule unique-representation*[*OF basis*])
show *representation basis b v* $\neq$ *0* $\Longrightarrow$ $v \in$ *basis* **for** *v*
using *representation-ne-zero* **.**
show *finite* {*v*. *representation basis b v* $\neq$ *0*}
using *finite-representation* **.**
show (*if v* = *b then 1 else 0*) $\neq$ *0* $\Longrightarrow$ $v \in$ *basis* **for** *v*
by (*cases v* = *b*) (*auto simp*: *b*)
have $*$: {*v*. (*if v* = *b then 1 else 0* :: *'a*) $\neq$ *0*} = {*b*}
by *auto*
show *finite* {*v*. (*if v* = *b then 1 else 0*) $\neq$ *0*} **unfolding** $*$ **by** *auto*
show ($\sum$ *v* | *representation basis b v* $\neq$ *0*. *representation basis b v* $*s$ *v*) =
($\sum$ *v* | (*if v* = *b then 1 else 0*::*'a*) $\neq$ *0*. (*if v* = *b then 1 else 0*) $*s$ *v*)
unfolding $*$ *sum-nonzero-representation-eq*[*OF basis span-base*[*OF b*]] **by** *auto*
qed

lemma *representation-zero*: *representation basis 0* = (λb. *0*)
proof *cases*
assume *basis*: *independent basis* **show** *?thesis*
by (*rule representation-eqI*[*OF basis span-zero*]) *auto*
qed (*simp add*: *representation-def*)

lemma *representation-diff*:
assumes *basis*: *independent basis* **and** *v*: $v \in$ *span basis* **and** *u*: $u \in$ *span basis*
shows *representation basis* ($u - v$) = (λb. *representation basis u b* $-$ *representation basis v b*)
proof (*rule representation-eqI*[*OF basis span-diff*[*OF u v*]])
let *?R* = *representation basis*
note *finite-representation*[*simp*] *u*[*simp*] *v*[*simp*]
have $*$: {*b*. *?R u b* $-$ *?R v b* $\neq$ *0*} $\subseteq$ {*b*. *?R u b* $\neq$ *0*} $\cup$ {*b*. *?R v b* $\neq$ *0*}
by *auto*
then show *?R u b* $-$ *?R v b* $\neq$ *0* $\Longrightarrow$ $b \in$ *basis* **for** *b*
by (*auto dest*: *representation-ne-zero*)
show *finite* {*b*. *?R u b* $-$ *?R v b* $\neq$ *0*}
by (*intro finite-subset*[*OF* $*$]) *simp-all*
have ($\sum$ *b* | *?R u b* $-$ *?R v b* $\neq$ *0*. (*?R u b* $-$ *?R v b*) $*s$ *b*) =
($\sum$ *b*$\in${*b*. *?R u b* $\neq$ *0*} $\cup$ {*b*. *?R v b* $\neq$ *0*}. (*?R u b* $-$ *?R v b*) $*s$ *b*)
by (*intro sum.mono-neutral-cong-left* $*$) *auto*
also have ... =
($\sum$ *b*$\in${*b*. *?R u b* $\neq$ *0*} $\cup$ {*b*. *?R v b* $\neq$ *0*}. *?R u b* $*s$ *b*) $-$ ($\sum$ *b*$\in${*b*. *?R u b* $\neq$ *0*} $\cup$ {*b*. *?R v b* $\neq$ *0*}. *?R v b* $*s$ *b*)
by (*simp add*: *algebra-simps sum-subtractf*)
also have ... = ($\sum$ *b* | *?R u b* $\neq$ *0*. *?R u b* $*s$ *b*) $-$ ($\sum$ *b* | *?R v b* $\neq$ *0*. *?R v b*

$*s$ b)
by (*intro arg-cong2*[**where** f= $(-)$] *sum.mono-neutral-cong-right*) *auto*
finally show $(\sum b \mid ?R\ u\ b - ?R\ v\ b \neq 0.\ (?R\ u\ b - ?R\ v\ b) *s\ b) = u - v$
by (*simp add*: *sum-nonzero-representation-eq*[*OF basis*])
qed

lemma *representation-neg*:
independent basis $\Longrightarrow$ $v \in$ *span basis* $\Longrightarrow$ *representation basis* $(-\ v) = (\lambda b.$ $-$ *representation basis v b*)
using *representation-diff*[*of basis v 0*] **by** (*simp add*: *representation-zero span-zero*)

lemma *representation-add*:
independent basis $\Longrightarrow$ $v \in$ *span basis* $\Longrightarrow$ $u \in$ *span basis* $\Longrightarrow$
representation basis $(u + v) = (\lambda b.$ *representation basis u b* $+$ *representation basis v b*)
using *representation-diff*[*of basis* $-v$ u] **by** (*simp add*: *representation-neg representation-diff span-neg*)

lemma *representation-sum*:
independent basis $\Longrightarrow$ $(\bigwedge i.\ i \in I \Longrightarrow v\ i \in$ *span basis*$) \Longrightarrow$
representation basis $(sum\ v\ I) = (\lambda b.\ \sum i{\in}I.$ *representation basis* $(v\ i)\ b$)
by (*induction I rule*: *infinite-finite-induct*)
(*auto simp*: *representation-zero representation-add span-sum*)

lemma *representation-scale*:
assumes *basis*: *independent basis* **and** v: $v \in$ *span basis*
shows *representation basis* $(r *s\ v) = (\lambda b.\ r *$ *representation basis v b*)
proof (*rule representation-eqI*[*OF basis span-scale*[*OF v*]])
let *?R* = *representation basis*
note *finite-representation*[*simp*] *v*[*simp*]
have $*$: $\{b.\ r * ?R\ v\ b \neq 0\} \subseteq \{b.\ ?R\ v\ b \neq 0\}$
by *auto*
then show $r *$ *representation basis v b* $\neq 0 \Longrightarrow b \in$ *basis* **for** b
using *representation-ne-zero* **by** *auto*
show *finite* $\{b.\ r * ?R\ v\ b \neq 0\}$
by (*intro finite-subset*[*OF* $*$]) *simp-all*
have $(\sum b \mid r * ?R\ v\ b \neq 0.\ (r * ?R\ v\ b) *s\ b) = (\sum b{\in}\{b.\ ?R\ v\ b \neq 0\}.\ (r * ?R\ v\ b) *s\ b)$
by (*intro sum.mono-neutral-cong-left* $*$) *auto*
also have $\ldots = r *s\ (\sum b \mid ?R\ v\ b \neq 0.\ ?R\ v\ b *s\ b)$
by (*simp add*: *scale-scale*[*symmetric*] *scale-sum-right del*: *scale-scale*)
finally show $(\sum b \mid r * ?R\ v\ b \neq 0.\ (r * ?R\ v\ b) *s\ b) = r *s\ v$
by (*simp add*: *sum-nonzero-representation-eq*[*OF basis*])
qed

lemma *representation-extend*:
assumes *basis*: *independent basis* **and** v: $v \in$ *span basis'* **and** *basis'*: *basis'* $\subseteq$ *basis*
shows *representation basis v* $=$ *representation basis' v*

proof (*rule representation-eqI*[*OF basis*])
  show *v'*: *v* ∈ *span basis* **using** *span-mono*[*OF basis'*] *v* **by** *auto*
  have ∗: *independent basis'* **using** *basis' basis* **by** (*auto intro*: *dependent-mono*)
  show *representation basis' v b* ≠ *0* ⟹ *b* ∈ *basis* **for** *b*
    using *representation-ne-zero basis'* **by** *auto*
  show *finite* {*b*. *representation basis' v b* ≠ *0*}
    using *finite-representation* **.**
  show (∑ *b* | *representation basis' v b* ≠ *0*. *representation basis' v b* ∗*s b*) = *v*
    using *sum-nonzero-representation-eq*[*OF* ∗ *v*] **.**
qed

The set B is the maximal independent set for *span* B, or A is the minimal spanning set

lemma *spanning-subset-independent*:
  assumes *BA*: *B* ⊆ *A*
    and *iA*: *independent A*
    and *AsB*: *A* ⊆ *span B*
  shows *A* = *B*
proof (*intro antisym*[*OF* - *BA*] *subsetI*)
  have *iB*: *independent B* **using** *independent-mono* [*OF iA BA*] **.**
  fix *v* **assume** *v* ∈ *A*
  with *AsB* **have** *v* ∈ *span B* **by** *auto*
  let *?RB* = *representation B v* **and** *?RA* = *representation A v*
  have *?RB v* = *1*
    unfolding *representation-extend*[*OF iA* ‹*v* ∈ *span B*› *BA*, *symmetric*] *representation-basis*[*OF iA* ‹*v* ∈ *A*›] **by** *simp*
  then show *v* ∈ *B*
    using *representation-ne-zero*[*of B v v*] **by** *auto*
qed

end

A linear function is a mapping between two modules over the same ring.

locale *module-hom* = *m1*: *module s1* + *m2*: *module s2*
    for *s1* :: *'a*::*comm-ring-1* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*a*› *75*)
    and *s2* :: *'a*::*comm-ring-1* ⇒ *'c*::*ab-group-add* ⇒ *'c* (**infixr** ‹∗*b*› *75*) +
  fixes *f* :: *'b* ⇒ *'c*
  assumes *add*: *f* (*b1* + *b2*) = *f b1* + *f b2*
    and *scale*: *f* (*r* ∗*a b*) = *r* ∗*b f b*
begin

lemma *zero*[*simp*]: *f 0* = *0*
  using *scale*[*of 0 0*] **by** *simp*

lemma *neg*: *f* (− *x*) = − *f x*
  using *scale* [**where** *r*=−*1*] **by** (*metis add add-eq-0-iff zero*)

lemma *diff*: *f* (*x* − *y*) = *f x* − *f y*
  by (*metis diff-conv-add-uminus add neg*)

lemma *sum*: $f\ (sum\ g\ S) = (\sum a{\in}S.\ f\ (g\ a))$
proof (*induct S rule*: *infinite-finite-induct*)
case (*insert x F*)
have $f\ (sum\ g\ (insert\ x\ F)) = f\ (g\ x + sum\ g\ F)$
using *insert.hyps* **by** *simp*
also have $\ldots = f\ (g\ x) + f\ (sum\ g\ F)$
using *add* **by** *simp*
also have $\ldots = (\sum a{\in}insert\ x\ F.\ f\ (g\ a))$
using *insert.hyps* **by** *simp*
finally show *?case* **.**
qed *simp-all*

lemma *inj-on-iff-eq-0*:
assumes *s*: *m1.subspace s*
shows *inj-on f s* $\longleftrightarrow$ $(\forall x{\in}s.\ f\ x = 0 \longrightarrow x = 0)$
proof −
have *inj-on f s* $\longleftrightarrow$ $(\forall x{\in}s.\ \forall y{\in}s.\ f\ x - f\ y = 0 \longrightarrow x - y = 0)$
by (*simp add*: *inj-on-def*)
also have $\ldots \longleftrightarrow$ $(\forall x{\in}s.\ \forall y{\in}s.\ f\ (x - y) = 0 \longrightarrow x - y = 0)$
by (*simp add*: *diff*)
also have $\ldots \longleftrightarrow$ $(\forall x{\in}s.\ f\ x = 0 \longrightarrow x = 0)$ (**is** *?l* = *?r*)
proof *safe*
fix *x* **assume** *?l* **assume** $x \in s$ *f x* = *0* **with** ‹*?l*›[*rule-format*, *of x 0*] *s* **show** *x* = *0*
by (*auto simp*: *m1.subspace-0*)
next
fix *x y* **assume** *?r* **assume** $x \in s$ $y \in s$ *f* $(x - y) = 0$
with ‹*?r*›[*rule-format*, *of x* − *y*] *s*
show *x* − *y* = *0*
by (*auto simp*: *m1.subspace-diff*)
qed
finally show *?thesis*
by *auto*
qed

lemma *inj-iff-eq-0*: *inj f* = $(\forall x.\ f\ x = 0 \longrightarrow x = 0)$
by (*rule inj-on-iff-eq-0*[*OF m1.subspace-UNIV*, *unfolded ball-UNIV*])

lemma *subspace-image*: **assumes** *S*: *m1.subspace S* **shows** *m2.subspace* (*f* ‘ *S*)
unfolding *m2.subspace-def*
proof *safe*
show *0* ∈ *f* ‘ *S*
by (*rule image-eqI*[*of* - - *0*]) (*auto simp*: *S m1.subspace-0*)
show $x \in S \Longrightarrow y \in S \Longrightarrow f\ x + f\ y \in f$ ‘ *S* **for** *x y*
by (*rule image-eqI*[*of* - - *x* + *y*]) (*auto simp*: *S m1.subspace-add add*)
show $x \in S \Longrightarrow r *b\ f\ x \in f$ ‘ *S* **for** *r x*
by (*rule image-eqI*[*of* - - *r* ∗*a x*]) (*auto simp*: *S m1.subspace-scale scale*)
qed

lemma *subspace-vimage*: *m2.subspace S* $\Longrightarrow$ *m1.subspace* (*f* $-$‘ *S*)
by (*simp add*: *vimage-def add scale m1.subspace-def m2.subspace-0 m2.subspace-add m2.subspace-scale*)

lemma *subspace-kernel*: *m1.subspace* {*x*. *f x* = *0*}
using *subspace-vimage*[*OF m2.subspace-single-0*] **by** (*simp add*: *vimage-def*)

lemma *span-image*: *m2.span* (*f* ‘ *S*) = *f* ‘ (*m1.span S*)
proof (*rule m2.span-unique*)
show *f* ‘ *S* ⊆ *f* ‘ *m1.span S*
by (*rule image-mono*, *rule m1.span-superset*)
show *m2.subspace* (*f* ‘ *m1.span S*)
using *m1.subspace-span* **by** (*rule subspace-image*)
next
fix *T* **assume** *f* ‘ *S* ⊆ *T* **and** *m2.subspace T* **then show** *f* ‘ *m1.span S* ⊆ *T*
unfolding *image-subset-iff-subset-vimage* **by** (*metis subspace-vimage m1.span-minimal*)
qed

lemma *dependent-inj-imageD*:
assumes *d*: *m2.dependent* (*f* ‘ *s*) **and** *i*: *inj-on f* (*m1.span s*)
shows *m1.dependent s*
proof −
have [*intro*]: *inj-on f s*
using ‹*inj-on f* (*m1.span s*)› *m1.span-superset* **by** (*rule inj-on-subset*)
from *d* **obtain** *s′ r v* **where** ∗: *finite s′ s′* ⊆ *s* (∑ *v*∈*f* ‘ *s′*. *r v* ∗*b v*) = *0 v* ∈ *s′*
r (*f v*) ≠ *0*
by (*auto simp*: *m2.dependent-explicit subset-image-iff dest*!: *finite-imageD intro*: *inj-on-subset*)
have *f* (∑ *v*∈*s′*. *r* (*f v*) ∗*a v*) = (∑ *v*∈*s′*. *r* (*f v*) ∗*b f v*)
by (*simp add*: *sum scale*)
also have ... = (∑ *v*∈*f* ‘ *s′*. *r v* ∗*b v*)
using ‹*s′* ⊆ *s*› **by** (*subst sum.reindex*) (*auto dest*!: *finite-imageD intro*: *inj-on-subset*)
finally have *f* (∑ *v*∈*s′*. *r* (*f v*) ∗*a v*) = *0*
by (*simp add*: ∗)
with ‹*s′* ⊆ *s*› **have** (∑ *v*∈*s′*. *r* (*f v*) ∗*a v*) = *0*
by (*intro inj-onD*[*OF i*] *m1.span-zero m1.span-sum m1.span-scale*) (*auto intro*: *m1.span-base*)
then show *m1.dependent s*
using ‹*finite s′*› ‹*s′* ⊆ *s*› ‹*v* ∈ *s′*› ‹*r* (*f v*) ≠ *0*› **by** (*force simp add*: *m1.dependent-explicit*)
qed

lemma *eq-0-on-span*:
assumes *f0*: ⋀*x*. *x* ∈ *b* $\Longrightarrow$ *f x* = *0* **and** *x*: *x* ∈ *m1.span b* **shows** *f x* = *0*
using *m1.span-induct*[*OF x subspace-kernel*] *f0* **by** *simp*

lemma *independent-injective-image*: *m1.independent s* $\Longrightarrow$ *inj-on f* (*m1.span s*)
$\Longrightarrow$ *m2.independent* (*f* ‘ *s*)
using *dependent-inj-imageD*[*of s*] **by** *auto*

lemma *inj-on-span-independent-image*:
assumes *ifB*: *m2.independent* (*f* ' *B*) **and** *f*: *inj-on f B* **shows** *inj-on f* (*m1.span B*)
unfolding *inj-on-iff-eq-0*[*OF m1.subspace-span*] **unfolding** *m1.span-explicit'*
proof *safe*
fix *r* **assume** *fr*: *finite* {*v*. *r v* ≠ *0*} **and** *r*: ∀ *v*. *r v* ≠ *0* ⟶ *v* ∈ *B*
and *eq0*: *f* (∑ *v* | *r v* ≠ *0*. *r v* ∗*a v*) = *0*
have *0* = (∑ *v* | *r v* ≠ *0*. *r v* ∗*b f v*)
using *eq0* **by** (*simp add*: *sum scale*)
also have ... = (∑ *v*∈*f* ' {*v*. *r v* ≠ *0*}. *r* (*the-inv-into B f v*) ∗*b v*)
using *r* **by** (*subst sum.reindex*) (*auto simp*: *the-inv-into-f-f*[*OF f*] *intro*!: *inj-on-subset*[*OF f*] *sum.cong*)
finally have *r v* ≠ *0* ⟹ *r* (*the-inv-into B f* (*f v*)) = *0* **for** *v*
using *fr r ifB*[*unfolded m2.independent-explicit-module*, *rule-format*, *of f* ' {*v*. *r v* ≠ *0*} λ*v*. *r* (*the-inv-into B f v*)]
by *auto*
then have *r v* = *0* **for** *v*
using *the-inv-into-f-f*[*OF f*] *r* **by** *auto*
then show (∑ *v* | *r v* ≠ *0*. *r v* ∗*a v*) = *0* **by** *auto*
qed

lemma *inj-on-span-iff-independent-image*: *m2.independent* (*f* ' *B*) ⟹ *inj-on f* (*m1.span B*) ⟷ *inj-on f B*
using *inj-on-span-independent-image*[*of B*] *inj-on-subset*[*OF - m1.span-superset*, *of f B*] **by** *auto*

lemma *subspace-linear-preimage*: *m2.subspace S* ⟹ *m1.subspace* {*x*. *f x* ∈ *S*}
by (*simp add*: *add scale m1.subspace-def m2.subspace-def*)

lemma *spans-image*: *V* ⊆ *m1.span B* ⟹ *f* ' *V* ⊆ *m2.span* (*f* ' *B*)
by (*metis image-mono span-image*)

Relation between bases and injectivity/surjectivity of map.

lemma *spanning-surjective-image*:
assumes *us*: *UNIV* ⊆ *m1.span S*
and *sf*: *surj f*
shows *UNIV* ⊆ *m2.span* (*f* ' *S*)
proof −
have *UNIV* ⊆ *f* ' *UNIV*
using *sf* **by** (*auto simp add*: *surj-def*)
also have ... ⊆ *m2.span* (*f* ' *S*)
using *spans-image*[*OF us*] **.**
finally show *?thesis* **.**
qed

lemmas *independent-inj-on-image* = *independent-injective-image*

lemma *independent-inj-image*:

m1.independent S $\Longrightarrow$ *inj f* $\Longrightarrow$ *m2.independent* (*f* ' *S*)
using *independent-inj-on-image*[*of S*] **by** (*auto simp*: *inj-on-subset*)

end

lemma *module-hom-iff*:
module-hom s1 s2 f $\longleftrightarrow$
module s1 $\wedge$ *module s2* $\wedge$
($\forall$ *x y*. *f* (*x* + *y*) = *f x* + *f y*) $\wedge$ ($\forall$ *c x*. *f* (*s1 c x*) = *s2 c* (*f x*))
by (*simp add*: *module-hom-def module-hom-axioms-def*)

locale *module-pair* = *m1*: *module s1* + *m2*: *module s2*
for *s1* :: 'a :: *comm-ring-1* $\Rightarrow$ 'b $\Rightarrow$ 'b :: *ab-group-add*
and *s2* :: 'a :: *comm-ring-1* $\Rightarrow$ 'c $\Rightarrow$ 'c :: *ab-group-add*
begin

lemma *module-hom-zero*: *module-hom s1 s2* (λ*x*. *0*)
by (*simp add*: *module-hom-iff m1.module-axioms m2.module-axioms*)

lemma *module-hom-add*: *module-hom s1 s2 f* $\Longrightarrow$ *module-hom s1 s2 g* $\Longrightarrow$ *module-hom s1 s2* (λ*x*. *f x* + *g x*)
by (*simp add*: *module-hom-iff module.scale-right-distrib*)

lemma *module-hom-sub*: *module-hom s1 s2 f* $\Longrightarrow$ *module-hom s1 s2 g* $\Longrightarrow$ *module-hom s1 s2* (λ*x*. *f x* − *g x*)
by (*simp add*: *module-hom-iff module.scale-right-diff-distrib*)

lemma *module-hom-neg*: *module-hom s1 s2 f* $\Longrightarrow$ *module-hom s1 s2* (λ*x*. − *f x*)
by (*simp add*: *module-hom-iff module.scale-minus-right*)

lemma *module-hom-scale*: *module-hom s1 s2 f* $\Longrightarrow$ *module-hom s1 s2* (λ*x*. *s2 c* (*f x*))
by (*simp add*: *module-hom-iff module.scale-scale module.scale-right-distrib ac-simps*)

lemma *module-hom-compose-scale*:
module-hom s1 s2 (λ*x*. *s2* (*f x*) (*c*))
if *module-hom s1* (∗) *f*
proof −
interpret *mh*: *module-hom s1* (∗) *f* **by** *fact*
show *?thesis*
by *unfold-locales* (*simp-all add*: *mh.add mh.scale m2.scale-left-distrib*)
qed

lemma *bij-module-hom-imp-inv-module-hom*: *module-hom scale1 scale2 f* $\Longrightarrow$ *bij f* $\Longrightarrow$
module-hom scale2 scale1 (*inv f*)
by (*auto simp*: *module-hom-iff bij-is-surj bij-is-inj surj-f-inv-f*
intro!: *Hilbert-Choice.inv-f-eq*)

lemma *module-hom-sum*: $(\bigwedge i.\ i \in I \Longrightarrow$ *module-hom s1 s2* $(f\ i)) \Longrightarrow (I = \{\} \Longrightarrow$ *module s1* $\wedge$ *module s2*$) \Longrightarrow$ *module-hom s1 s2* $(\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
apply (*induction I rule*: *infinite-finite-induct*)
apply (*auto intro*!: *module-hom-zero module-hom-add*)
using *m1.module-axioms m2.module-axioms* **by** *blast*

lemma *module-hom-eq-on-span*: $f\ x = g\ x$
if *module-hom s1 s2 f module-hom s1 s2 g*
and $(\bigwedge x.\ x \in B \Longrightarrow f\ x = g\ x)\ x \in$ *m1.span B*
proof −
interpret *module-hom s1 s2* $\lambda x.\ f\ x - g\ x$
by (*rule module-hom-sub that*)+
from *eq-0-on-span*[*OF - that*(*4*)] *that*(*3*) **show** *?thesis* **by** *auto*
qed

end

context *module* **begin**

lemma *module-hom-scale-self*[*simp*]:
module-hom scale scale $(\lambda x.\ scale\ c\ x)$
using *module-axioms module-hom-iff scale-left-commute scale-right-distrib* **by** *blast*

lemma *module-hom-scale-left*[*simp*]:
module-hom $(*)$ *scale* $(\lambda r.\ scale\ r\ x)$
by *unfold-locales* (*auto simp*: *algebra-simps*)

lemma *module-hom-id*: *module-hom scale scale id*
by (*simp add*: *module-hom-iff module-axioms*)

lemma *module-hom-ident*: *module-hom scale scale* $(\lambda x.\ x)$
by (*simp add*: *module-hom-iff module-axioms*)

lemma *module-hom-uminus*: *module-hom scale scale uminus*
by (*simp add*: *module-hom-iff module-axioms*)

end

lemma *module-hom-compose*: *module-hom s1 s2 f* $\Longrightarrow$ *module-hom s2 s3 g* $\Longrightarrow$ *module-hom s1 s3* (*g o f*)
by (*auto simp*: *module-hom-iff*)

end

106 Vector Spaces

theory *Vector-Spaces*
imports *Modules*

begin

lemma *isomorphism-expand*:
f ∘ *g* = *id* ∧ *g* ∘ *f* = *id* ⟷ (∀ *x*. *f* (*g x*) = *x*) ∧ (∀ *x*. *g* (*f x*) = *x*)
by (*simp add*: *fun-eq-iff o-def id-def*)

lemma *left-right-inverse-eq*:
assumes *fg*: *f* ∘ *g* = *id*
and *gh*: *g* ∘ *h* = *id*
shows *f* = *h*
proof −
have *f* = *f* ∘ (*g* ∘ *h*)
unfolding *gh* **by** *simp*
also have ... = (*f* ∘ *g*) ∘ *h*
by (*simp add*: *o-assoc*)
finally show *f* = *h*
unfolding *fg* **by** *simp*
qed

lemma *ordLeq3-finite-infinite*:
assumes *A*: *finite A* **and** *B*: *infinite B* **shows** *ordLeq3* (*card-of A*) (*card-of B*)
proof −
have ‹*ordLeq3* (*card-of A*) (*card-of B*) ∨ *ordLeq3* (*card-of B*) (*card-of A*)›
by (*intro ordLeq-total card-of-Well-order*)
moreover have ¬ *ordLeq3* (*card-of B*) (*card-of A*)
using *B A card-of-ordLeq-finite*[*of B A*] **by** *auto*
ultimately show *?thesis* **by** *auto*
qed

locale *vector-space* =
fixes *scale* :: *'a*::*field* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*s*› *75*)
assumes *vector-space-assms*:— re-stating the assumptions of *module* instead of extending *module* allows us to rewrite in the sublocale.
a ∗*s* (*x* + *y*) = *a* ∗*s* *x* + *a* ∗*s* *y*
(*a* + *b*) ∗*s* *x* = *a* ∗*s* *x* + *b* ∗*s* *x*
a ∗*s* (*b* ∗*s* *x*) = (*a* ∗ *b*) ∗*s* *x*
1 ∗*s* *x* = *x*

lemma *module-iff-vector-space*: *module s* ⟷ *vector-space s*
unfolding *module-def vector-space-def* **..**

locale *linear* = *vs1*: *vector-space s1* + *vs2*: *vector-space s2* + *module-hom s1 s2 f*
for *s1* :: *'a*::*field* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*a*› *75*)
and *s2* :: *'a*::*field* ⇒ *'c*::*ab-group-add* ⇒ *'c* (**infixr** ‹∗*b*› *75*)
and *f* :: *'b* ⇒ *'c*

lemma *module-hom-iff-linear*: *module-hom s1 s2 f* ⟷ *linear s1 s2 f*
unfolding *module-hom-def linear-def module-iff-vector-space* **by** *auto*
lemmas *module-hom-eq-linear* = *module-hom-iff-linear*[*abs-def*, *THEN meta-eq-to-obj-eq*]

lemmas *linear-iff-module-hom* = *module-hom-iff-linear*[*symmetric*]
lemmas *linear-module-homI* = *module-hom-iff-linear*[*THEN iffD1*]
  and *module-hom-linearI* = *module-hom-iff-linear*[*THEN iffD2*]

context *vector-space* **begin**

sublocale *module scale* **rewrites** *module-hom* = *linear*
  by *unfold-locales* (*fact vector-space-assms module-hom-eq-linear*)+

lemmas— from *module*
    linear-id = *module-hom-id*
  and *linear-ident* = *module-hom-ident*
  and *linear-scale-self* = *module-hom-scale-self*
  and *linear-scale-left* = *module-hom-scale-left*
  and *linear-uminus* = *module-hom-uminus*

lemma *linear-representation*:
  assumes *independent B span B* = *UNIV*
  shows *linear scale* (∗) (λ*v*. *representation B v b*)
proof (*unfold-locales*, *goal-cases*)
  case (*5 x y*)
  show *?case*
    using *assms* **by** (*subst representation-add*) *auto*
next
  case (*6 r x*)
  show *?case*
    using *assms* **by** (*subst representation-scale*) *auto*
qed (*simp-all add*: *algebra-simps*)

lemma *linear-imp-scale*:
  fixes *D*::*'a* ⇒ *'b*
  assumes *linear* (∗) *scale D*
  obtains *d* **where** *D* = (λ*x*. *scale x d*)
proof −
  interpret *linear* (∗) *scale D* **by** *fact*
  show *?thesis*
    by (*metis mult.commute mult.left-neutral scale that*)
qed

lemma *scale-eq-0-iff* [*simp*]: *scale a x* = *0* ⟷ *a* = *0* ∨ *x* = *0*
  by (*metis scale-left-commute right-inverse scale-one scale-scale scale-zero-left*)

lemma *scale-left-imp-eq*:
  assumes *nonzero*: *a* ≠ *0*
    and *scale*: *scale a x* = *scale a y*
  shows *x* = *y*
proof −
  from *scale* **have** *scale a* (*x* − *y*) = *0*
    by (*simp add*: *scale-right-diff-distrib*)

with *nonzero* **have** $x - y = 0$ **by** *simp*
then show $x = y$ **by** (*simp only*: *right-minus-eq*)
qed

lemma *scale-right-imp-eq*:
assumes *nonzero*: $x \neq 0$
and *scale*: *scale a x* = *scale b x*
shows $a = b$
proof −
from *scale* **have** *scale* $(a - b)$ $x = 0$
by (*simp add*: *scale-left-diff-distrib*)
with *nonzero* **have** $a - b = 0$ **by** *simp*
then show $a = b$ **by** (*simp only*: *right-minus-eq*)
qed

lemma *scale-cancel-left* [*simp*]: *scale a x* = *scale a y* $\longleftrightarrow$ $x = y \vee a = 0$
by (*auto intro*: *scale-left-imp-eq*)

lemma *scale-cancel-right* [*simp*]: *scale a x* = *scale b x* $\longleftrightarrow$ $a = b \vee x = 0$
by (*auto intro*: *scale-right-imp-eq*)

lemma *injective-scale*: $c \neq 0 \Longrightarrow$ *inj* $(\lambda x.$ *scale c x*)
by (*simp add*: *inj-on-def*)

lemma *dependent-def*: *dependent P* $\longleftrightarrow$ $(\exists\, a \in P.\ a \in$ *span* $(P - \{a\}))$
unfolding *dependent-explicit*
proof *safe*
fix *a* **assume** *aP*: $a \in P$ **and** $a \in$ *span* $(P - \{a\})$
then obtain *a S u*
where *aP*: $a \in P$ **and** *fS*: *finite S* **and** *SP*: $S \subseteq P$ $a \notin S$ **and** *ua*: $(\sum v{\in}S.\ u\ v *s\ v) = a$
unfolding *span-explicit* **by** *blast*
let *?S* = *insert a S*
let *?u* = $\lambda y.$ *if y* = *a then* − *1 else u y*
from *fS SP* **have** $(\sum v{\in}?S.\ ?u\ v *s\ v) = 0$
by (*simp add*: *if-distrib*[*of* $\lambda r.\ r *s\ a$ **for** *a*] *sum.If-cases field-simps Diff-eq*[*symmetric*] *ua*)
moreover have *finite ?S ?S* $\subseteq P$ $a \in ?S$ *?u a* $\neq 0$
using *fS SP aP* **by** *auto*
ultimately show $\exists\, t\ u.$ *finite t* $\wedge$ $t \subseteq P \wedge (\sum v{\in}t.\ u\ v *s\ v) = 0 \wedge (\exists\, v{\in}t.\ u\ v \neq 0)$ **by** *fast*
next
fix *S u v*
assume *fS*: *finite S* **and** *SP*: $S \subseteq P$ **and** *vS*: $v \in S$
and *uv*: *u v* $\neq 0$ **and** *u*: $(\sum v{\in}S.\ u\ v *s\ v) = 0$
let *?a* = *v*
let *?S* = $S - \{v\}$
let *?u* = $\lambda i.$ $(-\ u\ i)$ / *u v*
have *th0*: *?a* $\in P$ *finite ?S ?S* $\subseteq P$

```
    using fS SP vS by auto
  have (∑ v∈?S. ?u v *s v) = (∑ v∈S. (- (inverse (u ?a))) *s (u v *s v)) - ?u
v *s v
    using fS vS uv by (simp add: sum-diff1 field-simps)
  also have ... = ?a
    unfolding scale-sum-right[symmetric] u using uv by simp
  finally have (∑ v∈?S. ?u v *s v) = ?a .
  with th0 show ∃ a ∈ P. a ∈ span (P - {a})
    unfolding span-explicit by (auto intro!: bexI[where x=?a] exI[where x=?S]
exI[where x=?u])
qed

lemma dependent-single[simp]: dependent {x} ⟷ x = 0
  unfolding dependent-def by auto

lemma in-span-insert:
  assumes a: a ∈ span (insert b S)
    and na: a ∉ span S
  shows b ∈ span (insert a S)
proof -
  from span-breakdown[of b insert b S a, OF insertI1 a]
  obtain k where k: a - k *s b ∈ span (S - {b}) by auto
  have k ≠ 0
  proof
    assume k = 0
    with k span-mono[of S - {b} S] have a ∈ span S by auto
    with na show False by blast
  qed
  then have eq: b = (1/k) *s a - (1/k) *s (a - k *s b)
    by (simp add: algebra-simps)

  from k have (1/k) *s (a - k *s b) ∈ span (S - {b})
    by (rule span-scale)
  also have ... ⊆ span (insert a S)
    by (rule span-mono) auto
  finally show ?thesis
    using k by (subst eq) (blast intro: span-diff span-scale span-base)
qed

lemma dependent-insertD: assumes a: a ∉ span S and S: dependent (insert a S)
shows dependent S
proof -
  have a ∉ S using a by (auto dest: span-base)
  obtain b where b: b = a ∨ b ∈ S b ∈ span (insert a S - {b})
    using S unfolding dependent-def by blast
  have b ≠ a b ∈ S
    using b ‹a ∉ S› a by auto
  with b have *: b ∈ span (insert a (S - {b}))
    by (auto simp: insert-Diff-if)
```

```
  show dependent S
  proof cases
    assume b ∈ span (S − {b}) with ‹b ∈ S› show ?thesis
      by (auto simp add: dependent-def)
  next
    assume b ∉ span (S − {b})
    with ∗ have a ∈ span (insert b (S − {b})) by (rule in-span-insert)
    with a show ?thesis
      using ‹b ∈ S› by (auto simp: insert-absorb)
  qed
qed

lemma independent-insertI: a ∉ span S ⟹ independent S ⟹ independent (insert
a S)
  by (auto dest: dependent-insertD)

lemma independent-insert:
  independent (insert a S) ⟷ (if a ∈ S then independent S else independent S ∧
a ∉ span S)
proof −
  have a ∉ S ⟹ a ∈ span S ⟹ dependent (insert a S)
    by (auto simp: dependent-def)
  then show ?thesis
    by (auto intro: dependent-mono simp: independent-insertI)
qed

lemma maximal-independent-subset-extend:
  assumes S ⊆ V independent S
  obtains B where S ⊆ B B ⊆ V independent B V ⊆ span B
proof −
  let ?C = {B. S ⊆ B ∧ independent B ∧ B ⊆ V}
  have ∃ M∈?C. ∀ X∈?C. M ⊆ X ⟶ X = M
  proof (rule subset-Zorn)
    fix C :: 'b set set assume subset.chain ?C C
    then have C: ⋀c. c ∈ C ⟹ c ⊆ V ⋀c. c ∈ C ⟹ S ⊆ c ⋀c. c ∈ C ⟹
independent c
      ⋀c d. c ∈ C ⟹ d ∈ C ⟹ c ⊆ d ∨ d ⊆ c
      unfolding subset.chain-def by blast+

    show ∃ U∈?C. ∀ X∈C. X ⊆ U
    proof cases
      assume C = {} with assms show ?thesis
        by (auto intro!: exI[of - S])
    next
      assume C ≠ {}
      with C(2) have S ⊆ ⋃ C
        by auto
      moreover have independent (⋃ C)
        by (intro independent-Union-directed C)
```

```
    moreover have ⋃ C ⊆ V
      using C by auto
    ultimately show ?thesis
      by auto
  qed
 qed
 then obtain B where B: independent B B ⊆ V S ⊆ B
  and max: ⋀S. independent S ⟹ S ⊆ V ⟹ B ⊆ S ⟹ S = B
  by auto
 moreover
 { assume ¬ V ⊆ span B
  then obtain v where v ∈ V v ∉ span B
   by auto
  with B have independent (insert v B) by (auto intro: dependent-insertD)
  from max[OF this] ‹v ∈ V› ‹B ⊆ V›
  have v ∈ B
   by auto
  with ‹v ∉ span B› have False
   by (auto intro: span-base) }
 ultimately show ?thesis
  by (meson that)
qed

lemma maximal-independent-subset:
 obtains B where B ⊆ V independent B V ⊆ span B
 by (metis maximal-independent-subset-extend[of {}] empty-subsetI independent-empty)
```

Extends a basis from B to a basis of the entire space.

```
definition extend-basis :: 'b set ⇒ 'b set
  where extend-basis B = (SOME B'. B ⊆ B' ∧ independent B' ∧ span B' =
UNIV)

lemma
 assumes B: independent B
 shows extend-basis-superset: B ⊆ extend-basis B
  and independent-extend-basis: independent (extend-basis B)
  and span-extend-basis[simp]: span (extend-basis B) = UNIV
proof −
 define p where p B' ≡ B ⊆ B' ∧ independent B' ∧ span B' = UNIV for B'
 obtain B' where p B'
  using maximal-independent-subset-extend[OF subset-UNIV B]
  by (metis top.extremum-uniqueI p-def)
 then have p (extend-basis B)
  unfolding extend-basis-def p-def[symmetric] by (rule someI)
 then show B ⊆ extend-basis B independent (extend-basis B) span (extend-basis
B) = UNIV
  by (auto simp: p-def)
qed
```

lemma *in-span-delete*:
  assumes *a*: $a \in span\ S$ **and** *na*: $a \notin span\ (S - \{b\})$
  shows $b \in span\ (insert\ a\ (S - \{b\}))$
  by (*metis Diff-empty Diff-insert0 a in-span-insert insert-Diff na*)

lemma *span-redundant*: $x \in span\ S \Longrightarrow span\ (insert\ x\ S) = span\ S$
  unfolding *span-def* **by** (*rule hull-redundant*)

lemma *span-trans*: $x \in span\ S \Longrightarrow y \in span\ (insert\ x\ S) \Longrightarrow y \in span\ S$
  by (*simp only*: *span-redundant*)

lemma *span-insert-0*[*simp*]: $span\ (insert\ 0\ S) = span\ S$
  by (*metis span-zero span-redundant*)

lemma *span-delete-0* [*simp*]: $span(S - \{0\}) = span\ S$
proof
  show $span\ (S - \{0\}) \subseteq span\ S$
    by (*blast intro*!: *span-mono*)
next
  have $span\ S \subseteq span(insert\ 0\ (S - \{0\}))$
    by (*blast intro*!: *span-mono*)
  also have $... \subseteq span(S - \{0\})$
    using *span-insert-0* **by** *blast*
  finally show $span\ S \subseteq span\ (S - \{0\})$ **.**
qed

lemma *span-image-scale*:
  assumes *finite S* **and** *nz*: $\bigwedge x.\ x \in S \Longrightarrow c\ x \neq 0$
    shows $span\ ((\lambda x.\ c\ x\ *s\ x)\ `\ S) = span\ S$
using *assms*
proof (*induction S arbitrary*: *c*)
  case (*empty c*) **show** *?case* **by** *simp*
next
  case (*insert x F c*)
  show *?case*
  proof (*intro set-eqI iffI*)
    fix *y*
      assume $y \in span\ ((\lambda x.\ c\ x\ *s\ x)\ `\ insert\ x\ F)$
      then show $y \in span\ (insert\ x\ F)$
        using *insert* **by** (*force simp*: *span-breakdown-eq*)
  next
    fix *y*
      assume $y \in span\ (insert\ x\ F)$
      then show $y \in span\ ((\lambda x.\ c\ x\ *s\ x)\ `\ insert\ x\ F)$
        using *insert*
        apply (*clarsimp simp*: *span-breakdown-eq*)
        apply (*rule-tac x*=*k* / *c x* **in** *exI*)
        by *simp*
  qed

qed

lemma *exchange-lemma*:
assumes *f*: *finite T*
and *i*: *independent S*
and *sp*: $S \subseteq$ *span T*
shows $\exists t'.$ *card* $t' =$ *card T* $\wedge$ *finite* $t' \wedge S \subseteq t' \wedge t' \subseteq S \cup T \wedge S \subseteq$ *span* t'
using *f i sp*
proof (*induct card* $(T - S)$ *arbitrary*: *S T rule*: *less-induct*)
case *less*
note *ft* = ‹*finite T*› **and** *S* = ‹*independent S*› **and** *sp* = ‹$S \subseteq$ *span T*›
let *?P* = $\lambda t'.$ *card* $t' =$ *card T* $\wedge$ *finite* $t' \wedge S \subseteq t' \wedge t' \subseteq S \cup T \wedge S \subseteq$ *span* t'
show *?case*
proof (*cases* $S \subseteq T \vee T \subseteq S$)
case *True*
then show *?thesis*
proof
assume $S \subseteq T$ **then show** *?thesis*
by (*metis ft Un-commute sp sup-ge1*)
next
assume $T \subseteq S$ **then show** *?thesis*
by (*metis Un-absorb sp spanning-subset-independent*[*OF - S sp*] *ft*)
qed
next
case *False*
then have *st*: $\neg\ S \subseteq T\ \neg\ T \subseteq S$
by *auto*
from *st*(*2*) **obtain** *b* **where** *b*: $b \in T\ b \notin S$
by *blast*
from *b* **have** $T - \{b\} - S \subset T - S$
by *blast*
then have *cardlt*: *card* $(T - \{b\} - S) <$ *card* $(T - S)$
using *ft* **by** (*auto intro*: *psubset-card-mono*)
from *b ft* **have** *ct0*: *card* $T \neq 0$
by *auto*
show *?thesis*
proof (*cases* $S \subseteq$ *span* $(T - \{b\})$)
case *True*
from *ft* **have** *ftb*: *finite* $(T - \{b\})$
by *auto*
from *less*(*1*)[*OF cardlt ftb S True*]
obtain *U* **where** *U*: *card* $U =$ *card* $(T - \{b\})\ S \subseteq U\ U \subseteq S \cup (T - \{b\})$
$S \subseteq$ *span U*
and *fu*: *finite U* **by** *blast*
let *?w* = *insert b U*
have *th0*: $S \subseteq$ *insert b U*
using *U* **by** *blast*
have *th1*: *insert b U* $\subseteq S \cup T$
using *U b* **by** *blast*

```
    have bu: b ∉ U
      using b U by blast
    from U(1) ft b have card U = (card T − 1)
      by auto
    then have th2: card (insert b U) = card T
      using card-insert-disjoint[OF fu bu] ct0 by auto
    from U(4) have S ⊆ span U .
    also have ... ⊆ span (insert b U)
      by (rule span-mono) blast
    finally have th3: S ⊆ span (insert b U) .
    from th0 th1 th2 th3 fu have th: ?P ?w
      by blast
    from th show ?thesis by blast
  next
    case False
    then obtain a where a: a ∈ S a ∉ span (T − {b})
      by blast
    have ab: a ≠ b
      using a b by blast
    have at: a ∉ T
      using a ab span-base[of a T− {b}] by auto
    have mlt: card ((insert a (T − {b})) − S) < card (T − S)
      using cardlt ft a b by auto
    have ft': finite (insert a (T − {b}))
      using ft by auto
    have sp': S ⊆ span (insert a (T − {b}))
    proof
      fix x
      assume xs: x ∈ S
      have T: T ⊆ insert b (insert a (T − {b}))
        using b by auto
      have bs: b ∈ span (insert a (T − {b}))
        by (rule in-span-delete) (use a sp in auto)
      from xs sp have x ∈ span T
        by blast
      with span-mono[OF T] have x: x ∈ span (insert b (insert a (T − {b}))) ..
      from span-trans[OF bs x] show x ∈ span (insert a (T − {b})) .
    qed
    from less(1)[OF mlt ft' S sp'] obtain U where U:
      card U = card (insert a (T − {b}))
      finite U S ⊆ U U ⊆ S ∪ insert a (T − {b})
      S ⊆ span U by blast
    from U a b ft at ct0 have ?P U
      by auto
    then show ?thesis by blast
  qed
 qed
qed
```

lemma *independent-span-bound*:
assumes *f*: *finite T*
and *i*: *independent S*
and *sp*: $S \subseteq$ *span T*
shows *finite S* $\wedge$ *card S* $\leq$ *card T*
by (*metis exchange-lemma*[*OF f i sp*] *finite-subset card-mono*)

lemma *independent-explicit-finite-subsets*:
independent A $\longleftrightarrow$ $(\forall S \subseteq A.$ *finite S* $\longrightarrow$ $(\forall u.$ $(\sum v{\in}S.$ *u v* $*s$ *v*) = *0* $\longrightarrow$ $(\forall v{\in}S.$ *u v* = *0*)))
unfolding *dependent-explicit* [*of A*] **by** (*simp add*: *disj-not2*)

lemma *independent-if-scalars-zero*:
assumes *fin-A*: *finite A*
and *sum*: $\bigwedge f\ x.$ $(\sum x{\in}A.$ *f x* $*s$ *x*) = *0* $\Longrightarrow$ *x* $\in$ *A* $\Longrightarrow$ *f x* = *0*
shows *independent A*
proof (*unfold independent-explicit-finite-subsets*, *clarify*)
fix *S v* **and** *u* :: *'b* $\Rightarrow$ *'a*
assume *S*: $S \subseteq A$ **and** *v*: $v \in S$
let *?g* = $\lambda x.$ *if x* $\in$ *S then u x else 0*
have $(\sum v{\in}A.$ *?g v* $*s$ *v*) = $(\sum v{\in}S.$ *u v* $*s$ *v*)
using *S fin-A* **by** (*auto intro*!: *sum.mono-neutral-cong-right*)
also assume $(\sum v{\in}S.$ *u v* $*s$ *v*) = *0*
finally have *?g v* = *0* **using** *v S sum* **by** *force*
thus *u v* = *0* **unfolding** *if-P*[*OF v*] **.**
qed

lemma *bij-if-span-eq-span-bases*:
assumes *B*: *independent B* **and** *C*: *independent C*
and *eq*: *span B* = *span C*
shows $\exists f.$ *bij-betw f B C*
proof *cases*
assume *finite B* $\vee$ *finite C*
then have *finite B* $\wedge$ *finite C* $\wedge$ *card C* = *card B*
using *independent-span-bound*[*of B C*] *independent-span-bound*[*of C B*] *assms*
span-superset[*of B*] *span-superset*[*of C*]
by *auto*
then show *?thesis*
by (*auto intro*!: *finite-same-card-bij*)
next
assume $\neg$ (*finite B* $\vee$ *finite C*)
then have *infinite B infinite C* **by** *auto*
{ fix *B C* **assume** *B*: *independent B* **and** *C*: *independent C* **and** *infinite B infinite C* **and** *eq*: *span B* = *span C*
let *?R* = *representation B* **and** *?R'* = *representation C* **let** *?U* = $\lambda c.$ $\{v.$ *?R c v* $\neq$ *0*$\}$
have *in-span-C* [*simp*, *intro*]: ‹*b* $\in$ *B* $\Longrightarrow$ *b* $\in$ *span C*› **for** *b* **unfolding** *eq*[*symmetric*] **by** (*rule span-base*)
have *in-span-B* [*simp*, *intro*]: ‹*c* $\in$ *C* $\Longrightarrow$ *c* $\in$ *span B*› **for** *c* **unfolding** *eq* **by**

(*rule span-base*)
have ‹$B \subseteq (\bigcup c{\in}C.\ ?U\ c)$›
proof
fix b **assume** ‹$b \in B$›
have ‹$b \in span\ C$›
using ‹$b \in B$› **unfolding** *eq*[*symmetric*] **by** (*rule span-base*)
have ‹$(\sum v \mid ?R'\ b\ v \neq 0.\ \sum w \mid ?R\ v\ w \neq 0.\ (?R'\ b\ v * ?R\ v\ w) *s\ w) =$
$(\sum v \mid ?R'\ b\ v \neq 0.\ ?R'\ b\ v *s (\sum w \mid ?R\ v\ w \neq 0.\ ?R\ v\ w *s\ w))$›
by (*simp add*: *scale-sum-right*)
also have ‹$\ldots = (\sum v \mid ?R'\ b\ v \neq 0.\ ?R'\ b\ v *s\ v)$›
by (*auto simp*: *sum-nonzero-representation-eq B eq span-base representation-ne-zero*)
also have ‹$\ldots = b$›
by (*rule sum-nonzero-representation-eq*[*OF C* ‹$b \in span\ C$›])
finally have $?R\ b\ b = ?R\ (\sum v \mid ?R'\ b\ v \neq 0.\ \sum w \mid ?R\ v\ w \neq 0.\ (?R'\ b\ v * ?R\ v\ w) *s\ w)\ b$
by *simp*
also have ... $= (\sum i{\in}\{v.\ ?R'\ b\ v \neq 0\}.\ ?R\ (\sum w \mid ?R\ i\ w \neq 0.\ (?R'\ b\ i * ?R\ i\ w) *s\ w)\ b)$
by (*subst representation-sum*[*OF B*]) (*auto intro*: *span-sum span-scale span-base representation-ne-zero*)
also have ... $= (\sum i{\in}\{v.\ ?R'\ b\ v \neq 0\}.$
$\sum j \in \{w.\ ?R\ i\ w \neq 0\}.\ ?R\ ((?R'\ b\ i * ?R\ i\ j\ ) *s\ \ j\ )\ b)$
by (*subst representation-sum*[*OF B*]) (*auto simp add*: *span-sum span-scale span-base representation-ne-zero*)
also have ‹$\ldots = (\sum v \mid ?R'\ b\ v \neq 0.\ \sum w \mid ?R\ v\ w \neq 0.\ ?R'\ b\ v * ?R\ v\ w * ?R\ w\ b)$›
using B ‹$b \in B$› **by** (*simp add*: *representation-scale*[*OF B*] *span-base representation-ne-zero*)
finally have $(\sum v \mid ?R'\ b\ v \neq 0.\ \sum w \mid ?R\ v\ w \neq 0.\ ?R'\ b\ v * ?R\ v\ w * ?R\ w\ b) \neq 0$
using *representation-basis*[*OF B* ‹$b \in B$›] **by** *auto*
then obtain $v\ w$ **where** *bv*: $?R'\ b\ v \neq 0$ **and** *vw*: $?R\ v\ w \neq 0$ **and** $?R'\ b\ v * ?R\ v\ w * ?R\ w\ b \neq 0$
by (*blast elim*: *sum.not-neutral-contains-not-neutral*)
with *representation-basis*[*OF B, of w*] *vw*[*THEN representation-ne-zero*]
have ‹$?R'\ b\ v \neq 0$› ‹$?R\ v\ b \neq 0$› **by** (*auto split*: *if-splits*)
then show ‹$b \in (\bigcup c{\in}C.\ ?U\ c)$›
by (*auto dest*: *representation-ne-zero*)
qed
then have *B-eq*: ‹$B = (\bigcup c{\in}C.\ ?U\ c)$›
by (*auto intro*: *span-base representation-ne-zero eq*)
have *ordLeq3* (*card-of B*) (*card-of C*)
proof (*subst B-eq*, *rule card-of-UNION-ordLeq-infinite*[*OF* ‹*infinite C*›])
show *ordLeq3* (*card-of C*) (*card-of C*)
by (*intro ordLeq-refl card-of-Card-order*)
show $\forall\ c{\in}C.$ *ordLeq3* (*card-of* $\{v.\ ?R\ c\ v \neq 0\}$) (*card-of C*)
by (*intro ballI ordLeq3-finite-infinite* ‹*infinite C*› *finite-representation*)
qed }

from *this*[*of B C*] *this*[*of C B*] *B C eq* ‹*infinite C*› ‹*infinite B*›
show *?thesis* **by** (*auto simp add*: *ordIso-iff-ordLeq card-of-ordIso*)
qed

definition *dim* :: *'b set* ⇒ *nat*
where *dim V* = (*if* ∃ *b*. *independent b* ∧ *span b* = *span V then*
card (*SOME b*. *independent b* ∧ *span b* = *span V*) *else 0*)

lemma *dim-eq-card*:
assumes *BV*: *span B* = *span V* **and** *B*: *independent B*
shows *dim V* = *card B*
proof −
define *p* **where** *p b* ≡ *independent b* ∧ *span b* = *span V* **for** *b*
have *p* (*SOME B*. *p B*)
using *assms* **by** (*intro someI*[*of p B*]) (*auto simp*: *p-def*)
then have ∃ *f*. *bij-betw f B* (*SOME B*. *p B*)
by (*subst* (*asm*) *p-def*, *intro bij-if-span-eq-span-bases*[*OF B*]) (*simp-all add*: *BV*)
then have *card B* = *card* (*SOME B*. *p B*)
by (*auto intro*: *bij-betw-same-card*)
then show *?thesis*
using *BV B*
by (*auto simp add*: *dim-def p-def*)
qed

lemma *basis-card-eq-dim*: *B* ⊆ *V* ⟹ *V* ⊆ *span B* ⟹ *independent B* ⟹ *card B* = *dim V*
using *dim-eq-card*[*of B V*] *span-mono*[*of B V*] *span-minimal*[*OF - subspace-span*, *of V B*] **by** *auto*

lemma *basis-exists*:
obtains *B* **where** *B* ⊆ *V independent B V* ⊆ *span B card B* = *dim V*
by (*meson basis-card-eq-dim empty-subsetI independent-empty maximal-independent-subset-extend*)

lemma *dim-eq-card-independent*: *independent B* ⟹ *dim B* = *card B*
by (*rule dim-eq-card*[*OF refl*])

lemma *dim-span*[*simp*]: *dim* (*span S*) = *dim S*
by (*auto simp add*: *dim-def span-span*)

lemma *dim-span-eq-card-independent*: *independent B* ⟹ *dim* (*span B*) = *card B*
by (*simp add*: *dim-eq-card*)

lemma *dim-le-card*: **assumes** *V* ⊆ *span W finite W* **shows** *dim V* ≤ *card W*
proof −
obtain *A* **where** *independent A A* ⊆ *V V* ⊆ *span A*
using *maximal-independent-subset*[*of V*] **by** *force*
with *assms independent-span-bound*[*of W A*] *basis-card-eq-dim*[*of A V*]
show *?thesis* **by** *auto*

qed

lemma *span-eq-dim*: *span S = span T ⟹ dim S = dim T*
by (*metis dim-span*)

corollary *dim-le-card'*:
finite s ⟹ dim s ≤ card s
by (*metis basis-exists card-mono*)

lemma *span-card-ge-dim*:
B ⊆ V ⟹ V ⊆ span B ⟹ finite B ⟹ dim V ≤ card B
by (*simp add*: *dim-le-card*)

lemma *dim-unique*:
B ⊆ V ⟹ V ⊆ span B ⟹ independent B ⟹ card B = n ⟹ dim V = n
by (*metis basis-card-eq-dim*)

lemma *subspace-sums*: ⟦*subspace S*; *subspace T*⟧ ⟹ *subspace* {*x + y*|*x y*. *x ∈ S ∧ y ∈ T*}
apply (*simp add*: *subspace-def*)
apply (*intro conjI impI allI*; *clarsimp simp*: *algebra-simps*)
using *add.left-neutral* **apply** *blast*
apply (*metis add.assoc*)
using *scale-right-distrib* **by** *blast*

end

lemma *linear-iff*: *linear s1 s2 f ⟷*
(*vector-space s1 ∧ vector-space s2 ∧* (∀ *x y*. *f* (*x + y*) = *f x + f y*) *∧* (∀ *c x*. *f* (*s1 c x*) = *s2 c* (*f x*)))
unfolding *linear-def module-hom-iff vector-space-def module-def* **by** *auto*

context begin
qualified lemma *linear-compose*: *linear s1 s2 f ⟹ linear s2 s3 g ⟹ linear s1 s3* (*g o f*)
unfolding *module-hom-iff-linear*[*symmetric*]
by (*rule module-hom-compose*)
end

locale *vector-space-pair = vs1*: *vector-space s1 + vs2*: *vector-space s2*
for *s1* :: *'a*::*field* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*a*› *75*)
and *s2* :: *'a*::*field* ⇒ *'c*::*ab-group-add* ⇒ *'c* (**infixr** ‹∗*b*› *75*)
begin

context fixes *f* **assumes** *linear s1 s2 f* **begin**
interpretation *linear s1 s2 f* **by** *fact*
lemmas— from locale *module-hom*
linear-0 = zero
and *linear-add = add*

and *linear-scale = scale*
and *linear-neg = neg*
and *linear-diff = diff*
and *linear-sum = sum*
and *linear-inj-on-iff-eq-0 = inj-on-iff-eq-0*
and *linear-inj-iff-eq-0 = inj-iff-eq-0*
and *linear-subspace-image = subspace-image*
and *linear-subspace-vimage = subspace-vimage*
and *linear-subspace-kernel = subspace-kernel*
and *linear-span-image = span-image*
and *linear-dependent-inj-imageD = dependent-inj-imageD*
and *linear-eq-0-on-span = eq-0-on-span*
and *linear-independent-injective-image = independent-injective-image*
and *linear-inj-on-span-independent-image = inj-on-span-independent-image*
and *linear-inj-on-span-iff-independent-image = inj-on-span-iff-independent-image*
and *linear-subspace-linear-preimage = subspace-linear-preimage*
and *linear-spans-image = spans-image*
and *linear-spanning-surjective-image = spanning-surjective-image*
end

sublocale *module-pair*
rewrites *module-hom = linear*
by *unfold-locales* (*fact module-hom-eq-linear*)

lemmas— from locale *module-pair*
linear-eq-on-span = module-hom-eq-on-span
and *linear-compose-scale-right = module-hom-scale*
and *linear-compose-add = module-hom-add*
and *linear-zero = module-hom-zero*
and *linear-compose-sub = module-hom-sub*
and *linear-compose-neg = module-hom-neg*
and *linear-compose-scale = module-hom-compose-scale*

lemma *linear-indep-image-lemma*:
assumes *lf*: *linear s1 s2 f*
and *fB*: *finite B*
and *ifB*: *vs2.independent* (*f ' B*)
and *fi*: *inj-on f B*
and *xsB*: *x ∈ vs1.span B*
and *fx*: *f x = 0*
shows *x = 0*
using *fB ifB fi xsB fx*
proof (*induction B arbitrary*: *x rule*: *finite-induct*)
case *empty*
then show *?case* **by** *auto*
next
case (*insert a b x*)
have *th0*: *f ' b ⊆ f ' (insert a b)*
by (*simp add*: *subset-insertI*)

```
  have ifb: vs2.independent (f ‘ b)
    using vs2.independent-mono insert.prems(1) th0 by blast
  have fib: inj-on f b
    using insert.prems(2) by blast
  from vs1.span-breakdown[of a insert a b, simplified, OF insert.prems(3)]
  obtain k where k: x − k ∗a a ∈ vs1.span (b − {a})
    by blast
  have f (x − k ∗a a) ∈ vs2.span (f ‘ b)
    unfolding linear-span-image[OF lf]
    using insert.hyps(2) k by auto
  then have f x − k ∗b f a ∈ vs2.span (f ‘ b)
    by (simp add: linear-diff linear-scale lf)
  then have th: −k ∗b f a ∈ vs2.span (f ‘ b)
    using insert.prems(4) by simp
  have xsb: x ∈ vs1.span b
  proof (cases k = 0)
    case True
    with k have x ∈ vs1.span (b − {a}) by simp
    then show ?thesis using vs1.span-mono[of b − {a} b]
      by blast
  next
    case False
    from inj-on-image-set-diff[OF insert.prems(2), of insert a b {a}, symmetric]
    have f ‘ insert a b − f ‘ {a} = f ‘ (insert a b − {a}) by blast
    then have f a ∉ vs2.span (f ‘ b)
      using vs2.dependent-def insert.hyps(2) insert.prems(1) by fastforce
    moreover have f a ∈ vs2.span (f ‘ b)
      using False vs2.span-scale[OF th, of − 1/ k] by auto
    ultimately have False
      by blast
    then show ?thesis by blast
  qed
  show x = 0
    using ifb fib xsb insert.IH insert.prems(4) by blast
qed

lemma linear-eq-on:
  assumes l: linear s1 s2 f linear s1 s2 g
  assumes x: x ∈ vs1.span B and eq: ⋀b. b ∈ B ⟹ f b = g b
  shows f x = g x
proof −
  interpret d: linear s1 s2 λx. f x − g x
    using l by (intro linear-compose-sub) (auto simp: module-hom-iff-linear)
  have f x − g x = 0
    by (rule d.eq-0-on-span[OF - x]) (auto simp: eq)
  then show ?thesis by auto
qed

definition construct :: 'b set ⇒ ('b ⇒ 'c) ⇒ ('b ⇒ 'c)
```

where *construct B g v* = $(\sum b \mid$ *vs1.representation* (*vs1.extend-basis B*) *v b* $\neq$ *0*.
vs1.representation (*vs1.extend-basis B*) *v b* $*b$ (*if b* $\in$ *B then g b else 0*))

lemma *construct-cong*: $(\bigwedge b.\ b \in B \Longrightarrow f\ b = g\ b) \Longrightarrow$ *construct B f* = *construct B g*
unfolding *construct-def* **by** (*auto intro*!: *sum.cong*)

lemma *linear-construct*:
assumes *B*[*simp*]: *vs1.independent B*
shows *linear s1 s2* (*construct B f*)
unfolding *module-hom-iff-linear linear-iff*
proof *safe*
have *eB*[*simp*]: *vs1.independent* (*vs1.extend-basis B*)
using *vs1.independent-extend-basis*[*OF B*] **.**
let *?R* = *vs1.representation* (*vs1.extend-basis B*)
fix *c x y*
have *construct B f* (*x* + *y*) =
($\sum b \in \{b.\ ?R\ x\ b \neq 0\} \cup \{b.\ ?R\ y\ b \neq 0\}.$ *?R* (*x* + *y*) *b* $*b$ (*if b* $\in$ *B then f b else 0*))
by (*auto intro*!: *sum.mono-neutral-cong-left simp*: *vs1.finite-representation vs1.representation-add construct-def*)
also have ... = *construct B f x* + *construct B f y*
by (*auto simp*: *construct-def vs1.representation-add vs2.scale-left-distrib sum.distrib intro*!: *arg-cong2*[**where** *f*=(+)] *sum.mono-neutral-cong-right vs1.finite-representation*)
finally show *construct B f* (*x* + *y*) = *construct B f x* + *construct B f y* **.**

show *construct B f* (*c* $*a$ *x*) = *c* $*b$ *construct B f x*
by (*auto simp del*: *vs2.scale-scale intro*!: *sum.mono-neutral-cong-left vs1.finite-representation simp add*: *construct-def vs2.scale-scale*[*symmetric*] *vs1.representation-scale vs2.scale-sum-right*)
qed *intro-locales*

lemma *construct-basis*:
assumes *B*[*simp*]: *vs1.independent B* **and** *b*: *b* $\in$ *B*
shows *construct B f b* = *f b*
proof −
have $*$: *vs1.representation* (*vs1.extend-basis B*) *b* = ($\lambda v.$ *if v* = *b then 1 else 0*)
using *vs1.extend-basis-superset*[*OF B*] *b*
by (*intro vs1.representation-basis vs1.independent-extend-basis*) *auto*
then have {*v*. *vs1.representation* (*vs1.extend-basis B*) *b v* $\neq$ *0*} = {*b*}
by *auto*
then show *?thesis*
unfolding *construct-def* **by** (*simp add*: $*$ *b*)
qed

lemma *construct-outside*:
assumes *B*: *vs1.independent B* **and** *v*: *v* $\in$ *vs1.span* (*vs1.extend-basis B* − *B*)
shows *construct B f v* = *0*

unfolding *construct-def*
proof (*clarsimp intro*!: *sum.neutral simp del*: *vs2.scale-eq-0-iff*)
fix *b* **assume** $b \in B$
then have *vs1.representation* (*vs1.extend-basis B − B*) *v b* = *0*
using *vs1.representation-ne-zero*[*of vs1.extend-basis B − B v b*] **by** *auto*
moreover have *vs1.representation* (*vs1.extend-basis B*) *v* = *vs1.representation* (*vs1.extend-basis B − B*) *v*
using *vs1.representation-extend*[*OF vs1.independent-extend-basis*[*OF B*] *v*] **by** *auto*
ultimately show *vs1.representation* (*vs1.extend-basis B*) *v b* ∗*b f b* = *0*
by *simp*
qed

lemma *construct-add*:
assumes *B*[*simp*]: *vs1.independent B*
shows *construct B* (λ*x*. *f x* + *g x*) *v* = *construct B f v* + *construct B g v*
proof (*rule linear-eq-on*)
show *v* ∈ *vs1.span* (*vs1.extend-basis B*) **by** *simp*
show *b* ∈ *vs1.extend-basis B* ⟹ *construct B* (λ*x*. *f x* + *g x*) *b* = *construct B f b* + *construct B g b* **for** *b*
using *construct-outside*[*OF B vs1.span-base*, *of b*] **by** (*cases b* ∈ *B*) (*auto simp*: *construct-basis*)
qed (*intro linear-compose-add linear-construct B*)+

lemma *construct-scale*:
assumes *B*[*simp*]: *vs1.independent B*
shows *construct B* (λ*x*. *c* ∗*b f x*) *v* = *c* ∗*b construct B f v*
proof (*rule linear-eq-on*)
show *v* ∈ *vs1.span* (*vs1.extend-basis B*) **by** *simp*
show *b* ∈ *vs1.extend-basis B* ⟹ *construct B* (λ*x*. *c* ∗*b f x*) *b* = *c* ∗*b construct B f b* **for** *b*
using *construct-outside*[*OF B vs1.span-base*, *of b*] **by** (*cases b* ∈ *B*) (*auto simp*: *construct-basis*)
qed (*intro linear-construct module-hom-scale B*)+

lemma *construct-in-span*:
assumes *B*[*simp*]: *vs1.independent B*
shows *construct B f v* ∈ *vs2.span* (*f* ‘ *B*)
proof −
interpret *c*: *linear s1 s2 construct B f* **by** (*rule linear-construct*) *fact*
let *?R* = *vs1.representation B*
have *v* ∈ *vs1.span* ((*vs1.extend-basis B − B*) ∪ *B*)
by (*auto simp*: *Un-absorb2 vs1.extend-basis-superset*)
then obtain *x y* **where** *v* = *x* + *y x* ∈ *vs1.span* (*vs1.extend-basis B − B*) *y* ∈ *vs1.span B*
unfolding *vs1.span-Un* **by** *auto*
moreover have *construct B f* (∑ *b* | *?R y b* ≠ *0*. *?R y b* ∗*a b*) ∈ *vs2.span* (*f* ‘ *B*)
by (*auto simp add*: *c.sum c.scale construct-basis vs1.representation-ne-zero*

```
    intro!: vs2.span-sum vs2.span-scale intro: vs2.span-base )
  ultimately show construct B f v ∈ vs2.span (f ‘ B)
    by (auto simp add: c.add construct-outside vs1.sum-nonzero-representation-eq)
qed

lemma linear-compose-sum:
  assumes lS: ∀ a ∈ S. linear s1 s2 (f a)
  shows linear s1 s2 (λx. sum (λa. f a x) S)
proof (cases finite S)
  case True
  then show ?thesis
    using lS by induct (simp-all add: linear-zero linear-compose-add)
next
  case False
  then show ?thesis
    by (simp add: linear-zero)
qed

lemma in-span-in-range-construct:
  x ∈ range (construct B f) if i: vs1.independent B and x: x ∈ vs2.span (f ‘ B)
proof −
  interpret linear (∗a) (∗b) construct B f
    using i by (rule linear-construct)
  obtain bb :: ('b ⇒ 'c) ⇒ ('b ⇒ 'c) ⇒ 'b set ⇒ 'b where
    ∀ x0 x1 x2. (∃ v4. v4 ∈ x2 ∧ x1 v4 ≠ x0 v4) = (bb x0 x1 x2 ∈ x2 ∧ x1 (bb x0
x1 x2) ≠ x0 (bb x0 x1 x2))
    by moura
  then have f2: ∀ B Ba f fa. (B ≠ Ba ∨ bb fa f Ba ∈ Ba ∧ f (bb fa f Ba) ≠ fa
(bb fa f Ba)) ∨ f ‘ B = fa ‘ Ba
    by (meson image-cong)
  have vs1.span B ⊆ vs1.span (vs1.extend-basis B)
    by (simp add: vs1.extend-basis-superset[OF i] vs1.span-mono)
  then show x ∈ range (construct B f)
    using f2 x by (metis (no-types) construct-basis[OF i, of - f]
        vs1.span-extend-basis[OF i] subsetD span-image spans-image)
qed

lemma range-construct-eq-span:
  range (construct B f) = vs2.span (f ‘ B)
  if vs1.independent B
  by (auto simp: that construct-in-span in-span-in-range-construct)

lemma linear-independent-extend-subspace:
  — legacy: use construct instead
  assumes vs1.independent B
  shows ∃ g. linear s1 s2 g ∧ (∀ x∈B. g x = f x) ∧ range g = vs2.span (f‘B)
  by (rule exI[where x=construct B f])
    (auto simp: linear-construct assms construct-basis range-construct-eq-span)
```

lemma *linear-independent-extend*:
vs1.independent B $\Longrightarrow$ $\exists g.$ *linear s1 s2 g* $\wedge$ $(\forall x{\in}B.\ g\ x = f\ x)$
using *linear-independent-extend-subspace*[*of B f*] **by** *auto*

lemma *linear-exists-left-inverse-on*:
assumes *lf*: *linear s1 s2 f*
assumes *V*: *vs1.subspace V* **and** *f*: *inj-on f V*
shows $\exists g.$ *g ' UNIV* $\subseteq$ *V* $\wedge$ *linear s2 s1 g* $\wedge$ $(\forall v{\in}V.\ g\ (f\ v) = v)$
proof −
interpret *linear s1 s2 f* **by** *fact*
obtain *B* **where** *V-eq*: *V = vs1.span B* **and** *B*: *vs1.independent B*
using *vs1.maximal-independent-subset*[*of V*] *vs1.span-minimal*[*OF - ‹vs1.subspace V›*]
by (*metis antisym-conv*)
have *f*: *inj-on f* (*vs1.span B*)
using *f* **unfolding** *V-eq* **.**
show *?thesis*
proof (*intro exI ballI conjI*)
interpret *p*: *vector-space-pair s2 s1* **by** *unfold-locales*
have *fB*: *vs2.independent* (*f ' B*)
using *independent-injective-image*[*OF B f*] **.**
let *?g = p.construct* (*f ' B*) (*the-inv-into B f*)
show *linear* (∗*b*) (∗*a*) *?g*
by (*rule p.linear-construct*[*OF fB*])
have *?g b* ∈ *vs1.span* (*the-inv-into B f ' f ' B*) **for** *b*
by (*intro p.construct-in-span fB*)
moreover have *the-inv-into B f ' f ' B = B*
by (*auto simp*: *image-comp comp-def the-inv-into-f-f inj-on-subset*[*OF f vs1.span-superset*]
cong: *image-cong*)
ultimately show *?g ' UNIV* $\subseteq$ *V*
by (*auto simp*: *V-eq*)
have (*?g* ∘ *f*) *v = id v* **if** *v* ∈ *vs1.span B* **for** *v*
proof (*rule vector-space-pair.linear-eq-on*[**where** *x=v*])
show *vector-space-pair* (∗*a*) (∗*a*) **by** *unfold-locales*
show *linear* (∗*a*) (∗*a*) (*?g* ∘ *f*)
proof (*rule Vector-Spaces.linear-compose*[*of* - (∗*b*)])
show *linear* (∗*a*) (∗*b*) *f*
by *unfold-locales*
qed *fact*
show *linear* (∗*a*) (∗*a*) *id* **by** (*rule vs1.linear-id*)
show *v* ∈ *vs1.span B* **by** *fact*
show *b* ∈ *B* $\Longrightarrow$ (*p.construct* (*f ' B*) (*the-inv-into B f*) ∘ *f*) *b = id b* **for** *b*
by (*simp add*: *p.construct-basis fB the-inv-into-f-f inj-on-subset*[*OF f vs1.span-superset*])
qed
then show *v* ∈ *V* $\Longrightarrow$ *?g* (*f v*) = *v* **for** *v* **by** (*auto simp*: *comp-def id-def V-eq*)
qed
qed

lemma *linear-exists-right-inverse-on*:
assumes *lf*: *linear s1 s2 f*
assumes *vs1.subspace V*
shows $\exists g.\ g\ `\ UNIV \subseteq V \wedge linear\ s2\ s1\ g \wedge (\forall v{\in}f\ `\ V.\ f\ (g\ v) = v)$
proof −
obtain *B* **where** *V-eq*: $V = vs1.span\ B$ **and** *B*: *vs1.independent B*
using *vs1.maximal-independent-subset*[*of V*] *vs1.span-minimal*[*OF* - ‹*vs1.subspace V*›]
by (*metis antisym-conv*)
obtain *C* **where** *C*: *vs2.independent C* **and** *fB-C*: $f\ `\ B \subseteq vs2.span\ C\ C \subseteq f\ `\ B$
using *vs2.maximal-independent-subset*[*of f ` B*] **by** *metis*
then have $\forall v{\in}C.\ \exists b{\in}B.\ v = f\ b$ **by** *auto*
then obtain *g* **where** *g*: $\bigwedge v.\ v \in C \Longrightarrow g\ v \in B\ \bigwedge v.\ v \in C \Longrightarrow f\ (g\ v) = v$
by *metis*
show *?thesis*
proof (*intro exI ballI conjI*)
interpret *p*: *vector-space-pair s2 s1* **by** *unfold-locales*
let *?g* = *p.construct C g*
show *linear* (∗*b*) (∗*a*) *?g*
by (*rule p.linear-construct*[*OF C*])
have *?g v* ∈ *vs1.span* (*g ` C*) **for** *v*
by (*rule p.construct-in-span*[*OF C*])
also have ... ⊆ *V* **unfolding** *V-eq* **using** *g* **by** (*intro vs1.span-mono*) *auto*
finally show *?g ` UNIV* ⊆ *V* **by** *auto*
have (*f* ∘ *?g*) *v* = *id v* **if** *v*: *v* ∈ *f ` V* **for** *v*
proof (*rule vector-space-pair.linear-eq-on*[**where** *x*=*v*])
show *vector-space-pair* (∗*b*) (∗*b*) **by** *unfold-locales*
show *linear* (∗*b*) (∗*b*) (*f* ∘ *?g*)
by (*rule Vector-Spaces.linear-compose*[*of* - (∗*a*)]) *fact+*
show *linear* (∗*b*) (∗*b*) *id* **by** (*rule vs2.linear-id*)
have *vs2.span* (*f ` B*) = *vs2.span C*
using *fB-C vs2.span-mono*[*of C f ` B*] *vs2.span-minimal*[*of f`B vs2.span C*]
by *auto*
then show *v* ∈ *vs2.span C*
using *v linear-span-image*[*OF lf*, *of B*] **by** (*simp add*: *V-eq*)
show (*f* ∘ *p.construct C g*) *b* = *id b* **if** *b*: *b* ∈ *C* **for** *b*
by (*auto simp*: *p.construct-basis g C b*)
qed
then show *v* ∈ *f ` V* ⟹ *f* (*?g v*) = *v* **for** *v* **by** (*auto simp*: *comp-def id-def*)
qed
qed

lemma *linear-inj-on-left-inverse*:
assumes *lf*: *linear s1 s2 f*
assumes *fi*: *inj-on f* (*vs1.span S*)
shows $\exists g.\ range\ g \subseteq vs1.span\ S \wedge linear\ s2\ s1\ g \wedge (\forall x{\in}vs1.span\ S.\ g\ (f\ x) = x)$

using *linear-exists-left-inverse-on*[*OF lf vs1.subspace-span fi*]
by (*auto simp*: *linear-iff-module-hom*)

lemma *linear-injective-left-inverse*: *linear s1 s2 f* $\Longrightarrow$ *inj f* $\Longrightarrow$ $\exists$ *g. linear s2 s1 g* $\wedge$ *g* $\circ$ *f* = *id*
using *linear-inj-on-left-inverse*[*of f UNIV*]
by *force*

lemma *linear-surj-right-inverse*:
assumes *lf*: *linear s1 s2 f*
assumes *sf*: *vs2.span T* $\subseteq$ *f*'*vs1.span S*
shows $\exists$ *g. range g* $\subseteq$ *vs1.span S* $\wedge$ *linear s2 s1 g* $\wedge$ ($\forall$ *x*$\in$*vs2.span T. f* (*g x*) = *x*)
using *linear-exists-right-inverse-on*[*OF lf vs1.subspace-span, of S*] *sf*
by (*force simp*: *linear-iff-module-hom*)

lemma *linear-surjective-right-inverse*: *linear s1 s2 f* $\Longrightarrow$ *surj f* $\Longrightarrow$ $\exists$ *g. linear s2 s1 g* $\wedge$ *f* $\circ$ *g* = *id*
using *linear-surj-right-inverse*[*of f UNIV UNIV*]
by (*auto simp*: *fun-eq-iff*)

lemma *finite-basis-to-basis-subspace-isomorphism*:
assumes *s*: *vs1.subspace S*
and *t*: *vs2.subspace T*
and *d*: *vs1.dim S* = *vs2.dim T*
and *fB*: *finite B*
and *B*: *B* $\subseteq$ *S vs1.independent B S* $\subseteq$ *vs1.span B card B* = *vs1.dim S*
and *fC*: *finite C*
and *C*: *C* $\subseteq$ *T vs2.independent C T* $\subseteq$ *vs2.span C card C* = *vs2.dim T*
shows $\exists$ *f. linear s1 s2 f* $\wedge$ *f* ' *B* = *C* $\wedge$ *f* ' *S* = *T* $\wedge$ *inj-on f S*
proof −
from *B*(4) *C*(4) *card-le-inj*[*of B C*] *d* **obtain** *f* **where**
f: *f* ' *B* $\subseteq$ *C inj-on f B* **using** ‹*finite B*› ‹*finite C*› **by** *auto*
from *linear-independent-extend*[*OF B*(2)] **obtain** *g* **where**
g: *linear s1 s2 g* $\forall$ *x* $\in$ *B. g x* = *f x* **by** *blast*
interpret *g*: *linear s1 s2 g* **by** *fact*
from *inj-on-iff-eq-card*[*OF fB, of f*] *f*(2)
have *card* (*f* ' *B*) = *card B* **by** *simp*
with *B*(4) *C*(4) **have** *ceq*: *card* (*f* ' *B*) = *card C* **using** *d*
by *simp*
have *g* ' *B* = *f* ' *B* **using** *g*(2)
by (*auto simp add*: *image-iff*)
also have ... = *C* **using** *card-subset-eq*[*OF fC f*(1) *ceq*] **.**
finally have *gBC*: *g* ' *B* = *C* **.**
have *gi*: *inj-on g B* **using** *f*(2) *g*(2)
by (*auto simp add*: *inj-on-def*)
note *g0* = *linear-indep-image-lemma*[*OF g*(1) *fB, unfolded gBC, OF C*(2) *gi*]
{
fix *x y*

```
    assume x: x ∈ S and y: y ∈ S and gxy: g x = g y
    from B(3) x y have x': x ∈ vs1.span B and y': y ∈ vs1.span B
      by blast+
    from gxy have th0: g (x − y) = 0
      by (simp add: g.diff)
    have th1: x − y ∈ vs1.span B using x' y'
      by (metis vs1.span-diff)
    have x = y using g0[OF th1 th0] by simp
  }
  then have giS: inj-on g S unfolding inj-on-def by blast
  from vs1.span-subspace[OF B(1,3) s]
  have g ‘ S = vs2.span (g ‘ B)
    by (simp add: g.span-image)
  also have ... = vs2.span C
    unfolding gBC ..
  also have ... = T
    using vs2.span-subspace[OF C(1,3) t] .
  finally have gS: g ‘ S = T .
  from g(1) gS giS gBC show ?thesis
    by blast
qed

end


locale finite-dimensional-vector-space = vector-space +
  fixes Basis :: 'b set
  assumes finite-Basis: finite Basis
  and independent-Basis: independent Basis
  and span-Basis: span Basis = UNIV
begin

definition dimension = card Basis

lemma finiteI-independent: independent B ⟹ finite B
  using independent-span-bound[OF finite-Basis, of B] by (auto simp: span-Basis)

lemma dim-empty [simp]: dim {} = 0
  by (rule dim-unique[OF order-refl]) (auto simp: dependent-def)

lemma dim-insert:
  dim (insert x S) = (if x ∈ span S then dim S else dim S + 1)
proof −
  show ?thesis
  proof (cases x ∈ span S)
    case True then show ?thesis
      by (metis dim-span span-redundant)
  next
    case False
```

```
    obtain B where B: B ⊆ span S independent B span S ⊆ span B card B = dim
(span S)
      using basis-exists [of span S] by blast
    have dim (span (insert x S)) = Suc (dim S)
    proof (rule dim-unique)
      show insert x B ⊆ span (insert x S)
          by (meson B(1) insertI1 insert-subset order-trans span-base span-mono
subset-insertI)
      show span (insert x S) ⊆ span (insert x B)
       by (metis ‹B ⊆ span S› ‹span S ⊆ span B› span-breakdown-eq span-subspace
subsetI subspace-span)
      show independent (insert x B)
        by (metis B(1−3) independent-insert span-subspace subspace-span False)
      show card (insert x B) = Suc (dim S)
        using B False finiteI-independent by force
    qed
    then show ?thesis
      by (metis False Suc-eq-plus1 dim-span)
  qed
qed

lemma dim-singleton [simp]: dim{x} = (if x = 0 then 0 else 1)
  by (simp add: dim-insert)

proposition choose-subspace-of-subspace:
  assumes n ≤ dim S
  obtains T where subspace T T ⊆ span S dim T = n
proof −
  have ∃ T. subspace T ∧ T ⊆ span S ∧ dim T = n
  using assms
  proof (induction n)
    case 0 then show ?case by (auto intro!: exI[where x={0}] span-zero)
  next
    case (Suc n)
    then obtain T where subspace T T ⊆ span S dim T = n
      by force
    then show ?case
    proof (cases span S ⊆ span T)
      case True
      have span T ⊆ span S
        by (simp add: ‹T ⊆ span S› span-minimal)
      then have dim S = dim T
        by (rule span-eq-dim [OF subset-antisym [OF True]])
      then show ?thesis
        using Suc.prems ‹dim T = n› by linarith
    next
      case False
      then obtain y where y: y ∈ S y ∉ T
        by (meson span-mono subsetI)
```

```
    then have span (insert y T) ⊆ span S
      by (metis (no-types) ‹T ⊆ span S› subsetD insert-subset span-superset
span-mono span-span)
    with ‹dim T = n› ‹subspace T› y show ?thesis
     apply (rule-tac x=span(insert y T) in exI)
     using span-eq-iff by (fastforce simp: dim-insert)
   qed
  qed
  with that show ?thesis by blast
qed

lemma basis-subspace-exists:
  assumes subspace S
  obtains B where finite B B ⊆ S independent B span B = S card B = dim S
  by (metis assms span-subspace basis-exists finiteI-independent)

lemma dim-mono: assumes V ⊆ span W shows dim V ≤ dim W
proof −
  obtain B where independent B B ⊆ W W ⊆ span B
    using maximal-independent-subset[of W] by force
  with dim-le-card[of V B] assms independent-span-bound[of Basis B] basis-card-eq-dim[of
B W]
    span-mono[of B W] span-minimal[OF - subspace-span, of W B]
  show ?thesis
    by (auto simp: finite-Basis span-Basis)
qed

lemma dim-subset: S ⊆ T ⟹ dim S ≤ dim T
  using dim-mono[of S T] by (auto intro: span-base)

lemma dim-eq-0 [simp]:
  dim S = 0 ⟷ S ⊆ {0}
  by (metis basis-exists card-eq-0-iff dim-span finiteI-independent span-empty sub-
set-empty subset-singletonD)

lemma dim-UNIV[simp]: dim UNIV = card Basis
  using dim-eq-card[of Basis UNIV] by (simp add: independent-Basis span-Basis)

lemma independent-card-le-dim: assumes B ⊆ V and independent B shows card
B ≤ dim V
  by (subst dim-eq-card[symmetric, OF refl ‹independent B›]) (rule dim-subset[OF
‹B ⊆ V›])

lemma dim-subset-UNIV: dim S ≤ dimension
  by (metis dim-subset subset-UNIV dim-UNIV dimension-def)

lemma card-ge-dim-independent:
  assumes BV: B ⊆ V
    and iB: independent B
```

```
    and dVB: dim V ≤ card B
  shows V ⊆ span B
proof
  fix a
  assume aV: a ∈ V
  {
    assume aB: a ∉ span B
    then have iaB: independent (insert a B)
      using iB aV BV by (simp add: independent-insert)
    from aV BV have th0: insert a B ⊆ V
      by blast
    from aB have a ∉B
      by (auto simp add: span-base)
    with independent-card-le-dim[OF th0 iaB] dVB finiteI-independent[OF iB]
    have False by auto
  }
  then show a ∈ span B by blast
qed

lemma card-le-dim-spanning:
  assumes BV: B ⊆ V
    and VB: V ⊆ span B
    and fB: finite B
    and dVB: dim V ≥ card B
  shows independent B
proof −
  {
    fix a
    assume a: a ∈ B a ∈ span (B − {a})
    from a fB have c0: card B ≠ 0
      by auto
    from a fB have cb: card (B − {a}) = card B − 1
      by auto
    {
      fix x
      assume x: x ∈ V
      from a have eq: insert a (B − {a}) = B
        by blast
      from x VB have x': x ∈ span B
        by blast
      from span-trans[OF a(2), unfolded eq, OF x']
      have x ∈ span (B − {a}) .
    }
    then have th1: V ⊆ span (B − {a})
      by blast
    have th2: finite (B − {a})
      using fB by auto
    from dim-le-card[OF th1 th2]
    have c: dim V ≤ card (B − {a}) .
```

from *c c0 dVB cb* **have** *False* **by** *simp*
}
then show *?thesis*
unfolding *dependent-def* **by** *blast*
qed

lemma *card-eq-dim*: $B \subseteq V \Longrightarrow$ *card* $B =$ *dim* $V \Longrightarrow$ *finite* $B \Longrightarrow$ *independent* $B \longleftrightarrow V \subseteq$ *span* B
by (*metis order-eq-iff card-le-dim-spanning card-ge-dim-independent*)

lemma *subspace-dim-equal*:
assumes *subspace* S
and *subspace* T
and $S \subseteq T$
and *dim* $S \geq$ *dim* T
shows $S = T$
proof −
obtain B **where** B: $B \leq S$ *independent* $B \wedge S \subseteq$ *span* B *card* $B =$ *dim* S
using *basis-exists*[*of* S] **by** *metis*
then have *span* $B \subseteq S$
using *span-mono*[*of* B S] *span-eq-iff*[*of* S] *assms* **by** *metis*
then have *span* $B = S$
using B **by** *auto*
have *dim* $S =$ *dim* T
using *assms dim-subset*[*of* S T] **by** *auto*
then have $T \subseteq$ *span* B
using *card-eq-dim*[*of* B T] B *finiteI-independent assms* **by** *auto*
then show *?thesis*
using *assms* ‹*span* $B = S$› **by** *auto*
qed

corollary *dim-eq-span*:
shows ⟦$S \subseteq T$; *dim* $T \leq$ *dim* S⟧ $\Longrightarrow$ *span* $S =$ *span* T
by (*simp add*: *span-mono subspace-dim-equal*)

lemma *dim-psubset*:
span $S \subset$ *span* $T \Longrightarrow$ *dim* $S <$ *dim* T
by (*metis* (*no-types*, *opaque-lifting*) *dim-span less-le not-le subspace-dim-equal subspace-span*)

lemma *dim-eq-full*:
shows *dim* $S =$ *dimension* $\longleftrightarrow$ *span* $S =$ *UNIV*
by (*metis dim-eq-span dim-subset-UNIV span-Basis span-span subset-UNIV dim-UNIV dim-span dimension-def*)

lemma *indep-card-eq-dim-span*:
assumes *independent* B
shows *finite* $B \wedge$ *card* $B =$ *dim* (*span* B)
using *dim-span-eq-card-independent*[*OF assms*] *finiteI-independent*[*OF assms*] **by**

auto

More general size bound lemmas.

lemma *independent-bound-general*:
independent S $\Longrightarrow$ *finite S* $\wedge$ *card S* $\leq$ *dim S*
by (*simp add*: *dim-eq-card-independent finiteI-independent*)

lemma *independent-explicit*:
shows *independent B* $\longleftrightarrow$ *finite B* $\wedge$ ($\forall$ *c*. ($\sum$ *v*$\in$*B*. *c v* $*_s$ *v*) = *0* $\longrightarrow$ ($\forall$ *v* $\in$ *B*. *c v* = *0*))
using *independent-bound-general*
by (*fastforce simp*: *dependent-finite*)

proposition *dim-sums-Int*:
assumes *subspace S subspace T*
shows *dim* {*x* + *y* |*x y*. *x* $\in$ *S* $\wedge$ *y* $\in$ *T*} + *dim*(*S* $\cap$ *T*) = *dim S* + *dim T* (**is** *dim ?ST* + - = -)
proof −
obtain *B* **where** *B*: *B* $\subseteq$ *S* $\cap$ *T S* $\cap$ *T* $\subseteq$ *span B*
and *indB*: *independent B*
and *cardB*: *card B* = *dim* (*S* $\cap$ *T*)
using *basis-exists* **by** *metis*
then obtain *C D* **where** *B* $\subseteq$ *C C* $\subseteq$ *S independent C S* $\subseteq$ *span C*
and *B* $\subseteq$ *D D* $\subseteq$ *T independent D T* $\subseteq$ *span D*
using *maximal-independent-subset-extend*
by (*metis Int-subset-iff* ‹*B* $\subseteq$ *S* $\cap$ *T*› *indB*)
then have *finite B finite C finite D*
by (*simp-all add*: *finiteI-independent indB independent-bound-general*)
have *Beq*: *B* = *C* $\cap$ *D*
proof (*rule spanning-subset-independent* [*symmetric*])
show *independent* (*C* $\cap$ *D*)
by (*meson* ‹*independent C*› *independent-mono inf.cobounded1*)
qed (*use B* ‹*C* $\subseteq$ *S*› ‹*D* $\subseteq$ *T*› ‹*B* $\subseteq$ *C*› ‹*B* $\subseteq$ *D*› **in** *auto*)
then have *Deq*: *D* = *B* $\cup$ (*D* − *C*)
by *blast*
have *CUD*: *C* $\cup$ *D* $\subseteq$ *?ST*
proof (*simp*, *intro conjI*)
show *C* $\subseteq$ *?ST*
using *span-zero span-minimal* [*OF* - ‹*subspace T*›] ‹*C* $\subseteq$ *S*› **by** *force*
show *D* $\subseteq$ *?ST*
using *span-zero span-minimal* [*OF* - ‹*subspace S*›] ‹*D* $\subseteq$ *T*› **by** *force*
qed
have *a v* = *0* **if** *0*: ($\sum$ *v*$\in$*C*. *a v* $*_s$ *v*) + ($\sum$ *v*$\in$*D* − *C*. *a v* $*_s$ *v*) = *0*
and *v*: *v* $\in$ *C* $\cup$ (*D*−*C*) **for** *a v*
proof −
have *CsS*: $\bigwedge$*x*. *x* $\in$ *C* $\Longrightarrow$ *a x* $*_s$ *x* $\in$ *S*
using ‹*C* $\subseteq$ *S*› ‹*subspace S*› *subspace-scale* **by** *auto*
have *eq*: ($\sum$ *v*$\in$*D* − *C*. *a v* $*_s$ *v*) = − ($\sum$ *v*$\in$*C*. *a v* $*_s$ *v*)
using *that add-eq-0-iff* **by** *blast*

have $(\sum v{\in}D - C.\ a\ v *s\ v) \in S$
by (*simp add*: *eq CsS* ‹*subspace S*› *subspace-neg subspace-sum*)
moreover have $(\sum v{\in}D - C.\ a\ v *s\ v) \in T$
apply (*rule subspace-sum* [*OF* ‹*subspace T*›])
by (*meson DiffD1* ‹$D \subseteq T$› ‹*subspace T*› *subset-eq subspace-def*)
ultimately have $(\sum v \in D{-}C.\ a\ v *s\ v) \in span\ B$
using *B* **by** *blast*
then obtain *e* **where** *e*: $(\sum v{\in}B.\ e\ v *s\ v) = (\sum v \in D{-}C.\ a\ v *s\ v)$
using *span-finite* [*OF* ‹*finite B*›] **by** *force*
have $\bigwedge c\ v.\ [\![(\sum v{\in}C.\ c\ v *s\ v) = 0;\ v \in C]\!] \Longrightarrow c\ v = 0$
using ‹*finite C*› ‹*independent C*› *independentD* **by** *blast*
define *cc* **where** *cc x* = (*if x* $\in$ *B then a x* + *e x else a x*) **for** *x*
have [*simp*]: $C \cap B = B\ D \cap B = B\ C \cap - B = C{-}D\ B \cap (D - C) = \{\}$
using ‹$B \subseteq C$› ‹$B \subseteq D$› *Beq* **by** *blast+*
have *f2*: $(\sum v{\in}C \cap D.\ e\ v *s\ v) = (\sum v{\in}D - C.\ a\ v *s\ v)$
using *Beq e* **by** *presburger*
have *f3*: $(\sum v{\in}C \cup D.\ a\ v *s\ v) = (\sum v{\in}C - D.\ a\ v *s\ v) + (\sum v{\in}D - C.\ a\ v *s\ v) + (\sum v{\in}C \cap D.\ a\ v *s\ v)$
using ‹*finite C*› ‹*finite D*› *sum.union-diff2* **by** *blast*
have *f4*: $(\sum v{\in}C \cup (D - C).\ a\ v *s\ v) = (\sum v{\in}C.\ a\ v *s\ v) + (\sum v{\in}D - C.\ a\ v *s\ v)$
by (*meson Diff-disjoint* ‹*finite C*› ‹*finite D*› *finite-Diff sum.union-disjoint*)
have $(\sum v{\in}C.\ cc\ v *s\ v) = 0$
using *0 f2 f3 f4*
apply (*simp add*: *cc-def Beq* ‹*finite C*› *sum.If-cases algebra-simps sum.distrib if-distrib if-distribR*)
apply (*simp add*: *add.commute add.left-commute diff-eq*)
done
then have $\bigwedge v.\ v \in C \Longrightarrow cc\ v = 0$
using *independent-explicit* ‹*independent C*› ‹*finite C*› **by** *blast*
then have *C0*: $\bigwedge v.\ v \in C - B \Longrightarrow a\ v = 0$
by (*simp add*: *cc-def Beq*) *meson*
then have [*simp*]: $(\sum x{\in}C - B.\ a\ x *s\ x) = 0$
by *simp*
have $(\sum x{\in}C.\ a\ x *s\ x) = (\sum x{\in}B.\ a\ x *s\ x)$
proof −
have $C - D = C - B$
using *Beq* **by** *blast*
then show *?thesis*
using *Beq* ‹$(\sum x{\in}C - B.\ a\ x *s\ x) = 0$› *f3 f4* **by** *auto*
qed
with *0* **have** *Dcc0*: $(\sum v{\in}D.\ a\ v *s\ v) = 0$
by (*subst Deq*) (*simp add*: ‹*finite B*› ‹*finite D*› *sum-Un*)
then have *D0*: $\bigwedge v.\ v \in D \Longrightarrow a\ v = 0$
using *independent-explicit* ‹*independent D*› ‹*finite D*› **by** *blast*
show *?thesis*
using *v C0 D0 Beq* **by** *blast*
qed
then have *independent* $(C \cup (D - C))$

unfolding *independent-explicit*
using *independent-explicit*
by (*simp add*: *independent-explicit* ‹*finite C*› ‹*finite D*› *sum-Un del*: *Un-Diff-cancel*)
then have *indCUD*: *independent* (*C* ∪ *D*) **by** *simp*
have *dim* (*S* ∩ *T*) = *card B*
by (*rule dim-unique* [*OF B indB refl*])
moreover have *dim S* = *card C*
by (*metis* ‹*C* ⊆ *S*› ‹*independent C*› ‹*S* ⊆ *span C*› *basis-card-eq-dim*)
moreover have *dim T* = *card D*
by (*metis* ‹*D* ⊆ *T*› ‹*independent D*› ‹*T* ⊆ *span D*› *basis-card-eq-dim*)
moreover have *dim ?ST* = *card*(*C* ∪ *D*)
proof −
have ∗: ⋀*x y*. ⟦*x* ∈ *S*; *y* ∈ *T*⟧ ⟹ *x* + *y* ∈ *span* (*C* ∪ *D*)
by (*meson* ‹*S* ⊆ *span C*› ‹*T* ⊆ *span D*› *span-add span-mono subsetCE sup.cobounded1 sup.cobounded2*)
show *?thesis*
by (*auto intro*: ∗ *dim-unique* [*OF CUD - indCUD refl*])
qed
ultimately show *?thesis*
using ‹*B* = *C* ∩ *D*› [*symmetric*]
by (*simp add*: ‹*independent C*› ‹*independent D*› *card-Un-Int finiteI-independent*)
qed

lemma *dependent-biggerset-general*:
(*finite S* ⟹ *card S* > *dim S*) ⟹ *dependent S*
using *independent-bound-general*[*of S*] **by** (*metis linorder-not-le*)

lemma *subset-le-dim*:
S ⊆ *span T* ⟹ *dim S* ≤ *dim T*
by (*metis dim-span dim-subset*)

lemma *linear-inj-imp-surj*:
assumes *lf*: *linear scale scale f*
and *fi*: *inj f*
shows *surj f*
proof −
interpret *lf*: *linear scale scale f* **by** *fact*
from *basis-exists*[*of UNIV*] **obtain** *B*
where *B*: *B* ⊆ *UNIV independent B UNIV* ⊆ *span B card B* = *dim UNIV*
by *blast*
from *B*(4) **have** *d*: *dim UNIV* = *card B*
by *simp*
have *UNIV* ⊆ *span* (*f* ‘ *B*)
proof (*rule card-ge-dim-independent*)
show *independent* (*f* ‘ *B*)
by (*simp add*: *B*(2) *fi lf.independent-inj-image*)
have *card* (*f* ‘ *B*) = *dim UNIV*
by (*metis B*(1) *card-image d fi inj-on-subset*)
then show *dim UNIV* ≤ *card* (*f* ‘ *B*)

```
    by simp
  qed blast
  then show ?thesis
    unfolding lf.span-image surj-def
    using B(3) by blast
qed

end

locale finite-dimensional-vector-space-pair-1 =
  vs1: finite-dimensional-vector-space s1 B1 + vs2: vector-space s2
  for s1 :: 'a::field ⇒ 'b::ab-group-add ⇒ 'b (infixr ‹∗a› 75)
  and B1 :: 'b set
  and s2 :: 'a::field ⇒ 'c::ab-group-add ⇒ 'c (infixr ‹∗b› 75)
begin

sublocale vector-space-pair s1 s2 by unfold-locales

lemma dim-image-eq:
  assumes lf: linear s1 s2 f
    and fi: inj-on f (vs1.span S)
  shows vs2.dim (f ‘ S) = vs1.dim S
proof −
  interpret lf: linear by fact
  obtain B where B: B ⊆ S vs1.independent B S ⊆ vs1.span B card B = vs1.dim
S
    using vs1.basis-exists[of S] by auto
  then have vs1.span S = vs1.span B
    using vs1.span-mono[of B S] vs1.span-mono[of S vs1.span B] vs1.span-span[of
B] by auto
  moreover have card (f ‘ B) = card B
   using assms card-image[of f B] inj-on-subset[of f vs1.span S B] B vs1.span-superset
by auto
  moreover have (f ‘ B) ⊆ (f ‘ S)
    using B by auto
  ultimately show ?thesis
   by (metis B(2) B(4) fi lf.dependent-inj-imageD lf.span-image vs2.dim-eq-card-independent
vs2.dim-span)
qed

lemma dim-image-le:
  assumes lf: linear s1 s2 f
  shows vs2.dim (f ‘ S) ≤ vs1.dim (S)
proof −
  from vs1.basis-exists[of S] obtain B where
    B: B ⊆ S vs1.independent B S ⊆ vs1.span B card B = vs1.dim S by blast
  from B have fB: finite B card B = vs1.dim S
    using vs1.independent-bound-general by blast+
  have vs2.dim (f ‘ S) ≤ card (f ‘ B)
```

apply (*rule vs2.span-card-ge-dim*)
using *lf B fB*
apply (*auto simp add*: *module-hom.span-image module-hom.spans-image subset-image-iff*
linear-iff-module-hom)
done
also have ... ≤ *vs1.dim S*
using *card-image-le*[*OF fB*(*1*)] *fB* **by** *simp*
finally show *?thesis* **.**
qed

end

locale *finite-dimensional-vector-space-pair* =
vs1: *finite-dimensional-vector-space s1 B1* + *vs2*: *finite-dimensional-vector-space s2 B2*
for *s1* :: *'a*::*field* ⇒ *'b*::*ab-group-add* ⇒ *'b* (**infixr** ‹∗*a*› *75*)
and *B1* :: *'b set*
and *s2* :: *'a*::*field* ⇒ *'c*::*ab-group-add* ⇒ *'c* (**infixr** ‹∗*b*› *75*)
and *B2* :: *'c set*
begin

sublocale *finite-dimensional-vector-space-pair-1* **..**

lemma *linear-surjective-imp-injective*:
assumes *lf*: *linear s1 s2 f* **and** *sf*: *surj f* **and** *eq*: *vs2.dim UNIV* = *vs1.dim UNIV*
shows *inj f*
proof −
interpret *linear s1 s2 f* **by** *fact*
have ∗: *card* (*f* ‘ *B1*) ≤ *vs2.dim UNIV*
using *vs1.finite-Basis vs1.dim-eq-card*[*of B1 UNIV*] *sf*
by (*auto simp*: *vs1.span-Basis vs1.independent-Basis eq*
simp del: *vs2.dim-UNIV*
intro!: *card-image-le*)
have *indep-fB*: *vs2.independent* (*f* ‘ *B1*)
using *vs1.finite-Basis vs1.dim-eq-card*[*of B1 UNIV*] *sf* ∗
by (*intro vs2.card-le-dim-spanning*[*of f* ‘ *B1 UNIV*]) (*auto simp*: *span-image vs1.span-Basis*)
have *vs2.dim UNIV* ≤ *card* (*f* ‘ *B1*)
unfolding *eq sf*[*symmetric*] *vs2.dim-span-eq-card-independent*[*symmetric*, *OF indep-fB*]
vs2.dim-span
by (*intro vs2.dim-mono*) (*auto simp*: *span-image vs1.span-Basis*)
with ∗ **have** *card* (*f* ‘ *B1*) = *vs2.dim UNIV* **by** *auto*
also have ... = *card B1*
unfolding *eq vs1.dim-UNIV* **..**
finally have *inj-on f B1*
by (*subst inj-on-iff-eq-card*[*OF vs1.finite-Basis*])

then show *inj f*
using *inj-on-span-iff-independent-image*[*OF indep-fB*] *vs1.span-Basis* **by** *auto*
qed

lemma *linear-injective-imp-surjective*:
assumes *lf*: *linear s1 s2 f* **and** *sf*: *inj f* **and** *eq*: *vs2.dim UNIV* = *vs1.dim UNIV*
shows *surj f*
proof −
interpret *linear s1 s2 f* **by** *fact*
have ∗: *False* **if** *b*: *b* ∉ *vs2.span* (*f* ' *B1*) **for** *b*
proof −
have ∗: *vs2.independent* (*f* ' *B1*)
using *vs1.independent-Basis* **by** (*intro independent-injective-image inj-on-subset*[*OF sf*]) *auto*
have ∗∗: *vs2.independent* (*insert b* (*f* ' *B1*))
using *b* ∗ **by** (*rule vs2.independent-insertI*)

have *b* ∉ *f* ' *B1* **using** *vs2.span-base*[*of b f* ' *B1*] *b* **by** *auto*
then have *Suc* (*card B1*) = *card* (*insert b* (*f* ' *B1*))
using *sf*[*THEN inj-on-subset*, *of B1*] **by** (*subst card.insert-remove*) (*auto intro*: *vs1.finite-Basis simp*: *card-image*)
also have ... = *vs2.dim* (*insert b* (*f* ' *B1*))
using *vs2.dim-eq-card-independent*[*OF* ∗∗] **by** *simp*
also have *vs2.dim* (*insert b* (*f* ' *B1*)) ≤ *vs2.dim B2*
by (*rule vs2.dim-mono*) (*auto simp*: *vs2.span-Basis*)
also have ... = *card B1*
using *vs1.dim-span*[*of B1*] *vs2.dim-span*[*of B2*] **unfolding** *vs1.span-Basis vs2.span-Basis eq*
vs1.dim-eq-card-independent[*OF vs1.independent-Basis*] **by** *simp*
finally show *False* **by** *simp*
qed
have *f* ' *UNIV* = *f* ' *vs1.span B1* **unfolding** *vs1.span-Basis* **..**
also have ... = *vs2.span* (*f* ' *B1*) **unfolding** *span-image* **..**
also have ... = *UNIV* **using** ∗ **by** *blast*
finally show *?thesis* **.**
qed

lemma *linear-injective-isomorphism*:
assumes *lf*: *linear s1 s2 f*
and *fi*: *inj f*
and *dims*: *vs2.dim UNIV* = *vs1.dim UNIV*
shows ∃*f'*. *linear s2 s1 f'* ∧ (∀ *x*. *f'* (*f x*) = *x*) ∧ (∀ *x*. *f* (*f'* *x*) = *x*)
unfolding *isomorphism-expand*[*symmetric*]
using *linear-injective-imp-surjective*[*OF lf fi dims*]
using *fi left-right-inverse-eq lf linear-injective-left-inverse linear-surjective-right-inverse* **by** *blast*

lemma *linear-surjective-isomorphism*:
assumes *lf*: *linear s1 s2 f*

```
    and sf: surj f
    and dims: vs2.dim UNIV = vs1.dim UNIV
  shows ∃f'. linear s2 s1 f' ∧ (∀x. f' (f x) = x) ∧ (∀x. f (f' x) = x)
  using linear-surjective-imp-injective[OF lf sf dims] sf
    linear-exists-right-inverse-on[OF lf vs1.subspace-UNIV]
    linear-exists-left-inverse-on[OF lf vs1.subspace-UNIV]
    dims lf linear-injective-isomorphism by auto

lemma basis-to-basis-subspace-isomorphism:
  assumes s: vs1.subspace S
    and t: vs2.subspace T
    and d: vs1.dim S = vs2.dim T
    and B: B ⊆ S vs1.independent B S ⊆ vs1.span B card B = vs1.dim S
    and C: C ⊆ T vs2.independent C T ⊆ vs2.span C card C = vs2.dim T
  shows ∃f. linear s1 s2 f ∧ f ‘ B = C ∧ f ‘ S = T ∧ inj-on f S
proof −
  from B have fB: finite B
    by (simp add: vs1.finiteI-independent)
  from C have fC: finite C
    by (simp add: vs2.finiteI-independent)
  from finite-basis-to-basis-subspace-isomorphism[OF s t d fB B fC C] show ?thesis
.
qed

lemma basis-change-exists':
  assumes vs1.independent B vs2.independent B'
  assumes vs1.span B = UNIV vs2.span B' = UNIV vs1.dimension = vs2.dimension
  shows   ∃g. linear s1 s2 g ∧ bij g ∧ bij-betw g B B'
proof −
  have finite B finite B'
    using assms by (auto intro: vs1.finiteI-independent vs2.finiteI-independent)
  moreover from assms have card B = card B'
   by (metis vs1.dim-span vs2.dim-span vs1.dim-UNIV vs2.dim-UNIV vs1.dimension-def
vs2.dimension-def
            vs1.dim-eq-card-independent vs2.dim-eq-card-independent)
  ultimately obtain h where h: bij-betw h B B'
    using ‹card B = card B'› by (meson bij-betw-iff-card)

  define g where g = construct B h
  interpret g: Vector-Spaces.linear s1 s2 g
    unfolding g-def by (rule linear-construct) fact

  have bij-betw g B B' ⟷ bij-betw h B B'
    using assms(1) by (intro bij-betw-cong) (auto simp: g-def construct-basis)
  hence bij-betw g B B'
    using h by simp
  moreover from this have bij g
    using assms(3−5)
    by (metis bij-betw-imp-surj-on bij-def g.linear-axioms local.linear-span-image
```

local.linear-surjective-imp-injective vs1.dim-UNIV vs1.dimension-def vs2.dim-UNIV vs2.dimension-def)
ultimately show *?thesis*
using *g.linear-axioms* **by** *blast*
qed

lemma *basis-change-exists*:
assumes *vs1.dimension = vs2.dimension*
shows $\exists g.$ *linear s1 s2 g* $\wedge$ *bij g* $\wedge$ *bij-betw g B1 B2*
using *basis-change-exists'*[*OF vs1.independent-Basis vs2.independent-Basis vs1.span-Basis vs2.span-Basis assms*] **.**

end

context *finite-dimensional-vector-space* **begin**

lemma *linear-surj-imp-inj*:
assumes *lf*: *linear scale scale f* **and** *sf*: *surj f*
shows *inj f*
proof −
interpret *finite-dimensional-vector-space-pair scale Basis scale Basis* **by** *unfold-locales*
let *?U = UNIV* :: *'b set*
from *basis-exists*[*of ?U*] **obtain** *B*
where *B*: *B* ⊆ *?U independent B ?U* ⊆ *span B* **and** *d*: *card B = dim ?U*
by *blast*
{
fix *x*
assume *x*: *x* ∈ *span B* **and** *fx*: *f x = 0*
from *B*(*2*) **have** *fB*: *finite B*
using *finiteI-independent* **by** *auto*
have *Uspan*: *UNIV* ⊆ *span* (*f* ' *B*)
by (*simp add*: *B*(*3*) *lf linear-spanning-surjective-image sf*)
have *fBi*: *independent* (*f* ' *B*)
proof (*rule card-le-dim-spanning*)
show *card* (*f* ' *B*) ≤ *dim ?U*
using *card-image-le d fB* **by** *fastforce*
qed (*use fB Uspan* **in** *auto*)
have *th0*: *dim ?U* ≤ *card* (*f* ' *B*)
by (*rule span-card-ge-dim*) (*use Uspan fB* **in** *auto*)
moreover have *card* (*f* ' *B*) ≤ *card B*
by (*rule card-image-le*, *rule fB*)
ultimately have *th1*: *card B = card* (*f* ' *B*)
unfolding *d* **by** *arith*
have *fiB*: *inj-on f B*
by (*simp add*: *eq-card-imp-inj-on fB th1*)
from *linear-indep-image-lemma*[*OF lf fB fBi fiB x*] *fx*
have *x = 0* **by** *blast*
}

then show *?thesis*
unfolding *linear-inj-iff-eq-0*[*OF lf*] **using** *B*(*3*) **by** *blast*
qed

lemma *linear-inverse-left*:
assumes *lf*: *linear scale scale f*
and *lf'*: *linear scale scale f'*
shows $f \circ f' = id \longleftrightarrow f' \circ f = id$
proof −
{
fix *f f'*:: *'b* ⇒ *'b*
assume *lf*: *linear scale scale f linear scale scale f'*
assume *f*: $f \circ f' = id$
from *f* **have** *sf*: *surj f*
by (*auto simp add*: *o-def id-def surj-def*) *metis*
interpret *finite-dimensional-vector-space-pair scale Basis scale Basis* **by** *unfold-locales*
from *linear-surjective-isomorphism*[*OF lf*(*1*) *sf*] *lf f*
have $f' \circ f = id$
unfolding *fun-eq-iff o-def id-def* **by** *metis*
}
then show *?thesis*
using *lf lf'* **by** *metis*
qed

lemma *left-inverse-linear*:
assumes *lf*: *linear scale scale f*
and *gf*: $g \circ f = id$
shows *linear scale scale g*
proof −
from *gf* **have** *fi*: *inj f*
by (*auto simp add*: *inj-on-def o-def id-def fun-eq-iff*) *metis*
interpret *finite-dimensional-vector-space-pair scale Basis scale Basis* **by** *unfold-locales*
from *linear-injective-isomorphism*[*OF lf fi*]
obtain *h* :: *'b* ⇒ *'b* **where** *linear scale scale h* **and** *h*: $\forall x.\ h\ (f\ x) = x\ \forall x.\ f\ (h\ x) = x$
by *blast*
have $h = g$
by (*metis gf h isomorphism-expand left-right-inverse-eq*)
with ‹*linear scale scale h*› **show** *?thesis* **by** *blast*
qed

lemma *inj-linear-imp-inv-linear*:
assumes *linear scale scale f inj f* **shows** *linear scale scale* (*inv f*)
using *assms inj-iff left-inverse-linear* **by** *blast*

lemma *right-inverse-linear*:
assumes *lf*: *linear scale scale f*

```
    and gf: f ∘ g = id
  shows linear scale scale g
proof −
  from gf have fi: surj f
    by (auto simp add: surj-def o-def id-def) metis
   interpret finite-dimensional-vector-space-pair scale Basis scale Basis by un-
fold-locales
  from linear-surjective-isomorphism[OF lf fi]
  obtain h:: 'b ⇒ 'b where h: linear scale scale h ∀ x. h (f x) = x ∀ x. f (h x) = x
    by blast
  then have h = g
    by (metis gf isomorphism-expand left-right-inverse-eq)
  with h(1) show ?thesis by blast
qed

lemma linear-independent-extend-inj:
  assumes independent B independent (f ‘ B) inj-on f B
  shows ∃ g. linear scale scale g ∧ inj g ∧ (∀ x∈B. g x = f x)
proof −
  obtain B' where B': span B' = UNIV B ⊆ B' independent B'
   using assms(1) by (meson extend-basis-superset independent-extend-basis span-extend-basis)
  obtain B'' where B'': span B'' = UNIV f ‘ B ⊆ B'' independent B''
   using assms(2) by (meson extend-basis-superset independent-extend-basis span-extend-basis)
  have card B' = card B''
    using B' B'' by (metis local.dim-eq-card)
  hence card (B' − B) = card (B'' − f ‘ B)
    using finiteI-independent[OF assms(1)] B' B''
    by (subst (1 2) card-Diff-subset) (auto simp: card-image[OF ‹inj-on f B›])
  hence ∃ h. bij-betw h (B' − B) (B'' − f ‘ B)
    using finiteI-independent[of B'] finiteI-independent[of B''] B' B''
    by (intro finite-same-card-bij) auto
  then obtain h where h: bij-betw h (B' − B) (B'' − f ‘ B)
    by blast

  have [simp, intro]: finite B'
    using finiteI-independent[of B'] B' by auto
  define r where r = representation B'
  define g where g = (λv. ∑ b∈B'. scale (r v b) (if b ∈ B then f b else h b))
  interpret pair: vector-space-pair scale scale ..
  interpret g: Vector-Spaces.linear scale scale g
    unfolding g-def r-def
    by (intro pair.module-hom-sum pair.linear-compose-scale linear-representation
B' conjI module-axioms)

  have g-B: g b = f b if b ∈ B for b
  proof −
    from that have g b = (∑ b'∈{b}. f b)
      unfolding g-def using B'
     by (intro sum.mono-neutral-cong-right) (auto simp: r-def representation-basis)
```

```
    thus g b = f b
      by simp
  qed

  have g-not-B: g b = h b if b ∈ B' − B for b
  proof −
    from that have g b = (∑ b'∈{b}. h b)
      unfolding g-def using B'
     by (intro sum.mono-neutral-cong-right) (auto simp: r-def representation-basis)
    thus g b = h b
      by simp
  qed

  show ?thesis
  proof (rule exI[of - g]; safe)
    have B'' ⊆ range g
    proof
      fix b assume b: b ∈ B''
      thus b ∈ range g
      proof (cases b ∈ f ‘ B)
        case True
        then obtain b' where b': b' ∈ B f b' = b
          by blast
        thus ?thesis
          by (auto simp: image-def g-B intro!: exI[of - b'])
      next
        case False
        with b have b ∈ B'' − f ‘ B
          by blast
        then obtain b' where h b' = b b' ∈ B' − B
          using h by (metis bij-betw-iff-bijections)
        thus ?thesis
          by (intro rev-image-eqI[of b']) (simp-all add: g-not-B)
      qed
    qed
    hence span B'' ≤ span (range g)
      by (intro span-mono)
    also have span B'' = UNIV
      using B'' by simp
    finally show inj g
     by (simp add: g.linear-axioms g.span-image local.linear-surj-imp-inj set-eq-subset)
  qed (use g-B g.linear-axioms in auto)
qed

end

context finite-dimensional-vector-space-pair begin

lemma subspace-isomorphism:
```

assumes *s*: *vs1.subspace S*
and *t*: *vs2.subspace T*
and *d*: *vs1.dim S = vs2.dim T*
shows $\exists f.$ *linear s1 s2 f* $\wedge$ *f ' S = T* $\wedge$ *inj-on f S*
proof −
from *vs1.basis-exists*[*of S*] *vs1.finiteI-independent*
obtain *B* **where** *B*: *B* $\subseteq$ *S vs1.independent B S* $\subseteq$ *vs1.span B card B = vs1.dim S* **and** *fB*: *finite B*
by *metis*
from *vs2.basis-exists*[*of T*] *vs2.finiteI-independent*
obtain *C* **where** *C*: *C* $\subseteq$ *T vs2.independent C T* $\subseteq$ *vs2.span C card C = vs2.dim T* **and** *fC*: *finite C*
by *metis*
from *B*(*4*) *C*(*4*) *card-le-inj*[*of B C*] *d*
obtain *f* **where** *f*: *f ' B* $\subseteq$ *C inj-on f B* **using** ‹*finite B*› ‹*finite C*›
by *auto*
from *linear-independent-extend*[*OF B*(*2*)]
obtain *g* **where** *g*: *linear s1 s2 g* $\forall$ *x*$\in$ *B. g x = f x*
by *blast*
interpret *g*: *linear s1 s2 g* **by** *fact*
from *inj-on-iff-eq-card*[*OF fB, of f*] *f*(*2*) **have** *card* (*f ' B*) = *card B*
by *simp*
with *B*(*4*) *C*(*4*) **have** *ceq*: *card* (*f ' B*) = *card C*
using *d* **by** *simp*
have *g ' B = f ' B*
using *g*(*2*) **by** (*auto simp add*: *image-iff*)
also have ... = *C* **using** *card-subset-eq*[*OF fC f*(*1*) *ceq*] **.**
finally have *gBC*: *g ' B = C* **.**
have *gi*: *inj-on g B*
using *f*(*2*) *g*(*2*) **by** (*auto simp add*: *inj-on-def*)
note *g0 = linear-indep-image-lemma*[*OF g*(*1*) *fB, unfolded gBC, OF C*(*2*) *gi*]
{
fix *x y*
assume *x*: *x* $\in$ *S* **and** *y*: *y* $\in$ *S* **and** *gxy*: *g x = g y*
from *B*(*3*) *x y* **have** *x'*: *x* $\in$ *vs1.span B* **and** *y'*: *y* $\in$ *vs1.span B*
by *blast*+
from *gxy* **have** *th0*: *g* (*x* − *y*) = *0*
by (*simp add*: *linear-diff g*)
have *th1*: *x* − *y* $\in$ *vs1.span B*
using *x' y'* **by** (*metis vs1.span-diff*)
have *x = y*
using *g0*[*OF th1 th0*] **by** *simp*
}
then have *giS*: *inj-on g S*
unfolding *inj-on-def* **by** *blast*
from *vs1.span-subspace*[*OF B*(*1,3*) *s*] **have** *g ' S = vs2.span* (*g ' B*)
by (*simp add*: *module-hom.span-image*[*OF g*(*1*)[*unfolded linear-iff-module-hom*]])
also have ... = *vs2.span C* **unfolding** *gBC* **..**
also have ... = *T* **using** *vs2.span-subspace*[*OF C*(*1,3*) *t*] **.**

finally have *gS*: g ' $S = T$ **.**
from *g*(*1*) *gS giS* **show** *?thesis*
by *blast*
qed

end

hide-const (**open**) *linear*

end

107 Vector Spaces and Algebras over the Reals

theory *Real-Vector-Spaces*
imports *Real Topological-Spaces Vector-Spaces*
begin

107.1 Real vector spaces

class *scaleR* =
fixes *scaleR* :: *real* ⇒ *'a* ⇒ *'a* (**infixr** ‹$*_R$› *75*)
begin

abbreviation *divideR* :: *'a* ⇒ *real* ⇒ *'a* (**infixl** ‹$'/_R$› *70*)
where $x /_R r \equiv inverse\ r *_R x$

end

class *real-vector* = *scaleR* + *ab-group-add* +
assumes *scaleR-add-right*: $a *_R (x + y) = a *_R x + a *_R y$
and *scaleR-add-left*: $(a + b) *_R x = a *_R x + b *_R x$
and *scaleR-scaleR*: $a *_R b *_R x = (a * b) *_R x$
and *scaleR-one*: $1 *_R x = x$

class *real-algebra* = *real-vector* + *ring* +
assumes *mult-scaleR-left* [*simp*]: $a *_R x * y = a *_R (x * y)$
and *mult-scaleR-right* [*simp*]: $x * a *_R y = a *_R (x * y)$

class *real-algebra-1* = *real-algebra* + *ring-1*

class *real-div-algebra* = *real-algebra-1* + *division-ring*

class *real-field* = *real-div-algebra* + *field*

instantiation *real* :: *real-field*
begin

definition *real-scaleR-def* [*simp*]: *scaleR a x* = *a* * *x*

instance
by *standard* (*simp-all add*: *algebra-simps*)

end

locale *linear* = *Vector-Spaces.linear scaleR*::-⇒-⇒'*a*::*real-vector scaleR*::-⇒-⇒'*b*::*real-vector*
begin

lemmas *scaleR* = *scale*

end

global-interpretation *real-vector?*: *vector-space scaleR* :: *real* ⇒ '*a* ⇒ '*a* :: *real-vector*
rewrites *Vector-Spaces.linear* (∗$_R$) (∗$_R$) = *linear*
and *Vector-Spaces.linear* (∗) (∗$_R$) = *linear*
defines *dependent-raw-def*: *dependent* = *real-vector.dependent*
and *representation-raw-def*: *representation* = *real-vector.representation*
and *subspace-raw-def*: *subspace* = *real-vector.subspace*
and *span-raw-def*: *span* = *real-vector.span*
and *extend-basis-raw-def*: *extend-basis* = *real-vector.extend-basis*
and *dim-raw-def*: *dim* = *real-vector.dim*
proof *unfold-locales*
show *Vector-Spaces.linear* (∗$_R$) (∗$_R$) = *linear Vector-Spaces.linear* (∗) (∗$_R$) = *linear*
by (*force simp*: *linear-def real-scaleR-def*[*abs-def*])+
qed (*use scaleR-add-right scaleR-add-left scaleR-scaleR scaleR-one* **in** *auto*)

hide-const (**open**)— locale constants
real-vector.dependent
real-vector.independent
real-vector.representation
real-vector.subspace
real-vector.span
real-vector.extend-basis
real-vector.dim

abbreviation *independent x* ≡ ¬ *dependent x*

global-interpretation *real-vector?*: *vector-space-pair scaleR*::-⇒-⇒'*a*::*real-vector scaleR*::-⇒-⇒'*b*::*real-vector*
rewrites *Vector-Spaces.linear* (∗$_R$) (∗$_R$) = *linear*
and *Vector-Spaces.linear* (∗) (∗$_R$) = *linear*
defines *construct-raw-def*: *construct* = *real-vector.construct*
proof *unfold-locales*
show *Vector-Spaces.linear* (∗) (∗$_R$) = *linear*
unfolding *linear-def real-scaleR-def* **by** *auto*
qed (*auto simp*: *linear-def*)

hide-const (**open**)— locale constants

real-vector.construct

lemma *linear-compose*: *linear f* $\Longrightarrow$ *linear g* $\Longrightarrow$ *linear* $(g \circ f)$
unfolding *linear-def* **by** (*rule Vector-Spaces.linear-compose*)

Recover original theorem names

lemmas *scaleR-left-commute* = *real-vector.scale-left-commute*
lemmas *scaleR-zero-left* = *real-vector.scale-zero-left*
lemmas *scaleR-minus-left* = *real-vector.scale-minus-left*
lemmas *scaleR-diff-left* = *real-vector.scale-left-diff-distrib*
lemmas *scaleR-sum-left* = *real-vector.scale-sum-left*
lemmas *scaleR-zero-right* = *real-vector.scale-zero-right*
lemmas *scaleR-minus-right* = *real-vector.scale-minus-right*
lemmas *scaleR-diff-right* = *real-vector.scale-right-diff-distrib*
lemmas *scaleR-sum-right* = *real-vector.scale-sum-right*
lemmas *scaleR-eq-0-iff* = *real-vector.scale-eq-0-iff*
lemmas *scaleR-left-imp-eq* = *real-vector.scale-left-imp-eq*
lemmas *scaleR-right-imp-eq* = *real-vector.scale-right-imp-eq*
lemmas *scaleR-cancel-left* = *real-vector.scale-cancel-left*
lemmas *scaleR-cancel-right* = *real-vector.scale-cancel-right*

lemma [*field-simps*]:
$c \neq 0 \Longrightarrow a = b \mathbin{/_R} c \longleftrightarrow c *_R a = b$
$c \neq 0 \Longrightarrow b \mathbin{/_R} c = a \longleftrightarrow b = c *_R a$
$c \neq 0 \Longrightarrow a + b \mathbin{/_R} c = (c *_R a + b) \mathbin{/_R} c$
$c \neq 0 \Longrightarrow a \mathbin{/_R} c + b = (a + c *_R b) \mathbin{/_R} c$
$c \neq 0 \Longrightarrow a - b \mathbin{/_R} c = (c *_R a - b) \mathbin{/_R} c$
$c \neq 0 \Longrightarrow a \mathbin{/_R} c - b = (a - c *_R b) \mathbin{/_R} c$
$c \neq 0 \Longrightarrow - (a \mathbin{/_R} c) + b = (- a + c *_R b) \mathbin{/_R} c$
$c \neq 0 \Longrightarrow - (a \mathbin{/_R} c) - b = (- a - c *_R b) \mathbin{/_R} c$
for $a\ b :: {'a} ::$ *real-vector*
by (*auto simp add*: *scaleR-add-right scaleR-add-left scaleR-diff-right scaleR-diff-left*)

Legacy names

lemmas *scaleR-left-distrib* = *scaleR-add-left*
lemmas *scaleR-right-distrib* = *scaleR-add-right*
lemmas *scaleR-left-diff-distrib* = *scaleR-diff-left*
lemmas *scaleR-right-diff-distrib* = *scaleR-diff-right*

lemmas *linear-injective-0* = *linear-inj-iff-eq-0*
and *linear-injective-on-subspace-0* = *linear-inj-on-iff-eq-0*
and *linear-cmul* = *linear-scale*
and *linear-scaleR* = *linear-scale-self*
and *subspace-mul* = *subspace-scale*
and *span-linear-image* = *linear-span-image*
and *span-0* = *span-zero*
and *span-mul* = *span-scale*
and *injective-scaleR* = *injective-scale*

lemma *scaleR-minus1-left* [*simp*]: *scaleR* (−*1*) *x* = − *x*
for *x* :: *'a*::*real-vector*
by *simp*

lemma *scaleR-2*:
fixes *x* :: *'a*::*real-vector*
shows *scaleR 2 x* = *x* + *x*
unfolding *one-add-one* [*symmetric*] *scaleR-left-distrib* **by** *simp*

lemma *scaleR-half-double* [*simp*]:
fixes *a* :: *'a*::*real-vector*
shows (*1* / *2*) *∗R* (*a* + *a*) = *a*
proof −
have ⋀*r*. *r ∗R* (*a* + *a*) = (*r* ∗ *2*) *∗R a*
by (*metis scaleR-2 scaleR-scaleR*)
then show *?thesis*
by *simp*
qed

lemma *shift-zero-ident* [*simp*]:
fixes *f* :: *'a* ⇒ *'b*::*real-vector*
shows (+)*0* ∘ *f* = *f*
by *force*

lemma *linear-scale-real*:
fixes *r*::*real* **shows** *linear f* ⟹ *f* (*r* ∗ *b*) = *r* ∗ *f b*
using *linear-scale* **by** *fastforce*

interpretation *scaleR-left*: *additive* (λ*a*. *scaleR a x* :: *'a*::*real-vector*)
by *standard* (*rule scaleR-left-distrib*)

interpretation *scaleR-right*: *additive* (λ*x*. *scaleR a x* :: *'a*::*real-vector*)
by *standard* (*rule scaleR-right-distrib*)

lemma *nonzero-inverse-scaleR-distrib*:
a ≠ *0* ⟹ *x* ≠ *0* ⟹ *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
for *x* :: *'a*::*real-div-algebra*
by (*rule inverse-unique*) *simp*

lemma *inverse-scaleR-distrib*: *inverse* (*scaleR a x*) = *scaleR* (*inverse a*) (*inverse x*)
for *x* :: *'a*::{*real-div-algebra*,*division-ring*}
by (*metis inverse-zero nonzero-inverse-scaleR-distrib scale-eq-0-iff*)

lemmas *sum-constant-scaleR* = *real-vector.sum-constant-scale*— legacy name

named-theorems *vector-add-divide-simps to simplify sums of scaled vectors*

lemma [*vector-add-divide-simps*]:

$v + (b \,/\, z) *_R w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_R v + b *_R w) /_R z)$
$a *_R v + (b \,/\, z) *_R w = (\textit{if } z = 0 \textit{ then } a *_R v \textit{ else } ((a * z) *_R v + b *_R w) /_R z)$
$(a \,/\, z) *_R v + w = (\textit{if } z = 0 \textit{ then } w \textit{ else } (a *_R v + z *_R w) /_R z)$
$(a \,/\, z) *_R v + b *_R w = (\textit{if } z = 0 \textit{ then } b *_R w \textit{ else } (a *_R v + (b * z) *_R w) /_R z)$
$v - (b \,/\, z) *_R w = (\textit{if } z = 0 \textit{ then } v \textit{ else } (z *_R v - b *_R w) /_R z)$
$a *_R v - (b \,/\, z) *_R w = (\textit{if } z = 0 \textit{ then } a *_R v \textit{ else } ((a * z) *_R v - b *_R w) /_R z)$
$(a \,/\, z) *_R v - w = (\textit{if } z = 0 \textit{ then } -w \textit{ else } (a *_R v - z *_R w) /_R z)$
$(a \,/\, z) *_R v - b *_R w = (\textit{if } z = 0 \textit{ then } -b *_R w \textit{ else } (a *_R v - (b * z) *_R w) /_R z)$
for $v :: 'a :: \textit{real-vector}$
by (*simp-all add*: *divide-inverse-commute scaleR-add-right scaleR-diff-right*)

lemma *eq-vector-fraction-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \textit{real-vector}$
shows $(x = (u \,/\, v) *_R a) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } v *_R x = u *_R a)$
by *auto* (*metis* (*no-types*) *divide-eq-1-iff divide-inverse-commute scaleR-one scaleR-scaleR*)

lemma *vector-fraction-eq-iff* [*vector-add-divide-simps*]:
fixes $x :: 'a :: \textit{real-vector}$
shows $((u \,/\, v) *_R a = x) \longleftrightarrow (\textit{if } v{=}0 \textit{ then } x = 0 \textit{ else } u *_R a = v *_R x)$
by (*metis eq-vector-fraction-iff*)

lemma *real-vector-affinity-eq*:
fixes $x :: 'a :: \textit{real-vector}$
assumes *m0*: $m \neq 0$
shows $m *_R x + c = y \longleftrightarrow x = \textit{inverse } m *_R y - (\textit{inverse } m *_R c)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have $m *_R x = y - c$ **by** (*simp add*: *field-simps*)
then have $\textit{inverse } m *_R (m *_R x) = \textit{inverse } m *_R (y - c)$ **by** *simp*
then show $x = \textit{inverse } m *_R y - (\textit{inverse } m *_R c)$
using *m0*
by (*simp add*: *scaleR-diff-right*)
next
assume *?rhs*
with *m0* **show** $m *_R x + c = y$
by (*simp add*: *scaleR-diff-right*)
qed

lemma *real-vector-eq-affinity*: $m \neq 0 \Longrightarrow y = m *_R x + c \longleftrightarrow \textit{inverse } m *_R y - (\textit{inverse } m *_R c) = x$
for $x :: 'a{::}\textit{real-vector}$
using *real-vector-affinity-eq*[**where** $m{=}m$ **and** $x{=}x$ **and** $y{=}y$ **and** $c{=}c$]
by *metis*

lemma *scaleR-eq-iff* [*simp*]: $b + u *_R a = a + u *_R b \longleftrightarrow a = b \vee u = 1$
for $a :: 'a$::*real-vector*
proof (*cases* $u = 1$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
have $a = b$ **if** $b + u *_R a = a + u *_R b$
proof −
from *that* **have** $(u - 1) *_R a = (u - 1) *_R b$
by (*simp add*: *algebra-simps*)
with *False* **show** *?thesis*
by *auto*
qed
then show *?thesis* **by** *auto*
qed

lemma *scaleR-collapse* [*simp*]: $(1 - u) *_R a + u *_R a = a$
for $a :: 'a$::*real-vector*
by (*simp add*: *algebra-simps*)

107.2 Embedding of the Reals into any *real-algebra-1*: *of-real*

definition *of-real* :: *real* ⇒ $'a$::*real-algebra-1*
where *of-real r* = *scaleR r 1*

lemma *scaleR-conv-of-real*: *scaleR r x* = *of-real r* ∗ *x*
by (*simp add*: *of-real-def*)

lemma *of-real-0* [*simp*]: *of-real 0* = *0*
by (*simp add*: *of-real-def*)

lemma *of-real-1* [*simp*]: *of-real 1* = *1*
by (*simp add*: *of-real-def*)

lemma *of-real-add* [*simp*]: *of-real* $(x + y)$ = *of-real x* + *of-real y*
by (*simp add*: *of-real-def scaleR-left-distrib*)

lemma *of-real-minus* [*simp*]: *of-real* $(- x)$ = − *of-real x*
by (*simp add*: *of-real-def*)

lemma *of-real-diff* [*simp*]: *of-real* $(x - y)$ = *of-real x* − *of-real y*
by (*simp add*: *of-real-def scaleR-left-diff-distrib*)

lemma *of-real-mult* [*simp*]: *of-real* $(x * y)$ = *of-real x* ∗ *of-real y*
by (*simp add*: *of-real-def*)

lemma *of-real-sum*[*simp*]: *of-real* (*sum f s*) = $(\sum x \in s.$ *of-real* $(f\ x))$

by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *of-real-prod*[*simp*]: *of-real* (*prod f s*) = ($\prod$ *x*∈*s*. *of-real* (*f x*))
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *sum-list-of-real*: *sum-list* (*map of-real xs*) = *of-real* (*sum-list xs*)
by (*induction xs*) *auto*

lemma *nonzero-of-real-inverse*:
x ≠ *0* ⟹ *of-real* (*inverse x*) = *inverse* (*of-real x* :: *'a*::*real-div-algebra*)
by (*simp add*: *of-real-def nonzero-inverse-scaleR-distrib*)

lemma *of-real-inverse* [*simp*]:
of-real (*inverse x*) = *inverse* (*of-real x* :: *'a*::{*real-div-algebra*,*division-ring*})
by (*simp add*: *of-real-def inverse-scaleR-distrib*)

lemma *nonzero-of-real-divide*:
y ≠ *0* ⟹ *of-real* (*x* / *y*) = (*of-real x* / *of-real y* :: *'a*::*real-field*)
by (*simp add*: *divide-inverse nonzero-of-real-inverse*)

lemma *of-real-divide* [*simp*]:
of-real (*x* / *y*) = (*of-real x* / *of-real y* :: *'a*::*real-div-algebra*)
by (*simp add*: *divide-inverse*)

lemma *of-real-power* [*simp*]:
of-real (*x* ^ *n*) = (*of-real x* :: *'a*::{*real-algebra-1*}) ^ *n*
by (*induct n*) *simp-all*

lemma *of-real-power-int* [*simp*]:
of-real (*power-int x n*) = *power-int* (*of-real x* :: *'a* :: {*real-div-algebra*,*division-ring*})
n
by (*auto simp*: *power-int-def*)

lemma *of-real-eq-iff* [*simp*]: *of-real x* = *of-real y* ⟷ *x* = *y*
by (*simp add*: *of-real-def*)

lemma *inj-of-real*: *inj of-real*
by (*auto intro*: *injI*)

lemmas *of-real-eq-0-iff* [*simp*] = *of-real-eq-iff* [*of - 0*, *simplified*]
lemmas *of-real-eq-1-iff* [*simp*] = *of-real-eq-iff* [*of - 1*, *simplified*]

lemma *minus-of-real-eq-of-real-iff* [*simp*]: −*of-real x* = *of-real y* ⟷ −*x* = *y*
using *of-real-eq-iff*[*of* −*x y*] **by** (*simp only*: *of-real-minus*)

lemma *of-real-eq-minus-of-real-iff* [*simp*]: *of-real x* = −*of-real y* ⟷ *x* = −*y*
using *of-real-eq-iff*[*of x* −*y*] **by** (*simp only*: *of-real-minus*)

lemma *of-real-eq-id* [*simp*]: *of-real* = (*id* :: *real* ⇒ *real*)

by (*rule ext*) (*simp add*: *of-real-def*)

Collapse nested embeddings.

lemma *of-real-of-nat-eq* [*simp*]: *of-real* (*of-nat n*) = *of-nat n*
by (*induct n*) *auto*

lemma *of-real-of-int-eq* [*simp*]: *of-real* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *of-real-numeral* [*simp*]: *of-real* (*numeral w*) = *numeral w*
using *of-real-of-int-eq* [*of numeral w*] **by** *simp*

lemma *of-real-neg-numeral* [*simp*]: *of-real* (− *numeral w*) = − *numeral w*
using *of-real-of-int-eq* [*of* − *numeral w*] **by** *simp*

lemma *numeral-power-int-eq-of-real-cancel-iff* [*simp*]:
power-int (*numeral x*) *n* = (*of-real y* :: *'a* :: {*real-div-algebra*, *division-ring*}) ⟷
power-int (*numeral x*) *n* = *y*
proof −
have *power-int* (*numeral x*) *n* = (*of-real* (*power-int* (*numeral x*) *n*) :: *'a*)
by *simp*
also have ... = *of-real y* ⟷ *power-int* (*numeral x*) *n* = *y*
by (*subst of-real-eq-iff*) *auto*
finally show *?thesis* **.**
qed

lemma *of-real-eq-numeral-power-int-cancel-iff* [*simp*]:
(*of-real y* :: *'a* :: {*real-div-algebra*, *division-ring*}) = *power-int* (*numeral x*) *n* ⟷
y = *power-int* (*numeral x*) *n*
by (*subst* (*1 2*) *eq-commute*) *simp*

lemma *of-real-eq-of-real-power-int-cancel-iff* [*simp*]:
power-int (*of-real b* :: *'a* :: {*real-div-algebra*, *division-ring*}) *w* = *of-real x* ⟷
power-int b w = *x*
by (*metis of-real-power-int of-real-eq-iff*)

lemma *of-real-in-Ints-iff* [*simp*]: *of-real x* ∈ ℤ ⟷ *x* ∈ ℤ
proof *safe*
fix *x* **assume** (*of-real x* :: *'a*) ∈ ℤ
then obtain *n* **where** (*of-real x* :: *'a*) = *of-int n*
by (*auto simp*: *Ints-def*)
also have *of-int n* = *of-real* (*real-of-int n*)
by *simp*
finally have *x* = *real-of-int n*
by (*subst* (*asm*) *of-real-eq-iff*)
thus *x* ∈ ℤ
by *auto*
qed (*auto simp*: *Ints-def*)

lemma *Ints-of-real* [*intro*]: $x \in \mathbb{Z} \Longrightarrow$ *of-real* $x \in \mathbb{Z}$
by *simp*

Every real algebra has characteristic zero.

instance *real-algebra-1* $<$ *ring-char-0*
proof
from *inj-of-real inj-of-nat* **have** *inj* (*of-real* $\circ$ *of-nat*)
by (*rule inj-compose*)
then show *inj* (*of-nat* :: *nat* $\Rightarrow$ *'a*)
by (*simp add*: *comp-def*)
qed

lemma *fraction-scaleR-times* [*simp*]:
fixes *a* :: *'a*::*real-algebra-1*
shows (*numeral u* / *numeral v*) $*_R$ (*numeral w* $*$ *a*) = (*numeral u* $*$ *numeral w* / *numeral v*) $*_R$ *a*
by (*metis* (*no-types*, *lifting*) *of-real-numeral scaleR-conv-of-real scaleR-scaleR times-divide-eq-left*)

lemma *inverse-scaleR-times* [*simp*]:
fixes *a* :: *'a*::*real-algebra-1*
shows (*1* / *numeral v*) $*_R$ (*numeral w* $*$ *a*) = (*numeral w* / *numeral v*) $*_R$ *a*
by (*metis divide-inverse-commute inverse-eq-divide of-real-numeral scaleR-conv-of-real scaleR-scaleR*)

lemma *scaleR-times* [*simp*]:
fixes *a* :: *'a*::*real-algebra-1*
shows (*numeral u*) $*_R$ (*numeral w* $*$ *a*) = (*numeral u* $*$ *numeral w*) $*_R$ *a*
by (*simp add*: *scaleR-conv-of-real*)

instance *real-field* $<$ *field-char-0* **..**

107.3 The Set of Real Numbers

definition *Reals* :: *'a*::*real-algebra-1 set* (‹$\mathbb{R}$›)
where $\mathbb{R}$ = *range of-real*

lemma *Reals-of-real* [*simp*]: *of-real r* $\in \mathbb{R}$
by (*simp add*: *Reals-def*)

lemma *Reals-of-int* [*simp*]: *of-int z* $\in \mathbb{R}$
by (*subst of-real-of-int-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-of-nat* [*simp*]: *of-nat n* $\in \mathbb{R}$
by (*subst of-real-of-nat-eq* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-numeral* [*simp*]: *numeral w* $\in \mathbb{R}$
by (*subst of-real-numeral* [*symmetric*], *rule Reals-of-real*)

lemma *Reals-0* [*simp*]: *0* $\in \mathbb{R}$ **and** *Reals-1* [*simp*]: *1* $\in \mathbb{R}$

by (*simp-all add*: *Reals-def*)

lemma *Reals-add* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a + b \in \mathbb{R}$
by (*metis* (*no-types*, *opaque-lifting*) *Reals-def Reals-of-real imageE of-real-add*)

lemma *Reals-minus* [*simp*]: $a \in \mathbb{R} \Longrightarrow - a \in \mathbb{R}$
by (*auto simp*: *Reals-def*)

lemma *Reals-minus-iff* [*simp*]: $- a \in \mathbb{R} \longleftrightarrow a \in \mathbb{R}$
using *Reals-minus* **by** *fastforce*

lemma *Reals-diff* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a - b \in \mathbb{R}$
by (*metis Reals-add Reals-minus-iff add-uminus-conv-diff*)

lemma *Reals-mult* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a * b \in \mathbb{R}$
by (*metis* (*no-types*, *lifting*) *Reals-def Reals-of-real imageE of-real-mult*)

lemma *nonzero-Reals-inverse*: $a \in \mathbb{R} \Longrightarrow a \neq 0 \Longrightarrow$ *inverse* $a \in \mathbb{R}$
for a :: *'a*::*real-div-algebra*
by (*metis Reals-def Reals-of-real imageE of-real-inverse*)

lemma *Reals-inverse*: $a \in \mathbb{R} \Longrightarrow$ *inverse* $a \in \mathbb{R}$
for a :: *'a*::{*real-div-algebra*,*division-ring*}
using *nonzero-Reals-inverse* **by** *fastforce*

lemma *Reals-inverse-iff* [*simp*]: *inverse* $x \in \mathbb{R} \longleftrightarrow x \in \mathbb{R}$
for x :: *'a*::{*real-div-algebra*,*division-ring*}
by (*metis Reals-inverse inverse-inverse-eq*)

lemma *nonzero-Reals-divide*: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow b \neq 0 \Longrightarrow a \mathbin{/} b \in \mathbb{R}$
for a b :: *'a*::*real-field*
by (*simp add*: *divide-inverse*)

lemma *Reals-divide* [*simp*]: $a \in \mathbb{R} \Longrightarrow b \in \mathbb{R} \Longrightarrow a \mathbin{/} b \in \mathbb{R}$
for a b :: *'a*::{*real-field*,*field*}
using *nonzero-Reals-divide* **by** *fastforce*

lemma *Reals-power* [*simp*]: $a \in \mathbb{R} \Longrightarrow a \mathbin{\hat{}} n \in \mathbb{R}$
for a :: *'a*::*real-algebra-1*
by (*metis Reals-def Reals-of-real imageE of-real-power*)

lemma *Reals-cases* [*cases set*: *Reals*]:
assumes $q \in \mathbb{R}$
obtains (*of-real*) r **where** $q =$ *of-real* r
unfolding *Reals-def*
proof −
from ‹$q \in \mathbb{R}$› **have** $q \in$ *range of-real* **unfolding** *Reals-def* **.**
then obtain r **where** $q =$ *of-real* r **..**
then show *thesis* **..**

qed

lemma *sum-in-Reals* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{R}) \Longrightarrow sum\ f\ s \in \mathbb{R}$
proof (*induct s rule*: *infinite-finite-induct*)
  case *infinite*
  then show *?case* **by** (*metis Reals-0 sum.infinite*)
qed *simp-all*

lemma *prod-in-Reals* [*intro*,*simp*]: $(\bigwedge i.\ i \in s \Longrightarrow f\ i \in \mathbb{R}) \Longrightarrow prod\ f\ s \in \mathbb{R}$
proof (*induct s rule*: *infinite-finite-induct*)
  case *infinite*
  then show *?case* **by** (*metis Reals-1 prod.infinite*)
qed *simp-all*

lemma *Reals-induct* [*case-names of-real*, *induct set*: *Reals*]:
  $q \in \mathbb{R} \Longrightarrow (\bigwedge r.\ P\ (of\text{-}real\ r)) \Longrightarrow P\ q$
  by (*rule Reals-cases*) *auto*

107.4 Ordered real vector spaces

class *ordered-real-vector* = *real-vector* + *ordered-ab-group-add* +
  assumes *scaleR-left-mono*: $x \le y \Longrightarrow 0 \le a \Longrightarrow a *_R x \le a *_R y$
    and *scaleR-right-mono*: $a \le b \Longrightarrow 0 \le x \Longrightarrow a *_R x \le b *_R x$
begin

lemma *scaleR-mono*:
  $a \le b \Longrightarrow x \le y \Longrightarrow 0 \le b \Longrightarrow 0 \le x \Longrightarrow a *_R x \le b *_R y$
  by (*meson order-trans scaleR-left-mono scaleR-right-mono*)

lemma *scaleR-mono'*:
  $a \le b \Longrightarrow c \le d \Longrightarrow 0 \le a \Longrightarrow 0 \le c \Longrightarrow a *_R c \le b *_R d$
  by (*rule scaleR-mono*) (*auto intro*: *order.trans*)

lemma *pos-le-divideR-eq* [*field-simps*]:
  $a \le b /_R c \longleftrightarrow c *_R a \le b$ (**is** *?P* $\longleftrightarrow$ *?Q*) **if** $0 < c$
proof
  assume *?P*
  with *scaleR-left-mono that* **have** $c *_R a \le c *_R (b /_R c)$
    by *simp*
  with *that* **show** *?Q*
    by (*simp add*: *scaleR-one scaleR-scaleR inverse-eq-divide*)
next
  assume *?Q*
  with *scaleR-left-mono that* **have** $c *_R a /_R c \le b /_R c$
    by *simp*
  with *that* **show** *?P*
    by (*simp add*: *scaleR-one scaleR-scaleR inverse-eq-divide*)
qed

lemma *pos-less-divideR-eq* [*field-simps*]:
a < *b* /$_R$ *c* ⟷ *c* *$_R$ *a* < *b* **if** *c* > *0*
using *that pos-le-divideR-eq* [*of c a b*]
by (*auto simp add*: *le-less scaleR-scaleR scaleR-one*)

lemma *pos-divideR-le-eq* [*field-simps*]:
b /$_R$ *c* ≤ *a* ⟷ *b* ≤ *c* *$_R$ *a* **if** *c* > *0*
using *that pos-le-divideR-eq* [*of inverse c b a*] **by** *simp*

lemma *pos-divideR-less-eq* [*field-simps*]:
b /$_R$ *c* < *a* ⟷ *b* < *c* *$_R$ *a* **if** *c* > *0*
using *that pos-less-divideR-eq* [*of inverse c b a*] **by** *simp*

lemma *pos-le-minus-divideR-eq* [*field-simps*]:
a ≤ − (*b* /$_R$ *c*) ⟷ *c* *$_R$ *a* ≤ − *b* **if** *c* > *0*
using *that* **by** (*metis add-minus-cancel diff-0 left-minus minus-minus neg-le-iff-le scaleR-add-right uminus-add-conv-diff pos-le-divideR-eq*)

lemma *pos-less-minus-divideR-eq* [*field-simps*]:
a < − (*b* /$_R$ *c*) ⟷ *c* *$_R$ *a* < − *b* **if** *c* > *0*
using *that* **by** (*metis le-less less-le-not-le pos-divideR-le-eq pos-divideR-less-eq pos-le-minus-divideR-eq*)

lemma *pos-minus-divideR-le-eq* [*field-simps*]:
− (*b* /$_R$ *c*) ≤ *a* ⟷ − *b* ≤ *c* *$_R$ *a* **if** *c* > *0*
using *that* **by** (*metis pos-divideR-le-eq pos-le-minus-divideR-eq that inverse-positive-iff-positive le-imp-neg-le minus-minus*)

lemma *pos-minus-divideR-less-eq* [*field-simps*]:
− (*b* /$_R$ *c*) < *a* ⟷ − *b* < *c* *$_R$ *a* **if** *c* > *0*
using *that* **by** (*simp add*: *less-le-not-le pos-le-minus-divideR-eq pos-minus-divideR-le-eq*)

lemma *scaleR-image-atLeastAtMost*: *c* > *0* ⟹ *scaleR c* ‘ {*x*..*y*} = {*c* *$_R$ *x*..*c* *$_R$ *y*}
apply (*auto intro*!: *scaleR-left-mono simp*: *image-iff Bex-def*)
using *pos-divideR-le-eq* [*of c*] *pos-le-divideR-eq* [*of c*]
apply (*meson local.order-eq-iff*)
done

end

lemma *neg-le-divideR-eq* [*field-simps*]:
a ≤ *b* /$_R$ *c* ⟷ *b* ≤ *c* *$_R$ *a* (**is** *?P* ⟷ *?Q*) **if** *c* < *0*
for *a b* :: *'a* :: *ordered-real-vector*
using *that pos-le-divideR-eq* [*of − c a − b*] **by** *simp*

lemma *neg-less-divideR-eq* [*field-simps*]:
a < *b* /$_R$ *c* ⟷ *b* < *c* *$_R$ *a* **if** *c* < *0*

for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that neg-le-divideR-eq* [*of c a b*] **by** (*auto simp add*: *le-less*)

lemma *neg-divideR-le-eq* [*field-simps*]:
$b /_R c \leq a \longleftrightarrow c *_R a \leq b$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that pos-divideR-le-eq* [*of* $-\ c\ -\ b\ a$] **by** *simp*

lemma *neg-divideR-less-eq* [*field-simps*]:
$b /_R c < a \longleftrightarrow c *_R a < b$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that neg-divideR-le-eq* [*of c b a*] **by** (*auto simp add*: *le-less*)

lemma *neg-le-minus-divideR-eq* [*field-simps*]:
$a \leq - (b /_R c) \longleftrightarrow - b \leq c *_R a$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that pos-le-minus-divideR-eq* [*of* $-\ c\ a\ -\ b$] **by** (*simp add*: *minus-le-iff*)

lemma *neg-less-minus-divideR-eq* [*field-simps*]:
$a < - (b /_R c) \longleftrightarrow - b < c *_R a$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
proof −
have $*$: $- b = c *_R a \longleftrightarrow b = - (c *_R a)$
by (*metis add.inverse-inverse*)
from *that neg-le-minus-divideR-eq* [*of c a b*]
show *?thesis* **by** (*auto simp add*: *le-less* $*$)
qed

lemma *neg-minus-divideR-le-eq* [*field-simps*]:
$- (b /_R c) \leq a \longleftrightarrow c *_R a \leq - b$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that pos-minus-divideR-le-eq* [*of* $-\ c\ -\ b\ a$] **by** (*simp add*: *le-minus-iff*)

lemma *neg-minus-divideR-less-eq* [*field-simps*]:
$- (b /_R c) < a \longleftrightarrow c *_R a < - b$ **if** $c < 0$
for $a\ b :: 'a :: ordered\text{-}real\text{-}vector$
using *that* **by** (*simp add*: *less-le-not-le neg-le-minus-divideR-eq neg-minus-divideR-le-eq*)

lemma [*field-split-simps*]:
$a = b /_R c \longleftrightarrow (if\ c = 0\ then\ a = 0\ else\ c *_R a = b)$
$b /_R c = a \longleftrightarrow (if\ c = 0\ then\ a = 0\ else\ b = c *_R a)$
$a + b /_R c = (if\ c = 0\ then\ a\ else\ (c *_R a + b) /_R c)$
$a /_R c + b = (if\ c = 0\ then\ b\ else\ (a + c *_R b) /_R c)$
$a - b /_R c = (if\ c = 0\ then\ a\ else\ (c *_R a - b) /_R c)$
$a /_R c - b = (if\ c = 0\ then - b\ else\ (a - c *_R b) /_R c)$
$- (a /_R c) + b = (if\ c = 0\ then\ b\ else\ (- a + c *_R b) /_R c)$
$- (a /_R c) - b = (if\ c = 0\ then - b\ else\ (- a - c *_R b) /_R c)$
for $a\ b :: 'a :: real\text{-}vector$
by (*auto simp add*: *field-simps*)

lemma [*field-split-simps*]:
*0 < c ⟹ a ≤ b /R c ⟷ (if c > 0 then c *R a ≤ b else if c < 0 then b ≤ c *R a else a ≤ 0)*
*0 < c ⟹ a < b /R c ⟷ (if c > 0 then c *R a < b else if c < 0 then b < c *R a else a < 0)*
*0 < c ⟹ b /R c ≤ a ⟷ (if c > 0 then b ≤ c *R a else if c < 0 then c *R a ≤ b else a ≥ 0)*
*0 < c ⟹ b /R c < a ⟷ (if c > 0 then b < c *R a else if c < 0 then c *R a < b else a > 0)*
*0 < c ⟹ a ≤ − (b /R c) ⟷ (if c > 0 then c *R a ≤ − b else if c < 0 then − b ≤ c *R a else a ≤ 0)*
*0 < c ⟹ a < − (b /R c) ⟷ (if c > 0 then c *R a < − b else if c < 0 then − b < c *R a else a < 0)*
*0 < c ⟹ − (b /R c) ≤ a ⟷ (if c > 0 then − b ≤ c *R a else if c < 0 then c *R a ≤ − b else a ≥ 0)*
*0 < c ⟹ − (b /R c) < a ⟷ (if c > 0 then − b < c *R a else if c < 0 then c *R a < − b else a > 0)*
for *a b* :: *'a* :: *ordered-real-vector*
by (*clarsimp intro*!: *field-simps*)+

lemma *scaleR-nonneg-nonneg*: *0 ≤ a ⟹ 0 ≤ x ⟹ 0 ≤ a *R x*
for *x* :: *'a*::*ordered-real-vector*
using *scaleR-left-mono* [*of 0 x a*] **by** *simp*

lemma *scaleR-nonneg-nonpos*: *0 ≤ a ⟹ x ≤ 0 ⟹ a *R x ≤ 0*
for *x* :: *'a*::*ordered-real-vector*
using *scaleR-left-mono* [*of x 0 a*] **by** *simp*

lemma *scaleR-nonpos-nonneg*: *a ≤ 0 ⟹ 0 ≤ x ⟹ a *R x ≤ 0*
for *x* :: *'a*::*ordered-real-vector*
using *scaleR-right-mono* [*of a 0 x*] **by** *simp*

lemma *split-scaleR-neg-le*: *(0 ≤ a ∧ x ≤ 0) ∨ (a ≤ 0 ∧ 0 ≤ x) ⟹ a *R x ≤ 0*
for *x* :: *'a*::*ordered-real-vector*
by (*auto simp*: *scaleR-nonneg-nonpos scaleR-nonpos-nonneg*)

lemma *le-add-iff1*: *a *R e + c ≤ b *R e + d ⟷ (a − b) *R e + c ≤ d*
for *c d e* :: *'a*::*ordered-real-vector*
by (*simp add*: *algebra-simps*)

lemma *le-add-iff2*: *a *R e + c ≤ b *R e + d ⟷ c ≤ (b − a) *R e + d*
for *c d e* :: *'a*::*ordered-real-vector*
by (*simp add*: *algebra-simps*)

lemma *scaleR-left-mono-neg*: *b ≤ a ⟹ c ≤ 0 ⟹ c *R a ≤ c *R b*
for *a b* :: *'a*::*ordered-real-vector*
by (*drule scaleR-left-mono* [*of - - − c*], *simp-all*)

lemma *scaleR-right-mono-neg*: $b \leq a \Longrightarrow c \leq 0 \Longrightarrow a *_R c \leq b *_R c$
for $c :: 'a$::*ordered-real-vector*
by (*drule scaleR-right-mono* [*of* - - − *c*], *simp-all*)

lemma *scaleR-nonpos-nonpos*: $a \leq 0 \Longrightarrow b \leq 0 \Longrightarrow 0 \leq a *_R b$
for $b :: 'a$::*ordered-real-vector*
using *scaleR-right-mono-neg* [*of a 0 b*] **by** *simp*

lemma *split-scaleR-pos-le*: $(0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0) \Longrightarrow 0 \leq a *_R b$
for $b :: 'a$::*ordered-real-vector*
by (*auto simp*: *scaleR-nonneg-nonneg scaleR-nonpos-nonpos*)

lemma *zero-le-scaleR-iff*:
fixes $b :: 'a$::*ordered-real-vector*
shows $0 \leq a *_R b \longleftrightarrow 0 < a \wedge 0 \leq b \vee a < 0 \wedge b \leq 0 \vee a = 0$
(**is** *?lhs* = *?rhs*)
proof (*cases* $a = 0$)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
show *?thesis*
proof
assume *?lhs*
from ‹$a \neq 0$› **consider** $a > 0 \mid a < 0$ **by** *arith*
then show *?rhs*
proof *cases*
case *1*
with ‹*?lhs*› **have** *inverse a* $*_R$ $0 \leq$ *inverse a* $*_R$ ($a *_R b$)
by (*intro scaleR-mono*) *auto*
with *1* **show** *?thesis*
by *simp*
next
case *2*
with ‹*?lhs*› **have** $-$ *inverse a* $*_R$ $0 \leq -$ *inverse a* $*_R$ ($a *_R b$)
by (*intro scaleR-mono*) *auto*
with *2* **show** *?thesis*
by *simp*
qed
next
assume *?rhs*
then show *?lhs*
by (*auto simp*: *not-le* ‹$a \neq 0$› *intro*!: *split-scaleR-pos-le*)
qed
qed

lemma *scaleR-le-0-iff*: $a *_R b \leq 0 \longleftrightarrow 0 < a \wedge b \leq 0 \vee a < 0 \wedge 0 \leq b \vee a = 0$
for *b*::$'a$::*ordered-real-vector*
by (*insert zero-le-scaleR-iff* [*of* $-a$ *b*]) *force*

lemma *scaleR-le-cancel-left*: $c *_R a \leq c *_R b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
for $b :: 'a$::*ordered-real-vector*
by (*auto simp*: *neq-iff scaleR-left-mono scaleR-left-mono-neg*
dest: *scaleR-left-mono*[**where** *a*=*inverse c*] *scaleR-left-mono-neg*[**where** *c*=*inverse c*])

lemma *scaleR-le-cancel-left-pos*: $0 < c \Longrightarrow c *_R a \leq c *_R b \longleftrightarrow a \leq b$
for $b :: 'a$::*ordered-real-vector*
by (*auto simp*: *scaleR-le-cancel-left*)

lemma *scaleR-le-cancel-left-neg*: $c < 0 \Longrightarrow c *_R a \leq c *_R b \longleftrightarrow b \leq a$
for $b :: 'a$::*ordered-real-vector*
by (*auto simp*: *scaleR-le-cancel-left*)

lemma *scaleR-left-le-one-le*: $0 \leq x \Longrightarrow a \leq 1 \Longrightarrow a *_R x \leq x$
for $x :: 'a$::*ordered-real-vector* **and** $a :: real$
using *scaleR-right-mono*[*of a 1 x*] **by** *simp*

107.5 Real normed vector spaces

class *dist* =
fixes *dist* :: $'a \Rightarrow 'a \Rightarrow real$

class *norm* =
fixes *norm* :: $'a \Rightarrow real$

class *sgn-div-norm* = *scaleR* + *norm* + *sgn* +
assumes *sgn-div-norm*: $sgn\ x = x\ /_R\ norm\ x$

class *dist-norm* = *dist* + *norm* + *minus* +
assumes *dist-norm*: $dist\ x\ y = norm\ (x - y)$

class *uniformity-dist* = *dist* + *uniformity* +
assumes *uniformity-dist*: $uniformity = (INF\ e{\in}\{0 <..\}.\ principal\ \{(x, y).\ dist\ x\ y < e\})$
begin

lemma *eventually-uniformity-metric*:
$eventually\ P\ uniformity \longleftrightarrow (\exists e{>}0.\ \forall x\ y.\ dist\ x\ y < e \longrightarrow P\ (x, y))$
unfolding *uniformity-dist*
by (*subst eventually-INF-base*)
(*auto simp*: *eventually-principal subset-eq intro*: *bexI*[*of - min - -*])

end

class *real-normed-vector* = *real-vector* + *sgn-div-norm* + *dist-norm* + *uniformity-dist* + *open-uniformity* +

assumes *norm-eq-zero* [*simp*]: *norm x = 0 ⟷ x = 0*
and *norm-triangle-ineq*: *norm (x + y) ≤ norm x + norm y*
and *norm-scaleR* [*simp*]: *norm (scaleR a x) = |a| * norm x*
begin

lemma *norm-ge-zero* [*simp*]: *0 ≤ norm x*
proof −
have *0 = norm (x + −1 *$_R$ x)*
using *scaleR-add-left*[*of 1 −1 x*] *norm-scaleR*[*of 0 x*] **by** (*simp add*: *scaleR-one*)
also have *... ≤ norm x + norm (−1 *$_R$ *x)* **by** (*rule norm-triangle-ineq*)
finally show *?thesis* **by** *simp*
qed

lemma *bdd-below-norm-image*: *bdd-below (norm ‘ A)*
by (*meson bdd-belowI2 norm-ge-zero*)

end

class *real-normed-algebra = real-algebra + real-normed-vector +*
assumes *norm-mult-ineq*: *norm (x * y) ≤ norm x * norm y*

class *real-normed-algebra-1 = real-algebra-1 + real-normed-algebra +*
assumes *norm-one* [*simp*]: *norm 1 = 1*

lemma (**in** *real-normed-algebra-1*) *scaleR-power* [*simp*]: *(scaleR x y) ^ n = scaleR (x^n) (y^n)*
by (*induct n*) (*simp-all add*: *scaleR-one scaleR-scaleR mult-ac*)

class *real-normed-div-algebra = real-div-algebra + real-normed-vector +*
assumes *norm-mult*: *norm (x * y) = norm x * norm y*

lemma *divideR-right*:
fixes *x y :: 'a::real-normed-vector*
shows *r ≠ 0 ⟹ y = x /$_R$ r ⟷ r *$_R$ *y = x*
by *auto*

class *real-normed-field = real-field + real-normed-div-algebra*

lemma *dist-mult-left*:
*dist (a * b) (a * c :: 'a :: real-normed-field) = norm a * dist b c*
unfolding *dist-norm right-diff-distrib* [*symmetric*] *norm-mult* **by** *simp*

lemma *dist-mult-right*:
*dist (b * a) (c * a :: 'a :: real-normed-field) = norm a * dist b c*
using *dist-mult-left*[*of a b c*] **by** (*simp add*: *mult-ac*)

instance *real-normed-div-algebra < real-normed-algebra-1*
proof
show *norm (x * y) ≤ norm x * norm y* **for** *x y :: 'a*

```
    by (simp add: norm-mult)
next
  have norm (1 * 1::'a) = norm (1::'a) * norm (1::'a)
    by (rule norm-mult)
  then show norm (1::'a) = 1 by simp
qed

context real-normed-vector begin

lemma norm-zero [simp]: norm (0::'a) = 0
  by simp

lemma zero-less-norm-iff [simp]: norm x > 0 ⟷ x ≠ 0
  by (simp add: order-less-le)

lemma norm-not-less-zero [simp]: ¬ norm x < 0
  by (simp add: linorder-not-less)

lemma norm-le-zero-iff [simp]: norm x ≤ 0 ⟷ x = 0
  by (simp add: order-le-less)

lemma norm-minus-cancel [simp]: norm (- x) = norm x
proof -
  have - 1 *R x = - (1 *R x)
    unfolding add-eq-0-iff2[symmetric] scaleR-add-left[symmetric]
    using norm-eq-zero
    by fastforce
  then have norm (- x) = norm (scaleR (- 1) x)
    by (simp only: scaleR-one)
  also have ... = |- 1| * norm x
    by (rule norm-scaleR)
  finally show ?thesis by simp
qed

lemma norm-minus-commute: norm (a - b) = norm (b - a)
proof -
  have norm (- (b - a)) = norm (b - a)
    by (rule norm-minus-cancel)
  then show ?thesis by simp
qed

lemma dist-add-cancel [simp]: dist (a + b) (a + c) = dist b c
  by (simp add: dist-norm)

lemma dist-add-cancel2 [simp]: dist (b + a) (c + a) = dist b c
  by (simp add: dist-norm)

lemma norm-uminus-minus: norm (- x - y) = norm (x + y)
  by (subst (2) norm-minus-cancel[symmetric], subst minus-add-distrib) simp
```

lemma *norm-triangle-ineq2*: *norm a − norm b ≤ norm (a − b)*
proof −
  have *norm (a − b + b) ≤ norm (a − b) + norm b*
    by (*rule norm-triangle-ineq*)
  then show *?thesis* **by** *simp*
qed

lemma *norm-triangle-ineq3*: *|norm a − norm b| ≤ norm (a − b)*
proof −
  have *norm a − norm b ≤ norm (a − b)*
    by (*simp add*: *norm-triangle-ineq2*)
  moreover have *norm b − norm a ≤ norm (a − b)*
    by (*metis norm-minus-commute norm-triangle-ineq2*)
  ultimately show *?thesis*
    by (*simp add*: *abs-le-iff*)
qed

lemma *norm-triangle-ineq4*: *norm (a − b) ≤ norm a + norm b*
proof −
  have *norm (a + − b) ≤ norm a + norm (− b)*
    by (*rule norm-triangle-ineq*)
  then show *?thesis* **by** *simp*
qed

lemma *norm-triangle-le-diff*: *norm x + norm y ≤ e ⟹ norm (x − y) ≤ e*
    by (*meson norm-triangle-ineq4 order-trans*)

lemma *norm-diff-ineq*: *norm a − norm b ≤ norm (a + b)*
proof −
  have *norm a − norm (− b) ≤ norm (a − − b)*
    by (*rule norm-triangle-ineq2*)
  then show *?thesis* **by** *simp*
qed

lemma *norm-triangle-sub*: *norm x ≤ norm y + norm (x − y)*
  using *norm-triangle-ineq*[*of y x − y*] **by** (*simp add*: *field-simps*)

lemma *norm-triangle-le*: *norm x + norm y ≤ e ⟹ norm (x + y) ≤ e*
  by (*rule norm-triangle-ineq* [*THEN order-trans*])

lemma *norm-triangle-lt*: *norm x + norm y < e ⟹ norm (x + y) < e*
  by (*rule norm-triangle-ineq* [*THEN le-less-trans*])

lemma *norm-add-leD*: *norm (a + b) ≤ c ⟹ norm b ≤ norm a + c*
 by (*metis ab-semigroup-add-class.add.commute add-commute diff-le-eq norm-diff-ineq order-trans*)

lemma *norm-diff-triangle-ineq*: *norm ((a + b) − (c + d)) ≤ norm (a − c) +*

norm $(b - d)$
proof −
have *norm* $((a + b) - (c + d)) = norm\ ((a - c) + (b - d))$
by (*simp add*: *algebra-simps*)
also have $\ldots \leq norm\ (a - c) + norm\ (b - d)$
by (*rule norm-triangle-ineq*)
finally show *?thesis* **.**
qed

lemma *norm-diff-triangle-le*: $norm\ (x - z) \leq e1 + e2$
if $norm\ (x - y) \leq e1$ $norm\ (y - z) \leq e2$
proof −
have $norm\ (x - (y + z - y)) \leq norm\ (x - y) + norm\ (y - z)$
using *norm-diff-triangle-ineq that diff-diff-eq2* **by** *presburger*
with *that* **show** *?thesis* **by** *simp*
qed

lemma *norm-diff-triangle-less*: $norm\ (x - z) < e1 + e2$
if $norm\ (x - y) < e1$ $norm\ (y - z) < e2$
proof −
have $norm\ (x - z) \leq norm\ (x - y) + norm\ (y - z)$
by (*metis norm-diff-triangle-ineq add-diff-cancel-left' diff-diff-eq2*)
with *that* **show** *?thesis* **by** *auto*
qed

lemma *norm-triangle-mono*:
$norm\ a \leq r \Longrightarrow norm\ b \leq s \Longrightarrow norm\ (a + b) \leq r + s$
by (*metis* (*mono-tags*) *add-mono-thms-linordered-semiring*(*1*) *norm-triangle-ineq order.trans*)

lemma *norm-sum*: $norm\ (sum\ f\ A) \leq (\sum i{\in}A.\ norm\ (f\ i))$
for $f{::}'b \Rightarrow 'a$
by (*induct A rule*: *infinite-finite-induct*) (*auto intro*: *norm-triangle-mono*)

lemma *sum-norm-le*: $norm\ (sum\ f\ S) \leq sum\ g\ S$
if $\bigwedge x.\ x \in S \Longrightarrow norm\ (f\ x) \leq g\ x$
for $f{::}'b \Rightarrow 'a$
by (*rule order-trans* [*OF norm-sum sum-mono*]) (*simp add*: *that*)

lemma *abs-norm-cancel* [*simp*]: $|norm\ a| = norm\ a$
by (*rule abs-of-nonneg* [*OF norm-ge-zero*])

lemma *sum-norm-bound*:
$norm\ (sum\ f\ S) \leq of\text{-}nat\ (card\ S){*}K$
if $\bigwedge x.\ x \in S \Longrightarrow norm\ (f\ x) \leq K$
for $f :: 'b \Rightarrow 'a$
using *sum-norm-le*[*OF that*] *sum-constant*[*symmetric*]
by *simp*

lemma *norm-add-less*: *norm x < r ⟹ norm y < s ⟹ norm (x + y) < r + s*
by (*rule order-le-less-trans* [*OF norm-triangle-ineq add-strict-mono*])

end

lemma *dist-sum-le*:
fixes *f* :: *'a ⇒ 'b* :: *real-normed-vector*
shows *dist* (∑ *x*∈*A*. *f x*) (∑ *x*∈*A*. *g x*) ≤ (∑ *x*∈*A*. *dist* (*f x*) (*g x*))
proof −
have *dist* (∑ *x*∈*A*. *f x*) (∑ *x*∈*A*. *g x*) = *norm* (∑ *x*∈*A*. *f x* − *g x*)
by (*simp add*: *dist-norm sum-subtractf*)
also have ... ≤ (∑ *x*∈*A*. *norm* (*f x* − *g x*))
by (*rule norm-sum*)
finally show *?thesis*
by (*simp add*: *dist-norm*)
qed

lemma *dist-scaleR* [*simp*]: *dist* (*x* ∗*R* *a*) (*y* ∗*R* *a*) = |*x* − *y*| ∗ *norm a*
for *a* :: *'a*::*real-normed-vector*
by (*metis dist-norm norm-scaleR scaleR-left.diff*)

lemma *norm-mult-less*: *norm x < r ⟹ norm y < s ⟹ norm* (*x* ∗ *y*) < *r* ∗ *s*
for *x y* :: *'a*::*real-normed-algebra*
by (*rule order-le-less-trans* [*OF norm-mult-ineq*]) (*simp add*: *mult-strict-mono'*)

lemma *norm-of-real* [*simp*]: *norm* (*of-real r* :: *'a*::*real-normed-algebra-1*) = |*r*|
by (*simp add*: *of-real-def*)

lemma *norm-numeral* [*simp*]: *norm* (*numeral w*::*'a*::*real-normed-algebra-1*) = *numeral w*
by (*subst of-real-numeral* [*symmetric*], *subst norm-of-real*, *simp*)

lemma *norm-neg-numeral* [*simp*]: *norm* (− *numeral w*::*'a*::*real-normed-algebra-1*) = *numeral w*
by (*subst of-real-neg-numeral* [*symmetric*], *subst norm-of-real*, *simp*)

lemma *norm-of-real-add1* [*simp*]: *norm* (*of-real x* + *1* :: *'a* :: *real-normed-div-algebra*) = |*x* + *1*|
by (*metis norm-of-real of-real-1 of-real-add*)

lemma *norm-of-real-addn* [*simp*]:
norm (*of-real x* + *numeral b* :: *'a* :: *real-normed-div-algebra*) = |*x* + *numeral b*|
by (*metis norm-of-real of-real-add of-real-numeral*)

lemma *norm-of-int* [*simp*]: *norm* (*of-int z*::*'a*::*real-normed-algebra-1*) = |*of-int z*|
by (*subst of-real-of-int-eq* [*symmetric*], *rule norm-of-real*)

lemma *norm-of-nat* [*simp*]: *norm* (*of-nat n*::*'a*::*real-normed-algebra-1*) = *of-nat n*
by (*metis abs-of-nat norm-of-real of-real-of-nat-eq*)

lemma *nonzero-norm-inverse*: $a \neq 0 \Longrightarrow norm\ (inverse\ a) = inverse\ (norm\ a)$
for $a :: 'a$::*real-normed-div-algebra*
by (*metis inverse-unique norm-mult norm-one right-inverse*)

lemma *norm-inverse*: $norm\ (inverse\ a) = inverse\ (norm\ a)$
for $a :: 'a$::{*real-normed-div-algebra*,*division-ring*}
by (*metis inverse-zero nonzero-norm-inverse norm-zero*)

lemma *nonzero-norm-divide*: $b \neq 0 \Longrightarrow norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
for $a\ b :: 'a$::*real-normed-field*
by (*simp add*: *divide-inverse norm-mult nonzero-norm-inverse*)

lemma *norm-divide*: $norm\ (a\ /\ b) = norm\ a\ /\ norm\ b$
for $a\ b :: 'a$::{*real-normed-field*,*field*}
by (*simp add*: *divide-inverse norm-mult norm-inverse*)

lemma *dist-divide-right*: $dist\ (a/c)\ (b/c) = dist\ a\ b\ /\ norm\ c$ **for** $c :: 'a$ :: *real-normed-field*
by (*metis diff-divide-distrib dist-norm norm-divide*)

lemma *norm-inverse-le-norm*:
fixes $x :: 'a$::*real-normed-div-algebra*
shows $r \leq norm\ x \Longrightarrow 0 < r \Longrightarrow norm\ (inverse\ x) \leq inverse\ r$
by (*simp add*: *le-imp-inverse-le norm-inverse*)

lemma *norm-power-ineq*: $norm\ (x\ \hat{}\ n) \leq norm\ x\ \hat{}\ n$
for $x :: 'a$::*real-normed-algebra-1*
proof (*induct n*)
case *0*
show $norm\ (x\ \hat{}\ 0) \leq norm\ x\ \hat{}\ 0$ **by** *simp*
next
case (*Suc n*)
have $norm\ (x * x\ \hat{}\ n) \leq norm\ x * norm\ (x\ \hat{}\ n)$
by (*rule norm-mult-ineq*)
also from *Suc* **have** $\ldots \leq norm\ x * norm\ x\ \hat{}\ n$
using *norm-ge-zero* **by** (*rule mult-left-mono*)
finally show $norm\ (x\ \hat{}\ Suc\ n) \leq norm\ x\ \hat{}\ Suc\ n$
by *simp*
qed

lemma *norm-power*: $norm\ (x\ \hat{}\ n) = norm\ x\ \hat{}\ n$
for $x :: 'a$::*real-normed-div-algebra*
by (*induct n*) (*simp-all add*: *norm-mult*)

lemma *norm-power-int*: $norm\ (power\text{-}int\ x\ n) = power\text{-}int\ (norm\ x)\ n$
for $x :: 'a$::*real-normed-div-algebra*
by (*cases n rule*: *int-cases4*) (*auto simp*: *norm-power power-int-minus norm-inverse*)

lemma *power-eq-imp-eq-norm*:
fixes w :: *'a*::*real-normed-div-algebra*
assumes *eq*: $w \hat{} n = z \hat{} n$ **and** $n > 0$
shows *norm* w = *norm* z
proof −
have *norm* $w \hat{} n$ = *norm* $z \hat{} n$
by (*metis* (*no-types*) *eq norm-power*)
then show *?thesis*
using *assms* **by** (*force intro*: *power-eq-imp-eq-base*)
qed

lemma *power-eq-1-iff*:
fixes w :: *'a*::*real-normed-div-algebra*
shows $w \hat{} n = 1 \Longrightarrow$ *norm* $w = 1 \vee n = 0$
by (*metis norm-one power-0-left power-eq-0-iff power-eq-imp-eq-norm power-one*)

lemma *norm-mult-numeral1* [*simp*]: *norm* (*numeral* $w * a$) = *numeral* $w *$ *norm* a
for a b :: *'a*::{*real-normed-field*,*field*}
by (*simp add*: *norm-mult*)

lemma *norm-mult-numeral2* [*simp*]: *norm* ($a *$ *numeral* w) = *norm* $a *$ *numeral* w
for a b :: *'a*::{*real-normed-field*,*field*}
by (*simp add*: *norm-mult*)

lemma *norm-divide-numeral* [*simp*]: *norm* (a / *numeral* w) = *norm* a / *numeral* w
for a b :: *'a*::{*real-normed-field*,*field*}
by (*simp add*: *norm-divide*)

lemma *norm-of-real-diff* [*simp*]:
norm (*of-real* b − *of-real* a :: *'a*::*real-normed-algebra-1*) ≤ $|b - a|$
by (*metis norm-of-real of-real-diff order-refl*)

Despite a superficial resemblance, *norm-eq-1* is not relevant.

lemma *square-norm-one*:
fixes x :: *'a*::*real-normed-div-algebra*
assumes $x^2 = 1$
shows *norm* $x = 1$
by (*metis assms norm-minus-cancel norm-one power2-eq-1-iff*)

lemma *norm-less-p1*: *norm* $x <$ *norm* (*of-real* (*norm* x) + *1* :: *'a*)
for x :: *'a*::*real-normed-algebra-1*
proof −
have *norm* $x <$ *norm* (*of-real* (*norm* x + *1*) :: *'a*)
by (*simp add*: *of-real-def*)
then show *?thesis*
by *simp*

qed

lemma *prod-norm*: *prod* (λ*x*. *norm* (*f x*)) *A* = *norm* (*prod f A*)
for *f* :: *'a* ⇒ *'b*::{*comm-semiring-1*,*real-normed-div-algebra*}
by (*induct A rule*: *infinite-finite-induct*) (*auto simp*: *norm-mult*)

lemma *norm-prod-le*:
norm (*prod f A*) ≤ (∏ *a*∈*A*. *norm* (*f a* :: *'a* :: {*real-normed-algebra-1*,*comm-monoid-mult*}))
proof (*induct A rule*: *infinite-finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
then have *norm* (*prod f* (*insert a A*)) ≤ *norm* (*f a*) ∗ *norm* (*prod f A*)
by (*simp add*: *norm-mult-ineq*)
also have *norm* (*prod f A*) ≤ (∏ *a*∈*A*. *norm* (*f a*))
by (*rule insert*)
finally show *?case*
by (*simp add*: *insert mult-left-mono*)
next
case *infinite*
then show *?case* **by** *simp*
qed

lemma *norm-prod-diff*:
fixes *z w* :: *'i* ⇒ *'a*::{*real-normed-algebra-1*, *comm-monoid-mult*}
shows (⋀*i*. *i* ∈ *I* ⟹ *norm* (*z i*) ≤ *1*) ⟹ (⋀*i*. *i* ∈ *I* ⟹ *norm* (*w i*) ≤ *1*) ⟹
norm ((∏ *i*∈*I*. *z i*) − (∏ *i*∈*I*. *w i*)) ≤ (∑ *i*∈*I*. *norm* (*z i* − *w i*))
proof (*induction I rule*: *infinite-finite-induct*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert i I*)
note *insert.hyps*[*simp*]

have *norm* ((∏ *i*∈*insert i I*. *z i*) − (∏ *i*∈*insert i I*. *w i*)) =
norm ((∏ *i*∈*I*. *z i*) ∗ (*z i* − *w i*) + ((∏ *i*∈*I*. *z i*) − (∏ *i*∈*I*. *w i*)) ∗ *w i*)
(**is** - = *norm* (*?t1* + *?t2*))
by (*auto simp*: *field-simps*)
also have ... ≤ *norm ?t1* + *norm ?t2*
by (*rule norm-triangle-ineq*)
also have *norm ?t1* ≤ *norm* (∏ *i*∈*I*. *z i*) ∗ *norm* (*z i* − *w i*)
by (*rule norm-mult-ineq*)
also have ... ≤ (∏ *i*∈*I*. *norm* (*z i*)) ∗ *norm*(*z i* − *w i*)
by (*rule mult-right-mono*) (*auto intro*: *norm-prod-le*)
also have (∏ *i*∈*I*. *norm* (*z i*)) ≤ (∏ *i*∈*I*. *1*)
by (*intro prod-mono*) (*auto intro*!: *insert*)
also have *norm ?t2* ≤ *norm* ((∏ *i*∈*I*. *z i*) − (∏ *i*∈*I*. *w i*)) ∗ *norm* (*w i*)
by (*rule norm-mult-ineq*)

also have *norm* $(w\ i) \leq 1$
by (*auto intro*: *insert*)
also have *norm* $((\prod i{\in}I.\ z\ i) - (\prod i{\in}I.\ w\ i)) \leq (\sum i{\in}I.\ norm\ (z\ i - w\ i))$
using *insert* **by** *auto*
finally show *?case*
by (*auto simp*: *ac-simps mult-right-mono mult-left-mono*)
next
case *infinite*
then show *?case* **by** *simp*
qed

lemma *norm-power-diff*:
fixes *z w* :: *'a*::{*real-normed-algebra-1*, *comm-monoid-mult*}
assumes *norm z* $\leq$ *1 norm w* $\leq$ *1*
shows *norm* $(z\hat{}m - w\hat{}m) \leq m * norm\ (z - w)$
proof −
have *norm* $(z\hat{}m - w\hat{}m) = norm\ ((\prod\ i < m.\ z) - (\prod\ i < m.\ w))$
by *simp*
also have $\ldots \leq (\sum i{<}m.\ norm\ (z - w))$
by (*intro norm-prod-diff*) (*auto simp*: *assms*)
also have $\ldots = m * norm\ (z - w)$
by *simp*
finally show *?thesis* **.**
qed

107.6 Metric spaces

class *metric-space* = *uniformity-dist* + *open-uniformity* +
assumes *dist-eq-0-iff* [*simp*]: *dist x y* = *0* $\longleftrightarrow$ *x* = *y*
and *dist-triangle2*: *dist x y* $\leq$ *dist x z* + *dist y z*
begin

lemma *dist-self* [*simp*]: *dist x x* = *0*
by *simp*

lemma *zero-le-dist* [*simp*]: *0* $\leq$ *dist x y*
using *dist-triangle2* [*of x x y*] **by** *simp*

lemma *zero-less-dist-iff*: *0* < *dist x y* $\longleftrightarrow$ *x* $\neq$ *y*
by (*simp add*: *less-le*)

lemma *dist-not-less-zero* [*simp*]: ¬ *dist x y* < *0*
by (*simp add*: *not-less*)

lemma *dist-le-zero-iff* [*simp*]: *dist x y* $\leq$ *0* $\longleftrightarrow$ *x* = *y*
by (*simp add*: *le-less*)

lemma *dist-commute*: *dist x y* = *dist y x*
proof (*rule order-antisym*)

show *dist x y ≤ dist y x*
using *dist-triangle2* [*of x y x*] **by** *simp*
show *dist y x ≤ dist x y*
using *dist-triangle2* [*of y x y*] **by** *simp*
qed

lemma *dist-commute-lessI*: *dist y x < e ⟹ dist x y < e*
by (*simp add*: *dist-commute*)

lemma *dist-triangle*: *dist x z ≤ dist x y + dist y z*
using *dist-triangle2* [*of x z y*] **by** (*simp add*: *dist-commute*)

lemma *dist-triangle3*: *dist x y ≤ dist a x + dist a y*
using *dist-triangle2* [*of x y a*] **by** (*simp add*: *dist-commute*)

lemma *abs-dist-diff-le*: *|dist a b − dist b c| ≤ dist a c*
using *dist-triangle3*[*of b c a*] *dist-triangle2*[*of a b c*] **by** *simp*

lemma *dist-pos-lt*: *x ≠ y ⟹ 0 < dist x y*
by (*simp add*: *zero-less-dist-iff*)

lemma *dist-nz*: *x ≠ y ⟷ 0 < dist x y*
by (*simp add*: *zero-less-dist-iff*)

declare *dist-nz* [*symmetric*, *simp*]

lemma *dist-triangle-le*: *dist x z + dist y z ≤ e ⟹ dist x y ≤ e*
by (*rule order-trans* [*OF dist-triangle2*])

lemma *dist-triangle-lt*: *dist x z + dist y z < e ⟹ dist x y < e*
by (*rule le-less-trans* [*OF dist-triangle2*])

lemma *dist-triangle-less-add*: *dist x1 y < e1 ⟹ dist x2 y < e2 ⟹ dist x1 x2 < e1 + e2*
by (*rule dist-triangle-lt* [**where** *z*=*y*]) *simp*

lemma *dist-triangle-half-l*: *dist x1 y < e / 2 ⟹ dist x2 y < e / 2 ⟹ dist x1 x2 < e*
by (*rule dist-triangle-lt* [**where** *z*=*y*]) *simp*

lemma *dist-triangle-half-r*: *dist y x1 < e / 2 ⟹ dist y x2 < e / 2 ⟹ dist x1 x2 < e*
by (*rule dist-triangle-half-l*) (*simp-all add*: *dist-commute*)

lemma *dist-triangle-third*:
assumes *dist x1 x2 < e/3 dist x2 x3 < e/3 dist x3 x4 < e/3*
shows *dist x1 x4 < e*
proof −
have *dist x1 x3 < e/3 + e/3*

```
    by (metis assms(1) assms(2) dist-commute dist-triangle-less-add)
  then have dist x1 x4 < (e/3 + e/3) + e/3
    by (metis assms(3) dist-commute dist-triangle-less-add)
  then show ?thesis
    by simp
qed

subclass uniform-space
proof
  fix E x
  assume eventually E uniformity
  then obtain e where E: 0 < e ⋀x y. dist x y < e ⟹ E (x, y)
    by (auto simp: eventually-uniformity-metric)
  then show E (x, x) ∀F (x, y) in uniformity. E (y, x)
    by (auto simp: eventually-uniformity-metric dist-commute)
  show ∃D. eventually D uniformity ∧ (∀x y z. D (x, y) ⟶ D (y, z) ⟶ E (x,
z))
    using E dist-triangle-half-l[where e=e]
    unfolding eventually-uniformity-metric
    by (intro exI[of - λ(x, y). dist x y < e / 2] exI[of - e/2] conjI)
      (auto simp: dist-commute)
qed

lemma open-dist: open S ⟷ (∀x∈S. ∃e>0. ∀y. dist y x < e ⟶ y ∈ S)
  by (simp add: dist-commute open-uniformity eventually-uniformity-metric)

lemma open-ball: open {y. dist x y < d}
  unfolding open-dist
proof (intro ballI)
  fix y
  assume ∗: y ∈ {y. dist x y < d}
  then show ∃e>0. ∀z. dist z y < e ⟶ z ∈ {y. dist x y < d}
    by (auto intro!: exI[of - d − dist x y] simp: field-simps dist-triangle-lt)
qed

subclass first-countable-topology
proof
  fix x
  show ∃A::nat ⇒ 'a set. (∀i. x ∈ A i ∧ open (A i)) ∧ (∀S. open S ∧ x ∈ S ⟶
(∃i. A i ⊆ S))
  proof (safe intro!: exI[of - λn. {y. dist x y < inverse (Suc n)}])
    fix S
    assume open S x ∈ S
    then obtain e where e: 0 < e and {y. dist x y < e} ⊆ S
      by (auto simp: open-dist subset-eq dist-commute)
    moreover
    from e obtain i where inverse (Suc i) < e
      by (auto dest!: reals-Archimedean)
    then have {y. dist x y < inverse (Suc i)} ⊆ {y. dist x y < e}
```

by *auto*
ultimately show $\exists i.\ \{y.\ dist\ x\ y < inverse\ (Suc\ i)\} \subseteq S$
by *blast*
qed (*auto intro*: *open-ball*)
qed

end

instance *metric-space* $\subseteq$ *t2-space*
proof
fix *x y* :: *'a*::*metric-space*
assume *xy*: $x \neq y$
let *?U* = $\{y'.\ dist\ x\ y' < dist\ x\ y\ /\ 2\}$
let *?V* = $\{x'.\ dist\ y\ x' < dist\ x\ y\ /\ 2\}$
have $*$: $d\ x\ z \leq d\ x\ y + d\ y\ z \Longrightarrow d\ y\ z = d\ z\ y \Longrightarrow \neg\ (d\ x\ y * 2 < d\ x\ z \wedge d\ z\ y * 2 < d\ x\ z)$
for *d* :: *'a* $\Rightarrow$ *'a* $\Rightarrow$ *real* **and** *x y z* :: *'a*
by *arith*
have *open ?U* $\wedge$ *open ?V* $\wedge$ $x \in$ *?U* $\wedge$ $y \in$ *?V* $\wedge$ $?U \cap ?V = \{\}$
using *dist-pos-lt*[*OF xy*] $*$[*of dist*, *OF dist-triangle dist-commute*]
using *open-ball*[*of - dist x y / 2*] **by** *auto*
then show $\exists\ U\ V.$ *open U* $\wedge$ *open V* $\wedge$ $x \in U \wedge y \in V \wedge U \cap V = \{\}$
by *blast*
qed

Every normed vector space is a metric space.

instance *real-normed-vector* $<$ *metric-space*
proof
fix *x y z* :: *'a*
show $dist\ x\ y = 0 \longleftrightarrow x = y$
by (*simp add*: *dist-norm*)
show $dist\ x\ y \leq dist\ x\ z + dist\ y\ z$
using *norm-triangle-ineq4* [*of x − z y − z*] **by** (*simp add*: *dist-norm*)
qed

107.7 Class instances for real numbers

instantiation *real* :: *real-normed-field*
begin

definition *dist-real-def*: $dist\ x\ y = |x - y|$

definition *uniformity-real-def*:
(*uniformity* :: (*real* $\times$ *real*) *filter*) = (*INF* e∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* $< e$})

definition *open-real-def*:
open (*U* :: *real set*) $\longleftrightarrow$ ($\forall\, x \in U.$ *eventually* (λ(*x'*, *y*). $x' = x \longrightarrow y \in U$) *uniformity*)

definition *real-norm-def* [*simp*]: *norm r* = |*r*|

instance
by *intro-classes* (*auto simp*: *abs-mult open-real-def dist-real-def sgn-real-def uniformity-real-def*)

end

lemma *dist-of-real* [*simp*]: *dist* (*of-real x* :: *'a*) (*of-real y*) = *dist x y*
for *a* :: *'a*::*real-normed-div-algebra*
by (*metis dist-norm norm-of-real of-real-diff real-norm-def*)

instance *real* :: *linorder-topology*
proof
show (*open* :: *real set* ⇒ *bool*) = *generate-topology* (*range lessThan* ∪ *range greaterThan*)
proof (*rule ext*, *safe*)
fix *S* :: *real set*
assume *open S*
then obtain *f* **where** ∀ *x*∈*S*. *0* < *f x* ∧ (∀ *y*. *dist y x* < *f x* ⟶ *y* ∈ *S*)
unfolding *open-dist bchoice-iff* **..**
then have ∗: (⋃ *x*∈*S*. {*x* − *f x* <..} ∩ {..< *x* + *f x*}) = *S* (**is** *?S* = *S*)
by (*fastforce simp*: *dist-real-def*)
moreover have *generate-topology* (*range lessThan* ∪ *range greaterThan*) *?S*
by (*force intro*: *generate-topology.Basis generate-topology-Union generate-topology.Int*)
ultimately show *generate-topology* (*range lessThan* ∪ *range greaterThan*) *S*
by *simp*
next
fix *S* :: *real set*
assume *generate-topology* (*range lessThan* ∪ *range greaterThan*) *S*
moreover have ⋀*a*::*real*. *open* {..<*a*}
unfolding *open-dist dist-real-def*
proof *clarify*
fix *x a* :: *real*
assume *x* < *a*
then have *0* < *a* − *x* ∧ (∀ *y*. |*y* − *x*| < *a* − *x* ⟶ *y* ∈ {..<*a*}) **by** *auto*
then show ∃ *e*>*0*. ∀ *y*. |*y* − *x*| < *e* ⟶ *y* ∈ {..<*a*} **..**
qed
moreover have ⋀*a*::*real*. *open* {*a* <..}
unfolding *open-dist dist-real-def*
proof *clarify*
fix *x a* :: *real*
assume *a* < *x*
then have *0* < *x* − *a* ∧ (∀ *y*. |*y* − *x*| < *x* − *a* ⟶ *y* ∈ {*a*<..}) **by** *auto*
then show ∃ *e*>*0*. ∀ *y*. |*y* − *x*| < *e* ⟶ *y* ∈ {*a*<..} **..**
qed
ultimately show *open S*
by *induct auto*

qed
qed

instance *real* :: *linear-continuum-topology* **..**

lemmas *open-real-greaterThan* = *open-greaterThan*[**where** *'a*=*real*]
lemmas *open-real-lessThan* = *open-lessThan*[**where** *'a*=*real*]
lemmas *open-real-greaterThanLessThan* = *open-greaterThanLessThan*[**where** *'a*=*real*]
lemmas *closed-real-atMost* = *closed-atMost*[**where** *'a*=*real*]
lemmas *closed-real-atLeast* = *closed-atLeast*[**where** *'a*=*real*]
lemmas *closed-real-atLeastAtMost* = *closed-atLeastAtMost*[**where** *'a*=*real*]

instance *real* :: *ordered-real-vector*
by *standard* (*auto intro*: *mult-left-mono mult-right-mono*)

107.8 Extra type constraints

Only allow *open* in class *topological-space*.

setup ‹*Sign.add-const-constraint*
(***const-name***‹*open*›, *SOME* ***typ***‹*'a*::*topological-space set* ⇒ *bool*›)›

Only allow *uniformity* in class *uniform-space*.

setup ‹*Sign.add-const-constraint*
(***const-name***‹*uniformity*›, *SOME* ***typ***‹(*'a*::*uniformity* × *'a*) *filter*›)›

Only allow *dist* in class *metric-space*.

setup ‹*Sign.add-const-constraint*
(***const-name***‹*dist*›, *SOME* ***typ***‹*'a*::*metric-space* ⇒ *'a* ⇒ *real*›)›

Only allow *norm* in class *real-normed-vector*.

setup ‹*Sign.add-const-constraint*
(***const-name***‹*norm*›, *SOME* ***typ***‹*'a*::*real-normed-vector* ⇒ *real*›)›

107.9 Sign function

lemma *norm-sgn*: *norm* (*sgn x*) = (*if x* = *0 then 0 else 1*)
for *x* :: *'a*::*real-normed-vector*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-zero* [*simp*]: *sgn* (*0*::*'a*::*real-normed-vector*) = *0*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-zero-iff*: *sgn x* = *0* ⟷ *x* = *0*
for *x* :: *'a*::*real-normed-vector*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-minus*: *sgn* (− *x*) = − *sgn x*
for *x* :: *'a*::*real-normed-vector*

by (*simp add*: *sgn-div-norm*)

lemma *sgn-scaleR*: *sgn* (*scaleR r x*) = *scaleR* (*sgn r*) (*sgn x*)
for *x* :: *'a*::*real-normed-vector*
by (*simp add*: *sgn-div-norm ac-simps*)

lemma *sgn-one* [*simp*]: *sgn* (*1*::*'a*::*real-normed-algebra-1*) = *1*
by (*simp add*: *sgn-div-norm*)

lemma *sgn-of-real*: *sgn* (*of-real r* :: *'a*::*real-normed-algebra-1*) = *of-real* (*sgn r*)
unfolding *of-real-def* **by** (*simp only*: *sgn-scaleR sgn-one*)

lemma *sgn-mult*: *sgn* (*x* * *y*) = *sgn x* * *sgn y*
for *x y* :: *'a*::*real-normed-div-algebra*
by (*simp add*: *sgn-div-norm norm-mult*)

hide-fact (**open**) *sgn-mult*

lemma *real-sgn-eq*: *sgn x* = *x* / |*x*|
for *x* :: *real*
by (*simp add*: *sgn-div-norm divide-inverse*)

lemma *zero-le-sgn-iff* [*simp*]: *0* ≤ *sgn x* ⟷ *0* ≤ *x*
for *x* :: *real*
by (*cases 0*::*real x rule*: *linorder-cases*) *simp-all*

lemma *sgn-le-0-iff* [*simp*]: *sgn x* ≤ *0* ⟷ *x* ≤ *0*
for *x* :: *real*
by (*cases 0*::*real x rule*: *linorder-cases*) *simp-all*

lemma *norm-conv-dist*: *norm x* = *dist x 0*
unfolding *dist-norm* **by** *simp*

declare *norm-conv-dist* [*symmetric*, *simp*]

lemma *dist-0-norm* [*simp*]: *dist 0 x* = *norm x*
for *x* :: *'a*::*real-normed-vector*
by (*simp add*: *dist-norm*)

lemma *dist-diff* [*simp*]: *dist a* (*a* − *b*) = *norm b* *dist* (*a* − *b*) *a* = *norm b*
by (*simp-all add*: *dist-norm*)

lemma *dist-of-int*: *dist* (*of-int m*) (*of-int n* :: *'a* :: *real-normed-algebra-1*) = *of-int* |*m* − *n*|
proof −
have *dist* (*of-int m*) (*of-int n* :: *'a*) = *dist* (*of-int m* :: *'a*) (*of-int m* − (*of-int* (*m* − *n*)))
by *simp*
also have ... = *of-int* |*m* − *n*| **by** (*subst dist-diff*, *subst norm-of-int*) *simp*

finally show *?thesis* **.**
qed

lemma *dist-of-nat*:
dist (*of-nat m*) (*of-nat n* :: *'a* :: *real-normed-algebra-1*) = *of-int* |*int m* − *int n*|
by (*subst* (*1 2*) *of-int-of-nat-eq* [*symmetric*]) (*rule dist-of-int*)

107.10 Bounded Linear and Bilinear Operators

lemma *linearI*: *linear f*
if ⋀*b1 b2*. *f* (*b1* + *b2*) = *f b1* + *f b2*
⋀*r b*. *f* (*r* *$_R$ *b*) = *r* *$_R$ *f b*
using *that*
by *unfold-locales* (*auto simp*: *algebra-simps*)

lemma *linear-iff*:
linear f ⟷ (∀ *x y*. *f* (*x* + *y*) = *f x* + *f y*) ∧ (∀ *c x*. *f* (*c* *$_R$ *x*) = *c* *$_R$ *f x*)
(**is** *linear f* ⟷ *?rhs*)
proof
assume *linear f*
then interpret *f*: *linear f* **.**
show *?rhs* **by** (*simp add*: *f.add f.scale*)
next
assume *?rhs*
then show *linear f* **by** (*intro linearI*) *auto*
qed

lemma *linear-of-real* [*simp*]: *linear of-real*
by (*simp add*: *linear-iff scaleR-conv-of-real*)

lemmas *linear-scaleR-left* = *linear-scale-left*
lemmas *linear-imp-scaleR* = *linear-imp-scale*

corollary *real-linearD*:
fixes *f* :: *real* ⇒ *real*
assumes *linear f* **obtains** *c* **where** *f* = (∗) *c*
by (*rule linear-imp-scaleR* [*OF assms*]) (*force simp*: *scaleR-conv-of-real*)

lemma *linear-times-of-real*: *linear* (λ*x*. *a* ∗ *of-real x*)
by (*auto intro*!: *linearI simp*: *distrib-left*)
(*metis mult-scaleR-right scaleR-conv-of-real*)

locale *bounded-linear* = *linear f* **for** *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-vector*
+
assumes *bounded*: ∃ *K*. ∀ *x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
begin

lemma *pos-bounded*: ∃ *K*>*0*. ∀ *x*. *norm* (*f x*) ≤ *norm x* ∗ *K*
proof −

```
  obtain K where K: ⋀x. norm (f x) ≤ norm x * K
    using bounded by blast
  show ?thesis
  proof (intro exI impI conjI allI)
    show 0 < max 1 K
      by (rule order-less-le-trans [OF zero-less-one max.cobounded1])
  next
    fix x
    have norm (f x) ≤ norm x * K using K .
    also have ... ≤ norm x * max 1 K
      by (rule mult-left-mono [OF max.cobounded2 norm-ge-zero])
    finally show norm (f x) ≤ norm x * max 1 K .
  qed
qed

lemma nonneg-bounded: ∃K≥0. ∀x. norm (f x) ≤ norm x * K
  using pos-bounded by (auto intro: order-less-imp-le)

lemma linear: linear f
  by (fact local.linear-axioms)

end

lemma bounded-linear-intro:
  assumes ⋀x y. f (x + y) = f x + f y
    and ⋀r x. f (scaleR r x) = scaleR r (f x)
    and ⋀x. norm (f x) ≤ norm x * K
  shows bounded-linear f
  by standard (blast intro: assms)+

locale bounded-bilinear =
 fixes prod :: 'a::real-normed-vector ⇒ 'b::real-normed-vector ⇒ 'c::real-normed-vector
    (infixl ‹**› 70)
  assumes add-left: prod (a + a') b = prod a b + prod a' b
    and add-right: prod a (b + b') = prod a b + prod a b'
    and scaleR-left: prod (scaleR r a) b = scaleR r (prod a b)
    and scaleR-right: prod a (scaleR r b) = scaleR r (prod a b)
    and bounded: ∃K. ∀a b. norm (prod a b) ≤ norm a * norm b * K
begin

lemma pos-bounded: ∃K>0. ∀a b. norm (a ** b) ≤ norm a * norm b * K
proof −
  obtain K where ⋀a b. norm (a ** b) ≤ norm a * norm b * K
    using bounded by blast
  then have norm (a ** b) ≤ norm a * norm b * (max 1 K) for a b
    by (rule order.trans) (simp add: mult-left-mono)
  then show ?thesis
    by force
qed
```

lemma *nonneg-bounded*: $\exists K{\geq}0.\ \forall a\ b.\ norm\ (a\ **\ b) \leq norm\ a * norm\ b * K$
using *pos-bounded* **by** (*auto intro*: *order-less-imp-le*)

lemma *additive-right*: *additive* ($\lambda b.$ *prod a b*)
by (*rule additive.intro*, *rule add-right*)

lemma *additive-left*: *additive* ($\lambda a.$ *prod a b*)
by (*rule additive.intro*, *rule add-left*)

lemma *zero-left*: *prod 0 b = 0*
by (*rule additive.zero* [*OF additive-left*])

lemma *zero-right*: *prod a 0 = 0*
by (*rule additive.zero* [*OF additive-right*])

lemma *minus-left*: *prod* (− *a*) *b* = − *prod a b*
by (*rule additive.minus* [*OF additive-left*])

lemma *minus-right*: *prod a* (− *b*) = − *prod a b*
by (*rule additive.minus* [*OF additive-right*])

lemma *diff-left*: *prod* (*a* − *a'*) *b* = *prod a b* − *prod a' b*
by (*rule additive.diff* [*OF additive-left*])

lemma *diff-right*: *prod a* (*b* − *b'*) = *prod a b* − *prod a b'*
by (*rule additive.diff* [*OF additive-right*])

lemma *sum-left*: *prod* (*sum g S*) *x* = *sum* (($\lambda i.$ *prod* (*g i*) *x*)) *S*
by (*rule additive.sum* [*OF additive-left*])

lemma *sum-right*: *prod x* (*sum g S*) = *sum* (($\lambda i.$ (*prod x* (*g i*)))) *S*
by (*rule additive.sum* [*OF additive-right*])

lemma *bounded-linear-left*: *bounded-linear* ($\lambda a.$ *a* ** *b*)
proof −
obtain *K* **where** $\bigwedge a\ b.$ *norm* (*a* ** *b*) ≤ *norm a* * *norm b* * *K*
using *pos-bounded* **by** *blast*
then show *?thesis*
by (*rule-tac K=norm b* * *K* **in** *bounded-linear-intro*) (*auto simp*: *algebra-simps scaleR-left add-left*)
qed

lemma *bounded-linear-right*: *bounded-linear* ($\lambda b.$ *a* ** *b*)
proof −
obtain *K* **where** $\bigwedge a\ b.$ *norm* (*a* ** *b*) ≤ *norm a* * *norm b* * *K*
using *pos-bounded* **by** *blast*
then show *?thesis*

by (*rule-tac K=norm a * K* **in** *bounded-linear-intro*) (*auto simp*: *algebra-simps scaleR-right add-right*)
qed

lemma *prod-diff-prod*: $(x ** y - a ** b) = (x - a) ** (y - b) + (x - a) ** b + a ** (y - b)$
by (*simp add*: *diff-left diff-right*)

lemma *flip*: *bounded-bilinear* $(\lambda x\ y.\ y ** x)$
proof
show $\exists K.\ \forall a\ b.\ norm\ (b ** a) \leq norm\ a * norm\ b * K$
by (*metis bounded mult.commute*)
qed (*simp-all add*: *add-right add-left scaleR-right scaleR-left*)

lemma *comp1*:
assumes *bounded-linear g*
shows *bounded-bilinear* $(\lambda x.\ (**)\ (g\ x))$
proof *unfold-locales*
interpret *g*: *bounded-linear g* **by** *fact*
show $\bigwedge a\ a'\ b.\ g\ (a + a') ** b = g\ a ** b + g\ a' ** b$
$\bigwedge a\ b\ b'.\ g\ a ** (b + b') = g\ a ** b + g\ a ** b'$
$\bigwedge r\ a\ b.\ g\ (r *_R a) ** b = r *_R (g\ a ** b)$
$\bigwedge a\ r\ b.\ g\ a ** (r *_R b) = r *_R (g\ a ** b)$
by (*auto simp*: *g.add add-left add-right g.scaleR scaleR-left scaleR-right*)
from *g.nonneg-bounded nonneg-bounded* **obtain** *K L*
where *nn*: $0 \leq K\ 0 \leq L$
and *K*: $\bigwedge x.\ norm\ (g\ x) \leq norm\ x * K$
and *L*: $\bigwedge a\ b.\ norm\ (a ** b) \leq norm\ a * norm\ b * L$
by *auto*
have $norm\ (g\ a ** b) \leq norm\ a * K * norm\ b * L$ **for** *a b*
by (*auto intro*!: *order-trans*[*OF K*] *order-trans*[*OF L*] *mult-mono simp*: *nn*)
then show $\exists K.\ \forall a\ b.\ norm\ (g\ a ** b) \leq norm\ a * norm\ b * K$
by (*auto intro*!: *exI*[**where** *x=K * L*] *simp*: *ac-simps*)
qed

lemma *comp*: *bounded-linear f* $\Longrightarrow$ *bounded-linear g* $\Longrightarrow$ *bounded-bilinear* $(\lambda x\ y.\ f\ x ** g\ y)$
by (*rule bounded-bilinear.flip*[*OF bounded-bilinear.comp1*[*OF bounded-bilinear.flip*[*OF comp1*]]])

end

lemma *bounded-linear-ident*[*simp*]: *bounded-linear* $(\lambda x.\ x)$
by *standard* (*auto intro*!: *exI*[*of - 1*])

lemma *bounded-linear-zero*[*simp*]: *bounded-linear* $(\lambda x.\ 0)$
by *standard* (*auto intro*!: *exI*[*of - 1*])

lemma *bounded-linear-add*:

assumes *bounded-linear f*
and *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ x + g\ x)$
proof −
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
show *?thesis*
proof
from *f.bounded* **obtain** *Kf* **where** *Kf*: *norm* $(f\ x) \leq$ *norm* $x *$ *Kf* **for** *x*
by *blast*
from *g.bounded* **obtain** *Kg* **where** *Kg*: *norm* $(g\ x) \leq$ *norm* $x *$ *Kg* **for** *x*
by *blast*
show $\exists K.\ \forall x.$ *norm* $(f\ x + g\ x) \leq$ *norm* $x * K$
using *add-mono*[*OF Kf Kg*]
by (*intro exI*[*of - Kf + Kg*]) (*auto simp*: *field-simps intro*: *norm-triangle-ineq order-trans*)
qed (*simp-all add*: *f.add g.add f.scaleR g.scaleR scaleR-right-distrib*)
qed

lemma *bounded-linear-minus*:
assumes *bounded-linear f*
shows *bounded-linear* $(\lambda x.\ -\ f\ x)$
proof −
interpret *f*: *bounded-linear f* **by** *fact*
show *?thesis*
by *unfold-locales* (*simp-all add*: *f.add f.scaleR f.bounded*)
qed

lemma *bounded-linear-sub*: *bounded-linear f* $\Longrightarrow$ *bounded-linear g* $\Longrightarrow$ *bounded-linear* $(\lambda x.\ f\ x - g\ x)$
using *bounded-linear-add*[*of f* $\lambda x.\ -\ g\ x$] *bounded-linear-minus*[*of g*]
by (*auto simp*: *algebra-simps*)

lemma *bounded-linear-sum*:
fixes *f* :: $'i \Rightarrow$ $'a$::*real-normed-vector* $\Rightarrow$ $'b$::*real-normed-vector*
shows $(\bigwedge i.\ i \in I \Longrightarrow$ *bounded-linear* $(f\ i)) \Longrightarrow$ *bounded-linear* $(\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
by (*induct I rule*: *infinite-finite-induct*) (*auto intro*!: *bounded-linear-add*)

lemma *bounded-linear-compose*:
assumes *bounded-linear f*
and *bounded-linear g*
shows *bounded-linear* $(\lambda x.\ f\ (g\ x))$
proof −
interpret *f*: *bounded-linear f* **by** *fact*
interpret *g*: *bounded-linear g* **by** *fact*
show *?thesis*
proof *unfold-locales*
show $f\ (g\ (x + y)) = f\ (g\ x) + f\ (g\ y)$ **for** *x y*

```
      by (simp only: f.add g.add)
    show f (g (scaleR r x)) = scaleR r (f (g x)) for r x
      by (simp only: f.scaleR g.scaleR)
    from f.pos-bounded obtain Kf where f: ⋀x. norm (f x) ≤ norm x * Kf and
Kf: 0 < Kf
      by blast
    from g.pos-bounded obtain Kg where g: ⋀x. norm (g x) ≤ norm x * Kg
      by blast
    show ∃K. ∀x. norm (f (g x)) ≤ norm x * K
    proof (intro exI allI)
      fix x
      have norm (f (g x)) ≤ norm (g x) * Kf
        using f .
      also have ... ≤ (norm x * Kg) * Kf
        using g Kf [THEN order-less-imp-le] by (rule mult-right-mono)
      also have (norm x * Kg) * Kf = norm x * (Kg * Kf)
        by (rule mult.assoc)
      finally show norm (f (g x)) ≤ norm x * (Kg * Kf) .
    qed
  qed
qed

lemma bounded-bilinear-mult: bounded-bilinear ((*) :: 'a ⇒ 'a ⇒ 'a::real-normed-algebra)
proof (rule bounded-bilinear.intro)
  show ∃K. ∀a b::'a. norm (a * b) ≤ norm a * norm b * K
    by (rule-tac x=1 in exI) (simp add: norm-mult-ineq)
qed (auto simp: algebra-simps)

lemma bounded-linear-mult-left: bounded-linear (λx::'a::real-normed-algebra. x *
y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-left)

lemma bounded-linear-mult-right: bounded-linear (λy::'a::real-normed-algebra. x *
y)
  using bounded-bilinear-mult
  by (rule bounded-bilinear.bounded-linear-right)

lemmas bounded-linear-mult-const =
  bounded-linear-mult-left [THEN bounded-linear-compose]

lemmas bounded-linear-const-mult =
  bounded-linear-mult-right [THEN bounded-linear-compose]

lemma bounded-linear-divide: bounded-linear (λx. x / y)
  for y :: 'a::real-normed-field
  unfolding divide-inverse by (rule bounded-linear-mult-left)

lemma bounded-bilinear-scaleR: bounded-bilinear scaleR
```

proof (*rule bounded-bilinear.intro*)
show $\exists K.\ \forall a\ b.\ norm\ (a *_R b) \leq norm\ a * norm\ b * K$
using *less-eq-real-def* **by** *auto*
qed (*auto simp*: *algebra-simps*)

lemma *bounded-linear-scaleR-left*: *bounded-linear* $(\lambda r.\ scaleR\ r\ x)$
using *bounded-bilinear-scaleR*
by (*rule bounded-bilinear.bounded-linear-left*)

lemma *bounded-linear-scaleR-right*: *bounded-linear* $(\lambda x.\ scaleR\ r\ x)$
using *bounded-bilinear-scaleR*
by (*rule bounded-bilinear.bounded-linear-right*)

lemmas *bounded-linear-scaleR-const* =
bounded-linear-scaleR-left[*THEN bounded-linear-compose*]

lemmas *bounded-linear-const-scaleR* =
bounded-linear-scaleR-right[*THEN bounded-linear-compose*]

lemma *bounded-linear-of-real*: *bounded-linear* $(\lambda r.\ of\text{-}real\ r)$
unfolding *of-real-def* **by** (*rule bounded-linear-scaleR-left*)

lemma *real-bounded-linear*: *bounded-linear* $f \longleftrightarrow (\exists\, c{::}real.\ f = (\lambda x.\ x * c))$
for $f :: real \Rightarrow real$
proof −
{
fix x
assume *bounded-linear f*
then interpret *bounded-linear f* **.**
from *scaleR*[*of x 1*] **have** $f\ x = x * f\ 1$
by *simp*
}
then show *?thesis*
by (*auto intro*: *exI*[*of - f 1*] *bounded-linear-mult-left*)
qed

instance *real-normed-algebra-1* $\subseteq$ *perfect-space*
proof
fix $x{::}'a$
have $\bigwedge e.\ 0 < e \Longrightarrow \exists\, y.\ norm\ (y - x) < e \wedge y \neq x$
by (*rule-tac* $x = x + of\text{-}real\ (e/2)$ **in** *exI*) *auto*
then show $\neg$ *open* $\{x\}$
by (*clarsimp simp*: *open-dist dist-norm*)
qed

107.11 Filters and Limits on Metric Space

lemma (**in** *metric-space*) *nhds-metric*: *nhds* $x = (INF\ e{\in}\{0 <..\}.\ principal\ \{y.\ dist\ y\ x < e\})$

unfolding *nhds-def*
proof (*safe intro*!: *INF-eq*)
fix *S*
assume *open S x* ∈ *S*
then obtain *e* **where** {*y*. *dist y x* < *e*} ⊆ *S* 0 < *e*
by (*auto simp*: *open-dist subset-eq*)
then show ∃ *e*∈{*0*<..}. *principal* {*y*. *dist y x* < *e*} ≤ *principal S*
by *auto*
qed (*auto intro*!: *exI*[*of* - {*y*. *dist x y* < *e*} **for** *e*] *open-ball simp*: *dist-commute*)

lemma *tendsto-iff-uniformity*:
— More general analogus of *tendsto-iff* below. Applies to all uniform spaces, not just metric ones.
fixes *l* :: ‹'*b* :: *uniform-space*›
shows ‹(*f* ⟶ *l*) *F* ⟷ (∀ *E*. *eventually E uniformity* ⟶ (∀ *F x in F*. *E* (*f x*, *l*)))›
proof (*intro iffI allI impI*)
fix *E* :: ‹('*b* × '*b*) ⇒ *bool*›
assume ‹(*f* ⟶ *l*) *F*› **and** ‹*eventually E uniformity*›
from ‹*eventually E uniformity*›
have ‹*eventually* (λ(*x*, *y*). *E* (*y*, *x*)) *uniformity*›
by (*simp add*: *uniformity-sym*)
then have ‹∀ *F* (*y*, *x*) *in uniformity*. *y* = *l* ⟶ *E* (*x*, *y*)›
using *eventually-mono* **by** *fastforce*
with ‹(*f* ⟶ *l*) *F*› **have** ‹*eventually* (λ*x*. *E* (*x* ,*l*)) (*filtermap f F*)›
by (*simp add*: *filterlim-def le-filter-def eventually-nhds-uniformity*)
then show ‹∀ *F x in F*. *E* (*f x*, *l*)›
by (*simp add*: *eventually-filtermap*)
next
assume *assm*: ‹∀ *E*. *eventually E uniformity* ⟶ (∀ *F x in F*. *E* (*f x*, *l*))›
have ‹*eventually P* (*filtermap f F*)› **if** ‹∀ *F* (*x*, *y*) *in uniformity*. *x* = *l* ⟶ *P y*›
for *P*
proof −
from *that* **have** ‹∀ *F* (*y*, *x*) *in uniformity*. *x* = *l* ⟶ *P y*›
using *uniformity-sym*[**where** *E*=‹λ(*x*,*y*). *x*=*l* ⟶ *P y*›] **by** *auto*
with *assm* **have** ‹∀ *F x in F*. *P* (*f x*)›
by *auto*
then show *?thesis*
by (*auto simp*: *eventually-filtermap*)
qed
then show ‹(*f* ⟶ *l*) *F*›
by (*simp add*: *filterlim-def le-filter-def eventually-nhds-uniformity*)
qed

lemma (**in** *metric-space*) *tendsto-iff*: (*f* ⟶ *l*) *F* ⟷ (∀ *e*>*0*. *eventually* (λ*x*. *dist* (*f x*) *l* < *e*) *F*)
unfolding *nhds-metric filterlim-INF filterlim-principal* **by** *auto*

lemma *tendsto-dist-iff*:
(($f \longrightarrow l$) F) $\longleftrightarrow$ ((($\lambda x.$ *dist* ($f\ x$) l) $\longrightarrow$ 0) F)
unfolding *tendsto-iff* **by** *simp*

lemma (**in** *metric-space*) *tendstoI* [*intro?*]:
($\bigwedge e.$ $0 < e \Longrightarrow$ *eventually* ($\lambda x.$ *dist* ($f\ x$) $l < e$) F) $\Longrightarrow$ ($f \longrightarrow l$) F
by (*auto simp*: *tendsto-iff*)

lemma (**in** *metric-space*) *tendstoD*: ($f \longrightarrow l$) $F \Longrightarrow 0 < e \Longrightarrow$ *eventually* ($\lambda x.$ *dist* ($f\ x$) $l < e$) F
by (*auto simp*: *tendsto-iff*)

lemma (**in** *metric-space*) *eventually-nhds-metric*:
eventually P (*nhds* a) $\longleftrightarrow$ ($\exists\, d{>}0.\ \forall x.$ *dist* $x\ a < d \longrightarrow P\ x$)
unfolding *nhds-metric*
by (*subst eventually-INF-base*)
(*auto simp*: *eventually-principal Bex-def subset-eq intro*: *exI*[*of* - *min a b* **for** *a b*])

lemma *eventually-at*: *eventually* P (*at* a *within* S) $\longleftrightarrow$ ($\exists\, d{>}0.\ \forall x{\in}S.\ x \neq a \wedge$ *dist* $x\ a < d \longrightarrow P\ x$)
for $a :: {'a} ::$ *metric-space*
by (*auto simp*: *eventually-at-filter eventually-nhds-metric*)

lemma *frequently-at*: *frequently* P (*at* a *within* S) $\longleftrightarrow$ ($\forall\, d{>}0.\ \exists x{\in}S.\ x \neq a \wedge$ *dist* $x\ a < d \wedge P\ x$)
for $a :: {'a} ::$ *metric-space*
unfolding *frequently-def eventually-at* **by** *auto*

lemma *eventually-at-le*: *eventually* P (*at* a *within* S) $\longleftrightarrow$ ($\exists\, d{>}0.\ \forall x{\in}S.\ x \neq a \wedge$ *dist* $x\ a \leq d \longrightarrow P\ x$)
for $a :: {'a}{::}$*metric-space*
unfolding *eventually-at-filter eventually-nhds-metric*
apply *safe*
apply (*rule-tac* $x{=}d\ /\ 2$ **in** *exI*, *auto*)
done

lemma *eventually-at-left-real*: $a > (b :: real) \Longrightarrow$ *eventually* ($\lambda x.$ $x \in \{b{<}..{<}a\}$) (*at-left* a)
by (*subst eventually-at*, *rule exI*[*of* - $a - b$]) (*force simp*: *dist-real-def*)

lemma *eventually-at-right-real*: $a < (b :: real) \Longrightarrow$ *eventually* ($\lambda x.$ $x \in \{a{<}..{<}b\}$) (*at-right* a)
by (*subst eventually-at*, *rule exI*[*of* - $b - a$]) (*force simp*: *dist-real-def*)

lemma *metric-tendsto-imp-tendsto*:
fixes $a :: {'a} ::$ *metric-space*
and $b :: {'b} ::$ *metric-space*
assumes f: ($f \longrightarrow a$) F

```
    and le: eventually (λx. dist (g x) b ≤ dist (f x) a) F
  shows (g ⟶ b) F
proof (rule tendstoI)
  fix e :: real
  assume 0 < e
  with f have eventually (λx. dist (f x) a < e) F by (rule tendstoD)
  with le show eventually (λx. dist (g x) b < e) F
    using le-less-trans by (rule eventually-elim2)
qed

lemma filterlim-real-sequentially: LIM x sequentially. real x :> at-top
proof (clarsimp simp: filterlim-at-top)
  fix Z
  show ∀F x in sequentially. Z ≤ real x
    by (meson eventually-sequentiallyI nat-ceiling-le-eq)
qed

lemma filterlim-nat-sequentially: filterlim nat sequentially at-top
proof −
  have ∀F x in at-top. Z ≤ nat x for Z
    by (auto intro!: eventually-at-top-linorderI[where c=int Z])
  then show ?thesis
    unfolding filterlim-at-top ..
qed

lemma filterlim-floor-sequentially: filterlim floor at-top at-top
proof −
  have ∀F x in at-top. Z ≤ ⌊x⌋ for Z
   by (auto simp: le-floor-iff intro!: eventually-at-top-linorderI[where c=of-int Z])
  then show ?thesis
    unfolding filterlim-at-top ..
qed

lemma filterlim-sequentially-iff-filterlim-real:
  filterlim f sequentially F ⟷ filterlim (λx. real (f x)) at-top F (is ?lhs = ?rhs)
proof
  assume ?lhs then show ?rhs
    using filterlim-compose filterlim-real-sequentially by blast
next
  assume R: ?rhs
  show ?lhs
  proof −
    have filterlim (λx. nat (floor (real (f x)))) sequentially F
      by (intro filterlim-compose[OF filterlim-nat-sequentially]
          filterlim-compose[OF filterlim-floor-sequentially] R)
    then show ?thesis by simp
  qed
qed
```

107.11.1 Limits of Sequences

lemma *lim-sequentially*: $X \longrightarrow L \longleftrightarrow (\forall r{>}0.\ \exists no.\ \forall n{\geq}no.\ dist\ (X\ n)\ L < r)$
for L :: *'a*::*metric-space*
unfolding *tendsto-iff eventually-sequentially* **..**

lemmas *LIMSEQ-def* = *lim-sequentially*

lemma *LIMSEQ-iff-nz*: $X \longrightarrow L \longleftrightarrow (\forall r{>}0.\ \exists no{>}0.\ \forall n{\geq}no.\ dist\ (X\ n)\ L < r)$
for L :: *'a*::*metric-space*
unfolding *lim-sequentially* **by** (*metis Suc-leD zero-less-Suc*)

lemma *metric-LIMSEQ-I*: $(\bigwedge r.\ 0 < r \Longrightarrow \exists no.\ \forall n{\geq}no.\ dist\ (X\ n)\ L < r) \Longrightarrow X \longrightarrow L$
for L :: *'a*::*metric-space*
by (*simp add*: *lim-sequentially*)

lemma *metric-LIMSEQ-D*: $X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists no.\ \forall n{\geq}no.\ dist\ (X\ n)\ L < r$
for L :: *'a*::*metric-space*
by (*simp add*: *lim-sequentially*)

lemma *LIMSEQ-norm-0*:
assumes $\bigwedge n$::*nat*. *norm* $(f\ n) < 1\ /\ real\ (Suc\ n)$
shows $f \longrightarrow 0$
proof (*rule metric-LIMSEQ-I*)
fix ε :: *real*
assume $\varepsilon > 0$
then obtain N::*nat* **where** $\varepsilon > inverse\ N\ N > 0$
by (*metis neq0-conv real-arch-inverse*)
then have *norm* $(f\ n) < \varepsilon$ **if** $n \geq N$ **for** n
proof −
have $1\ /\ (Suc\ n) \leq 1\ /\ N$
using ‹$0 < N$› *inverse-of-nat-le le-SucI that* **by** *blast*
also have $\ldots < \varepsilon$
by (*metis* (*no-types*) ‹*inverse* (*real N*) $< \varepsilon$› *inverse-eq-divide*)
finally show *?thesis*
by (*meson assms less-eq-real-def not-le order-trans*)
qed
then show $\exists no.\ \forall n{\geq}no.\ dist\ (f\ n)\ 0 < \varepsilon$
by *auto*
qed

107.11.2 Limits of Functions

lemma *LIM-def*: $f\ {-}a{\rightarrow}\ L \longleftrightarrow (\forall r > 0.\ \exists s > 0.\ \forall x.\ x \neq a \wedge dist\ x\ a < s \longrightarrow dist\ (f\ x)\ L < r)$
for a :: *'a*::*metric-space* **and** L :: *'b*::*metric-space*

unfolding *tendsto-iff eventually-at* **by** *simp*

lemma *metric-LIM-I*:
(⋀*r*. 0 < *r* ⟹ ∃ *s*>0. ∀ *x*. *x* ≠ *a* ∧ *dist x a* < *s* ⟶ *dist* (*f x*) *L* < *r*) ⟹ *f* −*a*→ *L*
for *a* :: 'a::*metric-space* **and** *L* :: 'b::*metric-space*
by (*simp add*: *LIM-def*)

lemma *metric-LIM-D*: *f* −*a*→ *L* ⟹ 0 < *r* ⟹ ∃ *s*>0. ∀ *x*. *x* ≠ *a* ∧ *dist x a* < *s* ⟶ *dist* (*f x*) *L* < *r*
for *a* :: 'a::*metric-space* **and** *L* :: 'b::*metric-space*
by (*simp add*: *LIM-def*)

lemma *metric-LIM-imp-LIM*:
fixes *l* :: 'a::*metric-space*
and *m* :: 'b::*metric-space*
assumes *f*: *f* −*a*→ *l*
and *le*: ⋀*x*. *x* ≠ *a* ⟹ *dist* (*g x*) *m* ≤ *dist* (*f x*) *l*
shows *g* −*a*→ *m*
by (*rule metric-tendsto-imp-tendsto* [*OF f*]) (*auto simp*: *eventually-at-topological le*)

lemma *metric-LIM-equal2*:
fixes *a* :: 'a::*metric-space*
assumes *g* −*a*→ *l* 0 < *R*
and ⋀*x*. *x* ≠ *a* ⟹ *dist x a* < *R* ⟹ *f x* = *g x*
shows *f* −*a*→ *l*
proof −
have ⋀*S*. ⟦*open S*; *l* ∈ *S*; ∀ *F x in at a*. *g x* ∈ *S*⟧ ⟹ ∀ *F x in at a*. *f x* ∈ *S*
apply (*simp add*: *eventually-at*)
by (*metis assms*(*2*) *assms*(*3*) *dual-order.strict-trans linorder-neqE-linordered-idom*)
then show *?thesis*
using *assms* **by** (*simp add*: *tendsto-def*)
qed

lemma *metric-LIM-compose2*:
fixes *a* :: 'a::*metric-space*
assumes *f*: *f* −*a*→ *b*
and *g*: *g* −*b*→ *c*
and *inj*: ∃ *d*>0. ∀ *x*. *x* ≠ *a* ∧ *dist x a* < *d* ⟶ *f x* ≠ *b*
shows (λ*x*. *g* (*f x*)) −*a*→ *c*
using *inj* **by** (*intro tendsto-compose-eventually*[*OF g f*]) (*auto simp*: *eventually-at*)

lemma *metric-isCont-LIM-compose2*:
fixes *f* :: 'a :: *metric-space* ⇒ -
assumes *f* [*unfolded isCont-def*]: *isCont f a*
and *g*: *g* −*f a*→ *l*
and *inj*: ∃ *d*>0. ∀ *x*. *x* ≠ *a* ∧ *dist x a* < *d* ⟶ *f x* ≠ *f a*

shows $(\lambda x.\ g\ (f\ x)) \longrightarrow a \rightarrow l$
by (*rule metric-LIM-compose2* [*OF f g inj*])

107.12 Complete metric spaces

107.13 Cauchy sequences

lemma (**in** *metric-space*) *Cauchy-def*: $\mathit{Cauchy}\ X = (\forall e{>}0.\ \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ \mathit{dist}\ (X\ m)\ (X\ n) < e)$
proof −
have ∗: $\mathit{eventually}\ P\ (\mathit{INF}\ M.\ \mathit{principal}\ \{(X\ m,\ X\ n) \mid n\ m.\ m \geq M \wedge n \geq M\}) \longleftrightarrow$
$(\exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ P\ (X\ m,\ X\ n))$ **for** P
apply (*subst eventually-INF-base*)
subgoal by *simp*
subgoal for *a b*
by (*intro bexI*[*of - max a b*]) (*auto simp*: *eventually-principal subset-eq*)
subgoal by (*auto simp*: *eventually-principal*, *blast*)
done
have $\mathit{Cauchy}\ X \longleftrightarrow (\mathit{INF}\ M.\ \mathit{principal}\ \{(X\ m,\ X\ n) \mid n\ m.\ m \geq M \wedge n \geq M\}) \leq \mathit{uniformity}$
unfolding *Cauchy-uniform-iff le-filter-def* ∗ **..**
also have ... $= (\forall e{>}0.\ \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ \mathit{dist}\ (X\ m)\ (X\ n) < e)$
unfolding *uniformity-dist le-INF-iff* **by** (*auto simp*: ∗ *le-principal*)
finally show *?thesis* **.**
qed

lemma (**in** *metric-space*) *Cauchy-altdef*: $\mathit{Cauchy}\ f \longleftrightarrow (\forall e{>}0.\ \exists M.\ \forall m{\geq}M.\ \forall n{>}m.\ \mathit{dist}\ (f\ m)\ (f\ n) < e)$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?rhs*
show *?lhs*
unfolding *Cauchy-def*
proof (*intro allI impI*)
fix e :: *real* **assume** e: $e > 0$
with ‹*?rhs*› **obtain** M **where** M: $m \geq M \Longrightarrow n > m \Longrightarrow \mathit{dist}\ (f\ m)\ (f\ n) < e$ **for** $m\ n$
by *blast*
have $\mathit{dist}\ (f\ m)\ (f\ n) < e$ **if** $m \geq M\ n \geq M$ **for** $m\ n$
using M[*of m n*] M[*of n m*] *e that* **by** (*cases m n rule*: *linorder-cases*) (*auto simp*: *dist-commute*)
then show $\exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ \mathit{dist}\ (f\ m)\ (f\ n) < e$
by *blast*
qed
next
assume *?lhs*
show *?rhs*
proof (*intro allI impI*)
fix e :: *real*

assume *e*: $e > 0$
with ‹*Cauchy f*› **obtain** *M* **where** $\bigwedge m\ n.\ m \geq M \Longrightarrow n \geq M \Longrightarrow dist\ (f\ m)\ (f\ n) < e$
unfolding *Cauchy-def* **by** *blast*
then show $\exists M.\ \forall m{\geq}M.\ \forall n{>}m.\ dist\ (f\ m)\ (f\ n) < e$
by (*intro exI*[*of* - *M*]) *force*
qed
qed

lemma (**in** *metric-space*) *Cauchy-altdef2*: *Cauchy s* $\longleftrightarrow$ $(\forall e{>}0.\ \exists N{::}nat.\ \forall n{\geq}N.\ dist(s\ n)(s\ N) < e)$ (**is** *?lhs* = *?rhs*)
proof
assume *Cauchy s*
then show *?rhs* **by** (*force simp*: *Cauchy-def*)
next
assume *?rhs*
{
fix *e*::*real*
assume *e*>*0*
with ‹*?rhs*› **obtain** *N* **where** *N*: $\forall n{\geq}N.\ dist\ (s\ n)\ (s\ N) < e/2$
by (*erule-tac x*=*e*/*2* **in** *allE*) *auto*
{
fix *n m*
assume *nm*: $N \leq m \wedge N \leq n$
then have *dist* (*s m*) (*s n*) < *e* **using** *N*
using *dist-triangle-half-l*[*of s m s N e s n*]
by *blast*
}
then have $\exists N.\ \forall m\ n.\ N \leq m \wedge N \leq n \longrightarrow dist\ (s\ m)\ (s\ n) < e$
by *blast*
}
then have *?lhs*
unfolding *Cauchy-def* **by** *blast*
then show *?lhs*
by *blast*
qed

lemma (**in** *metric-space*) *metric-CauchyI*:
$(\bigwedge e.\ 0 < e \Longrightarrow \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ dist\ (X\ m)\ (X\ n) < e) \Longrightarrow Cauchy\ X$
by (*simp add*: *Cauchy-def*)

lemma (**in** *metric-space*) *CauchyI'*:
$(\bigwedge e.\ 0 < e \Longrightarrow \exists M.\ \forall m{\geq}M.\ \forall n{>}m.\ dist\ (X\ m)\ (X\ n) < e) \Longrightarrow Cauchy\ X$
unfolding *Cauchy-altdef* **by** *blast*

lemma (**in** *metric-space*) *metric-CauchyD*:
$Cauchy\ X \Longrightarrow 0 < e \Longrightarrow \exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ dist\ (X\ m)\ (X\ n) < e$
by (*simp add*: *Cauchy-def*)

lemma (**in** *metric-space*) *metric-Cauchy-iff2*:
Cauchy X = (∀*j*. (∃*M*. ∀*m* ≥ *M*. ∀*n* ≥ *M*. *dist* (*X m*) (*X n*) < *inverse*(*real* (*Suc j*))))
apply (*auto simp add*: *Cauchy-def*)
by (*metis less-trans of-nat-Suc reals-Archimedean*)

lemma *Cauchy-iff2*: *Cauchy X* ⟷ (∀*j*. (∃*M*. ∀*m* ≥ *M*. ∀*n* ≥ *M*. |*X m* − *X n*| < *inverse* (*real* (*Suc j*))))
by (*simp only*: *metric-Cauchy-iff2 dist-real-def*)

lemma *lim-1-over-n* [*tendsto-intros*]: ((λ*n*. *1* / *of-nat n*) ⟶ (*0*::'*a*::*real-normed-field*)) *sequentially*
proof (*subst lim-sequentially*, *intro allI impI exI*)
fix *e*::*real* **and** *n*
assume *e*: *e* > *0*
have *inverse e* < *of-nat* (*nat* ⌈*inverse e* + *1*⌉) **by** *linarith*
also assume *n* ≥ *nat* ⌈*inverse e* + *1*⌉
finally show *dist* (*1* / *of-nat n* :: '*a*) *0* < *e*
using *e* **by** (*simp add*: *field-split-simps norm-divide*)
qed

lemma (**in** *metric-space*) *complete-def*:
shows *complete S* = (∀*f*. (∀*n*. *f n* ∈ *S*) ∧ *Cauchy f* ⟶ (∃*l*∈*S*. *f* ⟶⟶ *l*))
unfolding *complete-uniform*
proof *safe*
fix *f* :: *nat* ⇒ '*a*
assume *f*: ∀*n*. *f n* ∈ *S Cauchy f*
and ∗: ∀*F*≤*principal S*. *F* ≠ *bot* ⟶ *cauchy-filter F* ⟶ (∃*x*∈*S*. *F* ≤ *nhds x*)
then show ∃*l*∈*S*. *f* ⟶⟶ *l*
unfolding *filterlim-def* **using** *f*
by (*intro* ∗[*rule-format*])
(*auto simp*: *filtermap-sequentually-ne-bot le-principal eventually-filtermap Cauchy-uniform*)
next
fix *F* :: '*a filter*
assume *F* ≤ *principal S F* ≠ *bot cauchy-filter F*
assume *seq*: ∀*f*. (∀*n*. *f n* ∈ *S*) ∧ *Cauchy f* ⟶ (∃*l*∈*S*. *f* ⟶⟶ *l*)

from ‹*F* ≤ *principal S*› ‹*cauchy-filter F*›
have *FF-le*: *F* ×$_F$ *F* ≤ *uniformity-on S*
by (*simp add*: *cauchy-filter-def principal-prod-principal*[*symmetric*] *prod-filter-mono*)

let *?P* = λ*P e*. *eventually P F* ∧ (∀*x*. *P x* ⟶ *x* ∈ *S*) ∧ (∀*x y*. *P x* ⟶ *P y* ⟶ *dist x y* < *e*)
have *P*: ∃*P*. *?P P ε* **if** *0* < *ε* **for** *ε* :: *real*
proof −
from *that* **have** *eventually* (λ(*x*, *y*). *x* ∈ *S* ∧ *y* ∈ *S* ∧ *dist x y* < *ε*) (*uniformity-on S*)
by (*auto simp*: *eventually-inf-principal eventually-uniformity-metric*)

```
  from filter-leD[OF FF-le this] show ?thesis
    by (auto simp: eventually-prod-same)
qed

have ∃P. ∀n. ?P (P n) (1 / Suc n) ∧ P (Suc n) ≤ P n
proof (rule dependent-nat-choice)
  show ∃P. ?P P (1 / Suc 0)
    using P[of 1] by auto
next
  fix P n assume ?P P (1/Suc n)
  moreover obtain Q where ?P Q (1 / Suc (Suc n))
    using P[of 1/Suc (Suc n)] by auto
  ultimately show ∃Q. ?P Q (1 / Suc (Suc n)) ∧ Q ≤ P
    by (intro exI[of - λx. P x ∧ Q x]) (auto simp: eventually-conj-iff)
qed
then obtain P where P: eventually (P n) F P n x ⟹ x ∈ S
  P n x ⟹ P n y ⟹ dist x y < 1 / Suc n P (Suc n) ≤ P n
  for n x y
  by metis
have antimono P
  using P(4) by (rule decseq-SucI)

obtain X where X: P n (X n) for n
  using P(1)[THEN eventually-happens'[OF ‹F ≠ bot›]] by metis
have Cauchy X
  unfolding metric-Cauchy-iff2 inverse-eq-divide
proof (intro exI allI impI)
  fix j m n :: nat
  assume j ≤ m j ≤ n
  with ‹antimono P› X have P j (X m) P j (X n)
    by (auto simp: antimono-def)
  then show dist (X m) (X n) < 1 / Suc j
    by (rule P)
qed
moreover have ∀n. X n ∈ S
  using P(2) X by auto
ultimately obtain x where X ⟶ x x ∈ S
  using seq by blast

show ∃x∈S. F ≤ nhds x
proof (rule bexI)
  have eventually (λy. dist y x < e) F if 0 < e for e :: real
  proof −
    from that have (λn. 1 / Suc n :: real) ⟶ 0 ∧ 0 < e / 2
      by (subst filterlim-sequentially-Suc) (auto intro!: lim-1-over-n)
    then have ∀F n in sequentially. dist (X n) x < e / 2 ∧ 1 / Suc n < e / 2
      using ‹X ⟶ x›
      unfolding tendsto-iff order-tendsto-iff[where 'a=real] eventually-conj-iff
      by blast
```

```
    then obtain n where dist x (X n) < e / 2 1 / Suc n < e / 2
      by (auto simp: eventually-sequentially dist-commute)
    show ?thesis
      using ‹eventually (P n) F›
    proof eventually-elim
      case (elim y)
      then have dist y (X n) < 1 / Suc n
        by (intro X P)
      also have ... < e / 2 by fact
      finally show dist y x < e
        by (rule dist-triangle-half-l) fact
    qed
  qed
  then show F ≤ nhds x
    unfolding nhds-metric le-INF-iff le-principal by auto
qed fact
qed
```

apparently unused

```
lemma (in metric-space) totally-bounded-metric:
  totally-bounded S ⟷ (∀ e>0. ∃ k. finite k ∧ S ⊆ (⋃ x∈k. {y. dist x y < e}))
  unfolding totally-bounded-def eventually-uniformity-metric imp-ex
  apply (subst all-comm)
  apply (intro arg-cong[where f=All] ext, safe)
  subgoal for e
    apply (erule allE[of - λ(x, y). dist x y < e])
    apply auto
    done
  subgoal for e P k
    apply (intro exI[of - k])
    apply (force simp: subset-eq)
    done
  done
```

```
setup ‹Sign.add-const-constraint (const-name‹dist›, SOME typ‹'a::dist ⇒ 'a
⇒ real›)›
```

```
lemma cauchy-filter-metric:
  fixes F :: 'a::{uniformity-dist,uniform-space} filter
  shows cauchy-filter F ⟷ (∀ e. e>0 ⟶ (∃ P. eventually P F ∧ (∀ x y. P x ∧
P y ⟶ dist x y < e)))
proof (unfold cauchy-filter-def le-filter-def, auto)
  assume assm: ‹∀ e>0. ∃ P. eventually P F ∧ (∀ x y. P x ∧ P y ⟶ dist x y <
e)›
  then show ‹eventually P uniformity ⟹ eventually P (F ×F F)› for P
    apply (auto simp: eventually-uniformity-metric)
    using eventually-prod-same by blast
```

next
fix *e* :: *real*
assume ‹*e* > *0*›
assume *asm*: ‹∀ *P*. *eventually P uniformity* ⟶ *eventually P* (*F* ×$_F$ *F*)›

define *P* **where** ‹*P* ≡ λ(*x*,*y* :: 'a). *dist x y* < *e*›
with *asm* ‹*e* > *0*› **have** ‹*eventually P* (*F* ×$_F$ *F*)›
by (*metis case-prod-conv eventually-uniformity-metric*)
then
show ‹∃ *P*. *eventually P F* ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist x y* < *e*)›
by (*auto simp add*: *eventually-prod-same P-def*)
qed

lemma *cauchy-filter-metric-filtermap*:
fixes *f* :: 'a ⇒ 'b::{*uniformity-dist*,*uniform-space*}
shows *cauchy-filter* (*filtermap f F*) ⟷ (∀ *e*. *e*>*0* ⟶ (∃ *P*. *eventually P F* ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist* (*f x*) (*f y*) < *e*)))
proof (*subst cauchy-filter-metric*, *intro iffI allI impI*)
assume ‹∀ *e*>*0*. ∃ *P*. *eventually P* (*filtermap f F*) ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist x y* < *e*)›
then show ‹*e*>*0* ⟹ ∃ *P*. *eventually P F* ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist* (*f x*) (*f y*) < *e*)› **for** *e*
unfolding *eventually-filtermap* **by** *blast*
next
assume *asm*: ‹∀ *e*>*0*. ∃ *P*. *eventually P F* ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist* (*f x*) (*f y*) < *e*)›
fix *e*::*real* **assume** ‹*e* > *0*›
then obtain *P* **where** ‹*eventually P F*› **and** *PPe*: ‹*P x* ∧ *P y* ⟶ *dist* (*f x*) (*f y*) < *e*› **for** *x y*
using *asm* **by** *blast*

show ‹∃ *P*. *eventually P* (*filtermap f F*) ∧ (∀ *x y*. *P x* ∧ *P y* ⟶ *dist x y* < *e*)›
apply (*rule exI*[*of* - ‹λ*x*. ∃ *y*. *P y* ∧ *x* = *f y*›])
using *PPe* ‹*eventually P F*› **apply** (*auto simp*: *eventually-filtermap*)
by (*smt* (*verit*, *ccfv-SIG*) *eventually-elim2*)
qed

setup ‹*Sign.add-const-constraint* (***const-name*** ‹*dist*›, *SOME* ***typ*** ‹'a::*metric-space* ⇒ 'a ⇒ *real*›)›

107.13.1 Cauchy Sequences are Convergent

class *complete-space* = *metric-space* +
assumes *Cauchy-convergent*: *Cauchy X* ⟹ *convergent X*

lemma *Cauchy-convergent-iff*: *Cauchy X* ⟷ *convergent X*
for *X* :: *nat* ⇒ 'a::*complete-space*
by (*blast intro*: *Cauchy-convergent convergent-Cauchy*)

To prove that a Cauchy sequence converges, it suffices to show that a subsequence converges.

lemma *Cauchy-converges-subseq*:
fixes *u*::*nat* ⇒ *'a*::*metric-space*
assumes *Cauchy u*
strict-mono r
(*u* ∘ *r*) ⟶ *l*
shows *u* ⟶ *l*
proof −
have ∗: *eventually* (λ*n*. *dist* (*u n*) *l* < *e*) *sequentially* **if** *e* > *0* **for** *e*
proof −
have *e*/*2* > *0* **using** *that* **by** *auto*
then obtain *N1* **where** *N1*: ⋀*m n*. *m* ≥ *N1* ⟹ *n* ≥ *N1* ⟹ *dist* (*u m*) (*u n*) < *e*/*2*
using ‹*Cauchy u*› **unfolding** *Cauchy-def* **by** *blast*
obtain *N2* **where** *N2*: ⋀*n*. *n* ≥ *N2* ⟹ *dist* ((*u* ∘ *r*) *n*) *l* < *e* / *2*
using *order-tendstoD*(*2*)[*OF iffD1*[*OF tendsto-dist-iff* ‹(*u* ∘ *r*) ⟶ *l*›] ‹*e*/*2* > *0*›]
unfolding *eventually-sequentially* **by** *auto*
have *dist* (*u n*) *l* < *e* **if** *n* ≥ *max N1 N2* **for** *n*
proof −
have *dist* (*u n*) *l* ≤ *dist* (*u n*) ((*u* ∘ *r*) *n*) + *dist* ((*u* ∘ *r*) *n*) *l*
by (*rule dist-triangle*)
also have ... < *e*/*2* + *e*/*2*
proof (*intro add-strict-mono*)
show *dist* (*u n*) ((*u* ∘ *r*) *n*) < *e* / *2*
using *N1*[*of n r n*] *N2*[*of n*] *that* **unfolding** *comp-def*
by (*meson assms*(*2*) *le-trans max.bounded-iff strict-mono-imp-increasing*)
show *dist* ((*u* ∘ *r*) *n*) *l* < *e* / *2*
using *N2 that* **by** *auto*
qed
finally show *?thesis* **by** *simp*
qed
then show *?thesis* **unfolding** *eventually-sequentially* **by** *blast*
qed
have (λ*n*. *dist* (*u n*) *l*) ⟶ *0*
by (*simp add*: *less-le-trans* ∗ *order-tendstoI*)
then show *?thesis* **using** *tendsto-dist-iff* **by** *auto*
qed

107.14 The set of real numbers is a complete metric space

Proof that Cauchy sequences converge based on the one from http://pirate.shu.edu/~wachsmut/ira/numseq/proofs/cauconv.html

If sequence X is Cauchy, then its limit is the lub of $\{r.\ \exists N.\ \forall n{\geq}N.\ r < X\ n\}$

lemma *increasing-LIMSEQ*:

```
  fixes f :: nat ⇒ real
  assumes inc: ⋀n. f n ≤ f (Suc n)
    and bdd: ⋀n. f n ≤ l
    and en: ⋀e. 0 < e ⟹ ∃n. l ≤ f n + e
  shows f ⟶ l
proof (rule increasing-tendsto)
  fix x
  assume x < l
  with dense[of 0 l − x] obtain e where 0 < e e < l − x
    by auto
  from en[OF ‹0 < e›] obtain n where l − e ≤ f n
    by (auto simp: field-simps)
  with ‹e < l − x› ‹0 < e› have x < f n
    by simp
  with incseq-SucI[of f, OF inc] show eventually (λn. x < f n) sequentially
    by (auto simp: eventually-sequentially incseq-def intro: less-le-trans)
qed (use bdd in auto)

lemma real-Cauchy-convergent:
  fixes X :: nat ⇒ real
  assumes X: Cauchy X
  shows convergent X
proof −
  define S :: real set where S = {x. ∃N. ∀n≥N. x < X n}
  then have mem-S: ⋀N x. ∀n≥N. x < X n ⟹ x ∈ S
    by auto

  have bound-isUb: y ≤ x if N: ∀n≥N. X n < x and y ∈ S for N and x y :: real
  proof −
    from that have ∃M. ∀n≥M. y < X n
      by (simp add: S-def)
    then obtain M where ∀n≥M. y < X n ..
    then have y < X (max M N) by simp
    also have ... < x using N by simp
    finally show ?thesis by (rule order-less-imp-le)
  qed

  obtain N where ∀m≥N. ∀n≥N. dist (X m) (X n) < 1
    using X[THEN metric-CauchyD, OF zero-less-one] by auto
  then have N: ∀n≥N. dist (X n) (X N) < 1 by simp
  have [simp]: S ≠ {}
  proof (intro exI ex-in-conv[THEN iffD1])
    from N have ∀n≥N. X N − 1 < X n
      by (simp add: abs-diff-less-iff dist-real-def)
    then show X N − 1 ∈ S by (rule mem-S)
  qed
  have [simp]: bdd-above S
  proof
    from N have ∀n≥N. X n < X N + 1
```

```
    by (simp add: abs-diff-less-iff dist-real-def)
  then show ⋀s. s ∈ S ⟹ s ≤ X N + 1
    by (rule bound-isUb)
 qed
 have X ⟶ Sup S
 proof (rule metric-LIMSEQ-I)
  fix r :: real
  assume 0 < r
  then have r: 0 < r/2 by simp
  obtain N where ∀n≥N. ∀m≥N. dist (X n) (X m) < r/2
   using metric-CauchyD [OF X r] by auto
  then have ∀n≥N. dist (X n) (X N) < r/2 by simp
  then have N: ∀n≥N. X N − r/2 < X n ∧ X n < X N + r/2
   by (simp only: dist-real-def abs-diff-less-iff)

  from N have ∀n≥N. X N − r/2 < X n by blast
  then have X N − r/2 ∈ S by (rule mem-S)
  then have 1: X N − r/2 ≤ Sup S by (simp add: cSup-upper)

  from N have ∀n≥N. X n < X N + r/2 by blast
  from bound-isUb[OF this]
  have 2: Sup S ≤ X N + r/2
   by (intro cSup-least) simp-all

  show ∃N. ∀n≥N. dist (X n) (Sup S) < r
  proof (intro exI allI impI)
   fix n
   assume n: N ≤ n
   from N n have X n < X N + r/2 and X N − r/2 < X n
    by simp-all
   then show dist (X n) (Sup S) < r using 1 2
    by (simp add: abs-diff-less-iff dist-real-def)
  qed
 qed
 then show ?thesis by (auto simp: convergent-def)
qed

instance real :: complete-space
 by intro-classes (rule real-Cauchy-convergent)

class banach = real-normed-vector + complete-space

instance real :: banach ..

lemma tendsto-at-topI-sequentially:
 fixes f :: real ⇒ 'b::first-countable-topology
 assumes *: ⋀X. filterlim X at-top sequentially ⟹ (λn. f (X n)) ⟶ y
 shows (f ⟶ y) at-top
proof −
```

obtain A **where** A: *decseq A open* $(A\ n)$ $y \in A\ n$ *nhds* $y = (INF\ n.\ principal\ (A\ n))$ **for** n
by (*rule nhds-countable*[*of y*]) (*rule that*)

have $\forall m.\ \exists k.\ \forall x{\geq}k.\ f\ x \in A\ m$
proof (*rule ccontr*)
assume $\neg\ (\forall m.\ \exists k.\ \forall x{\geq}k.\ f\ x \in A\ m)$
then obtain m **where** $\bigwedge k.\ \exists x{\geq}k.\ f\ x \notin A\ m$
by *auto*
then have $\exists X.\ \forall n.\ (f\ (X\ n) \notin A\ m) \wedge max\ n\ (X\ n) + 1 \leq X\ (Suc\ n)$
by (*intro dependent-nat-choice*) (*auto simp del*: *max.bounded-iff*)
then obtain X **where** X: $\bigwedge n.\ f\ (X\ n) \notin A\ m\ \bigwedge n.\ max\ n\ (X\ n) + 1 \leq X\ (Suc\ n)$
by *auto*
have $1 \leq n \Longrightarrow real\ n \leq X\ n$ **for** n
using X[*of n − 1*] **by** *auto*
then have *filterlim X at-top sequentially*
by (*force intro*!: *filterlim-at-top-mono*[*OF filterlim-real-sequentially*]
simp: *eventually-sequentially*)
from *topological-tendstoD*[*OF* ∗[*OF this*] *A*(*2*, *3*), *of m*] *X*(*1*) **show** *False*
by *auto*
qed
then obtain k **where** $k\ m \leq x \Longrightarrow f\ x \in A\ m$ **for** $m\ x$
by *metis*
then show *?thesis*
unfolding *at-top-def A* **by** (*intro filterlim-base*[**where** *i*=*k*]) *auto*
qed

lemma *tendsto-at-topI-sequentially-real*:
fixes f :: *real* ⇒ *real*
assumes *mono*: *mono f*
and *limseq*: $(\lambda n.\ f\ (real\ n)) \longrightarrow y$
shows $(f \longrightarrow y)$ *at-top*
proof (*rule tendstoI*)
fix e :: *real*
assume $0 < e$
with *limseq* **obtain** N :: *nat* **where** N: $N \leq n \Longrightarrow |f\ (real\ n) - y| < e$ **for** n
by (*auto simp*: *lim-sequentially dist-real-def*)
have *le*: $f\ x \leq y$ **for** x :: *real*
proof −
obtain n **where** $x \leq real\text{-}of\text{-}nat\ n$
using *real-arch-simple*[*of x*] **..**
note *monoD*[*OF mono this*]
also have $f\ (real\text{-}of\text{-}nat\ n) \leq y$
by (*rule LIMSEQ-le-const*[*OF limseq*]) (*auto intro*!: *exI*[*of* - *n*] *monoD*[*OF mono*])
finally show *?thesis* **.**
qed
have *eventually* $(\lambda x.\ real\ N \leq x)$ *at-top*

```
    by (rule eventually-ge-at-top)
  then show eventually (λx. dist (f x) y < e) at-top
  proof eventually-elim
    case (elim x)
    with N[of N] le have y − f (real N) < e by auto
    moreover note monoD[OF mono elim]
    ultimately show dist (f x) y < e
      using le[of x] by (auto simp: dist-real-def field-simps)
  qed
qed

end
```

108 Limits on Real Vector Spaces

```
theory Limits
  imports Real-Vector-Spaces
begin

lemma range-mult [simp]:
  fixes a::real shows range ((∗) a) = (if a=0 then {0} else UNIV)
  by (simp add: surj-def) (meson dvdE dvd-field-iff)
```

108.1 Filter going to infinity norm

```
definition at-infinity :: 'a::real-normed-vector filter
  where at-infinity = (INF r. principal {x. r ≤ norm x})

lemma eventually-at-infinity: eventually P at-infinity ⟷ (∃ b. ∀ x. b ≤ norm x
⟶ P x)
  unfolding at-infinity-def
  by (subst eventually-INF-base)
     (auto simp: subset-eq eventually-principal intro!: exI[of - max a b for a b])

lemma eventually-at-infinityI:
  fixes P::'a::real-normed-vector ⇒ bool
  assumes ⋀x. c ≤ norm x ⟹ P x
  shows eventually P at-infinity
unfolding eventually-at-infinity using assms by auto

corollary eventually-at-infinity-pos:
  eventually p at-infinity ⟷ (∃ b. 0 < b ∧ (∀ x. norm x ≥ b ⟶ p x))
  unfolding eventually-at-infinity
  by (meson le-less-trans norm-ge-zero not-le zero-less-one)

lemma at-infinity-eq-at-top-bot: (at-infinity :: real filter) = sup at-top at-bot
proof −
  have 1: ⟦∀ n≥u. A n; ∀ n≤v. A n⟧
       ⟹ ∃ b. ∀ x. b ≤ |x| ⟶ A x for A and u v::real
```

by (*rule-tac x=max* (− *v*) *u* **in** *exI*) (*auto simp*: *abs-real-def*)
have *2*: ∀ *x*. *u* ≤ |*x*| ⟶ *A x* ⟹ ∃ *N*. ∀ *n*≥*N*. *A n* **for** *A* **and** *u*::*real*
by (*meson abs-less-iff le-cases less-le-not-le*)
have *3*: ∀ *x*. *u* ≤ |*x*| ⟶ *A x* ⟹ ∃ *N*. ∀ *n*≤*N*. *A n* **for** *A* **and** *u*::*real*
by (*metis* (*full-types*) *abs-ge-self abs-minus-cancel le-minus-iff order-trans*)
show *?thesis*
by (*auto simp*: *filter-eq-iff eventually-sup eventually-at-infinity*
eventually-at-top-linorder eventually-at-bot-linorder intro: *1 2 3*)
qed

lemma *at-top-le-at-infinity*: *at-top* ≤ (*at-infinity* :: *real filter*)
unfolding *at-infinity-eq-at-top-bot* **by** *simp*

lemma *at-bot-le-at-infinity*: *at-bot* ≤ (*at-infinity* :: *real filter*)
unfolding *at-infinity-eq-at-top-bot* **by** *simp*

lemma *filterlim-at-top-imp-at-infinity*: *filterlim f at-top F* ⟹ *filterlim f at-infinity F*
for *f* :: - ⇒ *real*
by (*rule filterlim-mono*[*OF - at-top-le-at-infinity order-refl*])

lemma *filterlim-real-at-infinity-sequentially*: *filterlim real at-infinity sequentially*
by (*simp add*: *filterlim-at-top-imp-at-infinity filterlim-real-sequentially*)

lemma *lim-infinity-imp-sequentially*: (*f* ⟶ *l*) *at-infinity* ⟹ ((λ*n*. *f*(*n*)) ⟶ *l*) *sequentially*
by (*simp add*: *filterlim-at-top-imp-at-infinity filterlim-compose filterlim-real-sequentially*)

108.1.1 Boundedness

definition *Bfun* :: ('*a* ⇒ '*b*::*metric-space*) ⇒ '*a filter* ⇒ *bool*
where *Bfun-metric-def*: *Bfun f F* = (∃ *y*. ∃ *K*>*0*. *eventually* (λ*x*. *dist* (*f x*) *y* ≤ *K*) *F*)

abbreviation *Bseq* :: (*nat* ⇒ '*a*::*metric-space*) ⇒ *bool*
where *Bseq X* ≡ *Bfun X sequentially*

lemma *Bseq-conv-Bfun*: *Bseq X* ⟷ *Bfun X sequentially* **..**

lemma *Bseq-ignore-initial-segment*: *Bseq X* ⟹ *Bseq* (λ*n*. *X* (*n* + *k*))
unfolding *Bfun-metric-def* **by** (*subst eventually-sequentially-seg*)

lemma *Bseq-offset*: *Bseq* (λ*n*. *X* (*n* + *k*)) ⟹ *Bseq X*
unfolding *Bfun-metric-def* **by** (*subst* (*asm*) *eventually-sequentially-seg*)

lemma *Bfun-def*: *Bfun f F* ⟷ (∃ *K*>*0*. *eventually* (λ*x*. *norm* (*f x*) ≤ *K*) *F*)
unfolding *Bfun-metric-def norm-conv-dist*
proof *safe*
fix *y K*

assume *K*: *0 < K* **and** ∗: *eventually* (λ*x*. *dist* (*f x*) *y* ≤ *K*) *F*
moreover have *eventually* (λ*x*. *dist* (*f x*) *0* ≤ *dist* (*f x*) *y* + *dist 0 y*) *F*
by (*intro always-eventually*) (*metis dist-commute dist-triangle*)
with ∗ **have** *eventually* (λ*x*. *dist* (*f x*) *0* ≤ *K* + *dist 0 y*) *F*
by *eventually-elim auto*
with ‹*0 < K*› **show** ∃ *K*>*0*. *eventually* (λ*x*. *dist* (*f x*) *0* ≤ *K*) *F*
by (*intro exI*[*of - K + dist 0 y*] *add-pos-nonneg conjI zero-le-dist*) *auto*
qed (*force simp del*: *norm-conv-dist* [*symmetric*])

lemma *BfunI*:
assumes *K*: *eventually* (λ*x*. *norm* (*f x*) ≤ *K*) *F*
shows *Bfun f F*
unfolding *Bfun-def*
proof (*intro exI conjI allI*)
show *0 < max K 1* **by** *simp*
show *eventually* (λ*x*. *norm* (*f x*) ≤ *max K 1*) *F*
using *K* **by** (*rule eventually-mono*) *simp*
qed

lemma *BfunE*:
assumes *Bfun f F*
obtains *B* **where** *0 < B* **and** *eventually* (λ*x*. *norm* (*f x*) ≤ *B*) *F*
using *assms* **unfolding** *Bfun-def* **by** *blast*

lemma *Cauchy-Bseq*:
assumes *Cauchy X* **shows** *Bseq X*
proof −
have ∃ *y K*. *0 < K* ∧ (∃ *N*. ∀ *n*≥*N*. *dist* (*X n*) *y* ≤ *K*)
if ⋀*m n*. ⟦*m* ≥ *M*; *n* ≥ *M*⟧ ⟹ *dist* (*X m*) (*X n*) < *1* **for** *M*
by (*meson order.order-iff-strict that zero-less-one*)
with *assms* **show** *?thesis*
by (*force simp*: *Cauchy-def Bfun-metric-def eventually-sequentially*)
qed

108.1.2 Bounded Sequences

lemma *BseqI'*: (⋀*n*. *norm* (*X n*) ≤ *K*) ⟹ *Bseq X*
by (*intro BfunI*) (*auto simp*: *eventually-sequentially*)

lemma *Bseq-def*: *Bseq X* ⟷ (∃ *K*>*0*. ∀ *n*. *norm* (*X n*) ≤ *K*)
unfolding *Bfun-def eventually-sequentially*
proof *safe*
fix *N K*
assume *0 < K* ∀ *n*≥*N*. *norm* (*X n*) ≤ *K*
then show ∃ *K*>*0*. ∀ *n*. *norm* (*X n*) ≤ *K*
by (*intro exI*[*of - max* (*Max* (*norm* ‘ *X* ‘ {..*N*})) *K*] *max.strict-coboundedI2*)
(*auto intro*!: *imageI not-less*[**where** *'a*=*nat*, *THEN iffD1*] *Max-ge simp*: *le-max-iff-disj*)
qed *auto*

lemma *BseqE*: *Bseq X* $\Longrightarrow$ ($\bigwedge$*K*. *0* < *K* $\Longrightarrow$ $\forall$ *n*. *norm* (*X n*) $\leq$ *K* $\Longrightarrow$ *Q*) $\Longrightarrow$ *Q*
unfolding *Bseq-def* **by** *auto*

lemma *BseqD*: *Bseq X* $\Longrightarrow$ $\exists$ *K*. *0* < *K* $\wedge$ ($\forall$ *n*. *norm* (*X n*) $\leq$ *K*)
by (*simp add*: *Bseq-def*)

lemma *BseqI*: *0* < *K* $\Longrightarrow$ $\forall$ *n*. *norm* (*X n*) $\leq$ *K* $\Longrightarrow$ *Bseq X*
by (*auto simp*: *Bseq-def*)

lemma *Bseq-bdd-above*: *Bseq X* $\Longrightarrow$ *bdd-above* (*range X*)
for *X* :: *nat* $\Rightarrow$ *real*
proof (*elim BseqE*, *intro bdd-aboveI2*)
fix *K n*
assume *0* < *K* $\forall$ *n*. *norm* (*X n*) $\leq$ *K*
then show *X n* $\leq$ *K*
by (*auto elim*!: *allE*[*of* - *n*])
qed

lemma *Bseq-bdd-above'*: *Bseq X* $\Longrightarrow$ *bdd-above* (*range* (λ*n*. *norm* (*X n*)))
for *X* :: *nat* $\Rightarrow$ *'a* :: *real-normed-vector*
proof (*elim BseqE*, *intro bdd-aboveI2*)
fix *K n*
assume *0* < *K* $\forall$ *n*. *norm* (*X n*) $\leq$ *K*
then show *norm* (*X n*) $\leq$ *K*
by (*auto elim*!: *allE*[*of* - *n*])
qed

lemma *Bseq-bdd-below*: *Bseq X* $\Longrightarrow$ *bdd-below* (*range X*)
for *X* :: *nat* $\Rightarrow$ *real*
proof (*elim BseqE*, *intro bdd-belowI2*)
fix *K n*
assume *0* < *K* $\forall$ *n*. *norm* (*X n*) $\leq$ *K*
then show − *K* $\leq$ *X n*
by (*auto elim*!: *allE*[*of* - *n*])
qed

lemma *Bseq-eventually-mono*:
assumes *eventually* (λ*n*. *norm* (*f n*) $\leq$ *norm* (*g n*)) *sequentially Bseq g*
shows *Bseq f*
proof −
from *assms*(*2*) **obtain** *K* **where** *0* < *K* **and** *eventually* (λ*n*. *norm* (*g n*) $\leq$ *K*) *sequentially*
unfolding *Bfun-def* **by** *fast*
with *assms*(*1*) **have** *eventually* (λ*n*. *norm* (*f n*) $\leq$ *K*) *sequentially*
by (*fast elim*: *eventually-elim2 order-trans*)
with ‹*0* < *K*› **show** *Bseq f*
unfolding *Bfun-def* **by** *fast*

qed

lemma *lemma-NBseq-def*: $(\exists K > 0.\ \forall n.\ norm\ (X\ n) \leq K) \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) \leq real(Suc\ N))$
proof *safe*
fix K :: *real*
from *reals-Archimedean2* **obtain** n :: *nat* **where** $K < real\ n$ **..**
then have $K \leq real\ (Suc\ n)$ **by** *auto*
moreover assume $\forall m.\ norm\ (X\ m) \leq K$
ultimately have $\forall m.\ norm\ (X\ m) \leq real\ (Suc\ n)$
by (*blast intro*: *order-trans*)
then show $\exists N.\ \forall n.\ norm\ (X\ n) \leq real\ (Suc\ N)$ **..**
next
show $\bigwedge N.\ \forall n.\ norm\ (X\ n) \leq real\ (Suc\ N) \Longrightarrow \exists K{>}0.\ \forall n.\ norm\ (X\ n) \leq K$
using *of-nat-0-less-iff* **by** *blast*
qed

Alternative definition for *Bseq*.

lemma *Bseq-iff*: $Bseq\ X \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) \leq real(Suc\ N))$
by (*simp add*: *Bseq-def*) (*simp add*: *lemma-NBseq-def*)

lemma *lemma-NBseq-def2*: $(\exists K > 0.\ \forall n.\ norm\ (X\ n) \leq K) = (\exists N.\ \forall n.\ norm\ (X\ n) < real(Suc\ N))$
proof −
have $*$: $\bigwedge N.\ \forall n.\ norm\ (X\ n) \leq 1 + real\ N \Longrightarrow$
$\exists N.\ \forall n.\ norm\ (X\ n) < 1 + real\ N$
by (*metis add.commute le-less-trans less-add-one of-nat-Suc*)
then show *?thesis*
unfolding *lemma-NBseq-def*
by (*metis less-le-not-le not-less-iff-gr-or-eq of-nat-Suc*)
qed

Yet another definition for Bseq.

lemma *Bseq-iff1a*: $Bseq\ X \longleftrightarrow (\exists N.\ \forall n.\ norm\ (X\ n) < real\ (Suc\ N))$
by (*simp add*: *Bseq-def lemma-NBseq-def2*)

108.1.3 A Few More Equivalence Theorems for Boundedness

Alternative formulation for boundedness.

lemma *Bseq-iff2*: $Bseq\ X \longleftrightarrow (\exists k > 0.\ \exists x.\ \forall n.\ norm\ (X\ n + - x) \leq k)$
by (*metis BseqE BseqI' add.commute add-cancel-right-left add-uminus-conv-diff norm-add-leD*
norm-minus-cancel norm-minus-commute)

Alternative formulation for boundedness.

lemma *Bseq-iff3*: $Bseq\ X \longleftrightarrow (\exists k{>}0.\ \exists N.\ \forall n.\ norm\ (X\ n + - X\ N) \leq k)$
(**is** *?P* $\longleftrightarrow$ *?Q*)
proof

assume *?P*
then obtain *K* **where** ∗: *0 < K* **and** ∗∗: ⋀*n*. *norm* (*X n*) ≤ *K*
by (*auto simp*: *Bseq-def*)
from ∗ **have** *0 < K + norm* (*X 0*) **by** (*rule order-less-le-trans*) *simp*
from ∗∗ **have** ∀ *n*. *norm* (*X n − X 0*) ≤ *K + norm* (*X 0*)
by (*auto intro*: *order-trans norm-triangle-ineq4*)
then have ∀ *n*. *norm* (*X n + − X 0*) ≤ *K + norm* (*X 0*)
by *simp*
with ‹*0 < K + norm* (*X 0*)› **show** *?Q* **by** *blast*
next
assume *?Q*
then show *?P* **by** (*auto simp*: *Bseq-iff2*)
qed

108.1.4 Upper Bounds and Lubs of Bounded Sequences

lemma *Bseq-minus-iff*: *Bseq* (λ*n*. − (*X n*) :: *'a*::*real-normed-vector*) ⟷ *Bseq X*
by (*simp add*: *Bseq-def*)

lemma *Bseq-add*:
fixes *f* :: *nat* ⇒ *'a*::*real-normed-vector*
assumes *Bseq f*
shows *Bseq* (λ*x*. *f x + c*)
proof −
from *assms* **obtain** *K* **where** *K*: ⋀*x*. *norm* (*f x*) ≤ *K*
unfolding *Bseq-def* **by** *blast*
{
fix *x* :: *nat*
have *norm* (*f x + c*) ≤ *norm* (*f x*) + *norm c* **by** (*rule norm-triangle-ineq*)
also have *norm* (*f x*) ≤ *K* **by** (*rule K*)
finally have *norm* (*f x + c*) ≤ *K + norm c* **by** *simp*
}
then show *?thesis* **by** (*rule BseqI'*)
qed

lemma *Bseq-add-iff*: *Bseq* (λ*x*. *f x + c*) ⟷ *Bseq f*
for *f* :: *nat* ⇒ *'a*::*real-normed-vector*
using *Bseq-add*[*of f c*] *Bseq-add*[*of* λ*x*. *f x + c* −*c*] **by** *auto*

lemma *Bseq-mult*:
fixes *f g* :: *nat* ⇒ *'a*::*real-normed-field*
assumes *Bseq f* **and** *Bseq g*
shows *Bseq* (λ*x*. *f x* ∗ *g x*)
proof −
from *assms* **obtain** *K1 K2* **where** *K*: *norm* (*f x*) ≤ *K1 K1 > 0 norm* (*g x*) ≤ *K2 K2 > 0*
for *x*
unfolding *Bseq-def* **by** *blast*
then have *norm* (*f x* ∗ *g x*) ≤ *K1* ∗ *K2* **for** *x*

```
    by (auto simp: norm-mult intro!: mult-mono)
  then show ?thesis by (rule BseqI')
qed

lemma Bfun-const [simp]: Bfun (λ-. c) F
  unfolding Bfun-metric-def by (auto intro!: exI[of - c] exI[of - 1::real])

lemma Bseq-cmult-iff:
  fixes c :: 'a::real-normed-field
  assumes c ≠ 0
  shows Bseq (λx. c * f x) ⟷ Bseq f
proof
  assume Bseq (λx. c * f x)
  with Bfun-const have Bseq (λx. inverse c * (c * f x))
    by (rule Bseq-mult)
  with ‹c ≠ 0› show Bseq f
    by (simp add: field-split-simps)
qed (intro Bseq-mult Bfun-const)

lemma Bseq-subseq: Bseq f ⟹ Bseq (λx. f (g x))
  for f :: nat ⇒ 'a::real-normed-vector
  unfolding Bseq-def by auto

lemma Bseq-Suc-iff: Bseq (λn. f (Suc n)) ⟷ Bseq f
  for f :: nat ⇒ 'a::real-normed-vector
  using Bseq-offset[of f 1] by (auto intro: Bseq-subseq)

lemma increasing-Bseq-subseq-iff:
  assumes ⋀x y. x ≤ y ⟹ norm (f x :: 'a::real-normed-vector) ≤ norm (f y)
strict-mono g
  shows Bseq (λx. f (g x)) ⟷ Bseq f
proof
  assume Bseq (λx. f (g x))
  then obtain K where K: ⋀x. norm (f (g x)) ≤ K
    unfolding Bseq-def by auto
  {
    fix x :: nat
    from filterlim-subseq[OF assms(2)] obtain y where g y ≥ x
      by (auto simp: filterlim-at-top eventually-at-top-linorder)
    then have norm (f x) ≤ norm (f (g y))
      using assms(1) by blast
    also have norm (f (g y)) ≤ K by (rule K)
    finally have norm (f x) ≤ K .
  }
  then show Bseq f by (rule BseqI')
qed (use Bseq-subseq[of f g] in simp-all)

lemma nonneg-incseq-Bseq-subseq-iff:
  fixes f :: nat ⇒ real
```

and $g :: nat \Rightarrow nat$
assumes $\bigwedge x.\ f\ x \geq 0$ *incseq f strict-mono g*
shows *Bseq* $(\lambda x.\ f\ (g\ x)) \longleftrightarrow$ *Bseq f*
using *assms* **by** (*intro increasing-Bseq-subseq-iff*) (*auto simp*: *incseq-def*)

lemma *Bseq-eq-bounded*: *range f* $\subseteq \{a..b\} \Longrightarrow$ *Bseq f*
for *a b* :: *real*
proof (*rule BseqI'*[**where** *K*=*max* (*norm a*) (*norm b*)])
fix *n* **assume** *range f* $\subseteq \{a..b\}$
then have *f n* $\in \{a..b\}$
by *blast*
then show *norm* (*f n*) $\leq$ *max* (*norm a*) (*norm b*)
by *auto*
qed

lemma *incseq-bounded*: *incseq X* $\Longrightarrow \forall i.\ X\ i \leq B \Longrightarrow$ *Bseq X*
for *B* :: *real*
by (*intro Bseq-eq-bounded*[*of X X 0 B*]) (*auto simp*: *incseq-def*)

lemma *decseq-bounded*: *decseq X* $\Longrightarrow \forall i.\ B \leq X\ i \Longrightarrow$ *Bseq X*
for *B* :: *real*
by (*intro Bseq-eq-bounded*[*of X B X 0*]) (*auto simp*: *decseq-def*)

108.1.5 Polynomal function extremal theorem, from HOL Light

lemma *polyfun-extremal-lemma*:
fixes *c* :: *nat* $\Rightarrow$ *'a*::*real-normed-div-algebra*
assumes *0* $<$ *e*
shows $\exists M.\ \forall z.\ M \leq norm(z) \longrightarrow norm\ (\sum i{\leq}n.\ c(i) * z\hat{}i) \leq e * norm(z) \hat{}\ (Suc\ n)$
proof (*induct n*)
case *0* **with** *assms*
show *?case*
apply (*rule-tac x*=*norm* (*c 0*) / *e* **in** *exI*)
apply (*auto simp*: *field-simps*)
done
next
case (*Suc n*)
obtain *M* **where** *M*: $\bigwedge z.\ M \leq norm\ z \Longrightarrow norm\ (\sum i{\leq}n.\ c\ i * z\hat{}i) \leq e * norm\ z \hat{}\ Suc\ n$
using *Suc assms* **by** *blast*
show *?case*
proof (*rule exI* [**where** *x*= *max M* (*1* + *norm*(*c*(*Suc n*)) / *e*)], *clarsimp simp del*: *power-Suc*)
fix *z*::*'a*
assume *z1*: *M* $\leq$ *norm z* **and** *1* + *norm* (*c* (*Suc n*)) / *e* $\leq$ *norm z*
then have *z2*: *e* + *norm* (*c* (*Suc n*)) $\leq$ *e* $*$ *norm z*
using *assms* **by** (*simp add*: *field-simps*)
have *norm* $(\sum i{\leq}n.\ c\ i * z\hat{}i) \leq e * norm\ z \hat{}\ Suc\ n$

```
      using M [OF z1] by simp
    then have norm (∑i≤n. c i * z^i) + norm (c (Suc n) * z ^ Suc n) ≤ e *
norm z ^ Suc n + norm (c (Suc n) * z ^ Suc n)
      by simp
    then have norm ((∑i≤n. c i * z^i) + c (Suc n) * z ^ Suc n) ≤ e * norm z
^ Suc n + norm (c (Suc n) * z ^ Suc n)
      by (blast intro: norm-triangle-le elim: )
    also have ... ≤ (e + norm (c (Suc n))) * norm z ^ Suc n
      by (simp add: norm-power norm-mult algebra-simps)
    also have ... ≤ (e * norm z) * norm z ^ Suc n
      by (metis z2 mult.commute mult-left-mono norm-ge-zero norm-power)
    finally show norm ((∑i≤n. c i * z^i) + c (Suc n) * z ^ Suc n) ≤ e * norm
z ^ Suc (Suc n)
      by simp
  qed
qed

lemma polyfun-extremal:
    fixes c :: nat ⇒ 'a::real-normed-div-algebra
  assumes k: c k ≠ 0 1≤k and kn: k≤n
    shows eventually (λz. norm (∑i≤n. c(i) * z^i) ≥ B) at-infinity
using kn
proof (induction n)
  case 0
  then show ?case
    using k by simp
next
  case (Suc m)
  show ?case
  proof (cases c (Suc m) = 0)
    case True
    then show ?thesis using Suc k
      by auto (metis antisym-conv less-eq-Suc-le not-le)
  next
    case False
    then obtain M where M:
          ⋀z. M ≤ norm z ⟹ norm (∑i≤m. c i * z^i) ≤ norm (c (Suc m)) / 2
* norm z ^ Suc m
      using polyfun-extremal-lemma [of norm(c (Suc m)) / 2 c m] Suc
      by auto
    have ∃ b. ∀ z. b ≤ norm z ⟶ B ≤ norm (∑i≤Suc m. c i * z^i)
    proof (rule exI [where x=max M (max 1 (|B| / (norm(c (Suc m)) / 2)))],
clarsimp simp del: power-Suc)
      fix z::'a
      assume z1: M ≤ norm z 1 ≤ norm z
        and |B| * 2 / norm (c (Suc m)) ≤ norm z
      then have z2: |B| ≤ norm (c (Suc m)) * norm z / 2
        using False by (simp add: field-simps)
      have nz: norm z ≤ norm z ^ Suc m
```

by (*metis ‹1 ≤ norm z› One-nat-def less-eq-Suc-le power-increasing power-one-right zero-less-Suc*)
have ∗: ⋀*y x. norm* (*c* (*Suc m*)) ∗ *norm z* / *2* ≤ *norm y* − *norm x* ⟹ *B* ≤ *norm* (*x* + *y*)
by (*metis abs-le-iff add.commute norm-diff-ineq order-trans z2*)
have *norm z* ∗ *norm* (*c* (*Suc m*)) + *2* ∗ *norm* (∑ *i*≤*m. c i* ∗ *z*^*i*)
≤ *norm* (*c* (*Suc m*)) ∗ *norm z* + *norm* (*c* (*Suc m*)) ∗ *norm z* ^ *Suc m*
using *M* [*of z*] *Suc z1* **by** *auto*
also have ... ≤ *2* ∗ (*norm* (*c* (*Suc m*)) ∗ *norm z* ^ *Suc m*)
using *nz* **by** (*simp add*: *mult-mono del*: *power-Suc*)
finally show *B* ≤ *norm* ((∑ *i*≤*m. c i* ∗ *z*^*i*) + *c* (*Suc m*) ∗ *z* ^ *Suc m*)
using *Suc.IH*
apply (*auto simp*: *eventually-at-infinity*)
apply (*rule* ∗)
apply (*simp add*: *field-simps norm-mult norm-power*)
done
qed
then show *?thesis*
by (*simp add*: *eventually-at-infinity*)
qed
qed

108.2 Convergence to Zero

definition *Zfun* :: (*'a* ⇒ *'b*::*real-normed-vector*) ⇒ *'a filter* ⇒ *bool*
where *Zfun f F* = (∀ *r*>*0. eventually* (λ*x. norm* (*f x*) < *r*) *F*)

lemma *ZfunI*: (⋀*r. 0* < *r* ⟹ *eventually* (λ*x. norm* (*f x*) < *r*) *F*) ⟹ *Zfun f F*
by (*simp add*: *Zfun-def*)

lemma *ZfunD*: *Zfun f F* ⟹ *0* < *r* ⟹ *eventually* (λ*x. norm* (*f x*) < *r*) *F*
by (*simp add*: *Zfun-def*)

lemma *Zfun-ssubst*: *eventually* (λ*x. f x* = *g x*) *F* ⟹ *Zfun g F* ⟹ *Zfun f F*
unfolding *Zfun-def* **by** (*auto elim*!: *eventually-rev-mp*)

lemma *Zfun-zero*: *Zfun* (λ*x. 0*) *F*
unfolding *Zfun-def* **by** *simp*

lemma *Zfun-norm-iff*: *Zfun* (λ*x. norm* (*f x*)) *F* = *Zfun* (λ*x. f x*) *F*
unfolding *Zfun-def* **by** *simp*

lemma *Zfun-cong*: *eventually* (λ*x. f x* = *g x*) *F* ⟹ *Zfun f F* = *Zfun g F*
by (*smt* (*verit*) *Zfun-ssubst eventually-mono*)

lemma *Zfun-imp-Zfun*:
assumes *f*: *Zfun f F*
and *g*: *eventually* (λ*x. norm* (*g x*) ≤ *norm* (*f x*) ∗ *K*) *F*
shows *Zfun* (λ*x. g x*) *F*

```
proof (cases 0 < K)
  case K: True
  show ?thesis
  proof (rule ZfunI)
    fix r :: real
    assume 0 < r
    then have 0 < r / K using K by simp
    then have eventually (λx. norm (f x) < r / K) F
      using ZfunD [OF f] by blast
    with g show eventually (λx. norm (g x) < r) F
    proof eventually-elim
      case (elim x)
      then have norm (f x) * K < r
        by (simp add: pos-less-divide-eq K)
      then show ?case
        by (simp add: order-le-less-trans [OF elim(1)])
    qed
  qed
next
  case False
  then have K: K ≤ 0 by (simp only: not-less)
  show ?thesis
  proof (rule ZfunI)
    fix r :: real
    assume 0 < r
    from g show eventually (λx. norm (g x) < r) F
    proof eventually-elim
      case (elim x)
      also have norm (f x) * K ≤ norm (f x) * 0
        using K norm-ge-zero by (rule mult-left-mono)
      finally show ?case
        using ‹0 < r› by simp
    qed
  qed
qed

lemma Zfun-le: Zfun g F ⟹ ∀x. norm (f x) ≤ norm (g x) ⟹ Zfun f F
  by (erule Zfun-imp-Zfun [where K = 1]) simp

lemma Zfun-add:
  assumes f: Zfun f F
    and g: Zfun g F
  shows Zfun (λx. f x + g x) F
proof (rule ZfunI)
  fix r :: real
  assume 0 < r
  then have r: 0 < r / 2 by simp
  have eventually (λx. norm (f x) < r/2) F
    using f r by (rule ZfunD)
```

```
  moreover
  have eventually (λx. norm (g x) < r/2) F
    using g r by (rule ZfunD)
  ultimately
  show eventually (λx. norm (f x + g x) < r) F
  proof eventually-elim
    case (elim x)
    have norm (f x + g x) ≤ norm (f x) + norm (g x)
      by (rule norm-triangle-ineq)
    also have ... < r/2 + r/2
      using elim by (rule add-strict-mono)
    finally show ?case
      by simp
  qed
qed

lemma Zfun-minus: Zfun f F ⟹ Zfun (λx. − f x) F
  unfolding Zfun-def by simp

lemma Zfun-diff: Zfun f F ⟹ Zfun g F ⟹ Zfun (λx. f x − g x) F
  using Zfun-add [of f F λx. − g x] by (simp add: Zfun-minus)

lemma (in bounded-linear) Zfun:
  assumes g: Zfun g F
  shows Zfun (λx. f (g x)) F
proof −
  obtain K where norm (f x) ≤ norm x * K for x
    using bounded by blast
  then have eventually (λx. norm (f (g x)) ≤ norm (g x) * K) F
    by simp
  with g show ?thesis
    by (rule Zfun-imp-Zfun)
qed

lemma (in bounded-bilinear) Zfun:
  assumes f: Zfun f F
    and g: Zfun g F
  shows Zfun (λx. f x ** g x) F
proof (rule ZfunI)
  fix r :: real
  assume r: 0 < r
  obtain K where K: 0 < K
    and norm-le: norm (x ** y) ≤ norm x * norm y * K for x y
    using pos-bounded by blast
  from K have K': 0 < inverse K
    by (rule positive-imp-inverse-positive)
  have eventually (λx. norm (f x) < r) F
    using f r by (rule ZfunD)
  moreover
```

```
  have eventually (λx. norm (g x) < inverse K) F
    using g K' by (rule ZfunD)
  ultimately
  show eventually (λx. norm (f x ** g x) < r) F
  proof eventually-elim
    case (elim x)
    have norm (f x ** g x) ≤ norm (f x) * norm (g x) * K
      by (rule norm-le)
    also have norm (f x) * norm (g x) * K < r * inverse K * K
      by (intro mult-strict-right-mono mult-strict-mono' norm-ge-zero elim K)
    also from K have r * inverse K * K = r
      by simp
    finally show ?case .
  qed
qed

lemma (in bounded-bilinear) Zfun-left: Zfun f F ⟹ Zfun (λx. f x ** a) F
  by (rule bounded-linear-left [THEN bounded-linear.Zfun])

lemma (in bounded-bilinear) Zfun-right: Zfun f F ⟹ Zfun (λx. a ** f x) F
  by (rule bounded-linear-right [THEN bounded-linear.Zfun])

lemmas Zfun-mult = bounded-bilinear.Zfun [OF bounded-bilinear-mult]
lemmas Zfun-mult-right = bounded-bilinear.Zfun-right [OF bounded-bilinear-mult]
lemmas Zfun-mult-left = bounded-bilinear.Zfun-left [OF bounded-bilinear-mult]

lemma tendsto-Zfun-iff: (f ⟶ a) F = Zfun (λx. f x - a) F
  by (simp only: tendsto-iff Zfun-def dist-norm)

lemma tendsto-0-le:
  (f ⟶ 0) F ⟹ eventually (λx. norm (g x) ≤ norm (f x) * K) F ⟹ (g ⟶
0) F
  by (simp add: Zfun-imp-Zfun tendsto-Zfun-iff)
```

108.3 Distance and norms

```
lemma tendsto-dist [tendsto-intros]:
  fixes l m :: 'a::metric-space
  assumes f: (f ⟶ l) F
    and g: (g ⟶ m) F
  shows ((λx. dist (f x) (g x)) ⟶ dist l m) F
proof (rule tendstoI)
  fix e :: real
  assume 0 < e
  then have e2: 0 < e/2 by simp
  from tendstoD [OF f e2] tendstoD [OF g e2]
  show eventually (λx. dist (dist (f x) (g x)) (dist l m) < e) F
  proof (eventually-elim)
    case (elim x)
```

then show *dist* (*dist* (*f x*) (*g x*)) (*dist l m*) < *e*
unfolding *dist-real-def*
using *dist-triangle2* [*of f x g x l*]
and *dist-triangle2* [*of g x l m*]
and *dist-triangle3* [*of l m f x*]
and *dist-triangle* [*of f x m g x*]
by *arith*
qed
qed

lemma *continuous-dist*[*continuous-intros*]:
fixes *f g* :: - ⇒ *'a* :: *metric-space*
shows *continuous F f* ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. *dist* (*f x*) (*g x*))
unfolding *continuous-def* **by** (*rule tendsto-dist*)

lemma *continuous-on-dist*[*continuous-intros*]:
fixes *f g* :: - ⇒ *'a* :: *metric-space*
shows *continuous-on s f* ⟹ *continuous-on s g* ⟹ *continuous-on s* (λ*x*. *dist* (*f x*) (*g x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-dist*)

lemma *continuous-at-dist*: *isCont* (*dist a*) *b*
using *continuous-on-dist* [*OF continuous-on-const continuous-on-id*] *continuous-on-eq-continuous-within* **by** *blast*

lemma *tendsto-norm* [*tendsto-intros*]: (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *norm* (*f x*)) ⟶ *norm a*) *F*
unfolding *norm-conv-dist* **by** (*intro tendsto-intros*)

lemma *continuous-norm* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *norm* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-norm*)

lemma *continuous-on-norm* [*continuous-intros*]:
continuous-on s f ⟹ *continuous-on s* (λ*x*. *norm* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-norm*)

lemma *continuous-on-norm-id* [*continuous-intros*]: *continuous-on S norm*
by (*intro continuous-on-id continuous-on-norm*)

lemma *tendsto-norm-zero*: (*f* ⟶ *0*) *F* ⟹ ((λ*x*. *norm* (*f x*)) ⟶ *0*) *F*
by (*drule tendsto-norm*) *simp*

lemma *tendsto-norm-zero-cancel*: ((λ*x*. *norm* (*f x*)) ⟶ *0*) *F* ⟹ (*f* ⟶ *0*) *F*
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *tendsto-norm-zero-iff*: ((λ*x*. *norm* (*f x*)) ⟶ *0*) *F* ⟷ (*f* ⟶ *0*) *F*

unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *tendsto-rabs* [*tendsto-intros*]: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ |f\ x|) \longrightarrow |l|)\ F$
for l :: *real*
by (*fold real-norm-def*) (*rule tendsto-norm*)

lemma *continuous-rabs* [*continuous-intros*]:
continuous F f $\Longrightarrow$ *continuous F* $(\lambda x.\ |f\ x :: real|)$
unfolding *real-norm-def*[*symmetric*] **by** (*rule continuous-norm*)

lemma *continuous-on-rabs* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* $(\lambda x.\ |f\ x :: real|)$
unfolding *real-norm-def*[*symmetric*] **by** (*rule continuous-on-norm*)

lemma *tendsto-rabs-zero*: $(f \longrightarrow (0::real))\ F \Longrightarrow ((\lambda x.\ |f\ x|) \longrightarrow 0)\ F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero*)

lemma *tendsto-rabs-zero-cancel*: $((\lambda x.\ |f\ x|) \longrightarrow (0::real))\ F \Longrightarrow (f \longrightarrow 0)\ F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero-cancel*)

lemma *tendsto-rabs-zero-iff*: $((\lambda x.\ |f\ x|) \longrightarrow (0::real))\ F \longleftrightarrow (f \longrightarrow 0)\ F$
by (*fold real-norm-def*) (*rule tendsto-norm-zero-iff*)

108.4 Topological Monoid

class *topological-monoid-add* = *topological-space* + *monoid-add* +
assumes *tendsto-add-Pair*: *LIM x* (*nhds a* $\times_F$ *nhds b*). *fst x* + *snd x* :> *nhds* (*a* + *b*)

class *topological-comm-monoid-add* = *topological-monoid-add* + *comm-monoid-add*

lemma *tendsto-add* [*tendsto-intros*]:
fixes *a b* :: *'a*::*topological-monoid-add*
shows $(f \longrightarrow a)\ F \Longrightarrow (g \longrightarrow b)\ F \Longrightarrow ((\lambda x.\ f\ x + g\ x) \longrightarrow a + b)\ F$
using *filterlim-compose*[*OF tendsto-add-Pair*, *of* $\lambda x.\ (f\ x,\ g\ x)$ *a b F*]
by (*simp add*: *nhds-prod*[*symmetric*] *tendsto-Pair*)

lemma *continuous-add* [*continuous-intros*]:
fixes *f g* :: - $\Rightarrow$ *'b*::*topological-monoid-add*
shows *continuous F f* $\Longrightarrow$ *continuous F g* $\Longrightarrow$ *continuous F* $(\lambda x.\ f\ x + g\ x)$
unfolding *continuous-def* **by** (*rule tendsto-add*)

lemma *continuous-on-add* [*continuous-intros*]:
fixes *f g* :: - $\Rightarrow$ *'b*::*topological-monoid-add*
shows *continuous-on s f* $\Longrightarrow$ *continuous-on s g* $\Longrightarrow$ *continuous-on s* $(\lambda x.\ f\ x + g\ x)$
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-add*)

lemma *tendsto-add-zero*:

fixes $f\ g :: - \Rightarrow 'b$::*topological-monoid-add*
shows $(f \longrightarrow 0)\ F \Longrightarrow (g \longrightarrow 0)\ F \Longrightarrow ((\lambda x.\ f\ x + g\ x) \longrightarrow 0)\ F$
by (*drule* (*1*) *tendsto-add*) *simp*

lemma *tendsto-sum* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow (f\ i \longrightarrow a\ i)\ F) \Longrightarrow ((\lambda x.\ \sum i{\in}I.\ f\ i\ x) \longrightarrow (\sum i{\in}I.\ a\ i))\ F$
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *tendsto-add*)

lemma *tendsto-null-sum*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$::*topological-comm-monoid-add*
assumes $\bigwedge i.\ i \in I \Longrightarrow ((\lambda x.\ f\ x\ i) \longrightarrow 0)\ F$
shows $((\lambda i.\ sum\ (f\ i)\ I) \longrightarrow 0)\ F$
using *tendsto-sum* [*of I* $\lambda x\ y.\ f\ y\ x\ \lambda x.\ 0$] *assms* **by** *simp*

lemma *continuous-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b$::*t2-space* $\Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow continuous\ F\ (f\ i)) \Longrightarrow continuous\ F\ (\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
unfolding *continuous-def* **by** (*rule tendsto-sum*)

lemma *continuous-on-sum* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b$::*topological-space* $\Rightarrow 'c$::*topological-comm-monoid-add*
shows $(\bigwedge i.\ i \in I \Longrightarrow continuous\text{-}on\ S\ (f\ i)) \Longrightarrow continuous\text{-}on\ S\ (\lambda x.\ \sum i{\in}I.\ f\ i\ x)$
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-sum*)

instance *nat* :: *topological-comm-monoid-add*
by *standard*
(*simp add*: *nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

instance *int* :: *topological-comm-monoid-add*
by *standard*
(*simp add*: *nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

108.4.1 Topological group

class *topological-group-add* = *topological-monoid-add* + *group-add* +
assumes *tendsto-uminus-nhds*: $(uminus \longrightarrow -\ a)\ (nhds\ a)$
begin

lemma *tendsto-minus* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ -\ f\ x) \longrightarrow -\ a)\ F$
by (*rule filterlim-compose*[*OF tendsto-uminus-nhds*])

end

class *topological-ab-group-add* = *topological-group-add* + *ab-group-add*

instance *topological-ab-group-add* < *topological-comm-monoid-add* **..**

lemma *continuous-minus* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. − *f x*)
for *f* :: *'a*::*t2-space* ⇒ *'b*::*topological-group-add*
unfolding *continuous-def* **by** (*rule tendsto-minus*)

lemma *continuous-on-minus* [*continuous-intros*]: *continuous-on s f* ⟹ *continuous-on s* (λ*x*. − *f x*)
for *f* :: - ⇒ *'b*::*topological-group-add*
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-minus*)

lemma *tendsto-minus-cancel*: ((λ*x*. − *f x*) ⟶ − *a*) *F* ⟹ (*f* ⟶ *a*) *F*
for *a* :: *'a*::*topological-group-add*
by (*drule tendsto-minus*) *simp*

lemma *tendsto-minus-cancel-left*:
(*f* ⟶ − (*y*::-::*topological-group-add*)) *F* ⟷ ((λ*x*. − *f x*) ⟶ *y*) *F*
using *tendsto-minus-cancel*[*of f* − *y F*] *tendsto-minus*[*of f* − *y F*]
by *auto*

lemma *tendsto-diff* [*tendsto-intros*]:
fixes *a b* :: *'a*::*topological-group-add*
shows (*f* ⟶ *a*) *F* ⟹ (*g* ⟶ *b*) *F* ⟹ ((λ*x*. *f x* − *g x*) ⟶ *a* − *b*) *F*
using *tendsto-add* [*of f a F* λ*x*. − *g x* − *b*] **by** (*simp add*: *tendsto-minus*)

lemma *continuous-diff* [*continuous-intros*]:
fixes *f g* :: *'a*::*t2-space* ⇒ *'b*::*topological-group-add*
shows *continuous F f* ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. *f x* − *g x*)
unfolding *continuous-def* **by** (*rule tendsto-diff*)

lemma *continuous-on-diff* [*continuous-intros*]:
fixes *f g* :: - ⇒ *'b*::*topological-group-add*
shows *continuous-on s f* ⟹ *continuous-on s g* ⟹ *continuous-on s* (λ*x*. *f x* − *g x*)
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-diff*)

lemma *continuous-on-op-minus*: *continuous-on* (*s*::*'a*::*topological-group-add set*) ((−) *x*)
by (*rule continuous-intros* | *simp*)+

instance *real-normed-vector* < *topological-ab-group-add*
proof
fix *a b* :: *'a*
show ((λ*x*. *fst x* + *snd x*) ⟶ *a* + *b*) (*nhds a* ×$_F$ *nhds b*)
unfolding *tendsto-Zfun-iff add-diff-add*
using *tendsto-fst*[*OF filterlim-ident*, *of* (*a*,*b*)] *tendsto-snd*[*OF filterlim-ident*, *of* (*a*,*b*)]
by (*intro Zfun-add*)

(*auto simp*: *tendsto-Zfun-iff*[*symmetric*] *nhds-prod*[*symmetric*] *intro*!: *tendsto-fst*)
show (*uminus* ⟶ − *a*) (*nhds a*)
unfolding *tendsto-Zfun-iff minus-diff-minus*
using *filterlim-ident*[*of nhds a*]
by (*intro Zfun-minus*) (*simp add*: *tendsto-Zfun-iff*)
qed

lemmas *real-tendsto-sandwich* = *tendsto-sandwich*[**where** *'a*=*real*]

108.4.2 Linear operators and multiplication

lemma *linear-times* [*simp*]: *linear* (λ*x*. *c* ∗ *x*)
for *c* :: *'a*::*real-algebra*
by (*auto simp*: *linearI distrib-left*)

lemma (**in** *bounded-linear*) *tendsto*: (*g* ⟶ *a*) *F* ⟹ ((λ*x*. *f* (*g x*)) ⟶ *f a*) *F*
by (*simp only*: *tendsto-Zfun-iff diff* [*symmetric*] *Zfun*)

lemma (**in** *bounded-linear*) *continuous*: *continuous F g* ⟹ *continuous F* (λ*x*. *f* (*g x*))
using *tendsto*[*of g - F*] **by** (*auto simp*: *continuous-def*)

lemma (**in** *bounded-linear*) *continuous-on*: *continuous-on s g* ⟹ *continuous-on s* (λ*x*. *f* (*g x*))
using *tendsto*[*of g*] **by** (*auto simp*: *continuous-on-def*)

lemma (**in** *bounded-linear*) *tendsto-zero*: (*g* ⟶ *0*) *F* ⟹ ((λ*x*. *f* (*g x*)) ⟶ *0*) *F*
by (*drule tendsto*) (*simp only*: *zero*)

lemma (**in** *bounded-bilinear*) *tendsto*:
(*f* ⟶ *a*) *F* ⟹ (*g* ⟶ *b*) *F* ⟹ ((λ*x*. *f x* ∗∗ *g x*) ⟶ *a* ∗∗ *b*) *F*
by (*simp only*: *tendsto-Zfun-iff prod-diff-prod Zfun-add Zfun Zfun-left Zfun-right*)

lemma (**in** *bounded-bilinear*) *continuous*:
continuous F f ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. *f x* ∗∗ *g x*)
using *tendsto*[*of f - F g*] **by** (*auto simp*: *continuous-def*)

lemma (**in** *bounded-bilinear*) *continuous-on*:
continuous-on s f ⟹ *continuous-on s g* ⟹ *continuous-on s* (λ*x*. *f x* ∗∗ *g x*)
using *tendsto*[*of f - - g*] **by** (*auto simp*: *continuous-on-def*)

lemma (**in** *bounded-bilinear*) *tendsto-zero*:
assumes *f*: (*f* ⟶ *0*) *F*
and *g*: (*g* ⟶ *0*) *F*
shows ((λ*x*. *f x* ∗∗ *g x*) ⟶ *0*) *F*
using *tendsto* [*OF f g*] **by** (*simp add*: *zero-left*)

lemma (**in** *bounded-bilinear*) *tendsto-left-zero*:
(*f* ⟶ *0*) *F* ⟹ ((λ*x*. *f x ** *c*) ⟶ *0*) *F*
by (*rule bounded-linear.tendsto-zero* [*OF bounded-linear-left*])

lemma (**in** *bounded-bilinear*) *tendsto-right-zero*:
(*f* ⟶ *0*) *F* ⟹ ((λ*x*. *c ** *f x*) ⟶ *0*) *F*
by (*rule bounded-linear.tendsto-zero* [*OF bounded-linear-right*])

lemmas *tendsto-of-real* [*tendsto-intros*] =
bounded-linear.tendsto [*OF bounded-linear-of-real*]

lemmas *tendsto-scaleR* [*tendsto-intros*] =
bounded-bilinear.tendsto [*OF bounded-bilinear-scaleR*]

Analogous type class for multiplication

class *topological-semigroup-mult* = *topological-space* + *semigroup-mult* +
assumes *tendsto-mult-Pair*: *LIM x* (*nhds a* ×$_F$ *nhds b*). *fst x* ∗ *snd x* :> *nhds* (*a* ∗ *b*)

instance *real-normed-algebra* < *topological-semigroup-mult*
proof
fix *a b* :: *'a*
show ((λ*x*. *fst x* ∗ *snd x*) ⟶ *a* ∗ *b*) (*nhds a* ×$_F$ *nhds b*)
unfolding *nhds-prod*[*symmetric*]
using *tendsto-fst*[*OF filterlim-ident*, *of* (*a*,*b*)] *tendsto-snd*[*OF filterlim-ident*, *of* (*a*,*b*)]
by (*simp add*: *bounded-bilinear.tendsto* [*OF bounded-bilinear-mult*])
qed

lemma *tendsto-mult* [*tendsto-intros*]:
fixes *a b* :: *'a*::*topological-semigroup-mult*
shows (*f* ⟶ *a*) *F* ⟹ (*g* ⟶ *b*) *F* ⟹ ((λ*x*. *f x* ∗ *g x*) ⟶ *a* ∗ *b*) *F*
using *filterlim-compose*[*OF tendsto-mult-Pair*, *of* λ*x*. (*f x*, *g x*) *a b F*]
by (*simp add*: *nhds-prod*[*symmetric*] *tendsto-Pair*)

lemma *tendsto-mult-left*: (*f* ⟶ *l*) *F* ⟹ ((λ*x*. *c* ∗ (*f x*)) ⟶ *c* ∗ *l*) *F*
for *c* :: *'a*::*topological-semigroup-mult*
by (*rule tendsto-mult* [*OF tendsto-const*])

lemma *tendsto-mult-right*: (*f* ⟶ *l*) *F* ⟹ ((λ*x*. (*f x*) ∗ *c*) ⟶ *l* ∗ *c*) *F*
for *c* :: *'a*::*topological-semigroup-mult*
by (*rule tendsto-mult* [*OF* - *tendsto-const*])

lemma *tendsto-mult-left-iff* [*simp*]:
c ≠ *0* ⟹ *tendsto*(λ*x*. *c* ∗ *f x*) (*c* ∗ *l*) *F* ⟷ *tendsto f l F* **for** *c* :: *'a*::{*topological-semigroup-mult*,*field*}
by (*auto simp*: *tendsto-mult-left dest*: *tendsto-mult-left* [**where** *c* = *1*/*c*])

lemma *tendsto-mult-right-iff* [*simp*]:
c ≠ *0* ⟹ *tendsto*(λ*x*. *f x* ∗ *c*) (*l* ∗ *c*) *F* ⟷ *tendsto f l F* **for** *c* :: *'a*::{*topological-semigroup-mult*,*field*}

by (*auto simp*: *tendsto-mult-right dest*: *tendsto-mult-left* [**where** $c = 1/c$])

lemma *tendsto-zero-mult-left-iff* [*simp*]:
fixes $c::'a::\{topological\text{-}semigroup\text{-}mult,field\}$ **assumes** $c \neq 0$ **shows** $(\lambda n.\ c * a\ n) \longrightarrow 0 \longleftrightarrow a \longrightarrow 0$
using *assms tendsto-mult-left tendsto-mult-left-iff* **by** *fastforce*

lemma *tendsto-zero-mult-right-iff* [*simp*]:
fixes $c::'a::\{topological\text{-}semigroup\text{-}mult,field\}$ **assumes** $c \neq 0$ **shows** $(\lambda n.\ a\ n * c) \longrightarrow 0 \longleftrightarrow a \longrightarrow 0$
using *assms tendsto-mult-right tendsto-mult-right-iff* **by** *fastforce*

lemma *tendsto-zero-divide-iff* [*simp*]:
fixes $c::'a::\{topological\text{-}semigroup\text{-}mult,field\}$ **assumes** $c \neq 0$ **shows** $(\lambda n.\ a\ n\ /\ c) \longrightarrow 0 \longleftrightarrow a \longrightarrow 0$
using *tendsto-zero-mult-right-iff* [*of* $1/c$ *a*] *assms* **by** (*simp add*: *field-simps*)

lemma *lim-const-over-n* [*tendsto-intros*]:
fixes $a :: 'a::real\text{-}normed\text{-}field$
shows $(\lambda n.\ a\ /\ of\text{-}nat\ n) \longrightarrow 0$
using *tendsto-mult* [*OF tendsto-const* [*of a*] *lim-1-over-n*] **by** *simp*

lemmas *continuous-of-real* [*continuous-intros*] =
bounded-linear.continuous [*OF bounded-linear-of-real*]

lemmas *continuous-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-scaleR*]

lemmas *continuous-mult* [*continuous-intros*] =
bounded-bilinear.continuous [*OF bounded-bilinear-mult*]

lemmas *continuous-on-of-real* [*continuous-intros*] =
bounded-linear.continuous-on [*OF bounded-linear-of-real*]

lemmas *continuous-on-scaleR* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-scaleR*]

lemmas *continuous-on-mult* [*continuous-intros*] =
bounded-bilinear.continuous-on [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-zero* =
bounded-bilinear.tendsto-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-left-zero* =
bounded-bilinear.tendsto-left-zero [*OF bounded-bilinear-mult*]

lemmas *tendsto-mult-right-zero* =
bounded-bilinear.tendsto-right-zero [*OF bounded-bilinear-mult*]

lemma *continuous-mult-left*:
fixes *c*::*'a*::*real-normed-algebra*
shows *continuous F f* $\Longrightarrow$ *continuous F* ($\lambda x.$ *c* $*$ *f x*)
by (*rule continuous-mult* [*OF continuous-const*])

lemma *continuous-mult-right*:
fixes *c*::*'a*::*real-normed-algebra*
shows *continuous F f* $\Longrightarrow$ *continuous F* ($\lambda x.$ *f x* $*$ *c*)
by (*rule continuous-mult* [*OF - continuous-const*])

lemma *continuous-on-mult-left*:
fixes *c*::*'a*::*real-normed-algebra*
shows *continuous-on s f* $\Longrightarrow$ *continuous-on s* ($\lambda x.$ *c* $*$ *f x*)
by (*rule continuous-on-mult* [*OF continuous-on-const*])

lemma *continuous-on-mult-right*:
fixes *c*::*'a*::*real-normed-algebra*
shows *continuous-on s f* $\Longrightarrow$ *continuous-on s* ($\lambda x.$ *f x* $*$ *c*)
by (*rule continuous-on-mult* [*OF - continuous-on-const*])

lemma *continuous-on-mult-const* [*simp*]:
fixes *c*::*'a*::*real-normed-algebra*
shows *continuous-on s* (($*$) *c*)
by (*intro continuous-on-mult-left continuous-on-id*)

lemma *tendsto-divide-zero*:
fixes *c* :: *'a*::*real-normed-field*
shows (*f* $\longrightarrow$ *0*) *F* $\Longrightarrow$ (($\lambda x.$ *f x* / *c*) $\longrightarrow$ *0*) *F*
by (*cases c=0*) (*simp-all add*: *divide-inverse tendsto-mult-left-zero*)

lemma *tendsto-power* [*tendsto-intros*]: (*f* $\longrightarrow$ *a*) *F* $\Longrightarrow$ (($\lambda x.$ *f x* ^ *n*) $\longrightarrow$ *a* ^ *n*) *F*
for *f* :: *'a* $\Rightarrow$ *'b*::{*power*,*real-normed-algebra*}
by (*induct n*) (*simp-all add*: *tendsto-mult*)

lemma *tendsto-null-power*: $[\![$(*f* $\longrightarrow$ *0*) *F*; *0* < *n*$]\!]$ $\Longrightarrow$ (($\lambda x.$ *f x* ^ *n*) $\longrightarrow$ *0*) *F*
for *f* :: *'a* $\Rightarrow$ *'b*::{*power*,*real-normed-algebra-1*}
using *tendsto-power* [*of f 0 F n*] **by** (*simp add*: *power-0-left*)

lemma *continuous-power* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* ($\lambda x.$ (*f x*)^*n*)
for *f* :: *'a*::*t2-space* $\Rightarrow$ *'b*::{*power*,*real-normed-algebra*}
unfolding *continuous-def* **by** (*rule tendsto-power*)

lemma *continuous-on-power* [*continuous-intros*]:
fixes *f* :: *-* $\Rightarrow$ *'b*::{*power*,*real-normed-algebra*}
shows *continuous-on s f* $\Longrightarrow$ *continuous-on s* ($\lambda x.$ (*f x*)^*n*)
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-power*)

lemma *tendsto-prod* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow (f\ i \longrightarrow L\ i)\ F) \Longrightarrow ((\lambda x.\ \prod i{\in}S.\ f\ i\ x) \longrightarrow (\prod i{\in}S.\ L\ i))\ F$
by (*induct S rule*: *infinite-finite-induct*) (*simp-all add*: *tendsto-mult*)

lemma *continuous-prod* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b$::*t2-space* $\Rightarrow 'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow$ *continuous* $F\ (f\ i)) \Longrightarrow$ *continuous* $F\ (\lambda x.\ \prod i{\in}S.\ f\ i\ x)$
unfolding *continuous-def* **by** (*rule tendsto-prod*)

lemma *continuous-on-prod* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow$ - $\Rightarrow 'c$::{*real-normed-algebra*,*comm-ring-1*}
shows $(\bigwedge i.\ i \in S \Longrightarrow$ *continuous-on* $s\ (f\ i)) \Longrightarrow$ *continuous-on* $s\ (\lambda x.\ \prod i{\in}S.\ f\ i\ x)$
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-prod*)

lemma *tendsto-of-real-iff*:
$((\lambda x.$ *of-real* $(f\ x) :: 'a$::*real-normed-div-algebra*$) \longrightarrow$ *of-real* $c)\ F \longleftrightarrow (f \longrightarrow c)\ F$
unfolding *tendsto-iff* **by** *simp*

lemma *tendsto-add-const-iff*:
$((\lambda x.\ c + f\ x :: 'a$::*topological-group-add*$) \longrightarrow c + d)\ F \longleftrightarrow (f \longrightarrow d)\ F$
using *tendsto-add*[*OF tendsto-const*[*of c*], *of f d*]
and *tendsto-add*[*OF tendsto-const*[*of* $-c$], *of* $\lambda x.\ c + f\ x\ c + d$] **by** *auto*

class *topological-monoid-mult* = *topological-semigroup-mult* + *monoid-mult*
class *topological-comm-monoid-mult* = *topological-monoid-mult* + *comm-monoid-mult*

lemma *tendsto-power-strong* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow 'b ::$ *topological-monoid-mult*
assumes $(f \longrightarrow a)\ F\ (g \longrightarrow b)\ F$
shows $((\lambda x.\ f\ x\ \hat{}\ g\ x) \longrightarrow a\ \hat{}\ b)\ F$
proof −
have $((\lambda x.\ f\ x\ \hat{}\ b) \longrightarrow a\ \hat{}\ b)\ F$
by (*induction b*) (*auto intro*: *tendsto-intros assms*)
also from *assms*(*2*) **have** *eventually* $(\lambda x.\ g\ x = b)\ F$
by (*simp add*: *nhds-discrete filterlim-principal*)
hence *eventually* $(\lambda x.\ f\ x\ \hat{}\ b = f\ x\ \hat{}\ g\ x)\ F$
by *eventually-elim simp*
hence $((\lambda x.\ f\ x\ \hat{}\ b) \longrightarrow a\ \hat{}\ b)\ F \longleftrightarrow ((\lambda x.\ f\ x\ \hat{}\ g\ x) \longrightarrow a\ \hat{}\ b)\ F$
by (*intro filterlim-cong refl*)
finally show *?thesis* **.**
qed

lemma *continuous-mult'* [*continuous-intros*]:

fixes *f g* :: - ⇒ *'b*::*topological-semigroup-mult*
shows *continuous F f* ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. *f x* ∗ *g x*)
unfolding *continuous-def* **by** (*rule tendsto-mult*)

lemma *continuous-power'* [*continuous-intros*]:
fixes *f* :: - ⇒ *'b*::*topological-monoid-mult*
shows *continuous F f* ⟹ *continuous F g* ⟹ *continuous F* (λ*x*. *f x* ^ *g x*)
unfolding *continuous-def* **by** (*rule tendsto-power-strong*) *auto*

lemma *continuous-on-mult'* [*continuous-intros*]:
fixes *f g* :: - ⇒ *'b*::*topological-semigroup-mult*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *f x* ∗ *g x*)
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-mult*)

lemma *continuous-on-power'* [*continuous-intros*]:
fixes *f* :: - ⇒ *'b*::*topological-monoid-mult*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *f x* ^ *g x*)
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-power-strong*)

lemma *tendsto-mult-one*:
fixes *f g* :: - ⇒ *'b*::*topological-monoid-mult*
shows (*f* ⟶ *1*) *F* ⟹ (*g* ⟶ *1*) *F* ⟹ ((λ*x*. *f x* ∗ *g x*) ⟶ *1*) *F*
by (*drule* (*1*) *tendsto-mult*) *simp*

lemma *tendsto-prod'* [*tendsto-intros*]:
fixes *f* :: *'a* ⇒ *'b* ⇒ *'c*::*topological-comm-monoid-mult*
shows (⋀*i*. *i* ∈ *I* ⟹ (*f i* ⟶ *a i*) *F*) ⟹ ((λ*x*. ∏ *i*∈*I*. *f i x*) ⟶ (∏ *i*∈*I*. *a i*)) *F*
by (*induct I rule*: *infinite-finite-induct*) (*simp-all add*: *tendsto-mult*)

lemma *tendsto-one-prod'*:
fixes *f* :: *'a* ⇒ *'b* ⇒ *'c*::*topological-comm-monoid-mult*
assumes ⋀*i*. *i* ∈ *I* ⟹ ((λ*x*. *f x i*) ⟶ *1*) *F*
shows ((λ*i*. *prod* (*f i*) *I*) ⟶ *1*) *F*
using *tendsto-prod'* [*of I* λ*x y*. *f y x* λ*x*. *1*] *assms* **by** *simp*

lemma *LIMSEQ-prod-0*:
fixes *f* :: *nat* ⇒ *'a*::{*semidom*,*topological-space*}
assumes *f i* = *0*
shows (λ*n*. *prod f* {..*n*}) ⟶ *0*
proof (*subst tendsto-cong*)
show ∀ *F n in sequentially*. *prod f* {..*n*} = *0*
using *assms eventually-at-top-linorder* **by** *auto*
qed *auto*

lemma *LIMSEQ-prod-nonneg*:
fixes *f* :: *nat* ⇒ *'a*::{*linordered-semidom*,*linorder-topology*}

assumes *0*: $\bigwedge n.\ 0 \leq f\ n$ **and** *a*: $(\lambda n.\ prod\ f\ \{..n\}) \longrightarrow a$
shows $a \geq 0$
by (*simp add*: *0 prod-nonneg LIMSEQ-le-const* [*OF a*])

lemma *continuous-prod'* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b$::*t2-space* $\Rightarrow 'c$::*topological-comm-monoid-mult*
shows $(\bigwedge i.\ i \in I \Longrightarrow continuous\ F\ (f\ i)) \Longrightarrow continuous\ F\ (\lambda x.\ \prod i{\in}I.\ f\ i\ x)$
unfolding *continuous-def* **by** (*rule tendsto-prod'*)

lemma *continuous-on-prod'* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'b$::*topological-space* $\Rightarrow 'c$::*topological-comm-monoid-mult*
shows $(\bigwedge i.\ i \in I \Longrightarrow continuous\text{-}on\ S\ (f\ i)) \Longrightarrow continuous\text{-}on\ S\ (\lambda x.\ \prod i{\in}I.\ f\ i\ x)$
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-prod'*)

instance *nat* :: *topological-comm-monoid-mult*
by *standard*
(*simp add*: *nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

instance *int* :: *topological-comm-monoid-mult*
by *standard*
(*simp add*: *nhds-discrete principal-prod-principal filterlim-principal eventually-principal*)

class *comm-real-normed-algebra-1* = *real-normed-algebra-1* + *comm-monoid-mult*

context *real-normed-field*
begin

subclass *comm-real-normed-algebra-1*
proof
from *norm-mult*[*of 1* :: *'a 1*] **show** *norm 1* = *1* **by** *simp*
qed (*simp-all add*: *norm-mult*)

end

108.4.3 Inverse and division

lemma (**in** *bounded-bilinear*) *Zfun-prod-Bfun*:
assumes *f*: *Zfun f F*
and *g*: *Bfun g F*
shows *Zfun* $(\lambda x.\ f\ x\ **\ g\ x)$ *F*
proof −
obtain *K* **where** *K*: $0 \leq K$
and *norm-le*: $\bigwedge x\ y.\ norm\ (x ** y) \leq norm\ x * norm\ y * K$
using *nonneg-bounded* **by** *blast*
obtain *B* **where** *B*: $0 < B$
and *norm-g*: *eventually* $(\lambda x.\ norm\ (g\ x) \leq B)\ F$
using *g* **by** (*rule BfunE*)
have *eventually* $(\lambda x.\ norm\ (f\ x\ **\ g\ x) \leq norm\ (f\ x) * (B * K))\ F$

```
  using norm-g proof eventually-elim
    case (elim x)
    have norm (f x ** g x) ≤ norm (f x) * norm (g x) * K
      by (rule norm-le)
    also have ... ≤ norm (f x) * B * K
      by (intro mult-mono' order-refl norm-g norm-ge-zero mult-nonneg-nonneg K
elim)
    also have ... = norm (f x) * (B * K)
      by (rule mult.assoc)
    finally show norm (f x ** g x) ≤ norm (f x) * (B * K) .
  qed
  with f show ?thesis
    by (rule Zfun-imp-Zfun)
qed

lemma (in bounded-bilinear) Bfun-prod-Zfun:
  assumes f: Bfun f F
    and g: Zfun g F
  shows Zfun (λx. f x ** g x) F
  using flip g f by (rule bounded-bilinear.Zfun-prod-Bfun)

lemma Bfun-inverse:
  fixes a :: 'a::real-normed-div-algebra
  assumes f: (f ⟶ a) F
  assumes a: a ≠ 0
  shows Bfun (λx. inverse (f x)) F
proof -
  from a have 0 < norm a by simp
  then have ∃r>0. r < norm a by (rule dense)
  then obtain r where r1: 0 < r and r2: r < norm a
    by blast
  have eventually (λx. dist (f x) a < r) F
    using tendstoD [OF f r1] by blast
  then have eventually (λx. norm (inverse (f x)) ≤ inverse (norm a - r)) F
  proof eventually-elim
    case (elim x)
    then have 1: norm (f x - a) < r
      by (simp add: dist-norm)
    then have 2: f x ≠ 0 using r2 by auto
    then have norm (inverse (f x)) = inverse (norm (f x))
      by (rule nonzero-norm-inverse)
    also have ... ≤ inverse (norm a - r)
    proof (rule le-imp-inverse-le)
      show 0 < norm a - r
        using r2 by simp
      have norm a - norm (f x) ≤ norm (a - f x)
        by (rule norm-triangle-ineq2)
      also have ... = norm (f x - a)
        by (rule norm-minus-commute)
```

```
    also have ... < r using 1 .
    finally show norm a - r ≤ norm (f x)
      by simp
  qed
  finally show norm (inverse (f x)) ≤ inverse (norm a - r) .
 qed
 then show ?thesis by (rule BfunI)
qed

lemma tendsto-inverse [tendsto-intros]:
 fixes a :: 'a::real-normed-div-algebra
 assumes f: (f ⟶ a) F
  and a: a ≠ 0
 shows ((λx. inverse (f x)) ⟶ inverse a) F
proof -
 from a have 0 < norm a by simp
 with f have eventually (λx. dist (f x) a < norm a) F
  by (rule tendstoD)
 then have eventually (λx. f x ≠ 0) F
  unfolding dist-norm by (auto elim!: eventually-mono)
 with a have eventually (λx. inverse (f x) - inverse a =
  - (inverse (f x) * (f x - a) * inverse a)) F
  by (auto elim!: eventually-mono simp: inverse-diff-inverse)
 moreover have Zfun (λx. - (inverse (f x) * (f x - a) * inverse a)) F
  by (intro Zfun-minus Zfun-mult-left
    bounded-bilinear.Bfun-prod-Zfun [OF bounded-bilinear-mult]
    Bfun-inverse [OF f a] f [unfolded tendsto-Zfun-iff])
 ultimately show ?thesis
  unfolding tendsto-Zfun-iff by (rule Zfun-ssubst)
qed

lemma continuous-inverse:
 fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
 assumes continuous F f
  and f (Lim F (λx. x)) ≠ 0
 shows continuous F (λx. inverse (f x))
 using assms unfolding continuous-def by (rule tendsto-inverse)

lemma continuous-at-within-inverse[continuous-intros]:
 fixes f :: 'a::t2-space ⇒ 'b::real-normed-div-algebra
 assumes continuous (at a within s) f
  and f a ≠ 0
 shows continuous (at a within s) (λx. inverse (f x))
 using assms unfolding continuous-within by (rule tendsto-inverse)

lemma continuous-on-inverse[continuous-intros]:
 fixes f :: 'a::topological-space ⇒ 'b::real-normed-div-algebra
 assumes continuous-on s f
  and ∀x∈s. f x ≠ 0
```

shows *continuous-on s* (λ*x*. *inverse* (*f x*))
using *assms* **unfolding** *continuous-on-def* **by** (*blast intro*: *tendsto-inverse*)

lemma *tendsto-divide* [*tendsto-intros*]:
fixes *a b* :: *'a*::*real-normed-div-algebra*
shows (*f* ⟶ *a*) *F* ⟹ (*g* ⟶ *b*) *F* ⟹ *b* ≠ *0* ⟹ ((λ*x*. *f x* / *g x*) ⟶ *a* / *b*) *F*
by (*simp add*: *tendsto-mult tendsto-inverse divide-inverse*)

lemma *continuous-divide*:
fixes *f g* :: *'a*::*t2-space* ⇒ *'b*::*real-normed-field*
assumes *continuous F f*
and *continuous F g*
and *g* (*Lim F* (λ*x*. *x*)) ≠ *0*
shows *continuous F* (λ*x*. (*f x*) / (*g x*))
using *assms* **unfolding** *continuous-def* **by** (*rule tendsto-divide*)

lemma *continuous-at-within-divide*[*continuous-intros*]:
fixes *f g* :: *'a*::*t2-space* ⇒ *'b*::*real-normed-field*
assumes *continuous* (*at a within s*) *f continuous* (*at a within s*) *g*
and *g a* ≠ *0*
shows *continuous* (*at a within s*) (λ*x*. (*f x*) / (*g x*))
using *assms* **unfolding** *continuous-within* **by** (*rule tendsto-divide*)

lemma *isCont-divide*[*continuous-intros*, *simp*]:
fixes *f g* :: *'a*::*t2-space* ⇒ *'b*::*real-normed-field*
assumes *isCont f a isCont g a g a* ≠ *0*
shows *isCont* (λ*x*. (*f x*) / *g x*) *a*
using *assms* **unfolding** *continuous-at* **by** (*rule tendsto-divide*)

lemma *continuous-on-divide*[*continuous-intros*]:
fixes *f* :: *'a*::*topological-space* ⇒ *'b*::*real-normed-field*
assumes *continuous-on s f continuous-on s g*
and ∀*x*∈*s*. *g x* ≠ *0*
shows *continuous-on s* (λ*x*. (*f x*) / (*g x*))
using *assms* **unfolding** *continuous-on-def* **by** (*blast intro*: *tendsto-divide*)

lemma *continuous-cmult-left-iff*:
fixes *c*::*'a*::*real-normed-field*
assumes *c* ≠ *0*
shows *continuous F* (λ*x*. *c* ∗ *f x*) ⟷ *continuous F f*
by (*simp add*: *assms continuous-def*)

lemma *continuous-cmult-right-iff*:
fixes *c*::*'a*::*real-normed-field*
assumes *c* ≠ *0*
shows *continuous F* (λ*x*. *f x* ∗ *c*) ⟷ *continuous F f*
by (*simp add*: *assms continuous-def*)

lemma *continuous-cdivide-iff*:
fixes *c*::*'a*::*real-normed-field*
assumes $c \neq 0$
shows *continuous F* ($\lambda x.$ *f x / c*) $\longleftrightarrow$ *continuous F f*
using *assms* **by** (*auto simp*: *continuous-def divide-inverse*)

lemma *continuous-cong*:
assumes *eventually* ($\lambda x.$ *f x* = *g x*) *F f* (*Lim F* ($\lambda x.$ *x*)) = *g* (*Lim F* ($\lambda x.$ *x*))
shows *continuous F f* $\longleftrightarrow$ *continuous F g*
unfolding *continuous-def* **using** *assms filterlim-cong* **by** *force*

lemma *continuous-at-within-cong*:
assumes *f x* = *g x eventually* ($\lambda x.$ *f x* = *g x*) (*at x within S*)
shows *continuous* (*at x within S*) *f* $\longleftrightarrow$ *continuous* (*at x within S*) *g*
using *assms* **by** (*simp add*: *continuous-within filterlim-cong*)

lemma *tendsto-power-int* [*tendsto-intros*]:
fixes *a* :: *'a*::*real-normed-div-algebra*
assumes *f*: (*f* $\longrightarrow$ *a*) *F*
and *a*: $a \neq 0$
shows (($\lambda x.$ *power-int* (*f x*) *n*) $\longrightarrow$ *power-int a n*) *F*
using *assms* **by** (*cases n rule*: *int-cases4*) (*auto intro*!: *tendsto-intros simp*: *power-int-minus*)

lemma *continuous-power-int*:
fixes *f* :: *'a*::*t2-space* $\Rightarrow$ *'b*::*real-normed-div-algebra*
assumes *continuous F f*
and *f* (*Lim F* ($\lambda x.$ *x*)) $\neq 0$
shows *continuous F* ($\lambda x.$ *power-int* (*f x*) *n*)
using *assms* **unfolding** *continuous-def* **by** (*rule tendsto-power-int*)

lemma *continuous-at-within-power-int*[*continuous-intros*]:
fixes *f* :: *'a*::*t2-space* $\Rightarrow$ *'b*::*real-normed-div-algebra*
assumes *continuous* (*at a within s*) *f*
and *f a* $\neq 0$
shows *continuous* (*at a within s*) ($\lambda x.$ *power-int* (*f x*) *n*)
using *assms* **unfolding** *continuous-within* **by** (*rule tendsto-power-int*)

lemma *continuous-on-power-int* [*continuous-intros*]:
fixes *f* :: *'a*::*topological-space* $\Rightarrow$ *'b*::*real-normed-div-algebra*
assumes *continuous-on s f* **and** $n \geq 0 \vee (\forall x{\in}s.\ f\ x \neq 0)$
shows *continuous-on s* ($\lambda x.$ *power-int* (*f x*) *n*)
using *assms* **by** (*cases* $n \geq 0$) (*auto simp*: *power-int-def intro*!: *continuous-intros*)

lemma *tendsto-power-int'* [*tendsto-intros*]:
fixes *a* :: *'a*::*real-normed-div-algebra*
assumes *f*: (*f* $\longrightarrow$ *a*) *F*
and *a*: $a \neq 0 \vee n \geq 0$
shows (($\lambda x.$ *power-int* (*f x*) *n*) $\longrightarrow$ *power-int a n*) *F*

using *assms* **by** (*cases n rule*: *int-cases4*) (*auto intro*!: *tendsto-intros simp*: *power-int-minus*)

lemma *tendsto-sgn* [*tendsto-intros*]: $(f \longrightarrow l)\ F \Longrightarrow l \neq 0 \Longrightarrow ((\lambda x.\ sgn\ (f\ x)) \longrightarrow sgn\ l)\ F$
for $l :: 'a$::*real-normed-vector*
unfolding *sgn-div-norm* **by** (*simp add*: *tendsto-intros*)

lemma *continuous-sgn*:
fixes $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-vector*
assumes *continuous F f*
and $f\ (Lim\ F\ (\lambda x.\ x)) \neq 0$
shows *continuous F* $(\lambda x.\ sgn\ (f\ x))$
using *assms* **unfolding** *continuous-def* **by** (*rule tendsto-sgn*)

lemma *continuous-at-within-sgn*[*continuous-intros*]:
fixes $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-vector*
assumes *continuous* (*at a within s*) *f*
and $f\ a \neq 0$
shows *continuous* (*at a within s*) $(\lambda x.\ sgn\ (f\ x))$
using *assms* **unfolding** *continuous-within* **by** (*rule tendsto-sgn*)

lemma *isCont-sgn*[*continuous-intros*]:
fixes $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-vector*
assumes *isCont f a*
and $f\ a \neq 0$
shows *isCont* $(\lambda x.\ sgn\ (f\ x))\ a$
using *assms* **unfolding** *continuous-at* **by** (*rule tendsto-sgn*)

lemma *continuous-on-sgn*[*continuous-intros*]:
fixes $f :: 'a$::*topological-space* $\Rightarrow$ $'b$::*real-normed-vector*
assumes *continuous-on s f*
and $\forall x{\in}s.\ f\ x \neq 0$
shows *continuous-on s* $(\lambda x.\ sgn\ (f\ x))$
using *assms* **unfolding** *continuous-on-def* **by** (*blast intro*: *tendsto-sgn*)

lemma *filterlim-at-infinity*:
fixes $f :: - \Rightarrow 'a$::*real-normed-vector*
assumes $0 \leq c$
shows $(LIM\ x\ F.\ f\ x :> at\text{-}infinity) \longleftrightarrow (\forall r{>}c.\ eventually\ (\lambda x.\ r \leq norm\ (f\ x))\ F)$
unfolding *filterlim-iff eventually-at-infinity*
proof *safe*
fix $P :: 'a \Rightarrow bool$
fix b
assume $*$: $\forall r{>}c.\ eventually\ (\lambda x.\ r \leq norm\ (f\ x))\ F$
assume P: $\forall x.\ b \leq norm\ x \longrightarrow P\ x$
have $max\ b\ (c + 1) > c$ **by** *auto*
with $*$ **have** $eventually\ (\lambda x.\ max\ b\ (c + 1) \leq norm\ (f\ x))\ F$

```
    by auto
  then show eventually (λx. P (f x)) F
  proof eventually-elim
    case (elim x)
    with P show P (f x) by auto
  qed
qed force

lemma filterlim-at-infinity-imp-norm-at-top:
  fixes F
  assumes filterlim f at-infinity F
  shows   filterlim (λx. norm (f x)) at-top F
proof −
  {
    fix r :: real
    have ∀F x in F. r ≤ norm (f x) using filterlim-at-infinity[of 0 f F] assms
      by (cases r > 0)
       (auto simp: not-less intro: always-eventually order.trans[OF - norm-ge-zero])
  }
  thus ?thesis by (auto simp: filterlim-at-top)
qed

lemma filterlim-norm-at-top-imp-at-infinity:
  fixes F
  assumes filterlim (λx. norm (f x)) at-top F
  shows   filterlim f at-infinity F
  using filterlim-at-infinity[of 0 f F] assms by (auto simp: filterlim-at-top)

lemma filterlim-norm-at-top: filterlim norm at-top at-infinity
  by (rule filterlim-at-infinity-imp-norm-at-top) (rule filterlim-ident)

lemma filterlim-at-infinity-conv-norm-at-top:
  filterlim f at-infinity G ⟷ filterlim (λx. norm (f x)) at-top G
  by (auto simp: filterlim-at-infinity[OF order.refl] filterlim-at-top-gt[of - - 0])

lemma eventually-not-equal-at-infinity:
  eventually (λx. x ≠ (a :: 'a :: {real-normed-vector})) at-infinity
proof −
  from filterlim-norm-at-top[where 'a = 'a]
   have ∀F x in at-infinity. norm a < norm (x::'a) by (auto simp: filterlim-at-top-dense)
  thus ?thesis by eventually-elim auto
qed

lemma filterlim-int-of-nat-at-topD:
  fixes F
  assumes filterlim (λx. f (int x)) F at-top
  shows   filterlim f F at-top
proof −
  have filterlim (λx. f (int (nat x))) F at-top
```

```
    by (rule filterlim-compose[OF assms filterlim-nat-sequentially])
  also have ?this ⟷ filterlim f F at-top
    by (intro filterlim-cong refl eventually-mono [OF eventually-ge-at-top[of 0::int]])
auto
  finally show ?thesis .
qed

lemma filterlim-int-sequentially [tendsto-intros]:
  filterlim int at-top sequentially
  unfolding filterlim-at-top
proof
  fix C :: int
  show eventually (λn. int n ≥ C) at-top
    using eventually-ge-at-top[of nat ⌈C⌉] by eventually-elim linarith
qed

lemma filterlim-real-of-int-at-top [tendsto-intros]:
  filterlim real-of-int at-top at-top
  unfolding filterlim-at-top
proof
  fix C :: real
  show eventually (λn. real-of-int n ≥ C) at-top
    using eventually-ge-at-top[of ⌈C⌉] by eventually-elim linarith
qed

lemma filterlim-abs-real: filterlim (abs::real ⇒ real) at-top at-top
proof (subst filterlim-cong[OF refl refl])
  from eventually-ge-at-top[of 0::real] show eventually (λx::real. |x| = x) at-top
    by eventually-elim simp
qed (simp-all add: filterlim-ident)

lemma filterlim-of-real-at-infinity [tendsto-intros]:
  filterlim (of-real :: real ⇒ 'a :: real-normed-algebra-1) at-infinity at-top
  by (intro filterlim-norm-at-top-imp-at-infinity) (auto simp: filterlim-abs-real)

lemma not-tendsto-and-filterlim-at-infinity:
  fixes c :: 'a::real-normed-vector
  assumes F ≠ bot
    and (f ⟶ c) F
    and filterlim f at-infinity F
  shows False
proof −
  from tendstoD[OF assms(2), of 1/2]
  have eventually (λx. dist (f x) c < 1/2) F
    by simp
  moreover
  from filterlim-at-infinity[of norm c f F] assms(3)
  have eventually (λx. norm (f x) ≥ norm c + 1) F by simp
  ultimately have eventually (λx. False) F
```

proof *eventually-elim*
fix *x*
assume *A*: *dist* (*f x*) *c* < *1*/*2*
assume *norm* (*f x*) ≥ *norm c* + *1*
also have *norm* (*f x*) = *dist* (*f x*) *0* **by** *simp*
also have ... ≤ *dist* (*f x*) *c* + *dist c 0* **by** (*rule dist-triangle*)
finally show *False* **using** *A* **by** *simp*
qed
with *assms* **show** *False* **by** *simp*
qed

lemma *filterlim-at-infinity-imp-not-convergent*:
assumes *filterlim f at-infinity sequentially*
shows ¬ *convergent f*
by (*rule notI*, *rule not-tendsto-and-filterlim-at-infinity*[*OF - - assms*])
(*simp-all add*: *convergent-LIMSEQ-iff*)

lemma *filterlim-at-infinity-imp-eventually-ne*:
assumes *filterlim f at-infinity F*
shows *eventually* (λ*z*. *f z* ≠ *c*) *F*
proof −
have *norm c* + *1* > *0*
by (*intro add-nonneg-pos*) *simp-all*
with *filterlim-at-infinity*[*OF order.refl*, *of f F*] *assms*
have *eventually* (λ*z*. *norm* (*f z*) ≥ *norm c* + *1*) *F*
by *blast*
then show *?thesis*
by *eventually-elim auto*
qed

lemma *tendsto-of-nat* [*tendsto-intros*]:
filterlim (*of-nat* :: *nat* ⇒ *'a*::*real-normed-algebra-1*) *at-infinity sequentially*
proof (*subst filterlim-at-infinity*[*OF order.refl*], *intro allI impI*)
fix *r* :: *real*
assume *r*: *r* > *0*
define *n* **where** *n* = *nat* ⌈*r*⌉
from *r* **have** *n*: ∀ *m*≥*n*. *of-nat m* ≥ *r*
unfolding *n-def* **by** *linarith*
from *eventually-ge-at-top*[*of n*] **show** *eventually* (λ*m*. *norm* (*of-nat m* :: *'a*) ≥ *r*) *sequentially*
by *eventually-elim* (*use n* **in** *simp-all*)
qed

108.5 Relate *at*, *at-left* and *at-right*

This lemmas are useful for conversion between *at x* to *at-left x* and *at-right x* and also *at-right 0*.

lemmas *filterlim-split-at-real* = *filterlim-split-at*[**where** *'a*=*real*]

lemma *filtermap-nhds-shift*: *filtermap* ($\lambda x.\ x - d$) (*nhds a*) = *nhds* (*a* − *d*)
for *a d* :: *'a*::*real-normed-vector*
by (*rule filtermap-fun-inverse*[**where** *g*=$\lambda x.\ x + d$])
(*auto intro*!: *tendsto-eq-intros filterlim-ident*)

lemma *filtermap-nhds-minus*: *filtermap* ($\lambda x.\ -x$) (*nhds a*) = *nhds* (− *a*)
for *a* :: *'a*::*real-normed-vector*
by (*rule filtermap-fun-inverse*[**where** *g*=*uminus*])
(*auto intro*!: *tendsto-eq-intros filterlim-ident*)

lemma *filtermap-at-shift*: *filtermap* ($\lambda x.\ x - d$) (*at a*) = *at* (*a* − *d*)
for *a d* :: *'a*::*real-normed-vector*
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift*[*symmetric*])

lemma *filtermap-at-right-shift*: *filtermap* ($\lambda x.\ x - d$) (*at-right a*) = *at-right* (*a* − *d*)
for *a d* :: *real*
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-shift*[*symmetric*])

lemma *filterlim-shift*:
fixes *d* :: *'a*::*real-normed-vector*
assumes *filterlim f F* (*at a*)
shows *filterlim* ($f \circ (+)\ d$) *F* (*at* (*a* − *d*))
unfolding *filterlim-iff*
proof (*intro strip*)
fix *P*
assume *eventually P F*
then have $\forall_F\ x$ *in filtermap* ($\lambda y.\ y - d$) (*at a*). *P* (*f* (*d* + *x*))
using *assms* **by** (*force simp add*: *filterlim-iff eventually-filtermap*)
then show ($\forall_F\ x$ *in at* (*a* − *d*). *P* (($f \circ (+)\ d$) *x*))
by (*force simp add*: *filtermap-at-shift*)
qed

lemma *filterlim-shift-iff*:
fixes *d* :: *'a*::*real-normed-vector*
shows *filterlim* ($f \circ (+)\ d$) *F* (*at* (*a* − *d*)) = *filterlim f F* (*at a*) (**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs* **show** *?rhs*
using *filterlim-shift* [*OF L*, *of* $-d$] **by** (*simp add*: *filterlim-iff*)
qed (*metis filterlim-shift*)

lemma *at-right-to-0*: *at-right a* = *filtermap* ($\lambda x.\ x + a$) (*at-right 0*)
for *a* :: *real*
using *filtermap-at-right-shift*[*of* $-a$ *0*] **by** *simp*

lemma *filterlim-at-right-to-0*:
filterlim f F (*at-right a*) $\longleftrightarrow$ *filterlim* ($\lambda x.\ f\ (x + a)$) *F* (*at-right 0*)
for *a* :: *real*

unfolding *filterlim-def filtermap-filtermap at-right-to-0*[*of a*] **..**

lemma *eventually-at-right-to-0*:
eventually P (*at-right a*) ⟷ *eventually* (λ*x*. *P* (*x* + *a*)) (*at-right 0*)
for *a* :: *real*
unfolding *at-right-to-0*[*of a*] **by** (*simp add*: *eventually-filtermap*)

lemma *at-to-0*: *at a* = *filtermap* (λ*x*. *x* + *a*) (*at 0*)
for *a* :: *'a*::*real-normed-vector*
using *filtermap-at-shift*[*of* −*a 0*] **by** *simp*

lemma *filterlim-at-to-0*:
filterlim f F (*at a*) ⟷ *filterlim* (λ*x*. *f* (*x* + *a*)) *F* (*at 0*)
for *a* :: *'a*::*real-normed-vector*
unfolding *filterlim-def filtermap-filtermap at-to-0*[*of a*] **..**

lemma *eventually-at-to-0*:
eventually P (*at a*) ⟷ *eventually* (λ*x*. *P* (*x* + *a*)) (*at 0*)
for *a* :: *'a*::*real-normed-vector*
unfolding *at-to-0*[*of a*] **by** (*simp add*: *eventually-filtermap*)

lemma *filtermap-at-minus*: *filtermap* (λ*x*. − *x*) (*at a*) = *at* (− *a*)
for *a* :: *'a*::*real-normed-vector*
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *at-left-minus*: *at-left a* = *filtermap* (λ*x*. − *x*) (*at-right* (− *a*))
for *a* :: *real*
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *at-right-minus*: *at-right a* = *filtermap* (λ*x*. − *x*) (*at-left* (− *a*))
for *a* :: *real*
by (*simp add*: *filter-eq-iff eventually-filtermap eventually-at-filter filtermap-nhds-minus*[*symmetric*])

lemma *filtermap-linear-at-within*:
assumes *bij f* **and** *cont*: *isCont f a* **and** *open-map*: ⋀*S*. *open S* ⟹ *open* (*f*‘*S*)
shows *filtermap f* (*at a within S*) = *at* (*f a*) *within f*‘*S*
unfolding *filter-eq-iff*
proof *safe*
fix *P*
assume *eventually P* (*filtermap f* (*at a within S*))
then obtain *T* **where** *open T a* ∈ *T* **and** *impP*:∀ *x*∈*T*. *x*≠*a* ⟶ *x*∈*S*⟶ *P* (*f* *x*)
by (*auto simp*: *eventually-filtermap eventually-at-topological*)
then show *eventually P* (*at* (*f a*) *within f* ‘ *S*)
unfolding *eventually-at-topological*
apply (*intro exI*[*of* - *f*‘*T*])
using ‹*bij f*› *open-map* **by** (*metis bij-pointE image-iff*)
next
fix *P*

assume *eventually P (at (f a) within f ' S)*
then obtain *T1* **where** *open T1 f a ∈ T1* **and** *impP:∀ x∈T1. x≠f a ⟶ x∈f'S⟶ P (x)*
unfolding *eventually-at-topological* **by** *auto*
then obtain *T2* **where** *open T2 a ∈ T2 (∀ x'∈T2. f x' ∈ T1)*
using *cont[unfolded continuous-at-open,rule-format,of T1]* **by** *blast*
then have *∀ x∈T2. x≠a ⟶ x∈S⟶ P (f x)*
using *impP* **by** *(metis assms(1) bij-pointE imageI)*
then show *eventually P (filtermap f (at a within S))*
unfolding *eventually-filtermap eventually-at-topological*
apply *(intro exI[of - T2])*
using *‹open T2› ‹a ∈ T2›* **by** *auto*
qed

lemma *filterlim-at-left-to-right*:
filterlim f F (at-left a) ⟷ filterlim (λx. f (− x)) F (at-right (−a))
for *a :: real*
unfolding *filterlim-def filtermap-filtermap at-left-minus[of a]* **..**

lemma *eventually-at-left-to-right*:
eventually P (at-left a) ⟷ eventually (λx. P (− x)) (at-right (−a))
for *a :: real*
unfolding *at-left-minus[of a]* **by** *(simp add: eventually-filtermap)*

lemma *filterlim-uminus-at-top-at-bot*: *LIM x at-bot. − x :: real :> at-top*
unfolding *filterlim-at-top eventually-at-bot-dense*
by *(metis leI minus-less-iff order-less-asym)*

lemma *filterlim-uminus-at-bot-at-top*: *LIM x at-top. − x :: real :> at-bot*
unfolding *filterlim-at-bot eventually-at-top-dense*
by *(metis leI less-minus-iff order-less-asym)*

lemma *at-bot-mirror* :
shows *(at-bot::('a::{ordered-ab-group-add,linorder} filter)) = filtermap uminus at-top*
proof *(rule filtermap-fun-inverse[symmetric])*
show *filterlim uminus at-top (at-bot::'a filter)*
using *eventually-at-bot-linorder filterlim-at-top le-minus-iff* **by** *force*
show *filterlim uminus (at-bot::'a filter) at-top*
by *(simp add: filterlim-at-bot minus-le-iff)*
qed *auto*

lemma *at-top-mirror* :
shows *(at-top::('a::{ordered-ab-group-add,linorder} filter)) = filtermap uminus at-bot*
apply *(subst at-bot-mirror)*
by *(auto simp: filtermap-filtermap)*

lemma *filterlim-at-top-mirror*: *(LIM x at-top. f x :> F) ⟷ (LIM x at-bot. f*

$(-x::real) :> F)$
unfolding *filterlim-def at-top-mirror filtermap-filtermap* **..**

lemma *filterlim-at-bot-mirror*: $(LIM\ x\ at\text{-}bot.\ f\ x :> F) \longleftrightarrow (LIM\ x\ at\text{-}top.\ f\ (-x::real) :> F)$
unfolding *filterlim-def at-bot-mirror filtermap-filtermap* **..**

lemma *filterlim-uminus-at-top*: $(LIM\ x\ F.\ f\ x :> at\text{-}top) \longleftrightarrow (LIM\ x\ F.\ -\ (f\ x) :: real :> at\text{-}bot)$
using *filterlim-compose*[*OF filterlim-uminus-at-bot-at-top*, *of f F*]
and *filterlim-compose*[*OF filterlim-uminus-at-top-at-bot*, *of* $\lambda x.\ -\ f\ x\ F$]
by *auto*

lemma *tendsto-at-botI-sequentially*:
fixes $f :: real \Rightarrow 'b::first\text{-}countable\text{-}topology$
assumes $*$: $\bigwedge X.\ filterlim\ X\ at\text{-}bot\ sequentially \Longrightarrow (\lambda n.\ f\ (X\ n)) \longrightarrow y$
shows $(f \longrightarrow y)\ at\text{-}bot$
unfolding *filterlim-at-bot-mirror*
proof (*rule tendsto-at-topI-sequentially*)
fix $X :: nat \Rightarrow real$ **assume** *filterlim X at-top sequentially*
thus $(\lambda n.\ f\ (-X\ n)) \longrightarrow y$ **by** (*intro* $*$) (*auto simp*: *filterlim-uminus-at-top*)
qed

lemma *filterlim-at-infinity-imp-filterlim-at-top*:
assumes $filterlim\ (f :: 'a \Rightarrow real)\ at\text{-}infinity\ F$
assumes $eventually\ (\lambda x.\ f\ x > 0)\ F$
shows *filterlim f at-top F*
proof −
from *assms*(*2*) **have** $*$: $eventually\ (\lambda x.\ norm\ (f\ x) = f\ x)\ F$ **by** *eventually-elim simp*
from *assms*(*1*) **show** *?thesis* **unfolding** *filterlim-at-infinity-conv-norm-at-top*
by (*subst* (*asm*) *filterlim-cong*[*OF refl refl* $*$])
qed

lemma *filterlim-at-infinity-imp-filterlim-at-bot*:
assumes $filterlim\ (f :: 'a \Rightarrow real)\ at\text{-}infinity\ F$
assumes $eventually\ (\lambda x.\ f\ x < 0)\ F$
shows *filterlim f at-bot F*
proof −
from *assms*(*2*) **have** $*$: $eventually\ (\lambda x.\ norm\ (f\ x) = -f\ x)\ F$ **by** *eventually-elim simp*
from *assms*(*1*) **have** $filterlim\ (\lambda x.\ -\ f\ x)\ at\text{-}top\ F$
unfolding *filterlim-at-infinity-conv-norm-at-top*
by (*subst* (*asm*) *filterlim-cong*[*OF refl refl* $*$])
thus *?thesis* **by** (*simp add*: *filterlim-uminus-at-top*)
qed

lemma *filterlim-uminus-at-bot*: $(LIM\ x\ F.\ f\ x :> at\text{-}bot) \longleftrightarrow (LIM\ x\ F.\ -\ (f\ x) :: real :> at\text{-}top)$

unfolding *filterlim-uminus-at-top* **by** *simp*

lemma *filterlim-inverse-at-top-right*: *LIM x at-right (0::real). inverse x :> at-top*
unfolding *filterlim-at-top-gt*[**where** *c=0*] *eventually-at-filter*
proof *safe*
fix *Z* :: *real*
assume [*arith*]: *0 < Z*
then have *eventually* (λ*x*. *x < inverse Z*) (*nhds 0*)
by (*auto simp*: *eventually-nhds-metric dist-real-def intro*!: *exI*[*of - |inverse Z|*])
then show *eventually* (λ*x*. *x ≠ 0 ⟶ x ∈ {0<..} ⟶ Z ≤ inverse x*) (*nhds 0*)
by (*auto elim*!: *eventually-mono simp*: *inverse-eq-divide field-simps*)
qed

lemma *tendsto-inverse-0*:
fixes *x* :: *- ⇒ 'a::real-normed-div-algebra*
shows (*inverse ⟶⟶* (*0::'a*)) *at-infinity*
unfolding *tendsto-Zfun-iff diff-0-right Zfun-def eventually-at-infinity*
proof *safe*
fix *r* :: *real*
assume *0 < r*
show ∃ *b*. ∀ *x*. *b ≤ norm x ⟶ norm* (*inverse x* :: *'a*) *< r*
proof (*intro exI*[*of - inverse* (*r / 2*)] *allI impI*)
fix *x* :: *'a*
from ‹*0 < r*› **have** *0 < inverse* (*r / 2*) **by** *simp*
also assume ∗: *inverse* (*r / 2*) ≤ *norm x*
finally show *norm* (*inverse x*) *< r*
using ∗ ‹*0 < r*›
by (*subst nonzero-norm-inverse*) (*simp-all add*: *inverse-eq-divide field-simps*)
qed
qed

lemma *tendsto-add-filterlim-at-infinity*:
fixes *c* :: *'b::real-normed-vector*
and *F* :: *'a filter*
assumes (*f ⟶⟶ c*) *F*
and *filterlim g at-infinity F*
shows *filterlim* (λ*x*. *f x + g x*) *at-infinity F*
proof (*subst filterlim-at-infinity*[*OF order-refl*], *safe*)
fix *r* :: *real*
assume *r*: *r > 0*
from *assms*(*1*) **have** ((λ*x*. *norm* (*f x*)) ⟶⟶ *norm c*) *F*
by (*rule tendsto-norm*)
then have *eventually* (λ*x*. *norm* (*f x*) *< norm c + 1*) *F*
by (*rule order-tendstoD*) *simp-all*
moreover from *r* **have** *r + norm c + 1 > 0*
by (*intro add-pos-nonneg*) *simp-all*
with *assms*(*2*) **have** *eventually* (λ*x*. *norm* (*g x*) ≥ *r + norm c + 1*) *F*
unfolding *filterlim-at-infinity*[*OF order-refl*]
by (*elim allE*[*of - r + norm c + 1*]) *simp-all*

```
  ultimately show eventually (λx. norm (f x + g x) ≥ r) F
  proof eventually-elim
    fix x :: 'a
    assume A: norm (f x) < norm c + 1 and B: r + norm c + 1 ≤ norm (g x)
    from A B have r ≤ norm (g x) − norm (f x)
      by simp
    also have norm (g x) − norm (f x) ≤ norm (g x + f x)
      by (rule norm-diff-ineq)
    finally show r ≤ norm (f x + g x)
      by (simp add: add-ac)
  qed
qed

lemma tendsto-add-filterlim-at-infinity':
  fixes c :: 'b::real-normed-vector
    and F :: 'a filter
  assumes filterlim f at-infinity F
    and (g ⟶ c) F
  shows filterlim (λx. f x + g x) at-infinity F
  by (subst add.commute) (rule tendsto-add-filterlim-at-infinity assms)+

lemma filterlim-inverse-at-right-top: LIM x at-top. inverse x :> at-right (0::real)
  unfolding filterlim-at
  by (auto simp: eventually-at-top-dense)
     (metis tendsto-inverse-0 filterlim-mono at-top-le-at-infinity order-refl)

lemma filterlim-inverse-at-top:
  (f ⟶ (0 :: real)) F ⟹ eventually (λx. 0 < f x) F ⟹ LIM x F. inverse (f
x) :> at-top
  by (intro filterlim-compose[OF filterlim-inverse-at-top-right])
       (simp add: filterlim-def eventually-filtermap eventually-mono at-within-def
le-principal)

lemma filterlim-inverse-at-bot-neg:
  LIM x (at-left (0::real)). inverse x :> at-bot
 by (simp add: filterlim-inverse-at-top-right filterlim-uminus-at-bot filterlim-at-left-to-right)

lemma filterlim-inverse-at-bot:
  (f ⟶ (0 :: real)) F ⟹ eventually (λx. f x < 0) F ⟹ LIM x F. inverse (f
x) :> at-bot
  unfolding filterlim-uminus-at-bot inverse-minus-eq[symmetric]
 by (rule filterlim-inverse-at-top) (simp-all add: tendsto-minus-cancel-left[symmetric])

lemma at-right-to-top: (at-right (0::real)) = filtermap inverse at-top
  by (intro filtermap-fun-inverse[symmetric, where g=inverse])
     (auto intro: filterlim-inverse-at-top-right filterlim-inverse-at-right-top)

lemma eventually-at-right-to-top:
  eventually P (at-right (0::real)) ⟷ eventually (λx. P (inverse x)) at-top
```

unfolding *at-right-to-top eventually-filtermap* **..**

lemma *filterlim-at-right-to-top*:
filterlim f F (*at-right* (*0*::*real*)) ⟷ (*LIM x at-top. f* (*inverse x*) :> *F*)
unfolding *filterlim-def at-right-to-top filtermap-filtermap* **..**

lemma *at-top-to-right*: *at-top* = *filtermap inverse* (*at-right* (*0*::*real*))
unfolding *at-right-to-top filtermap-filtermap inverse-inverse-eq filtermap-ident* **..**

lemma *eventually-at-top-to-right*:
eventually P at-top ⟷ *eventually* (λ*x*. *P* (*inverse x*)) (*at-right* (*0*::*real*))
unfolding *at-top-to-right eventually-filtermap* **..**

lemma *filterlim-at-top-to-right*:
filterlim f F at-top ⟷ (*LIM x* (*at-right* (*0*::*real*)). *f* (*inverse x*) :> *F*)
unfolding *filterlim-def at-top-to-right filtermap-filtermap* **..**

lemma *filterlim-inverse-at-infinity*:
fixes *x* :: - ⇒ *'a*::{*real-normed-div-algebra*, *division-ring*}
shows *filterlim inverse at-infinity* (*at* (*0*::*'a*))
unfolding *filterlim-at-infinity*[*OF order-refl*]
proof *safe*
fix *r* :: *real*
assume *0* < *r*
then show *eventually* (λ*x*::*'a*. *r* ≤ *norm* (*inverse x*)) (*at 0*)
unfolding *eventually-at norm-inverse*
by (*intro exI*[*of* - *inverse r*])
(*auto simp*: *norm-conv-dist*[*symmetric*] *field-simps inverse-eq-divide*)
qed

lemma *filterlim-inverse-at-iff*:
fixes *g* :: *'a* ⇒ *'b*::{*real-normed-div-algebra*, *division-ring*}
shows (*LIM x F*. *inverse* (*g x*) :> *at 0*) ⟷ (*LIM x F*. *g x* :> *at-infinity*)
unfolding *filterlim-def filtermap-filtermap*[*symmetric*]
proof
assume *filtermap g F* ≤ *at-infinity*
then have *filtermap inverse* (*filtermap g F*) ≤ *filtermap inverse at-infinity*
by (*rule filtermap-mono*)
also have . . . ≤ *at 0*
using *tendsto-inverse-0*[**where** *'a*=*'b*]
by (*auto intro*!: *exI*[*of* - *1*]
simp: *le-principal eventually-filtermap filterlim-def at-within-def eventually-at-infinity*)
finally show *filtermap inverse* (*filtermap g F*) ≤ *at 0* **.**
next
assume *filtermap inverse* (*filtermap g F*) ≤ *at 0*
then have *filtermap inverse* (*filtermap inverse* (*filtermap g F*)) ≤ *filtermap inverse* (*at 0*)
by (*rule filtermap-mono*)

with *filterlim-inverse-at-infinity* **show** *filtermap g F* $\leq$ *at-infinity*
by (*auto intro*: *order-trans simp*: *filterlim-def filtermap-filtermap*)
qed

lemma *tendsto-mult-filterlim-at-infinity*:
fixes *c* :: *'a*::*real-normed-field*
assumes (*f* $\longrightarrow$ *c*) *F c* $\neq$ *0*
assumes *filterlim g at-infinity F*
shows *filterlim* (λ*x*. *f x* $*$ *g x*) *at-infinity F*
proof −
have ((λ*x*. *inverse* (*f x*) $*$ *inverse* (*g x*)) $\longrightarrow$ *inverse c* $*$ *0*) *F*
by (*intro tendsto-mult tendsto-inverse assms filterlim-compose*[*OF tendsto-inverse-0*])
then have *filterlim* (λ*x*. *inverse* (*f x*) $*$ *inverse* (*g x*)) (*at* (*inverse c* $*$ *0*)) *F*
unfolding *filterlim-at*
using *assms*
by (*auto intro*: *filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne eventually-conj*)
then show *?thesis*
by (*subst filterlim-inverse-at-iff*[*symmetric*]) *simp-all*
qed

lemma *filterlim-power-int-neg-at-infinity*:
fixes *f* :: - $\Rightarrow$ *'a*::{*real-normed-div-algebra*, *division-ring*}
assumes *n* $<$ *0* **and** *lim*: (*f* $\longrightarrow$ *0*) *F* **and** *ev*: *eventually* (λ*x*. *f x* $\neq$ *0*) *F*
shows *filterlim* (λ*x*. *f x powi n*) *at-infinity F*
proof −
have *lim'*: ((λ*x*. *f x* ^ *nat* (− *n*)) $\longrightarrow$ *0*) *F*
by (*rule tendsto-eq-intros lim*)+ (*use* ‹*n* $<$ *0*› **in** *auto*)
have *ev'*: *eventually* (λ*x*. *f x* ^ *nat* (−*n*) $\neq$ *0*) *F*
using *ev* **by** *eventually-elim* (*use* ‹*n* $<$ *0*› **in** *auto*)
have *filterlim* (λ*x*. *inverse* (*f x* ^ *nat* (−*n*))) *at-infinity F*
by (*intro filterlim-compose*[*OF filterlim-inverse-at-infinity*])
(*use lim' ev'* **in** ‹*auto simp*: *filterlim-at*›)
thus *?thesis*
using ‹*n* $<$ *0*› **by** (*simp add*: *power-int-def power-inverse*)
qed

lemma *tendsto-inverse-0-at-top*: *LIM x F*. *f x* :> *at-top* $\Longrightarrow$ ((λ*x*. *inverse* (*f x*) :: *real*) $\longrightarrow$ *0*) *F*
by (*metis filterlim-at filterlim-mono*[*OF - at-top-le-at-infinity order-refl*] *filterlim-inverse-at-iff*)

lemma *filterlim-inverse-at-top-iff*:
eventually (λ*x*. *0* $<$ *f x*) *F* $\Longrightarrow$ (*LIM x F*. *inverse* (*f x*) :> *at-top*) $\longleftrightarrow$ (*f* $\longrightarrow$ (*0* :: *real*)) *F*
by (*auto dest*: *tendsto-inverse-0-at-top filterlim-inverse-at-top*)

lemma *filterlim-at-top-iff-inverse-0*:
eventually (λ*x*. *0* $<$ *f x*) *F* $\Longrightarrow$ (*LIM x F*. *f x* :> *at-top*) $\longleftrightarrow$ ((*inverse* $\circ$ *f*) $\longrightarrow$

```
(0 :: real)) F
  using filterlim-inverse-at-top-iff [of inverse ∘ f] by auto

lemma real-tendsto-divide-at-top:
  fixes c::real
  assumes (f ⟶ c) F
  assumes filterlim g at-top F
  shows ((λx. f x / g x) ⟶ 0) F
  by (auto simp: divide-inverse-commute
      intro!: tendsto-mult[THEN tendsto-eq-rhs] tendsto-inverse-0-at-top assms)

lemma mult-nat-left-at-top: c > 0 ⟹ filterlim (λx. c * x) at-top sequentially
  for c :: nat
  by (rule filterlim-subseq) (auto simp: strict-mono-def)

lemma mult-nat-right-at-top: c > 0 ⟹ filterlim (λx. x * c) at-top sequentially
  for c :: nat
  by (rule filterlim-subseq) (auto simp: strict-mono-def)

lemma filterlim-times-pos:
  LIM x F1. c * f x :> at-right l
  if filterlim f (at-right p) F1 0 < c l = c * p
  for c::'a::{linordered-field, linorder-topology}
  unfolding filterlim-iff
proof safe
  fix P
  assume ∀⇩F x in at-right l. P x
  then obtain d where c * p < d ⋀y. y > c * p ⟹ y < d ⟹ P y
    unfolding ‹l = -› eventually-at-right-field
    by auto
  then have ∀⇩F a in at-right p. P (c * a)
    by (auto simp: eventually-at-right-field ‹0 < c› field-simps intro!: exI[where
x=d/c])
  from that(1)[unfolded filterlim-iff, rule-format, OF this]
  show ∀⇩F x in F1. P (c * f x) .
qed

lemma filtermap-nhds-times: c ≠ 0 ⟹ filtermap (times c) (nhds a) = nhds (c *
a)
  for a c :: 'a::real-normed-field
  by (rule filtermap-fun-inverse[where g=λx. inverse c * x])
    (auto intro!: tendsto-eq-intros filterlim-ident)

lemma filtermap-times-pos-at-right:
  fixes c::'a::{linordered-field, linorder-topology}
  assumes c > 0
  shows filtermap (times c) (at-right p) = at-right (c * p)
  using assms
  by (intro filtermap-fun-inverse[where g=λx. inverse c * x])
```

(*auto intro*!: *filterlim-ident filterlim-times-pos*)

lemma *at-to-infinity*: (*at* (*0*::*'a*::{*real-normed-field*,*field*})) = *filtermap inverse at-infinity*
proof (*rule antisym*)
have (*inverse* ⟶ (*0*::*'a*)) *at-infinity*
by (*fact tendsto-inverse-0*)
then show *filtermap inverse at-infinity* ≤ *at* (*0*::*'a*)
using *filterlim-def filterlim-ident filterlim-inverse-at-iff* **by** *fastforce*
next
have *filtermap inverse* (*filtermap inverse* (*at* (*0*::*'a*))) ≤ *filtermap inverse at-infinity*
using *filterlim-inverse-at-infinity* **unfolding** *filterlim-def*
by (*rule filtermap-mono*)
then show *at* (*0*::*'a*) ≤ *filtermap inverse at-infinity*
by (*simp add*: *filtermap-ident filtermap-filtermap*)
qed

lemma *lim-at-infinity-0*:
fixes *l* :: *'a*::{*real-normed-field*,*field*}
shows (*f* ⟶ *l*) *at-infinity* ⟷ ((*f* ∘ *inverse*) ⟶ *l*) (*at* (*0*::*'a*))
by (*simp add*: *tendsto-compose-filtermap at-to-infinity filtermap-filtermap*)

lemma *lim-zero-infinity*:
fixes *l* :: *'a*::{*real-normed-field*,*field*}
shows ((λ*x*. *f*(*1* / *x*)) ⟶ *l*) (*at* (*0*::*'a*)) ⟹ (*f* ⟶ *l*) *at-infinity*
by (*simp add*: *inverse-eq-divide lim-at-infinity-0 comp-def*)

We only show rules for multiplication and addition when the functions are either against a real value or against infinity. Further rules are easy to derive by using *filterlim ?f at-top ?F* = (*LIM x ?F*. − *?f x* :> *at-bot*).

lemma *filterlim-tendsto-pos-mult-at-top*:
assumes *f*: (*f* ⟶ *c*) *F*
and *c*: *0* < *c*
and *g*: *LIM x F*. *g x* :> *at-top*
shows *LIM x F*. (*f x* ∗ *g x* :: *real*) :> *at-top*
unfolding *filterlim-at-top-gt*[**where** *c*=*0*]
proof *safe*
fix *Z* :: *real*
assume *0* < *Z*
from *f* ‹*0* < *c*› **have** *eventually* (λ*x*. *c* / *2* < *f x*) *F*
by (*auto dest*!: *tendstoD*[**where** *e*=*c* / *2*] *elim*!: *eventually-mono*
simp: *dist-real-def abs-real-def split*: *if-split-asm*)
moreover from *g* **have** *eventually* (λ*x*. (*Z* / *c* ∗ *2*) ≤ *g x*) *F*
unfolding *filterlim-at-top* **by** *auto*
ultimately show *eventually* (λ*x*. *Z* ≤ *f x* ∗ *g x*) *F*
proof *eventually-elim*
case (*elim x*)
with ‹*0* < *Z*› ‹*0* < *c*› **have** *c* / *2* ∗ (*Z* / *c* ∗ *2*) ≤ *f x* ∗ *g x*
by (*intro mult-mono*) (*auto simp*: *zero-le-divide-iff*)
with ‹*0* < *c*› **show** *Z* ≤ *f x* ∗ *g x*

```
      by simp
  qed
qed

lemma filterlim-at-top-mult-at-top:
  assumes f: LIM x F. f x :> at-top
    and g: LIM x F. g x :> at-top
  shows LIM x F. (f x * g x :: real) :> at-top
  unfolding filterlim-at-top-gt[where c=0]
proof safe
  fix Z :: real
  assume 0 < Z
  from f have eventually (λx. 1 ≤ f x) F
    unfolding filterlim-at-top by auto
  moreover from g have eventually (λx. Z ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x * g x) F
  proof eventually-elim
    case (elim x)
    with ‹0 < Z› have 1 * Z ≤ f x * g x
      by (intro mult-mono) (auto simp: zero-le-divide-iff)
    then show Z ≤ f x * g x
      by simp
  qed
qed

lemma filterlim-at-top-mult-tendsto-pos:
  assumes f: (f ⟶ c) F
    and c: 0 < c
    and g: LIM x F. g x :> at-top
  shows LIM x F. (g x * f x:: real) :> at-top
  by (auto simp: mult.commute intro!: filterlim-tendsto-pos-mult-at-top f c g)

lemma filterlim-tendsto-pos-mult-at-bot:
  fixes c :: real
  assumes (f ⟶ c) F 0 < c filterlim g at-bot F
  shows LIM x F. f x * g x :> at-bot
  using filterlim-tendsto-pos-mult-at-top[OF assms(1,2), of λx. - g x] assms(3)
  unfolding filterlim-uminus-at-bot by simp

lemma filterlim-tendsto-neg-mult-at-bot:
  fixes c :: real
  assumes c: (f ⟶ c) F c < 0 and g: filterlim g at-top F
  shows LIM x F. f x * g x :> at-bot
  using c filterlim-tendsto-pos-mult-at-top[of λx. - f x - c F, OF - - g]
  unfolding filterlim-uminus-at-bot tendsto-minus-cancel-left by simp


lemma filterlim-cmult-at-bot-at-top:
```

assumes *filterlim* $(h :: - \Rightarrow real)$ *at-top* F $c \neq 0$ $G = (if\ c > 0\ then\ at\text{-}top\ else\ at\text{-}bot)$
shows *filterlim* $(\lambda x.\ c * h\ x)$ G F
using *assms filterlim-tendsto-pos-mult-at-top*[*OF tendsto-const*[*of c*], *of h F*]
filterlim-tendsto-neg-mult-at-bot[*OF tendsto-const*[*of c*], *of h F*] **by** *simp*

lemma *filterlim-pow-at-top*:
fixes $f :: 'a \Rightarrow real$
assumes $0 < n$
and f: *LIM x F. f x :> at-top*
shows *LIM x F.* $(f\ x)\hat{}n :: real :>$ *at-top*
using ‹$0 < n$›
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*) **with** f **show** *?case*
by (*cases* $n = 0$) (*auto intro*!: *filterlim-at-top-mult-at-top*)
qed

lemma *filterlim-pow-at-bot-even*:
fixes $f :: real \Rightarrow real$
shows $0 < n \Longrightarrow$ *LIM x F. f x :> at-bot* $\Longrightarrow$ *even n* $\Longrightarrow$ *LIM x F.* $(f\ x)\hat{}n :>$ *at-top*
using *filterlim-pow-at-top*[*of n* $\lambda x. -f\ x$ *F*] **by** (*simp add*: *filterlim-uminus-at-top*)

lemma *filterlim-pow-at-bot-odd*:
fixes $f :: real \Rightarrow real$
shows $0 < n \Longrightarrow$ *LIM x F. f x :> at-bot* $\Longrightarrow$ *odd n* $\Longrightarrow$ *LIM x F.* $(f\ x)\hat{}n :>$ *at-bot*
using *filterlim-pow-at-top*[*of n* $\lambda x. -f\ x$ *F*] **by** (*simp add*: *filterlim-uminus-at-bot*)

lemma *filterlim-power-at-infinity* [*tendsto-intros*]:
fixes F **and** $f :: 'a \Rightarrow 'b :: real\text{-}normed\text{-}div\text{-}algebra$
assumes *filterlim f at-infinity F n* > 0
shows *filterlim* $(\lambda x.\ f\ x\ \hat{}\ n)$ *at-infinity F*
by (*rule filterlim-norm-at-top-imp-at-infinity*)
(*auto simp*: *norm-power intro*!: *filterlim-pow-at-top assms*
intro: *filterlim-at-infinity-imp-norm-at-top*)

lemma *filterlim-tendsto-add-at-top*:
assumes f: $(f \longrightarrow c)$ F
and g: *LIM x F. g x :> at-top*
shows *LIM x F.* $(f\ x + g\ x :: real) :>$ *at-top*
unfolding *filterlim-at-top-gt*[**where** *c*=*0*]
proof *safe*
fix $Z :: real$
assume $0 < Z$
from f **have** *eventually* $(\lambda x.\ c - 1 < f\ x)$ F

by (*auto dest*!: *tendstoD*[**where** *e=1*] *elim*!: *eventually-mono simp*: *dist-real-def*)
moreover from *g* **have** *eventually* (λ*x*. *Z* − (*c* − *1*) ≤ *g x*) *F*
unfolding *filterlim-at-top* **by** *auto*
ultimately show *eventually* (λ*x*. *Z* ≤ *f x* + *g x*) *F*
by *eventually-elim simp*
qed

lemma *filterlim-tendsto-add-at-top-iff*:
assumes *f*: (*f* ⟶ *c*) *F*
shows (*LIM x F*. (*f x* + *g x* :: *real*) :> *at-top*) ⟷ (*LIM x F*. *g x* :> *at-top*)
proof
assume *LIM x F*. *f x* + *g x* :> *at-top*
moreover have ((λ*x*. − *f x*) ⟶ − *c*) *F*
by (*simp add*: *f tendsto-minus*)
ultimately show *filterlim g at-top F*
using *filterlim-tendsto-add-at-top* **by** *fastforce*
qed (*auto simp*: *filterlim-tendsto-add-at-top*[*OF f*])

lemma *filterlim-tendsto-add-at-bot-iff*:
fixes *c*::*real*
assumes *f*: (*f* ⟶ *c*) *F*
shows (*LIM x F*. *f x* + *g x* :> *at-bot*) ⟷ (*LIM x F*. *g x* :> *at-bot*)
proof −
have (*LIM x F*. *f x* + *g x* :> *at-bot*)
⟷ (*LIM x F*. − *f x* + (− *g x*) :> *at-top*)
by (*simp add*: *filterlim-uminus-at-bot*)
also have ... = (*LIM x F*. − *g x* :> *at-top*)
by (*metis f filterlim-tendsto-add-at-top-iff tendsto-minus*)
also have ... = (*LIM x F*. *g x* :> *at-bot*)
by (*simp add*: *filterlim-uminus-at-bot*)
finally show *?thesis* **.**
qed

lemma *LIM-at-top-divide*:
fixes *f g* :: *'a* ⇒ *real*
assumes *f*: (*f* ⟶ *a*) *F* *0* < *a*
and *g*: (*g* ⟶ *0*) *F* *eventually* (λ*x*. *0* < *g x*) *F*
shows *LIM x F*. *f x* / *g x* :> *at-top*
unfolding *divide-inverse*
by (*rule filterlim-tendsto-pos-mult-at-top*[*OF f*]) (*rule filterlim-inverse-at-top*[*OF g*])

lemma *filterlim-at-top-add-at-top*:
assumes *f*: *LIM x F*. *f x* :> *at-top*
and *g*: *LIM x F*. *g x* :> *at-top*
shows *LIM x F*. (*f x* + *g x* :: *real*) :> *at-top*
unfolding *filterlim-at-top-gt*[**where** *c=0*]
proof *safe*
fix *Z* :: *real*

```
  assume 0 < Z
  from f have eventually (λx. 0 ≤ f x) F
    unfolding filterlim-at-top by auto
  moreover from g have eventually (λx. Z ≤ g x) F
    unfolding filterlim-at-top by auto
  ultimately show eventually (λx. Z ≤ f x + g x) F
    by eventually-elim simp
qed

lemma tendsto-divide-0:
  fixes f :: - ⇒ 'a::{real-normed-div-algebra, division-ring}
  assumes f: (f ⟶ c) F
    and g: LIM x F. g x :> at-infinity
  shows ((λx. f x / g x) ⟶ 0) F
  using tendsto-mult[OF f filterlim-compose[OF tendsto-inverse-0 g]]
  by (simp add: divide-inverse)

lemma linear-plus-1-le-power:
  fixes x :: real
  assumes x: 0 ≤ x
  shows real n * x + 1 ≤ (x + 1) ^ n
proof (induct n)
  case 0
  then show ?case by simp
next
  case (Suc n)
  from x have real (Suc n) * x + 1 ≤ (x + 1) * (real n * x + 1)
    by (simp add: field-simps)
  also have ... ≤ (x + 1)^Suc n
    using Suc x by (simp add: mult-left-mono)
  finally show ?case .
qed

lemma filterlim-realpow-sequentially-gt1:
  fixes x :: 'a :: real-normed-div-algebra
  assumes x[arith]: 1 < norm x
  shows LIM n sequentially. x ^ n :> at-infinity
proof (intro filterlim-at-infinity[THEN iffD2] allI impI)
  fix y :: real
  assume 0 < y
  obtain N :: nat where y < real N * (norm x - 1)
    by (meson diff-gt-0-iff-gt reals-Archimedean3 x)
  also have ... ≤ real N * (norm x - 1) + 1
    by simp
  also have ... ≤ (norm x - 1 + 1) ^ N
    by (rule linear-plus-1-le-power) simp
  also have ... = norm x ^ N
    by simp
  finally have ∀ n≥N. y ≤ norm x ^ n
```

by (*metis order-less-le-trans power-increasing order-less-imp-le x*)
then show *eventually* (λ*n*. *y* ≤ *norm* (*x* ^ *n*)) *sequentially*
unfolding *eventually-sequentially*
by (*auto simp*: *norm-power*)
qed *simp*

lemma *filterlim-divide-at-infinity*:
fixes *f g* :: *'a* ⇒ *'a* :: *real-normed-field*
assumes *filterlim f* (*nhds c*) *F filterlim g* (*at 0*) *F c* ≠ *0*
shows *filterlim* (λ*x*. *f x* / *g x*) *at-infinity F*
proof −
have *filterlim* (λ*x*. *f x* ∗ *inverse* (*g x*)) *at-infinity F*
by (*intro tendsto-mult-filterlim-at-infinity*[*OF assms*(*1*,*3*)]
filterlim-compose [*OF filterlim-inverse-at-infinity assms*(*2*)])
thus *?thesis* **by** (*simp add*: *field-simps*)
qed

108.6 Floor and Ceiling

lemma *eventually-floor-less*:
fixes *f* :: *'a* ⇒ *'b*::{*order-topology*,*floor-ceiling*}
assumes *f*: (*f* ⟶ *l*) *F*
and *l*: *l* ∉ ℤ
shows ∀ *F x in F*. *of-int* (*floor l*) < *f x*
by (*intro order-tendstoD*[*OF f*]) (*metis Ints-of-int antisym-conv2 floor-correct l*)

lemma *eventually-less-ceiling*:
fixes *f* :: *'a* ⇒ *'b*::{*order-topology*,*floor-ceiling*}
assumes *f*: (*f* ⟶ *l*) *F*
and *l*: *l* ∉ ℤ
shows ∀ *F x in F*. *f x* < *of-int* (*ceiling l*)
by (*intro order-tendstoD*[*OF f*]) (*metis Ints-of-int l le-of-int-ceiling less-le*)

lemma *eventually-floor-eq*:
fixes *f*::*'a* ⇒ *'b*::{*order-topology*,*floor-ceiling*}
assumes *f*: (*f* ⟶ *l*) *F*
and *l*: *l* ∉ ℤ
shows ∀ *F x in F*. *floor* (*f x*) = *floor l*
using *eventually-floor-less*[*OF assms*] *eventually-less-ceiling*[*OF assms*]
by *eventually-elim* (*meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq*)

lemma *eventually-ceiling-eq*:
fixes *f*::*'a* ⇒ *'b*::{*order-topology*,*floor-ceiling*}
assumes *f*: (*f* ⟶ *l*) *F*
and *l*: *l* ∉ ℤ
shows ∀ *F x in F*. *ceiling* (*f x*) = *ceiling l*
using *eventually-floor-less*[*OF assms*] *eventually-less-ceiling*[*OF assms*]
by *eventually-elim* (*meson floor-less-iff less-ceiling-iff not-less-iff-gr-or-eq*)

lemma *tendsto-of-int-floor*:
fixes f::$'a \Rightarrow 'b$::{*order-topology*,*floor-ceiling*}
assumes $(f \longrightarrow l)$ *F*
and $l \notin \mathbb{Z}$
shows $((\lambda x.\ of\text{-}int\ (floor\ (f\ x)) :: 'c$::{*ring-1*,*topological-space*}$) \longrightarrow of\text{-}int\ (floor\ l))$ *F*
using *eventually-floor-eq*[*OF assms*]
by (*simp add*: *eventually-mono topological-tendstoI*)

lemma *tendsto-of-int-ceiling*:
fixes f::$'a \Rightarrow 'b$::{*order-topology*,*floor-ceiling*}
assumes $(f \longrightarrow l)$ *F*
and $l \notin \mathbb{Z}$
shows $((\lambda x.\ of\text{-}int\ (ceiling\ (f\ x)) :: 'c$::{*ring-1*,*topological-space*}$) \longrightarrow of\text{-}int\ (ceiling\ l))$ *F*
using *eventually-ceiling-eq*[*OF assms*]
by (*simp add*: *eventually-mono topological-tendstoI*)

lemma *continuous-on-of-int-floor*:
continuous-on (*UNIV* − $\mathbb{Z}$::$'a$::{*order-topology*, *floor-ceiling*} *set*)
$(\lambda x.\ of\text{-}int\ (floor\ x)$::$'b$::{*ring-1*, *topological-space*})
unfolding *continuous-on-def*
by (*auto intro*!: *tendsto-of-int-floor*)

lemma *continuous-on-of-int-ceiling*:
continuous-on (*UNIV* − $\mathbb{Z}$::$'a$::{*order-topology*, *floor-ceiling*} *set*)
$(\lambda x.\ of\text{-}int\ (ceiling\ x)$::$'b$::{*ring-1*, *topological-space*})
unfolding *continuous-on-def*
by (*auto intro*!: *tendsto-of-int-ceiling*)

108.7 Limits of Sequences

lemma [*trans*]: $X = Y \Longrightarrow Y \longrightarrow z \Longrightarrow X \longrightarrow z$
by *simp*

lemma *LIMSEQ-iff*:
fixes L :: $'a$::*real-normed-vector*
shows $(X \longrightarrow L) = (\forall r{>}0.\ \exists no.\ \forall n \geq no.\ norm\ (X\ n - L) < r)$
unfolding *lim-sequentially dist-norm* **..**

lemma *LIMSEQ-I*: $(\bigwedge r.\ 0 < r \Longrightarrow \exists no.\ \forall n{\geq}no.\ norm\ (X\ n - L) < r) \Longrightarrow X \longrightarrow L$
for L :: $'a$::*real-normed-vector*
by (*simp add*: *LIMSEQ-iff*)

lemma *LIMSEQ-D*: $X \longrightarrow L \Longrightarrow 0 < r \Longrightarrow \exists no.\ \forall n{\geq}no.\ norm\ (X\ n - L) < r$
for L :: $'a$::*real-normed-vector*

by (*simp add*: *LIMSEQ-iff*)

lemma *LIMSEQ-linear*: $X \longrightarrow x \Longrightarrow l > 0 \Longrightarrow (\lambda\ n.\ X\ (n * l)) \longrightarrow x$
unfolding *tendsto-def eventually-sequentially*
by (*metis div-le-dividend div-mult-self1-is-m le-trans mult.commute*)

Transformation of limit.

lemma *Lim-transform*: $(g \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow a)\ F$
for *a b* :: *'a*::*real-normed-vector*
using *tendsto-add* [*of g a F* $\lambda x.\ f\ x - g\ x\ 0$] **by** *simp*

lemma *Lim-transform2*: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (g \longrightarrow a)\ F$
for *a b* :: *'a*::*real-normed-vector*
by (*erule Lim-transform*) (*simp add*: *tendsto-minus-cancel*)

proposition *Lim-transform-eq*: $((\lambda x.\ f\ x - g\ x) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow a)\ F \longleftrightarrow (g \longrightarrow a)\ F$
for *a* :: *'a*::*real-normed-vector*
using *Lim-transform Lim-transform2* **by** *blast*

lemma *Lim-transform-eventually*:
$[\![(f \longrightarrow l)\ F;\ \textit{eventually}\ (\lambda x.\ f\ x = g\ x)\ F]\!] \Longrightarrow (g \longrightarrow l)\ F$
using *eventually-elim2* **by** (*fastforce simp add*: *tendsto-def*)

lemma *Lim-transform-within*:
assumes $(f \longrightarrow l)\ (at\ x\ within\ S)$
and $0 < d$
and $\bigwedge x'.\ x' \in S \Longrightarrow 0 < dist\ x'\ x \Longrightarrow dist\ x'\ x < d \Longrightarrow f\ x' = g\ x'$
shows $(g \longrightarrow l)\ (at\ x\ within\ S)$
proof (*rule Lim-transform-eventually*)
show *eventually* $(\lambda x.\ f\ x = g\ x)\ (at\ x\ within\ S)$
using *assms* **by** (*auto simp*: *eventually-at*)
show $(f \longrightarrow l)\ (at\ x\ within\ S)$
by *fact*
qed

lemma *filterlim-transform-within*:
assumes *filterlim g G* $(at\ x\ within\ S)$
assumes $G \leq F\ 0<d\ (\bigwedge x'.\ x' \in S \Longrightarrow 0 < dist\ x'\ x \Longrightarrow dist\ x'\ x < d \Longrightarrow f\ x' = g\ x')$
shows *filterlim f F* $(at\ x\ within\ S)$
using *assms*
apply (*elim filterlim-mono-eventually*)
unfolding *eventually-at* **by** *auto*

Common case assuming being away from some crucial point like 0.

lemma *Lim-transform-away-within*:

fixes $a\ b :: 'a{::}t1\text{-}space$
assumes $a \neq b$
and $\forall x{\in}S.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and $(f \longrightarrow l)\ (at\ a\ within\ S)$
shows $(g \longrightarrow l)\ (at\ a\ within\ S)$
proof (*rule Lim-transform-eventually*)
show $(f \longrightarrow l)\ (at\ a\ within\ S)$
by *fact*
show *eventually* $(\lambda x.\ f\ x = g\ x)\ (at\ a\ within\ S)$
unfolding *eventually-at-topological*
by (*rule exI* [**where** $x{=}-\ \{b\}$]) (*simp add*: *open-Compl assms*)
qed

lemma *Lim-transform-away-at*:
fixes $a\ b :: 'a{::}t1\text{-}space$
assumes *ab*: $a \neq b$
and *fg*: $\forall x.\ x \neq a \wedge x \neq b \longrightarrow f\ x = g\ x$
and *fl*: $(f \longrightarrow l)\ (at\ a)$
shows $(g \longrightarrow l)\ (at\ a)$
using *Lim-transform-away-within*[*OF ab*, *of UNIV f g l*] *fg fl* **by** *simp*

Alternatively, within an open set.

lemma *Lim-transform-within-open*:
assumes $(f \longrightarrow l)\ (at\ a\ within\ T)$
and *open s* **and** $a \in s$
and $\bigwedge x.\ x{\in}s \Longrightarrow x \neq a \Longrightarrow f\ x = g\ x$
shows $(g \longrightarrow l)\ (at\ a\ within\ T)$
proof (*rule Lim-transform-eventually*)
show *eventually* $(\lambda x.\ f\ x = g\ x)\ (at\ a\ within\ T)$
unfolding *eventually-at-topological*
using *assms* **by** *auto*
show $(f \longrightarrow l)\ (at\ a\ within\ T)$ **by** *fact*
qed

A congruence rule allowing us to transform limits assuming not at point.

lemma *Lim-cong-within*:
assumes $a = b$
and $x = y$
and $S = T$
and $\bigwedge x.\ x \neq b \Longrightarrow x \in T \Longrightarrow f\ x = g\ x$
shows $(f \longrightarrow x)\ (at\ a\ within\ S) \longleftrightarrow (g \longrightarrow y)\ (at\ b\ within\ T)$
unfolding *tendsto-def eventually-at-topological*
using *assms* **by** *simp*

An unbounded sequence's inverse tends to 0.

lemma *LIMSEQ-inverse-zero*:
assumes $\bigwedge r{::}real.\ \exists N.\ \forall n{\geq}N.\ r < X\ n$
shows $(\lambda n.\ inverse\ (X\ n)) \longrightarrow 0$
apply (*rule filterlim-compose*[*OF tendsto-inverse-0*])

by (*metis assms eventually-at-top-linorderI filterlim-at-top-dense filterlim-at-top-imp-at-infinity*)

The sequence $1 / n$ tends to 0 as n tends to infinity.

lemma *LIMSEQ-inverse-real-of-nat*: $(\lambda n.\ inverse\ (real\ (Suc\ n))) \longrightarrow 0$
by (*metis filterlim-compose tendsto-inverse-0 filterlim-mono order-refl filterlim-Suc filterlim-compose*[*OF filterlim-real-sequentially*] *at-top-le-at-infinity*)

The sequence $r + 1 / n$ tends to r as n tends to infinity is now easily proved.

lemma *LIMSEQ-inverse-real-of-nat-add*: $(\lambda n.\ r + inverse\ (real\ (Suc\ n))) \longrightarrow r$
using *tendsto-add* [*OF tendsto-const LIMSEQ-inverse-real-of-nat*] **by** *auto*

lemma *LIMSEQ-inverse-real-of-nat-add-minus*: $(\lambda n.\ r + -inverse\ (real\ (Suc\ n))) \longrightarrow r$
using *tendsto-add* [*OF tendsto-const tendsto-minus* [*OF LIMSEQ-inverse-real-of-nat*]]
by *auto*

lemma *LIMSEQ-inverse-real-of-nat-add-minus-mult*: $(\lambda n.\ r * (1 + - inverse\ (real\ (Suc\ n)))) \longrightarrow r$
using *tendsto-mult* [*OF tendsto-const LIMSEQ-inverse-real-of-nat-add-minus* [*of 1*]]
by *auto*

lemma *lim-inverse-n*: $((\lambda n.\ inverse(of\text{-}nat\ n)) \longrightarrow (0::'a::real\text{-}normed\text{-}field))\ sequentially$
using *lim-1-over-n* **by** (*simp add*: *inverse-eq-divide*)

lemma *lim-inverse-n'*: $((\lambda n.\ 1 / n) \longrightarrow 0)\ sequentially$
using *lim-inverse-n*
by (*simp add*: *inverse-eq-divide*)

lemma *LIMSEQ-Suc-n-over-n*: $(\lambda n.\ of\text{-}nat\ (Suc\ n) / of\text{-}nat\ n :: 'a :: real\text{-}normed\text{-}field) \longrightarrow 1$
proof (*rule Lim-transform-eventually*)
show *eventually* $(\lambda n.\ 1 + inverse\ (of\text{-}nat\ n :: 'a) = of\text{-}nat\ (Suc\ n) / of\text{-}nat\ n)\ sequentially$
using *eventually-gt-at-top*[*of 0*::*nat*]
by *eventually-elim* (*simp add*: *field-simps*)
have $(\lambda n.\ 1 + inverse\ (of\text{-}nat\ n) :: 'a) \longrightarrow 1 + 0$
by (*intro tendsto-add tendsto-const lim-inverse-n*)
then show $(\lambda n.\ 1 + inverse\ (of\text{-}nat\ n) :: 'a) \longrightarrow 1$
by *simp*
qed

lemma *LIMSEQ-n-over-Suc-n*: $(\lambda n.\ of\text{-}nat\ n / of\text{-}nat\ (Suc\ n) :: 'a :: real\text{-}normed\text{-}field) \longrightarrow 1$
proof (*rule Lim-transform-eventually*)
show *eventually* $(\lambda n.\ inverse\ (of\text{-}nat\ (Suc\ n) / of\text{-}nat\ n :: 'a) =$
$of\text{-}nat\ n / of\text{-}nat\ (Suc\ n))\ sequentially$

using *eventually-gt-at-top*[*of 0*::*nat*]
by *eventually-elim* (*simp add*: *field-simps del*: *of-nat-Suc*)
have (λ*n*. *inverse* (*of-nat* (*Suc n*) / *of-nat n* :: *'a*)) ⟶ *inverse 1*
by (*intro tendsto-inverse LIMSEQ-Suc-n-over-n*) *simp-all*
then show (λ*n*. *inverse* (*of-nat* (*Suc n*) / *of-nat n* :: *'a*)) ⟶ *1*
by *simp*
qed

108.8 Convergence on sequences

lemma *convergent-cong*:
assumes *eventually* (λ*x*. *f x* = *g x*) *sequentially*
shows *convergent f* ⟷ *convergent g*
unfolding *convergent-def*
by (*subst filterlim-cong*[*OF refl refl assms*]) (*rule refl*)

lemma *convergent-Suc-iff*: *convergent* (λ*n*. *f* (*Suc n*)) ⟷ *convergent f*
by (*auto simp*: *convergent-def filterlim-sequentially-Suc*)

lemma *convergent-ignore-initial-segment*: *convergent* (λ*n*. *f* (*n* + *m*)) = *convergent f*
proof (*induct m arbitrary*: *f*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc m*)
have *convergent* (λ*n*. *f* (*n* + *Suc m*)) ⟷ *convergent* (λ*n*. *f* (*Suc n* + *m*))
by *simp*
also have ... ⟷ *convergent* (λ*n*. *f* (*n* + *m*))
by (*rule convergent-Suc-iff*)
also have ... ⟷ *convergent f*
by (*rule Suc*)
finally show *?case* **.**
qed

lemma *convergent-add*:
fixes *X Y* :: *nat* ⇒ *'a*::*topological-monoid-add*
assumes *convergent* (λ*n*. *X n*)
and *convergent* (λ*n*. *Y n*)
shows *convergent* (λ*n*. *X n* + *Y n*)
using *assms* **unfolding** *convergent-def* **by** (*blast intro*: *tendsto-add*)

lemma *convergent-sum*:
fixes *X* :: *'a* ⇒ *nat* ⇒ *'b*::*topological-comm-monoid-add*
shows (⋀*i*. *i* ∈ *A* ⟹ *convergent* (λ*n*. *X i n*)) ⟹ *convergent* (λ*n*. ∑ *i*∈*A*. *X i n*)
by (*induct A rule*: *infinite-finite-induct*) (*simp-all add*: *convergent-const convergent-add*)

lemma (**in** *bounded-linear*) *convergent*:
assumes *convergent* (λ*n*. *X n*)
shows *convergent* (λ*n*. *f* (*X n*))
using *assms* **unfolding** *convergent-def* **by** (*blast intro*: *tendsto*)

lemma (**in** *bounded-bilinear*) *convergent*:
assumes *convergent* (λ*n*. *X n*)
and *convergent* (λ*n*. *Y n*)
shows *convergent* (λ*n*. *X n* ** *Y n*)
using *assms* **unfolding** *convergent-def* **by** (*blast intro*: *tendsto*)

lemma *convergent-minus-iff*:
fixes *X* :: *nat* ⇒ *'a*::*topological-group-add*
shows *convergent X* ⟷ *convergent* (λ*n*. − *X n*)
unfolding *convergent-def* **by** (*force dest*: *tendsto-minus*)

lemma *convergent-diff*:
fixes *X Y* :: *nat* ⇒ *'a*::*topological-group-add*
assumes *convergent* (λ*n*. *X n*)
assumes *convergent* (λ*n*. *Y n*)
shows *convergent* (λ*n*. *X n* − *Y n*)
using *assms* **unfolding** *convergent-def* **by** (*blast intro*: *tendsto-diff*)

lemma *convergent-norm*:
assumes *convergent f*
shows *convergent* (λ*n*. *norm* (*f n*))
proof −
from *assms* **have** *f* ⟶ *lim f*
by (*simp add*: *convergent-LIMSEQ-iff*)
then have (λ*n*. *norm* (*f n*)) ⟶ *norm* (*lim f*)
by (*rule tendsto-norm*)
then show *?thesis*
by (*auto simp*: *convergent-def*)
qed

lemma *convergent-of-real*:
convergent f ⟹ *convergent* (λ*n*. *of-real* (*f n*) :: *'a*::*real-normed-algebra-1*)
unfolding *convergent-def* **by** (*blast intro*!: *tendsto-of-real*)

lemma *convergent-add-const-iff*:
convergent (λ*n*. *c* + *f n* :: *'a*::*topological-ab-group-add*) ⟷ *convergent f*
proof
assume *convergent* (λ*n*. *c* + *f n*)
from *convergent-diff*[*OF this convergent-const*[*of c*]] **show** *convergent f*
by *simp*
next
assume *convergent f*
from *convergent-add*[*OF convergent-const*[*of c*] *this*] **show** *convergent* (λ*n*. *c* + *f n*)

by *simp*
qed

lemma *convergent-add-const-right-iff*:
convergent (λ*n*. *f n* + *c* :: *'a*::*topological-ab-group-add*) ⟷ *convergent f*
using *convergent-add-const-iff*[*of c f*] **by** (*simp add*: *add-ac*)

lemma *convergent-diff-const-right-iff*:
convergent (λ*n*. *f n* − *c* :: *'a*::*topological-ab-group-add*) ⟷ *convergent f*
using *convergent-add-const-right-iff*[*of f* −*c*] **by** (*simp add*: *add-ac*)

lemma *convergent-mult*:
fixes *X Y* :: *nat* ⇒ *'a*::*topological-semigroup-mult*
assumes *convergent* (λ*n*. *X n*)
and *convergent* (λ*n*. *Y n*)
shows *convergent* (λ*n*. *X n* ∗ *Y n*)
using *assms* **unfolding** *convergent-def* **by** (*blast intro*: *tendsto-mult*)

lemma *convergent-mult-const-iff*:
assumes *c* ≠ *0*
shows *convergent* (λ*n*. *c* ∗ *f n* :: *'a*::{*field*,*topological-semigroup-mult*}) ⟷ *convergent f*
proof
assume *convergent* (λ*n*. *c* ∗ *f n*)
from *assms convergent-mult*[*OF this convergent-const*[*of inverse c*]]
show *convergent f* **by** (*simp add*: *field-simps*)
next
assume *convergent f*
from *convergent-mult*[*OF convergent-const*[*of c*] *this*] **show** *convergent* (λ*n*. *c* ∗ *f n*)
by *simp*
qed

lemma *convergent-mult-const-right-iff*:
fixes *c* :: *'a*::{*field*,*topological-semigroup-mult*}
assumes *c* ≠ *0*
shows *convergent* (λ*n*. *f n* ∗ *c*) ⟷ *convergent f*
using *convergent-mult-const-iff*[*OF assms*, *of f*] **by** (*simp add*: *mult-ac*)

lemma *convergent-imp-Bseq*: *convergent f* ⟹ *Bseq f*
by (*simp add*: *Cauchy-Bseq convergent-Cauchy*)

A monotone sequence converges to its least upper bound.

lemma *LIMSEQ-incseq-SUP*:
fixes *X* :: *nat* ⇒ *'a*::{*conditionally-complete-linorder*,*linorder-topology*}
assumes *u*: *bdd-above* (*range X*)
and *X*: *incseq X*
shows *X* ⟶ (*SUP i*. *X i*)
by (*rule order-tendstoI*)

(*auto simp*: *eventually-sequentially u less-cSUP-iff*
intro: *X*[*THEN incseqD*] *less-le-trans cSUP-lessD*[*OF u*])

lemma *LIMSEQ-decseq-INF*:
fixes *X* :: *nat* ⇒ *'a*::{*conditionally-complete-linorder*, *linorder-topology*}
assumes *u*: *bdd-below* (*range X*)
and *X*: *decseq X*
shows *X* ⟶ (*INF i*. *X i*)
by (*rule order-tendstoI*)
(*auto simp*: *eventually-sequentially u cINF-less-iff*
intro: *X*[*THEN decseqD*] *le-less-trans less-cINF-D*[*OF u*])

Main monotonicity theorem.

lemma *Bseq-monoseq-convergent*: *Bseq X* ⟹ *monoseq X* ⟹ *convergent X*
for *X* :: *nat* ⇒ *real*
by (*auto simp*: *monoseq-iff convergent-def intro*: *LIMSEQ-decseq-INF LIMSEQ-incseq-SUP*
dest: *Bseq-bdd-above Bseq-bdd-below*)

lemma *Bseq-mono-convergent*: *Bseq X* ⟹ (∀ *m n*. *m* ≤ *n* ⟶ *X m* ≤ *X n*) ⟹
convergent X
for *X* :: *nat* ⇒ *real*
by (*auto intro*!: *Bseq-monoseq-convergent incseq-imp-monoseq simp*: *incseq-def*)

lemma *monoseq-imp-convergent-iff-Bseq*: *monoseq f* ⟹ *convergent f* ⟷ *Bseq f*
for *f* :: *nat* ⇒ *real*
using *Bseq-monoseq-convergent*[*of f*] *convergent-imp-Bseq*[*of f*] **by** *blast*

lemma *Bseq-monoseq-convergent'-inc*:
fixes *f* :: *nat* ⇒ *real*
shows *Bseq* (λ*n*. *f* (*n* + *M*)) ⟹ (⋀*m n*. *M* ≤ *m* ⟹ *m* ≤ *n* ⟹ *f m* ≤ *f n*)
⟹ *convergent f*
by (*subst convergent-ignore-initial-segment* [*symmetric*, *of* - *M*])
(*auto intro*!: *Bseq-monoseq-convergent simp*: *monoseq-def*)

lemma *Bseq-monoseq-convergent'-dec*:
fixes *f* :: *nat* ⇒ *real*
shows *Bseq* (λ*n*. *f* (*n* + *M*)) ⟹ (⋀*m n*. *M* ≤ *m* ⟹ *m* ≤ *n* ⟹ *f m* ≥ *f n*)
⟹ *convergent f*
by (*subst convergent-ignore-initial-segment* [*symmetric*, *of* - *M*])
(*auto intro*!: *Bseq-monoseq-convergent simp*: *monoseq-def*)

lemma *Cauchy-iff*: *Cauchy X* ⟷ (∀ *e*>*0*. ∃ *M*. ∀ *m*≥*M*. ∀ *n*≥*M*. *norm* (*X m* −
X n) < *e*)
for *X* :: *nat* ⇒ *'a*::*real-normed-vector*
unfolding *Cauchy-def dist-norm* **..**

lemma *CauchyI*: (⋀*e*. *0* < *e* ⟹ ∃ *M*. ∀ *m*≥*M*. ∀ *n*≥*M*. *norm* (*X m* − *X n*) <
e) ⟹ *Cauchy X*
for *X* :: *nat* ⇒ *'a*::*real-normed-vector*

by (*simp add*: *Cauchy-iff*)

lemma *CauchyD*: *Cauchy X* $\Longrightarrow$ *0* $<$ *e* $\Longrightarrow$ $\exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.$ *norm* (*X m* $-$ *X n*) $<$ *e*
for *X* :: *nat* $\Rightarrow$ *'a*::*real-normed-vector*
by (*simp add*: *Cauchy-iff*)

lemma *incseq-convergent*:
fixes *X* :: *nat* $\Rightarrow$ *real*
assumes *incseq X*
and $\forall i.$ *X i* $\leq$ *B*
obtains *L* **where** *X* $\longrightarrow$ *L* $\forall i.$ *X i* $\leq$ *L*
proof *atomize-elim*
from *incseq-bounded*[*OF assms*] ‹*incseq X*› *Bseq-monoseq-convergent*[*of X*]
obtain *L* **where** *X* $\longrightarrow$ *L*
by (*auto simp*: *convergent-def monoseq-def incseq-def*)
with ‹*incseq X*› **show** $\exists L.$ *X* $\longrightarrow$ *L* $\wedge$ $(\forall i.$ *X i* $\leq$ *L*)
by (*auto intro*!: *exI*[*of - L*] *incseq-le*)
qed

lemma *decseq-convergent*:
fixes *X* :: *nat* $\Rightarrow$ *real*
assumes *decseq X*
and $\forall i.$ *B* $\leq$ *X i*
obtains *L* **where** *X* $\longrightarrow$ *L* $\forall i.$ *L* $\leq$ *X i*
proof *atomize-elim*
from *decseq-bounded*[*OF assms*] ‹*decseq X*› *Bseq-monoseq-convergent*[*of X*]
obtain *L* **where** *X* $\longrightarrow$ *L*
by (*auto simp*: *convergent-def monoseq-def decseq-def*)
with ‹*decseq X*› **show** $\exists L.$ *X* $\longrightarrow$ *L* $\wedge$ $(\forall i.$ *L* $\leq$ *X i*)
by (*auto intro*!: *exI*[*of - L*] *decseq-ge*)
qed

lemma *monoseq-convergent*:
fixes *X* :: *nat* $\Rightarrow$ *real*
assumes *X*: *monoseq X* **and** *B*: $\bigwedge i.$ $|X\ i| \leq B$
obtains *L* **where** *X* $\longrightarrow$ *L*
using *X* **unfolding** *monoseq-iff*
proof
assume *incseq X*
show *thesis*
using *abs-le-D1* [*OF B*] *incseq-convergent* [*OF* ‹*incseq X*›] *that* **by** *meson*
next
assume *decseq X*
show *thesis*
using *decseq-convergent* [*OF* ‹*decseq X*›] *that*
by (*metis B abs-le-iff add.inverse-inverse neg-le-iff-le*)
qed

108.9 More about *filterlim* (thanks to Wenda Li)

lemma *filterlim-at-infinity-times*:
fixes $f :: 'a \Rightarrow 'b$::*real-normed-field*
assumes *filterlim f at-infinity F filterlim g at-infinity F*
shows *filterlim* $(\lambda x.\ f\ x * g\ x)$ *at-infinity F*
proof −
have $((\lambda x.\ inverse\ (f\ x) * inverse\ (g\ x)) \longrightarrow 0 * 0)\ F$
by (*intro tendsto-mult tendsto-inverse assms filterlim-compose*[*OF tendsto-inverse-0*])
then have *filterlim* $(\lambda x.\ inverse\ (f\ x) * inverse\ (g\ x))$ *(at 0) F*
unfolding *filterlim-at* **using** *assms*
by (*auto intro*: *filterlim-at-infinity-imp-eventually-ne tendsto-imp-eventually-ne eventually-conj*)
then show *?thesis*
by (*subst filterlim-inverse-at-iff*[*symmetric*]) *simp-all*
qed

lemma *filterlim-at-top-at-bot*[*elim*]:
fixes f::$'a \Rightarrow 'b$::*unbounded-dense-linorder* **and** F::$'a$ *filter*
assumes *top*:*filterlim f at-top F* **and** *bot*: *filterlim f at-bot F* **and** $F \neq bot$
shows *False*
proof −
obtain c::$'b$ **where** *True* **by** *auto*
have $\forall_F\ x\ in\ F.\ c < f\ x$
using *top* **unfolding** *filterlim-at-top-dense* **by** *auto*
moreover have $\forall_F\ x\ in\ F.\ f\ x < c$
using *bot* **unfolding** *filterlim-at-bot-dense* **by** *auto*
ultimately have $\forall_F\ x\ in\ F.\ c < f\ x \wedge f\ x < c$
using *eventually-conj* **by** *auto*
then have $\forall_F\ x\ in\ F.\ False$ **by** (*auto elim*:*eventually-mono*)
then show *False* **using** ‹$F \neq bot$› **by** *auto*
qed

lemma *filterlim-at-top-nhds*[*elim*]:
fixes f::$'a \Rightarrow 'b$::{*unbounded-dense-linorder*,*order-topology*} **and** F::$'a$ *filter*
assumes *top*:*filterlim f at-top F* **and** *tendsto*: $(f \longrightarrow c)\ F$ **and** $F \neq bot$
shows *False*
proof −
obtain c'::$'b$ **where** $c' > c$ **using** *gt-ex* **by** *blast*
have $\forall_F\ x\ in\ F.\ c' < f\ x$
using *top* **unfolding** *filterlim-at-top-dense* **by** *auto*
moreover have $\forall_F\ x\ in\ F.\ f\ x < c'$
using *order-tendstoD*[*OF tendsto*,*of c'*] ‹$c' > c$› **by** *auto*
ultimately have $\forall_F\ x\ in\ F.\ c' < f\ x \wedge f\ x < c'$
using *eventually-conj* **by** *auto*
then have $\forall_F\ x\ in\ F.\ False$ **by** (*auto elim*:*eventually-mono*)
then show *False* **using** ‹$F \neq bot$› **by** *auto*
qed

lemma *filterlim-at-bot-nhds*[*elim*]:

fixes $f::'a \Rightarrow 'b::\{unbounded\text{-}dense\text{-}linorder,order\text{-}topology\}$ **and** $F::'a\ filter$
assumes *top:filterlim f at-bot F* **and** *tendsto*: $(f \longrightarrow c)\ F$ **and** $F \neq bot$
shows *False*
proof −
obtain $c'::'b$ **where** $c'<c$ **using** *lt-ex* **by** *blast*
have $\forall_F\ x\ in\ F.\ c' > f\ x$
using *top* **unfolding** *filterlim-at-bot-dense* **by** *auto*
moreover have $\forall_F\ x\ in\ F.\ f\ x > c'$
using *order-tendstoD*[*OF tendsto*,*of c'*] ‹$c'<c$› **by** *auto*
ultimately have $\forall_F\ x\ in\ F.\ c' < f\ x \wedge f\ x < c'$
using *eventually-conj* **by** *auto*
then have $\forall_F\ x\ in\ F.\ False$ **by** (*auto elim:eventually-mono*)
then show *False* **using** ‹$F \neq bot$› **by** *auto*
qed

lemma *eventually-times-inverse-1*:
fixes $f::'a \Rightarrow 'b::\{field,t2\text{-}space\}$
assumes $(f \longrightarrow c)\ F\ c \neq 0$
shows $\forall_F\ x\ in\ F.\ inverse\ (f\ x) * f\ x = 1$
by (*smt* (*verit*) *assms eventually-mono mult.commute right-inverse tendsto-imp-eventually-ne*)

lemma *filterlim-at-infinity-divide-iff*:
fixes $f::'a \Rightarrow 'b::real\text{-}normed\text{-}field$
assumes $(f \longrightarrow c)\ F\ c \neq 0$
shows $(LIM\ x\ F.\ f\ x\ /\ g\ x :> at\text{-}infinity) \longleftrightarrow (LIM\ x\ F.\ g\ x :> at\ 0)$
proof
assume $LIM\ x\ F.\ f\ x\ /\ g\ x :> at\text{-}infinity$
then have $LIM\ x\ F.\ inverse\ (f\ x) * (f\ x\ /\ g\ x) :> at\text{-}infinity$
using *assms tendsto-inverse tendsto-mult-filterlim-at-infinity* **by** *fastforce*
then have $LIM\ x\ F.\ inverse\ (g\ x) :> at\text{-}infinity$
apply (*elim filterlim-mono-eventually*)
using *eventually-times-inverse-1*[*OF assms*]
by (*auto elim:eventually-mono simp add:field-simps*)
then show *filterlim g* (*at 0*) *F* **using** *filterlim-inverse-at-iff*[*symmetric*] **by** *force*

next
assume *filterlim g* (*at 0*) *F*
then have *filterlim* ($\lambda x.\ inverse\ (g\ x)$) *at-infinity F*
using *filterlim-compose filterlim-inverse-at-infinity* **by** *blast*
then have $LIM\ x\ F.\ f\ x * inverse\ (g\ x) :> at\text{-}infinity$
using *tendsto-mult-filterlim-at-infinity*[*OF assms*, *of* $\lambda x.\ inverse(g\ x)$]
by *simp*
then show $LIM\ x\ F.\ f\ x\ /\ g\ x :> at\text{-}infinity$ **by** (*simp add*: *divide-inverse*)
qed

lemma *filterlim-tendsto-pos-mult-at-top-iff*:
fixes $f::'a \Rightarrow real$
assumes $(f \longrightarrow c)\ F$ **and** $0 < c$
shows $(LIM\ x\ F.\ (f\ x * g\ x) :> at\text{-}top) \longleftrightarrow (LIM\ x\ F.\ g\ x :> at\text{-}top)$

proof
assume *filterlim g at-top F*
then show *LIM x F. f x * g x :> at-top*
using *filterlim-tendsto-pos-mult-at-top[OF assms]* **by** *auto*
next
assume *asm:LIM x F. f x * g x :> at-top*
have *((λx. inverse (f x)) ⟶ inverse c) F*
using *tendsto-inverse[OF assms(1)] ‹0<c›* **by** *auto*
moreover have *inverse c >0* **using** *assms(2)* **by** *auto*
ultimately have *LIM x F. inverse (f x) * (f x * g x) :> at-top*
using *filterlim-tendsto-pos-mult-at-top[OF - - asm,of λx. inverse (f x) inverse c]* **by** *auto*
then show *LIM x F. g x :> at-top*
apply *(elim filterlim-mono-eventually)*
apply *simp-all[2]*
using *eventually-times-inverse-1[OF assms(1)] ‹c>0› eventually-mono* **by** *fastforce*
qed

lemma *filterlim-tendsto-pos-mult-at-bot-iff*:
fixes *c :: real*
assumes *(f ⟶ c) F 0 < c*
shows *(LIM x F. f x * g x :> at-bot) ⟷ filterlim g at-bot F*
using *filterlim-tendsto-pos-mult-at-top-iff[OF assms(1,2), of λx. - g x]*
unfolding *filterlim-uminus-at-bot* **by** *simp*

lemma *filterlim-tendsto-neg-mult-at-top-iff*:
fixes *f::'a ⇒ real*
assumes *(f ⟶ c) F* **and** *c < 0*
shows *(LIM x F. (f x * g x) :> at-top) ⟷ (LIM x F. g x :> at-bot)*
proof −
have *(LIM x F. f x * g x :> at-top) = (LIM x F. - g x :> at-top)*
apply *(rule filterlim-tendsto-pos-mult-at-top-iff[of λx. - f x -c F λx. - g x, simplified])*
using *assms* **by** *(auto intro: tendsto-intros)*
also have *... = (LIM x F. g x :> at-bot)*
using *filterlim-uminus-at-bot[symmetric]* **by** *auto*
finally show *?thesis* **.**
qed

lemma *filterlim-tendsto-neg-mult-at-bot-iff*:
fixes *c :: real*
assumes *(f ⟶ c) F 0 > c*
shows *(LIM x F. f x * g x :> at-bot) ⟷ filterlim g at-top F*
using *filterlim-tendsto-neg-mult-at-top-iff[OF assms(1,2), of λx. - g x]*
unfolding *filterlim-uminus-at-top* **by** *simp*

108.10 Power Sequences

lemma *Bseq-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow Bseq\ (\lambda n.\ x \mathbin{\hat{}} n)$
for $x :: real$
by (*metis decseq-bounded decseq-def power-decreasing zero-le-power*)

lemma *monoseq-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow monoseq\ (\lambda n.\ x \mathbin{\hat{}} n)$
for $x :: real$
using *monoseq-def power-decreasing* **by** *blast*

lemma *convergent-realpow*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow convergent\ (\lambda n.\ x \mathbin{\hat{}} n)$
for $x :: real$
by (*blast intro*!: *Bseq-monoseq-convergent Bseq-realpow monoseq-realpow*)

lemma *LIMSEQ-inverse-realpow-zero*: $1 < x \Longrightarrow (\lambda n.\ inverse\ (x \mathbin{\hat{}} n)) \longrightarrow 0$
for $x :: real$
by (*rule filterlim-compose*[*OF tendsto-inverse-0 filterlim-realpow-sequentially-gt1*])
simp

lemma *LIMSEQ-realpow-zero*:
fixes $x :: real$
assumes $0 \leq x\ x < 1$
shows $(\lambda n.\ x \mathbin{\hat{}} n) \longrightarrow 0$
proof (*cases* $x = 0$)
case *False*
with ‹$0 \leq x$› **have** $1 < inverse\ x$
using ‹$x < 1$› **by** (*simp add*: *one-less-inverse*)
then have $(\lambda n.\ inverse\ (inverse\ x \mathbin{\hat{}} n)) \longrightarrow 0$
by (*rule LIMSEQ-inverse-realpow-zero*)
then show *?thesis* **by** (*simp add*: *power-inverse*)
next
case *True*
show *?thesis*
by (*rule LIMSEQ-imp-Suc*) (*simp add*: *True*)
qed

lemma *LIMSEQ-power-zero* [*tendsto-intros*]: $norm\ x < 1 \Longrightarrow (\lambda n.\ x \mathbin{\hat{}} n) \longrightarrow 0$
for $x :: 'a$::*real-normed-algebra-1*
apply (*drule LIMSEQ-realpow-zero* [*OF norm-ge-zero*])
by (*simp add*: *Zfun-le norm-power-ineq tendsto-Zfun-iff*)

lemma *LIMSEQ-divide-realpow-zero*: $1 < x \Longrightarrow (\lambda n.\ a\ /\ (x \mathbin{\hat{}} n) :: real) \longrightarrow 0$
by (*rule tendsto-divide-0* [*OF tendsto-const filterlim-realpow-sequentially-gt1*])
simp

lemma
tendsto-power-zero:
fixes x::$'a$::*real-normed-algebra-1*
assumes *filterlim f at-top F*

assumes *norm x < 1*
shows $((\lambda y.\ x\ \hat{}\ (f\ y)) \longrightarrow 0)\ F$
proof (*rule tendstoI*)
fix *e*::*real* **assume** *0 < e*
from *tendstoD*[*OF LIMSEQ-power-zero*[*OF* ‹*norm x < 1*›] ‹*0 < e*›]
have $\forall_F\ xa$ *in sequentially. norm* $(x\ \hat{}\ xa) < e$
by *simp*
then obtain *N* **where** *N*: *norm* $(x\ \hat{}\ n) < e$ **if** $n \geq N$ **for** *n*
by (*auto simp*: *eventually-sequentially*)
have $\forall_F\ i$ *in F. f i* $\geq N$
using ‹*filterlim f sequentially F*›
by (*simp add*: *filterlim-at-top*)
then show $\forall_F\ i$ *in F. dist* $(x\ \hat{}\ f\ i)\ 0 < e$
by *eventually-elim* (*auto simp*: *N*)
qed

Limit of c^n for $|c| < 1$.

lemma *LIMSEQ-abs-realpow-zero*: $|c| < 1 \Longrightarrow (\lambda n.\ |c|\ \hat{}\ n :: real) \longrightarrow 0$
by (*rule LIMSEQ-realpow-zero* [*OF abs-ge-zero*])

lemma *LIMSEQ-abs-realpow-zero2*: $|c| < 1 \Longrightarrow (\lambda n.\ c\ \hat{}\ n :: real) \longrightarrow 0$
by (*rule LIMSEQ-power-zero*) *simp*

108.11 Limits of Functions

lemma *LIM-eq*: $f\ -a\rightarrow\ L = (\forall r{>}0.\ \exists s{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x - L) < r)$
for *a* :: *'a*::*real-normed-vector* **and** *L* :: *'b*::*real-normed-vector*
by (*simp add*: *LIM-def dist-norm*)

lemma *LIM-I*:
$(\bigwedge r.\ 0 < r \Longrightarrow \exists s{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x - L) < r) \Longrightarrow f\ -a\rightarrow\ L$
for *a* :: *'a*::*real-normed-vector* **and** *L* :: *'b*::*real-normed-vector*
by (*simp add*: *LIM-eq*)

lemma *LIM-D*: $f\ -a\rightarrow\ L \Longrightarrow 0 < r \Longrightarrow \exists s{>}0.\forall x.\ x \neq a \wedge norm\ (x - a) < s \longrightarrow norm\ (f\ x - L) < r$
for *a* :: *'a*::*real-normed-vector* **and** *L* :: *'b*::*real-normed-vector*
by (*simp add*: *LIM-eq*)

lemma *LIM-offset*: $f\ -a\rightarrow\ L \Longrightarrow (\lambda x.\ f\ (x + k))\ -(a - k)\rightarrow\ L$
for *a* :: *'a*::*real-normed-vector*
by (*simp add*: *filtermap-at-shift*[*symmetric*, *of a k*] *filterlim-def filtermap-filtermap*)

lemma *LIM-offset-zero*: $f\ -a\rightarrow\ L \Longrightarrow (\lambda h.\ f\ (a + h))\ -0\rightarrow\ L$
for *a* :: *'a*::*real-normed-vector*
by (*drule LIM-offset* [**where** *k* = *a*]) (*simp add*: *add.commute*)

lemma *LIM-offset-zero-cancel*: $(\lambda h.\ f\ (a + h)) \ -0\rightarrow\ L \Longrightarrow f\ -a\rightarrow\ L$
for $a :: 'a$::*real-normed-vector*
by (*drule LIM-offset* [**where** $k = -a$]) *simp*

lemma *LIM-offset-zero-iff*: *NO-MATCH 0 a* $\Longrightarrow f\ -a\rightarrow\ L \longleftrightarrow (\lambda h.\ f\ (a + h))$
$-0\rightarrow\ L$
for $f :: 'a ::$ *real-normed-vector* $\Rightarrow$ -
using *LIM-offset-zero-cancel*[*of f a L*] *LIM-offset-zero*[*of f L a*] **by** *auto*

lemma *tendsto-offset-zero-iff*:
fixes $f :: 'a ::$ *real-normed-vector* $\Rightarrow$ -
assumes *NO-MATCH 0 a* $a \in S$ *open* S
shows $(f \longrightarrow L)\ (at\ a\ within\ S) \longleftrightarrow ((\lambda h.\ f\ (a + h)) \longrightarrow L)\ (at\ 0)$
using *assms* **by** (*simp add*: *tendsto-within-open-NO-MATCH LIM-offset-zero-iff*)

lemma *LIM-zero*: $(f \longrightarrow l)\ F \Longrightarrow ((\lambda x.\ f\ x - l) \longrightarrow 0)\ F$
for $f :: 'a \Rightarrow 'b$::*real-normed-vector*
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-zero-cancel*:
fixes $f :: 'a \Rightarrow 'b$::*real-normed-vector*
shows $((\lambda x.\ f\ x - l) \longrightarrow 0)\ F \Longrightarrow (f \longrightarrow l)\ F$
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-zero-iff*: $((\lambda x.\ f\ x - l) \longrightarrow 0)\ F = (f \longrightarrow l)\ F$
for $f :: 'a \Rightarrow 'b$::*real-normed-vector*
unfolding *tendsto-iff dist-norm* **by** *simp*

lemma *LIM-imp-LIM*:
fixes $f :: 'a$::*topological-space* $\Rightarrow 'b$::*real-normed-vector*
fixes $g :: 'a$::*topological-space* $\Rightarrow 'c$::*real-normed-vector*
assumes *f*: $f\ -a\rightarrow\ l$
and *le*: $\bigwedge x.\ x \neq a \Longrightarrow norm\ (g\ x - m) \leq norm\ (f\ x - l)$
shows $g\ -a\rightarrow\ m$
by (*rule metric-LIM-imp-LIM* [*OF f*]) (*simp add*: *dist-norm le*)

lemma *LIM-equal2*:
fixes $f\ g :: 'a$::*real-normed-vector* $\Rightarrow 'b$::*topological-space*
assumes $0 < R$
and $\bigwedge x.\ x \neq a \Longrightarrow norm\ (x - a) < R \Longrightarrow f\ x = g\ x$
shows $g\ -a\rightarrow\ l \Longrightarrow f\ -a\rightarrow\ l$
by (*rule metric-LIM-equal2* [*OF* - *assms*]) (*simp-all add*: *dist-norm*)

lemma *LIM-compose2*:
fixes $a :: 'a$::*real-normed-vector*
assumes *f*: $f\ -a\rightarrow\ b$
and *g*: $g\ -b\rightarrow\ c$
and *inj*: $\exists d{>}0.\ \forall x.\ x \neq a \wedge norm\ (x - a) < d \longrightarrow f\ x \neq b$
shows $(\lambda x.\ g\ (f\ x))\ -a\rightarrow\ c$

by (*rule metric-LIM-compose2* [*OF f g inj* [*folded dist-norm*]])

lemma *real-LIM-sandwich-zero*:
fixes *f g* :: *'a*::*topological-space* ⇒ *real*
assumes *f*: *f* −*a*→ *0*
and *1*: ⋀*x*. *x* ≠ *a* ⟹ *0* ≤ *g x*
and *2*: ⋀*x*. *x* ≠ *a* ⟹ *g x* ≤ *f x*
shows *g* −*a*→ *0*
proof (*rule LIM-imp-LIM* [*OF f*])
fix *x*
assume *x*: *x* ≠ *a*
with *1* **have** *norm* (*g x* − *0*) = *g x* **by** *simp*
also have *g x* ≤ *f x* **by** (*rule 2* [*OF x*])
also have *f x* ≤ |*f x*| **by** (*rule abs-ge-self*)
also have |*f x*| = *norm* (*f x* − *0*) **by** *simp*
finally show *norm* (*g x* − *0*) ≤ *norm* (*f x* − *0*) **.**
qed

108.12 Continuity

lemma *LIM-isCont-iff*: (*f* −*a*→ *f a*) = ((λ*h*. *f* (*a* + *h*)) −*0*→ *f a*)
for *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*topological-space*
by (*rule iffI* [*OF LIM-offset-zero LIM-offset-zero-cancel*])

lemma *isCont-iff*: *isCont f x* = (λ*h*. *f* (*x* + *h*)) −*0*→ *f x*
for *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*topological-space*
by (*simp add*: *isCont-def LIM-isCont-iff*)

lemma *isCont-LIM-compose2*:
fixes *a* :: *'a*::*real-normed-vector*
assumes *f* [*unfolded isCont-def*]: *isCont f a*
and *g*: *g* −*f a*→ *l*
and *inj*: ∃ *d*>*0*. ∀ *x*. *x* ≠ *a* ∧ *norm* (*x* − *a*) < *d* ⟶ *f x* ≠ *f a*
shows (λ*x*. *g* (*f x*)) −*a*→ *l*
by (*rule LIM-compose2* [*OF f g inj*])

lemma *isCont-norm* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *norm* (*f x*)) *a*
for *f* :: *'a*::*t2-space* ⇒ *'b*::*real-normed-vector*
by (*fact continuous-norm*)

lemma *isCont-rabs* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. |*f x*|) *a*
for *f* :: *'a*::*t2-space* ⇒ *real*
by (*fact continuous-rabs*)

lemma *isCont-add* [*simp*]: *isCont f a* ⟹ *isCont g a* ⟹ *isCont* (λ*x*. *f x* + *g x*) *a*
for *f* :: *'a*::*t2-space* ⇒ *'b*::*topological-monoid-add*
by (*fact continuous-add*)

lemma *isCont-minus* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. − *f x*) *a*

for $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-vector*
by (*fact continuous-minus*)

lemma *isCont-diff* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont g a* $\Longrightarrow$ *isCont* $(\lambda x.\ f\ x - g\ x)$ *a*
for $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-vector*
by (*fact continuous-diff*)

lemma *isCont-mult* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont g a* $\Longrightarrow$ *isCont* $(\lambda x.\ f\ x * g\ x)$ *a*
for *f g* $:: 'a$::*t2-space* $\Rightarrow$ $'b$::*real-normed-algebra*
by (*fact continuous-mult*)

lemma (**in** *bounded-linear*) *isCont*: *isCont g a* $\Longrightarrow$ *isCont* $(\lambda x.\ f\ (g\ x))$ *a*
by (*fact continuous*)

lemma (**in** *bounded-bilinear*) *isCont*: *isCont f a* $\Longrightarrow$ *isCont g a* $\Longrightarrow$ *isCont* $(\lambda x.\ f\ x **\ g\ x)$ *a*
by (*fact continuous*)

lemmas *isCont-scaleR* [*simp*] =
bounded-bilinear.isCont [*OF bounded-bilinear-scaleR*]

lemmas *isCont-of-real* [*simp*] =
bounded-linear.isCont [*OF bounded-linear-of-real*]

lemma *isCont-power* [*simp*]: *isCont f a* $\Longrightarrow$ *isCont* $(\lambda x.\ f\ x\ \hat{}\ n)$ *a*
for $f :: 'a$::*t2-space* $\Rightarrow$ $'b$::{*power*,*real-normed-algebra*}
by (*fact continuous-power*)

lemma *isCont-sum* [*simp*]: $\forall\, i{\in}A.$ *isCont* $(f\ i)$ *a* $\Longrightarrow$ *isCont* $(\lambda x.\ \sum i{\in}A.\ f\ i\ x)$ *a*
for $f :: 'a \Rightarrow 'b$::*t2-space* $\Rightarrow$ $'c$::*topological-comm-monoid-add*
by (*auto intro*: *continuous-sum*)

108.13 Uniform Continuity

lemma *uniformly-continuous-on-def*:
fixes $f :: 'a$::*metric-space* $\Rightarrow$ $'b$::*metric-space*
shows *uniformly-continuous-on s f* $\longleftrightarrow$
$(\forall\, e{>}0.\ \exists\, d{>}0.\ \forall\, x{\in}s.\ \forall\, x'{\in}s.\ dist\ x'\ x < d \longrightarrow dist\ (f\ x')\ (f\ x) < e)$
unfolding *uniformly-continuous-on-uniformity*
uniformity-dist filterlim-INF filterlim-principal eventually-inf-principal
by (*force simp*: *Ball-def uniformity-dist*[*symmetric*] *eventually-uniformity-metric*)

abbreviation *isUCont* :: [$'a$::*metric-space* $\Rightarrow$ $'b$::*metric-space*] $\Rightarrow$ *bool*
where *isUCont f* $\equiv$ *uniformly-continuous-on UNIV f*

lemma *isUCont-def*: *isUCont f* $\longleftrightarrow$ $(\forall\, r{>}0.\ \exists\, s{>}0.\ \forall\, x\ y.\ dist\ x\ y < s \longrightarrow dist\ (f\ x)\ (f\ y) < r)$
by (*auto simp*: *uniformly-continuous-on-def dist-commute*)

lemma *isUCont-isCont*: *isUCont f* $\Longrightarrow$ *isCont f x*
by (*drule uniformly-continuous-imp-continuous*) (*simp add*: *continuous-on-eq-continuous-at*)

lemma *uniformly-continuous-on-Cauchy*:
fixes *f* :: *'a*::*metric-space* $\Rightarrow$ *'b*::*metric-space*
assumes *uniformly-continuous-on S f Cauchy X* $\bigwedge n$. *X n* $\in$ *S*
shows *Cauchy* (λn. *f* (*X n*))
using *assms*
unfolding *uniformly-continuous-on-def* **by** (*meson Cauchy-def*)

lemma *isUCont-Cauchy*: *isUCont f* $\Longrightarrow$ *Cauchy X* $\Longrightarrow$ *Cauchy* (λn. *f* (*X n*))
by (*rule uniformly-continuous-on-Cauchy*[**where** *S*=*UNIV* **and** *f*=*f*]) *simp-all*

lemma (**in** *bounded-linear*) *isUCont*: *isUCont f*
unfolding *isUCont-def dist-norm*
proof (*intro allI impI*)
fix *r* :: *real*
assume *r*: *0* < *r*
obtain *K* **where** *K*: *0* < *K* **and** *norm-le*: *norm* (*f x*) $\leq$ *norm x* $*$ *K* **for** *x*
using *pos-bounded* **by** *blast*
show $\exists$ *s*>*0*. $\forall$ *x y*. *norm* (*x* − *y*) < *s* $\longrightarrow$ *norm* (*f x* − *f y*) < *r*
proof (*rule exI*, *safe*)
from *r K* **show** *0* < *r* / *K* **by** *simp*
next
fix *x y* :: *'a*
assume *xy*: *norm* (*x* − *y*) < *r* / *K*
have *norm* (*f x* − *f y*) = *norm* (*f* (*x* − *y*)) **by** (*simp only*: *diff*)
also have ... $\leq$ *norm* (*x* − *y*) $*$ *K* **by** (*rule norm-le*)
also from *K xy* **have** ... < *r* **by** (*simp only*: *pos-less-divide-eq*)
finally show *norm* (*f x* − *f y*) < *r* **.**
qed
qed

lemma (**in** *bounded-linear*) *Cauchy*: *Cauchy X* $\Longrightarrow$ *Cauchy* (λn. *f* (*X n*))
by (*rule isUCont* [*THEN isUCont-Cauchy*])

lemma *LIM-less-bound*:
fixes *f* :: *real* $\Rightarrow$ *real*
assumes *ev*: *b* < *x* $\forall$ *x'* $\in$ { *b* <..< *x*}. *0* $\leq$ *f x'* **and** *isCont f x*
shows *0* $\leq$ *f x*
proof (*rule tendsto-lowerbound*)
show (*f* $\longrightarrow$ *f x*) (*at-left x*)
using ‹*isCont f x*› **by** (*simp add*: *filterlim-at-split isCont-def*)
show *eventually* (λx. *0* $\leq$ *f x*) (*at-left x*)
using *ev* **by** (*auto simp*: *eventually-at dist-real-def intro*!: *exI*[*of* - *x* − *b*])
qed *simp*

108.14 Nested Intervals and Bisection – Needed for Compactness

lemma *nested-sequence-unique*:
assumes $\forall n.\ f\ n \le f\ (Suc\ n)\ \forall n.\ g\ (Suc\ n) \le g\ n\ \forall n.\ f\ n \le g\ n\ (\lambda n.\ f\ n - g\ n) \longrightarrow 0$
shows $\exists l{::}real.\ ((\forall n.\ f\ n \le l) \wedge f \longrightarrow l) \wedge ((\forall n.\ l \le g\ n) \wedge g \longrightarrow l)$
proof −
have *incseq f* **unfolding** *incseq-Suc-iff* **by** *fact*
have *decseq g* **unfolding** *decseq-Suc-iff* **by** *fact*
have $f\ n \le g\ 0$ **for** n
proof −
from ‹*decseq g*› **have** $g\ n \le g\ 0$
by (*rule decseqD*) *simp*
with ‹$\forall n.\ f\ n \le g\ n$›[*THEN spec*, *of n*] **show** *?thesis*
by *auto*
qed
then obtain u **where** $f \longrightarrow u\ \forall i.\ f\ i \le u$
using *incseq-convergent*[*OF* ‹*incseq f*›] **by** *auto*
moreover have $f\ 0 \le g\ n$ **for** n
proof −
from ‹*incseq f*› **have** $f\ 0 \le f\ n$ **by** (*rule incseqD*) *simp*
with ‹$\forall n.\ f\ n \le g\ n$›[*THEN spec*, *of n*] **show** *?thesis*
by *simp*
qed
then obtain l **where** $g \longrightarrow l\ \forall i.\ l \le g\ i$
using *decseq-convergent*[*OF* ‹*decseq g*›] **by** *auto*
moreover note *LIMSEQ-unique*[*OF assms*(*4*) *tendsto-diff*[*OF* ‹$f \longrightarrow u$› ‹$g \longrightarrow l$›]]
ultimately show *?thesis* **by** *auto*
qed

lemma *Bolzano*[*consumes 1*, *case-names trans local*]:
fixes $P :: real \Rightarrow real \Rightarrow bool$
assumes [*arith*]: $a \le b$
and *trans*: $\bigwedge a\ b\ c.\ P\ a\ b \Longrightarrow P\ b\ c \Longrightarrow a \le b \Longrightarrow b \le c \Longrightarrow P\ a\ c$
and *local*: $\bigwedge x.\ a \le x \Longrightarrow x \le b \Longrightarrow \exists d{>}0.\ \forall a\ b.\ a \le x \wedge x \le b \wedge b - a < d \longrightarrow P\ a\ b$
shows $P\ a\ b$
proof −
define *bisect* **where** $bisect \equiv \lambda(x,y).\ if\ P\ x\ ((x+y)\ /\ 2)\ then\ ((x+y)/2,\ y)\ else\ (x,\ (x+y)/2)$
define $l\ u$ **where** $l\ n \equiv fst\ ((bisect\widehat{\ }n)(a,b))$ **and** $u\ n \equiv snd\ ((bisect\widehat{\ }n)(a,b))$ **for** n
have l[*simp*]: $l\ 0 = a\ \bigwedge n.\ l\ (Suc\ n) = (if\ P\ (l\ n)\ ((l\ n + u\ n)\ /\ 2)\ then\ (l\ n + u\ n)\ /\ 2\ else\ l\ n)$
and u[*simp*]: $u\ 0 = b\ \bigwedge n.\ u\ (Suc\ n) = (if\ P\ (l\ n)\ ((l\ n + u\ n)\ /\ 2)\ then\ u\ n\ else\ (l\ n + u\ n)\ /\ 2)$
by (*simp-all add*: *l-def u-def bisect-def split*: *prod.split*)

```
  have [simp]: l n ≤ u n for n by (induct n) auto

  have ∃x. ((∀n. l n ≤ x) ∧ l ⟶ x) ∧ ((∀n. x ≤ u n) ∧ u ⟶ x)
  proof (safe intro!: nested-sequence-unique)
    show l n ≤ l (Suc n) u (Suc n) ≤ u n for n
      by (induct n) auto
  next
    have l n − u n = (a − b) / 2^n for n
      by (induct n) (auto simp: field-simps)
    then show (λn. l n − u n) ⟶ 0
      by (simp add: LIMSEQ-divide-realpow-zero)
  qed fact
  then obtain x where x: ⋀n. l n ≤ x ⋀n. x ≤ u n and l ⟶ x u ⟶ x
    by auto
  obtain d where 0 < d and d: a ≤ x ⟹ x ≤ b ⟹ b − a < d ⟹ P a b for
a b
    using ‹l 0 ≤ x› ‹x ≤ u 0› local[of x] by auto

  show P a b
  proof (rule ccontr)
    assume ¬ P a b
    have ¬ P (l n) (u n) for n
    proof (induct n)
      case 0
      then show ?case
        by (simp add: ‹¬ P a b›)
    next
      case (Suc n)
      with trans[of l n (l n + u n) / 2 u n] show ?case
        by auto
    qed
    moreover
    {
      have eventually (λn. x − d / 2 < l n) sequentially
        using ‹0 < d› ‹l ⟶ x› by (intro order-tendstoD[of - x]) auto
      moreover have eventually (λn. u n < x + d / 2) sequentially
        using ‹0 < d› ‹u ⟶ x› by (intro order-tendstoD[of - x]) auto
      ultimately have eventually (λn. P (l n) (u n)) sequentially
      proof eventually-elim
        case (elim n)
        from add-strict-mono[OF this] have u n − l n < d by simp
        with x show P (l n) (u n) by (rule d)
      qed
    }
    ultimately show False by simp
  qed
qed

lemma compact-Icc[simp, intro]: compact {a .. b::real}
```

```
proof (cases a ≤ b, rule compactI)
  fix C
  assume C: a ≤ b ∀ t∈C. open t {a..b} ⊆ ⋃ C
  define T where T = {a .. b}
  from C(1,3) show ∃ C'⊆C. finite C' ∧ {a..b} ⊆ ⋃ C'
  proof (induct rule: Bolzano)
    case (trans a b c)
    then have *: {a..c} = {a..b} ∪ {b..c}
      by auto
    with trans obtain C1 C2
      where C1⊆C finite C1 {a..b} ⊆ ⋃ C1 C2⊆C finite C2 {b..c} ⊆ ⋃ C2
      by auto
    with trans show ?case
      unfolding * by (intro exI[of - C1 ∪ C2]) auto
  next
    case (local x)
    with C have x ∈ ⋃ C by auto
    with C(2) obtain c where x ∈ c open c c ∈ C
      by auto
    then obtain e where 0 < e {x − e <..< x + e} ⊆ c
      by (auto simp: open-dist dist-real-def subset-eq Ball-def abs-less-iff)
    with ‹c ∈ C› show ?case
      by (safe intro!: exI[of - e/2] exI[of - {c}]) auto
  qed
qed simp


lemma continuous-image-closed-interval:
  fixes a b and f :: real ⇒ real
  defines S ≡ {a..b}
  assumes a ≤ b and f: continuous-on S f
  shows ∃ c d. f`S = {c..d} ∧ c ≤ d
proof −
  have S: compact S S ≠ {}
    using ‹a ≤ b› by (auto simp: S-def)
  obtain c where c ∈ S ∀ d∈S. f d ≤ f c
    using continuous-attains-sup[OF S f] by auto
  moreover obtain d where d ∈ S ∀ c∈S. f d ≤ f c
    using continuous-attains-inf[OF S f] by auto
  moreover have connected (f`S)
    using connected-continuous-image[OF f] connected-Icc by (auto simp: S-def)
  ultimately have f ` S = {f d .. f c} ∧ f d ≤ f c
    by (auto simp: connected-iff-interval)
  then show ?thesis
    by auto
qed

lemma open-Collect-positive:
  fixes f :: 'a::topological-space ⇒ real
```

assumes f: *continuous-on s f*
shows $\exists A.$ *open* $A \wedge A \cap s = \{x{\in}s.\ 0 < f\ x\}$
using *continuous-on-open-invariant*[*THEN iffD1*, *OF f*, *rule-format*, *of* $\{0 <..\}$]
by (*auto simp*: *Int-def field-simps*)

lemma *open-Collect-less-Int*:
fixes $f\ g$ :: *'a*::*topological-space* ⇒ *real*
assumes f: *continuous-on s f*
and g: *continuous-on s g*
shows $\exists A.$ *open* $A \wedge A \cap s = \{x{\in}s.\ f\ x < g\ x\}$
using *open-Collect-positive*[*OF continuous-on-diff*[*OF g f*]] **by** (*simp add*: *field-simps*)

108.15 Boundedness of continuous functions

By bisection, function continuous on closed interval is bounded above

lemma *isCont-eq-Ub*:
fixes f :: *real* ⇒ *'a*::*linorder-topology*
shows $a \le b \Longrightarrow \forall x{::}real.\ a \le x \wedge x \le b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists M.\ (\forall x.\ a \le x \wedge x \le b \longrightarrow f\ x \le M) \wedge (\exists x.\ a \le x \wedge x \le b \wedge f\ x = M)$
using *continuous-attains-sup*[*of* $\{a..b\}$ f]
by (*auto simp*: *continuous-at-imp-continuous-on Ball-def Bex-def*)

lemma *isCont-eq-Lb*:
fixes f :: *real* ⇒ *'a*::*linorder-topology*
shows $a \le b \Longrightarrow \forall x.\ a \le x \wedge x \le b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists M.\ (\forall x.\ a \le x \wedge x \le b \longrightarrow M \le f\ x) \wedge (\exists x.\ a \le x \wedge x \le b \wedge f\ x = M)$
using *continuous-attains-inf*[*of* $\{a..b\}$ f]
by (*auto simp*: *continuous-at-imp-continuous-on Ball-def Bex-def*)

lemma *isCont-bounded*:
fixes f :: *real* ⇒ *'a*::*linorder-topology*
shows $a \le b \Longrightarrow \forall x.\ a \le x \wedge x \le b \longrightarrow isCont\ f\ x \Longrightarrow \exists M.\ \forall x.\ a \le x \wedge x \le b \longrightarrow f\ x \le M$
using *isCont-eq-Ub*[*of a b f*] **by** *auto*

lemma *isCont-has-Ub*:
fixes f :: *real* ⇒ *'a*::*linorder-topology*
shows $a \le b \Longrightarrow \forall x.\ a \le x \wedge x \le b \longrightarrow isCont\ f\ x \Longrightarrow$
$\exists M.\ (\forall x.\ a \le x \wedge x \le b \longrightarrow f\ x \le M) \wedge (\forall N.\ N < M \longrightarrow (\exists x.\ a \le x \wedge x \le b \wedge N < f\ x))$
using *isCont-eq-Ub*[*of a b f*] **by** *auto*

lemma *isCont-Lb-Ub*:
fixes f :: *real* ⇒ *real*
assumes $a \le b\ \forall x.\ a \le x \wedge x \le b \longrightarrow isCont\ f\ x$
shows $\exists L\ M.\ (\forall x.\ a \le x \wedge x \le b \longrightarrow L \le f\ x \wedge f\ x \le M) \wedge$
$(\forall y.\ L \le y \wedge y \le M \longrightarrow (\exists x.\ a \le x \wedge x \le b \wedge (f\ x = y)))$
proof −
obtain M **where** M: $a \le M\ M \le b\ \forall x.\ a \le x \wedge x \le b \longrightarrow f\ x \le f\ M$

using *isCont-eq-Ub*[*OF assms*] **by** *auto*
obtain *L* **where** *L*: $a \leq L$ $L \leq b$ $\forall x.\ a \leq x \wedge x \leq b \longrightarrow f\ L \leq f\ x$
using *isCont-eq-Lb*[*OF assms*] **by** *auto*
have $(\forall x.\ a \leq x \wedge x \leq b \longrightarrow f\ L \leq f\ x \wedge f\ x \leq f\ M)$
using *M L* **by** *simp*
moreover
have $(\forall y.\ f\ L \leq y \wedge y \leq f\ M \longrightarrow (\exists x{\geq}a.\ x \leq b \wedge f\ x = y))$
proof (*cases* $L \leq M$)
case *True* **then show** *?thesis*
using *IVT*[*of f L - M*] *M L assms* **by** (*metis order.trans*)
next
case *False* **then show** *?thesis*
using *IVT2*[*of f L - M*]
by (*metis L*(*2*) *M*(*1*) *assms*(*2*) *le-cases order.trans*)
qed
ultimately show *?thesis*
by *blast*
qed

Continuity of inverse function.

lemma *isCont-inverse-function*:
fixes *f g* :: *real* ⇒ *real*
assumes *d*: $0 < d$
and *inj*: $\bigwedge z.\ |z-x| \leq d \Longrightarrow g\ (f\ z) = z$
and *cont*: $\bigwedge z.\ |z-x| \leq d \Longrightarrow$ *isCont f z*
shows *isCont g* (*f x*)
proof −
let *?A* = *f* $(x - d)$
let *?B* = *f* $(x + d)$
let *?D* = $\{x - d..x + d\}$

have *f*: *continuous-on ?D f*
using *cont* **by** (*intro continuous-at-imp-continuous-on ballI*) *auto*
then have *g*: *continuous-on* (*f*‘*?D*) *g*
using *inj* **by** (*intro continuous-on-inv*) *auto*

from *d f* **have** $\{min\ ?A\ ?B <..< max\ ?A\ ?B\} \subseteq f\ `\ ?D$
by (*intro connected-contains-Ioo connected-continuous-image*) (*auto split*: *split-min split-max*)
with *g* **have** *continuous-on* $\{min\ ?A\ ?B <..< max\ ?A\ ?B\}$ *g*
by (*rule continuous-on-subset*)
moreover
have $(?A < f\ x \wedge f\ x < ?B) \vee (?B < f\ x \wedge f\ x < ?A)$
using *d inj* **by** (*intro continuous-inj-imp-mono*[*OF - - f*] *inj-on-imageI2*[*of g, OF inj-onI*]) *auto*
then have $f\ x \in \{min\ ?A\ ?B <..< max\ ?A\ ?B\}$
by *auto*
ultimately show *?thesis*
by (*simp add*: *continuous-on-eq-continuous-at*)

qed

lemma *isCont-inverse-function2*:
fixes $f\ g :: real \Rightarrow real$
shows
$[\![a < x;\ x < b;$
$\bigwedge z.\ [\![a \leq z;\ z \leq b]\!] \Longrightarrow g\ (f\ z) = z;$
$\bigwedge z.\ [\![a \leq z;\ z \leq b]\!] \Longrightarrow isCont\ f\ z]\!] \Longrightarrow isCont\ g\ (f\ x)$
apply (*rule isCont-inverse-function* [**where** *f*=*f* **and** *d*=*min* $(x - a)\ (b - x)$])
apply (*simp-all add*: *abs-le-iff*)
done

Bartle/Sherbert: Introduction to Real Analysis, Theorem 4.2.9, p. 110.

lemma *LIM-fun-gt-zero*: $f\ -c\rightarrow\ l \Longrightarrow 0 < l \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow 0 < f\ x)$
for $f :: real \Rightarrow real$
by (*force simp*: *dest*: *LIM-D*)

lemma *LIM-fun-less-zero*: $f\ -c\rightarrow\ l \Longrightarrow l < 0 \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow f\ x < 0)$
for $f :: real \Rightarrow real$
by (*drule LIM-D* [**where** *r*=$-l$]) *force*+

lemma *LIM-fun-not-zero*: $f\ -c\rightarrow\ l \Longrightarrow l \neq 0 \Longrightarrow \exists\, r.\ 0 < r \wedge (\forall\, x.\ x \neq c \wedge |c - x| < r \longrightarrow f\ x \neq 0)$
for $f :: real \Rightarrow real$
using *LIM-fun-gt-zero*[*of f l c*] *LIM-fun-less-zero*[*of f l c*] **by** (*auto simp*: *neq-iff*)

lemma *Lim-topological*:
$(f \longrightarrow l)\ net \longleftrightarrow$
trivial-limit net $\vee$ $(\forall\, S.\ open\ S \longrightarrow l \in S \longrightarrow eventually\ (\lambda x.\ f\ x \in S)\ net)$
unfolding *tendsto-def trivial-limit-eq* **by** *auto*

lemma *eventually-within-Un*:
eventually P $(at\ x\ within\ (s \cup t)) \longleftrightarrow$
eventually P $(at\ x\ within\ s) \wedge$ *eventually P* $(at\ x\ within\ t)$
unfolding *eventually-at-filter*
by (*auto elim*!: *eventually-rev-mp*)

lemma *Lim-within-Un*:
$(f \longrightarrow l)\ (at\ x\ within\ (s \cup t)) \longleftrightarrow$
$(f \longrightarrow l)\ (at\ x\ within\ s) \wedge (f \longrightarrow l)\ (at\ x\ within\ t)$
unfolding *tendsto-def*
by (*auto simp*: *eventually-within-Un*)

end

theory *Inequalities*

imports *Real-Vector-Spaces*
begin

lemma *Chebyshev-sum-upper*:
fixes *a b*::*nat* ⇒ *'a*::*linordered-idom*
assumes ⋀*i j*. *i* ≤ *j* ⟹ *j* < *n* ⟹ *a i* ≤ *a j*
assumes ⋀*i j*. *i* ≤ *j* ⟹ *j* < *n* ⟹ *b i* ≥ *b j*
shows *of-nat n* ∗ (∑ *k*=*0*..<*n*. *a k* ∗ *b k*) ≤ (∑ *k*=*0*..<*n*. *a k*) ∗ (∑ *k*=*0*..<*n*. *b k*)
proof −
let *?S* = (∑ *j*=*0*..<*n*. (∑ *k*=*0*..<*n*. (*a j* − *a k*) ∗ (*b j* − *b k*)))
have *2* ∗ (*of-nat n* ∗ (∑ *j*=*0*..<*n*. (*a j* ∗ *b j*)) − (∑ *j*=*0*..<*n*. *b j*) ∗ (∑ *k*=*0*..<*n*. *a k*)) = *?S*
by (*simp only*: *one-add-one*[*symmetric*] *algebra-simps*)
(*simp add*: *algebra-simps sum-subtractf sum.distrib sum.swap*[*of* λ*i j*. *a i* ∗ *b j*] *sum-distrib-left*)
also
{ fix *i j*::*nat* **assume** *i*<*n j*<*n*
hence *a i* − *a j* ≤ *0* ∧ *b i* − *b j* ≥ *0* ∨ *a i* − *a j* ≥ *0* ∧ *b i* − *b j* ≤ *0*
using *assms* **by** (*cases i* ≤ *j*) (*auto simp*: *algebra-simps*)
} then have *?S* ≤ *0*
by (*auto intro*!: *sum-nonpos simp*: *mult-le-0-iff*)
finally show *?thesis* **by** (*simp add*: *algebra-simps*)
qed

lemma *Chebyshev-sum-upper-nat*:
fixes *a b* :: *nat* ⇒ *nat*
shows (⋀*i j*. ⟦ *i*≤*j*; *j*<*n* ⟧ ⟹ *a i* ≤ *a j*) ⟹
(⋀*i j*. ⟦ *i*≤*j*; *j*<*n* ⟧ ⟹ *b i* ≥ *b j*) ⟹
n ∗ (∑ *i*=*0*..<*n*. *a i* ∗ *b i*) ≤ (∑ *i*=*0*..<*n*. *a i*) ∗ (∑ *i*=*0*..<*n*. *b i*)
using *Chebyshev-sum-upper*[**where** *'a*=*real*, *of n a b*]
by (*simp del*: *of-nat-mult of-nat-sum add*: *of-nat-mult*[*symmetric*] *of-nat-sum*[*symmetric*])

end

109 Infinite Series

theory *Series*
imports *Limits Inequalities*
begin

109.1 Definition of infinite summability

definition *sums* :: (*nat* ⇒ *'a*::{*topological-space*, *comm-monoid-add*}) ⇒ *'a* ⇒ *bool*
(**infixr** ‹*sums*› *80*)
where *f sums s* ⟷ (λ*n*. ∑ *i*<*n*. *f i*) ⟶ *s*

definition *summable* :: (*nat* ⇒ *'a*::{*topological-space*, *comm-monoid-add*}) ⇒ *bool*
where *summable f* ⟷ (∃ *s*. *f sums s*)

definition *suminf* :: (*nat* ⇒ *'a*::{*topological-space*, *comm-monoid-add*}) ⇒ *'a*
(**binder** ‹∑› *10*)
where *suminf f* = (*THE s*. *f sums s*)

Variants of the definition

lemma *sums-def'*: *f sums s* ⟷ (λ*n*. ∑ *i* = *0*..*n*. *f i*) ⟶ *s*
unfolding *sums-def*
using *LIMSEQ-lessThan-iff-atMost atMost-atLeast0* **by** *auto*

lemma *sums-def-le*: *f sums s* ⟷ (λ*n*. ∑ *i*≤*n*. *f i*) ⟶ *s*
by (*simp add*: *sums-def' atMost-atLeast0*)

lemma *bounded-imp-summable*:
fixes *a* :: *nat* ⇒ *'a*::{*conditionally-complete-linorder*,*linorder-topology*,*linordered-comm-semiring-strict*}
assumes *0*: ⋀*n*. *a n* ≥ *0* **and** *bounded*: ⋀*n*. (∑ *k*≤*n*. *a k*) ≤ *B*
shows *summable a*
proof −
have *bdd-above* (*range*(λ*n*. ∑ *k*≤*n*. *a k*))
by (*meson bdd-aboveI2 bounded*)
moreover have *incseq* (λ*n*. ∑ *k*≤*n*. *a k*)
by (*simp add*: *mono-def 0 sum-mono2*)
ultimately obtain *s* **where** (λ*n*. ∑ *k*≤*n*. *a k*) ⟶ *s*
using *LIMSEQ-incseq-SUP* **by** *blast*
then show *?thesis*
by (*auto simp*: *sums-def-le summable-def*)
qed

109.2 Infinite summability on topological monoids

lemma *sums-subst*[*trans*]: *f* = *g* ⟹ *g sums z* ⟹ *f sums z*
by *simp*

lemma *sums-cong*: (⋀*n*. *f n* = *g n*) ⟹ *f sums c* ⟷ *g sums c*
by *presburger*

lemma *sums-summable*: *f sums l* ⟹ *summable f*
by (*simp add*: *sums-def summable-def*, *blast*)

lemma *summable-iff-convergent*: *summable f* ⟷ *convergent* (λ*n*. ∑ *i*<*n*. *f i*)
by (*simp add*: *summable-def sums-def convergent-def*)

lemma *summable-iff-convergent'*: *summable f* ⟷ *convergent* (λ*n*. *sum f* {..*n*})
by (*simp add*: *convergent-def summable-def sums-def-le*)

lemma *suminf-eq-lim*: *suminf f* = *lim* (λ*n*. ∑ *i*<*n*. *f i*)
by (*simp add*: *suminf-def sums-def lim-def*)

lemma *sums-zero*[*simp*, *intro*]: (λ*n*. *0*) *sums 0*

unfolding *sums-def* **by** *simp*

lemma *summable-zero*[*simp*, *intro*]: *summable* (λ*n*. *0*)
by (*rule sums-zero* [*THEN sums-summable*])

lemma *sums-group*: *f sums s* ⟹ *0* < *k* ⟹ (λ*n*. *sum f* {*n* ∗ *k* ..< *n* ∗ *k* + *k*}) *sums s*
apply (*simp only*: *sums-def sum.nat-group tendsto-def eventually-sequentially*)
apply (*erule all-forward imp-forward exE*| *assumption*)+
by (*metis le-square mult.commute mult.left-neutral mult-le-cancel2 mult-le-mono*)

lemma *suminf-cong*: (⋀*n*. *f n* = *g n*) ⟹ *suminf f* = *suminf g*
by *presburger*

lemma *summable-cong*:
fixes *f g* :: *nat* ⇒ *'a*::*real-normed-vector*
assumes *eventually* (λ*x*. *f x* = *g x*) *sequentially*
shows *summable f* = *summable g*
proof −
from *assms* **obtain** *N* **where** *N*: ∀ *n*≥*N*. *f n* = *g n*
by (*auto simp*: *eventually-at-top-linorder*)
define *C* **where** *C* = (∑ *k*<*N*. *f k* − *g k*)
from *eventually-ge-at-top*[*of N*]
have *eventually* (λ*n*. *sum f* {..<*n*} = *C* + *sum g* {..<*n*}) *sequentially*
proof *eventually-elim*
case (*elim n*)
then have {..<*n*} = {..<*N*} ∪ {*N*..<*n*}
by *auto*
also have *sum f* ... = *sum f* {..<*N*} + *sum f* {*N*..<*n*}
by (*intro sum.union-disjoint*) *auto*
also from *N* **have** *sum f* {*N*..<*n*} = *sum g* {*N*..<*n*}
by (*intro sum.cong*) *simp-all*
also have *sum f* {..<*N*} + *sum g* {*N*..<*n*} = *C* + (*sum g* {..<*N*} + *sum g* {*N*..<*n*})
unfolding *C-def* **by** (*simp add*: *algebra-simps sum-subtractf*)
also have *sum g* {..<*N*} + *sum g* {*N*..<*n*} = *sum g* ({..<*N*} ∪ {*N*..<*n*})
by (*intro sum.union-disjoint* [*symmetric*]) *auto*
also from *elim* **have** {..<*N*} ∪ {*N*..<*n*} = {..<*n*}
by *auto*
finally show *sum f* {..<*n*} = *C* + *sum g* {..<*n*} .
qed
from *convergent-cong*[*OF this*] **show** *?thesis*
by (*simp add*: *summable-iff-convergent convergent-add-const-iff*)
qed

lemma *sums-finite*:
assumes [*simp*]: *finite N*
and *f*: ⋀*n*. *n* ∉ *N* ⟹ *f n* = *0*
shows *f sums* (∑ *n*∈*N*. *f n*)

proof −
have *eq*: *sum f* {..<*n* + *Suc* (*Max N*)} = *sum f N* **for** *n*
by (*rule sum.mono-neutral-right*) (*auto simp*: *add-increasing less-Suc-eq-le f*)
show *?thesis*
unfolding *sums-def*
by (*rule LIMSEQ-offset*[*of* - *Suc* (*Max N*)])
(*simp add*: *eq atLeast0LessThan del*: *add-Suc-right*)
qed

corollary *sums-0*: (⋀*n*. *f n* = *0*) ⟹ (*f sums 0*)
by (*metis* (*no-types*) *finite.emptyI sum.empty sums-finite*)

lemma *summable-finite*: *finite N* ⟹ (⋀*n*. *n* ∉ *N* ⟹ *f n* = *0*) ⟹ *summable f*
by (*rule sums-summable*) (*rule sums-finite*)

lemma *sums-If-finite-set*: *finite A* ⟹ (λ*r*. *if r* ∈ *A then f r else 0*) *sums* (∑ *r*∈*A*. *f r*)
using *sums-finite*[*of A* (λ*r*. *if r* ∈ *A then f r else 0*)] **by** *simp*

lemma *summable-If-finite-set*[*simp*, *intro*]: *finite A* ⟹ *summable* (λ*r*. *if r* ∈ *A then f r else 0*)
by (*rule sums-summable*) (*rule sums-If-finite-set*)

lemma *sums-If-finite*: *finite* {*r*. *P r*} ⟹ (λ*r*. *if P r then f r else 0*) *sums* (∑ *r* | *P r*. *f r*)
using *sums-If-finite-set*[*of* {*r*. *P r*}] **by** *simp*

lemma *summable-If-finite*[*simp*, *intro*]: *finite* {*r*. *P r*} ⟹ *summable* (λ*r*. *if P r then f r else 0*)
by (*rule sums-summable*) (*rule sums-If-finite*)

lemma *sums-single*: (λ*r*. *if r* = *i then f r else 0*) *sums f i*
using *sums-If-finite*[*of* λ*r*. *r* = *i*] **by** *simp*

lemma *summable-single*[*simp*, *intro*]: *summable* (λ*r*. *if r* = *i then f r else 0*)
by (*rule sums-summable*) (*rule sums-single*)

context
fixes *f* :: *nat* ⇒ *'a*::{*t2-space*,*comm-monoid-add*}
begin

lemma *summable-sums*[*intro*]: *summable f* ⟹ *f sums* (*suminf f*)
by (*simp add*: *summable-def sums-def suminf-def*)
(*metis convergent-LIMSEQ-iff convergent-def lim-def*)

lemma *summable-LIMSEQ*: *summable f* ⟹ (λ*n*. ∑ *i*<*n*. *f i*) ⟶ *suminf f*
by (*rule summable-sums* [*unfolded sums-def*])

lemma *summable-LIMSEQ'*: *summable f* ⟹ (λ*n*. ∑ *i*≤*n*. *f i*) ⟶ *suminf f*

using *sums-def-le* **by** *blast*

lemma *sums-unique*: f *sums* $s \Longrightarrow s =$ *suminf* f
by (*metis limI suminf-eq-lim sums-def*)

lemma *sums-iff*: f *sums* $x \longleftrightarrow$ *summable* $f \wedge$ *suminf* $f = x$
by (*metis summable-sums sums-summable sums-unique*)

lemma *summable-sums-iff*: *summable* $f \longleftrightarrow f$ *sums* *suminf* f
by (*auto simp*: *sums-iff summable-sums*)

lemma *sums-unique2*: f *sums* $a \Longrightarrow f$ *sums* $b \Longrightarrow a = b$
for a b :: $'a$
by (*simp add*: *sums-iff*)

lemma *sums-Uniq*: $\exists_{\leq 1} a.$ f *sums* a
for a b :: $'a$
by (*simp add*: *sums-unique2 Uniq-def*)

lemma *suminf-finite*:
assumes N: *finite* N
and f: $\bigwedge n.$ $n \notin N \Longrightarrow f\ n = 0$
shows *suminf* $f = (\sum n{\in}N.\ f\ n)$
using *sums-finite*[*OF assms*, *THEN sums-unique*] **by** *simp*

end

lemma *suminf-zero*[*simp*]: *suminf* $(\lambda n.\ 0{::}'a{::}\{$*t2-space*, *comm-monoid-add*$\}) = 0$
by (*rule sums-zero* [*THEN sums-unique*, *symmetric*])

109.3 Infinite summability on ordered, topological monoids

lemma *sums-le*: $(\bigwedge n.\ f\ n \leq g\ n) \Longrightarrow f$ *sums* $s \Longrightarrow g$ *sums* $t \Longrightarrow s \leq t$
for f g :: *nat* $\Rightarrow$ $'a{::}\{$*ordered-comm-monoid-add*,*linorder-topology*$\}$
by (*rule LIMSEQ-le*) (*auto intro*: *sum-mono simp*: *sums-def*)

context
fixes f :: *nat* $\Rightarrow$ $'a{::}\{$*ordered-comm-monoid-add*,*linorder-topology*$\}$
begin

lemma *suminf-le*: $(\bigwedge n.\ f\ n \leq g\ n) \Longrightarrow$ *summable* $f \Longrightarrow$ *summable* $g \Longrightarrow$ *suminf* $f \leq$ *suminf* g
using *sums-le* **by** *blast*

lemma *sum-le-suminf*:
shows *summable* $f \Longrightarrow$ *finite* $I \Longrightarrow (\bigwedge n.\ n \in\!-\ I \Longrightarrow 0 \leq f\ n) \Longrightarrow$ *sum* $f\ I \leq$ *suminf* f
by (*rule sums-le*[*OF - sums-If-finite-set summable-sums*]) *auto*

lemma *suminf-nonneg*: *summable f* $\Longrightarrow$ ($\bigwedge$*n. 0* $\leq$ *f n*) $\Longrightarrow$ *0* $\leq$ *suminf f*
using *sum-le-suminf* **by** *force*

lemma *suminf-le-const*: *summable f* $\Longrightarrow$ ($\bigwedge$*n. sum f* {..<*n*} $\leq$ *x*) $\Longrightarrow$ *suminf f* $\leq$ *x*
by (*metis LIMSEQ-le-const2 summable-LIMSEQ*)

lemma *suminf-eq-zero-iff*:
assumes *summable f* **and** *pos*: $\bigwedge$*n. 0* $\leq$ *f n*
shows *suminf f* = *0* $\longleftrightarrow$ ($\forall$ *n. f n* = *0*)
proof
assume *L*: *suminf f* = *0*
then have *f*: (λ*n.* $\sum$ *i*<*n. f i*) $\longrightarrow$ *0*
using *summable-LIMSEQ*[*of f*] *assms* **by** *simp*
then have $\bigwedge$*i.* ($\sum$ *n*$\in${*i*}*. f n*) $\leq$ *0*
by (*metis L* ‹*summable f*› *order-refl pos sum.infinite sum-le-suminf*)
with *pos* **show** $\forall$ *n. f n* = *0*
by (*simp add*: *order.antisym*)
qed (*metis suminf-zero fun-eq-iff*)

lemma *suminf-pos-iff*: *summable f* $\Longrightarrow$ ($\bigwedge$*n. 0* $\leq$ *f n*) $\Longrightarrow$ *0* < *suminf f* $\longleftrightarrow$ ($\exists$ *i. 0* < *f i*)
using *sum-le-suminf*[*of* {}] *suminf-eq-zero-iff* **by** (*simp add*: *less-le*)

lemma *suminf-pos2*:
assumes *summable f* $\bigwedge$*n. 0* $\leq$ *f n 0* < *f i*
shows *0* < *suminf f*
proof −
have *0* < ($\sum$ *n*<*Suc i. f n*)
using *assms* **by** (*intro sum-pos2*[**where** *i*=*i*]) *auto*
also have ... $\leq$ *suminf f*
using *assms* **by** (*intro sum-le-suminf*) *auto*
finally show *?thesis* **.**
qed

lemma *suminf-pos*: *summable f* $\Longrightarrow$ ($\bigwedge$*n. 0* < *f n*) $\Longrightarrow$ *0* < *suminf f*
by (*intro suminf-pos2*[**where** *i*=*0*]) (*auto intro*: *less-imp-le*)

end

context
fixes *f* :: *nat* $\Rightarrow$ *'a*::{*ordered-cancel-comm-monoid-add*,*linorder-topology*}
begin

lemma *sum-less-suminf2*:
summable f $\Longrightarrow$ ($\bigwedge$*m. m*$\geq$*n* $\Longrightarrow$ *0* $\leq$ *f m*) $\Longrightarrow$ *n* $\leq$ *i* $\Longrightarrow$ *0* < *f i* $\Longrightarrow$ *sum f* {..<*n*} < *suminf f*
using *sum-le-suminf*[*of f* {..< *Suc i*}]
and *add-strict-increasing*[*of f i sum f* {..<*n*} *sum f* {..<*i*}]

and *sum-mono2*[*of* {..<*i*} {..<*n*} *f*]
by (*auto simp*: *less-imp-le ac-simps*)

lemma *sum-less-suminf*: *summable f* ⟹ (⋀*m*. *m*≥*n* ⟹ *0* < *f m*) ⟹ *sum f* {..<*n*} < *suminf f*
using *sum-less-suminf2*[*of n n*] **by** (*simp add*: *less-imp-le*)

end

lemma *summableI-nonneg-bounded*:
fixes *f* :: *nat* ⇒ *'a*::{*ordered-comm-monoid-add*,*linorder-topology*,*conditionally-complete-linorder*}
assumes *pos*[*simp*]: ⋀*n*. *0* ≤ *f n*
and *le*: ⋀*n*. (∑ *i*<*n*. *f i*) ≤ *x*
shows *summable f*
unfolding *summable-def sums-def* [*abs-def*]
proof (*rule exI LIMSEQ-incseq-SUP*)+
show *bdd-above* (*range* (λ*n*. *sum f* {..<*n*}))
using *le* **by** (*auto simp*: *bdd-above-def*)
show *incseq* (λ*n*. *sum f* {..<*n*})
by (*auto simp*: *mono-def intro*!: *sum-mono2*)
qed

lemma *summableI*[*intro*, *simp*]: *summable f*
for *f* :: *nat* ⇒ *'a*::{*canonically-ordered-monoid-add*,*linorder-topology*,*complete-linorder*}
by (*intro summableI-nonneg-bounded*[**where** *x*=*top*] *zero-le top-greatest*)

lemma *suminf-eq-SUP-real*:
assumes *X*: *summable X* ⋀*i*. *0* ≤ *X i* **shows** *suminf X* = (*SUP i*. ∑ *n*<*i*. *X n*::*real*)
by (*intro LIMSEQ-unique*[*OF summable-LIMSEQ*] *X LIMSEQ-incseq-SUP*)
(*auto intro*!: *bdd-aboveI2*[**where** *M*=∑ *i*. *X i*] *sum-le-suminf X monoI sum-mono2*)

109.4 Infinite summability on topological monoids

context
fixes *f g* :: *nat* ⇒ *'a*::{*t2-space*,*topological-comm-monoid-add*}
begin

lemma *sums-Suc*:
assumes (λ*n*. *f* (*Suc n*)) *sums l*
shows *f sums* (*l* + *f 0*)
proof −
have (λ*n*. (∑ *i*<*n*. *f* (*Suc i*)) + *f 0*) ⟶ *l* + *f 0*
using *assms* **by** (*auto intro*!: *tendsto-add simp*: *sums-def*)
moreover have (∑ *i*<*n*. *f* (*Suc i*)) + *f 0* = (∑ *i*<*Suc n*. *f i*) **for** *n*
unfolding *lessThan-Suc-eq-insert-0*
by (*simp add*: *ac-simps sum.atLeast1-atMost-eq image-Suc-lessThan*)
ultimately show *?thesis*
by (*auto simp*: *sums-def simp del*: *sum.lessThan-Suc intro*: *filterlim-sequentially-Suc*[*THEN*

iffD1])
qed

lemma *sums-add*: *f sums a* $\Longrightarrow$ *g sums b* $\Longrightarrow$ $(\lambda n.\ f\ n + g\ n)$ *sums* $(a + b)$
unfolding *sums-def* **by** (*simp add*: *sum.distrib tendsto-add*)

lemma *summable-add*: *summable f* $\Longrightarrow$ *summable g* $\Longrightarrow$ *summable* $(\lambda n.\ f\ n + g\ n)$
unfolding *summable-def* **by** (*auto intro*: *sums-add*)

lemma *suminf-add*: *summable f* $\Longrightarrow$ *summable g* $\Longrightarrow$ *suminf f* + *suminf g* = $(\sum n.\ f\ n + g\ n)$
by (*intro sums-unique sums-add summable-sums*)

end

context
fixes *f* :: *'i* $\Rightarrow$ *nat* $\Rightarrow$ *'a*::{*t2-space,topological-comm-monoid-add*}
and *I* :: *'i set*
begin

lemma *sums-sum*: $(\bigwedge i.\ i \in I \Longrightarrow (f\ i)\ sums\ (x\ i)) \Longrightarrow (\lambda n.\ \sum i{\in}I.\ f\ i\ n)\ sums\ (\sum i{\in}I.\ x\ i)$
by (*induct I rule*: *infinite-finite-induct*) (*auto intro*!: *sums-add*)

lemma *suminf-sum*: $(\bigwedge i.\ i \in I \Longrightarrow summable\ (f\ i)) \Longrightarrow (\sum n.\ \sum i{\in}I.\ f\ i\ n) = (\sum i{\in}I.\ \sum n.\ f\ i\ n)$
using *sums-unique*[*OF sums-sum*, *OF summable-sums*] **by** *simp*

lemma *summable-sum*: $(\bigwedge i.\ i \in I \Longrightarrow summable\ (f\ i)) \Longrightarrow summable\ (\lambda n.\ \sum i{\in}I.\ f\ i\ n)$
using *sums-summable*[*OF sums-sum*[*OF summable-sums*]] **.**

end

lemma *sums-If-finite-set'*:
fixes *f g* :: *nat* $\Rightarrow$ *'a*::{*t2-space,topological-ab-group-add*}
assumes *g sums S* **and** *finite A* **and** $S' = S + (\sum n{\in}A.\ f\ n - g\ n)$
shows $(\lambda n.\ if\ n \in A\ then\ f\ n\ else\ g\ n)\ sums\ S'$
proof −
have $(\lambda n.\ g\ n + (if\ n \in A\ then\ f\ n - g\ n\ else\ 0))\ sums\ (S + (\sum n{\in}A.\ f\ n - g\ n))$
by (*intro sums-add assms sums-If-finite-set*)
also have $(\lambda n.\ g\ n + (if\ n \in A\ then\ f\ n - g\ n\ else\ 0)) = (\lambda n.\ if\ n \in A\ then\ f\ n\ else\ g\ n)$
by (*simp add*: *fun-eq-iff*)
finally show *?thesis* **using** *assms* **by** *simp*
qed

109.5 Infinite summability on real normed vector spaces

context
fixes $f :: nat \Rightarrow 'a::real\text{-}normed\text{-}vector$
begin

lemma *sums-Suc-iff*: $(\lambda n.\ f\ (Suc\ n))\ sums\ s \longleftrightarrow f\ sums\ (s + f\ 0)$
proof −
have $f\ sums\ (s + f\ 0) \longleftrightarrow (\lambda i.\ \sum j{<}Suc\ i.\ f\ j) \longrightarrow s + f\ 0$
by (*subst filterlim-sequentially-Suc*) (*simp add*: *sums-def*)
also have $\ldots \longleftrightarrow (\lambda i.\ (\sum j{<}i.\ f\ (Suc\ j)) + f\ 0) \longrightarrow s + f\ 0$
by (*simp add*: *ac-simps lessThan-Suc-eq-insert-0 image-Suc-lessThan sum.atLeast1-atMost-eq*)
also have $\ldots \longleftrightarrow (\lambda n.\ f\ (Suc\ n))\ sums\ s$
proof
assume $(\lambda i.\ (\sum j{<}i.\ f\ (Suc\ j)) + f\ 0) \longrightarrow s + f\ 0$
with *tendsto-add*[*OF this tendsto-const*, *of* $-f\ 0$] **show** $(\lambda i.\ f\ (Suc\ i))\ sums\ s$
by (*simp add*: *sums-def*)
qed (*auto intro*: *tendsto-add simp*: *sums-def*)
finally show *?thesis* **..**
qed

lemma *summable-Suc-iff*: $summable\ (\lambda n.\ f\ (Suc\ n)) = summable\ f$
proof
assume *summable f*
then have *f sums suminf f*
by (*rule summable-sums*)
then have $(\lambda n.\ f\ (Suc\ n))\ sums\ (suminf\ f - f\ 0)$
by (*simp add*: *sums-Suc-iff*)
then show $summable\ (\lambda n.\ f\ (Suc\ n))$
unfolding *summable-def* **by** *blast*
qed (*auto simp*: *sums-Suc-iff summable-def*)

lemma *sums-Suc-imp*: $f\ 0 = 0 \Longrightarrow (\lambda n.\ f\ (Suc\ n))\ sums\ s \Longrightarrow (\lambda n.\ f\ n)\ sums\ s$
using *sums-Suc-iff* **by** *simp*

end

context
fixes $f :: nat \Rightarrow 'a::real\text{-}normed\text{-}vector$
begin

lemma *sums-diff*: $f\ sums\ a \Longrightarrow g\ sums\ b \Longrightarrow (\lambda n.\ f\ n - g\ n)\ sums\ (a - b)$
unfolding *sums-def* **by** (*simp add*: *sum-subtractf tendsto-diff*)

lemma *summable-diff*: $summable\ f \Longrightarrow summable\ g \Longrightarrow summable\ (\lambda n.\ f\ n - g\ n)$
unfolding *summable-def* **by** (*auto intro*: *sums-diff*)

lemma *suminf-diff*: $summable\ f \Longrightarrow summable\ g \Longrightarrow suminf\ f - suminf\ g = (\sum n.\ f\ n - g\ n)$

by (*intro sums-unique sums-diff summable-sums*)

lemma *sums-minus*: *f sums a* $\Longrightarrow$ $(\lambda n.\ -\ f\ n)$ *sums* $(-\ a)$
unfolding *sums-def* **by** (*simp add*: *sum-negf tendsto-minus*)

lemma *summable-minus*: *summable f* $\Longrightarrow$ *summable* $(\lambda n.\ -\ f\ n)$
unfolding *summable-def* **by** (*auto intro*: *sums-minus*)

lemma *suminf-minus*: *summable f* $\Longrightarrow$ $(\sum n.\ -\ f\ n) = -\ (\sum n.\ f\ n)$
by (*intro sums-unique* [*symmetric*] *sums-minus summable-sums*)

lemma *sums-iff-shift*: $(\lambda i.\ f\ (i\ +\ n))$ *sums s* $\longleftrightarrow$ *f sums* $(s\ +\ (\sum i{<}n.\ f\ i))$
proof (*induct n arbitrary*: *s*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have $(\lambda i.\ f\ (Suc\ i\ +\ n))$ *sums s* $\longleftrightarrow$ $(\lambda i.\ f\ (i\ +\ n))$ *sums* $(s\ +\ f\ n)$
by (*subst sums-Suc-iff*) *simp*
with *Suc* **show** *?case*
by (*simp add*: *ac-simps*)
qed

corollary *sums-iff-shift'*: $(\lambda i.\ f\ (i\ +\ n))$ *sums* $(s\ -\ (\sum i{<}n.\ f\ i))$ $\longleftrightarrow$ *f sums s*
by (*simp add*: *sums-iff-shift*)

lemma *sums-zero-iff-shift*:
assumes $\bigwedge i.\ i < n \Longrightarrow f\ i = 0$
shows $(\lambda i.\ f\ (i{+}n))$ *sums s* $\longleftrightarrow$ $(\lambda i.\ f\ i)$ *sums s*
by (*simp add*: *assms sums-iff-shift*)

lemma *summable-iff-shift* [*simp*]: *summable* $(\lambda n.\ f\ (n\ +\ k))$ $\longleftrightarrow$ *summable f*
by (*metis diff-add-cancel summable-def sums-iff-shift* [*abs-def*])

lemma *sums-split-initial-segment*: *f sums s* $\Longrightarrow$ $(\lambda i.\ f\ (i\ +\ n))$ *sums* $(s\ -\ (\sum i{<}n.$ *f i*))
by (*simp add*: *sums-iff-shift*)

lemma *summable-ignore-initial-segment*: *summable f* $\Longrightarrow$ *summable* $(\lambda n.\ f(n\ +$ *k*))
by (*simp add*: *summable-iff-shift*)

lemma *suminf-minus-initial-segment*: *summable f* $\Longrightarrow$ $(\sum n.\ f\ (n\ +\ k)) = (\sum n.$ *f n*) $-$ $(\sum i{<}k.\ f\ i)$
by (*rule sums-unique*[*symmetric*]) (*auto simp*: *sums-iff-shift*)

lemma *suminf-split-initial-segment*: *summable f* $\Longrightarrow$ *suminf f* $= (\sum n.\ f(n\ +\ k))$ $+$ $(\sum i{<}k.\ f\ i)$
by (*auto simp add*: *suminf-minus-initial-segment*)

lemma *suminf-split-head*: *summable f* $\Longrightarrow$ $(\sum n.\ f\ (Suc\ n)) = suminf\ f - f\ 0$
using *suminf-split-initial-segment*[*of 1*] **by** *simp*

lemma *suminf-exist-split*:
fixes r :: *real*
assumes $0 < r$ **and** *summable f*
shows $\exists N.\ \forall n{\geq}N.\ norm\ (\sum i.\ f\ (i + n)) < r$
proof −
from *LIMSEQ-D*[*OF summable-LIMSEQ*[*OF* ‹*summable f*›] ‹$0 < r$›]
obtain N :: *nat* **where** $\forall\ n \geq N.\ norm\ (sum\ f\ \{..{<}n\} - suminf\ f) < r$
by *auto*
then show *?thesis*
by (*auto simp*: *norm-minus-commute suminf-minus-initial-segment*[*OF* ‹*summable f*›])
qed

lemma *summable-LIMSEQ-zero*:
assumes *summable f* **shows** $f \longrightarrow 0$
proof −
have *Cauchy* $(\lambda n.\ sum\ f\ \{..{<}n\})$
using *LIMSEQ-imp-Cauchy assms summable-LIMSEQ* **by** *blast*
then show *?thesis*
unfolding *Cauchy-iff LIMSEQ-iff*
by (*metis add.commute add-diff-cancel-right′ diff-zero le-SucI sum.lessThan-Suc*)
qed

lemma *summable-imp-convergent*: *summable f* $\Longrightarrow$ *convergent f*
by (*force dest*!: *summable-LIMSEQ-zero simp*: *convergent-def*)

lemma *summable-imp-Bseq*: *summable f* $\Longrightarrow$ *Bseq f*
by (*simp add*: *convergent-imp-Bseq summable-imp-convergent*)

end

lemma *summable-minus-iff*: *summable* $(\lambda n.\ -\ f\ n) \longleftrightarrow$ *summable f*
for f :: *nat* $\Rightarrow$ *'a*::*real-normed-vector*
by (*auto dest*: *summable-minus*)

lemma (**in** *bounded-linear*) *sums*: $(\lambda n.\ X\ n)$ *sums a* $\Longrightarrow$ $(\lambda n.\ f\ (X\ n))$ *sums* $(f\ a)$
unfolding *sums-def* **by** (*drule tendsto*) (*simp only*: *sum*)

lemma (**in** *bounded-linear*) *summable*: *summable* $(\lambda n.\ X\ n)$ $\Longrightarrow$ *summable* $(\lambda n.\ f\ (X\ n))$
unfolding *summable-def* **by** (*auto intro*: *sums*)

lemma (**in** *bounded-linear*) *suminf*: *summable* $(\lambda n.\ X\ n)$ $\Longrightarrow$ $f\ (\sum n.\ X\ n) = (\sum n.\ f\ (X\ n))$
by (*intro sums-unique sums summable-sums*)

lemmas *sums-of-real* = *bounded-linear.sums* [*OF bounded-linear-of-real*]
lemmas *summable-of-real* = *bounded-linear.summable* [*OF bounded-linear-of-real*]
lemmas *suminf-of-real* = *bounded-linear.suminf* [*OF bounded-linear-of-real*]

lemmas *sums-scaleR-left* = *bounded-linear.sums*[*OF bounded-linear-scaleR-left*]
lemmas *summable-scaleR-left* = *bounded-linear.summable*[*OF bounded-linear-scaleR-left*]
lemmas *suminf-scaleR-left* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-left*]

lemmas *sums-scaleR-right* = *bounded-linear.sums*[*OF bounded-linear-scaleR-right*]
lemmas *summable-scaleR-right* = *bounded-linear.summable*[*OF bounded-linear-scaleR-right*]
lemmas *suminf-scaleR-right* = *bounded-linear.suminf*[*OF bounded-linear-scaleR-right*]

lemma *summable-const-iff*: *summable* (*λ-. c*) *⟷ c = 0*
  for *c* :: *'a*::*real-normed-vector*
proof −
  have ¬ *summable* (*λ-. c*) **if** *c ≠ 0*
  proof −
    from *that* **have** *filterlim* (*λn. of-nat n ∗ norm c*) *at-top sequentially*
      by (*subst mult.commute*)
        (*auto intro*!: *filterlim-tendsto-pos-mult-at-top filterlim-real-sequentially*)
    then have ¬ *convergent* (*λn. norm* (*∑ k<n. c*))
    by (*intro filterlim-at-infinity-imp-not-convergent filterlim-at-top-imp-at-infinity*)
        (*simp-all add*: *sum-constant-scaleR*)
    then show *?thesis*
      unfolding *summable-iff-convergent* **using** *convergent-norm* **by** *blast*
  qed
  then show *?thesis* **by** *auto*
qed

109.6 Infinite summability on real normed algebras

context
  fixes *f* :: *nat ⇒ 'a*::*real-normed-algebra*
begin

lemma *sums-mult*: *f sums a ⟹* (*λn. c ∗ f n*) *sums* (*c ∗ a*)
  by (*rule bounded-linear.sums* [*OF bounded-linear-mult-right*])

lemma *summable-mult*: *summable f ⟹ summable* (*λn. c ∗ f n*)
  by (*rule bounded-linear.summable* [*OF bounded-linear-mult-right*])

lemma *suminf-mult*: *summable f ⟹ suminf* (*λn. c ∗ f n*) = *c ∗ suminf f*
  by (*rule bounded-linear.suminf* [*OF bounded-linear-mult-right*, *symmetric*])

lemma *sums-mult2*: *f sums a ⟹* (*λn. f n ∗ c*) *sums* (*a ∗ c*)
  by (*rule bounded-linear.sums* [*OF bounded-linear-mult-left*])

lemma *summable-mult2*: *summable f ⟹ summable* (*λn. f n ∗ c*)

by (*rule bounded-linear.summable* [*OF bounded-linear-mult-left*])

lemma *suminf-mult2*: *summable f* $\Longrightarrow$ *suminf f* $*$ *c* $=$ $(\sum n.\ f\ n * c)$
by (*rule bounded-linear.suminf* [*OF bounded-linear-mult-left*])

end

lemma *sums-mult-iff*:
fixes *f* :: *nat* $\Rightarrow$ *'a*::{*real-normed-algebra,field*}
assumes *c* $\neq$ *0*
shows $(\lambda n.\ c * f\ n)$ *sums* $(c * d) \longleftrightarrow$ *f sums d*
using *sums-mult*[*of f d c*] *sums-mult*[*of* $\lambda n.\ c * f\ n\ c * d$ *inverse c*]
by (*force simp*: *field-simps assms*)

lemma *sums-mult2-iff*:
fixes *f* :: *nat* $\Rightarrow$ *'a*::{*real-normed-algebra,field*}
assumes *c* $\neq$ *0*
shows $(\lambda n.\ f\ n * c)$ *sums* $(d * c) \longleftrightarrow$ *f sums d*
using *sums-mult-iff*[*OF assms*, *of f d*] **by** (*simp add*: *mult.commute*)

lemma *sums-of-real-iff*:
$(\lambda n.$ *of-real* $(f\ n)$ :: *'a*::*real-normed-div-algebra*$)$ *sums of-real c* $\longleftrightarrow$ *f sums c*
by (*simp add*: *sums-def of-real-sum*[*symmetric*] *tendsto-of-real-iff del*: *of-real-sum*)

109.7 Infinite summability on real normed fields

context
fixes *c* :: *'a*::*real-normed-field*
begin

lemma *sums-divide*: *f sums a* $\Longrightarrow$ $(\lambda n.\ f\ n\ /\ c)$ *sums* $(a\ /\ c)$
by (*rule bounded-linear.sums* [*OF bounded-linear-divide*])

lemma *summable-divide*: *summable f* $\Longrightarrow$ *summable* $(\lambda n.\ f\ n\ /\ c)$
by (*rule bounded-linear.summable* [*OF bounded-linear-divide*])

lemma *suminf-divide*: *summable f* $\Longrightarrow$ *suminf* $(\lambda n.\ f\ n\ /\ c)$ $=$ *suminf f* $/$ *c*
by (*rule bounded-linear.suminf* [*OF bounded-linear-divide*, *symmetric*])

lemma *summable-inverse-divide*: *summable* $(inverse \circ f) \Longrightarrow$ *summable* $(\lambda n.\ c\ /$ *f n*$)$
by (*auto dest*: *summable-mult* [*of - c*] *simp*: *field-simps*)

lemma *sums-mult-D*: $(\lambda n.\ c * f\ n)$ *sums a* $\Longrightarrow$ *c* $\neq$ *0* $\Longrightarrow$ *f sums* (a/c)
using *sums-mult-iff* **by** *fastforce*

lemma *summable-mult-D*: *summable* $(\lambda n.\ c * f\ n) \Longrightarrow$ *c* $\neq$ *0* $\Longrightarrow$ *summable f*
by (*auto dest*: *summable-divide*)

Sum of a geometric progression.

lemma *geometric-sums*:
assumes *norm c < 1*
shows *(λn. c^n) sums (1 / (1 − c))*
proof −
have *neq-0*: *c − 1 ≠ 0*
using *assms* **by** *auto*
then have *(λn. c ^ n / (c − 1) − 1 / (c − 1)) ⟶ 0 / (c − 1) − 1 / (c − 1)*
by (*intro tendsto-intros assms*)
then have *(λn. (c ^ n − 1) / (c − 1)) ⟶ 1 / (1 − c)*
by (*simp add*: *nonzero-minus-divide-right* [*OF neq-0*] *diff-divide-distrib*)
with *neq-0* **show** *(λn. c ^ n) sums (1 / (1 − c))*
by (*simp add*: *sums-def geometric-sum*)
qed

lemma *summable-geometric*: *norm c < 1 ⟹ summable (λn. c^n)*
by (*rule geometric-sums* [*THEN sums-summable*])

lemma *suminf-geometric*: *norm c < 1 ⟹ suminf (λn. c^n) = 1 / (1 − c)*
by (*rule sums-unique*[*symmetric*]) (*rule geometric-sums*)

lemma *summable-geometric-iff* [*simp*]: *summable (λn. c ^ n) ⟷ norm c < 1*
proof
assume *summable (λn. c ^ n :: 'a :: real-normed-field)*
then have *(λn. norm c ^ n) ⟶ 0*
by (*simp add*: *norm-power* [*symmetric*] *tendsto-norm-zero-iff summable-LIMSEQ-zero*)
from *order-tendstoD*(*2*)[*OF this zero-less-one*] **obtain** *n* **where** *norm c ^ n < 1*
by (*auto simp*: *eventually-at-top-linorder*)
then show *norm c < 1* **using** *one-le-power*[*of norm c n*]
by (*cases norm c ≥ 1*) (*linarith*, *simp*)
qed (*rule summable-geometric*)

end

Biconditional versions for constant factors

context
fixes *c* :: *'a*::*real-normed-field*
begin

lemma *summable-cmult-iff* [*simp*]: *summable (λn. c * f n) ⟷ c=0 ∨ summable f*
proof −
have *⟦summable (λn. c * f n); c ≠ 0⟧ ⟹ summable f*
using *summable-mult-D* **by** *blast*
then show *?thesis*
by (*auto simp*: *summable-mult*)
qed

lemma *summable-divide-iff* [*simp*]: *summable (λn. f n / c) ⟷ c=0 ∨ summable*

f
proof −
have ⟦*summable* (λ*n*. *f n* / *c*); *c* ≠ *0*⟧ ⟹ *summable f*
by (*auto dest*: *summable-divide* [**where** *c* = *1*/*c*])
then show *?thesis*
by (*auto simp*: *summable-divide*)
qed

end

lemma *power-half-series*: (λ*n*. (*1*/*2*::*real*)^*Suc n*) *sums 1*
proof −
have *2*: (λ*n*. (*1*/*2*::*real*)^*n*) *sums 2*
using *geometric-sums* [*of 1*/*2*::*real*] **by** *auto*
have (λ*n*. (*1*/*2*::*real*)^*Suc n*) = (λ*n*. (*1* / *2*) ^ *n* / *2*)
by (*simp add*: *mult.commute*)
then show *?thesis*
using *sums-divide* [*OF 2*, *of 2*] **by** *simp*
qed

109.8 Telescoping

lemma *telescope-sums*:
fixes *c* :: *'a*::*real-normed-vector*
assumes *f* ⟶ *c*
shows (λ*n*. *f* (*Suc n*) − *f n*) *sums* (*c* − *f 0*)
unfolding *sums-def*
proof (*subst filterlim-sequentially-Suc* [*symmetric*])
have (λ*n*. ∑*k*<*Suc n*. *f* (*Suc k*) − *f k*) = (λ*n*. *f* (*Suc n*) − *f 0*)
by (*simp add*: *lessThan-Suc-atMost atLeast0AtMost* [*symmetric*] *sum-Suc-diff*)
also have ... ⟶ *c* − *f 0*
by (*intro tendsto-diff LIMSEQ-Suc*[*OF assms*] *tendsto-const*)
finally show (λ*n*. ∑*n*<*Suc n*. *f* (*Suc n*) − *f n*) ⟶ *c* − *f 0* **.**
qed

lemma *telescope-sums'*:
fixes *c* :: *'a*::*real-normed-vector*
assumes *f* ⟶ *c*
shows (λ*n*. *f n* − *f* (*Suc n*)) *sums* (*f 0* − *c*)
using *sums-minus*[*OF telescope-sums*[*OF assms*]] **by** (*simp add*: *algebra-simps*)

lemma *telescope-summable*:
fixes *c* :: *'a*::*real-normed-vector*
assumes *f* ⟶ *c*
shows *summable* (λ*n*. *f* (*Suc n*) − *f n*)
using *telescope-sums*[*OF assms*] **by** (*simp add*: *sums-iff*)

lemma *telescope-summable'*:
fixes *c* :: *'a*::*real-normed-vector*

assumes $f \longrightarrow c$
shows *summable* $(\lambda n.\ f\ n - f\ (Suc\ n))$
using *summable-minus*[*OF telescope-summable*[*OF assms*]] **by** (*simp add*: *algebra-simps*)

109.9 Infinite summability on Banach spaces

Cauchy-type criterion for convergence of series (c.f. Harrison).

lemma *summable-Cauchy*: *summable f* $\longleftrightarrow$ $(\forall\, e{>}0.\ \exists\, N.\ \forall\, m{\geq}N.\ \forall\, n.\ norm\ (sum\ f\ \{m..{<}n\}) < e)$ (**is** - = *?rhs*)
for $f :: nat \Rightarrow {'a}{::}banach$
proof
assume *f*: *summable f*
show *?rhs*
proof *clarify*
fix $e :: real$
assume $0 < e$
then obtain M **where** M: $\bigwedge m\ n.\ [\![m{\geq}M;\ n{\geq}M]\!] \Longrightarrow norm\ (sum\ f\ \{..{<}m\} - sum\ f\ \{..{<}n\}) < e$
using *f* **by** (*force simp add*: *summable-iff-convergent Cauchy-convergent-iff* [*symmetric*] *Cauchy-iff*)
have *norm* $(sum\ f\ \{m..{<}n\}) < e$ **if** $m \geq M$ **for** $m\ n$
proof (*cases m n rule*: *linorder-class.le-cases*)
assume $m \leq n$
then show *?thesis*
by (*metis* (*mono-tags*, *opaque-lifting*) *M atLeast0LessThan order-trans sum-diff-nat-ivl that zero-le*)
next
assume $n \leq m$
then show *?thesis*
by (*simp add*: ‹$0 < e$›)
qed
then show $\exists\, N.\ \forall\, m{\geq}N.\ \forall\, n.\ norm\ (sum\ f\ \{m..{<}n\}) < e$
by *blast*
qed
next
assume *r*: *?rhs*
then show *summable f*
unfolding *summable-iff-convergent Cauchy-convergent-iff* [*symmetric*] *Cauchy-iff*
proof *clarify*
fix $e :: real$
assume $0 < e$
with *r* **obtain** N **where** N: $\bigwedge m\ n.\ m \geq N \Longrightarrow norm\ (sum\ f\ \{m..{<}n\}) < e$
by *blast*
have *norm* $(sum\ f\ \{..{<}m\} - sum\ f\ \{..{<}n\}) < e$ **if** $m{\geq}N\ n{\geq}N$ **for** $m\ n$
proof (*cases m n rule*: *linorder-class.le-cases*)
assume $m \leq n$
then show *?thesis*
by (*metis N finite-lessThan lessThan-minus-lessThan lessThan-subset-iff*

norm-minus-commute sum-diff ‹m≥N›)
next
assume $n \leq m$
then show *?thesis*
by (*metis N finite-lessThan lessThan-minus-lessThan lessThan-subset-iff sum-diff ‹n≥N›*)
qed
then show $\exists M.\ \forall m{\geq}M.\ \forall n{\geq}M.\ norm\ (sum\ f\ \{..{<}m\} - sum\ f\ \{..{<}n\}) < e$
by *blast*
qed
qed

lemma *summable-Cauchy'*:
fixes $f :: nat \Rightarrow 'a :: banach$
assumes *ev*: *eventually* $(\lambda m.\ \forall n{\geq}m.\ norm\ (sum\ f\ \{m..{<}n\}) \leq g\ m)$ *sequentially*
assumes *g0*: $g \longrightarrow 0$
shows *summable f*
proof (*subst summable-Cauchy*, *intro allI impI*, *goal-cases*)
case (*1 e*)
then have $\forall_F\ x$ *in sequentially*. $g\ x < e$
using *g0 order-tendstoD*(*2*) **by** *blast*
with *ev* **have** *eventually* $(\lambda m.\ \forall n.\ norm\ (sum\ f\ \{m..{<}n\}) < e)$ *at-top*
proof *eventually-elim*
case (*elim m*)
show *?case*
proof
fix *n*
from *elim* **show** $norm\ (sum\ f\ \{m..{<}n\}) < e$
by (*cases* $n \geq m$) *auto*
qed
qed
thus *?case* **by** (*auto simp*: *eventually-at-top-linorder*)
qed

context
fixes $f :: nat \Rightarrow 'a{::}banach$
begin

Absolute convergence imples normal convergence.

lemma *summable-norm-cancel*: *summable* $(\lambda n.\ norm\ (f\ n)) \Longrightarrow$ *summable f*
unfolding *summable-Cauchy*
apply (*erule all-forward imp-forward ex-forward* | *assumption*)+
apply (*fastforce simp add*: *order-le-less-trans* [*OF norm-sum*] *order-le-less-trans* [*OF abs-ge-self*])
done

lemma *summable-norm*: *summable* $(\lambda n.\ norm\ (f\ n)) \Longrightarrow norm\ (suminf\ f) \leq (\sum n.\ norm\ (f\ n))$
by (*auto intro*: *LIMSEQ-le tendsto-norm summable-norm-cancel summable-LIMSEQ*

norm-sum)

Comparison tests.

lemma *summable-comparison-test*:
assumes *fg*: $\exists N.\ \forall n{\geq}N.\ norm\ (f\ n) \leq g\ n$ **and** *g*: *summable g*
shows *summable f*
proof −
obtain *N* **where** *N*: $\bigwedge n.\ n{\geq}N \Longrightarrow norm\ (f\ n) \leq g\ n$
using *assms* **by** *blast*
show *?thesis*
proof (*clarsimp simp add*: *summable-Cauchy*)
fix *e* :: *real*
assume $0 < e$
then obtain *Ng* **where** *Ng*: $\bigwedge m\ n.\ m \geq Ng \Longrightarrow norm\ (sum\ g\ \{m..{<}n\}) < e$
using *g* **by** (*fastforce simp*: *summable-Cauchy*)
with *N* **have** $norm\ (sum\ f\ \{m..{<}n\}) < e$ **if** $m{\geq}max\ N\ Ng$ **for** *m n*
proof −
have $norm\ (sum\ f\ \{m..{<}n\}) \leq sum\ g\ \{m..{<}n\}$
using *N that* **by** (*force intro*: *sum-norm-le*)
also have $... \leq norm\ (sum\ g\ \{m..{<}n\})$
by *simp*
also have $... < e$
using *Ng that* **by** *auto*
finally show *?thesis* **.**
qed
then show $\exists N.\ \forall m{\geq}N.\ \forall n.\ norm\ (sum\ f\ \{m..{<}n\}) < e$
by *blast*
qed
qed

lemma *summable-comparison-test-ev*:
eventually $(\lambda n.\ norm\ (f\ n) \leq g\ n)$ *sequentially* $\Longrightarrow$ *summable g* $\Longrightarrow$ *summable f*
by (*rule summable-comparison-test*) (*auto simp*: *eventually-at-top-linorder*)

A better argument order.

lemma *summable-comparison-test′*: *summable g* $\Longrightarrow$ $(\bigwedge n.\ n \geq N \Longrightarrow norm\ (f\ n) \leq g\ n) \Longrightarrow$ *summable f*
by (*rule summable-comparison-test*) *auto*

109.10 The Ratio Test

lemma *summable-ratio-test*:
assumes $c < 1$ $\bigwedge n.\ n \geq N \Longrightarrow norm\ (f\ (Suc\ n)) \leq c * norm\ (f\ n)$
shows *summable f*
proof (*cases* $0 < c$)
case *True*
show *summable f*
proof (*rule summable-comparison-test*)
show $\exists N'.\ \forall n{\geq}N'.\ norm\ (f\ n) \leq (norm\ (f\ N)\ /\ (c\ \hat{}\ N)) * c\ \hat{}\ n$

```
    proof (intro exI allI impI)
      fix n
      assume N ≤ n
      then show norm (f n) ≤ (norm (f N) / (c ^ N)) * c ^ n
      proof (induct rule: inc-induct)
        case base
        with True show ?case by simp
      next
        case (step m)
        have norm (f (Suc m)) / c ^ Suc m * c ^ n ≤ norm (f m) / c ^ m * c ^ n
          using ‹0 < c› ‹c < 1› assms(2)[OF ‹N ≤ m›] by (simp add: field-simps)
        with step show ?case by simp
      qed
    qed
    show summable (λn. norm (f N) / c ^ N * c ^ n)
      using ‹0 < c› ‹c < 1› by (intro summable-mult summable-geometric) simp
  qed
next
  case False
  have f (Suc n) = 0 if n ≥ N for n
  proof −
    from that have norm (f (Suc n)) ≤ c * norm (f n)
      by (rule assms(2))
    also have ... ≤ 0
      using False by (simp add: not-less mult-nonpos-nonneg)
    finally show ?thesis
      by auto
  qed
  then show summable f
    by (intro sums-summable[OF sums-finite, of {.. Suc N}]) (auto simp: not-le
Suc-less-eq2)
qed

end
```

Application to convergence of the log function

```
lemma norm-summable-ln-series:
  fixes z :: 'a :: {real-normed-field, banach}
  assumes norm z < 1
  shows   summable (λn. norm (z ^ n / of-nat n))
proof (rule summable-comparison-test)
  show summable (λn. norm (z ^ n))
    using assms unfolding norm-power by (intro summable-geometric) auto
  have norm z ^ n / real n ≤ norm z ^ n for n
  proof (cases n = 0)
    case False
    hence norm z ^ n * 1 ≤ norm z ^ n * real n
      by (intro mult-left-mono) auto
    thus ?thesis
```

using *False* **by** (*simp add*: *field-simps*)
qed *auto*
thus $\exists N.\ \forall n{\geq}N.\ norm\ (norm\ (z\ \hat{}\ n\ /\ of\text{-}nat\ n)) \leq norm\ (z\ \hat{}\ n)$
by (*intro exI*[*of* - *0*]) (*auto simp*: *norm-power norm-divide*)
qed

Relations among convergence and absolute convergence for power series.

lemma *Abel-lemma*:
fixes *a* :: *nat* ⇒ *'a*::*real-normed-vector*
assumes *r*: $0 \leq r$
and *r0*: $r < r0$
and *M*: $\bigwedge n.\ norm\ (a\ n) * r0\hat{}n \leq M$
shows *summable* (λ*n*. *norm* (*a n*) ∗ *r*^*n*)
proof (*rule summable-comparison-test'*)
show *summable* (λ*n*. *M* ∗ (*r* / *r0*) ^ *n*)
using *assms* **by** (*auto simp add*: *summable-mult summable-geometric*)
show *norm* (*norm* (*a n*) ∗ *r* ^ *n*) ≤ *M* ∗ (*r* / *r0*) ^ *n* **for** *n*
using *r r0 M* [*of n*] *dual-order.order-iff-strict*
by (*fastforce simp add*: *abs-mult field-simps*)
qed

Summability of geometric series for real algebras.

lemma *complete-algebra-summable-geometric*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*banach*}
assumes *norm x* < *1*
shows *summable* (λ*n*. *x* ^ *n*)
proof (*rule summable-comparison-test*)
show $\exists N.\ \forall n{\geq}N.\ norm\ (x\ \hat{}\ n) \leq norm\ x\ \hat{}\ n$
by (*simp add*: *norm-power-ineq*)
from *assms* **show** *summable* (λ*n*. *norm x* ^ *n*)
by (*simp add*: *summable-geometric*)
qed

109.11 Cauchy Product Formula

Proof based on Analysis WebNotes: Chapter 07, Class 41 http://www.math.unl.edu/~webnotes/classes/class41/prp77.htm

lemma *Cauchy-product-sums*:
fixes *a b* :: *nat* ⇒ *'a*::{*real-normed-algebra*,*banach*}
assumes *a*: *summable* (λ*k*. *norm* (*a k*))
and *b*: *summable* (λ*k*. *norm* (*b k*))
shows (λ*k*. $\sum i{\leq}k.\ a\ i * b\ (k - i)$) *sums* (($\sum k.\ a\ k$) ∗ ($\sum k.\ b\ k$))
proof −
let *?S1* = λ*n*::*nat*. {..<*n*} × {..<*n*}
let *?S2* = λ*n*::*nat*. {(*i*,*j*). *i* + *j* < *n*}
have *S1-mono*: $\bigwedge m\ n.\ m \leq n \Longrightarrow$ *?S1 m* ⊆ *?S1 n* **by** *auto*
have *S2-le-S1*: $\bigwedge n.$ *?S2 n* ⊆ *?S1 n* **by** *auto*
have *S1-le-S2*: $\bigwedge n.$ *?S1* (*n div 2*) ⊆ *?S2 n* **by** *auto*

have *finite-S1*: $\bigwedge n.$ *finite* (*?S1 n*) **by** *simp*
with *S2-le-S1* **have** *finite-S2*: $\bigwedge n.$ *finite* (*?S2 n*) **by** (*rule finite-subset*)

let *?g* = $\lambda(i,j).$ *a i* $*$ *b j*
let *?f* = $\lambda(i,j).$ *norm* (*a i*) $*$ *norm* (*b j*)
have *f-nonneg*: $\bigwedge x.$ *0* $\leq$ *?f x* **by** *auto*
then have *norm-sum-f*: $\bigwedge A.$ *norm* (*sum ?f A*) = *sum ?f A*
unfolding *real-norm-def*
by (*simp only*: *abs-of-nonneg sum-nonneg* [*rule-format*])

have ($\lambda n.$ ($\sum k{<}n.$ *a k*) $*$ ($\sum k{<}n.$ *b k*)) $\longrightarrow$ ($\sum k.$ *a k*) $*$ ($\sum k.$ *b k*)
by (*intro tendsto-mult summable-LIMSEQ summable-norm-cancel* [*OF a*] *summable-norm-cancel* [*OF b*])
then have *1*: ($\lambda n.$ *sum ?g* (*?S1 n*)) $\longrightarrow$ ($\sum k.$ *a k*) $*$ ($\sum k.$ *b k*)
by (*simp only*: *sum-product sum.Sigma* [*rule-format*] *finite-lessThan*)

have ($\lambda n.$ ($\sum k{<}n.$ *norm* (*a k*)) $*$ ($\sum k{<}n.$ *norm* (*b k*))) $\longrightarrow$ ($\sum k.$ *norm* (*a k*)) $*$ ($\sum k.$ *norm* (*b k*))
using *a b* **by** (*intro tendsto-mult summable-LIMSEQ*)
then have ($\lambda n.$ *sum ?f* (*?S1 n*)) $\longrightarrow$ ($\sum k.$ *norm* (*a k*)) $*$ ($\sum k.$ *norm* (*b k*))
by (*simp only*: *sum-product sum.Sigma* [*rule-format*] *finite-lessThan*)
then have *convergent* ($\lambda n.$ *sum ?f* (*?S1 n*))
by (*rule convergentI*)
then have *Cauchy*: *Cauchy* ($\lambda n.$ *sum ?f* (*?S1 n*))
by (*rule convergent-Cauchy*)
have *Zfun* ($\lambda n.$ *sum ?f* (*?S1 n* $-$ *?S2 n*)) *sequentially*
proof (*rule ZfunI*, *simp only*: *eventually-sequentially norm-sum-f*)
fix *r* :: *real*
assume *r*: *0* $<$ *r*
from *CauchyD* [*OF Cauchy r*] **obtain** *N*
where $\forall\, m{\geq}N.\ \forall\, n{\geq}N.$ *norm* (*sum ?f* (*?S1 m*) $-$ *sum ?f* (*?S1 n*)) $<$ *r* **..**
then have $\bigwedge m\ n.$ *N* $\leq$ *n* $\Longrightarrow$ *n* $\leq$ *m* $\Longrightarrow$ *norm* (*sum ?f* (*?S1 m* $-$ *?S1 n*)) $<$ *r*
by (*simp only*: *sum-diff finite-S1 S1-mono*)
then have *N*: $\bigwedge m\ n.$ *N* $\leq$ *n* $\Longrightarrow$ *n* $\leq$ *m* $\Longrightarrow$ *sum ?f* (*?S1 m* $-$ *?S1 n*) $<$ *r*
by (*simp only*: *norm-sum-f*)
show $\exists\, N.\ \forall\, n{\geq}N.$ *sum ?f* (*?S1 n* $-$ *?S2 n*) $<$ *r*
proof (*intro exI allI impI*)
fix *n*
assume *2* $*$ *N* $\leq$ *n*
then have *n*: *N* $\leq$ *n div 2* **by** *simp*
have *sum ?f* (*?S1 n* $-$ *?S2 n*) $\leq$ *sum ?f* (*?S1 n* $-$ *?S1* (*n div 2*))
by (*intro sum-mono2 finite-Diff finite-S1 f-nonneg Diff-mono subset-refl S1-le-S2*)
also have ... $<$ *r*
using *n div-le-dividend* **by** (*rule N*)
finally show *sum ?f* (*?S1 n* $-$ *?S2 n*) $<$ *r* **.**
qed
qed

then have *Zfun* (λ*n*. *sum ?g* (*?S1 n* − *?S2 n*)) *sequentially*
apply (*rule Zfun-le* [*rule-format*])
apply (*simp only*: *norm-sum-f*)
apply (*rule order-trans* [*OF norm-sum sum-mono*])
apply (*auto simp add*: *norm-mult-ineq*)
done
then have *2*: (λ*n*. *sum ?g* (*?S1 n*) − *sum ?g* (*?S2 n*)) ⟶ *0*
unfolding *tendsto-Zfun-iff diff-0-right*
by (*simp only*: *sum-diff finite-S1 S2-le-S1*)
with *1* **have** (λ*n*. *sum ?g* (*?S2 n*)) ⟶ (∑ *k*. *a k*) ∗ (∑ *k*. *b k*)
by (*rule Lim-transform2*)
then show *?thesis*
by (*simp only*: *sums-def sum.triangle-reindex*)
qed

lemma *Cauchy-product*:
fixes *a b* :: *nat* ⇒ *'a*::{*real-normed-algebra*,*banach*}
assumes *summable* (λ*k*. *norm* (*a k*))
and *summable* (λ*k*. *norm* (*b k*))
shows (∑ *k*. *a k*) ∗ (∑ *k*. *b k*) = (∑ *k*. ∑ *i*≤*k*. *a i* ∗ *b* (*k* − *i*))
using *assms* **by** (*rule Cauchy-product-sums* [*THEN sums-unique*])

lemma *summable-Cauchy-product*:
fixes *a b* :: *nat* ⇒ *'a*::{*real-normed-algebra*,*banach*}
assumes *summable* (λ*k*. *norm* (*a k*))
and *summable* (λ*k*. *norm* (*b k*))
shows *summable* (λ*k*. ∑ *i*≤*k*. *a i* ∗ *b* (*k* − *i*))
using *Cauchy-product-sums*[*OF assms*] **by** (*simp add*: *sums-iff*)

109.12 Series on *real*s

lemma *summable-norm-comparison-test*:
∃ *N*. ∀ *n*≥*N*. *norm* (*f n*) ≤ *g n* ⟹ *summable g* ⟹ *summable* (λ*n*. *norm* (*f n*))
by (*rule summable-comparison-test*) *auto*

lemma *summable-rabs-comparison-test*: ∃ *N*. ∀ *n*≥*N*. |*f n*| ≤ *g n* ⟹ *summable g*
⟹ *summable* (λ*n*. |*f n*|)
for *f* :: *nat* ⇒ *real*
by (*rule summable-comparison-test*) *auto*

lemma *summable-rabs-cancel*: *summable* (λ*n*. |*f n*|) ⟹ *summable f*
for *f* :: *nat* ⇒ *real*
by (*rule summable-norm-cancel*) *simp*

lemma *summable-rabs*: *summable* (λ*n*. |*f n*|) ⟹ |*suminf f*| ≤ (∑ *n*. |*f n*|)
for *f* :: *nat* ⇒ *real*
by (*fold real-norm-def*) (*rule summable-norm*)

lemma *norm-suminf-le*:

assumes $\bigwedge n.\ norm\ (f\ n :: 'a :: banach) \leq g\ n\ summable\ g$
shows $norm\ (suminf\ f) \leq suminf\ g$
proof −
have $*$: $summable\ (\lambda n.\ norm\ (f\ n))$
using *assms summable-norm-comparison-test* **by** *blast*
hence $norm\ (suminf\ f) \leq (\sum n.\ norm\ (f\ n))$ **by** (*intro summable-norm*) *auto*
also have $\ldots \leq suminf\ g$ **by** (*intro suminf-le* $*$ *assms allI*)
finally show *?thesis* **.**
qed

lemma *norm-sums-le*:
assumes *f sums F g sums G*
assumes $\bigwedge n.\ norm\ (f\ n :: 'a :: banach) \leq g\ n$
shows $norm\ F \leq G$
using *assms* **by** (*metis norm-suminf-le sums-iff*)

lemma *summable-zero-power* [*simp*]: $summable\ (\lambda n.\ 0\ \hat{}\ n :: 'a :: \{comm\text{-}ring\text{-}1, topological\text{-}space\})$
by (*simp add*: *power-0-left*)

lemma *summable-zero-power'* [*simp*]: $summable\ (\lambda n.\ f\ n * 0\ \hat{}\ n :: 'a :: \{ring\text{-}1, topological\text{-}space\})$
proof −
have $(\lambda n.\ f\ n * 0\ \hat{}\ n :: 'a) = (\lambda n.\ if\ n = 0\ then\ f\ 0 * 0\hat{}0\ else\ 0)$
by (*intro ext*) (*simp add*: *zero-power*)
moreover have *summable* ... **by** *simp*
ultimately show *?thesis* **by** *simp*
qed

lemma *summable-power-series*:
fixes $z :: real$
assumes *le-1*: $\bigwedge i.\ f\ i \leq 1$
and *nonneg*: $\bigwedge i.\ 0 \leq f\ i$
and *z*: $0 \leq z\ z < 1$
shows $summable\ (\lambda i.\ f\ i * z\hat{}i)$
proof (*rule summable-comparison-test*[*OF - summable-geometric*])
show $norm\ z < 1$
using *z* **by** (*auto simp*: *less-imp-le*)
show $\bigwedge n.\ \exists N.\ \forall na \geq N.\ norm\ (f\ na * z\ \hat{}\ na) \leq z\ \hat{}\ na$
using *z*
by (*auto intro*!: *exI*[*of - 0*] *mult-left-le-one-le simp*: *abs-mult nonneg power-abs less-imp-le le-1*)
qed

lemma *summable-0-powser*: $summable\ (\lambda n.\ f\ n * 0\ \hat{}\ n :: 'a :: real\text{-}normed\text{-}div\text{-}algebra)$
by *simp*

lemma *summable-powser-split-head*:
$summable\ (\lambda n.\ f\ (Suc\ n) * z\ \hat{}\ n :: 'a :: real\text{-}normed\text{-}div\text{-}algebra) = summable\ (\lambda n.\ f\ n * z\ \hat{}\ n)$
proof −

```
  have summable (λn. f (Suc n) * z ^ n) ⟷ summable (λn. f (Suc n) * z ^ Suc
n)
    (is ?lhs ⟷ ?rhs)
  proof
    show ?rhs if ?lhs
      using summable-mult2[OF that, of z]
      by (simp add: power-commutes algebra-simps)
    show ?lhs if ?rhs
      using summable-mult2[OF that, of inverse z]
      by (cases z ≠ 0, subst (asm) power-Suc2) (simp-all add: algebra-simps)
  qed
  also have ... ⟷ summable (λn. f n * z ^ n) by (rule summable-Suc-iff)
  finally show ?thesis .
qed

lemma summable-powser-ignore-initial-segment:
  fixes f :: nat ⇒ 'a :: real-normed-div-algebra
  shows summable (λn. f (n + m) * z ^ n) ⟷ summable (λn. f n * z ^ n)
proof (induction m)
  case (Suc m)
  have summable (λn. f (n + Suc m) * z ^ n) = summable (λn. f (Suc n + m) *
z ^ n)
    by simp
  also have ... = summable (λn. f (n + m) * z ^ n)
    by (rule summable-powser-split-head)
  also have ... = summable (λn. f n * z ^ n)
    by (rule Suc.IH)
  finally show ?case .
qed simp-all

lemma powser-split-head:
  fixes f :: nat ⇒ 'a::{real-normed-div-algebra,banach}
  assumes summable (λn. f n * z ^ n)
  shows suminf (λn. f n * z ^ n) = f 0 + suminf (λn. f (Suc n) * z ^ n) * z
    and suminf (λn. f (Suc n) * z ^ n) * z = suminf (λn. f n * z ^ n) - f 0
    and summable (λn. f (Suc n) * z ^ n)
proof -
  from assms show summable (λn. f (Suc n) * z ^ n)
    by (subst summable-powser-split-head)
  from suminf-mult2[OF this, of z]
    have (∑n. f (Suc n) * z ^ n) * z = (∑n. f (Suc n) * z ^ Suc n)
    by (simp add: power-commutes algebra-simps)
  also from assms have ... = suminf (λn. f n * z ^ n) - f 0
    by (subst suminf-split-head) simp-all
  finally show suminf (λn. f n * z ^ n) = f 0 + suminf (λn. f (Suc n) * z ^ n)
* z
    by simp
  then show suminf (λn. f (Suc n) * z ^ n) * z = suminf (λn. f n * z ^ n) - f 0
    by simp
```

qed

lemma *summable-partial-sum-bound*:
fixes f :: $nat \Rightarrow 'a$ :: *banach*
and e :: *real*
assumes *summable*: *summable* f
and e: $e > 0$
obtains N **where** $\bigwedge m\ n.\ m \geq N \Longrightarrow norm\ (\sum k{=}m..n.\ f\ k) < e$
proof −
from *summable* **have** *Cauchy* $(\lambda n.\ \sum k{<}n.\ f\ k)$
by (*simp add*: *Cauchy-convergent-iff summable-iff-convergent*)
from *CauchyD* [*OF this e*] **obtain** N
where N: $\bigwedge m\ n.\ m \geq N \Longrightarrow n \geq N \Longrightarrow norm\ ((\sum k{<}m.\ f\ k) - (\sum k{<}n.\ f\ k)) < e$
by *blast*
have $norm\ (\sum k{=}m..n.\ f\ k) < e$ **if** m: $m \geq N$ **for** $m\ n$
proof (*cases* $n \geq m$)
case *True*
with m **have** $norm\ ((\sum k{<}Suc\ n.\ f\ k) - (\sum k{<}m.\ f\ k)) < e$
by (*intro N*) *simp-all*
also from *True* **have** $(\sum k{<}Suc\ n.\ f\ k) - (\sum k{<}m.\ f\ k) = (\sum k{=}m..n.\ f\ k)$
by (*subst sum-diff* [*symmetric*]) (*simp-all add*: *sum.last-plus*)
finally show *?thesis* **.**
next
case *False*
with e **show** *?thesis* **by** *simp-all*
qed
then show *?thesis* **by** (*rule that*)
qed

lemma *powser-sums-if*:
$(\lambda n.\ (if\ n = m\ then\ (1 :: 'a{::}\{ring\text{-}1, topological\text{-}space\})\ else\ 0) * z\hat{}n)\ sums\ z\hat{}m$
proof −
have $(\lambda n.\ (if\ n = m\ then\ 1\ else\ 0) * z\hat{}n) = (\lambda n.\ if\ n = m\ then\ z\hat{}n\ else\ 0)$
by (*intro ext*) *auto*
then show *?thesis*
by (*simp add*: *sums-single*)
qed

lemma
fixes f :: $nat \Rightarrow real$
assumes *summable* f
and *inj* g
and *pos*: $\bigwedge x.\ 0 \leq f\ x$
shows *summable-reindex*: *summable* $(f \circ g)$
and *suminf-reindex-mono*: $suminf\ (f \circ g) \leq suminf\ f$
and *suminf-reindex*: $(\bigwedge x.\ x \notin range\ g \Longrightarrow f\ x = 0) \Longrightarrow suminf\ (f \circ g) = suminf\ f$
proof −

from ‹*inj g*› **have** [*simp*]: $\bigwedge A.$ *inj-on g A*
by (*rule inj-on-subset*) *simp*

have *smaller*: $\forall n.$ $(\sum i{<}n.$ $(f \circ g)$ $i) \leq$ *suminf f*
proof
fix *n*
have $\forall$ $n' \in (g$ ‘ $\{..{<}n\}).$ $n' <$ *Suc* (*Max* $(g$ ‘ $\{..{<}n\}))$
by (*metis Max-ge finite-imageI finite-lessThan not-le not-less-eq*)
then obtain *m* **where** *n*: $\bigwedge n'.$ $n' < n \Longrightarrow g\ n' < m$
by *blast*

have $(\sum i{<}n.$ f $(g$ $i)) =$ *sum f* $(g$ ‘ $\{..{<}n\})$
by (*simp add*: *sum.reindex*)
also have $\ldots \leq (\sum i{<}m.$ f $i)$
by (*rule sum-mono2*) (*auto simp add*: *pos n*[*rule-format*])
also have $\ldots \leq$ *suminf f*
using ‹*summable f*›
by (*rule sum-le-suminf*) (*simp-all add*: *pos*)
finally show $(\sum i{<}n.$ $(f \circ$ $g)$ $i) \leq$ *suminf f*
by *simp*
qed

have *incseq* $(\lambda n.$ $\sum i{<}n.$ $(f \circ g)$ $i)$
by (*rule incseq-SucI*) (*auto simp add*: *pos*)
then obtain L **where** L: $(\lambda$ $n.$ $\sum i{<}n.$ $(f \circ g)$ $i) \longrightarrow L$
using *smaller* **by**(*rule incseq-convergent*)
then have $(f \circ g)$ *sums L*
by (*simp add*: *sums-def*)
then show *summable* $(f \circ g)$
by (*auto simp add*: *sums-iff*)

then have $(\lambda n.$ $\sum i{<}n.$ $(f \circ g)$ $i) \longrightarrow$ *suminf* $(f \circ g)$
by (*rule summable-LIMSEQ*)
then show *le*: *suminf* $(f \circ g) \leq$ *suminf f*
by(*rule LIMSEQ-le-const2*)(*blast intro*: *smaller*[*rule-format*])

assume *f*: $\bigwedge x.$ $x \notin$ *range* $g \Longrightarrow f\ x = 0$

from ‹*summable f*› **have** *suminf f* $\leq$ *suminf* $(f \circ g)$
proof (*rule suminf-le-const*)
fix *n*
have $\forall$ $n' \in (g -$‘ $\{..{<}n\}).$ $n' <$ *Suc* (*Max* $(g -$‘ $\{..{<}n\}))$
by(*auto intro*: *Max-ge simp add*: *finite-vimageI less-Suc-eq-le*)
then obtain *m* **where** *n*: $\bigwedge n'.$ $g\ n' < n \Longrightarrow n' < m$
by *blast*
have $(\sum i{<}n.$ f $i) = (\sum i{\in}\{..{<}n\} \cap$ *range g*. f $i)$
using *f* **by**(*auto intro*: *sum.mono-neutral-cong-right*)
also have $\ldots = (\sum i{\in}g -$‘ $\{..{<}n\}.$ $(f \circ g)$ $i)$
by (*rule sum.reindex-cong*[**where** *l*=*g*])(*auto*)

```
    also have ... ≤ (∑ i<m. (f ∘ g) i)
      by (rule sum-mono2)(auto simp add: pos n)
    also have ... ≤ suminf (f ∘ g)
      using ‹summable (f ∘ g)› by (rule sum-le-suminf) (simp-all add: pos)
    finally show sum f {..<n} ≤ suminf (f ∘ g) .
  qed
  with le show suminf (f ∘ g) = suminf f
    by (rule antisym)
qed

lemma sums-mono-reindex:
  assumes subseq: strict-mono g
    and zero: ⋀n. n ∉ range g ⟹ f n = 0
  shows (λn. f (g n)) sums c ⟷ f sums c
  unfolding sums-def
proof
  assume lim: (λn. ∑ k<n. f k) ⟶ c
  have (λn. ∑ k<n. f (g k)) = (λn. ∑ k<g n. f k)
  proof
    fix n :: nat
    from subseq have (∑ k<n. f (g k)) = (∑ k∈g`{..<n}. f k)
      by (subst sum.reindex) (auto intro: strict-mono-imp-inj-on)
    also from subseq have ... = (∑ k<g n. f k)
      by (intro sum.mono-neutral-left ballI zero)
        (auto simp: strict-mono-less strict-mono-less-eq)
    finally show (∑ k<n. f (g k)) = (∑ k<g n. f k) .
  qed
  also from LIMSEQ-subseq-LIMSEQ[OF lim subseq] have ... ⟶ c
    by (simp only: o-def)
  finally show (λn. ∑ k<n. f (g k)) ⟶ c .
next
  assume lim: (λn. ∑ k<n. f (g k)) ⟶ c
  define g-inv where g-inv n = (LEAST m. g m ≥ n) for n
  from filterlim-subseq[OF subseq] have g-inv-ex: ∃ m. g m ≥ n for n
    by (auto simp: filterlim-at-top eventually-at-top-linorder)
  then have g-inv: g (g-inv n) ≥ n for n
    unfolding g-inv-def by (rule LeastI-ex)
  have g-inv-least: m ≥ g-inv n if g m ≥ n for m n
    using that unfolding g-inv-def by (rule Least-le)
  have g-inv-least': g m < n if m < g-inv n for m n
    using that g-inv-least[of n m] by linarith
  have (λn. ∑ k<n. f k) = (λn. ∑ k<g-inv n. f (g k))
  proof
    fix n :: nat
    {
      fix k
      assume k: k ∈ {..<n} - g`{..<g-inv n}
      have k ∉ range g
      proof (rule notI, elim imageE)
```

```
      fix l
      assume l: k = g l
      have g l < g (g-inv n)
        by (rule less-le-trans[OF - g-inv]) (use k l in simp-all)
      with subseq have l < g-inv n
        by (simp add: strict-mono-less)
      with k l show False
        by simp
    qed
    then have f k = 0
      by (rule zero)
  }
  with g-inv-least' g-inv have (∑ k<n. f k) = (∑ k∈g`{..<g-inv n}. f k)
    by (intro sum.mono-neutral-right) auto
  also from subseq have ... = (∑ k<g-inv n. f (g k))
    using strict-mono-imp-inj-on by (subst sum.reindex) simp-all
  finally show (∑ k<n. f k) = (∑ k<g-inv n. f (g k)) .
qed
also {
  fix K n :: nat
  assume g K ≤ n
  also have n ≤ g (g-inv n)
    by (rule g-inv)
  finally have K ≤ g-inv n
    using subseq by (simp add: strict-mono-less-eq)
}
then have filterlim g-inv at-top sequentially
  by (auto simp: filterlim-at-top eventually-at-top-linorder)
with lim have (λn. ∑ k<g-inv n. f (g k)) ⟶ c
  by (rule filterlim-compose)
finally show (λn. ∑ k<n. f k) ⟶ c .
qed

lemma summable-mono-reindex:
  assumes subseq: strict-mono g
    and zero: ⋀n. n ∉ range g ⟹ f n = 0
  shows summable (λn. f (g n)) ⟷ summable f
  using sums-mono-reindex[of g f, OF assms] by (simp add: summable-def)

lemma suminf-mono-reindex:
  fixes f :: nat ⇒ 'a::{t2-space,comm-monoid-add}
  assumes strict-mono g ⋀n. n ∉ range g ⟹ f n = 0
  shows   suminf (λn. f (g n)) = suminf f
proof (cases summable f)
  case True
  with sums-mono-reindex [of g f, OF assms]
    and summable-mono-reindex [of g f, OF assms]
  show ?thesis
    by (simp add: sums-iff)
```

next
case *False*
then have ¬(∃ *c*. *f sums c*)
unfolding *summable-def* **by** *blast*
then have *suminf f* = *The* (λ-. *False*)
by (*simp add*: *suminf-def*)
moreover from *False* **have** ¬ *summable* (λ*n*. *f* (*g n*))
using *summable-mono-reindex*[*of g f*, *OF assms*] **by** *simp*
then have ¬(∃ *c*. (λ*n*. *f* (*g n*)) *sums c*)
unfolding *summable-def* **by** *blast*
then have *suminf* (λ*n*. *f* (*g n*)) = *The* (λ-. *False*)
by (*simp add*: *suminf-def*)
ultimately show *?thesis* **by** *simp*
qed

lemma *summable-bounded-partials*:
fixes *f* :: *nat* ⇒ *'a* :: {*real-normed-vector*,*complete-space*}
assumes *bound*: *eventually* (λ*x0*. ∀ *a*≥*x0*. ∀ *b*>*a*. *norm* (*sum f* {*a*<..*b*}) ≤ *g a*) *sequentially*
assumes *g*: *g* ⟶ *0*
shows *summable f* **unfolding** *summable-iff-convergent'*
proof (*intro Cauchy-convergent CauchyI'*, *goal-cases*)
case (*1* ε)
with *g* **have** *eventually* (λ*x*. |*g x*| < ε) *sequentially*
by (*auto simp*: *tendsto-iff*)
from *eventually-conj*[*OF this bound*] **obtain** *x0* **where** *x0*:
⋀*x*. *x* ≥ *x0* ⟹ |*g x*| < ε ⋀*a b*. *x0* ≤ *a* ⟹ *a* < *b* ⟹ *norm* (*sum f* {*a*<..*b*}) ≤ *g a*
unfolding *eventually-at-top-linorder* **by** *auto*
show *?case*
proof (*intro exI*[*of - x0*] *allI impI*)
fix *m n* **assume** *mn*: *x0* ≤ *m m* < *n*
have *dist* (*sum f* {..*m*}) (*sum f* {..*n*}) = *norm* (*sum f* {..*n*} − *sum f* {..*m*})
by (*simp add*: *dist-norm norm-minus-commute*)
also have *sum f* {..*n*} − *sum f* {..*m*} = *sum f* ({..*n*} − {..*m*})
using *mn* **by** (*intro Groups-Big.sum-diff* [*symmetric*]) *auto*
also have {..*n*} − {..*m*} = {*m*<..*n*} **using** *mn* **by** *auto*
also have *norm* (*sum f* {*m*<..*n*}) ≤ *g m* **using** *mn* **by** (*intro x0*) *auto*
also have ... ≤ |*g m*| **by** *simp*
also have ... < ε **using** *mn* **by** (*intro x0*) *auto*
finally show *dist* (*sum f* {..*m*}) (*sum f* {..*n*}) < ε **.**
qed
qed

end

110 Differentiation

theory *Deriv*

imports *Limits*
begin

110.1 Frechet derivative

definition *has-derivative* :: ('*a*::*real-normed-vector* ⇒ '*b*::*real-normed-vector*) ⇒
('*a* ⇒ '*b*) ⇒ '*a filter* ⇒ *bool* (**infix** ‹(*has'-derivative*)› *50*)
where (*f has-derivative f'*) *F* ⟷
bounded-linear f' ∧
((λ*y*. ((*f y* − *f* (*Lim F* (λ*x*. *x*))) − *f'* (*y* − *Lim F* (λ*x*. *x*))) /$_R$ *norm* (*y* − *Lim F* (λ*x*. *x*))) ⟶ *0*) *F*

Usually the filter *F* is *at x within s*. (*f has-derivative D*) (*at x within s*) means: *D* is the derivative of function *f* at point *x* within the set *s*. Where *s* is used to express left or right sided derivatives. In most cases *s* is either a variable or *UNIV*.

These are the only cases we'll care about, probably.

lemma *has-derivative-within*: (*f has-derivative f'*) (*at x within s*) ⟷
bounded-linear f' ∧ ((λ*y*. (*1* / *norm*(*y* − *x*)) *$*$_R$ (*f y* − (*f x* + *f'* (*y* − *x*)))) ⟶ *0*) (*at x within s*)
unfolding *has-derivative-def tendsto-iff*
by (*subst eventually-Lim-ident-at*) (*auto simp add*: *field-simps*)

lemma *has-derivative-eq-rhs*: (*f has-derivative f'*) *F* ⟹ *f'* = *g'* ⟹ (*f has-derivative g'*) *F*
by *simp*

definition *has-field-derivative* :: ('*a*::*real-normed-field* ⇒ '*a*) ⇒ '*a* ⇒ '*a filter* ⇒ *bool*
(**infix** ‹(*has'-field'-derivative*)› *50*)
where (*f has-field-derivative D*) *F* ⟷ (*f has-derivative* (∗) *D*) *F*

lemma *DERIV-cong*: (*f has-field-derivative X*) *F* ⟹ *X* = *Y* ⟹ (*f has-field-derivative Y*) *F*
by *simp*

definition *has-vector-derivative* :: (*real* ⇒ '*b*::*real-normed-vector*) ⇒ '*b* ⇒ *real filter* ⇒ *bool*
(**infix** ‹*has'-vector'-derivative*› *50*)
where (*f has-vector-derivative f'*) *net* ⟷ (*f has-derivative* (λ*x*. *x* *$*$_R$ *f'*)) *net*

lemma *has-vector-derivative-eq-rhs*:
(*f has-vector-derivative X*) *F* ⟹ *X* = *Y* ⟹ (*f has-vector-derivative Y*) *F*
by *simp*

named-theorems *derivative-intros*
named-theorems *derivative-cong*

lemmas [*derivative-cong*] =
has-derivative-eq-rhs DERIV-cong has-vector-derivative-eq-rhs

setup ‹
let
fun eq-thms ctxt = Named-Theorems.get ctxt ***named-theorems*** ‹*derivative-cong*›
fun eq-rule ctxt thm = get-first (try (fn eq-thm => eq-thm OF [thm])) (eq-thms ctxt)
in
Global-Theory.add-thms-dynamic
(***binding*** ‹*derivative-eq-intros*›,
fn context =>
Named-Theorems.get (Context.proof-of context) ***named-theorems*** ‹*derivative-intros*›
|> map-filter (eq-rule (Context.proof-of context)))
end
›

The following syntax is only used as a legacy syntax.

abbreviation (*input*)
FDERIV :: (*'a*::*real-normed-vector* ⇒ *'b*::*real-normed-vector*) ⇒ *'a* ⇒ (*'a* ⇒ *'b*) ⇒ *bool*
(‹(‹*notation*=‹*mixfix FDERIV*››*FDERIV* (-)/ (-)/ :> (-))› [*1000*, *1000*, *60*] *60*)
where *FDERIV f x :> f'* ≡ (*f has-derivative f'*) (*at x*)

lemma *has-derivative-bounded-linear*: (*f has-derivative f'*) *F* ⟹ *bounded-linear f'*
by (*simp add*: *has-derivative-def*)

lemma *has-derivative-linear*: (*f has-derivative f'*) *F* ⟹ *linear f'*
using *bounded-linear.linear*[*OF has-derivative-bounded-linear*] **.**

lemma *has-derivative-ident*[*derivative-intros*, *simp*]: ((λ*x*. *x*) *has-derivative* (λ*x*. *x*)) *F*
by (*simp add*: *has-derivative-def*)

lemma *has-derivative-id* [*derivative-intros*, *simp*]: (*id has-derivative id*) *F*
by (*metis eq-id-iff has-derivative-ident*)

lemma *shift-has-derivative-id*: ((+) *d has-derivative* (λ*x*. *x*)) *F*
using *has-derivative-def* **by** *fastforce*

lemma *has-derivative-const*[*derivative-intros*, *simp*]: ((λ*x*. *c*) *has-derivative* (λ*x*. *0*)) *F*
by (*simp add*: *has-derivative-def*)

lemma (**in** *bounded-linear*) *bounded-linear*: *bounded-linear f* **..**

lemma (**in** *bounded-linear*) *has-derivative*:
(*g has-derivative g'*) *F* ⟹ ((λ*x*. *f* (*g x*)) *has-derivative* (λ*x*. *f* (*g' x*))) *F*

unfolding *has-derivative-def*
by (*auto simp add*: *bounded-linear-compose* [*OF bounded-linear*] *scaleR diff dest*: *tendsto*)

lemma *has-derivative-bot* [*intro*]: *bounded-linear f'* ⟹ (*f has-derivative f'*) *bot*
by (*auto simp*: *has-derivative-def*)

lemma *has-field-derivative-bot* [*simp*, *intro*]: (*f has-field-derivative f'*) *bot*
by (*auto simp*: *has-field-derivative-def intro*!: *has-derivative-bot bounded-linear-mult-right*)

lemmas *has-derivative-scaleR-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-scaleR-right*]

lemmas *has-derivative-scaleR-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-scaleR-left*]

lemmas *has-derivative-mult-right* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-mult-right*]

lemmas *has-derivative-mult-left* [*derivative-intros*] =
bounded-linear.has-derivative [*OF bounded-linear-mult-left*]

lemmas *has-derivative-of-real*[*derivative-intros*, *simp*] =
bounded-linear.has-derivative[*OF bounded-linear-of-real*]

lemma *has-derivative-add*[*simp*, *derivative-intros*]:
assumes *f*: (*f has-derivative f'*) *F*
and *g*: (*g has-derivative g'*) *F*
shows ((λ*x*. *f x* + *g x*) *has-derivative* (λ*x*. *f' x* + *g' x*)) *F*
unfolding *has-derivative-def*
proof *safe*
let *?x* = *Lim F* (λ*x*. *x*)
let *?D* = λ*f f' y*. ((*f y* − *f ?x*) − *f'* (*y* − *?x*)) /$_R$ *norm* (*y* − *?x*)
have ((λ*x*. *?D f f' x* + *?D g g' x*) ⟶ (*0* + *0*)) *F*
using *f g* **by** (*intro tendsto-add*) (*auto simp*: *has-derivative-def*)
then show (*?D* (λ*x*. *f x* + *g x*) (λ*x*. *f' x* + *g' x*) ⟶ *0*) *F*
by (*simp add*: *field-simps scaleR-add-right scaleR-diff-right*)
qed (*blast intro*: *bounded-linear-add f g has-derivative-bounded-linear*)

lemma *has-derivative-sum*[*simp*, *derivative-intros*]:
(⋀*i*. *i* ∈ *I* ⟹ (*f i has-derivative f' i*) *F*) ⟹
((λ*x*. ∑ *i*∈*I*. *f i x*) *has-derivative* (λ*x*. ∑ *i*∈*I*. *f' i x*)) *F*
by (*induct I rule*: *infinite-finite-induct*) *simp-all*

lemma *has-derivative-minus*[*simp*, *derivative-intros*]:
(*f has-derivative f'*) *F* ⟹ ((λ*x*. − *f x*) *has-derivative* (λ*x*. − *f' x*)) *F*
using *has-derivative-scaleR-right*[*of f f' F* −*1*] **by** *simp*

lemma *has-derivative-diff*[*simp*, *derivative-intros*]:

(*f has-derivative f'*) *F* ⟹ (*g has-derivative g'*) *F* ⟹
((λ*x*. *f x* − *g x*) *has-derivative* (λ*x*. *f' x* − *g' x*)) *F*
by (*simp only*: *diff-conv-add-uminus has-derivative-add has-derivative-minus*)

lemma *has-derivative-at-within*:
(*f has-derivative f'*) (*at x within s*) ⟷
(*bounded-linear f'* ∧ ((λ*y*. ((*f y* − *f x*) − *f'* (*y* − *x*)) /$_R$ *norm* (*y* − *x*)) ⟶ *0*) (*at x within s*))
proof (*cases at x within s* = *bot*)
case *True*
then show *?thesis*
by (*metis* (*no-types*, *lifting*) *has-derivative-within tendsto-bot*)
next
case *False*
then show *?thesis*
by (*simp add*: *Lim-ident-at has-derivative-def*)
qed

lemma *has-derivative-iff-norm*:
(*f has-derivative f'*) (*at x within s*) ⟷
bounded-linear f' ∧ ((λ*y*. *norm* ((*f y* − *f x*) − *f'* (*y* − *x*)) / *norm* (*y* − *x*)) ⟶ *0*) (*at x within s*)
using *tendsto-norm-zero-iff*[*of* - *at x within s*, **where** *'b*=*'b*, *symmetric*]
by (*simp add*: *has-derivative-at-within divide-inverse ac-simps*)

lemma *has-derivative-at*:
(*f has-derivative D*) (*at x*) ⟷
(*bounded-linear D* ∧ (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D h*) / *norm h*) −*0*→ *0*)
by (*simp add*: *has-derivative-iff-norm LIM-offset-zero-iff*)

lemma *field-has-derivative-at*:
fixes *x* :: *'a*::*real-normed-field*
shows (*f has-derivative* (∗) *D*) (*at x*) ⟷ (λ*h*. (*f* (*x* + *h*) − *f x*) / *h*) −*0*→ *D* (**is** *?lhs* = *?rhs*)
proof −
have *?lhs* = (λ*h*. *norm* (*f* (*x* + *h*) − *f x* − *D* ∗ *h*) / *norm h*) −*0* → *0*
by (*simp add*: *bounded-linear-mult-right has-derivative-at*)
also have ... = (λ*y*. *norm* ((*f* (*x* + *y*) − *f x* − *D* ∗ *y*) / *y*)) −*0*→ *0*
by (*simp cong*: *LIM-cong flip*: *nonzero-norm-divide*)
also have ... = (λ*y*. *norm* ((*f* (*x* + *y*) − *f x*) / *y* − *D* / *y* ∗ *y*)) −*0*→ *0*
by (*simp only*: *diff-divide-distrib times-divide-eq-left* [*symmetric*])
also have ... = *?rhs*
by (*simp add*: *tendsto-norm-zero-iff LIM-zero-iff cong*: *LIM-cong*)
finally show *?thesis* **.**
qed

lemma *has-derivative-iff-Ex*:
(*f has-derivative f'*) (*at x*) ⟷
bounded-linear f' ∧ (∃ *e*. (∀ *h*. *f* (*x*+*h*) = *f x* + *f' h* + *e h*) ∧ ((λ*h*. *norm* (*e h*)

/ norm h) ⟶ 0) (at 0))
unfolding *has-derivative-at* **by** *force*

lemma *has-derivative-at-within-iff-Ex*:
assumes *x ∈ S open S*
shows *(f has-derivative f') (at x within S) ⟷*
bounded-linear f' ∧ (∃ e. (∀ h. x+h ∈ S ⟶ f (x+h) = f x + f' h + e h) ∧
((λh. norm (e h) / norm h) ⟶ 0) (at 0))
(is *?lhs = ?rhs***)**
proof *safe*
show *bounded-linear f'*
if *(f has-derivative f') (at x within S)*
using *has-derivative-bounded-linear that* **by** *blast*
show *∃ e. (∀ h. x + h ∈ S ⟶ f (x + h) = f x + f' h + e h) ∧ (λh. norm (e h)*
/ norm h) −0→ 0
if *(f has-derivative f') (at x within S)*
by *(metis (full-types) assms that has-derivative-iff-Ex at-within-open)*
show *(f has-derivative f') (at x within S)*
if *bounded-linear f'*
and *eq [rule-format]: ∀ h. x + h ∈ S ⟶ f (x + h) = f x + f' h + e h*
and *0: (λh. norm (e (h::'a)::'b) / norm h) −0→ 0*
for *e*
proof *−*
have *1: f y − f x = f' (y−x) + e (y−x)* **if** *y ∈ S* **for** *y*
using *eq [of y−x] that* **by** *simp*
have *2: ((λy. norm (e (y−x)) / norm (y − x)) ⟶ 0) (at x within S)*
by *(simp add: 0 assms tendsto-offset-zero-iff)*
have *((λy. norm (f y − f x − f' (y − x)) / norm (y − x)) ⟶ 0) (at x*
within S)
by *(simp add: Lim-cong-within 1 2)*
then show *?thesis*
by *(simp add: has-derivative-iff-norm ‹bounded-linear f'›)*
qed
qed

lemma *has-derivativeI*:
bounded-linear f' ⟹
((λy. ((f y − f x) − f' (y − x)) /ᵣ norm (y − x)) ⟶ 0) (at x within s) ⟹
(f has-derivative f') (at x within s)
by *(simp add: has-derivative-at-within)*

lemma *has-derivativeI-sandwich*:
assumes *e: 0 < e*
and *bounded: bounded-linear f'*
and *sandwich: (⋀y. y ∈ s ⟹ y ≠ x ⟹ dist y x < e ⟹*
norm ((f y − f x) − f' (y − x)) / norm (y − x) ≤ H y)
and *(H ⟶ 0) (at x within s)*
shows *(f has-derivative f') (at x within s)*
unfolding *has-derivative-iff-norm*

proof *safe*
show $((\lambda y.\ norm\ (f\ y - f\ x - f'\ (y - x))\ /\ norm\ (y - x)) \longrightarrow 0)\ (at\ x\ within\ s)$
proof (*rule tendsto-sandwich*[**where** $f{=}\lambda x.\ 0$])
show $(H \longrightarrow 0)\ (at\ x\ within\ s)$ **by** *fact*
show $eventually\ (\lambda n.\ norm\ (f\ n - f\ x - f'\ (n - x))\ /\ norm\ (n - x) \leq H\ n)\ (at\ x\ within\ s)$
unfolding *eventually-at* **using** *e sandwich* **by** *auto*
qed (*auto simp*: *le-divide-eq*)
qed *fact*

lemma *has-derivative-subset*:
$(f\ has\text{-}derivative\ f')\ (at\ x\ within\ s) \Longrightarrow t \subseteq s \Longrightarrow (f\ has\text{-}derivative\ f')\ (at\ x\ within\ t)$
by (*auto simp add*: *has-derivative-iff-norm intro*: *tendsto-within-subset*)

lemma *has-derivative-within-singleton-iff*:
$(f\ has\text{-}derivative\ g)\ (at\ x\ within\ \{x\}) \longleftrightarrow bounded\text{-}linear\ g$
by (*auto intro*!: *has-derivativeI-sandwich*[**where** *e=1*] *has-derivative-bounded-linear*)

110.1.1 Limit transformation for derivatives

lemma *has-derivative-transform-within*:
assumes $(f\ has\text{-}derivative\ f')\ (at\ x\ within\ s)$
and $0 < d$
and $x \in s$
and $\bigwedge x'.\ [\![x' \in s;\ dist\ x'\ x < d]\!] \Longrightarrow f\ x' = g\ x'$
shows $(g\ has\text{-}derivative\ f')\ (at\ x\ within\ s)$
using *assms*
unfolding *has-derivative-within*
by (*force simp add*: *intro*: *Lim-transform-within*)

lemma *has-derivative-transform-within-open*:
assumes $(f\ has\text{-}derivative\ f')\ (at\ x\ within\ t)$
and *open s*
and $x \in s$
and $\bigwedge x.\ x{\in}s \Longrightarrow f\ x = g\ x$
shows $(g\ has\text{-}derivative\ f')\ (at\ x\ within\ t)$
using *assms* **unfolding** *has-derivative-within*
by (*force simp add*: *intro*: *Lim-transform-within-open*)

lemma *has-derivative-transform*:
assumes $x \in s\ \bigwedge x.\ x \in s \Longrightarrow g\ x = f\ x$
assumes $(f\ has\text{-}derivative\ f')\ (at\ x\ within\ s)$
shows $(g\ has\text{-}derivative\ f')\ (at\ x\ within\ s)$
using *assms*
by (*intro has-derivative-transform-within*[*OF - zero-less-one*, **where** $g{=}g$]) *auto*

lemma *has-derivative-transform-eventually*:

assumes (*f has-derivative f'*) (*at x within s*)
(∀$_F$ *x'* *in at x within s*. *f x' = g x'*)
assumes *f x = g x x ∈ s*
shows (*g has-derivative f'*) (*at x within s*)
using *assms*
proof −
from *assms*(*2*,*3*) **obtain** *d* **where** *d > 0* ⋀*x'*. *x' ∈ s* ⟹ *dist x' x < d* ⟹ *f x' = g x'*
by (*force simp*: *eventually-at*)
from *has-derivative-transform-within*[*OF assms*(*1*) *this*(*1*) *assms*(*4*) *this*(*2*)]
show *?thesis* **.**
qed

lemma *has-field-derivative-transform-within*:
assumes (*f has-field-derivative f'*) (*at a within S*)
and *0 < d*
and *a ∈ S*
and ⋀*x*. ⟦*x ∈ S*; *dist x a < d*⟧ ⟹ *f x = g x*
shows (*g has-field-derivative f'*) (*at a within S*)
using *assms* **unfolding** *has-field-derivative-def*
by (*metis has-derivative-transform-within*)

lemma *has-field-derivative-transform-within-open*:
assumes (*f has-field-derivative f'*) (*at a*)
and *open S a ∈ S*
and ⋀*x*. *x ∈ S* ⟹ *f x = g x*
shows (*g has-field-derivative f'*) (*at a*)
using *assms* **unfolding** *has-field-derivative-def*
by (*metis has-derivative-transform-within-open*)

110.2 Continuity

lemma *has-derivative-continuous*:
assumes *f*: (*f has-derivative f'*) (*at x within s*)
shows *continuous* (*at x within s*) *f*
proof −
from *f* **interpret** *F*: *bounded-linear f'*
by (*rule has-derivative-bounded-linear*)
note *F.tendsto*[*tendsto-intros*]
let *?L* = λ*f*. (*f* ⟶ *0*) (*at x within s*)
have *?L* (λ*y*. *norm* ((*f y − f x*) − *f'* (*y − x*)) / *norm* (*y − x*))
using *f* **unfolding** *has-derivative-iff-norm* **by** *blast*
then have *?L* (λ*y*. *norm* ((*f y − f x*) − *f'* (*y − x*)) / *norm* (*y − x*) ∗ *norm* (*y − x*)) (**is** *?m*)
by (*rule tendsto-mult-zero*) (*auto intro*!: *tendsto-eq-intros*)
also have *?m* ⟷ *?L* (λ*y*. *norm* ((*f y − f x*) − *f'* (*y − x*)))
by (*intro filterlim-cong*) (*simp-all add*: *eventually-at-filter*)
finally have *?L* (λ*y*. (*f y − f x*) − *f'* (*y − x*))
by (*rule tendsto-norm-zero-cancel*)

then have *?L* (λ*y*. ((*f y* − *f x*) − *f′* (*y* − *x*)) + *f′* (*y* − *x*))
by (*rule tendsto-eq-intros*) (*auto intro*!: *tendsto-eq-intros simp*: *F.zero*)
then have *?L* (λ*y*. *f y* − *f x*)
by *simp*
from *tendsto-add*[*OF this tendsto-const*, *of f x*] **show** *?thesis*
by (*simp add*: *continuous-within*)
qed

110.3 Composition

lemma *tendsto-at-iff-tendsto-nhds-within*:
f x = *y* ⟹ (*f* ⟶ *y*) (*at x within s*) ⟷ (*f* ⟶ *y*) (*inf* (*nhds x*) (*principal s*))
unfolding *tendsto-def eventually-inf-principal eventually-at-filter*
by (*intro ext all-cong imp-cong*) (*auto elim*!: *eventually-mono*)

lemma *has-derivative-in-compose*:
assumes *f*: (*f has-derivative f′*) (*at x within s*)
and *g*: (*g has-derivative g′*) (*at* (*f x*) *within* (*f*‘*s*))
shows ((λ*x*. *g* (*f x*)) *has-derivative* (λ*x*. *g′* (*f′ x*))) (*at x within s*)
proof −
from *f* **interpret** *F*: *bounded-linear f′*
by (*rule has-derivative-bounded-linear*)
from *g* **interpret** *G*: *bounded-linear g′*
by (*rule has-derivative-bounded-linear*)
from *F.bounded* **obtain** *kF* **where** *kF*: ⋀*x*. *norm* (*f′ x*) ≤ *norm x* ∗ *kF*
by *fast*
from *G.bounded* **obtain** *kG* **where** *kG*: ⋀*x*. *norm* (*g′ x*) ≤ *norm x* ∗ *kG*
by *fast*
note *G.tendsto*[*tendsto-intros*]

let *?L* = λ*f*. (*f* ⟶ *0*) (*at x within s*)
let *?D* = λ*f f′ x y*. (*f y* − *f x*) − *f′* (*y* − *x*)
let *?N* = λ*f f′ x y*. *norm* (*?D f f′ x y*) / *norm* (*y* − *x*)
let *?gf* = λ*x*. *g* (*f x*) **and** *?gf′* = λ*x*. *g′* (*f′ x*)
define *Nf* **where** *Nf* = *?N f f′ x*
define *Ng* **where** [*abs-def*]: *Ng y* = *?N g g′* (*f x*) (*f y*) **for** *y*

show *?thesis*
proof (*rule has-derivativeI-sandwich*[*of 1*])
show *bounded-linear* (λ*x*. *g′* (*f′ x*))
using *f g* **by** (*blast intro*: *bounded-linear-compose has-derivative-bounded-linear*)
next
fix *y* :: *′a*
assume *neq*: *y* ≠ *x*
have *?N ?gf ?gf′ x y* = *norm* (*g′* (*?D f f′ x y*) + *?D g g′* (*f x*) (*f y*)) / *norm* (*y* − *x*)
by (*simp add*: *G.diff G.add field-simps*)
also have ... ≤ *norm* (*g′* (*?D f f′ x y*)) / *norm* (*y* − *x*) + *Ng y* ∗ (*norm* (*f y*

```
− f x) / norm (y − x))
    by (simp add: add-divide-distrib[symmetric] divide-right-mono norm-triangle-ineq
G.zero Ng-def)
    also have ... ≤ Nf y * kG + Ng y * (Nf y + kF)
    proof (intro add-mono mult-left-mono)
      have norm (f y − f x) = norm (?D f f' x y + f' (y − x))
        by simp
      also have ... ≤ norm (?D f f' x y) + norm (f' (y − x))
        by (rule norm-triangle-ineq)
      also have ... ≤ norm (?D f f' x y) + norm (y − x) * kF
        using kF by (intro add-mono) simp
      finally show norm (f y − f x) / norm (y − x) ≤ Nf y + kF
        by (simp add: neq Nf-def field-simps)
    qed (use kG in ‹simp-all add: Ng-def Nf-def neq zero-le-divide-iff field-simps›)
    finally show ?N ?gf ?gf' x y ≤ Nf y * kG + Ng y * (Nf y + kF) .
  next
    have [tendsto-intros]: ?L Nf
      using f unfolding has-derivative-iff-norm Nf-def ..
    from f have (f ⟶ f x) (at x within s)
      by (blast intro: has-derivative-continuous continuous-within[THEN iffD1])
    then have f': LIM x at x within s. f x :> inf (nhds (f x)) (principal (f‘s))
      unfolding filterlim-def
      by (simp add: eventually-filtermap eventually-at-filter le-principal)

    have ((?N g  g' (f x)) ⟶ 0) (at (f x) within f‘s)
      using g unfolding has-derivative-iff-norm ..
    then have g': ((?N g  g' (f x)) ⟶ 0) (inf (nhds (f x)) (principal (f‘s)))
      by (rule tendsto-at-iff-tendsto-nhds-within[THEN iffD1, rotated]) simp

    have [tendsto-intros]: ?L Ng
      unfolding Ng-def by (rule filterlim-compose[OF g' f'])
    show ((λy. Nf y * kG + Ng y * (Nf y + kF)) ⟶ 0) (at x within s)
      by (intro tendsto-eq-intros) auto
  qed simp
qed

lemma has-derivative-compose:
  (f has-derivative f') (at x within s) ⟹ (g has-derivative g') (at (f x)) ⟹
  ((λx. g (f x)) has-derivative (λx. g' (f' x))) (at x within s)
  by (blast intro: has-derivative-in-compose has-derivative-subset)

lemma has-derivative-in-compose2:
  assumes ⋀x. x ∈ t ⟹ (g has-derivative g' x) (at x within t)
  assumes f ‘ s ⊆ t x ∈ s
  assumes (f has-derivative f') (at x within s)
  shows ((λx. g (f x)) has-derivative (λy. g' (f x) (f' y))) (at x within s)
  using assms
  by (auto intro: has-derivative-subset intro!: has-derivative-in-compose[of f f' x s
g])
```

lemma (**in** *bounded-bilinear*) *FDERIV*:
assumes *f*: (*f has-derivative f'*) (*at x within s*) **and** *g*: (*g has-derivative g'*) (*at x within s*)
shows ((λ*x*. *f x ** g x*) *has-derivative* (λ*h*. *f x ** *g' h* + *f' h ** *g x*)) (*at x within s*)
proof −
from *bounded-linear.bounded* [*OF has-derivative-bounded-linear* [*OF f*]]
obtain *KF* **where** *norm-F*: ⋀*x*. *norm* (*f' x*) ≤ *norm x* ∗ *KF* **by** *fast*

from *pos-bounded* **obtain** *K*
where *K*: *0* < *K* **and** *norm-prod*: ⋀*a b*. *norm* (*a ** *b*) ≤ *norm a* ∗ *norm b* ∗ *K*
by *fast*
let *?D* = λ*f f' y*. *f y* − *f x* − *f'* (*y* − *x*)
let *?N* = λ*f f' y*. *norm* (*?D f f' y*) / *norm* (*y* − *x*)
define *Ng* **where** *Ng* = *?N g g'*
define *Nf* **where** *Nf* = *?N f f'*

let *?fun1* = λ*y*. *norm* (*f y ** *g y* − *f x ** *g x* − (*f x ** *g'* (*y* − *x*) + *f'* (*y* − *x*) ** *g x*)) / *norm* (*y* − *x*)
let *?fun2* = λ*y*. *norm* (*f x*) ∗ *Ng y* ∗ *K* + *Nf y* ∗ *norm* (*g y*) ∗ *K* + *KF* ∗ *norm* (*g y* − *g x*) ∗ *K*
let *?F* = *at x within s*

show *?thesis*
proof (*rule has-derivativeI-sandwich*[*of 1*])
show *bounded-linear* (λ*h*. *f x ** *g' h* + *f' h ** *g x*)
by (*intro bounded-linear-add*
bounded-linear-compose [*OF bounded-linear-right*] *bounded-linear-compose* [*OF bounded-linear-left*]
has-derivative-bounded-linear [*OF g*] *has-derivative-bounded-linear* [*OF f*])
next
from *g* **have** (*g* ⟶ *g x*) *?F*
by (*intro continuous-within*[*THEN iffD1*] *has-derivative-continuous*)
moreover from *f g* **have** (*Nf* ⟶ *0*) *?F* (*Ng* ⟶ *0*) *?F*
by (*simp-all add*: *has-derivative-iff-norm Ng-def Nf-def*)
ultimately have (*?fun2* ⟶ *norm* (*f x*) ∗ *0* ∗ *K* + *0* ∗ *norm* (*g x*) ∗ *K* + *KF* ∗ *norm* (*0*::*'b*) ∗ *K*) *?F*
by (*intro tendsto-intros*) (*simp-all add*: *LIM-zero-iff*)
then show (*?fun2* ⟶ *0*) *?F*
by *simp*
next
fix *y* :: *'d*
assume *y* ≠ *x*
have *?fun1 y* =
norm (*f x ** *?D g g' y* + *?D f f' y ** *g y* + *f'* (*y* − *x*) ** (*g y* − *g x*)) / *norm* (*y* − *x*)
by (*simp add*: *diff-left diff-right add-left add-right field-simps*)

also have ... ≤ (*norm* (*f x*) * *norm* (*?D g g' y*) * *K* + *norm* (*?D f f' y*) * *norm* (*g y*) * *K* +
norm (*y* − *x*) * *KF* * *norm* (*g y* − *g x*) * *K*) / *norm* (*y* − *x*)
by (*intro divide-right-mono mult-mono'*
order-trans [*OF norm-triangle-ineq add-mono*]
order-trans [*OF norm-prod mult-right-mono*]
mult-nonneg-nonneg order-refl norm-ge-zero norm-F
K [*THEN order-less-imp-le*])
also have ... = *?fun2 y*
by (*simp add*: *add-divide-distrib Ng-def Nf-def*)
finally show *?fun1 y* ≤ *?fun2 y* **.**
qed *simp*
qed

lemmas *has-derivative-mult*[*simp*, *derivative-intros*] = *bounded-bilinear.FDERIV*[*OF bounded-bilinear-mult*]
lemmas *has-derivative-scaleR*[*simp*, *derivative-intros*] = *bounded-bilinear.FDERIV*[*OF bounded-bilinear-scaleR*]

lemma *has-derivative-prod*[*simp*, *derivative-intros*]:
fixes *f* :: *'i* ⇒ *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-field*
shows (⋀*i*. *i* ∈ *I* ⟹ (*f i has-derivative f' i*) (*at x within S*)) ⟹
((λ*x*. ∏ *i*∈*I*. *f i x*) *has-derivative* (λ*y*. ∑ *i*∈*I*. *f' i y* * (∏ *j*∈*I* − {*i*}. *f j x*))) (*at x within S*)
proof (*induct I rule*: *infinite-finite-induct*)
case *infinite*
then show *?case* **by** *simp*
next
case *empty*
then show *?case* **by** *simp*
next
case (*insert i I*)
let *?P* = λ*y*. *f i x* * (∑ *i*∈*I*. *f' i y* * (∏ *j*∈*I* − {*i*}. *f j x*)) + (*f' i y*) * (∏ *i*∈*I*. *f i x*)
have ((λ*x*. *f i x* * (∏ *i*∈*I*. *f i x*)) *has-derivative ?P*) (*at x within S*)
using *insert* **by** (*intro has-derivative-mult*) *auto*
also have *?P* = (λ*y*. ∑ *i'*∈*insert i I*. *f' i' y* * (∏ *j*∈*insert i I* − {*i'*}. *f j x*))
using *insert*(*1*,*2*)
by (*auto simp add*: *sum-distrib-left insert-Diff-if intro*!: *ext sum.cong*)
finally show *?case*
using *insert* **by** *simp*
qed

lemma *has-derivative-power*[*simp*, *derivative-intros*]:
fixes *f* :: *'a* :: *real-normed-vector* ⇒ *'b* :: *real-normed-field*
assumes *f*: (*f has-derivative f'*) (*at x within S*)
shows ((λ*x*. *f x* ^ *n*) *has-derivative* (λ*y*. *of-nat n* * *f' y* * *f x* ^ (*n* − *1*))) (*at x within S*)
using *has-derivative-prod*[*OF f*, *of* {..< *n*}] **by** (*simp add*: *prod-constant ac-simps*)

lemma *has-derivative-inverse'*:
fixes x :: *'a*::*real-normed-div-algebra*
assumes *x*: $x \neq 0$
shows (*inverse has-derivative* ($\lambda h.\ -\ (inverse\ x * h * inverse\ x)$)) (*at x within S*)
(**is** (- *has-derivative ?f*) -)
proof (*rule has-derivativeI-sandwich*)
show *bounded-linear* ($\lambda h.\ -\ (inverse\ x * h * inverse\ x)$)
by (*simp add*: *bounded-linear-minus bounded-linear-mult-const bounded-linear-mult-right*)
show $0 < norm\ x$ **using** *x* **by** *simp*
have (*inverse* $\longrightarrow$ *inverse x*) (*at x within S*)
using *tendsto-inverse tendsto-ident-at x* **by** *auto*
then show (($\lambda y.\ norm\ (inverse\ y - inverse\ x) * norm\ (inverse\ x)$) $\longrightarrow$ *0*) (*at x within S*)
by (*simp add*: *LIM-zero-iff tendsto-mult-left-zero tendsto-norm-zero*)
next
fix *y* :: *'a*
assume *h*: $y \neq x$ *dist y x* $<$ *norm x*
then have $y \neq 0$ **by** *auto*
have *norm* (*inverse y* $-$ *inverse x* $-$ *?f* (*y* $-x$)) / *norm* (*y* $-$ *x*)
$= norm\ (-\ (inverse\ y * (y - x) * inverse\ x - inverse\ x * (y - x) * inverse\ x)) /$
$norm\ (y - x)$
by (*simp add*: ‹$y \neq 0$› *inverse-diff-inverse x*)
also have ... = *norm* ((*inverse y* $-$ *inverse x*) $*$ (*y* $-$ *x*) $*$ *inverse x*) / *norm* (*y* $-$ *x*)
by (*simp add*: *left-diff-distrib norm-minus-commute*)
also have ... $\leq$ *norm* (*inverse y* $-$ *inverse x*) $*$ *norm* (*y* $-$ *x*) $*$ *norm* (*inverse x*) / *norm* (*y* $-$ *x*)
by (*simp add*: *norm-mult*)
also have ... = *norm* (*inverse y* $-$ *inverse x*) $*$ *norm* (*inverse x*)
by *simp*
finally show *norm* (*inverse y* $-$ *inverse x* $-$ *?f* (*y* $-x$)) / *norm* (*y* $-$ *x*) $\leq$
norm (*inverse y* $-$ *inverse x*) $*$ *norm* (*inverse x*) **.**
qed

lemma *has-derivative-inverse*[*simp*, *derivative-intros*]:
fixes *f* :: - $\Rightarrow$ *'a*::*real-normed-div-algebra*
assumes *x*: $f\ x \neq 0$
and *f*: (*f has-derivative f'*) (*at x within S*)
shows (($\lambda x.\ inverse\ (f\ x)$) *has-derivative* ($\lambda h.\ -\ (inverse\ (f\ x) * f'\ h * inverse\ (f\ x))$))
(*at x within S*)
using *has-derivative-compose*[*OF f has-derivative-inverse'*, *OF x*] **.**

lemma *has-derivative-divide*[*simp*, *derivative-intros*]:
fixes *f* :: - $\Rightarrow$ *'a*::*real-normed-div-algebra*
assumes *f*: (*f has-derivative f'*) (*at x within S*)

and *g*: (*g has-derivative g'*) (*at x within S*)
assumes *x*: *g x* ≠ *0*
shows ((λ*x*. *f x* / *g x*) *has-derivative*
(λ*h*. − *f x* ∗ (*inverse* (*g x*) ∗ *g' h* ∗ *inverse* (*g x*)) + *f' h* / *g x*)) (*at x within S*)
using *has-derivative-mult*[*OF f has-derivative-inverse*[*OF x g*]]
by (*simp add*: *field-simps*)

lemma *has-derivative-power-int'*:
fixes *x* :: *'a*::*real-normed-field*
assumes *x*: *x* ≠ *0*
shows ((λ*x*. *power-int x n*) *has-derivative* (λ*y*. *y* ∗ (*of-int n* ∗ *power-int x* (*n* − *1*)))) (*at x within S*)
proof (*cases n rule*: *int-cases4*)
case (*nonneg n*)
thus *?thesis* **using** *x*
by (*cases n* = *0*) (*auto intro*!: *derivative-eq-intros simp*: *field-simps power-int-diff fun-eq-iff*
simp flip: *power-Suc*)
next
case (*neg n*)
thus *?thesis* **using** *x*
by (*auto intro*!: *derivative-eq-intros simp*: *field-simps power-int-diff power-int-minus*
simp flip: *power-Suc power-Suc2 power-add*)
qed

lemma *has-derivative-power-int*[*simp*, *derivative-intros*]:
fixes *f* :: - ⇒ *'a*::*real-normed-field*
assumes *x*: *f x* ≠ *0*
and *f*: (*f has-derivative f'*) (*at x within S*)
shows ((λ*x*. *power-int* (*f x*) *n*) *has-derivative* (λ*h*. *f' h* ∗ (*of-int n* ∗ *power-int* (*f x*) (*n* − *1*))))
(*at x within S*)
using *has-derivative-compose*[*OF f has-derivative-power-int'*, *OF x*] **.**

Conventional form requires mult-AC laws. Types real and complex only.

lemma *has-derivative-divide'*[*derivative-intros*]:
fixes *f* :: - ⇒ *'a*::*real-normed-field*
assumes *f*: (*f has-derivative f'*) (*at x within S*)
and *g*: (*g has-derivative g'*) (*at x within S*)
and *x*: *g x* ≠ *0*
shows ((λ*x*. *f x* / *g x*) *has-derivative* (λ*h*. (*f' h* ∗ *g x* − *f x* ∗ *g' h*) / (*g x* ∗ *g x*))) (*at x within S*)
proof −
have *f' h* / *g x* − *f x* ∗ (*inverse* (*g x*) ∗ *g' h* ∗ *inverse* (*g x*)) =
(*f' h* ∗ *g x* − *f x* ∗ *g' h*) / (*g x* ∗ *g x*) **for** *h*
by (*simp add*: *field-simps x*)
then show *?thesis*
using *has-derivative-divide* [*OF f g*] *x*

by *simp*
qed

110.4 Uniqueness

This can not generally shown for (*has-derivative*), as we need to approach the point from all directions. There is a proof in *Analysis* for *euclidean-space*.

lemma *has-derivative-at2*: (*f has-derivative f'*) (*at x*) ⟷
bounded-linear f' ∧ ((λ*y*. (*1* / (*norm*(*y* − *x*))) ∗$_R$ (*f y* − (*f x* + *f'* (*y* − *x*)))) ⟶ *0*) (*at x*)
using *has-derivative-within* [*of f f' x UNIV*]
by *simp*

lemma *has-derivative-zero-unique*:
assumes ((λ*x*. *0*) *has-derivative F*) (*at x*)
shows *F* = (λ*h*. *0*)
proof −
interpret *F*: *bounded-linear F*
using *assms* **by** (*rule has-derivative-bounded-linear*)
let *?r* = λ*h*. *norm* (*F h*) / *norm h*
have ∗: *?r* −*0*→ *0*
using *assms* **unfolding** *has-derivative-at* **by** *simp*
show *F* = (λ*h*. *0*)
proof
show *F h* = *0* **for** *h*
proof (*rule ccontr*)
assume ∗∗: ¬ *?thesis*
then have *h*: *h* ≠ *0*
by (*auto simp add*: *F.zero*)
with ∗∗ **have** *0* < *?r h*
by *simp*
from *LIM-D* [*OF* ∗ *this*] **obtain** *S*
where *S*: *0* < *S* **and** *r*: ⋀*x*. *x* ≠ *0* ⟹ *norm x* < *S* ⟹ *?r x* < *?r h*
by *auto*
from *dense* [*OF S*] **obtain** *t* **where** *t*: *0* < *t* ∧ *t* < *S* **..**
let *?x* = *scaleR* (*t* / *norm h*) *h*
have *?x* ≠ *0* **and** *norm ?x* < *S*
using *t h* **by** *simp-all*
then have *?r ?x* < *?r h*
by (*rule r*)
then show *False*
using *t h* **by** (*simp add*: *F.scaleR*)
qed
qed
qed

lemma *has-derivative-unique*:
assumes (*f has-derivative F*) (*at x*)
and (*f has-derivative F'*) (*at x*)

shows $F = F'$
proof −
have $((\lambda x.\ 0)$ *has-derivative* $(\lambda h.\ F\ h - F'\ h))$ $(at\ x)$
using *has-derivative-diff* [*OF assms*] **by** *simp*
then have $(\lambda h.\ F\ h - F'\ h) = (\lambda h.\ 0)$
by (*rule has-derivative-zero-unique*)
then show $F = F'$
unfolding *fun-eq-iff right-minus-eq* **.**
qed

lemma *has-derivative-Uniq*: $\exists_{\leq 1} F.$ $(f$ *has-derivative* $F)$ $(at\ x)$
by (*simp add*: *Uniq-def has-derivative-unique*)

110.5 Differentiability predicate

definition *differentiable* :: ('*a*::*real-normed-vector* ⇒ '*b*::*real-normed-vector*) ⇒ '*a filter* ⇒ *bool*
(**infix** ‹*differentiable*› *50*)
where *f differentiable F* ⟷ (∃ *D*. (*f has-derivative D*) *F*)

lemma *differentiable-subset*:
f differentiable (*at x within s*) ⟹ *t* ⊆ *s* ⟹ *f differentiable* (*at x within t*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-subset*)

lemmas *differentiable-within-subset* = *differentiable-subset*

lemma *differentiable-ident* [*simp*, *derivative-intros*]: $(\lambda x.\ x)$ *differentiable F*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-ident*)

lemma *differentiable-const* [*simp*, *derivative-intros*]: $(\lambda z.\ a)$ *differentiable F*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-const*)

lemma *differentiable-in-compose*:
f differentiable (*at* (*g x*) *within* (*g*‘*s*)) ⟹ *g differentiable* (*at x within s*) ⟹
$(\lambda x.\ f\ (g\ x))$ *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-in-compose*)

lemma *differentiable-compose*:
f differentiable (*at* (*g x*)) ⟹ *g differentiable* (*at x within s*) ⟹
$(\lambda x.\ f\ (g\ x))$ *differentiable* (*at x within s*)
by (*blast intro*: *differentiable-in-compose differentiable-subset*)

lemma *differentiable-add* [*simp*, *derivative-intros*]:
f differentiable F ⟹ *g differentiable F* ⟹ $(\lambda x.\ f\ x + g\ x)$ *differentiable F*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-add*)

lemma *differentiable-sum*[*simp*, *derivative-intros*]:
assumes *finite s* ∀ *a*∈*s*. (*f a*) *differentiable net*
shows $(\lambda x.\ sum\ (\lambda a.\ f\ a\ x)\ s)$ *differentiable net*

proof −
from *bchoice*[*OF assms*(*2*)[*unfolded differentiable-def*]]
show *?thesis*
by (*auto intro*!: *has-derivative-sum simp*: *differentiable-def*)
qed

lemma *differentiable-minus* [*simp*, *derivative-intros*]:
f differentiable F ⟹ (λ*x*. − *f x*) *differentiable F*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-minus*)

lemma *differentiable-diff* [*simp*, *derivative-intros*]:
f differentiable F ⟹ *g differentiable F* ⟹ (λ*x*. *f x* − *g x*) *differentiable F*
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-diff*)

lemma *differentiable-mult* [*simp*, *derivative-intros*]:
fixes *f g* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-algebra*
shows *f differentiable* (*at x within s*) ⟹ *g differentiable* (*at x within s*) ⟹
(λ*x*. *f x* ∗ *g x*) *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-mult*)

lemma *differentiable-cmult-left-iff* [*simp*]:
fixes *c*::*'a*::*real-normed-field*
shows (λ*t*. *c* ∗ *q t*) *differentiable at t* ⟷ *c* = *0* ∨ (λ*t*. *q t*) *differentiable at t*
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
{assume *c* ≠ *0*
then have *q differentiable at t*
using *differentiable-mult* [*OF differentiable-const L*, *of concl*: *1* / *c*] **by** *auto*
} then show *?rhs*
by *auto*
qed *auto*

lemma *differentiable-cmult-right-iff* [*simp*]:
fixes *c*::*'a*::*real-normed-field*
shows (λ*t*. *q t* ∗ *c*) *differentiable at t* ⟷ *c* = *0* ∨ (λ*t*. *q t*) *differentiable at t*
(**is** *?lhs* = *?rhs*)
by (*simp add*: *mult.commute flip*: *differentiable-cmult-left-iff*)

lemma *differentiable-inverse* [*simp*, *derivative-intros*]:
fixes *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-field*
shows *f differentiable* (*at x within s*) ⟹ *f x* ≠ *0* ⟹
(λ*x*. *inverse* (*f x*)) *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-inverse*)

lemma *differentiable-divide* [*simp*, *derivative-intros*]:
fixes *f g* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-field*
shows *f differentiable* (*at x within s*) ⟹ *g differentiable* (*at x within s*) ⟹
g x ≠ *0* ⟹ (λ*x*. *f x* / *g x*) *differentiable* (*at x within s*)

unfolding *divide-inverse* **by** *simp*

lemma *differentiable-power* [*simp*, *derivative-intros*]:
fixes *f g* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-field*
shows *f differentiable* (*at x within s*) ⟹ (λ*x*. *f x* ^ *n*) *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-power*)

lemma *differentiable-power-int* [*simp*, *derivative-intros*]:
fixes *f* :: *'a*::*real-normed-vector* ⇒ *'b*::*real-normed-field*
shows *f differentiable* (*at x within s*) ⟹ *f x* ≠ *0* ⟹
(λ*x*. *power-int* (*f x*) *n*) *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-power-int*)

lemma *differentiable-scaleR* [*simp*, *derivative-intros*]:
f differentiable (*at x within s*) ⟹ *g differentiable* (*at x within s*) ⟹
(λ*x*. *f x* ∗*R* *g x*) *differentiable* (*at x within s*)
unfolding *differentiable-def* **by** (*blast intro*: *has-derivative-scaleR*)

lemma *has-derivative-imp-has-field-derivative*:
(*f has-derivative D*) *F* ⟹ (⋀*x*. *x* ∗ *D'* = *D x*) ⟹ (*f has-field-derivative D'*) *F*
unfolding *has-field-derivative-def*
by (*rule has-derivative-eq-rhs*[*of f D*]) (*simp-all add*: *fun-eq-iff mult.commute*)

lemma *has-field-derivative-imp-has-derivative*:
(*f has-field-derivative D*) *F* ⟹ (*f has-derivative* (∗) *D*) *F*
by (*simp add*: *has-field-derivative-def*)

lemma *DERIV-subset*:
(*f has-field-derivative f'*) (*at x within s*) ⟹ *t* ⊆ *s* ⟹
(*f has-field-derivative f'*) (*at x within t*)
by (*simp add*: *has-field-derivative-def has-derivative-subset*)

lemma *has-field-derivative-at-within*:
(*f has-field-derivative f'*) (*at x*) ⟹ (*f has-field-derivative f'*) (*at x within s*)
using *DERIV-subset* **by** *blast*

abbreviation (*input*)
DERIV :: (*'a*::*real-normed-field* ⇒ *'a*) ⇒ *'a* ⇒ *'a* ⇒ *bool*
(‹(‹*notation*=‹*mixfix DERIV*››*DERIV* (-)/ (-)/ :> (-))› [*1000*, *1000*, *60*] *60*)
where *DERIV f x* :> *D* ≡ (*f has-field-derivative D*) (*at x*)

abbreviation *has-real-derivative* :: (*real* ⇒ *real*) ⇒ *real* ⇒ *real filter* ⇒ *bool*
(**infix** ‹(*has'-real'-derivative*)› *50*)
where (*f has-real-derivative D*) *F* ≡ (*f has-field-derivative D*) *F*

lemma *real-differentiable-def*:
f differentiable at x within s ⟷ (∃ *D*. (*f has-real-derivative D*) (*at x within s*))
proof *safe*

assume *f differentiable at x within s*
then obtain *f'* **where** ∗: (*f has-derivative f'*) (*at x within s*)
unfolding *differentiable-def* **by** *auto*
then obtain *c* **where** *f' = ((∗) c)*
by (*metis real-bounded-linear has-derivative-bounded-linear mult.commute fun-eq-iff*)
with ∗ **show** ∃ *D*. (*f has-real-derivative D*) (*at x within s*)
unfolding *has-field-derivative-def* **by** *auto*
qed (*auto simp*: *differentiable-def has-field-derivative-def*)

lemma *real-differentiableE* [*elim?*]:
assumes *f*: *f differentiable* (*at x within s*)
obtains *df* **where** (*f has-real-derivative df*) (*at x within s*)
using *assms* **by** (*auto simp*: *real-differentiable-def*)

lemma *has-field-derivative-iff*:
(*f has-field-derivative D*) (*at x within S*) ⟷
((λ*y*. (*f y − f x*) / (*y − x*)) ⟶ *D*) (*at x within S*)
proof −
have ((λ*y*. *norm* (*f y − f x − D ∗* (*y − x*)) / *norm* (*y − x*)) ⟶ *0*) (*at x within S*)
= ((λ*y*. (*f y − f x*) / (*y − x*) *− D*) ⟶ *0*) (*at x within S*)
by (*smt* (*verit*, *best*) *Lim-cong-within divide-diff-eq-iff norm-divide right-minus-eq tendsto-norm-zero-iff*)
then show *?thesis*
by (*simp add*: *has-field-derivative-def has-derivative-iff-norm bounded-linear-mult-right LIM-zero-iff*)
qed

lemma *DERIV-def*: *DERIV f x :> D* ⟷ (λ*h*. (*f* (*x* + *h*) − *f x*) / *h*) −*0*→ *D*
unfolding *field-has-derivative-at has-field-derivative-def has-field-derivative-iff* **..**

lemma *has-field-derivative-unique*:
assumes (*f has-field-derivative f'1*) (*at x within A*)
assumes (*f has-field-derivative f'2*) (*at x within A*)
assumes *at x within A ≠ bot*
shows *f'1 = f'2*
using *assms* **unfolding** *has-field-derivative-iff* **using** *tendsto-unique* **by** *blast*

due to Christian Pardillo Laursen, replacing a proper epsilon-delta horror

lemma *field-derivative-lim-unique*:
assumes *f*: (*f has-field-derivative df*) (*at z*)
and *s*: *s* ⟶ *0* ⋀*n*. *s n ≠ 0*
and *a*: (λ*n*. (*f* (*z* + *s n*) − *f z*) / *s n*) ⟶ *a*
shows *df = a*
proof −
have ((λ*k*. (*f* (*z* + *k*) − *f z*) / *k*) ⟶ *df*) (*at 0*)
using *f* **by** (*simp add*: *DERIV-def*)
with *s* **have** ((λ*n*. (*f* (*z* + *s n*) − *f z*) / *s n*) ⟶ *df*)
by (*simp flip*: *LIMSEQ-SEQ-conv*)

then show *?thesis*
using *a* **by** (*rule LIMSEQ-unique*)
qed

lemma *mult-commute-abs*: $(\lambda x.\ x * c) = (*)\ c$
for $c :: 'a$::*ab-semigroup-mult*
by (*simp add*: *fun-eq-iff mult.commute*)

lemma *DERIV-compose-FDERIV*:
fixes *f*::*real*⇒*real*
assumes *DERIV f* (*g x*) :> *f'*
assumes (*g has-derivative g'*) (*at x within s*)
shows ((λ*x*. *f* (*g x*)) *has-derivative* (λ*x*. *g' x* ∗ *f'*)) (*at x within s*)
using *assms has-derivative-compose*[*of g g' x s f* (∗) *f'*]
by (*auto simp*: *has-field-derivative-def ac-simps*)

110.6 Vector derivative

It's for real derivatives only, and not obviously generalisable to field derivatives

lemma *has-real-derivative-iff-has-vector-derivative*:
(*f has-real-derivative y*) *F* ⟷ (*f has-vector-derivative y*) *F*
unfolding *has-vector-derivative-def has-field-derivative-def real-scaleR-def mult-commute-abs*
..

lemma *has-field-derivative-subset*:
(*f has-field-derivative y*) (*at x within s*) ⟹ *t* ⊆ *s* ⟹
(*f has-field-derivative y*) (*at x within t*)
by (*fact DERIV-subset*)

lemma *has-vector-derivative-const*[*simp*, *derivative-intros*]: ((λ*x*. *c*) *has-vector-derivative 0*) *net*
by (*auto simp*: *has-vector-derivative-def*)

lemma *has-vector-derivative-id*[*simp*, *derivative-intros*]: ((λ*x*. *x*) *has-vector-derivative 1*) *net*
by (*auto simp*: *has-vector-derivative-def*)

lemma *has-vector-derivative-minus*[*derivative-intros*]:
(*f has-vector-derivative f'*) *net* ⟹ ((λ*x*. − *f x*) *has-vector-derivative* (− *f'*)) *net*
by (*auto simp*: *has-vector-derivative-def*)

lemma *has-vector-derivative-add*[*derivative-intros*]:
(*f has-vector-derivative f'*) *net* ⟹ (*g has-vector-derivative g'*) *net* ⟹
((λ*x*. *f x* + *g x*) *has-vector-derivative* (*f'* + *g'*)) *net*
by (*auto simp*: *has-vector-derivative-def scaleR-right-distrib*)

lemma *has-vector-derivative-sum*[*derivative-intros*]:
(⋀*i*. *i* ∈ *I* ⟹ (*f i has-vector-derivative f' i*) *net*) ⟹

((λx. ∑ i∈I. f i x) *has-vector-derivative* (∑ i∈I. f′ i)) *net*
by (*auto simp*: *has-vector-derivative-def fun-eq-iff scaleR-sum-right intro*!: *derivative-eq-intros*)

lemma *has-vector-derivative-diff*[*derivative-intros*]:
(*f has-vector-derivative f′*) *net* ⟹ (*g has-vector-derivative g′*) *net* ⟹
((λx. f x − g x) *has-vector-derivative* (f′ − g′)) *net*
by (*auto simp*: *has-vector-derivative-def scaleR-diff-right*)

lemma *has-vector-derivative-add-const*:
((λt. g t + z) *has-vector-derivative f′*) *net* = ((λt. g t) *has-vector-derivative f′*) *net*
apply (*intro iffI*)
apply (*force dest*: *has-vector-derivative-diff* [**where** *g* = λt. *z*, *OF* - *has-vector-derivative-const*])
apply (*force dest*: *has-vector-derivative-add* [*OF* - *has-vector-derivative-const*])
done

lemma *has-vector-derivative-diff-const*:
((λt. g t − z) *has-vector-derivative f′*) *net* = ((λt. g t) *has-vector-derivative f′*) *net*
using *has-vector-derivative-add-const* [**where** *z* = −*z*]
by *simp*

lemma (**in** *bounded-linear*) *has-vector-derivative*:
assumes (*g has-vector-derivative g′*) *F*
shows ((λx. f (g x)) *has-vector-derivative* f g′) *F*
using *has-derivative*[*OF assms*[*unfolded has-vector-derivative-def*]]
by (*simp add*: *has-vector-derivative-def scaleR*)

lemma (**in** *bounded-bilinear*) *has-vector-derivative*:
assumes (*f has-vector-derivative f′*) (*at x within s*)
and (*g has-vector-derivative g′*) (*at x within s*)
shows ((λx. f x ** g x) *has-vector-derivative* (f x ** g′ + f′ ** g x)) (*at x within s*)
using *FDERIV*[*OF assms*(*1*−*2*)[*unfolded has-vector-derivative-def*]]
by (*simp add*: *has-vector-derivative-def scaleR-right scaleR-left scaleR-right-distrib*)

lemma *has-vector-derivative-scaleR*[*derivative-intros*]:
(*f has-field-derivative f′*) (*at x within s*) ⟹ (*g has-vector-derivative g′*) (*at x within s*) ⟹
((λx. f x *R g x) *has-vector-derivative* (f x *R g′ + f′ *R g x)) (*at x within s*)
unfolding *has-real-derivative-iff-has-vector-derivative*
by (*rule bounded-bilinear.has-vector-derivative*[*OF bounded-bilinear-scaleR*])

lemma *has-vector-derivative-mult*[*derivative-intros*]:
(*f has-vector-derivative f′*) (*at x within s*) ⟹ (*g has-vector-derivative g′*) (*at x within s*) ⟹
((λx. f x * g x) *has-vector-derivative* (f x * g′ + f′ * g x)) (*at x within s*)
for *f g* :: *real* ⇒ *'a*::*real-normed-algebra*

by (*rule bounded-bilinear.has-vector-derivative*[*OF bounded-bilinear-mult*])

lemma *has-vector-derivative-of-real*[*derivative-intros*]:
(*f has-field-derivative D*) *F* ⟹ ((λ*x*. *of-real* (*f x*)) *has-vector-derivative* (*of-real D*)) *F*
by (*rule bounded-linear.has-vector-derivative*[*OF bounded-linear-of-real*])
(*simp add*: *has-real-derivative-iff-has-vector-derivative*)

lemma *has-vector-derivative-real-field*:
(*f has-field-derivative f'*) (*at* (*of-real a*)) ⟹ ((λ*x*. *f* (*of-real x*)) *has-vector-derivative f'*) (*at a within s*)
using *has-derivative-compose*[*of of-real of-real a - f* (∗) *f'*]
by (*simp add*: *scaleR-conv-of-real ac-simps has-vector-derivative-def has-field-derivative-def*)

lemma *has-vector-derivative-continuous*:
(*f has-vector-derivative D*) (*at x within s*) ⟹ *continuous* (*at x within s*) *f*
by (*auto intro*: *has-derivative-continuous simp*: *has-vector-derivative-def*)

lemma *continuous-on-vector-derivative*:
(⋀*x*. *x* ∈ *S* ⟹ (*f has-vector-derivative f' x*) (*at x within S*)) ⟹ *continuous-on S f*
by (*auto simp*: *continuous-on-eq-continuous-within intro*!: *has-vector-derivative-continuous*)

lemma *has-vector-derivative-mult-right*[*derivative-intros*]:
fixes *a* :: *'a*::*real-normed-algebra*
shows (*f has-vector-derivative x*) *F* ⟹ ((λ*x*. *a* ∗ *f x*) *has-vector-derivative* (*a* ∗ *x*)) *F*
by (*rule bounded-linear.has-vector-derivative*[*OF bounded-linear-mult-right*])

lemma *has-vector-derivative-mult-left*[*derivative-intros*]:
fixes *a* :: *'a*::*real-normed-algebra*
shows (*f has-vector-derivative x*) *F* ⟹ ((λ*x*. *f x* ∗ *a*) *has-vector-derivative* (*x* ∗ *a*)) *F*
by (*rule bounded-linear.has-vector-derivative*[*OF bounded-linear-mult-left*])

lemma *has-vector-derivative-divide*[*derivative-intros*]:
fixes *a* :: *'a*::*real-normed-field*
shows (*f has-vector-derivative x*) *F* ⟹ ((λ*x*. *f x* / *a*) *has-vector-derivative* (*x* / *a*)) *F*
using *has-vector-derivative-mult-left* [*of f x F inverse a*]
by (*simp add*: *field-class.field-divide-inverse*)

lemma *has-vector-derivative-within-1D*:
fixes *f* :: *real* ⇒ *'a*::*real-normed-vector*
shows (*f has-vector-derivative f'*) (*at x within S*) ⟷
((λ*y*. (*f y* − *f x*) /$_R$ (*y* − *x*)) ⟶ *f'*) (*at x within S*)
proof −
have *ev-eq*: ∀$_F$ *y in at x within S*. (*f y* − *f x*) /$_R$ (*y* − *x*) − *f'* = (*f y* − *f x* − (*y* − *x*) ∗$_R$ *f'*) /$_R$ (*y* − *x*)

unfolding *eventually-at-filter* **by** (*simp add*: *scaleR-diff-right scaleR-scaleR*)
show *?thesis*
proof
assume (*f has-vector-derivative f'*) (*at x within S*)
then have *Zfun* (λ*y*. (*f y* − *f x* − (*y* − *x*) ∗$_R$ *f'*) /$_R$ |*y* − *x*|) (*at x within S*)
unfolding *has-vector-derivative-def has-derivative-at-within tendsto-Zfun-iff*
by *auto*
then have *Zfun* (λ*y*. *norm* ((*f y* − *f x* − (*y* − *x*) ∗$_R$ *f'*) /$_R$ |*y* − *x*|)) (*at x within S*)
using *Zfun-norm-iff* **by** *fastforce*
then show ((λ*y*. (*f y* − *f x*) /$_R$ (*y* − *x*)) ⟶ *f'*) (*at x within S*)
using *Zfun-norm-iff Zfun-ssubst ev-eq tendsto-Zfun-iff* **by** *fastforce*
next
assume *R*: ((λ*y*. (*f y* − *f x*) /$_R$ (*y* − *x*)) ⟶ *f'*) (*at x within S*)
have *Zfun* (λ*y*. (*f y* − *f x*) /$_R$ (*y* − *x*) − *f'*) (*at x within S*)
using *R* **by** (*simp add*: *tendsto-Zfun-iff*)
then have *Zfun* (λ*y*. (*f y* − *f x* − (*y* − *x*) ∗$_R$ *f'*) /$_R$ (*y* − *x*)) (*at x within S*)
by (*smt* (*verit*, *del-insts*) *Zfun-ssubst ev-eq eventually-mono*)
then have *Zfun* (λ*y*. (*f y* − *f x* − (*y* − *x*) ∗$_R$ *f'*) /$_R$ |*y* − *x*|) (*at x within S*)
using *Zfun-norm-iff* **by** (*fastforce simp add*: *Zfun-le*)
then show (*f has-vector-derivative f'*) (*at x within S*)
unfolding *has-vector-derivative-def has-derivative-at-within tendsto-Zfun-iff*
using *bounded-linear-scaleR-left* **by** *auto*
qed
qed

lemma *norm-vector-derivatives-le-within*:
fixes *f* :: *real* ⇒ *'a*::*real-normed-vector* **and** *g* :: *real* ⇒ *'b*::*real-normed-vector*
assumes *fderiv*: (*f has-vector-derivative f'*) (*at x within S*)
and *gderiv*: (*g has-vector-derivative g'*) (*at x within S*)
and *ev*: *eventually* (λ*y*. *norm* (*f y* − *f x*) ≤ *norm* (*g y* − *g x*)) (*at x within S*)
and *nontrivial*: *at x within S* ≠ *bot*
shows *norm f'* ≤ *norm g'*
proof (*rule tendsto-le* [*OF nontrivial*])
let *?f* = λ*y*. *norm*(*inverse*(*y* − *x*) ∗$_R$ (*f y* − *f x*))
let *?g* = λ*y*. *norm*(*inverse*(*y* − *x*) ∗$_R$ (*g y* − *g x*))
show (*?f* ⟶ *norm f'*) (*at x within S*)
(*?g* ⟶ *norm g'*) (*at x within S*)
using *fderiv gderiv has-vector-derivative-within-1D tendsto-norm* **by** *blast*+
show ∀$_F$ *x in at x within S*. *?f x* ≤ *?g x*
using *eventually-mono* [*OF ev*] **by** (*simp add*: *mult-left-mono*)
qed

110.7 Derivatives

lemma *DERIV-D*: *DERIV f x* :> *D* ⟹ (λ*h*. (*f* (*x* + *h*) − *f x*) / *h*) −*0*→ *D*
by (*simp add*: *DERIV-def*)

lemma *has-field-derivativeD*:

(*f has-field-derivative D*) (*at x within S*) ⟹
((λ*y*. (*f y* − *f x*) / (*y* − *x*)) ⟶ *D*) (*at x within S*)
by (*simp add*: *has-field-derivative-iff*)

lemma *DERIV-const* [*simp*, *derivative-intros*]: ((λ*x*. *k*) *has-field-derivative 0*) *F*
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-const*]) *auto*

lemma *DERIV-ident* [*simp*, *derivative-intros*]: ((λ*x*. *x*) *has-field-derivative 1*) *F*
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-ident*]) *auto*

lemma *field-differentiable-add*[*derivative-intros*]:
(*f has-field-derivative f'*) *F* ⟹ (*g has-field-derivative g'*) *F* ⟹
((λ*z*. *f z* + *g z*) *has-field-derivative f'* + *g'*) *F*
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-add*])
(*auto simp*: *has-field-derivative-def field-simps mult-commute-abs*)

corollary *DERIV-add*:
(*f has-field-derivative D*) (*at x within s*) ⟹ (*g has-field-derivative E*) (*at x within s*) ⟹
((λ*x*. *f x* + *g x*) *has-field-derivative D* + *E*) (*at x within s*)
by (*rule field-differentiable-add*)

lemma *field-differentiable-minus*[*derivative-intros*]:
(*f has-field-derivative f'*) *F* ⟹ ((λ*z*. − (*f z*)) *has-field-derivative* −*f'*) *F*
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-minus*])
(*auto simp*: *has-field-derivative-def field-simps mult-commute-abs*)

corollary *DERIV-minus*:
(*f has-field-derivative D*) (*at x within s*) ⟹
((λ*x*. − *f x*) *has-field-derivative* −*D*) (*at x within s*)
by (*rule field-differentiable-minus*)

lemma *field-differentiable-diff*[*derivative-intros*]:
(*f has-field-derivative f'*) *F* ⟹
(*g has-field-derivative g'*) *F* ⟹ ((λ*z*. *f z* − *g z*) *has-field-derivative f'* − *g'*) *F*
by (*simp only*: *diff-conv-add-uminus field-differentiable-add field-differentiable-minus*)

corollary *DERIV-diff*:
(*f has-field-derivative D*) (*at x within s*) ⟹
(*g has-field-derivative E*) (*at x within s*) ⟹
((λ*x*. *f x* − *g x*) *has-field-derivative D* − *E*) (*at x within s*)
by (*rule field-differentiable-diff*)

lemma *DERIV-continuous*: (*f has-field-derivative D*) (*at x within s*) ⟹ *continuous* (*at x within s*) *f*
by (*drule has-derivative-continuous*[*OF has-field-derivative-imp-has-derivative*])
simp

corollary *DERIV-isCont*: *DERIV f x* :> *D* ⟹ *isCont f x*

by (*rule DERIV-continuous*)

lemma *DERIV-atLeastAtMost-imp-continuous-on*:
assumes $\bigwedge x.\ [\![a \le x;\ x \le b]\!] \Longrightarrow \exists\, y.\ DERIV\ f\ x :> y$
shows *continuous-on* $\{a..b\}$ *f*
by (*meson DERIV-isCont assms atLeastAtMost-iff continuous-at-imp-continuous-at-within continuous-on-eq-continuous-within*)

lemma *DERIV-continuous-on*:
$(\bigwedge x.\ x \in s \Longrightarrow (f\ has\text{-}field\text{-}derivative\ (D\ x))\ (at\ x\ within\ s)) \Longrightarrow continuous\text{-}on\ s\ f$
unfolding *continuous-on-eq-continuous-within*
by (*intro continuous-at-imp-continuous-on ballI DERIV-continuous*)

lemma *DERIV-mult'*:
$(f\ has\text{-}field\text{-}derivative\ D)\ (at\ x\ within\ s) \Longrightarrow (g\ has\text{-}field\text{-}derivative\ E)\ (at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * g\ x)\ has\text{-}field\text{-}derivative\ f\ x * E + D * g\ x)\ (at\ x\ within\ s)$
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-mult*])
(*auto simp*: *field-simps mult-commute-abs dest*: *has-field-derivative-imp-has-derivative*)

lemma *DERIV-mult*[*derivative-intros*]:
$(f\ has\text{-}field\text{-}derivative\ Da)\ (at\ x\ within\ s) \Longrightarrow (g\ has\text{-}field\text{-}derivative\ Db)\ (at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * g\ x)\ has\text{-}field\text{-}derivative\ Da * g\ x + Db * f\ x)\ (at\ x\ within\ s)$
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-mult*])
(*auto simp*: *field-simps dest*: *has-field-derivative-imp-has-derivative*)

Derivative of linear multiplication

lemma *DERIV-cmult*:
$(f\ has\text{-}field\text{-}derivative\ D)\ (at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ c * f\ x)\ has\text{-}field\text{-}derivative\ c * D)\ (at\ x\ within\ s)$
by (*drule DERIV-mult'* [*OF DERIV-const*]) *simp*

lemma *DERIV-cmult-right*:
$(f\ has\text{-}field\text{-}derivative\ D)\ (at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x * c)\ has\text{-}field\text{-}derivative\ D * c)\ (at\ x\ within\ s)$
using *DERIV-cmult* **by** (*auto simp add*: *ac-simps*)

lemma *DERIV-cmult-Id* [*simp*]: $((*)\ c\ has\text{-}field\text{-}derivative\ c)\ (at\ x\ within\ s)$
using *DERIV-ident* [*THEN DERIV-cmult*, **where** $c = c$ **and** $x = x$] **by** *simp*

lemma *DERIV-cdivide*:
$(f\ has\text{-}field\text{-}derivative\ D)\ (at\ x\ within\ s) \Longrightarrow$
$((\lambda x.\ f\ x\ /\ c)\ has\text{-}field\text{-}derivative\ D\ /\ c)\ (at\ x\ within\ s)$
using *DERIV-cmult-right*[*of f D x s 1 / c*] **by** *simp*

lemma *DERIV-unique*: $DERIV\ f\ x :> D \Longrightarrow DERIV\ f\ x :> E \Longrightarrow D = E$
unfolding *DERIV-def* **by** (*rule LIM-unique*)

lemma *DERIV-Uniq*: $\exists_{\leq 1} D.\ DERIV\ f\ x :> D$
by (*simp add*: *DERIV-unique Uniq-def*)

lemma *DERIV-sum*[*derivative-intros*]:
$(\bigwedge n.\ n \in S \Longrightarrow ((\lambda x.\ f\ x\ n)\ has\text{-}field\text{-}derivative\ (f'\ n))\ F) \Longrightarrow$
$((\lambda x.\ sum\ (f\ x)\ S)\ has\text{-}field\text{-}derivative\ sum\ f'\ S)\ F$
by (*rule has-derivative-imp-has-field-derivative* [*OF has-derivative-sum*])
(*auto simp*: *sum-distrib-left mult-commute-abs dest*: *has-field-derivative-imp-has-derivative*)

lemma *DERIV-sum-list*[*derivative-intros*]:
$(\bigwedge n.\ n \in set\ xs \Longrightarrow ((\lambda x.\ f\ x\ n)\ has\text{-}field\text{-}derivative\ (f'\ n))\ F) \Longrightarrow$
$((\lambda x.\ sum\text{-}list\ (map\ (f\ x)\ xs))\ has\text{-}field\text{-}derivative\ sum\text{-}list\ (map\ f'\ xs))\ F$
by (*induction xs*) (*auto intro*!: *derivative-intros*)

lemma *DERIV-inverse'*[*derivative-intros*]:
assumes ($f\ has\text{-}field\text{-}derivative\ D$) ($at\ x\ within\ s$)
and $f\ x \neq 0$
shows $((\lambda x.\ inverse\ (f\ x))\ has\text{-}field\text{-}derivative\ -\ (inverse\ (f\ x) * D * inverse\ (f\ x)))$
($at\ x\ within\ s$)
proof −
have ($f\ has\text{-}derivative\ (\lambda x.\ x * D)$) = ($f\ has\text{-}derivative\ (*)\ D$)
by (*rule arg-cong* [*of* $\lambda x.\ x * D$]) (*simp add*: *fun-eq-iff*)
with *assms* **have** ($f\ has\text{-}derivative\ (\lambda x.\ x * D)$) ($at\ x\ within\ s$)
by (*auto dest*!: *has-field-derivative-imp-has-derivative*)
then show *?thesis* **using** ‹$f\ x \neq 0$›
by (*auto intro*: *has-derivative-imp-has-field-derivative has-derivative-inverse*)
qed

Power of -1

lemma *DERIV-inverse*:
$x \neq 0 \Longrightarrow ((\lambda x.\ inverse(x))\ has\text{-}field\text{-}derivative\ -\ (inverse\ x\ \hat{}\ Suc\ (Suc\ 0)))\ (at\ x\ within\ s)$
by (*drule DERIV-inverse'* [*OF DERIV-ident*]) *simp*

Derivative of inverse

lemma *DERIV-inverse-fun*:
($f\ has\text{-}field\text{-}derivative\ d$) ($at\ x\ within\ s$) $\Longrightarrow f\ x \neq 0 \Longrightarrow$
$((\lambda x.\ inverse\ (f\ x))\ has\text{-}field\text{-}derivative\ (-\ (d * inverse(f\ x\ \hat{}\ Suc\ (Suc\ 0)))))$
($at\ x\ within\ s$)
by (*drule* (*1*) *DERIV-inverse'*) (*simp add*: *ac-simps nonzero-inverse-mult-distrib*)

Derivative of quotient

lemma *DERIV-divide*[*derivative-intros*]:
($f\ has\text{-}field\text{-}derivative\ D$) ($at\ x\ within\ s$) $\Longrightarrow$
($g\ has\text{-}field\text{-}derivative\ E$) ($at\ x\ within\ s$) $\Longrightarrow g\ x \neq 0 \Longrightarrow$
$((\lambda x.\ f\ x\ /\ g\ x)\ has\text{-}field\text{-}derivative\ (D * g\ x - f\ x * E)\ /\ (g\ x * g\ x))\ (at\ x\ within\ s)$

by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-divide*])
(*auto dest*: *has-field-derivative-imp-has-derivative simp*: *field-simps*)

lemma *DERIV-quotient*:
(*f has-field-derivative d*) (*at x within s*) ⟹
(*g has-field-derivative e*) (*at x within s*)⟹ *g x* ≠ *0* ⟹
((λ*y*. *f y* / *g y*) *has-field-derivative* (*d* ∗ *g x* − (*e* ∗ *f x*)) / (*g x* ^ *Suc* (*Suc 0*)))
(*at x within s*)
by (*drule* (*2*) *DERIV-divide*) (*simp add*: *mult.commute*)

lemma *DERIV-power-Suc*:
(*f has-field-derivative D*) (*at x within s*) ⟹
((λ*x*. *f x* ^ *Suc n*) *has-field-derivative* (*1* + *of-nat n*) ∗ (*D* ∗ *f x* ^ *n*)) (*at x*
within s)
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-power*])
(*auto simp*: *has-field-derivative-def*)

lemma *DERIV-power*[*derivative-intros*]:
(*f has-field-derivative D*) (*at x within s*) ⟹
((λ*x*. *f x* ^ *n*) *has-field-derivative of-nat n* ∗ (*D* ∗ *f x* ^ (*n* − *Suc 0*))) (*at x*
within s)
by (*rule has-derivative-imp-has-field-derivative*[*OF has-derivative-power*])
(*auto simp*: *has-field-derivative-def*)

lemma *DERIV-pow*: ((λ*x*. *x* ^ *n*) *has-field-derivative real n* ∗ (*x* ^ (*n* − *Suc 0*)))
(*at x within s*)
using *DERIV-power* [*OF DERIV-ident*] **by** *simp*

lemma *DERIV-power-int* [*derivative-intros*]:
assumes [*derivative-intros*]: (*f has-field-derivative d*) (*at x within s*)
and *n* ≥ *0* ∨ *f x* ≠ *0*
shows ((λ*x*. *power-int* (*f x*) *n*) *has-field-derivative*
(*of-int n* ∗ *power-int* (*f x*) (*n* − *1*) ∗ *d*)) (*at x within s*)
proof (*cases n rule*: *int-cases4*)
case (*nonneg n*)
thus *?thesis*
by (*cases n* = *0*; *cases f x* = *0*)
(*auto intro*!: *derivative-eq-intros simp*: *field-simps power-int-diff*
power-diff power-int-0-left-if)
next
case (*neg n*)
thus *?thesis* **using** *assms*(*2*)
by (*auto intro*!: *derivative-eq-intros simp*: *field-simps power-int-diff power-int-minus*
simp flip: *power-Suc power-Suc2 power-add*)
qed

lemma *DERIV-chain'*: (*f has-field-derivative D*) (*at x within s*) ⟹ *DERIV g* (*f*
x) :> *E* ⟹
((λ*x*. *g* (*f x*)) *has-field-derivative E* ∗ *D*) (*at x within s*)

using *has-derivative-compose*[*of f* (∗) *D x s g* (∗) *E*]
by (*simp only*: *has-field-derivative-def mult-commute-abs ac-simps*)

corollary *DERIV-chain2*: *DERIV f* (*g x*) :> *Da* ⟹ (*g has-field-derivative Db*) (*at x within s*) ⟹
((λ*x*. *f* (*g x*)) *has-field-derivative Da* ∗ *Db*) (*at x within s*)
by (*rule DERIV-chain′*)

Derivative of a finite product

lemma *has-field-derivative-prod*:
assumes ⋀*x*. *x* ∈ *A* ⟹ (*f x has-field-derivative f′ x*) (*at z*)
shows ((λ*u*. ∏ *x*∈*A*. *f x u*) *has-field-derivative* (∑ *x*∈*A*. *f′ x* ∗ (∏ *y*∈*A*−{*x*}. *f y z*))) (*at z*)
using *assms*
proof (*induction A rule*: *infinite-finite-induct*)
case (*insert x A*)
have *eq*: *insert x A* − {*y*} = *insert x* (*A* − {*y*}) **if** *y* ∈ *A* **for** *y*
using *insert.hyps that* **by** *auto*
show *?case*
using *insert.hyps*
by (*auto intro*!: *derivative-eq-intros insert.prems insert.IH sum.cong*
simp: *sum-distrib-left sum-distrib-right eq*)
qed *auto*

lemma *has-field-derivative-prod′*:
assumes ⋀*x*. *x* ∈ *A* ⟹ *f x z* ≠ *0*
assumes ⋀*x*. *x* ∈ *A* ⟹ (*f x has-field-derivative f′ x*) (*at z*)
defines *P* ≡ (λ*A u*. ∏ *x*∈*A*. *f x u*)
shows (*P A has-field-derivative* (*P A z* ∗ (∑ *x*∈*A*. *f′ x* / *f x z*))) (*at z*)
proof (*cases finite A*)
case *True*
note [*derivative-intros*] = *has-field-derivative-prod*
show *?thesis* **using** *assms True*
by (*auto intro*!: *derivative-eq-intros*
simp: *prod-diff1 sum-distrib-left sum-distrib-right mult-ac*)
qed (*auto simp*: *P-def*)

Standard version

lemma *DERIV-chain*:
DERIV f (*g x*) :> *Da* ⟹ (*g has-field-derivative Db*) (*at x within s*) ⟹
(*f* ∘ *g has-field-derivative Da* ∗ *Db*) (*at x within s*)
by (*drule* (*1*) *DERIV-chain′*, *simp add*: *o-def mult.commute*)

lemma *DERIV-image-chain*:
(*f has-field-derivative Da*) (*at* (*g x*) *within* (*g* ‘ *s*)) ⟹
(*g has-field-derivative Db*) (*at x within s*) ⟹
(*f* ∘ *g has-field-derivative Da* ∗ *Db*) (*at x within s*)
using *has-derivative-in-compose* [*of g* (∗) *Db x s f* (∗) *Da*]
by (*simp add*: *has-field-derivative-def o-def mult-commute-abs ac-simps*)

lemma *DERIV-chain-s*:
assumes ($\bigwedge x.\ x \in s \Longrightarrow DERIV\ g\ x :> g'(x)$)
and $DERIV\ f\ x :> f'$
and $f\ x \in s$
shows $DERIV\ (\lambda x.\ g(f\ x))\ x :> f' * g'(f\ x)$
by (*metis* (*full-types*) *DERIV-chain'* *mult.commute* *assms*)

lemma *DERIV-chain3*:
assumes ($\bigwedge x.\ DERIV\ g\ x :> g'(x)$)
and $DERIV\ f\ x :> f'$
shows $DERIV\ (\lambda x.\ g(f\ x))\ x :> f' * g'(f\ x)$
by (*metis UNIV-I DERIV-chain-s* [*of UNIV*] *assms*)

Alternative definition for differentiability

lemma *DERIV-LIM-iff*:
fixes $f :: 'a::\{real\text{-}normed\text{-}vector,inverse\} \Rightarrow 'a$
shows $((\lambda h.\ (f\ (a + h) - f\ a)\ /\ h)\ -0\to\ D) = ((\lambda x.\ (f\ x - f\ a)\ /\ (x - a))\ -a\to\ D)$ (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then have $(\lambda x.\ (f\ (a + (x + - a)) - f\ a)\ /\ (x + - a))\ -0 - - a\to\ D$
by (*rule LIM-offset*)
then show *?rhs*
by *simp*
next
assume *?rhs*
then have $(\lambda x.\ (f\ (x+a) - f\ a)\ /\ ((x+a) - a))\ -a-a\to\ D$
by (*rule LIM-offset*)
then show *?lhs*
by (*simp add*: *add.commute*)
qed

lemma *has-field-derivative-cong-ev*:
assumes $x = y$
and $*$: $eventually\ (\lambda x.\ x \in S \longrightarrow f\ x = g\ x)\ (nhds\ x)$
and $u = v\ S = t\ x \in S$
shows (*f has-field-derivative u*) (*at x within S*) = (*g has-field-derivative v*) (*at y within t*)
unfolding *has-field-derivative-iff*
proof (*rule filterlim-cong*)
from *assms* **have** $f\ y = g\ y$
by (*auto simp*: *eventually-nhds*)
with $*$ **show** $\forall_F\ z\ in\ at\ x\ within\ S.\ (f\ z - f\ x)\ /\ (z - x) = (g\ z - g\ y)\ /\ (z - y)$
unfolding *eventually-at-filter*
by *eventually-elim* (*auto simp*: *assms* ‹$f\ y = g\ y$›)
qed (*simp-all add*: *assms*)

lemma *has-field-derivative-cong-eventually*:
assumes *eventually* (λ*x*. *f x* = *g x*) (*at x within S*) *f x* = *g x*
shows (*f has-field-derivative u*) (*at x within S*) = (*g has-field-derivative u*) (*at x within S*)
unfolding *has-field-derivative-iff*
proof (*rule tendsto-cong*)
show ∀*F y in at x within S*. (*f y* − *f x*) / (*y* − *x*) = (*g y* − *g x*) / (*y* − *x*)
using *assms* **by** (*auto elim*: *eventually-mono*)
qed

lemma *DERIV-cong-ev*:
x = *y* ⟹ *eventually* (λ*x*. *f x* = *g x*) (*nhds x*) ⟹ *u* = *v* ⟹
DERIV f x :> *u* ⟷ *DERIV g y* :> *v*
by (*rule has-field-derivative-cong-ev*) *simp-all*

lemma *DERIV-mirror*: (*DERIV f* (− *x*) :> *y*) ⟷ (*DERIV* (λ*x*. *f* (− *x*)) *x* :> − *y*)
for *f* :: *real* ⇒ *real* **and** *x y* :: *real*
by (*simp add*: *DERIV-def filterlim-at-split filterlim-at-left-to-right tendsto-minus-cancel-left field-simps conj-commute*)

lemma *DERIV-shift*:
(*f has-field-derivative y*) (*at* (*x* + *z*)) = ((λ*x*. *f* (*x* + *z*)) *has-field-derivative y*) (*at x*)
by (*simp add*: *DERIV-def field-simps*)

lemma *DERIV-at-within-shift-lemma*:
assumes (*f has-field-derivative y*) (*at* (*z*+*x*) *within* (+) *z* ‘ *S*)
shows (*f* ∘ (+)*z has-field-derivative y*) (*at x within S*)
proof −
have ((+)*z has-field-derivative 1*) (*at x within S*)
by (*rule derivative-eq-intros* | *simp*)+
with *assms DERIV-image-chain* **show** *?thesis*
by (*metis mult.right-neutral*)
qed

lemma *DERIV-at-within-shift*:
(*f has-field-derivative y*) (*at* (*z*+*x*) *within* (+) *z* ‘ *S*) ⟷
((λ*x*. *f* (*z*+*x*)) *has-field-derivative y*) (*at x within S*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs* **then show** *?rhs*
using *DERIV-at-within-shift-lemma* **unfolding** *o-def* **by** *blast*
next
have [*simp*]: (λ*x*. *x* − *z*) ‘ (+) *z* ‘ *S* = *S*
by *force*
assume *R*: *?rhs*
have (*f* ∘ (+) *z* ∘ (+) (− *z*) *has-field-derivative y*) (*at* (*z* + *x*) *within* (+) *z* ‘ *S*)
by (*rule DERIV-at-within-shift-lemma*) (*use R* **in** ‹*simp add*: *o-def*›)

then show *?lhs*
by (*simp add*: *o-def*)
qed

lemma *floor-has-real-derivative*:
fixes f :: *real* ⇒ 'a::{*floor-ceiling*,*order-topology*}
assumes *isCont f x*
and $f\ x \notin \mathbb{Z}$
shows ((λ*x*. *floor* (*f x*)) *has-real-derivative 0*) (*at x*)
proof (*subst DERIV-cong-ev*[*OF refl - refl*])
show ((λ-. *floor* (*f x*)) *has-real-derivative 0*) (*at x*)
by *simp*
have ∀$_F$ *y in at x*. ⌊*f y*⌋ = ⌊*f x*⌋
by (*rule eventually-floor-eq*[*OF assms*[*unfolded continuous-at*]])
then show ∀$_F$ *y in nhds x*. *real-of-int* ⌊*f y*⌋ = *real-of-int* ⌊*f x*⌋
unfolding *eventually-at-filter*
by *eventually-elim auto*
qed

lemmas *has-derivative-floor*[*derivative-intros*] =
floor-has-real-derivative[*THEN DERIV-compose-FDERIV*]

lemma *continuous-floor*:
fixes *x*::*real*
shows $x \notin \mathbb{Z} \Longrightarrow$ *continuous* (*at x*) (*real-of-int* ∘ *floor*)
using *floor-has-real-derivative* [**where** *f*=*id*]
by (*auto simp*: *o-def has-field-derivative-def intro*: *has-derivative-continuous*)

lemma *continuous-frac*:
fixes *x*::*real*
assumes $x \notin \mathbb{Z}$
shows *continuous* (*at x*) *frac*
proof −
have *isCont* (λ*x*. *real-of-int* ⌊*x*⌋) *x*
using *continuous-floor* [*OF assms*] **by** (*simp add*: *o-def*)
then have ∗: *continuous* (*at x*) (λ*x*. *x* − *real-of-int* ⌊*x*⌋)
by (*intro continuous-intros*)
moreover have ∀$_F$ *x in nhds x*. *frac x* = *x* − *real-of-int* ⌊*x*⌋
by (*simp add*: *frac-def*)
ultimately show *?thesis*
by (*simp add*: *LIM-imp-LIM frac-def isCont-def*)
qed

Caratheodory formulation of derivative at a point

lemma *CARAT-DERIV*:
(*DERIV f x* :> *l*) ⟷ (∃ *g*. (∀ *z*. *f z* − *f x* = *g z* ∗ (*z* − *x*)) ∧ *isCont g x* ∧ *g x* = *l*)
(**is** *?lhs* = *?rhs*)
proof

assume *?lhs*
show $\exists g.\ (\forall z.\ f\ z - f\ x = g\ z * (z - x)) \wedge isCont\ g\ x \wedge g\ x = l$
proof (*intro exI conjI*)
let *?g* = $(\lambda z.\ if\ z = x\ then\ l\ else\ (f\ z - f\ x)\ /\ (z - x))$
show $\forall z.\ f\ z - f\ x = ?g\ z * (z - x)$
by *simp*
show *isCont ?g x*
using ‹*?lhs*› **by** (*simp add*: *isCont-iff DERIV-def cong*: *LIM-equal* [*rule-format*])
show *?g x* = *l*
by *simp*
qed
next
assume *?rhs*
then show *?lhs*
by (*auto simp add*: *isCont-iff DERIV-def cong*: *LIM-cong*)
qed

110.8 Local extrema

If $0 < f'\ x$ then x is Locally Strictly Increasing At The Right.

lemma *has-real-derivative-pos-inc-right*:
fixes *f* :: *real* ⇒ *real*
assumes *der*: (*f has-real-derivative l*) (*at x within S*)
and *l*: *0* < *l*
shows $\exists d > 0.\ \forall h > 0.\ x + h \in S \longrightarrow h < d \longrightarrow f\ x < f\ (x + h)$
using *assms*
proof −
from *der* [*THEN has-field-derivativeD*, *THEN tendstoD*, *OF l*, *unfolded eventually-at*]
obtain *s* **where** *s*: *0* < *s*
and *all*: $\bigwedge xa.\ xa \in S \Longrightarrow xa \neq x \wedge dist\ xa\ x < s \longrightarrow |(f\ xa - f\ x)\ /\ (xa - x) - l| < l$
by (*auto simp*: *dist-real-def*)
then show *?thesis*
proof (*intro exI conjI strip*)
show *0* < *s* **by** (*rule s*)
next
fix *h* :: *real*
assume *0* < *h h* < *s x* + *h* ∈ *S*
with *all* [*of x* + *h*] **show** *f x* < *f* (*x*+*h*)
proof (*simp add*: *abs-if dist-real-def pos-less-divide-eq split*: *if-split-asm*)
assume ¬ (*f* (*x* + *h*) − *f x*) / *h* < *l* **and** *h*: *0* < *h*
with *l* **have** *0* < (*f* (*x* + *h*) − *f x*) / *h*
by *arith*
then show *f x* < *f* (*x* + *h*)
by (*simp add*: *pos-less-divide-eq h*)
qed
qed
qed

lemma *DERIV-pos-inc-right*:
fixes f :: *real* ⇒ *real*
assumes *der*: *DERIV f x* :> l
and l: $0 < l$
shows ∃ $d > 0$. ∀ $h > 0$. $h < d \longrightarrow f\ x < f\ (x + h)$
using *has-real-derivative-pos-inc-right*[*OF assms*]
by *auto*

lemma *has-real-derivative-neg-dec-left*:
fixes f :: *real* ⇒ *real*
assumes *der*: (*f has-real-derivative l*) (*at x within S*)
and $l < 0$
shows ∃ $d > 0$. ∀ $h > 0$. $x - h \in S \longrightarrow h < d \longrightarrow f\ x < f\ (x - h)$
proof −
from ‹$l < 0$› **have** l: $- l > 0$
by *simp*
from *der* [*THEN has-field-derivativeD*, *THEN tendstoD*, *OF l*, *unfolded eventually-at*]
obtain s **where** s: $0 < s$
and *all*: ⋀*xa*. *xa*∈*S* ⟹ $xa \neq x \wedge$ *dist xa x* $< s \longrightarrow |(f\ xa - f\ x)\ /\ (xa - x) - l| < - l$
by (*auto simp*: *dist-real-def*)
then show *?thesis*
proof (*intro exI conjI strip*)
show $0 < s$ **by** (*rule s*)
next
fix h :: *real*
assume $0 < h$ $h < s$ $x - h \in S$
with *all* [*of x − h*] **show** $f\ x < f\ (x-h)$
proof (*simp add*: *abs-if pos-less-divide-eq dist-real-def split*: *if-split-asm*)
assume − $((f\ (x-h) - f\ x)\ /\ h) < l$ **and** h: $0 < h$
with l **have** $0 < (f\ (x-h) - f\ x)\ /\ h$
by *arith*
then show $f\ x < f\ (x - h)$
by (*simp add*: *pos-less-divide-eq h*)
qed
qed
qed

lemma *DERIV-neg-dec-left*:
fixes f :: *real* ⇒ *real*
assumes *der*: *DERIV f x* :> l
and l: $l < 0$
shows ∃ $d > 0$. ∀ $h > 0$. $h < d \longrightarrow f\ x < f\ (x - h)$
using *has-real-derivative-neg-dec-left*[*OF assms*]
by *auto*

lemma *has-real-derivative-pos-inc-left*:

fixes f :: *real* ⇒ *real*
shows (f *has-real-derivative* l) (*at* x *within* S) ⟹ $0 < l$ ⟹
$\exists\, d{>}0.\ \forall\, h{>}0.\ x - h \in S \longrightarrow h < d \longrightarrow f\ (x - h) < f\ x$
by (*rule has-real-derivative-neg-dec-left* [*of* $\lambda x.\ -f\ x$ $-l$ x S, *simplified*])
(*auto simp add*: *DERIV-minus*)

lemma *DERIV-pos-inc-left*:
fixes f :: *real* ⇒ *real*
shows *DERIV* $f\ x$:> l ⟹ $0 < l$ ⟹ $\exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\ (x - h) < f\ x$
using *has-real-derivative-pos-inc-left*
by *blast*

lemma *has-real-derivative-neg-dec-right*:
fixes f :: *real* ⇒ *real*
shows (f *has-real-derivative* l) (*at* x *within* S) ⟹ $l < 0$ ⟹
$\exists\, d > 0.\ \forall\, h > 0.\ x + h \in S \longrightarrow h < d \longrightarrow f\ x > f\ (x + h)$
by (*rule has-real-derivative-pos-inc-right* [*of* $\lambda x.\ -f\ x$ $-l$ x S, *simplified*])
(*auto simp add*: *DERIV-minus*)

lemma *DERIV-neg-dec-right*:
fixes f :: *real* ⇒ *real*
shows *DERIV* $f\ x$:> l ⟹ $l < 0$ ⟹ $\exists\, d > 0.\ \forall\, h > 0.\ h < d \longrightarrow f\ x > f\ (x + h)$
using *has-real-derivative-neg-dec-right* **by** *blast*

lemma *DERIV-local-max*:
fixes f :: *real* ⇒ *real*
assumes *der*: *DERIV* $f\ x$:> l
and d: $0 < d$
and *le*: $\forall\, y.\ |x - y| < d \longrightarrow f\ y \leq f\ x$
shows $l = 0$
proof (*cases rule*: *linorder-cases* [*of* l 0])
case *equal*
then show *?thesis* **.**
next
case *less*
from *DERIV-neg-dec-left* [*OF der less*]
obtain d' **where** d': $0 < d'$ **and** *lt*: $\forall\, h > 0.\ h < d' \longrightarrow f\ x < f\ (x - h)$
by *blast*
obtain e **where** $0 < e \wedge e < d \wedge e < d'$
using *field-lbound-gt-zero* [*OF d d'*] **..**
with *lt le* [*THEN spec* [**where** $x{=}x - e$]] **show** *?thesis*
by (*auto simp add*: *abs-if*)
next
case *greater*
from *DERIV-pos-inc-right* [*OF der greater*]
obtain d' **where** d': $0 < d'$ **and** *lt*: $\forall\, h > 0.\ h < d' \longrightarrow f\ x < f\ (x + h)$
by *blast*

obtain e **where** $0 < e \wedge e < d \wedge e < d'$
using *field-lbound-gt-zero* [*OF d d'*] **..**
with *lt le* [*THEN spec* [**where** $x{=}x + e$]] **show** *?thesis*
by (*auto simp add*: *abs-if*)
qed

Similar theorem for a local minimum

lemma *DERIV-local-min*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y.\ |x - y| < d \longrightarrow f\ x \leq f\ y \Longrightarrow l = 0$
by (*drule DERIV-minus* [*THEN DERIV-local-max*]) *auto*

In particular, if a function is locally flat

lemma *DERIV-local-const*:
fixes $f :: real \Rightarrow real$
shows $DERIV\ f\ x :> l \Longrightarrow 0 < d \Longrightarrow \forall y.\ |x - y| < d \longrightarrow f\ x = f\ y \Longrightarrow l = 0$
by (*auto dest*!: *DERIV-local-max*)

110.9 Rolle's Theorem

Lemma about introducing open ball in open interval

lemma *lemma-interval-lt*:
fixes $a\ b\ x :: real$
assumes $a < x\ x < b$
shows $\exists d.\ 0 < d \wedge (\forall y.\ |x - y| < d \longrightarrow a < y \wedge y < b)$
using *linorder-linear* [*of* $x - a\ b - x$]
proof
assume $x - a \leq b - x$
with *assms* **show** *?thesis*
by (*rule-tac* $x = x - a$ **in** *exI*) *auto*
next
assume $b - x \leq x - a$
with *assms* **show** *?thesis*
by (*rule-tac* $x = b - x$ **in** *exI*) *auto*
qed

lemma *lemma-interval*: $a < x \Longrightarrow x < b \Longrightarrow \exists d.\ 0 < d \wedge (\forall y.\ |x - y| < d \longrightarrow a \leq y \wedge y \leq b)$
for $a\ b\ x :: real$
by (*force dest*: *lemma-interval-lt*)

Rolle's Theorem. If f is defined and continuous on the closed interval $[a,b]$ and differentiable on the open interval (a,b), and $f\ a = f\ b$, then there exists $x0 \in (a,b)$ such that $f'\ x0 = 0$

theorem *Rolle-deriv*:
fixes $f :: real \Rightarrow real$
assumes $a < b$
and *fab*: $f\ a = f\ b$

```
    and contf: continuous-on {a..b} f
    and derf: ⋀x. ⟦a < x; x < b⟧ ⟹ (f has-derivative f' x) (at x)
  shows ∃z. a < z ∧ z < b ∧ f' z = (λv. 0)
proof −
  have le: a ≤ b
    using ‹a < b› by simp
    have (a + b) / 2 ∈ {a..b}
      using assms(1) by auto
    then have ∗: {a..b} ≠ {}
      by auto
  obtain x where x-max: ∀z. a ≤ z ∧ z ≤ b ⟶ f z ≤ f x and a ≤ x x ≤ b
    using continuous-attains-sup[OF compact-Icc ∗ contf]
    by (meson atLeastAtMost-iff)
  obtain x' where x'-min: ∀z. a ≤ z ∧ z ≤ b ⟶ f x' ≤ f z and a ≤ x' x' ≤ b
     using continuous-attains-inf[OF compact-Icc ∗ contf] by (meson atLeastAt-
Most-iff)
  consider a < x x < b | x = a ∨ x = b
    using ‹a ≤ x› ‹x ≤ b› by arith
  then show ?thesis
  proof cases
    case 1
    — f attains its maximum within the interval
    then obtain l where der: DERIV f x :> l
      using derf differentiable-def real-differentiable-def by blast
    obtain d where d: 0 < d and bound: ∀y. |x − y| < d ⟶ a ≤ y ∧ y ≤ b
      using lemma-interval [OF 1] by blast
    then have bound': ∀y. |x − y| < d ⟶ f y ≤ f x
      using x-max by blast
    — the derivative at a local maximum is zero
    have l = 0
      by (rule DERIV-local-max [OF der d bound'])
    with 1 der derf [of x] show ?thesis
      by (metis has-derivative-unique has-field-derivative-def mult-zero-left)
  next
    case 2
    then have fx: f b = f x by (auto simp add: fab)
    consider a < x' x' < b | x' = a ∨ x' = b
      using ‹a ≤ x'› ‹x' ≤ b› by arith
    then show ?thesis
    proof cases
      case 1
        — f attains its minimum within the interval
      then obtain l where der: DERIV f x' :> l
        using derf differentiable-def real-differentiable-def by blast
      from lemma-interval [OF 1]
      obtain d where d: 0<d and bound: ∀y. |x'−y| < d ⟶ a ≤ y ∧ y ≤ b
        by blast
      then have bound': ∀y. |x' − y| < d ⟶ f x' ≤ f y
        using x'-min by blast
```

```
    have l = 0 by (rule DERIV-local-min [OF der d bound'])
      — the derivative at a local minimum is zero
    then show ?thesis using 1 der derf [of x']
      by (metis has-derivative-unique has-field-derivative-def mult-zero-left)
  next
    case 2
      — f is constant throughout the interval
    then have fx': f b = f x' by (auto simp: fab)
    from dense [OF ‹a < b›] obtain r where r: a < r r < b by blast
    obtain d where d: 0 < d and bound: ∀ y. |r − y| < d ⟶ a ≤ y ∧ y ≤ b
      using lemma-interval [OF r] by blast
    have eq-fb: f z = f b if a ≤ z and z ≤ b for z
    proof (rule order-antisym)
      show f z ≤ f b by (simp add: fx x-max that)
      show f b ≤ f z by (simp add: fx' x'-min that)
    qed
    have bound': ∀ y. |r − y| < d ⟶ f r = f y
    proof (intro strip)
      fix y :: real
      assume lt: |r − y| < d
      then have f y = f b by (simp add: eq-fb bound)
      then show f r = f y by (simp add: eq-fb r order-less-imp-le)
    qed
    obtain l where der: DERIV f r :> l
      using derf differentiable-def r(1) r(2) real-differentiable-def by blast
    have l = 0
      by (rule DERIV-local-const [OF der d bound'])
      — the derivative of a constant function is zero
    with r der derf [of r] show ?thesis
      by (metis has-derivative-unique has-field-derivative-def mult-zero-left)
  qed
qed

corollary Rolle:
  fixes a b :: real
  assumes ab: a < b f a = f b continuous-on {a..b} f
    and dif [rule-format]: ⋀x. ⟦a < x; x < b⟧ ⟹ f differentiable (at x)
  shows ∃ z. a < z ∧ z < b ∧ DERIV f z :> 0
proof −
  obtain f' where f': ⋀x. ⟦a < x; x < b⟧ ⟹ (f has-derivative f' x) (at x)
    using dif unfolding differentiable-def by metis
  then have ∃ z. a < z ∧ z < b ∧ f' z = (λv. 0)
    by (metis Rolle-deriv [OF ab])
  then show ?thesis
    using f' has-derivative-imp-has-field-derivative by fastforce
qed
```

110.10 Mean Value Theorem

theorem *mvt*:
fixes $f :: real \Rightarrow real$
assumes $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *derf*: $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow (f\ has\text{-}derivative\ f'\ x)\ (at\ x)$
obtains ξ **where** $a < \xi\ \xi < b\ f\ b - f\ a = (f'\ \xi)\ (b - a)$
proof −
have $\exists \xi.\ a < \xi \wedge \xi < b \wedge (\lambda y.\ f'\ \xi\ y - (f\ b - f\ a)\ /\ (b - a) * y) = (\lambda v.\ 0)$
proof (*intro Rolle-deriv*[*OF* ‹$a < b$›])
fix x
assume *x*: $a < x\ x < b$
show $((\lambda x.\ f\ x - (f\ b - f\ a)\ /\ (b - a) * x)$
has-derivative $(\lambda y.\ f'\ x\ y - (f\ b - f\ a)\ /\ (b - a) * y))\ (at\ x)$
by (*intro derivative-intros derf*[*OF x*])
qed (*use assms* **in** ‹*auto intro*!: *continuous-intros simp*: *field-simps*›)
then show *?thesis*
by (*smt* (*verit*, *ccfv-SIG*) *pos-le-divide-eq pos-less-divide-eq that*)
qed

theorem *MVT*:
fixes $a\ b :: real$
assumes *lt*: $a < b$
and *contf*: *continuous-on* $\{a..b\}$ f
and *dif*: $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow f\ differentiable\ (at\ x)$
shows $\exists l\ z.\ a < z \wedge z < b \wedge DERIV\ f\ z :> l \wedge f\ b - f\ a = (b - a) * l$
proof −
obtain $f' :: real \Rightarrow real \Rightarrow real$
where *derf*: $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow (f\ has\text{-}derivative\ f'\ x)\ (at\ x)$
using *dif* **unfolding** *differentiable-def* **by** *metis*
then obtain *z* **where** $a < z\ z < b\ f\ b - f\ a = (f'\ z)\ (b - a)$
using *mvt* [*OF lt contf*] **by** *blast*
then show *?thesis*
by (*simp add*: *ac-simps*)
(*metis derf dif has-derivative-unique has-field-derivative-imp-has-derivative real-differentiable-def*)
qed

corollary *MVT2*:
assumes $a < b$ **and** *der*: $\bigwedge x.\ [\![a \le x;\ x \le b]\!] \Longrightarrow DERIV\ f\ x :> f'\ x$
shows $\exists z{::}real.\ a < z \wedge z < b \wedge (f\ b - f\ a = (b - a) * f'\ z)$
proof −
have $\exists l\ z.\ a < z \wedge$
$z < b \wedge$
$(f\ has\text{-}real\text{-}derivative\ l)\ (at\ z) \wedge$
$f\ b - f\ a = (b - a) * l$
proof (*rule MVT* [*OF* ‹$a < b$›])
show *continuous-on* $\{a..b\}$ f
by (*meson DERIV-continuous atLeastAtMost-iff continuous-at-imp-continuous-on*

der)
  show $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow f$ *differentiable* (*at* x)
   using *assms* **by** (*force dest*: *order-less-imp-le simp add*: *real-differentiable-def*)
 qed
 with *assms* **show** *?thesis*
  by (*blast dest*: *DERIV-unique order-less-imp-le*)
qed

110.10.1 A function is constant if its derivative is 0 over an interval.

lemma *DERIV-isconst-end*:
 fixes f :: *real* $\Rightarrow$ *real*
 assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f
  and *0*: $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow$ *DERIV* $f\ x$:> *0*
 shows $f\ b = f\ a$
 using *MVT* [*OF* ‹$a < b$›] *0 DERIV-unique contf real-differentiable-def*
 by (*fastforce simp*: *algebra-simps*)

lemma *DERIV-isconst2*:
 fixes f :: *real* $\Rightarrow$ *real*
 assumes $a < b$ **and** *contf*: *continuous-on* $\{a..b\}$ f **and** *derf*: $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow$ *DERIV* $f\ x$:> *0*
  and $a \leq x\ x \leq b$
shows $f\ x = f\ a$
proof (*cases* $a < x$)
 case *True*
 have $*$: *continuous-on* $\{a..x\}$ f
  using ‹$x \leq b$› *contf continuous-on-subset* **by** *fastforce*
 show *?thesis*
  by (*rule DERIV-isconst-end* [*OF True* $*$]) (*use* ‹$x \leq b$› *derf* **in** *auto*)
qed (*use* ‹$a \leq x$› **in** *auto*)

lemma *DERIV-isconst3*:
 fixes $a\ b\ x\ y$ :: *real*
 assumes $a < b$
  and $x \in \{a <..< b\}$
  and $y \in \{a <..< b\}$
  and *derivable*: $\bigwedge x.\ x \in \{a <..< b\} \Longrightarrow$ *DERIV* $f\ x$:> *0*
 shows $f\ x = f\ y$
proof (*cases* $x = y$)
 case *False*
 let *?a* = *min x y*
 let *?b* = *max x y*
 have $*$: *DERIV* $f\ z$:> *0* **if** *?a* $\leq z\ z \leq$ *?b* **for** z
 proof −
  have $a < z$ **and** $z < b$
   using *that* ‹$x \in \{a <..< b\}$› **and** ‹$y \in \{a <..< b\}$› **by** *auto*
  then have $z \in \{a<..<b\}$ **by** *auto*

```
    then show DERIV f z :> 0 by (rule derivable)
  qed
  have isCont: continuous-on {?a..?b} f
   by (meson * DERIV-continuous-on atLeastAtMost-iff has-field-derivative-at-within)
  have DERIV: ⋀z. ⟦?a < z; z < ?b⟧ ⟹ DERIV f z :> 0
    using * by auto
  have ?a < ?b using ‹x ≠ y› by auto
  from DERIV-isconst2[OF this isCont DERIV, of x] and DERIV-isconst2[OF
this isCont DERIV, of y]
  show ?thesis by auto
qed auto

lemma DERIV-isconst-all:
  fixes f :: real ⇒ real
  shows ∀x. DERIV f x :> 0 ⟹ f x = f y
  apply (rule linorder-cases [of x y])
 apply (metis DERIV-continuous DERIV-isconst-end continuous-at-imp-continuous-on)+
  done

lemma DERIV-const-ratio-const:
  fixes f :: real ⇒ real
  assumes a ≠ b and df: ⋀x. DERIV f x :> k
  shows f b − f a = (b − a) * k
proof (cases a b rule: linorder-cases)
  case less
  show ?thesis
    using MVT [OF less] df
    by (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on
real-differentiable-def)
next
  case greater
  have  f a − f b = (a − b) * k
    using MVT [OF greater] df
    by (metis DERIV-continuous DERIV-unique continuous-at-imp-continuous-on
real-differentiable-def)
  then show ?thesis
    by (simp add: algebra-simps)
qed auto

lemma DERIV-const-ratio-const2:
  fixes f :: real ⇒ real
  assumes a ≠ b and df: ⋀x. DERIV f x :> k
  shows (f b − f a) / (b − a) = k
  using DERIV-const-ratio-const [OF assms] ‹a ≠ b› by auto

lemma real-average-minus-first [simp]: (a + b) / 2 − a = (b − a) / 2
  for a b :: real
  by simp
```

lemma *real-average-minus-second* [*simp*]: $(b + a) / 2 - a = (b - a) / 2$
for $a\ b :: real$
by *simp*

Gallileo's "trick": average velocity = av. of end velocities.

lemma *DERIV-const-average*:
fixes $v :: real \Rightarrow real$
and $a\ b :: real$
assumes *neq*: $a \neq b$
and *der*: $\bigwedge x.\ DERIV\ v\ x :> k$
shows $v\ ((a + b) / 2) = (v\ a + v\ b) / 2$
proof (*cases rule*: *linorder-cases* [*of a b*])
case *equal*
with *neq* **show** *?thesis* **by** *simp*
next
case *less*
have $(v\ b - v\ a) / (b - a) = k$
by (*rule DERIV-const-ratio-const2* [*OF neq der*])
then have $(b - a) * ((v\ b - v\ a) / (b - a)) = (b - a) * k$
by *simp*
moreover have $(v\ ((a + b) / 2) - v\ a) / ((a + b) / 2 - a) = k$
by (*rule DERIV-const-ratio-const2* [*OF - der*]) (*simp add*: *neq*)
ultimately show *?thesis*
using *neq* **by** *force*
next
case *greater*
have $(v\ b - v\ a) / (b - a) = k$
by (*rule DERIV-const-ratio-const2* [*OF neq der*])
then have $(b - a) * ((v\ b - v\ a) / (b - a)) = (b - a) * k$
by *simp*
moreover have $(v\ ((b + a) / 2) - v\ a) / ((b + a) / 2 - a) = k$
by (*rule DERIV-const-ratio-const2* [*OF - der*]) (*simp add*: *neq*)
ultimately show *?thesis*
using *neq* **by** (*force simp add*: *add.commute*)
qed

110.10.2 A function with positive derivative is increasing

A simple proof using the MVT, by Jeremy Avigad. And variants.

lemma *DERIV-pos-imp-increasing-open*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a < b$
and $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y > 0)$
and *con*: *continuous-on* $\{a..b\}\ f$
shows $f\ a < f\ b$
proof (*rule ccontr*)
assume *f*: $\neg$ *?thesis*
have $\exists l\ z.\ a < z \wedge z < b \wedge DERIV\ f\ z :> l \wedge f\ b - f\ a = (b - a) * l$

by (*rule MVT*) (*use assms real-differentiable-def* **in** ‹*force*+›)
then obtain *l z* **where** *z*: $a < z\ z < b$ *DERIV f z :> l* **and** $f\ b - f\ a = (b - a) * l$
by *auto*
with *assms f* **have** $\neg\ l > 0$
by (*metis linorder-not-le mult-le-0-iff diff-le-0-iff-le*)
with *assms z* **show** *False*
by (*metis DERIV-unique*)
qed

lemma *DERIV-pos-imp-increasing*:
fixes *a b* :: *real* **and** *f* :: *real* ⇒ *real*
assumes $a < b$
and *der*: $\bigwedge x.\ [\![a \leq x; x \leq b]\!] \Longrightarrow \exists\, y.\ DERIV\ f\ x :> y \wedge y > 0$
shows $f\ a < f\ b$
by (*metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-pos-imp-increasing-open* [*OF* ‹$a < b$›] *der*)

lemma *DERIV-nonneg-imp-nondecreasing*:
fixes *a b* :: *real*
and *f* :: *real* ⇒ *real*
assumes $a \leq b$
and $\bigwedge x.\ [\![a \leq x; x \leq b]\!] \Longrightarrow \exists\, y.\ DERIV\ f\ x :> y \wedge y \geq 0$
shows $f\ a \leq f\ b$
proof (*rule ccontr*, *cases* $a = b$)
assume ¬ *?thesis* **and** $a = b$
then show *False* **by** *auto*
next
assume ∗: ¬ *?thesis*
assume $a \neq b$
with ‹$a \leq b$› **have** $a < b$
by *linarith*
moreover have *continuous-on* {*a*..*b*} *f*
by (*meson DERIV-isCont assms*(*2*) *atLeastAtMost-iff continuous-at-imp-continuous-on*)
ultimately have $\exists\, l\ z.\ a < z \wedge z < b \wedge DERIV\ f\ z :> l \wedge f\ b - f\ a = (b - a) * l$
using *assms MVT* [*OF* ‹$a < b$›, *of f*] *real-differentiable-def less-eq-real-def* **by** *blast*
then obtain *l z* **where** *lz*: $a < z\ z < b$ *DERIV f z :> l* **and** ∗∗: $f\ b - f\ a = (b - a) * l$
by *auto*
with ∗ **have** $a < b\ f\ b < f\ a$ **by** *auto*
with ∗∗ **have** $\neg\ l \geq 0$ **by** (*auto simp add*: *not-le algebra-simps*)
(*metis* ∗ *add-le-cancel-right assms*(*1*) *less-eq-real-def mult-right-mono add-left-mono linear order-refl*)
with *assms lz* **show** *False*
by (*metis DERIV-unique order-less-imp-le*)
qed

lemma *DERIV-neg-imp-decreasing-open*:
fixes *a b* :: *real*
and *f* :: *real* ⇒ *real*
assumes *a* < *b*
and ⋀*x*. *a* < *x* ⟹ *x* < *b* ⟹ ∃ *y*. *DERIV f x* :> *y* ∧ *y* < *0*
and *con*: *continuous-on* {*a*..*b*} *f*
shows *f a* > *f b*
proof −
have (λ*x*. −*f x*) *a* < (λ*x*. −*f x*) *b*
proof (*rule DERIV-pos-imp-increasing-open* [*of a b*])
show ⋀*x*. ⟦*a* < *x*; *x* < *b*⟧ ⟹ ∃ *y*. ((λ*x*. − *f x*) *has-real-derivative y*) (*at x*) ∧
0 < *y*
using *assms*
by *simp* (*metis field-differentiable-minus neg-0-less-iff-less*)
show *continuous-on* {*a*..*b*} (λ*x*. − *f x*)
using *con continuous-on-minus* **by** *blast*
qed (*use assms* **in** *auto*)
then show *?thesis*
by *simp*
qed

lemma *DERIV-neg-imp-decreasing*:
fixes *a b* :: *real* **and** *f* :: *real* ⇒ *real*
assumes *a* < *b*
and *der*: ⋀*x*. ⟦*a* ≤ *x*; *x* ≤ *b*⟧ ⟹ ∃ *y*. *DERIV f x* :> *y* ∧ *y* < *0*
shows *f a* > *f b*
by (*metis less-le-not-le DERIV-atLeastAtMost-imp-continuous-on DERIV-neg-imp-decreasing-open*
[*OF* ‹*a* < *b*›] *der*)

lemma *DERIV-nonpos-imp-nonincreasing*:
fixes *a b* :: *real*
and *f* :: *real* ⇒ *real*
assumes *a* ≤ *b*
and ⋀*x*. ⟦*a* ≤ *x*; *x* ≤ *b*⟧ ⟹ ∃ *y*. *DERIV f x* :> *y* ∧ *y* ≤ *0*
shows *f a* ≥ *f b*
proof −
have (λ*x*. −*f x*) *a* ≤ (λ*x*. −*f x*) *b*
using *DERIV-nonneg-imp-nondecreasing* [*of a b* λ*x*. −*f x*] *assms DERIV-minus*
by *fastforce*
then show *?thesis*
by *simp*
qed

lemma *DERIV-pos-imp-increasing-at-bot*:
fixes *f* :: *real* ⇒ *real*
assumes ⋀*x*. *x* ≤ *b* ⟹ (∃ *y*. *DERIV f x* :> *y* ∧ *y* > *0*)
and *lim*: (*f* ⟶ *flim*) *at-bot*
shows *flim* < *f b*
proof −

have $\exists N.\ \forall n{\leq}N.\ f\ n \leq f\ (b - 1)$
by (*rule-tac* $x{=}b - 2$ **in** *exI*) (*force intro*: *order.strict-implies-order DERIV-pos-imp-increasing assms*)
then have $flim \leq f\ (b - 1)$
by (*auto simp*: *eventually-at-bot-linorder tendsto-upperbound* [*OF lim*])
also have $\ldots < f\ b$
by (*force intro*: *DERIV-pos-imp-increasing* [**where** $f{=}f$] *assms*)
finally show *?thesis* **.**
qed

lemma *DERIV-neg-imp-decreasing-at-top*:
fixes $f :: real \Rightarrow real$
assumes *der*: $\bigwedge x.\ x \geq b \Longrightarrow \exists y.\ DERIV\ f\ x :> y \wedge y < 0$
and *lim*: $(f \longrightarrow flim)$ *at-top*
shows $flim < f\ b$
apply (*rule DERIV-pos-imp-increasing-at-bot* [**where** $f = \lambda i.\ f\ (-i)$ **and** $b = -b$, *simplified*])
apply (*metis DERIV-mirror der le-minus-iff neg-0-less-iff-less*)
apply (*metis filterlim-at-top-mirror lim*)
done

proposition *deriv-nonpos-imp-antimono*:
assumes *deriv*: $\bigwedge x.\ x \in \{a..b\} \Longrightarrow (g\ has\text{-}real\text{-}derivative\ g'\ x)\ (at\ x)$
assumes *nonneg*: $\bigwedge x.\ x \in \{a..b\} \Longrightarrow g'\ x \leq 0$
assumes $a \leq b$
shows $g\ b \leq g\ a$
proof −
have $-\ g\ a \leq -\ g\ b$
proof (*intro DERIV-nonneg-imp-nondecreasing* [**where** $f = \lambda x.\ -\ g\ x$] *conjI exI*)
fix x
assume x: $a \leq x\ x \leq b$
show $((\lambda x.\ -\ g\ x)\ has\text{-}real\text{-}derivative\ -\ g'\ x)\ (at\ x)$
by (*simp add*: *DERIV-minus deriv x*)
show $0 \leq -\ g'\ x$
by (*simp add*: *nonneg x*)
qed (*rule* ‹$a{\leq}b$›)
then show *?thesis* **by** *simp*
qed

lemma *DERIV-nonneg-imp-increasing-open*:
fixes $a\ b :: real$
and $f :: real \Rightarrow real$
assumes $a \leq b$
and $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow (\exists y.\ DERIV\ f\ x :> y \wedge y \geq 0)$
and *con*: *continuous-on* $\{a..b\}\ f$
shows $f\ a \leq f\ b$
proof (*cases* $a{=}b$)
case *False*

with ‹$a \leq b$› **have** $a<b$ **by** *simp*
show *?thesis*
proof (*rule ccontr*)
assume f: $\neg$ *?thesis*
have $\exists\, l\ z.\ a < z \wedge z < b \wedge$ *DERIV* $f\ z :> l \wedge f\ b - f\ a = (b - a) * l$
by (*rule MVT*) (*use assms* ‹$a<b$› *real-differentiable-def* **in** ‹*force*+›)
then obtain $l\ z$ **where** z: $a < z\ z < b$ *DERIV* $f\ z :> l$ **and** $f\ b - f\ a = (b - a) * l$
by *auto*
with *assms* $z\ f$ **show** *False*
by (*metis DERIV-unique diff-ge-0-iff-ge zero-le-mult-iff*)
qed
qed *auto*

lemma *DERIV-nonpos-imp-decreasing-open*:
fixes $a\ b$ :: *real*
and f :: *real* $\Rightarrow$ *real*
assumes $a \leq b$
and $\bigwedge x.\ a < x \Longrightarrow x < b \Longrightarrow \exists\, y.$ *DERIV* $f\ x :> y \wedge y \leq 0$
and *con*: *continuous-on* $\{a..b\}\ f$
shows $f\ a \geq f\ b$
proof −
have $(\lambda x.\ -f\ x)\ a \leq (\lambda x.\ -f\ x)\ b$
proof (*rule DERIV-nonneg-imp-increasing-open* [*of a b*])
show $\bigwedge x.\ [\![a < x;\ x < b]\!] \Longrightarrow \exists\, y.$ $((\lambda x.\ -\ f\ x)$ *has-real-derivative* $y)$ $(at\ x) \wedge 0 \leq y$
using *assms*
by (*metis Deriv.field-differentiable-minus neg-0-le-iff-le*)
show *continuous-on* $\{a..b\}$ $(\lambda x.\ -\ f\ x)$
using *con continuous-on-minus* **by** *blast*
qed (*use assms* **in** *auto*)
then show *?thesis*
by *simp*
qed

proposition *deriv-nonneg-imp-mono*:
assumes $\bigwedge x.\ x \in \{a..b\} \Longrightarrow (g$ *has-real-derivative* $g'\ x)$ $(at\ x)$
assumes $\bigwedge x.\ x \in \{a..b\} \Longrightarrow g'\ x \geq 0$
assumes $a \leq b$
shows $g\ a \leq g\ b$
by (*metis DERIV-nonneg-imp-nondecreasing atLeastAtMost-iff assms*)

Derivative of inverse function

lemma *DERIV-inverse-function*:
fixes $f\ g$ :: *real* $\Rightarrow$ *real*
assumes *der*: *DERIV* f $(g\ x) :> D$
and *neq*: $D \neq 0$
and x: $a < x\ x < b$

and *inj*: $\bigwedge y.\ [\![a < y;\ y < b]\!] \Longrightarrow f\ (g\ y) = y$
and *cont*: *isCont g x*
shows *DERIV g x :> inverse D*
unfolding *has-field-derivative-iff*
proof (*rule LIM-equal2*)
show $0 < min\ (x - a)\ (b - x)$
using *x* **by** *arith*
next
fix *y*
assume $norm\ (y - x) < min\ (x - a)\ (b - x)$
then have $a < y$ **and** $y < b$
by (*simp-all add*: *abs-less-iff*)
then show $(g\ y - g\ x)\ /\ (y - x) = inverse\ ((f\ (g\ y) - x)\ /\ (g\ y - g\ x))$
by (*simp add*: *inj*)
next
have $(\lambda z.\ (f\ z - f\ (g\ x))\ /\ (z - g\ x))\ -g\ x\rightarrow\ D$
by (*rule der* [*unfolded has-field-derivative-iff*])
then have *1*: $(\lambda z.\ (f\ z - x)\ /\ (z - g\ x))\ -g\ x\rightarrow\ D$
using *inj x* **by** *simp*
have *2*: $\exists\, d{>}0.\ \forall\, y.\ y \neq x \wedge norm\ (y - x) < d \longrightarrow g\ y \neq g\ x$
proof (*rule exI*, *safe*)
show $0 < min\ (x - a)\ (b - x)$
using *x* **by** *simp*
next
fix *y*
assume $norm\ (y - x) < min\ (x - a)\ (b - x)$
then have *y*: $a < y\ y < b$
by (*simp-all add*: *abs-less-iff*)
assume $g\ y = g\ x$
then have $f\ (g\ y) = f\ (g\ x)$ **by** *simp*
then have $y = x$ **using** *inj y x* **by** *simp*
also assume $y \neq x$
finally show *False* **by** *simp*
qed
have $(\lambda y.\ (f\ (g\ y) - x)\ /\ (g\ y - g\ x))\ -x\rightarrow\ D$
using *cont 1 2* **by** (*rule isCont-LIM-compose2*)
then show $(\lambda y.\ inverse\ ((f\ (g\ y) - x)\ /\ (g\ y - g\ x)))\ -x\rightarrow\ inverse\ D$
using *neq* **by** (*rule tendsto-inverse*)
qed

110.11 Generalized Mean Value Theorem

theorem *GMVT*:
fixes *a b* :: *real*
assumes *alb*: $a < b$
and *fc*: $\forall\, x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ f\ x$
and *fd*: $\forall\, x.\ a < x \wedge x < b \longrightarrow f\ differentiable\ (at\ x)$
and *gc*: $\forall\, x.\ a \leq x \wedge x \leq b \longrightarrow isCont\ g\ x$
and *gd*: $\forall\, x.\ a < x \wedge x < b \longrightarrow g\ differentiable\ (at\ x)$

shows $\exists g'c\ f'c\ c.$
$DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge a < c \wedge c < b \wedge (f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$
proof −
let $?h = \lambda x.\ (f\ b - f\ a) * g\ x - (g\ b - g\ a) * f\ x$
have $\exists l\ z.\ a < z \wedge z < b \wedge DERIV\ ?h\ z :> l \wedge ?h\ b - ?h\ a = (b - a) * l$
proof (*rule MVT*)
from *assms* **show** $a < b$ **by** *simp*
show *continuous-on* $\{a..b\}$ *?h*
by (*simp add*: *continuous-at-imp-continuous-on fc gc*)
show $\bigwedge x.\ [\![a < x; x < b]\!] \Longrightarrow$ *?h differentiable* (*at x*)
using *fd gd* **by** *simp*
qed
then obtain *l* **where** *l*: $\exists z.\ a < z \wedge z < b \wedge DERIV\ ?h\ z :> l \wedge ?h\ b - ?h\ a = (b - a) * l$ **..**
then obtain *c* **where** *c*: $a < c \wedge c < b \wedge DERIV\ ?h\ c :> l \wedge ?h\ b - ?h\ a = (b - a) * l$ **..**

from *c* **have** *cint*: $a < c \wedge c < b$ **by** *auto*
then obtain $g'c$ **where** $g'c$: $DERIV\ g\ c :> g'c$
using *gd real-differentiable-def* **by** *blast*
from *c* **have** $a < c \wedge c < b$ **by** *auto*
then obtain $f'c$ **where** $f'c$: $DERIV\ f\ c :> f'c$
using *fd real-differentiable-def* **by** *blast*

from *c* **have** $DERIV\ ?h\ c :> l$ **by** *auto*
moreover have $DERIV\ ?h\ c :>\ g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)$
using $g'c\ f'c$ **by** (*auto intro*!: *derivative-eq-intros*)
ultimately have *leq*: $l =\ g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)$ **by** (*rule DERIV-unique*)

have $?h\ b - ?h\ a = (b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a))$
proof −
from *c* **have** $?h\ b - ?h\ a = (b - a) * l$ **by** *auto*
also from *leq* **have** $\ldots = (b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a))$ **by** *simp*
finally show *?thesis* **by** *simp*
qed
moreover have $?h\ b - ?h\ a = 0$
proof −
have $?h\ b - ?h\ a =$
$((f\ b)*(g\ b) - (f\ a)*(g\ b) - (g\ b)*(f\ b) + (g\ a)*(f\ b)) -$
$((f\ b)*(g\ a) - (f\ a)*(g\ a) - (g\ b)*(f\ a) + (g\ a)*(f\ a))$
by (*simp add*: *algebra-simps*)
then show *?thesis* **by** *auto*
qed
ultimately have $(b - a) * (g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a)) = 0$ **by** *auto*
with *alb* **have** $g'c * (f\ b - f\ a) - f'c * (g\ b - g\ a) = 0$ **by** *simp*
then have $g'c * (f\ b - f\ a) = f'c * (g\ b - g\ a)$ **by** *simp*

then have $(f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$ **by** (*simp add*: *ac-simps*)
with *g'c f'c cint* **show** *?thesis* **by** *auto*
qed

lemma *GMVT'*:
fixes $f\ g :: real \Rightarrow real$
assumes $a < b$
and *isCont-f*: $\bigwedge z.\ a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ f\ z$
and *isCont-g*: $\bigwedge z.\ a \leq z \Longrightarrow z \leq b \Longrightarrow isCont\ g\ z$
and *DERIV-g*: $\bigwedge z.\ a < z \Longrightarrow z < b \Longrightarrow DERIV\ g\ z :> (g'\ z)$
and *DERIV-f*: $\bigwedge z.\ a < z \Longrightarrow z < b \Longrightarrow DERIV\ f\ z :> (f'\ z)$
shows $\exists c.\ a < c \wedge c < b \wedge (f\ b - f\ a) * g'\ c = (g\ b - g\ a) * f'\ c$
proof −
have $\exists g'c\ f'c\ c.\ DERIV\ g\ c :> g'c \wedge DERIV\ f\ c :> f'c \wedge$
$a < c \wedge c < b \wedge (f\ b - f\ a) * g'c = (g\ b - g\ a) * f'c$
using *assms* **by** (*intro GMVT*) (*force simp*: *real-differentiable-def*)+
then obtain *c* **where** $a < c\ c < b\ (f\ b - f\ a) * g'\ c = (g\ b - g\ a) * f'\ c$
using *DERIV-f DERIV-g* **by** (*force dest*: *DERIV-unique*)
then show *?thesis*
by *auto*
qed

110.12 L'Hopitals rule

lemma *isCont-If-ge*:
fixes $a :: 'a :: linorder\text{-}topology$
assumes *continuous* (*at-left a*) *g* **and** *f*: $(f \longrightarrow g\ a)$ (*at-right a*)
shows *isCont* $(\lambda x.\ if\ x \leq a\ then\ g\ x\ else\ f\ x)\ a$ (**is** *isCont ?gf a*)
proof −
have *g*: $(g \longrightarrow g\ a)$ (*at-left a*)
using *assms continuous-within* **by** *blast*
show *?thesis*
unfolding *isCont-def continuous-within*
proof (*intro filterlim-split-at*; *simp*)
show $(?gf \longrightarrow g\ a)$ (*at-left a*)
by (*subst filterlim-cong*[*OF refl refl*, **where** *g*=*g*]) (*simp-all add*: *eventually-at-filter less-le g*)
show $(?gf \longrightarrow g\ a)$ (*at-right a*)
by (*subst filterlim-cong*[*OF refl refl*, **where** *g*=*f*]) (*simp-all add*: *eventually-at-filter less-le f*)
qed
qed

lemma *lhopital-right-0*:
fixes *f0 g0* :: $real \Rightarrow real$
assumes *f-0*: $(f0 \longrightarrow 0)$ (*at-right 0*)
and *g-0*: $(g0 \longrightarrow 0)$ (*at-right 0*)
and *ev*:
eventually $(\lambda x.\ g0\ x \neq 0)$ (*at-right 0*)

```
      eventually (λx. g' x ≠ 0) (at-right 0)
      eventually (λx. DERIV f0 x :> f' x) (at-right 0)
      eventually (λx. DERIV g0 x :> g' x) (at-right 0)
    and lim: filterlim (λ x. (f' x / g' x)) F (at-right 0)
  shows filterlim (λ x. f0 x / g0 x) F (at-right 0)
proof −
  define f where [abs-def]: f x = (if x ≤ 0 then 0 else f0 x) for x
  then have f 0 = 0 by simp

  define g where [abs-def]: g x = (if x ≤ 0 then 0 else g0 x) for x
  then have g 0 = 0 by simp

  have eventually (λx. g0 x ≠ 0 ∧ g' x ≠ 0 ∧
      DERIV f0 x :> (f' x) ∧ DERIV g0 x :> (g' x)) (at-right 0)
    using ev by eventually-elim auto
  then obtain a where [arith]: 0 < a
    and g0-neq-0: ⋀x. 0 < x ⟹ x < a ⟹ g0 x ≠ 0
    and g'-neq-0: ⋀x. 0 < x ⟹ x < a ⟹ g' x ≠ 0
    and f0: ⋀x. 0 < x ⟹ x < a ⟹ DERIV f0 x :> (f' x)
    and g0: ⋀x. 0 < x ⟹ x < a ⟹ DERIV g0 x :> (g' x)
    unfolding eventually-at by (auto simp: dist-real-def)

  have g-neq-0: ⋀x. 0 < x ⟹ x < a ⟹ g x ≠ 0
    using g0-neq-0 by (simp add: g-def)

  have f: DERIV f x :> (f' x) if x: 0 < x x < a for x
    using that
    by (intro DERIV-cong-ev[THEN iffD1, OF - - - f0[OF x]])
      (auto simp: f-def eventually-nhds-metric dist-real-def intro!: exI[of - x])

  have g: DERIV g x :> (g' x) if x: 0 < x x < a for x
    using that
    by (intro DERIV-cong-ev[THEN iffD1, OF - - - g0[OF x]])
       (auto simp: g-def eventually-nhds-metric dist-real-def intro!: exI[of - x])

  have isCont f 0
    unfolding f-def by (intro isCont-If-ge f-0 continuous-const)

  have isCont g 0
    unfolding g-def by (intro isCont-If-ge g-0 continuous-const)

  have ∃ζ. ∀x∈{0 <..< a}. 0 < ζ x ∧ ζ x < x ∧ f x / g x = f' (ζ x) / g' (ζ x)
  proof (rule bchoice, rule ballI)
    fix x
    assume x ∈ {0 <..< a}
    then have x[arith]: 0 < x x < a by auto
    with g'-neq-0 g-neq-0 ‹g 0 = 0› have g': ⋀x. 0 < x ⟹ x < a  ⟹ 0 ≠ g' x
g 0 ≠ g x
      by auto
```

have $\bigwedge x.\ 0 \le x \Longrightarrow x < a \Longrightarrow$ *isCont f x*
using ‹*isCont f 0*› *f* **by** (*auto intro*: *DERIV-isCont simp*: *le-less*)
moreover have $\bigwedge x.\ 0 \le x \Longrightarrow x < a \Longrightarrow$ *isCont g x*
using ‹*isCont g 0*› *g* **by** (*auto intro*: *DERIV-isCont simp*: *le-less*)
ultimately have $\exists c.\ 0 < c \wedge c < x \wedge (f\ x - f\ 0) * g'\ c = (g\ x - g\ 0) * f'\ c$
using *f g* ‹$x < a$› **by** (*intro GMVT'*) *auto*
then obtain *c* **where** $*$: $0 < c\ c < x\ (f\ x - f\ 0) * g'\ c = (g\ x - g\ 0) * f'\ c$
by *blast*
moreover
from $*$ *g'(1)*[*of c*] *g'(2)* **have** $(f\ x - f\ 0)\ /\ (g\ x - g\ 0) = f'\ c\ /\ g'\ c$
by (*simp add*: *field-simps*)
ultimately show $\exists y.\ 0 < y \wedge y < x \wedge f\ x\ /\ g\ x = f'\ y\ /\ g'\ y$
using ‹$f\ 0 = 0$› ‹$g\ 0 = 0$› **by** (*auto intro*!: *exI*[*of - c*])
qed
then obtain ζ **where** $\forall x{\in}\{0 <..< a\}.\ 0 < \zeta\ x \wedge \zeta\ x < x \wedge f\ x\ /\ g\ x = f'\ (\zeta\ x)\ /\ g'\ (\zeta\ x)$ **..**
then have ζ: *eventually* $(\lambda x.\ 0 < \zeta\ x \wedge \zeta\ x < x \wedge f\ x\ /\ g\ x = f'\ (\zeta\ x)\ /\ g'\ (\zeta\ x))$ (*at-right 0*)
unfolding *eventually-at* **by** (*intro exI*[*of - a*]) (*auto simp*: *dist-real-def*)
moreover
from ζ **have** *eventually* $(\lambda x.\ norm\ (\zeta\ x) \le x)$ (*at-right 0*)
by *eventually-elim auto*
then have $((\lambda x.\ norm\ (\zeta\ x)) \longrightarrow 0)$ (*at-right 0*)
by (*rule-tac real-tendsto-sandwich*[**where** $f{=}\lambda x.\ 0$ **and** $h{=}\lambda x.\ x$]) *auto*
then have $(\zeta \longrightarrow 0)$ (*at-right 0*)
by (*rule tendsto-norm-zero-cancel*)
with ζ **have** *filterlim* ζ (*at-right 0*) (*at-right 0*)
by (*auto elim*!: *eventually-mono simp*: *filterlim-at*)
from *this lim* **have** *filterlim* $(\lambda t.\ f'\ (\zeta\ t)\ /\ g'\ (\zeta\ t))\ F$ (*at-right 0*)
by (*rule-tac filterlim-compose*[*of - - - ζ*])
ultimately have *filterlim* $(\lambda t.\ f\ t\ /\ g\ t)\ F$ (*at-right 0*) (**is** *?P*)
by (*rule-tac filterlim-cong*[*THEN iffD1*, *OF refl refl*])
(*auto elim*: *eventually-mono*)
also have *?P* $\longleftrightarrow$ *?thesis*
by (*rule filterlim-cong*) (*auto simp*: *f-def g-def eventually-at-filter*)
finally show *?thesis* **.**
qed

lemma *lhopital-right*:
$(f \longrightarrow 0)$ (*at-right x*) $\Longrightarrow$ $(g \longrightarrow 0)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ g\ x \neq 0)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ g'\ x \neq 0)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ f\ x :> f'\ x)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ g\ x :> g'\ x)$ (*at-right x*) $\Longrightarrow$
filterlim $(\lambda\ x.\ (f'\ x\ /\ g'\ x))\ F$ (*at-right x*) $\Longrightarrow$
filterlim $(\lambda\ x.\ f\ x\ /\ g\ x)\ F$ (*at-right x*)
for *x* :: *real*
unfolding *eventually-at-right-to-0*[*of - x*] *filterlim-at-right-to-0*[*of - - x*] *DERIV-shift*

by (*rule lhopital-right-0*)

lemma *lhopital-left*:
(*f* ⟶ *0*) (*at-left x*) ⟹ (*g* ⟶ *0*) (*at-left x*) ⟹
eventually (λ*x*. *g x* ≠ *0*) (*at-left x*) ⟹
eventually (λ*x*. *g′ x* ≠ *0*) (*at-left x*) ⟹
eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at-left x*) ⟹
eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at-left x*) ⟹
filterlim (λ *x*. (*f′ x* / *g′ x*)) *F* (*at-left x*) ⟹
filterlim (λ *x*. *f x* / *g x*) *F* (*at-left x*)
for *x* :: *real*
unfolding *eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror*
by (*rule lhopital-right*[**where** *f′*=λ*x*. − *f′* (− *x*)]) (*auto simp*: *DERIV-mirror*)

lemma *lhopital*:
(*f* ⟶ *0*) (*at x*) ⟹ (*g* ⟶ *0*) (*at x*) ⟹
eventually (λ*x*. *g x* ≠ *0*) (*at x*) ⟹
eventually (λ*x*. *g′ x* ≠ *0*) (*at x*) ⟹
eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at x*) ⟹
eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at x*) ⟹
filterlim (λ *x*. (*f′ x* / *g′ x*)) *F* (*at x*) ⟹
filterlim (λ *x*. *f x* / *g x*) *F* (*at x*)
for *x* :: *real*
unfolding *eventually-at-split filterlim-at-split*
by (*auto intro*!: *lhopital-right*[*of f x g g′ f′*] *lhopital-left*[*of f x g g′ f′*])

lemma *lhopital-right-0-at-top*:
fixes *f g* :: *real* ⇒ *real*
assumes *g-0*: *LIM x at-right 0*. *g x* :> *at-top*
and *ev*:
eventually (λ*x*. *g′ x* ≠ *0*) (*at-right 0*)
eventually (λ*x*. *DERIV f x* :> *f′ x*) (*at-right 0*)
eventually (λ*x*. *DERIV g x* :> *g′ x*) (*at-right 0*)
and *lim*: ((λ *x*. (*f′ x* / *g′ x*)) ⟶ *x*) (*at-right 0*)
shows ((λ *x*. *f x* / *g x*) ⟶ *x*) (*at-right 0*)
unfolding *tendsto-iff*
proof *safe*
fix *e* :: *real*
assume *0* < *e*
with *lim*[*unfolded tendsto-iff*, *rule-format*, *of e* / *4*]
have *eventually* (λ*t*. *dist* (*f′ t* / *g′ t*) *x* < *e* / *4*) (*at-right 0*)
by *simp*
from *eventually-conj*[*OF eventually-conj*[*OF ev*(*1*) *ev*(*2*)] *eventually-conj*[*OF ev*(*3*) *this*]]
obtain *a* **where** [*arith*]: *0* < *a*
and *g′-neq-0*: ⋀*x*. *0* < *x* ⟹ *x* < *a* ⟹ *g′ x* ≠ *0*
and *f0*: ⋀*x*. *0* < *x* ⟹ *x* ≤ *a* ⟹ *DERIV f x* :> (*f′ x*)
and *g0*: ⋀*x*. *0* < *x* ⟹ *x* ≤ *a* ⟹ *DERIV g x* :> (*g′ x*)

and *Df*: $\bigwedge t.\ 0 < t \Longrightarrow t < a \Longrightarrow dist\ (f'\ t\ /\ g'\ t)\ x < e\ /\ 4$
unfolding *eventually-at-le* **by** (*auto simp*: *dist-real-def*)

from *Df* **have** *eventually* $(\lambda t.\ t < a)$ (*at-right 0*) *eventually* $(\lambda t{::}real.\ 0 < t)$ (*at-right 0*)
unfolding *eventually-at* **by** (*auto intro*!: *exI*[*of* - *a*] *simp*: *dist-real-def*)

moreover
have *eventually* $(\lambda t.\ 0 < g\ t)$ (*at-right 0*) *eventually* $(\lambda t.\ g\ a < g\ t)$ (*at-right 0*)
using *g-0* **by** (*auto elim*: *eventually-mono simp*: *filterlim-at-top-dense*)

moreover
have *inv-g*: $((\lambda x.\ inverse\ (g\ x)) \longrightarrow 0)$ (*at-right 0*)
using *tendsto-inverse-0 filterlim-mono*[*OF g-0 at-top-le-at-infinity order-refl*]
by (*rule filterlim-compose*)
then have $((\lambda x.\ norm\ (1 - g\ a * inverse\ (g\ x))) \longrightarrow norm\ (1 - g\ a * 0))$ (*at-right 0*)
by (*intro tendsto-intros*)
then have $((\lambda x.\ norm\ (1 - g\ a\ /\ g\ x)) \longrightarrow 1)$ (*at-right 0*)
by (*simp add*: *inverse-eq-divide*)
from *this*[*unfolded tendsto-iff*, *rule-format*, *of 1*]
have *eventually* $(\lambda x.\ norm\ (1 - g\ a\ /\ g\ x) < 2)$ (*at-right 0*)
by (*auto elim*!: *eventually-mono simp*: *dist-real-def*)

moreover
from *inv-g* **have** $((\lambda t.\ norm\ ((f\ a - x * g\ a) * inverse\ (g\ t))) \longrightarrow norm\ ((f\ a - x * g\ a) * 0))$
(*at-right 0*)
by (*intro tendsto-intros*)
then have $((\lambda t.\ norm\ (f\ a - x * g\ a)\ /\ norm\ (g\ t)) \longrightarrow 0)$ (*at-right 0*)
by (*simp add*: *inverse-eq-divide*)
from *this*[*unfolded tendsto-iff*, *rule-format*, *of e / 2*] ‹*0 < e*›
have *eventually* $(\lambda t.\ norm\ (f\ a - x * g\ a)\ /\ norm\ (g\ t) < e\ /\ 2)$ (*at-right 0*)
by (*auto simp*: *dist-real-def*)

ultimately show *eventually* $(\lambda t.\ dist\ (f\ t\ /\ g\ t)\ x < e)$ (*at-right 0*)
proof *eventually-elim*
fix *t* **assume** *t*[*arith*]: *0 < t t < a g a < g t 0 < g t*
assume *ineq*: *norm* $(1 - g\ a\ /\ g\ t) < 2$ *norm* $(f\ a - x * g\ a)\ /\ norm\ (g\ t) < e\ /\ 2$

have $\exists y.\ t < y \wedge y < a \wedge (g\ a - g\ t) * f'\ y = (f\ a - f\ t) * g'\ y$
using *f0 g0 t*(*1*,*2*) **by** (*intro GMVT'*) (*force intro*!: *DERIV-isCont*)+
then obtain *y* **where** [*arith*]: *t < y y < a*
and *D-eq0*: $(g\ a - g\ t) * f'\ y = (f\ a - f\ t) * g'\ y$
by *blast*
from *D-eq0* **have** *D-eq*: $(f\ t - f\ a)\ /\ (g\ t - g\ a) = f'\ y\ /\ g'\ y$
using ‹*g a < g t*› *g'-neq-0*[*of y*] **by** (*auto simp add*: *field-simps*)

have $*$: $f\ t\ /\ g\ t - x = ((f\ t - f\ a)\ /\ (g\ t - g\ a) - x) * (1 - g\ a\ /\ g\ t) + (f\ a - x * g\ a)\ /\ g\ t$
by (*simp add*: *field-simps*)
have *norm* $(f\ t\ /\ g\ t - x) \leq$
norm $(((f\ t - f\ a)\ /\ (g\ t - g\ a) - x) * (1 - g\ a\ /\ g\ t)) + norm\ ((f\ a - x * g\ a)\ /\ g\ t)$
unfolding $*$ **by** (*rule norm-triangle-ineq*)
also have $\ldots = dist\ (f'\ y\ /\ g'\ y)\ x * norm\ (1 - g\ a\ /\ g\ t) + norm\ (f\ a - x * g\ a)\ /\ norm\ (g\ t)$
by (*simp add*: *abs-mult D-eq dist-real-def*)
also have $\ldots < (e\ /\ 4) * 2 + e\ /\ 2$
using *ineq Df*[*of y*] ‹$0 < e$› **by** (*intro add-le-less-mono mult-mono*) *auto*
finally show *dist* $(f\ t\ /\ g\ t)\ x < e$
by (*simp add*: *dist-real-def*)
qed
qed

lemma *lhopital-right-at-top*:
LIM x at-right x. $(g::real \Rightarrow real)\ x :>$ *at-top* $\Longrightarrow$
eventually $(\lambda x.\ g'\ x \neq 0)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ f\ x :> f'\ x)$ (*at-right x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ g\ x :> g'\ x)$ (*at-right x*) $\Longrightarrow$
$((\lambda\ x.\ (f'\ x\ /\ g'\ x)) \longrightarrow y)$ (*at-right x*) $\Longrightarrow$
$((\lambda\ x.\ f\ x\ /\ g\ x) \longrightarrow y)$ (*at-right x*)
unfolding *eventually-at-right-to-0*[*of - x*] *filterlim-at-right-to-0*[*of - - x*] *DERIV-shift*
by (*rule lhopital-right-0-at-top*)

lemma *lhopital-left-at-top*:
LIM x at-left x. g x :> at-top $\Longrightarrow$
eventually $(\lambda x.\ g'\ x \neq 0)$ (*at-left x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ f\ x :> f'\ x)$ (*at-left x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ g\ x :> g'\ x)$ (*at-left x*) $\Longrightarrow$
$((\lambda\ x.\ (f'\ x\ /\ g'\ x)) \longrightarrow y)$ (*at-left x*) $\Longrightarrow$
$((\lambda\ x.\ f\ x\ /\ g\ x) \longrightarrow y)$ (*at-left x*)
for *x* :: *real*
unfolding *eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror*
by (*rule lhopital-right-at-top*[**where** f'=$\lambda x.\ -f'\ (-x)$]) (*auto simp*: *DERIV-mirror*)

lemma *lhopital-at-top*:
LIM x at x. $(g::real \Rightarrow real)\ x :>$ *at-top* $\Longrightarrow$
eventually $(\lambda x.\ g'\ x \neq 0)$ (*at x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ f\ x :> f'\ x)$ (*at x*) $\Longrightarrow$
eventually $(\lambda x.\ DERIV\ g\ x :> g'\ x)$ (*at x*) $\Longrightarrow$
$((\lambda\ x.\ (f'\ x\ /\ g'\ x)) \longrightarrow y)$ (*at x*) $\Longrightarrow$
$((\lambda\ x.\ f\ x\ /\ g\ x) \longrightarrow y)$ (*at x*)
unfolding *eventually-at-split filterlim-at-split*
by (*auto intro*!: *lhopital-right-at-top*[*of g x g' f f'*] *lhopital-left-at-top*[*of g x g' f f'*])

lemma *lhospital-at-top-at-top*:
fixes *f g* :: *real* ⇒ *real*
assumes *g-0*: *LIM x at-top. g x :> at-top*
and *g'*: *eventually* (λ*x*. *g' x* ≠ *0*) *at-top*
and *Df*: *eventually* (λ*x*. *DERIV f x :> f' x*) *at-top*
and *Dg*: *eventually* (λ*x*. *DERIV g x :> g' x*) *at-top*
and *lim*: ((λ *x*. (*f' x* / *g' x*)) ⟶ *x*) *at-top*
shows ((λ *x*. *f x* / *g x*) ⟶ *x*) *at-top*
unfolding *filterlim-at-top-to-right*
proof (*rule lhopital-right-0-at-top*)
let *?F* = λ*x*. *f* (*inverse x*)
let *?G* = λ*x*. *g* (*inverse x*)
let *?R* = *at-right* (*0*::*real*)
let *?D* = λ*f' x*. *f'* (*inverse x*) ∗ − (*inverse x* ^ *Suc* (*Suc 0*))
show *LIM x ?R*. *?G x :> at-top*
using *g-0* **unfolding** *filterlim-at-top-to-right* **.**
show *eventually* (λ*x*. *DERIV ?G x :> ?D g' x*) *?R*
unfolding *eventually-at-right-to-top*
using *Dg eventually-ge-at-top*[**where** *c=1*]
by *eventually-elim* (*rule derivative-eq-intros DERIV-chain'*[**where** *f=inverse*]
| *simp*)+
show *eventually* (λ*x*. *DERIV ?F x :> ?D f' x*) *?R*
unfolding *eventually-at-right-to-top*
using *Df eventually-ge-at-top*[**where** *c=1*]
by *eventually-elim* (*rule derivative-eq-intros DERIV-chain'*[**where** *f=inverse*]
| *simp*)+
show *eventually* (λ*x*. *?D g' x* ≠ *0*) *?R*
unfolding *eventually-at-right-to-top*
using *g' eventually-ge-at-top*[**where** *c=1*]
by *eventually-elim auto*
show ((λ*x*. *?D f' x* / *?D g' x*) ⟶ *x*) *?R*
unfolding *filterlim-at-right-to-top*
apply (*intro filterlim-cong*[*THEN iffD2*, *OF refl refl - lim*])
using *eventually-ge-at-top*[**where** *c=1*]
by *eventually-elim simp*
qed

lemma *lhopital-right-at-top-at-top*:
fixes *f g* :: *real* ⇒ *real*
assumes *f-0*: *LIM x at-right a. f x :> at-top*
assumes *g-0*: *LIM x at-right a. g x :> at-top*
and *ev*:
eventually (λ*x*. *DERIV f x :> f' x*) (*at-right a*)
eventually (λ*x*. *DERIV g x :> g' x*) (*at-right a*)
and *lim*: *filterlim* (λ *x*. (*f' x* / *g' x*)) *at-top* (*at-right a*)
shows *filterlim* (λ *x*. *f x* / *g x*) *at-top* (*at-right a*)
proof −
from *lim* **have** *pos*: *eventually* (λ*x*. *f' x* / *g' x* > *0*) (*at-right a*)

```
    unfolding filterlim-at-top-dense by blast
  have ((λx. g x / f x) ⟶ 0) (at-right a)
  proof (rule lhopital-right-at-top)
    from pos show eventually (λx. f' x ≠ 0) (at-right a) by eventually-elim auto
    from tendsto-inverse-0-at-top[OF lim]
      show ((λx. g' x / f' x) ⟶ 0) (at-right a) by simp
  qed fact+
  moreover from f-0 g-0
    have eventually (λx. f x > 0) (at-right a) eventually (λx. g x > 0) (at-right a)
    unfolding filterlim-at-top-dense by blast+
  hence eventually (λx. g x / f x > 0) (at-right a) by eventually-elim simp
  ultimately have filterlim (λx. inverse (g x / f x)) at-top (at-right a)
    by (rule filterlim-inverse-at-top)
  thus ?thesis by simp
qed

lemma lhopital-right-at-top-at-bot:
  fixes f g :: real ⇒ real
  assumes f-0: LIM x at-right a. f x :> at-top
  assumes g-0: LIM x at-right a. g x :> at-bot
    and ev:
      eventually (λx. DERIV f x :> f' x) (at-right a)
      eventually (λx. DERIV g x :> g' x) (at-right a)
    and lim: filterlim (λ x. (f' x / g' x)) at-bot (at-right a)
  shows filterlim (λ x. f x / g x) at-bot (at-right a)
proof −
  from ev(2) have ev': eventually (λx. DERIV (λx. −g x) x :> −g' x) (at-right
a)
    by eventually-elim (auto intro: derivative-intros)
  have filterlim (λx. f x / (−g x)) at-top (at-right a)
    by (rule lhopital-right-at-top-at-top[where f' = f' and g' = λx. −g' x])
       (insert assms ev', auto simp: filterlim-uminus-at-bot)
  hence filterlim (λx. −(f x / g x)) at-top (at-right a) by simp
  thus ?thesis by (simp add: filterlim-uminus-at-bot)
qed

lemma lhopital-left-at-top-at-top:
  fixes f g :: real ⇒ real
  assumes f-0: LIM x at-left a. f x :> at-top
  assumes g-0: LIM x at-left a. g x :> at-top
    and ev:
      eventually (λx. DERIV f x :> f' x) (at-left a)
      eventually (λx. DERIV g x :> g' x) (at-left a)
    and lim: filterlim (λ x. (f' x / g' x)) at-top (at-left a)
  shows filterlim (λ x. f x / g x) at-top (at-left a)
  by (insert assms, unfold eventually-at-left-to-right filterlim-at-left-to-right DE-
RIV-mirror,
      rule lhopital-right-at-top-at-top[where f'=λx. − f' (− x)])
     (insert assms, auto simp: DERIV-mirror)
```

lemma *lhopital-left-at-top-at-bot*:
fixes *f g* :: *real* ⇒ *real*
assumes *f-0*: *LIM x at-left a. f x :> at-top*
assumes *g-0*: *LIM x at-left a. g x :> at-bot*
and *ev*:
eventually (λ*x*. *DERIV f x :> f' x*) (*at-left a*)
eventually (λ*x*. *DERIV g x :> g' x*) (*at-left a*)
and *lim*: *filterlim* (λ *x*. (*f' x* / *g' x*)) *at-bot* (*at-left a*)
shows *filterlim* (λ *x*. *f x* / *g x*) *at-bot* (*at-left a*)
by (*insert assms*, *unfold eventually-at-left-to-right filterlim-at-left-to-right DERIV-mirror*,
rule lhopital-right-at-top-at-bot[**where** *f'*=λ*x*. − *f'* (− *x*)])
(*insert assms*, *auto simp*: *DERIV-mirror*)

lemma *lhopital-at-top-at-top*:
fixes *f g* :: *real* ⇒ *real*
assumes *f-0*: *LIM x at a. f x :> at-top*
assumes *g-0*: *LIM x at a. g x :> at-top*
and *ev*:
eventually (λ*x*. *DERIV f x :> f' x*) (*at a*)
eventually (λ*x*. *DERIV g x :> g' x*) (*at a*)
and *lim*: *filterlim* (λ *x*. (*f' x* / *g' x*)) *at-top* (*at a*)
shows *filterlim* (λ *x*. *f x* / *g x*) *at-top* (*at a*)
using *assms* **unfolding** *eventually-at-split filterlim-at-split*
by (*auto intro*!: *lhopital-right-at-top-at-top*[*of f a g f' g'*]
lhopital-left-at-top-at-top[*of f a g f' g'*])

lemma *lhopital-at-top-at-bot*:
fixes *f g* :: *real* ⇒ *real*
assumes *f-0*: *LIM x at a. f x :> at-top*
assumes *g-0*: *LIM x at a. g x :> at-bot*
and *ev*:
eventually (λ*x*. *DERIV f x :> f' x*) (*at a*)
eventually (λ*x*. *DERIV g x :> g' x*) (*at a*)
and *lim*: *filterlim* (λ *x*. (*f' x* / *g' x*)) *at-bot* (*at a*)
shows *filterlim* (λ *x*. *f x* / *g x*) *at-bot* (*at a*)
using *assms* **unfolding** *eventually-at-split filterlim-at-split*
by (*auto intro*!: *lhopital-right-at-top-at-bot*[*of f a g f' g'*]
lhopital-left-at-top-at-bot[*of f a g f' g'*])

110.13 Logarithmic derivatives

definition *has-log-derivative* :: ('*a*::*real-normed-field* ⇒ '*a*) ⇒ '*a* ⇒ '*a filter* ⇒ *bool*
(**infix** ‹(*has'-log'-derivative*)› *50*)
where (*f has-log-derivative D*) *F* ⟷
(*F* = *bot* ∨ *f* (*Lim F* (λ*x*. *x*)) ≠ *0*) ∧ (*f has-field-derivative* (*D* ∗ *f* (*Lim F* (λ*x*. *x*)))) *F*

lemma *has-log-derivative-bot* [*simp*, *intro*]: (*f has-log-derivative f'*) *bot*
by (*auto simp*: *has-log-derivative-def*)

lemma *has-log-derivative-unique*:
assumes (*f has-log-derivative D1*) (*at x within A*) (*f has-log-derivative D2*) (*at x within A*)
assumes *at x within A* ≠ *bot*
shows *D1* = *D2*
proof −
from *assms* **have** *D1* ∗ *f* (*Lim* (*at x within A*) (λ*x*. *x*)) = *D2* ∗ *f* (*Lim* (*at x within A*) (λ*x*. *x*))
using *assms has-field-derivative-unique* **unfolding** *has-log-derivative-def* **by** *blast*
with *assms* **show** *D1* = *D2*
by (*auto simp*: *has-log-derivative-def*)
qed

lemma *has-log-derivative-cong-ev'*:
assumes *eventually* (λ*x*. *f x* = *g x*) (*nhds x*)
shows (*f has-log-derivative f'*) (*at x*) ⟷
(*g has-log-derivative f'*) (*at x*)
proof −
from *assms*(*1*) **have** [*simp*]: *f x* = *g x*
using *eventually-nhds-x-imp-x* **by** *blast*
show *?thesis*
unfolding *has-log-derivative-def*
apply (*intro conj-cong*)
using *DERIV-cong-ev*[*OF refl assms refl*] *assms*
apply (*simp-all add*: *Lim-ident-at*)
done
qed

lemma *has-log-derivative-cong-ev*:
assumes *eventually* (λ*x*. *f x* = *g x*) (*at x within A*) *f x* = *g x*
shows (*f has-log-derivative f'*) (*at x within A*) ⟷
(*g has-log-derivative f'*) (*at x within A*)
unfolding *has-log-derivative-def*
apply (*cases at x within A* = *bot*; *intro conj-cong*)
using *has-field-derivative-cong-eventually*[*OF assms*] *assms*
apply (*simp-all add*: *Lim-ident-at*)
done

lemma *has-log-derivative-cong* [*derivative-cong*]:
(*f has-log-derivative X*) *F* ⟹ *X* = *Y* ⟹ (*f has-log-derivative Y*) *F*
by *simp*

lemma *has-log-derivative-imp-has-field-derivative*:
assumes (*f has-log-derivative* (*f'* / *f x*)) (*at x within A*)

shows (*f has-field-derivative f'*) (*at x within A*)
using *assms* **by** (*cases at x within A = bot*) (*auto simp*: *has-log-derivative-def Lim-ident-at*)

lemma *has-log-derivative-imp-has-field-derivative'*:
assumes (*f has-log-derivative f'*) (*at x within A*)
shows (*f has-field-derivative* (*f' * f x*)) (*at x within A*)
using *assms* **by** (*cases at x within A = bot*) (*auto simp*: *has-log-derivative-def Lim-ident-at*)

lemma *has-log-derivative-imp-nonzero*:
assumes (*f has-log-derivative f'*) (*at x within A*) *at x within A* ≠ *bot*
shows *f x* ≠ *0*
using *assms* **unfolding** *has-log-derivative-def* **by** (*auto simp*: *Lim-ident-at*)

lemma *has-field-derivative-imp-has-log-derivative*:
assumes (*f has-field-derivative* (*f' * f x*)) (*at x within A*) *f x* ≠ *0*
shows (*f has-log-derivative f'*) (*at x within A*)
using *assms* **by** (*cases at x within A = bot*) (*auto simp*: *has-log-derivative-def Lim-ident-at*)

lemma *has-field-derivative-imp-has-log-derivative'*:
assumes (*f has-field-derivative f'*) (*at x within A*) *f x* ≠ *0*
shows (*f has-log-derivative* (*f' / f x*)) (*at x within A*)
using *assms* **by** (*cases at x within A = bot*) (*auto simp*: *has-log-derivative-def Lim-ident-at*)

lemma *has-log-derivative-chain*:
assumes (*f has-log-derivative f'*) (*at* (*g x*))
assumes (*g has-field-derivative g'*) (*at x within A*)
shows ((*f* ∘ *g*) *has-log-derivative* (*f' * g'*)) (*at x within A*)
proof (*cases at x within A = bot*)
case *False*
hence [*simp*]: *Lim* (*at x within A*) (λ*x*. *x*) = *x*
by (*simp add*: *Lim-ident-at*)
show *?thesis*
unfolding *has-log-derivative-def*
proof
have ((*f* ∘ *g*) *has-field-derivative f' * f* (*g x*) * *g'*) (*at x within A*)
by (*rule DERIV-chain*) (*use assms False* **in** ‹*auto simp*: *has-log-derivative-def Lim-ident-at*›)
thus (*f* ∘ *g has-field-derivative* (*f' * g'*) * (*f* ∘ *g*) (*Lim* (*at x within A*) (λ*x*. *x*))) (*at x within A*)
by (*simp add*: *mult-ac*)
qed (*use assms* **in** ‹*auto simp*: *has-log-derivative-def Lim-ident-at*›)
qed *auto*

lemma *has-log-derivative-const* [*intro*, *derivative-intros*]:
c ≠ *0* ⟹ ((λ-. *c*) *has-log-derivative 0*) (*at x within A*)

by (*auto simp*: *has-log-derivative-def intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-uminus* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
shows ((λ*x*. −*f x*) *has-log-derivative f'*) (*at x within A*)
using *assms*
by (*cases at x within A = bot*)
(*auto simp*: *has-log-derivative-def algebra-simps Lim-ident-at intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-mult* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
assumes ((λ*x*. *g x*) *has-log-derivative g'*) (*at x within A*)
shows ((λ*x*. *f x* ∗ *g x*) *has-log-derivative* (*f'* + *g'*)) (*at x within A*)
using *assms*
by (*cases at x within A = bot*)
(*auto simp*: *has-log-derivative-def algebra-simps Lim-ident-at intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-inverse* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
shows ((λ*x*. *inverse* (*f x*)) *has-log-derivative* (−*f'*)) (*at x within A*)
using *assms*
by (*cases at x within A = bot*)
(*auto simp*: *has-log-derivative-def algebra-simps Lim-ident-at intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-divide* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
assumes ((λ*x*. *g x*) *has-log-derivative g'*) (*at x within A*)
shows ((λ*x*. *f x* / *g x*) *has-log-derivative* (*f'* − *g'*)) (*at x within A*)
using *assms*
by (*cases at x within A = bot*)
(*auto simp*: *has-log-derivative-def field-simps Lim-ident-at intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-power* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
shows ((λ*x*. *f x* ^ *n*) *has-log-derivative* (*of-nat n* ∗ *f'*)) (*at x within A*)
using *assms* **by** (*induction n*) (*auto intro*!: *derivative-eq-intros simp*: *algebra-simps*)

lemma *has-log-derivative-power-int* [*derivative-intros*]:
assumes [*derivative-intros*]: ((λ*x*. *f x*) *has-log-derivative f'*) (*at x within A*)
shows ((λ*x*. *f x powi n*) *has-log-derivative* (*of-int n* ∗ *f'*)) (*at x within A*)
by (*cases n* ≥ *0*) (*auto simp*: *power-int-def intro*!: *derivative-eq-intros*)

lemma *has-log-derivative-prod* [*derivative-intros*]:
assumes ⋀*i*. *i* ∈ *I* ⟹ (*f i has-log-derivative f' i*) (*at x within A*)
shows ((λ*x*. ∏ *i*∈*I*. *f i x*) *has-log-derivative* (∑ *i*∈*I*. *f' i*)) (*at x within A*)
using *assms* **by** (*induction I rule*: *infinite-finite-induct*) (*auto intro*!: *deriva-*

tive-eq-intros)

lemma *has-log-derivative-prod-list* [*derivative-intros*]:
assumes $\bigwedge i.\ i \in set\ is \Longrightarrow (f\ i\ has\text{-}log\text{-}derivative\ f'\ i)\ (at\ x\ within\ A)$
shows $((\lambda x.\ \prod i{\leftarrow}is.\ f\ i\ x)\ has\text{-}log\text{-}derivative\ (\sum i{\leftarrow}is.\ f'\ i))\ (at\ x\ within\ A)$
using *assms* **by** (*induction is*) (*auto intro*!: *derivative-eq-intros*)

end

111 Nth Roots of Real Numbers

theory *NthRoot*
imports *Deriv*
begin

111.1 Existence of Nth Root

Existence follows from the Intermediate Value Theorem

lemma *realpow-pos-nth*:
fixes $a :: real$
assumes n: $0 < n$
and a: $0 < a$
shows $\exists r{>}0.\ r\ \hat{}\ n = a$
proof −
have $\exists r{\geq}0.\ r \leq (max\ 1\ a) \wedge r\ \hat{}\ n = a$
proof (*rule IVT*)
show $0\ \hat{}\ n \leq a$
using $n\ a$ **by** (*simp add*: *power-0-left*)
show $0 \leq max\ 1\ a$
by *simp*
from n **have** $n1$: $1 \leq n$
by *simp*
have $a \leq max\ 1\ a\ \hat{}\ 1$
by *simp*
also have $max\ 1\ a\ \hat{}\ 1 \leq max\ 1\ a\ \hat{}\ n$
using $n1$ **by** (*rule power-increasing*) *simp*
finally show $a \leq max\ 1\ a\ \hat{}\ n$ **.**
show $\forall r.\ 0 \leq r \wedge r \leq max\ 1\ a \longrightarrow isCont\ (\lambda x.\ x\ \hat{}\ n)\ r$
by *simp*
qed
then obtain r **where** r: $0 \leq r \wedge r\ \hat{}\ n = a$
by *fast*
with $n\ a$ **have** $r \neq 0$
by (*auto simp add*: *power-0-left*)
with r **have** $0 < r \wedge r\ \hat{}\ n = a$
by *simp*
then show *?thesis* **..**
qed

lemma *realpow-pos-nth2*: $(0::real) < a \Longrightarrow \exists r>0.\ r \mathbin{\hat{}} Suc\ n = a$
by (*blast intro*: *realpow-pos-nth*)

Uniqueness of nth positive root.

lemma *realpow-pos-nth-unique*: $0 < n \Longrightarrow 0 < a \Longrightarrow \exists!r.\ 0 < r \wedge r \mathbin{\hat{}} n = a$ **for** $a :: real$
by (*auto intro*!: *realpow-pos-nth simp*: *power-eq-iff-eq-base*)

111.2 Nth Root

We define roots of negative reals such that $root\ n\ (-x) = -\ root\ n\ x$. This allows us to omit side conditions from many theorems.

lemma *inj-sgn-power*:
assumes $0 < n$
shows *inj* $(\lambda y.\ sgn\ y * |y| \mathbin{\hat{}} n :: real)$
(**is** *inj ?f*)
proof (*rule injI*)
have *x*: $(0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b) \Longrightarrow a \neq b$ **for** $a\ b :: real$
by *auto*
fix *x y*
assume *?f x = ?f y*
with *power-eq-iff-eq-base*[*of n* |*x*| |*y*|] ‹$0 < n$› **show** $x = y$
by (*cases rule*: *linorder-cases*[*of 0 x*, *case-product linorder-cases*[*of 0 y*]])
(*simp-all add*: *x*)
qed

lemma *sgn-power-injE*:
$sgn\ a * |a| \mathbin{\hat{}} n = x \Longrightarrow x = sgn\ b * |b| \mathbin{\hat{}} n \Longrightarrow 0 < n \Longrightarrow a = b$
for $a\ b :: real$
using *inj-sgn-power*[*THEN injD*, *of n a b*] **by** *simp*

definition *root* :: $nat \Rightarrow real \Rightarrow real$
where $root\ n\ x = (if\ n = 0\ then\ 0\ else\ the\text{-}inv\ (\lambda y.\ sgn\ y * |y| \mathbin{\hat{}} n)\ x)$

lemma *root-0* [*simp*]: $root\ 0\ x = 0$
by (*simp add*: *root-def*)

lemma *root-sgn-power*: $0 < n \Longrightarrow root\ n\ (sgn\ y * |y| \mathbin{\hat{}} n) = y$
using *the-inv-f-f*[*OF inj-sgn-power*] **by** (*simp add*: *root-def*)

lemma *sgn-power-root*:
assumes $0 < n$
shows $sgn\ (root\ n\ x) * |(root\ n\ x)| \mathbin{\hat{}} n = x$
(**is** *?f* $(root\ n\ x) = x$)
proof (*cases* $x = 0$)
case *True*

with *assms root-sgn-power*[*of n 0*] **show** *?thesis*
by *simp*
next
case *False*
with *realpow-pos-nth*[*OF* ‹*0 < n*›, *of* |*x*|]
obtain *r* **where** *0 < r r* ^ *n* = |*x*|
by *auto*
with ‹*x* ≠ *0*› **have** *S*: *x* ∈ *range ?f*
by (*intro image-eqI*[*of - - sgn x* ∗ *r*])
(*auto simp*: *abs-mult sgn-mult power-mult-distrib abs-sgn-eq mult-sgn-abs*)
from ‹*0 < n*› *f-the-inv-into-f*[*OF inj-sgn-power*[*OF* ‹*0 < n*›] *this*] **show** *?thesis*
by (*simp add*: *root-def*)
qed

lemma *split-root*: *P* (*root n x*) ⟷ (*n* = *0* ⟶ *P 0*) ∧ (*0 < n* ⟶ (∀ *y*. *sgn y* ∗ |*y*| ^ *n* = *x* ⟶ *P y*))
proof (*cases n* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then show *?thesis*
by *simp* (*metis root-sgn-power sgn-power-root*)
qed

lemma *real-root-zero* [*simp*]: *root n 0* = *0*
by (*simp split*: *split-root add*: *sgn-zero-iff*)

lemma *real-root-minus*: *root n* (− *x*) = − *root n x*
by (*clarsimp split*: *split-root elim*!: *sgn-power-injE simp*: *sgn-minus*)

lemma *real-root-less-mono*: *0 < n* ⟹ *x < y* ⟹ *root n x < root n y*
proof (*clarsimp split*: *split-root*)
have ∗: *0 < b* ⟹ *a < 0* ⟹ ¬ *a > b* **for** *a b* :: *real*
by *auto*
fix *a b* :: *real*
assume *0 < n sgn a* ∗ |*a*| ^ *n < sgn b* ∗ |*b*| ^ *n*
then show *a < b*
using *power-less-imp-less-base*[*of a n b*]
power-less-imp-less-base[*of* − *b n* − *a*]
by (*simp add*: *sgn-real-def* ∗ [*of a* ^ *n* − ((−) ^ *n*)]
split: *if-split-asm*)
qed

lemma *real-root-gt-zero*: *0 < n* ⟹ *0 < x* ⟹ *0 < root n x*
using *real-root-less-mono*[*of n 0 x*] **by** *simp*

lemma *real-root-ge-zero*: *0* ≤ *x* ⟹ *0* ≤ *root n x*
using *real-root-gt-zero*[*of n x*]

by (*cases n = 0*) (*auto simp add*: *le-less*)

lemma *real-root-pow-pos*: $0 < n \Longrightarrow 0 < x \Longrightarrow root\ n\ x\ \hat{}\ n = x$
using *sgn-power-root*[*of n x*] *real-root-gt-zero*[*of n x*] **by** *simp*

lemma *real-root-pow-pos2* [*simp*]: $0 < n \Longrightarrow 0 \leq x \Longrightarrow root\ n\ x\ \hat{}\ n = x$
by (*auto simp add*: *order-le-less real-root-pow-pos*)

lemma *sgn-root*: $0 < n \Longrightarrow sgn\ (root\ n\ x) = sgn\ x$
by (*auto split*: *split-root simp*: *sgn-real-def*)

lemma *odd-real-root-pow*: $odd\ n \Longrightarrow root\ n\ x\ \hat{}\ n = x$
using *sgn-power-root*[*of n x*]
by (*simp add*: *odd-pos sgn-real-def split*: *if-split-asm*)

lemma *real-root-power-cancel*: $0 < n \Longrightarrow 0 \leq x \Longrightarrow root\ n\ (x\ \hat{}\ n) = x$
using *root-sgn-power*[*of n x*] **by** (*auto simp add*: *le-less power-0-left*)

lemma *odd-real-root-power-cancel*: $odd\ n \Longrightarrow root\ n\ (x\ \hat{}\ n) = x$
using *root-sgn-power*[*of n x*]
by (*simp add*: *odd-pos sgn-real-def power-0-left split*: *if-split-asm*)

lemma *real-root-pos-unique*: $0 < n \Longrightarrow 0 \leq y \Longrightarrow y\ \hat{}\ n = x \Longrightarrow root\ n\ x = y$
using *real-root-power-cancel* **by** *blast*

lemma *odd-real-root-unique*: $odd\ n \Longrightarrow y\ \hat{}\ n = x \Longrightarrow root\ n\ x = y$
using *odd-real-root-power-cancel* **by** *blast*

lemma *real-root-one* [*simp*]: $0 < n \Longrightarrow root\ n\ 1 = 1$
by (*simp add*: *real-root-pos-unique*)

Root function is strictly monotonic, hence injective.

lemma *real-root-le-mono*: $0 < n \Longrightarrow x \leq y \Longrightarrow root\ n\ x \leq root\ n\ y$
by (*auto simp add*: *order-le-less real-root-less-mono*)

lemma *real-root-less-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x < root\ n\ y \longleftrightarrow x < y$
by (*cases x < y*) (*simp-all add*: *real-root-less-mono linorder-not-less real-root-le-mono*)

lemma *real-root-le-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x \leq root\ n\ y \longleftrightarrow x \leq y$
by (*cases x ≤ y*) (*simp-all add*: *real-root-le-mono linorder-not-le real-root-less-mono*)

lemma *real-root-eq-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x = root\ n\ y \longleftrightarrow x = y$
by (*simp add*: *order-eq-iff*)

lemmas *real-root-gt-0-iff* [*simp*] = *real-root-less-iff* [**where** *x=0*, *simplified*]
lemmas *real-root-lt-0-iff* [*simp*] = *real-root-less-iff* [**where** *y=0*, *simplified*]
lemmas *real-root-ge-0-iff* [*simp*] = *real-root-le-iff* [**where** *x=0*, *simplified*]
lemmas *real-root-le-0-iff* [*simp*] = *real-root-le-iff* [**where** *y=0*, *simplified*]
lemmas *real-root-eq-0-iff* [*simp*] = *real-root-eq-iff* [**where** *y=0*, *simplified*]

lemma *real-root-gt-1-iff* [*simp*]: $0 < n \Longrightarrow 1 < root\ n\ y \longleftrightarrow 1 < y$
using *real-root-less-iff* [**where** *x=1*] **by** *simp*

lemma *real-root-lt-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x < 1 \longleftrightarrow x < 1$
using *real-root-less-iff* [**where** *y=1*] **by** *simp*

lemma *real-root-ge-1-iff* [*simp*]: $0 < n \Longrightarrow 1 \leq root\ n\ y \longleftrightarrow 1 \leq y$
using *real-root-le-iff* [**where** *x=1*] **by** *simp*

lemma *real-root-le-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x \leq 1 \longleftrightarrow x \leq 1$
using *real-root-le-iff* [**where** *y=1*] **by** *simp*

lemma *real-root-eq-1-iff* [*simp*]: $0 < n \Longrightarrow root\ n\ x = 1 \longleftrightarrow x = 1$
using *real-root-eq-iff* [**where** *y=1*] **by** *simp*

Roots of multiplication and division.

lemma *real-root-mult*: $root\ n\ (x * y) = root\ n\ x * root\ n\ y$
by (*auto split*: *split-root elim*!: *sgn-power-injE*
simp: *sgn-mult abs-mult power-mult-distrib*)

lemma *real-root-inverse*: $root\ n\ (inverse\ x) = inverse\ (root\ n\ x)$
by (*auto split*: *split-root elim*!: *sgn-power-injE*
simp: *power-inverse*)

lemma *real-root-divide*: $root\ n\ (x\ /\ y) = root\ n\ x\ /\ root\ n\ y$
by (*simp add*: *divide-inverse real-root-mult real-root-inverse*)

lemma *real-root-abs*: $0 < n \Longrightarrow root\ n\ |x| = |root\ n\ x|$
by (*simp add*: *abs-if real-root-minus*)

lemma *root-abs-power*: $n > 0 \Longrightarrow abs\ (root\ n\ (y\ \hat{}\ n)) = abs\ y$
using *root-sgn-power* [*of n*]
by (*metis abs-ge-zero power-abs real-root-abs real-root-power-cancel*)

lemma *real-root-power*: $0 < n \Longrightarrow root\ n\ (x\ \hat{}\ k) = root\ n\ x\ \hat{}\ k$
by (*induct k*) (*simp-all add*: *real-root-mult*)

Roots of roots.

lemma *real-root-Suc-0* [*simp*]: $root\ (Suc\ 0)\ x = x$
by (*simp add*: *odd-real-root-unique*)

lemma *real-root-mult-exp*: $root\ (m * n)\ x = root\ m\ (root\ n\ x)$
by (*auto split*: *split-root elim*!: *sgn-power-injE*
simp: *sgn-zero-iff sgn-mult power-mult*[*symmetric*]
abs-mult power-mult-distrib abs-sgn-eq)

lemma *real-root-commute*: $root\ m\ (root\ n\ x) = root\ n\ (root\ m\ x)$
by (*simp add*: *real-root-mult-exp* [*symmetric*] *mult.commute*)

Monotonicity in first argument.

lemma *real-root-strict-decreasing*:
assumes $0 < n$ $n < N$ $1 < x$
shows *root N x* < *root n x*
proof −
from *assms* **have** *root n* (*root N x*) ^ *n* < *root N* (*root n x*) ^ *N*
by (*simp add*: *real-root-commute power-strict-increasing del*: *real-root-pow-pos2*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *real-root-strict-increasing*:
assumes $0 < n$ $n < N$ $0 < x$ $x < 1$
shows *root n x* < *root N x*
proof −
from *assms* **have** *root N* (*root n x*) ^ *N* < *root n* (*root N x*) ^ *n*
by (*simp add*: *real-root-commute power-strict-decreasing del*: *real-root-pow-pos2*)
with *assms* **show** *?thesis* **by** *simp*
qed

lemma *real-root-decreasing*: $0 < n \Longrightarrow n \leq N \Longrightarrow 1 \leq x \Longrightarrow$ *root N x* ≤ *root n x*
by (*auto simp add*: *order-le-less real-root-strict-decreasing*)

lemma *real-root-increasing*: $0 < n \Longrightarrow n \leq N \Longrightarrow 0 \leq x \Longrightarrow x \leq 1 \Longrightarrow$ *root n x* ≤ *root N x*
by (*auto simp add*: *order-le-less real-root-strict-increasing*)

Continuity and derivatives.

lemma *isCont-real-root*: *isCont* (*root n*) *x*
proof (*cases n* > *0*)
case *True*
let *?f* = λ*y*::*real*. *sgn y* ∗ |*y*| ^ *n*
have *continuous-on* ({*0*..} ∪ {.. *0*}) (λ*x*. *if 0* < *x then x* ^ *n else* − ((−*x*) ^ *n*) :: *real*)
using *True* **by** (*intro continuous-on-If continuous-intros*) *auto*
then have *continuous-on UNIV ?f*
by (*rule continuous-on-cong*[*THEN iffD1*, *rotated 2*]) (*auto simp*: *not-less le-less True*)
then have [*simp*]: *isCont ?f x* **for** *x*
by (*simp add*: *continuous-on-eq-continuous-at*)
have *isCont* (*root n*) (*?f* (*root n x*))
by (*rule isCont-inverse-function* [**where** *f*=*?f* **and** *d*=*1*]) (*auto simp*: *root-sgn-power True*)
then show *?thesis*
by (*simp add*: *sgn-power-root True*)
next
case *False*
then show *?thesis*
by (*simp add*: *root-def*[*abs-def*])
qed

lemma *tendsto-real-root* [*tendsto-intros*]:
(*f* ⟶ *x*) *F* ⟹ ((λ*x*. *root n* (*f x*)) ⟶ *root n x*) *F*
using *isCont-tendsto-compose*[*OF isCont-real-root*, *of f x F*] **.**

lemma *continuous-real-root* [*continuous-intros*]:
continuous F f ⟹ *continuous F* (λ*x*. *root n* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-real-root*)

lemma *continuous-on-real-root* [*continuous-intros*]:
continuous-on s f ⟹ *continuous-on s* (λ*x*. *root n* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-real-root*)

lemma *DERIV-real-root*:
assumes *n*: *0* < *n*
and *x*: *0* < *x*
shows *DERIV* (*root n*) *x* :> *inverse* (*real n* ∗ *root n x* ^ (*n* − *Suc 0*))
proof (*rule DERIV-inverse-function*)
show *0* < *x*
using *x* **.**
show *x* < *x* + *1*
by *simp*
show *DERIV* (λ*x*. *x* ^ *n*) (*root n x*) :> *real n* ∗ *root n x* ^ (*n* − *Suc 0*)
by (*rule DERIV-pow*)
show *real n* ∗ *root n x* ^ (*n* − *Suc 0*) ≠ *0*
using *n x* **by** *simp*
show *isCont* (*root n*) *x*
by (*rule isCont-real-root*)
qed (*use n* **in** *auto*)

lemma *DERIV-odd-real-root*:
assumes *n*: *odd n*
and *x*: *x* ≠ *0*
shows *DERIV* (*root n*) *x* :> *inverse* (*real n* ∗ *root n x* ^ (*n* − *Suc 0*))
proof (*rule DERIV-inverse-function*)
show *x* − *1* < *x x* < *x* + *1*
by *auto*
show *DERIV* (λ*x*. *x* ^ *n*) (*root n x*) :> *real n* ∗ *root n x* ^ (*n* − *Suc 0*)
by (*rule DERIV-pow*)
show *real n* ∗ *root n x* ^ (*n* − *Suc 0*) ≠ *0*
using *odd-pos* [*OF n*] *x* **by** *simp*
show *isCont* (*root n*) *x*
by (*rule isCont-real-root*)
qed (*use n odd-real-root-pow* **in** *auto*)

lemma *DERIV-even-real-root*:
assumes *n*: *0* < *n*
and *even n*
and *x*: *x* < *0*

```
  shows DERIV (root n) x :> inverse (- real n * root n x ^ (n - Suc 0))
proof (rule DERIV-inverse-function)
  show x - 1 < x
    by simp
  show x < 0
    using x .
  show - (root n y ^ n) = y if x - 1 < y and y < 0 for y
  proof -
    have root n (-y) ^ n = -y
      using that ‹0 < n› by simp
    with real-root-minus and ‹even n›
    show - (root n y ^ n) = y by simp
  qed
  show DERIV (λx. - (x ^ n)) (root n x) :> - real n * root n x ^ (n - Suc 0)
    by  (auto intro!: derivative-eq-intros)
  show - real n * root n x ^ (n - Suc 0) ≠ 0
    using n x by simp
  show isCont (root n) x
    by (rule isCont-real-root)
qed

lemma DERIV-real-root-generic:
  assumes 0 < n
    and x ≠ 0
    and even n ⟹ 0 < x ⟹ D = inverse (real n * root n x ^ (n - Suc 0))
    and even n ⟹ x < 0 ⟹ D = - inverse (real n * root n x ^ (n - Suc 0))
    and odd n ⟹ D = inverse (real n * root n x ^ (n - Suc 0))
  shows DERIV (root n) x :> D
  using assms
  by (cases even n, cases 0 < x)
    (auto intro: DERIV-real-root[THEN DERIV-cong]
      DERIV-odd-real-root[THEN DERIV-cong]
      DERIV-even-real-root[THEN DERIV-cong])

lemma power-tendsto-0-iff [simp]:
  fixes f :: 'a ⇒ real
  assumes n > 0
  shows ((λx. f x ^ n) ⟶ 0) F ⟷ (f ⟶ 0) F
proof -
  have ((λx. |root n (f x ^ n)|) ⟶ 0) F ⟹ (f ⟶ 0) F
    by (auto simp: assms root-abs-power tendsto-rabs-zero-iff)
  then have ((λx. f x ^ n) ⟶ 0) F ⟹ (f ⟶ 0) F
    by (metis tendsto-real-root abs-0 real-root-zero tendsto-rabs)
  with assms show ?thesis
    by (auto simp: tendsto-null-power)
qed
```

111.3 Square Root

definition *sqrt* :: *real* ⇒ *real*
where *sqrt* = *root 2*

lemma *pos2*: *0* < (*2*::*nat*)
by *simp*

lemma *real-sqrt-unique*: $y^2 = x \Longrightarrow 0 \leq y \Longrightarrow sqrt\ x = y$
unfolding *sqrt-def* **by** (*rule real-root-pos-unique* [*OF pos2*])

lemma *real-sqrt-abs* [*simp*]: *sqrt* $(x^2) = |x|$
by (*metis power2-abs abs-ge-zero real-sqrt-unique*)

lemma *real-sqrt-pow2* [*simp*]: $0 \leq x \Longrightarrow (sqrt\ x)^2 = x$
unfolding *sqrt-def* **by** (*rule real-root-pow-pos2* [*OF pos2*])

lemma *real-sqrt-pow2-iff* [*simp*]: $(sqrt\ x)^2 = x \longleftrightarrow 0 \leq x$
by (*metis real-sqrt-pow2 zero-le-power2*)

lemma *real-sqrt-zero* [*simp*]: *sqrt 0* = *0*
unfolding *sqrt-def* **by** (*rule real-root-zero*)

lemma *real-sqrt-one* [*simp*]: *sqrt 1* = *1*
unfolding *sqrt-def* **by** (*rule real-root-one* [*OF pos2*])

lemma *real-sqrt-four* [*simp*]: *sqrt 4* = *2*
using *real-sqrt-abs*[*of 2*] **by** *simp*

lemma *real-sqrt-minus*: *sqrt* (− *x*) = − *sqrt x*
unfolding *sqrt-def* **by** (*rule real-root-minus*)

lemma *real-sqrt-mult*: *sqrt* (*x* ∗ *y*) = *sqrt x* ∗ *sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-mult*)

lemma *real-sqrt-mult-self*[*simp*]: *sqrt a* ∗ *sqrt a* = |*a*|
using *real-sqrt-abs*[*of a*] **unfolding** *power2-eq-square real-sqrt-mult* **.**

lemma *real-sqrt-inverse*: *sqrt* (*inverse x*) = *inverse* (*sqrt x*)
unfolding *sqrt-def* **by** (*rule real-root-inverse*)

lemma *real-sqrt-divide*: *sqrt* (*x* / *y*) = *sqrt x* / *sqrt y*
unfolding *sqrt-def* **by** (*rule real-root-divide*)

lemma *real-sqrt-power*: *sqrt* (*x* ^ *k*) = *sqrt x* ^ *k*
unfolding *sqrt-def* **by** (*rule real-root-power* [*OF pos2*])

lemma *real-sqrt-gt-zero*: *0* < *x* ⟹ *0* < *sqrt x*
unfolding *sqrt-def* **by** (*rule real-root-gt-zero* [*OF pos2*])

lemma *real-sqrt-ge-zero*: $0 \leq x \Longrightarrow 0 \leq sqrt\ x$
unfolding *sqrt-def* **by** (*rule real-root-ge-zero*)

lemma *real-sqrt-less-mono*: $x < y \Longrightarrow sqrt\ x < sqrt\ y$
unfolding *sqrt-def* **by** (*rule real-root-less-mono* [*OF pos2*])

lemma *real-sqrt-le-mono*: $x \leq y \Longrightarrow sqrt\ x \leq sqrt\ y$
unfolding *sqrt-def* **by** (*rule real-root-le-mono* [*OF pos2*])

lemma *real-sqrt-less-iff* [*simp*]: $sqrt\ x < sqrt\ y \longleftrightarrow x < y$
unfolding *sqrt-def* **by** (*rule real-root-less-iff* [*OF pos2*])

lemma *real-sqrt-le-iff* [*simp*]: $sqrt\ x \leq sqrt\ y \longleftrightarrow x \leq y$
unfolding *sqrt-def* **by** (*rule real-root-le-iff* [*OF pos2*])

lemma *real-sqrt-eq-iff* [*simp*]: $sqrt\ x = sqrt\ y \longleftrightarrow x = y$
unfolding *sqrt-def* **by** (*rule real-root-eq-iff* [*OF pos2*])

lemma *real-less-lsqrt*: $0 \leq y \Longrightarrow x < y^2 \Longrightarrow sqrt\ x < y$
using *real-sqrt-less-iff*[*of x* y^2] **by** *simp*

lemma *real-le-lsqrt*: $0 \leq y \Longrightarrow x \leq y^2 \Longrightarrow sqrt\ x \leq y$
using *real-sqrt-le-iff*[*of x* y^2] **by** *simp*

lemma *real-le-rsqrt*: $x^2 \leq y \Longrightarrow x \leq sqrt\ y$
using *real-sqrt-le-mono*[*of* x^2 *y*] **by** *simp*

lemma *real-less-rsqrt*: $x^2 < y \Longrightarrow x < sqrt\ y$
using *real-sqrt-less-mono*[*of* x^2 *y*] **by** *simp*

lemma *real-sqrt-power-even*:
assumes *even n* $x \geq 0$
shows *sqrt* $x \mathbin{\hat{}} n = x \mathbin{\hat{}} (n\ div\ 2)$
proof −
from *assms* **obtain** *k* **where** $n = 2*k$ **by** (*auto elim*!: *evenE*)
with *assms* **show** *?thesis* **by** (*simp add*: *power-mult*)
qed

lemma *sqrt-le-D*: $sqrt\ x \leq y \Longrightarrow x \leq y^2$
by (*meson not-le real-less-rsqrt*)

lemma *sqrt-ge-absD*: $|x| \leq sqrt\ y \Longrightarrow x^2 \leq y$
using *real-sqrt-le-iff*[*of* x^2] **by** *simp*

lemma *sqrt-even-pow2*:
assumes *n*: *even n*
shows *sqrt* $(2 \mathbin{\hat{}} n) = 2 \mathbin{\hat{}} (n\ div\ 2)$
proof −
from *n* **obtain** *m* **where** *m*: $n = 2 * m$ **..**

from *m* **have** *sqrt* $(2 \hat{} n) = sqrt\ ((2 \hat{} m)^2)$
by (*simp only*: *power-mult*[*symmetric*] *mult.commute*)
then show *?thesis*
using *m* **by** *simp*
qed

lemmas *real-sqrt-gt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** *x=0*, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-lt-0-iff* [*simp*] = *real-sqrt-less-iff* [**where** *y=0*, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-ge-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** *x=0*, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-le-0-iff* [*simp*] = *real-sqrt-le-iff* [**where** *y=0*, *unfolded real-sqrt-zero*]
lemmas *real-sqrt-eq-0-iff* [*simp*] = *real-sqrt-eq-iff* [**where** *y=0*, *unfolded real-sqrt-zero*]

lemmas *real-sqrt-gt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** *x=1*, *unfolded real-sqrt-one*]
lemmas *real-sqrt-lt-1-iff* [*simp*] = *real-sqrt-less-iff* [**where** *y=1*, *unfolded real-sqrt-one*]
lemmas *real-sqrt-ge-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** *x=1*, *unfolded real-sqrt-one*]
lemmas *real-sqrt-le-1-iff* [*simp*] = *real-sqrt-le-iff* [**where** *y=1*, *unfolded real-sqrt-one*]
lemmas *real-sqrt-eq-1-iff* [*simp*] = *real-sqrt-eq-iff* [**where** *y=1*, *unfolded real-sqrt-one*]

lemma *sqrt-add-le-add-sqrt*:
assumes $0 \leq x$ $0 \leq y$
shows $sqrt\ (x + y) \leq sqrt\ x + sqrt\ y$
by (*rule power2-le-imp-le*) (*simp-all add*: *power2-sum assms*)

lemma *isCont-real-sqrt*: *isCont sqrt x*
unfolding *sqrt-def* **by** (*rule isCont-real-root*)

lemma *tendsto-real-sqrt* [*tendsto-intros*]:
$(f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ sqrt\ (f\ x)) \longrightarrow sqrt\ x)\ F$
unfolding *sqrt-def* **by** (*rule tendsto-real-root*)

lemma *continuous-real-sqrt* [*continuous-intros*]:
continuous F f $\Longrightarrow$ *continuous F* $(\lambda x.\ sqrt\ (f\ x))$
unfolding *sqrt-def* **by** (*rule continuous-real-root*)

lemma *continuous-on-real-sqrt* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* $(\lambda x.\ sqrt\ (f\ x))$
unfolding *sqrt-def* **by** (*rule continuous-on-real-root*)

lemma *DERIV-real-sqrt-generic*:
assumes $x \neq 0$
and $x > 0 \Longrightarrow D = inverse\ (sqrt\ x)\ /\ 2$
and $x < 0 \Longrightarrow D = -\ inverse\ (sqrt\ x)\ /\ 2$
shows *DERIV sqrt x :> D*
using *assms* **unfolding** *sqrt-def*
by (*auto intro*!: *DERIV-real-root-generic*)

lemma *DERIV-real-sqrt*: $0 < x \Longrightarrow DERIV\ sqrt\ x :> inverse\ (sqrt\ x)\ /\ 2$
using *DERIV-real-sqrt-generic* **by** *simp*

declare
DERIV-real-sqrt-generic[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-real-root-generic[*THEN DERIV-chain2*, *derivative-intros*]

lemmas *has-derivative-real-sqrt*[*derivative-intros*] = *DERIV-real-sqrt*[*THEN DERIV-compose-FDERIV*]

lemma *not-real-square-gt-zero* [*simp*]: ¬ *0* < *x* * *x* ⟷ *x* = *0*
for *x* :: *real*
by (*metis linorder-neq-iff zero-less-mult-iff*)

lemma *real-sqrt-abs2* [*simp*]: *sqrt* (*x* * *x*) = |*x*|
by (*simp add*: *real-sqrt-mult*)

lemma *real-sqrt-abs'*: *sqrt* |*x*| = |*sqrt x*|
by (*metis real-sqrt-abs2 real-sqrt-mult*)

lemma *real-inv-sqrt-pow2*: *0* < *x* ⟹ (*inverse* (*sqrt x*))² = *inverse x*
by (*simp add*: *power-inverse*)

lemma *real-sqrt-eq-zero-cancel*: *0* ≤ *x* ⟹ *sqrt x* = *0* ⟹ *x* = *0*
by *simp*

lemma *real-sqrt-ge-one*: *1* ≤ *x* ⟹ *1* ≤ *sqrt x*
by *simp*

lemma *sqrt-divide-self-eq*:
assumes *nneg*: *0* ≤ *x*
shows *sqrt x* / *x* = *inverse* (*sqrt x*)
proof (*cases x* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *pos*: *0* < *x*
using *nneg* **by** *arith*
show *?thesis*
proof (*rule right-inverse-eq* [*THEN iffD1*, *symmetric*])
show *sqrt x* / *x* ≠ *0*
by (*simp add*: *divide-inverse nneg False*)
show *inverse* (*sqrt x*) / (*sqrt x* / *x*) = *1*
by (*simp add*: *divide-inverse mult.assoc* [*symmetric*]
power2-eq-square [*symmetric*] *real-inv-sqrt-pow2 pos False*)
qed
qed

lemma *real-div-sqrt*: *0* ≤ *x* ⟹ *x* / *sqrt x* = *sqrt x*
by (*cases x* = *0*) (*simp-all add*: *sqrt-divide-self-eq* [*of x*] *field-simps*)

lemma *real-divide-square-eq* [*simp*]: $(r * a) / (r * r) = a / r$
for *a r* :: *real*
by (*cases r = 0*) (*simp-all add*: *divide-inverse ac-simps*)

lemma *lemma-real-divide-sqrt-less*: $0 < u \Longrightarrow u / sqrt\ 2 < u$
by (*simp add*: *divide-less-eq*)

lemma *four-x-squared*: $4 * x^2 = (2 * x)^2$
for *x* :: *real*
by (*simp add*: *power2-eq-square*)

lemma *sqrt-at-top*: *LIM x at-top. sqrt x* :: *real* :> *at-top*
by (*rule filterlim-at-top-at-top*[**where** *Q*=$\lambda x.\ True$ **and** *P*=$\lambda x.\ 0 < x$ **and** *g*=*power2*])
(*auto intro*: *eventually-gt-at-top*)

111.4 Square Root of Sum of Squares

lemma *sum-squares-bound*: $2 * x * y \leq x^2 + y^2$
for *x y* :: *'a*::*linordered-field*
proof −
have $(x - y)^2 = x * x - 2 * x * y + y * y$
by *algebra*
then have $0 \leq x^2 - 2 * x * y + y^2$
by (*metis sum-power2-ge-zero zero-le-double-add-iff-zero-le-single-add power2-eq-square*)
then show *?thesis*
by *arith*
qed

lemma *arith-geo-mean*:
fixes *u* :: *'a*::*linordered-field*
assumes $u^2 = x * y\ x \geq 0\ y \geq 0$
shows $u \leq (x + y)/2$
apply (*rule power2-le-imp-le*)
using *sum-squares-bound assms*
apply (*auto simp*: *zero-le-mult-iff*)
apply (*auto simp*: *algebra-simps power2-eq-square*)
done

lemma *arith-geo-mean-sqrt*:
fixes *x* :: *real*
assumes $x \geq 0\ y \geq 0$
shows $sqrt\ (x * y) \leq (x + y)/2$
apply (*rule arith-geo-mean*)
using *assms*
apply (*auto simp*: *zero-le-mult-iff*)
done

lemma *real-sqrt-sum-squares-mult-ge-zero* [*simp*]: $0 \leq sqrt\ ((x^2 + y^2) * (xa^2 +$

$ya^2))$
by (*metis real-sqrt-ge-0-iff split-mult-pos-le sum-power2-ge-zero*)

lemma *real-sqrt-sum-squares-mult-squared-eq* [*simp*]:
$(sqrt\ ((x^2 + y^2) * (xa^2 + ya^2)))^2 = (x^2 + y^2) * (xa^2 + ya^2)$
by (*simp add*: *zero-le-mult-iff*)

lemma *real-sqrt-sum-squares-eq-cancel*: $sqrt\ (x^2 + y^2) = x \Longrightarrow y = 0$
by (*drule arg-cong* [**where** $f = \lambda x.\ x^2$]) *simp*

lemma *real-sqrt-sum-squares-eq-cancel2*: $sqrt\ (x^2 + y^2) = y \Longrightarrow x = 0$
by (*drule arg-cong* [**where** $f = \lambda x.\ x^2$]) *simp*

lemma *real-sqrt-sum-squares-ge1* [*simp*]: $x \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*) *simp-all*

lemma *real-sqrt-sum-squares-ge2* [*simp*]: $y \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*) *simp-all*

lemma *real-sqrt-ge-abs1* [*simp*]: $|x| \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*) *simp-all*

lemma *real-sqrt-ge-abs2* [*simp*]: $|y| \leq sqrt\ (x^2 + y^2)$
by (*rule power2-le-imp-le*) *simp-all*

lemma *le-real-sqrt-sumsq* [*simp*]: $x \leq sqrt\ (x * x + y * y)$
by (*simp add*: *power2-eq-square* [*symmetric*])

lemma *sqrt-sum-squares-le-sum*:
$[\![0 \leq x;\ 0 \leq y]\!] \Longrightarrow sqrt\ (x^2 + y^2) \leq x + y$
by (*rule power2-le-imp-le*) (*simp-all add*: *power2-sum*)

lemma *L2-set-mult-ineq-lemma*:
fixes *a b c d* :: *real*
shows $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
proof −
have $0 \leq (a * d - b * c)^2$ **by** *simp*
also have $\ldots = a^2 * d^2 + b^2 * c^2 - 2 * (a * d) * (b * c)$
by (*simp only*: *power2-diff power-mult-distrib*)
also have $\ldots = a^2 * d^2 + b^2 * c^2 - 2 * (a * c) * (b * d)$
by *simp*
finally show $2 * (a * c) * (b * d) \leq a^2 * d^2 + b^2 * c^2$
by *simp*
qed

lemma *sqrt-sum-squares-le-sum-abs*: $sqrt\ (x^2 + y^2) \leq |x| + |y|$
by (*rule power2-le-imp-le*) (*simp-all add*: *power2-sum*)

lemma *real-sqrt-sum-squares-triangle-ineq*:

sqrt $((a + c)^2 + (b + d)^2) \leq$ *sqrt* $(a^2 + b^2)$ + *sqrt* $(c^2 + d^2)$
proof −
have $(a * c + b * d) \leq$ (*sqrt* $(a^2 + b^2) *$ *sqrt* $(c^2 + d^2)$)
by (*rule power2-le-imp-le*) (*simp-all add*: *power2-sum power-mult-distrib ring-distribs L2-set-mult-ineq-lemma add.commute*)
then have $(a + c)^2 + (b + d)^2 \leq$ (*sqrt* $(a^2 + b^2)$ + *sqrt* $(c^2 + d^2))^2$
by (*simp add*: *power2-sum*)
then show *?thesis*
by (*auto intro*: *power2-le-imp-le*)
qed

lemma *real-sqrt-sum-squares-less*: $|x| <$ *u* / *sqrt 2* $\Longrightarrow |y| <$ *u* / *sqrt 2* $\Longrightarrow$ *sqrt* $(x^2 + y^2) <$ *u*
apply (*rule power2-less-imp-less*)
apply *simp*
apply (*drule power-strict-mono* [*OF - abs-ge-zero pos2*])
apply (*drule power-strict-mono* [*OF - abs-ge-zero pos2*])
apply (*simp add*: *power-divide*)
apply (*drule order-le-less-trans* [*OF abs-ge-zero*])
apply (*simp add*: *zero-less-divide-iff*)
done

lemma *sqrt2-less-2*: *sqrt 2* $<$ (*2*::*real*)
by (*metis not-less not-less-iff-gr-or-eq numeral-less-iff real-sqrt-four real-sqrt-le-iff semiring-norm*(*75*) *semiring-norm*(*78*) *semiring-norm*(*85*))

lemma *sqrt-sum-squares-half-less*:
x $<$ *u*/*2* $\Longrightarrow$ *y* $<$ *u*/*2* $\Longrightarrow$ *0* $\leq$ *x* $\Longrightarrow$ *0* $\leq$ *y* $\Longrightarrow$ *sqrt* $(x^2 + y^2) <$ *u*
apply (*rule real-sqrt-sum-squares-less*)
apply (*auto simp add*: *abs-if field-simps*)
apply (*rule le-less-trans* [**where** *y* = *x*∗*2*])
using *less-eq-real-def sqrt2-less-2* **apply** *force*
apply *assumption*
apply (*rule le-less-trans* [**where** *y* = *y*∗*2*])
using *less-eq-real-def sqrt2-less-2 mult-le-cancel-left*
apply *auto*
done

lemma *LIMSEQ-root*: (λ*n*. *root n n*) ⟶ *1*
proof −
define *x* **where** *x n* = *root n n* − *1* **for** *n*
have *x* ⟶ *sqrt 0*
proof (*rule tendsto-sandwich*[*OF - - tendsto-const*])
show (λ*x*. *sqrt* (*2* / *x*)) ⟶ *sqrt 0*
by (*intro tendsto-intros tendsto-divide-0*[*OF tendsto-const*] *filterlim-mono*[*OF filterlim-real-sequentially*])
(*simp-all add*: *at-infinity-eq-at-top-bot*)
have *x n* $\leq$ *sqrt* (*2* / *real n*) **if** *2* $<$ *n* **for** *n* :: *nat*
proof −

```
    have 1 + (real (n − 1) ∗ n) / 2 ∗ (x n)^2 = 1 + of-nat (n choose 2) ∗ (x n)^2
      by (auto simp add: choose-two field-char-0-class.of-nat-div mod-eq-0-iff-dvd)
    also have ... ≤ (∑ k∈{0, 2}. of-nat (n choose k) ∗ x n^k)
      by (simp add: x-def)
    also have ... ≤ (∑ k≤n. of-nat (n choose k) ∗ x n^k)
      using ‹2 < n›
      by (intro sum-mono2) (auto intro!: mult-nonneg-nonneg zero-le-power simp:
x-def le-diff-eq)
    also have ... = (x n + 1) ^ n
      by (simp add: binomial-ring)
    also have ... = n
      using ‹2 < n› by (simp add: x-def)
    finally have real (n − 1) ∗ (real n / 2 ∗ (x n)^2) ≤ real (n − 1) ∗ 1
      using that by auto
    then have (x n)^2 ≤ 2 / real n
      using ‹2 < n› unfolding mult-le-cancel-left by (simp add: field-simps)
    from real-sqrt-le-mono[OF this] show ?thesis
      by simp
  qed
  then show eventually (λn. x n ≤ sqrt (2 / real n)) sequentially
    by (auto intro!: exI[of - 3] simp: eventually-sequentially)
  show eventually (λn. sqrt 0 ≤ x n) sequentially
    by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)
 qed
 from tendsto-add[OF this tendsto-const[of 1]] show ?thesis
  by (simp add: x-def)
qed

lemma LIMSEQ-root-const:
 assumes 0 < c
 shows (λn. root n c) ⟶ 1
proof −
 have ge-1: (λn. root n c) ⟶ 1 if 1 ≤ c for c :: real
 proof −
  define x where x n = root n c − 1 for n
  have x ⟶ 0
  proof (rule tendsto-sandwich[OF - - tendsto-const])
   show (λn. c / n) ⟶ 0
   by (intro tendsto-divide-0[OF tendsto-const] filterlim-mono[OF filterlim-real-sequentially])
      (simp-all add: at-infinity-eq-at-top-bot)
   have x n ≤ c / n if 1 < n for n :: nat
   proof −
    have 1 + x n ∗ n = 1 + of-nat (n choose 1) ∗ x n^1
     by (simp add: choose-one)
    also have ... ≤ (∑ k∈{0, 1}. of-nat (n choose k) ∗ x n^k)
     by (simp add: x-def)
    also have ... ≤ (∑ k≤n. of-nat (n choose k) ∗ x n^k)
     using ‹1 < n› ‹1 ≤ c›
     by (intro sum-mono2)
```

```
        (auto intro!: mult-nonneg-nonneg zero-le-power simp: x-def le-diff-eq)
      also have ... = (x n + 1) ^ n
        by (simp add: binomial-ring)
      also have ... = c
        using ‹1 < n› ‹1 ≤ c› by (simp add: x-def)
      finally show ?thesis
        using ‹1 ≤ c› ‹1 < n› by (simp add: field-simps)
    qed
    then show eventually (λn. x n ≤ c / n) sequentially
      by (auto intro!: exI[of - 3] simp: eventually-sequentially)
    show eventually (λn. 0 ≤ x n) sequentially
      using ‹1 ≤ c›
      by (auto intro!: exI[of - 1] simp: eventually-sequentially le-diff-eq x-def)
  qed
  from tendsto-add[OF this tendsto-const[of 1]] show ?thesis
    by (simp add: x-def)
 qed
 show ?thesis
 proof (cases 1 ≤ c)
  case True
  with ge-1 show ?thesis by blast
 next
  case False
  with ‹0 < c› have 1 ≤ 1 / c
    by simp
  then have (λn. 1 / root n (1 / c)) ⟶ 1 / 1
    by (intro tendsto-divide tendsto-const ge-1 ‹1 ≤ 1 / c› one-neq-zero)
  then show ?thesis
    by (rule filterlim-cong[THEN iffD1, rotated 3])
      (auto intro!: exI[of - 1] simp: eventually-sequentially real-root-divide)
 qed
qed
```

Legacy theorem names:

```
lemmas real-root-pos2 = real-root-power-cancel
lemmas real-root-pos-pos = real-root-gt-zero [THEN order-less-imp-le]
lemmas real-root-pos-pos-le = real-root-ge-zero
lemmas real-sqrt-eq-zero-cancel-iff = real-sqrt-eq-0-iff

end
```

112 Power Series, Transcendental Functions etc.

```
theory Transcendental
imports Series Deriv NthRoot
begin
```

A theorem about the factcorial function on the reals.

```
lemma square-fact-le-2-fact: fact n * fact n ≤ (fact (2 * n) :: real)
```

proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have (*fact* (*Suc n*)) ∗ (*fact* (*Suc n*)) = *of-nat* (*Suc n*) ∗ *of-nat* (*Suc n*) ∗ (*fact n* ∗ *fact n* :: *real*)
by (*simp add*: *field-simps*)
also have ... ≤ *of-nat* (*Suc n*) ∗ *of-nat* (*Suc n*) ∗ *fact* (*2* ∗ *n*)
by (*rule mult-left-mono* [*OF Suc*]) *simp*
also have ... ≤ *of-nat* (*Suc* (*Suc* (*2* ∗ *n*))) ∗ *of-nat* (*Suc* (*2* ∗ *n*)) ∗ *fact* (*2* ∗ *n*)
by (*rule mult-right-mono*)+ (*auto simp*: *field-simps*)
also have ... = *fact* (*2* ∗ *Suc n*) **by** (*simp add*: *field-simps*)
finally show *?case* **.**
qed

lemma *fact-in-Reals*: *fact n* ∈ ℝ
by (*induction n*) *auto*

lemma *of-real-fact* [*simp*]: *of-real* (*fact n*) = *fact n*
by (*metis of-nat-fact of-real-of-nat-eq*)

lemma *pochhammer-of-real*: *pochhammer* (*of-real x*) *n* = *of-real* (*pochhammer x n*)
by (*simp add*: *pochhammer-prod*)

lemma *norm-fact* [*simp*]: *norm* (*fact n* :: *'a*::*real-normed-algebra-1*) = *fact n*
proof −
have (*fact n* :: *'a*) = *of-real* (*fact n*)
by *simp*
also have *norm* ... = *fact n*
by (*subst norm-of-real*) *simp*
finally show *?thesis* **.**
qed

lemma *root-test-convergence*:
fixes *f* :: *nat* ⇒ *'a*::*banach*
assumes *f*: (λ*n*. *root n* (*norm* (*f n*))) ⟶ *x* — could be weakened to lim sup
and *x* < *1*
shows *summable f*
proof −
have *0* ≤ *x*
by (*rule LIMSEQ-le*[*OF tendsto-const f*]) (*auto intro*!: *exI*[*of - 1*])
from ‹*x* < *1*› **obtain** *z* **where** *z*: *x* < *z z* < *1*
by (*metis dense*)
from *f* ‹*x* < *z*› **have** *eventually* (λ*n*. *root n* (*norm* (*f n*)) < *z*) *sequentially*
by (*rule order-tendstoD*)
then have *eventually* (λ*n*. *norm* (*f n*) ≤ *z* ^ *n*) *sequentially*

using *eventually-ge-at-top*
proof *eventually-elim*
fix *n*
assume *less*: *root n* (*norm* (*f n*)) < *z* **and** *n*: *1* ≤ *n*
from *power-strict-mono*[*OF less*, *of n*] *n* **show** *norm* (*f n*) ≤ *z* ^ *n*
by *simp*
qed
then show *summable f*
unfolding *eventually-sequentially*
using *z* ‹*0* ≤ *x*› **by** (*auto intro*!: *summable-comparison-test*[*OF* - *summable-geometric*])
qed

112.1 Properties of Power Series

lemma *powser-zero* [*simp*]: (∑ *n*. *f n* * *0* ^ *n*) = *f 0*
for *f* :: *nat* ⇒ 'a::*real-normed-algebra-1*
proof −
have (∑ *n*<*1*. *f n* * *0* ^ *n*) = (∑ *n*. *f n* * *0* ^ *n*)
by (*subst suminf-finite*[**where** *N*={*0*}]) (*auto simp*: *power-0-left*)
then show *?thesis* **by** *simp*
qed

lemma *powser-sums-zero*: (λ*n*. *a n* * *0*^*n*) *sums a 0*
for *a* :: *nat* ⇒ 'a::*real-normed-div-algebra*
using *sums-finite* [*of* {*0*} λ*n*. *a n* * *0* ^ *n*]
by *simp*

lemma *powser-sums-zero-iff* [*simp*]: (λ*n*. *a n* * *0*^*n*) *sums x* ⟷ *a 0* = *x*
for *a* :: *nat* ⇒ 'a::*real-normed-div-algebra*
using *powser-sums-zero sums-unique2* **by** *blast*

Power series has a circle or radius of convergence: if it sums for x, then it sums absolutely for z with $|z| < |x|$.

lemma *powser-insidea*:
fixes *x z* :: 'a::*real-normed-div-algebra*
assumes *1*: *summable* (λ*n*. *f n* * *x*^*n*)
and *2*: *norm z* < *norm x*
shows *summable* (λ*n*. *norm* (*f n* * *z* ^ *n*))
proof −
from *2* **have** *x-neq-0*: *x* ≠ *0* **by** *clarsimp*
from *1* **have** (λ*n*. *f n* * *x*^*n*) ⟶ *0*
by (*rule summable-LIMSEQ-zero*)
then have *convergent* (λ*n*. *f n* * *x*^*n*)
by (*rule convergentI*)
then have *Cauchy* (λ*n*. *f n* * *x*^*n*)
by (*rule convergent-Cauchy*)
then have *Bseq* (λ*n*. *f n* * *x*^*n*)
by (*rule Cauchy-Bseq*)
then obtain *K* **where** *3*: *0* < *K* **and** *4*: ∀ *n*. *norm* (*f n* * *x*^*n*) ≤ *K*

```
    by (auto simp: Bseq-def)
  have ∃N. ∀n≥N. norm (norm (f n * z ^ n)) ≤ K * norm (z ^ n) * inverse
(norm (x^n))
  proof (intro exI allI impI)
    fix n :: nat
    assume 0 ≤ n
    have norm (norm (f n * z ^ n)) * norm (x^n) =
          norm (f n * x^n) * norm (z ^ n)
      by (simp add: norm-mult abs-mult)
    also have ... ≤ K * norm (z ^ n)
      by (simp only: mult-right-mono 4 norm-ge-zero)
    also have ... = K * norm (z ^ n) * (inverse (norm (x^n)) * norm (x^n))
      by (simp add: x-neq-0)
    also have ... = K * norm (z ^ n) * inverse (norm (x^n)) * norm (x^n)
      by (simp only: mult.assoc)
    finally show norm (norm (f n * z ^ n)) ≤ K * norm (z ^ n) * inverse (norm
(x^n))
      by (simp add: mult-le-cancel-right x-neq-0)
  qed
  moreover have summable (λn. K * norm (z ^ n) * inverse (norm (x^n)))
  proof −
    from 2 have norm (norm (z * inverse x)) < 1
      using x-neq-0
     by (simp add: norm-mult nonzero-norm-inverse divide-inverse [where 'a=real,
symmetric])
    then have summable (λn. norm (z * inverse x) ^ n)
      by (rule summable-geometric)
    then have summable (λn. K * norm (z * inverse x) ^ n)
      by (rule summable-mult)
    then show summable (λn. K * norm (z ^ n) * inverse (norm (x^n)))
      using x-neq-0
      by (simp add: norm-mult nonzero-norm-inverse power-mult-distrib
          power-inverse norm-power mult.assoc)
  qed
  ultimately show summable (λn. norm (f n * z ^ n))
    by (rule summable-comparison-test)
qed

lemma powser-inside:
  fixes f :: nat ⇒ 'a::{real-normed-div-algebra,banach}
  shows
    summable (λn. f n * (x^n)) ⟹ norm z < norm x ⟹
      summable (λn. f n * (z ^ n))
  by (rule powser-insidea [THEN summable-norm-cancel])

lemma powser-times-n-limit-0:
  fixes x :: 'a::{real-normed-div-algebra,banach}
  assumes norm x < 1
    shows (λn. of-nat n * x ^ n) ⟶ 0
```

proof −
have *norm x / (1 − norm x) ≥ 0*
using *assms* **by** (*auto simp*: *field-split-simps*)
moreover obtain *N* **where** *N*: *norm x / (1 − norm x) < of-int N*
using *ex-le-of-int* **by** (*meson ex-less-of-int*)
ultimately have *N0*: *N>0*
by *auto*
then have ∗: *real-of-int (N + 1) ∗ norm x / real-of-int N < 1*
using *N assms* **by** (*auto simp*: *field-simps*)
have ∗∗: *real-of-int N ∗ (norm x ∗ (real-of-nat (Suc n) ∗ norm (x ^ n))) ≤*
real-of-nat n ∗ (norm x ∗ ((1 + N) ∗ norm (x ^ n))) **if** *N ≤ int n* **for** *n* :: *nat*
proof −
from *that* **have** *real-of-int N ∗ real-of-nat (Suc n) ≤ real-of-nat n ∗ real-of-int (1 + N)*
by (*simp add*: *algebra-simps*)
then have (*real-of-int N ∗ real-of-nat (Suc n)*) ∗ (*norm x ∗ norm (x ^ n)*) ≤
(*real-of-nat n ∗ (1 + N)*) ∗ (*norm x ∗ norm (x ^ n)*)
using *N0 mult-mono* **by** *fastforce*
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed
show *?thesis* **using** ∗
by (*rule summable-LIMSEQ-zero* [*OF summable-ratio-test*, **where** *N1=nat N*])
(*simp add*: *N0 norm-mult field-simps* ∗∗ *del*: *of-nat-Suc of-int-add*)
qed

corollary *lim-n-over-pown*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
shows *1 < norm x ⟹ ((λn. of-nat n / x^n) ⟶ 0) sequentially*
using *powser-times-n-limit-0* [*of inverse x*]
by (*simp add*: *norm-divide field-split-simps*)

lemma *sum-split-even-odd*:
fixes *f* :: *nat ⇒ real*
shows (∑ *i*<*2 ∗ n. if even i then f i else g i*) = (∑ *i*<*n. f (2 ∗ i)*) + (∑ *i*<*n. g (2 ∗ i + 1)*)
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
have (∑ *i*<*2 ∗ Suc n. if even i then f i else g i*) =
(∑ *i*<*n. f (2 ∗ i)*) + (∑ *i*<*n. g (2 ∗ i + 1)*) + (*f (2 ∗ n) + g (2 ∗ n + 1)*)
using *Suc.hyps* **unfolding** *One-nat-def* **by** *auto*
also have *. . .* = (∑ *i*<*Suc n. f (2 ∗ i)*) + (∑ *i*<*Suc n. g (2 ∗ i + 1)*)
by *auto*
finally show *?case* **.**
qed

lemma *sums-if'*:
fixes $g :: nat \Rightarrow real$
assumes g *sums* x
shows $(\lambda\ n.$ *if even* n *then* 0 *else* $g\ ((n - 1)\ div\ 2))$ *sums* x
unfolding *sums-def*
proof (*rule LIMSEQ-I*)
fix $r :: real$
assume $0 < r$
from ‹g *sums* x›[*unfolded sums-def*, *THEN LIMSEQ-D*, *OF this*]
obtain *no* **where** *no-eq*: $\bigwedge n.\ n \geq no \Longrightarrow (norm\ (sum\ g\ \{..<n\} - x) < r)$
by *blast*

let *?SUM* $= \lambda\ m. \sum i<m.$ *if even* i *then* 0 *else* $g\ ((i - 1)\ div\ 2)$
have $(norm\ (?SUM\ m - x) < r)$ **if** $m \geq 2 * no$ **for** m
proof −
from *that* **have** $m\ div\ 2 \geq no$ **by** *auto*
have *sum-eq*: *?SUM* $(2 * (m\ div\ 2)) = sum\ g\ \{..<\ m\ div\ 2\}$
using *sum-split-even-odd* **by** *auto*
then have $(norm\ (?SUM\ (2 * (m\ div\ 2)) - x) < r)$
using *no-eq* **unfolding** *sum-eq* **using** ‹$m\ div\ 2 \geq no$› **by** *auto*
moreover
have *?SUM* $(2 * (m\ div\ 2)) =$ *?SUM* m
proof (*cases even* m)
case *True*
then show *?thesis*
by (*auto simp*: *even-two-times-div-two*)
next
case *False*
then have *eq*: *Suc* $(2 * (m\ div\ 2)) = m$ **by** *simp*
then have *even* $(2 * (m\ div\ 2))$ **using** ‹*odd* m› **by** *auto*
have *?SUM* $m =$ *?SUM* $(Suc\ (2 * (m\ div\ 2)))$ **unfolding** *eq* **..**
also have $\ldots =$ *?SUM* $(2 * (m\ div\ 2))$ **using** ‹*even* $(2 * (m\ div\ 2))$› **by** *auto*
finally show *?thesis* **by** *auto*
qed
ultimately show *?thesis* **by** *auto*
qed
then show $\exists\ no.\ \forall\ m \geq no.\ norm\ (?SUM\ m - x) < r$
by *blast*
qed

lemma *sums-if*:
fixes $g :: nat \Rightarrow real$
assumes g *sums* x **and** f *sums* y
shows $(\lambda\ n.$ *if even* n *then* $f\ (n\ div\ 2)$ *else* $g\ ((n - 1)\ div\ 2))$ *sums* $(x + y)$
proof −
let *?s* $= \lambda\ n.$ *if even* n *then* 0 *else* $f\ ((n - 1)\ div\ 2)$
have *if-sum*: $(if\ B\ then\ (0 :: real)\ else\ E) + (if\ B\ then\ T\ else\ 0) = (if\ B\ then\ T\ else\ E)$

for *B T E*
by (*cases B*) *auto*
have *g-sums*: (λ *n*. *if even n then 0 else g* ((*n* − *1*) *div 2*)) *sums x*
using *sums-if'*[*OF* ‹*g sums x*›] **.**
have *if-eq*: ⋀*B T E*. (*if* ¬ *B then T else E*) = (*if B then E else T*)
by *auto*
have *?s sums y* **using** *sums-if'*[*OF* ‹*f sums y*›] **.**
from *this*[*unfolded sums-def*, *THEN LIMSEQ-Suc*]
have (λ*n*. *if even n then f* (*n div 2*) *else 0*) *sums y*
by (*simp add*: *lessThan-Suc-eq-insert-0 sum.atLeast1-atMost-eq image-Suc-lessThan if-eq sums-def cong del*: *if-weak-cong*)
from *sums-add*[*OF g-sums this*] **show** *?thesis*
by (*simp only*: *if-sum*)
qed

112.2 Alternating series test / Leibniz formula

lemma *sums-alternating-upper-lower*:
fixes *a* :: *nat* ⇒ *real*
assumes *mono*: ⋀*n*. *a* (*Suc n*) ≤ *a n*
and *a-pos*: ⋀*n*. *0* ≤ *a n*
and *a* ⟶ *0*
shows ∃ *l*. ((∀ *n*. (∑ *i*<*2*∗*n*. (− *1*) ^ *i*∗*a i*) ≤ *l*) ∧ (λ *n*. ∑ *i*<*2*∗*n*. (− *1*) ^ *i*∗*a i*) ⟶ *l*) ∧
((∀ *n*. *l* ≤ (∑ *i*<*2*∗*n* + *1*. (− *1*) ^ *i*∗*a i*)) ∧ (λ *n*. ∑ *i*<*2*∗*n* + *1*. (− *1*) ^ *i*∗*a i*) ⟶ *l*)
(**is** ∃ *l*. ((∀ *n*. *?f n* ≤ *l*) ∧ -) ∧ ((∀ *n*. *l* ≤ *?g n*) ∧ -))
proof (*rule nested-sequence-unique*)
have *fg-diff*: ⋀*n*. *?f n* − *?g n* = − *a* (*2* ∗ *n*) **by** *auto*

show ∀ *n*. *?f n* ≤ *?f* (*Suc n*)
proof
show *?f n* ≤ *?f* (*Suc n*) **for** *n*
using *mono*[*of 2*∗*n*] **by** *auto*
qed
show ∀ *n*. *?g* (*Suc n*) ≤ *?g n*
proof
show *?g* (*Suc n*) ≤ *?g n* **for** *n*
using *mono*[*of Suc* (*2*∗*n*)] **by** *auto*
qed
show ∀ *n*. *?f n* ≤ *?g n*
proof
show *?f n* ≤ *?g n* **for** *n*
using *fg-diff a-pos* **by** *auto*
qed
show (λ*n*. *?f n* − *?g n*) ⟶ *0*
unfolding *fg-diff*
proof (*rule LIMSEQ-I*)
fix *r* :: *real*

```
    assume 0 < r
     with ‹a ⟶ 0›[THEN LIMSEQ-D] obtain N where ⋀ n. n ≥ N ⟹
norm (a n − 0) < r
      by auto
    then have ∀ n ≥ N. norm (− a (2 * n) − 0) < r
      by auto
    then show ∃ N. ∀ n ≥ N. norm (− a (2 * n) − 0) < r
      by auto
  qed
qed

lemma summable-Leibniz':
  fixes a :: nat ⇒ real
  assumes a-zero: a ⟶ 0
    and a-pos: ⋀n. 0 ≤ a n
    and a-monotone: ⋀n. a (Suc n) ≤ a n
  shows summable: summable (λ n. (−1) ^ n * a n)
    and ⋀n. (∑ i<2*n. (−1) ^ i*a i) ≤ (∑ i. (−1) ^ i*a i)
    and (λn. ∑ i<2*n. (−1) ^ i*a i) ⟶ (∑ i. (−1) ^ i*a i)
    and ⋀n. (∑ i. (−1) ^ i*a i) ≤ (∑ i<2*n+1. (−1) ^ i*a i)
    and (λn. ∑ i<2*n+1. (−1) ^ i*a i) ⟶ (∑ i. (−1) ^ i*a i)
proof −
  let ?S = λn. (−1) ^ n * a n
  let ?P = λn. ∑ i<n. ?S i
  let ?f = λn. ?P (2 * n)
  let ?g = λn. ?P (2 * n + 1)
  obtain l :: real
    where below-l: ∀ n. ?f n ≤ l
      and ?f ⟶ l
      and above-l: ∀ n. l ≤ ?g n
      and ?g ⟶ l
    using sums-alternating-upper-lower[OF a-monotone a-pos a-zero] by blast

  let ?Sa = λm. ∑ n<m. ?S n
  have ?Sa ⟶ l
  proof (rule LIMSEQ-I)
    fix r :: real
    assume 0 < r
    with ‹?f ⟶ l›[THEN LIMSEQ-D]
    obtain f-no where f: ⋀n. n ≥ f-no ⟹ norm (?f n − l) < r
      by auto
    from ‹0 < r› ‹?g ⟶ l›[THEN LIMSEQ-D]
    obtain g-no where g: ⋀n. n ≥ g-no ⟹ norm (?g n − l) < r
      by auto
    have norm (?Sa n − l) < r if n ≥ (max (2 * f-no) (2 * g-no)) for n
    proof −
      from that have n ≥ 2 * f-no and n ≥ 2 * g-no by auto
      show ?thesis
      proof (cases even n)
```

```
      case True
      then have n-eq: 2 * (n div 2) = n
        by (simp add: even-two-times-div-two)
      with ‹n ≥ 2 * f-no› have n div 2 ≥ f-no
        by auto
      from f[OF this] show ?thesis
        unfolding n-eq atLeastLessThanSuc-atLeastAtMost .
    next
      case False
      then have even (n − 1) by simp
      then have n-eq: 2 * ((n − 1) div 2) = n − 1
        by (simp add: even-two-times-div-two)
      then have range-eq: n − 1 + 1 = n
        using odd-pos[OF False] by auto
      from n-eq ‹n ≥ 2 * g-no› have (n − 1) div 2 ≥ g-no
        by auto
      from g[OF this] show ?thesis
        by (simp only: n-eq range-eq)
    qed
  qed
  then show ∃ no. ∀ n ≥ no. norm (?Sa n − l) < r by blast
 qed
 then have sums-l: (λi. (−1) ^ i * a i) sums l
  by (simp only: sums-def)
 then show summable ?S
  by (auto simp: summable-def)

 have l = suminf ?S by (rule sums-unique[OF sums-l])

 fix n
 show suminf ?S ≤ ?g n
  unfolding sums-unique[OF sums-l, symmetric] using above-l by auto
 show ?f n ≤ suminf ?S
  unfolding sums-unique[OF sums-l, symmetric] using below-l by auto
 show ?g ⟶ suminf ?S
  using ‹?g ⟶ l› ‹l = suminf ?S› by auto
 show ?f ⟶ suminf ?S
  using ‹?f ⟶ l› ‹l = suminf ?S› by auto
qed

theorem summable-Leibniz:
 fixes a :: nat ⇒ real
 assumes a-zero: a ⟶ 0
  and monoseq a
 shows summable (λ n. (−1) ^ n * a n) (is ?summable)
  and 0 < a 0 ⟶
   (∀ n. (∑ i. (− 1) ^ i*a i) ∈ { ∑ i<2*n. (− 1) ^ i * a i .. ∑ i<2*n+1. (− 1) ^ i
* a i}) (is ?pos)
  and a 0 < 0 ⟶
```

```
    (∀ n. (∑ i. (− 1) ^ i * a i) ∈ { ∑ i<2*n+1. (− 1) ^ i * a i .. ∑ i<2*n. (− 1) ^ i
* a i}) (is ?neg)
   and (λn. ∑ i<2*n. (− 1) ^ i * a i) ⟶ (∑ i. (− 1) ^ i * a i) (is ?f)
   and (λn. ∑ i<2*n+1. (− 1) ^ i * a i) ⟶ (∑ i. (− 1) ^ i * a i) (is ?g)
proof −
  have ?summable ∧ ?pos ∧ ?neg ∧ ?f ∧ ?g
  proof (cases (∀ n. 0 ≤ a n) ∧ (∀ m. ∀ n≥m. a n ≤ a m))
    case True
    then have ord: ⋀n m. m ≤ n ⟹ a n ≤ a m
      and ge0: ⋀n. 0 ≤ a n
      by auto
    have mono: a (Suc n) ≤ a n for n
      using ord[where n=Suc n and m=n] by auto
    note leibniz = summable-Leibniz'[OF ‹a ⟶ 0› ge0]
    from leibniz[OF mono]
    show ?thesis using ‹0 ≤ a 0› by auto
  next
    let ?a = λn. − a n
    case False
    with monoseq-le[OF ‹monoseq a› ‹a ⟶ 0›]
    have (∀ n. a n ≤ 0) ∧ (∀ m. ∀ n≥m. a m ≤ a n) by auto
    then have ord: ⋀n m. m ≤ n ⟹ ?a n ≤ ?a m and ge0: ⋀ n. 0 ≤ ?a n
      by auto
    have monotone: ?a (Suc n) ≤ ?a n for n
      using ord[where n=Suc n and m=n] by auto
    note leibniz =
      summable-Leibniz'[OF - ge0, of λx. x,
        OF tendsto-minus[OF ‹a ⟶ 0›, unfolded minus-zero] monotone]
    have summable (λ n. (−1) ^ n * ?a n)
      using leibniz(1) by auto
    then obtain l where (λ n. (−1) ^ n * ?a n) sums l
      unfolding summable-def by auto
    from this[THEN sums-minus] have (λ n. (−1) ^ n * a n) sums −l
      by auto
    then have ?summable by (auto simp: summable-def)
    moreover
    have |− a − − b| = |a − b| for a b :: real
      unfolding minus-diff-minus by auto

    from suminf-minus[OF leibniz(1), unfolded mult-minus-right minus-minus]
    have move-minus: (∑ n. − ((− 1) ^ n * a n)) = − (∑ n. (− 1) ^ n * a n)
      by auto

    have ?pos using ‹0 ≤ ?a 0› by auto
    moreover have ?neg
      using leibniz(2,4)
      unfolding mult-minus-right sum-negf move-minus neg-le-iff-le
      by auto
    moreover have ?f and ?g
```

using *leibniz*(*3*,*5*)[*unfolded mult-minus-right sum-negf move-minus*, *THEN tendsto-minus-cancel*]
by *auto*
ultimately show *?thesis* **by** *auto*
qed
then show *?summable* **and** *?pos* **and** *?neg* **and** *?f* **and** *?g*
by *safe*
qed

112.3 Term-by-Term Differentiability of Power Series

definition *diffs* :: (*nat* ⇒ *'a*::*ring-1*) ⇒ *nat* ⇒ *'a*
where *diffs c* = (λ*n*. *of-nat* (*Suc n*) ∗ *c* (*Suc n*))

Lemma about distributing negation over it.

lemma *diffs-minus*: *diffs* (λ*n*. − *c n*) = (λ*n*. − *diffs c n*)
by (*simp add*: *diffs-def*)

lemma *diffs-equiv*:
fixes *x* :: *'a*::{*real-normed-vector*,*ring-1*}
shows *summable* (λ*n*. *diffs c n* ∗ *x*^*n*) ⟹
(λ*n*. *of-nat n* ∗ *c n* ∗ *x*^(*n* − *Suc 0*)) *sums* (∑ *n*. *diffs c n* ∗ *x*^*n*)
unfolding *diffs-def*
by (*simp add*: *summable-sums sums-Suc-imp*)

lemma *lemma-termdiff1*:
fixes *z* :: *'a* :: {*monoid-mult*,*comm-ring*}
shows (∑ *p*<*m*. (((*z* + *h*) ^ (*m* − *p*)) ∗ (*z* ^ *p*)) − (*z* ^ *m*)) =
(∑ *p*<*m*. (*z* ^ *p*) ∗ (((*z* + *h*) ^ (*m* − *p*)) − (*z* ^ (*m* − *p*))))
by (*auto simp*: *algebra-simps power-add* [*symmetric*])

lemma *sumr-diff-mult-const2*: *sum f* {..<*n*} − *of-nat n* ∗ *r* = (∑ *i*<*n*. *f i* − *r*)
for *r* :: *'a*::*ring-1*
by (*simp add*: *sum-subtractf*)

lemma *lemma-termdiff2*:
fixes *h* :: *'a*::*field*
assumes *h*: *h* ≠ *0*
shows ((*z* + *h*) ^ *n* − *z* ^ *n*) / *h* − *of-nat n* ∗ *z* ^ (*n* − *Suc 0*) =
h ∗ (∑ *p*< *n* − *Suc 0*. ∑ *q*< *n* − *Suc 0* − *p*. (*z* + *h*) ^ *q* ∗ *z* ^ (*n* − *2* − *q*))
(**is** *?lhs* = *?rhs*)
proof (*cases n*)
case (*Suc m*)
have *0*: ⋀*x k*. (∑ *n*<*Suc k*. *h* ∗ (*z* ^ *x* ∗ (*z* ^ (*k* − *n*) ∗ (*h* + *z*) ^ *n*))) =
(∑ *j*<*Suc k*. *h* ∗ ((*h* + *z*) ^ *j* ∗ *z* ^ (*x* + *k* − *j*)))
by (*auto simp add*: *power-add* [*symmetric*] *mult.commute intro*: *sum.cong*)
have ∗: (∑ *i*<*m*. *z* ^ *i* ∗ ((*z* + *h*) ^ (*m* − *i*) − *z* ^ (*m* − *i*))) =
(∑ *i*<*m*. ∑ *j*<*m* − *i*. *h* ∗ ((*z* + *h*) ^ *j* ∗ *z* ^ (*m* − *Suc j*)))

by (*force simp add*: *less-iff-Suc-add sum-distrib-left diff-power-eq-sum ac-simps 0*
simp del: *sum.lessThan-Suc power-Suc intro*: *sum.cong*)
have $h * ?lhs = (z + h) \hat{} n - z \hat{} n - h * of\text{-}nat\ n * z \hat{} (n - Suc\ 0)$
by (*simp add*: *right-diff-distrib diff-divide-distrib h mult.assoc* [*symmetric*])
also have $... = h * ((\sum p{<}Suc\ m.\ (z + h) \hat{} p * z \hat{} (m - p)) - of\text{-}nat\ (Suc\ m) * z \hat{} m)$
by (*simp add*: *Suc diff-power-eq-sum h right-diff-distrib* [*symmetric*] *mult.assoc*
del: *power-Suc sum.lessThan-Suc of-nat-Suc*)
also have $... = h * ((\sum p{<}Suc\ m.\ (z + h) \hat{} (m - p) * z \hat{} p) - of\text{-}nat\ (Suc\ m) * z \hat{} m)$
by (*subst sum.nat-diff-reindex*[*symmetric*]) *simp*
also have $... = h * (\sum i{<}Suc\ m.\ (z + h) \hat{} (m - i) * z \hat{} i - z \hat{} m)$
by (*simp add*: *sum-subtractf*)
also have $... = h * ?rhs$
by (*simp add*: *lemma-termdiff1 sum-distrib-left Suc* $*$)
finally have $h * ?lhs = h * ?rhs$ **.**
then show *?thesis*
by (*simp add*: *h*)
qed *auto*

lemma *real-sum-nat-ivl-bounded2*:
fixes $K :: 'a{::}linordered\text{-}semidom$
assumes *f*: $\bigwedge p{::}nat.\ p < n \Longrightarrow f\ p \leq K$ **and** *K*: $0 \leq K$
shows $sum\ f\ \{..{<}n{-}k\} \leq of\text{-}nat\ n * K$
proof −
have $sum\ f\ \{..{<}n{-}k\} \leq (\sum i{<}n - k.\ K)$
by (*rule sum-mono* [*OF f*]) *auto*
also have $... \leq of\text{-}nat\ n * K$
by (*auto simp*: *mult-right-mono K*)
finally show *?thesis* **.**
qed

lemma *lemma-termdiff3*:
fixes *h z* :: $'a{::}real\text{-}normed\text{-}field$
assumes *1*: $h \neq 0$
and *2*: $norm\ z \leq K$
and *3*: $norm\ (z + h) \leq K$
shows $norm\ (((z + h) \hat{} n - z \hat{} n)\ /\ h - of\text{-}nat\ n * z \hat{} (n - Suc\ 0)) \leq$
$of\text{-}nat\ n * of\text{-}nat\ (n - Suc\ 0) * K \hat{} (n - 2) * norm\ h$
proof −
have $norm\ (((z + h) \hat{} n - z \hat{} n)\ /\ h - of\text{-}nat\ n * z \hat{} (n - Suc\ 0)) =$
$norm\ (\sum p{<}n - Suc\ 0.\ \sum q{<}n - Suc\ 0 - p.\ (z + h) \hat{} q * z \hat{} (n - 2 - q)) * norm\ h$
by (*metis* (*lifting*, *no-types*) *lemma-termdiff2* [*OF 1*] *mult.commute norm-mult*)
also have $\ldots \leq of\text{-}nat\ n * (of\text{-}nat\ (n - Suc\ 0) * K \hat{} (n - 2)) * norm\ h$
proof (*rule mult-right-mono* [*OF - norm-ge-zero*])
from *norm-ge-zero 2* **have** *K*: $0 \leq K$

```
    by (rule order-trans)
  have le-Kn: norm ((z + h) ^ i * z ^ j) ≤ K ^ n if i + j = n for i j n
  proof −
    have norm (z + h) ^ i * norm z ^ j ≤ K ^ i * K ^ j
      by (intro mult-mono power-mono 2 3 norm-ge-zero zero-le-power K)
    also have ... = K^n
      by (metis power-add that)
    finally show ?thesis
      by (simp add: norm-mult norm-power)
  qed
  then have ⋀p q.
    ⟦p < n; q < n − Suc 0⟧ ⟹ norm ((z + h) ^ q * z ^ (n − 2 − q)) ≤ K ^
(n − 2)
    by (simp del: subst-all)
  then
  show norm (∑p<n − Suc 0. ∑q<n − Suc 0 − p. (z + h) ^ q * z ^ (n − 2
− q)) ≤
      of-nat n * (of-nat (n − Suc 0) * K ^ (n − 2))
    by (intro order-trans [OF norm-sum]
        real-sum-nat-ivl-bounded2 mult-nonneg-nonneg of-nat-0-le-iff zero-le-power
K)
 qed
 also have ... = of-nat n * of-nat (n − Suc 0) * K ^ (n − 2) * norm h
   by (simp only: mult.assoc)
 finally show ?thesis .
qed

lemma lemma-termdiff4:
 fixes f :: 'a::real-normed-vector ⇒ 'b::real-normed-vector
   and k :: real
 assumes k: 0 < k
   and le: ⋀h. h ≠ 0 ⟹ norm h < k ⟹ norm (f h) ≤ K * norm h
 shows f −0→ 0
proof (rule tendsto-norm-zero-cancel)
 show (λh. norm (f h)) −0→ 0
 proof (rule real-tendsto-sandwich)
   show eventually (λh. 0 ≤ norm (f h)) (at 0)
     by simp
   show eventually (λh. norm (f h) ≤ K * norm h) (at 0)
     using k by (auto simp: eventually-at dist-norm le)
   show (λh. 0) −(0::'a)→ (0::real)
     by (rule tendsto-const)
   have (λh. K * norm h) −(0::'a)→ K * norm (0::'a)
     by (intro tendsto-intros)
   then show (λh. K * norm h) −(0::'a)→ 0
     by simp
 qed
qed
```

lemma *lemma-termdiff5*:
fixes $g :: 'a::real\text{-}normed\text{-}vector \Rightarrow nat \Rightarrow 'b::banach$
and $k :: real$
assumes k: $0 < k$
and f: $summable\ f$
and le: $\bigwedge h\ n.\ h \neq 0 \Longrightarrow norm\ h < k \Longrightarrow norm\ (g\ h\ n) \leq f\ n * norm\ h$
shows $(\lambda h.\ suminf\ (g\ h)) -0\rightarrow 0$
proof (*rule lemma-termdiff4* [*OF k*])
fix $h :: 'a$
assume $h \neq 0$ **and** $norm\ h < k$
then have *1*: $\forall n.\ norm\ (g\ h\ n) \leq f\ n * norm\ h$
by (*simp add*: *le*)
then have $\exists N.\ \forall n \geq N.\ norm\ (norm\ (g\ h\ n)) \leq f\ n * norm\ h$
by *simp*
moreover from *f* **have** *2*: $summable\ (\lambda n.\ f\ n * norm\ h)$
by (*rule summable-mult2*)
ultimately have *3*: $summable\ (\lambda n.\ norm\ (g\ h\ n))$
by (*rule summable-comparison-test*)
then have $norm\ (suminf\ (g\ h)) \leq (\sum n.\ norm\ (g\ h\ n))$
by (*rule summable-norm*)
also from *1 3 2* **have** $(\sum n.\ norm\ (g\ h\ n)) \leq (\sum n.\ f\ n * norm\ h)$
by (*simp add*: *suminf-le*)
also from *f* **have** $(\sum n.\ f\ n * norm\ h) = suminf\ f * norm\ h$
by (*rule suminf-mult2* [*symmetric*])
finally show $norm\ (suminf\ (g\ h)) \leq suminf\ f * norm\ h$ **.**
qed

lemma *termdiffs-aux*:
fixes $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
assumes *1*: $summable\ (\lambda n.\ diffs\ (diffs\ c)\ n * K\ \hat{}\ n)$
and *2*: $norm\ x < norm\ K$
shows $(\lambda h.\ \sum n.\ c\ n * (((x + h)\ \hat{}\ n - x\hat{}n)\ /\ h - of\text{-}nat\ n * x\ \hat{}\ (n - Suc\ 0)))$
$-0\rightarrow 0$
proof −
from *dense* [*OF 2*] **obtain** *r* **where** *r1*: $norm\ x < r$ **and** *r2*: $r < norm\ K$
by *fast*
from *norm-ge-zero r1* **have** *r*: $0 < r$
by (*rule order-le-less-trans*)
then have *r-neq-0*: $r \neq 0$ **by** *simp*
show *?thesis*
proof (*rule lemma-termdiff5*)
show $0 < r - norm\ x$
using *r1* **by** *simp*
from *r r2* **have** $norm\ (of\text{-}real\ r::'a) < norm\ K$
by *simp*
with *1* **have** $summable\ (\lambda n.\ norm\ (diffs\ (diffs\ c)\ n * (of\text{-}real\ r\ \hat{}\ n)))$

```
    by (rule powser-insidea)
  then have summable (λn. diffs (diffs (λn. norm (c n))) n * r ^ n)
    using r by (simp add: diffs-def norm-mult norm-power del: of-nat-Suc)
  then have summable (λn. of-nat n * diffs (λn. norm (c n)) n * r ^ (n - Suc
0))
    by (rule diffs-equiv [THEN sums-summable])
  also have (λn. of-nat n * diffs (λn. norm (c n)) n * r ^ (n - Suc 0)) =
            (λn. diffs (λm. of-nat (m - Suc 0) * norm (c m) * inverse r) n * (r
^ n))
    by (simp add: diffs-def r-neq-0 fun-eq-iff split: nat-diff-split)
  finally have summable
   (λn. of-nat n * (of-nat (n - Suc 0) * norm (c n) * inverse r) * r ^ (n - Suc
0))
    by (rule diffs-equiv [THEN sums-summable])
  also have
   (λn. of-nat n * (of-nat (n - Suc 0) * norm (c n) * inverse r) * r ^ (n - Suc
0)) =
     (λn. norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2))
    by (rule ext) (simp add: r-neq-0 split: nat-diff-split)
  finally show summable (λn. norm (c n) * of-nat n * of-nat (n - Suc 0) * r
^ (n - 2)) .
 next
  fix h :: 'a and n
  assume h: h ≠ 0
  assume norm h < r - norm x
  then have norm x + norm h < r by simp
  with norm-triangle-ineq
  have xh: norm (x + h) < r
    by (rule order-le-less-trans)
  have norm (((x + h) ^ n - x ^ n) / h - of-nat n * x ^ (n - Suc 0))
  ≤ real n * (real (n - Suc 0) * (r ^ (n - 2) * norm h))
     by (metis (mono-tags, lifting) h mult.assoc lemma-termdiff3 less-eq-real-def
r1 xh)
  then show norm (c n * (((x + h) ^ n - x^n) / h - of-nat n * x ^ (n - Suc
0))) ≤
    norm (c n) * of-nat n * of-nat (n - Suc 0) * r ^ (n - 2) * norm h
    by (simp only: norm-mult mult.assoc mult-left-mono [OF - norm-ge-zero])
 qed
qed

lemma termdiffs:
 fixes K x :: 'a::{real-normed-field,banach}
 assumes 1: summable (λn. c n * K ^ n)
   and 2: summable (λn. (diffs c) n * K ^ n)
   and 3: summable (λn. (diffs (diffs c)) n * K ^ n)
   and 4: norm x < norm K
 shows DERIV (λx. ∑n. c n * x^n) x :> (∑n. (diffs c) n * x^n)
 unfolding DERIV-def
proof (rule LIM-zero-cancel)
```

show (λh. (*suminf* (λn. *c n* ∗ (*x* + *h*) ^ *n*) − *suminf* (λn. *c n* ∗ *x*^*n*)) / *h*
− *suminf* (λn. *diffs c n* ∗ *x*^*n*)) −0→ 0
proof (*rule LIM-equal2*)
show 0 < *norm K* − *norm x*
using *4* **by** (*simp add*: *less-diff-eq*)
next
fix *h* :: *'a*
assume *norm* (*h* − 0) < *norm K* − *norm x*
then have *norm x* + *norm h* < *norm K* **by** *simp*
then have *5*: *norm* (*x* + *h*) < *norm K*
by (*rule norm-triangle-ineq* [*THEN order-le-less-trans*])
have *summable* (λn. *c n* ∗ *x*^*n*)
and *summable* (λn. *c n* ∗ (*x* + *h*) ^ *n*)
and *summable* (λn. *diffs c n* ∗ *x*^*n*)
using *1 2 4 5* **by** (*auto elim*: *powser-inside*)
then have ((∑ n. *c n* ∗ (*x* + *h*) ^ *n*) − (∑ n. *c n* ∗ *x*^*n*)) / *h* − (∑ n. *diffs c n* ∗ *x*^*n*) =
(∑ n. (*c n* ∗ (*x* + *h*) ^ *n* − *c n* ∗ *x*^*n*) / *h* − *of-nat n* ∗ *c n* ∗ *x* ^ (*n* − *Suc 0*))
by (*intro sums-unique sums-diff sums-divide diffs-equiv summable-sums*)
then show ((∑ n. *c n* ∗ (*x* + *h*) ^ *n*) − (∑ n. *c n* ∗ *x*^*n*)) / *h* − (∑ n. *diffs c n* ∗ *x*^*n*) =
(∑ n. *c n* ∗ (((*x* + *h*) ^ *n* − *x*^*n*) / *h* − *of-nat n* ∗ *x* ^ (*n* − *Suc 0*)))
by (*simp add*: *algebra-simps*)
next
show (λh. ∑ n. *c n* ∗ (((*x* + *h*) ^ *n* − *x*^*n*) / *h* − *of-nat n* ∗ *x* ^ (*n* − *Suc 0*))) −0→ 0
by (*rule termdiffs-aux* [*OF 3 4*])
qed
qed

112.4 The Derivative of a Power Series Has the Same Radius of Convergence

lemma *termdiff-converges*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
assumes *K*: *norm x* < *K*
and *sm*: ⋀x. *norm x* < *K* ⟹ *summable*(λn. *c n* ∗ *x* ^ *n*)
shows *summable* (λn. *diffs c n* ∗ *x* ^ *n*)
proof (*cases x* = 0)
case *True*
then show *?thesis*
using *powser-sums-zero sums-summable* **by** *auto*
next
case *False*
then have *K* > 0
using *K less-trans zero-less-norm-iff* **by** *blast*
then obtain *r* :: *real* **where** *r*: *norm x* < *norm r norm r* < *K r* > 0
using *K False*

by (*auto simp*: *field-simps abs-less-iff add-pos-pos intro*: *that* [*of* (*norm x* + *K*) / *2*])
have *to0*: (λ*n*. *of-nat n* ∗ (*x* / *of-real r*) ^ *n*) ⟶ *0*
using *r* **by** (*simp add*: *norm-divide powser-times-n-limit-0* [*of x* / *of-real r*])
obtain *N* **where** *N*: ⋀*n*. *n*≥*N* ⟹ *real-of-nat n* ∗ *norm x* ^ *n* < *r* ^ *n*
using *r LIMSEQ-D* [*OF to0*, *of 1*]
by (*auto simp*: *norm-divide norm-mult norm-power field-simps*)
have *summable* (λ*n*. (*of-nat n* ∗ *c n*) ∗ *x* ^ *n*)
proof (*rule summable-comparison-test'*)
show *summable* (λ*n*. *norm* (*c n* ∗ *of-real r* ^ *n*))
apply (*rule powser-insidea* [*OF sm* [*of of-real* ((*r*+*K*)/*2*)]])
using *N r norm-of-real* [*of r* + *K*, **where** *'a* = *'a*] **by** *auto*
show ⋀*n*. *N* ≤ *n* ⟹ *norm* (*of-nat n* ∗ *c n* ∗ *x* ^ *n*) ≤ *norm* (*c n* ∗ *of-real r* ^ *n*)
using *N r* **by** (*fastforce simp add*: *norm-mult norm-power less-eq-real-def*)
qed
then have *summable* (λ*n*. (*of-nat* (*Suc n*) ∗ *c*(*Suc n*)) ∗ *x* ^ *Suc n*)
using *summable-iff-shift* [*of* λ*n*. *of-nat n* ∗ *c n* ∗ *x* ^ *n 1*]
by *simp*
then have *summable* (λ*n*. (*of-nat* (*Suc n*) ∗ *c*(*Suc n*)) ∗ *x* ^ *n*)
using *False summable-mult2* [*of* λ*n*. (*of-nat* (*Suc n*) ∗ *c*(*Suc n*) ∗ *x* ^ *n*) ∗ *x inverse x*]
by (*simp add*: *mult.assoc*) (*auto simp*: *ac-simps*)
then show *?thesis*
by (*simp add*: *diffs-def*)
qed

lemma *termdiff-converges-all*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
assumes ⋀*x*. *summable* (λ*n*. *c n* ∗ *x*^*n*)
shows *summable* (λ*n*. *diffs c n* ∗ *x*^*n*)
by (*rule termdiff-converges* [**where** *K* = *1* + *norm x*]) (*use assms* **in** *auto*)

lemma *termdiffs-strong*:
fixes *K x* :: *'a*::{*real-normed-field*,*banach*}
assumes *sm*: *summable* (λ*n*. *c n* ∗ *K* ^ *n*)
and *K*: *norm x* < *norm K*
shows *DERIV* (λ*x*. ∑ *n*. *c n* ∗ *x*^*n*) *x* :> (∑ *n*. *diffs c n* ∗ *x*^*n*)
proof −
have *norm K* + *norm x* < *norm K* + *norm K*
using *K* **by** *force*
then have *K2*: *norm* ((*of-real* (*norm K*) + *of-real* (*norm x*)) / *2* :: *'a*) < *norm K*
by (*auto simp*: *norm-triangle-lt norm-divide field-simps*)
then have [*simp*]: *norm* ((*of-real* (*norm K*) + *of-real* (*norm x*)) :: *'a*) < *norm K* ∗ *2*
by *simp*
have *summable* (λ*n*. *c n* ∗ (*of-real* (*norm x* + *norm K*) / *2*) ^ *n*)
by (*metis K2 summable-norm-cancel* [*OF powser-insidea* [*OF sm*]] *add.commute*

```
of-real-add)
  moreover have ⋀x. norm x < norm K ⟹ summable (λn. diffs c n * x ^ n)
    by (blast intro: sm termdiff-converges powser-inside)
  moreover have ⋀x. norm x < norm K ⟹ summable (λn. diffs(diffs c) n * x
^ n)
    by (blast intro: sm termdiff-converges powser-inside)
  ultimately show ?thesis
    by (rule termdiffs [where K = of-real (norm x + norm K) / 2])
       (use K in ‹auto simp: field-simps simp flip: of-real-add›)
qed

lemma termdiffs-strong-converges-everywhere:
  fixes K x :: 'a::{real-normed-field,banach}
  assumes ⋀y. summable (λn. c n * y ^ n)
  shows ((λx. ∑ n. c n * x^n) has-field-derivative (∑ n. diffs c n * x^n)) (at x)
  using termdiffs-strong[OF assms[of of-real (norm x + 1)], of x]
  by (force simp del: of-real-add)

lemma termdiffs-strong':
  fixes z :: 'a :: {real-normed-field,banach}
  assumes ⋀z. norm z < K ⟹ summable (λn. c n * z ^ n)
  assumes norm z < K
  shows   ((λz. ∑ n. c n * z^n) has-field-derivative (∑ n. diffs c n * z^n)) (at z)
proof (rule termdiffs-strong)
  define L :: real where L =  (norm z + K) / 2
  have 0 ≤ norm z by simp
  also note ‹norm z < K›
  finally have K: K ≥ 0 by simp
  from assms K have L: L ≥ 0 norm z < L L < K by (simp-all add: L-def)
  from L show norm z < norm (of-real L :: 'a) by simp
  from L show summable (λn. c n * of-real L ^ n) by (intro assms(1)) simp-all
qed

lemma termdiffs-sums-strong:
  fixes z :: 'a :: {banach,real-normed-field}
  assumes sums: ⋀z. norm z < K ⟹ (λn. c n * z ^ n) sums f z
  assumes deriv: (f has-field-derivative f') (at z)
  assumes norm: norm z < K
  shows   (λn. diffs c n * z ^ n) sums f'
proof -
  have summable: summable (λn. diffs c n * z^n)
    by (intro termdiff-converges[OF norm] sums-summable[OF sums])
  from norm have eventually (λz. z ∈ norm -‘ {..<K}) (nhds z)
    by (intro eventually-nhds-in-open open-vimage)
       (simp-all add: continuous-on-norm)
  hence eq: eventually (λz. (∑ n. c n * z^n) = f z) (nhds z)
    by eventually-elim (insert sums, simp add: sums-iff)

  have ((λz. ∑ n. c n * z^n) has-field-derivative (∑ n. diffs c n * z^n)) (at z)
```

```
    by (intro termdiffs-strong'[OF - norm] sums-summable[OF sums])
  hence (f has-field-derivative (∑ n. diffs c n * z^n)) (at z)
    by (subst (asm) DERIV-cong-ev[OF refl eq refl])
  from this and deriv have (∑ n. diffs c n * z^n) = f' by (rule DERIV-unique)
  with summable show ?thesis by (simp add: sums-iff)
qed

lemma isCont-powser:
  fixes K x :: 'a::{real-normed-field,banach}
  assumes summable (λn. c n * K ^ n)
  assumes norm x < norm K
  shows isCont (λx. ∑ n. c n * x^n) x
  using termdiffs-strong[OF assms] by (blast intro!: DERIV-isCont)

lemmas isCont-powser' = isCont-o2[OF - isCont-powser]

lemma isCont-powser-converges-everywhere:
  fixes K x :: 'a::{real-normed-field,banach}
  assumes ⋀y. summable (λn. c n * y ^ n)
  shows isCont (λx. ∑ n. c n * x^n) x
  using termdiffs-strong[OF assms[of of-real (norm x + 1)], of x]
  by (force intro!: DERIV-isCont simp del: of-real-add)

lemma powser-limit-0:
  fixes a :: nat ⇒ 'a::{real-normed-field,banach}
  assumes s: 0 < s
    and sm: ⋀x. norm x < s ⟹ (λn. a n * x ^ n) sums (f x)
  shows (f ⟶ a 0) (at 0)
proof −
  have norm (of-real s / 2 :: 'a) < s
    using s  by (auto simp: norm-divide)
  then have summable (λn. a n * (of-real s / 2) ^ n)
    by (rule sums-summable [OF sm])
  then have ((λx. ∑ n. a n * x ^ n) has-field-derivative (∑ n. diffs a n * 0 ^ n))
(at 0)
    by (rule termdiffs-strong) (use s in ‹auto simp: norm-divide›)
  then have isCont (λx. ∑ n. a n * x ^ n) 0
    by (blast intro: DERIV-continuous)
  then have ((λx. ∑ n. a n * x ^ n) ⟶ a 0) (at 0)
    by (simp add: continuous-within)
  moreover have (λx. f x − (∑ n. a n * x ^ n)) −0→ 0
    apply (clarsimp simp: LIM-eq)
    apply (rule-tac x=s in exI)
    using s sm sums-unique by fastforce
  ultimately show ?thesis
    by (rule Lim-transform)
qed

lemma powser-limit-0-strong:
```

fixes $a :: nat \Rightarrow 'a::\{real\text{-}normed\text{-}field,banach\}$
assumes s: $0 < s$
and sm: $\bigwedge x.\ x \neq 0 \Longrightarrow norm\ x < s \Longrightarrow (\lambda n.\ a\ n * x\ \hat{}\ n)\ sums\ (f\ x)$
shows $(f \longrightarrow a\ 0)\ (at\ 0)$
proof −
have $*$: $((\lambda x.\ if\ x = 0\ then\ a\ 0\ else\ f\ x) \longrightarrow a\ 0)\ (at\ 0)$
by (*rule powser-limit-0* [*OF s*]) (*auto simp*: *powser-sums-zero sm*)
show *?thesis*
using $*$ **by** (*auto cong*: *Lim-cong-within*)
qed

112.5 Derivability of power series

lemma *DERIV-series'*:
fixes $f :: real \Rightarrow nat \Rightarrow real$
assumes *DERIV-f*: $\bigwedge\ n.\ DERIV\ (\lambda\ x.\ f\ x\ n)\ x0\ :>\ (f'\ x0\ n)$
and *allf-summable*: $\bigwedge\ x.\ x \in \{a\ <..<\ b\} \Longrightarrow summable\ (f\ x)$
and *x0-in-I*: $x0 \in \{a\ <..<\ b\}$
and $summable\ (f'\ x0)$
and $summable\ L$
and *L-def*: $\bigwedge n\ x\ y.\ x \in \{a\ <..<\ b\} \Longrightarrow y \in \{a\ <..<\ b\} \Longrightarrow |f\ x\ n - f\ y\ n| \leq L\ n * |x - y|$
shows $DERIV\ (\lambda\ x.\ suminf\ (f\ x))\ x0\ :>\ (suminf\ (f'\ x0))$
unfolding *DERIV-def*
proof (*rule LIM-I*)
fix $r :: real$
assume $0 < r$ **then have** $0 < r/3$ **by** *auto*

obtain *N-L* **where** *N-L*: $\bigwedge\ n.\ N\text{-}L \leq n \Longrightarrow |\sum\ i.\ L\ (i + n)| < r/3$
using *suminf-exist-split*[*OF* ‹$0 < r/3$› ‹*summable L*›] **by** *auto*

obtain *N-f'* **where** *N-f'*: $\bigwedge\ n.\ N\text{-}f' \leq n \Longrightarrow |\sum\ i.\ f'\ x0\ (i + n)| < r/3$
using *suminf-exist-split*[*OF* ‹$0 < r/3$› ‹$summable\ (f'\ x0)$›] **by** *auto*

let *?N* = $Suc\ (max\ N\text{-}L\ N\text{-}f')$
have $|\sum\ i.\ f'\ x0\ (i + ?N)| < r/3$ (**is** *?f'-part* $< r/3$)
and *L-estimate*: $|\sum\ i.\ L\ (i + ?N)| < r/3$
using *N-L*[*of ?N*] **and** *N-f'* [*of ?N*] **by** *auto*

let *?diff* = $\lambda i\ x.\ (f\ (x0 + x)\ i - f\ x0\ i)\ /\ x$

let *?r* = $r\ /\ (3 * real\ ?N)$
from ‹$0 < r$› **have** $0 < ?r$ **by** *simp*

let *?s* = $\lambda n.\ SOME\ s.\ 0 < s \wedge (\forall\ x.\ x \neq 0 \wedge |x| < s \longrightarrow |\ ?diff\ n\ x - f'\ x0\ n| < ?r)$
define S' **where** $S' = Min\ (?s\ `\ \{..<\ ?N\ \})$

have $0 < S'$

```
    unfolding S'-def
  proof (rule iffD2[OF Min-gr-iff])
    show ∀x∈(?s ` {..< ?N }). 0 < x
    proof
      fix x
      assume x ∈ ?s ` {..<?N}
      then obtain n where x = ?s n and n ∈ {..<?N}
        using image-iff[THEN iffD1] by blast
       from DERIV-D[OF DERIV-f[where n=n], THEN LIM-D, OF ‹0 < ?r›,
unfolded real-norm-def]
      obtain s where s-bound: 0 < s ∧ (∀x. x ≠ 0 ∧ |x| < s ⟶ |?diff n x - f'
x0 n| < ?r)
        by auto
      have 0 < ?s n
        by (rule someI2[where a=s]) (auto simp: s-bound simp del: of-nat-Suc)
      then show 0 < x by (simp only: ‹x = ?s n›)
    qed
  qed auto

  define S where S = min (min (x0 - a) (b - x0)) S'
  then have 0 < S and S-a: S ≤ x0 - a and S-b: S ≤ b - x0
    and S ≤ S' using x0-in-I and ‹0 < S'›
    by auto

  have |(suminf (f (x0 + x)) - suminf (f x0)) / x - suminf (f' x0)| < r
    if x ≠ 0 and |x| < S for x
  proof -
    from that have x-in-I: x0 + x ∈ {a <..< b}
      using S-a S-b by auto

   note diff-smbl = summable-diff[OF allf-summable[OF x-in-I] allf-summable[OF
x0-in-I]]
    note div-smbl = summable-divide[OF diff-smbl]
    note all-smbl = summable-diff[OF div-smbl ‹summable (f' x0)›]
    note ign = summable-ignore-initial-segment[where k=?N]
      note diff-shft-smbl = summable-diff[OF ign[OF allf-summable[OF x-in-I]]
ign[OF allf-summable[OF x0-in-I]]]
    note div-shft-smbl = summable-divide[OF diff-shft-smbl]
    note all-shft-smbl = summable-diff[OF div-smbl ign[OF ‹summable (f' x0)›]]

    have 1: |(|?diff (n + ?N) x|)| ≤ L (n + ?N) for n
    proof -
      have |?diff (n + ?N) x| ≤ L (n + ?N) * |(x0 + x) - x0| / |x|
        using divide-right-mono[OF L-def[OF x-in-I x0-in-I] abs-ge-zero]
        by (simp only: abs-divide)
      with ‹x ≠ 0› show ?thesis by auto
    qed
    note 2 = summable-rabs-comparison-test[OF - ign[OF ‹summable L›]]
    from 1 have |∑ i. ?diff (i + ?N) x| ≤ (∑ i. L (i + ?N))
```

by (*metis* (*lifting*) *abs-idempotent*
order-trans[*OF summable-rabs*[*OF 2*] *suminf-le*[*OF - 2 ign*[*OF ‹summable L›*]]])
then have $|\sum i.\ ?diff\ (i + ?N)\ x| \leq r\ /\ 3$ (**is** *?L-part* $\leq r/3$)
using *L-estimate* **by** *auto*

have $|\sum n{<}?N.\ ?diff\ n\ x - f'\ x0\ n| \leq (\sum n{<}?N.\ |?diff\ n\ x - f'\ x0\ n|)$ **..**
also have $\ldots < (\sum n{<}?N.\ ?r)$
proof (*rule sum-strict-mono*)
fix *n*
assume $n \in \{..<?N\}$
have $|x| < S$ **using** ‹$|x| < S$› **.**
also have $S \leq S'$ **using** ‹$S \leq S'$› **.**
also have $S' \leq ?s\ n$
unfolding *S'-def*
proof (*rule Min-le-iff*[*THEN iffD2*])
have $?s\ n \in (?s\ `\ \{..<?N\}) \wedge ?s\ n \leq ?s\ n$
using ‹$n \in \{..<?N\}$› **by** *auto*
then show $\exists\ a \in (?s\ `\ \{..<?N\}).\ a \leq ?s\ n$
by *blast*
qed *auto*
finally have $|x| < ?s\ n$ **.**

from *DERIV-D*[*OF DERIV-f*[**where** *n=n*], *THEN LIM-D*, *OF* ‹$0 < ?r$›,
unfolded real-norm-def diff-0-right, *unfolded some-eq-ex*[*symmetric*], *THEN conjunct2*]
have $\forall x.\ x \neq 0 \wedge |x| < ?s\ n \longrightarrow |?diff\ n\ x - f'\ x0\ n| < ?r$ **.**
with ‹$x \neq 0$› **and** ‹$|x| < ?s\ n$› **show** $|?diff\ n\ x - f'\ x0\ n| < ?r$
by *blast*
qed *auto*
also have $\ldots = of\text{-}nat\ (card\ \{..<?N\}) * ?r$
by (*rule sum-constant*)
also have $\ldots = real\ ?N * ?r$
by *simp*
also have $\ldots = r/3$
by (*auto simp del*: *of-nat-Suc*)
finally have $|\sum n{<}?N.\ ?diff\ n\ x - f'\ x0\ n\ | < r\ /\ 3$ (**is** *?diff-part* $< r\ /\ 3$) **.**

from *suminf-diff*[*OF allf-summable*[*OF x-in-I*] *allf-summable*[*OF x0-in-I*]]
have $|(suminf\ (f\ (x0 + x)) - (suminf\ (f\ x0)))\ /\ x - suminf\ (f'\ x0)| =$
$|\sum n.\ ?diff\ n\ x - f'\ x0\ n|$
unfolding *suminf-diff*[*OF div-smbl* ‹*summable* $(f'\ x0)$›, *symmetric*]
using *suminf-divide*[*OF diff-smbl*, *symmetric*] **by** *auto*
also have $\ldots \leq ?diff\text{-}part + |(\sum n.\ ?diff\ (n + ?N)\ x) - (\sum\ n.\ f'\ x0\ (n + ?N))|$
unfolding *suminf-split-initial-segment*[*OF all-smbl*, **where** *k=?N*]
unfolding *suminf-diff*[*OF div-shft-smbl ign*[*OF* ‹*summable* $(f'\ x0)$›]]
apply (*simp only*: *add.commute*)
using *abs-triangle-ineq* **by** *blast*

```
    also have ... ≤ ?diff-part + ?L-part + ?f'-part
      using abs-triangle-ineq4 by auto
    also have ... < r /3 + r/3 + r/3
      using ‹?diff-part < r/3› ‹?L-part ≤ r/3› and ‹?f'-part < r/3›
      by (rule add-strict-mono [OF add-less-le-mono])
    finally show ?thesis
      by auto
  qed
  then show ∃ s > 0. ∀ x. x ≠ 0 ∧ norm (x − 0) < s ⟶
      norm (((∑ n. f (x0 + x) n) − (∑ n. f x0 n)) / x − (∑ n. f' x0 n)) < r
    using ‹0 < S› by auto
qed

lemma DERIV-power-series':
  fixes f :: nat ⇒ real
  assumes converges: ⋀x. x ∈ {−R <..< R} ⟹ summable (λn. f n * real (Suc
n) * x^n)
    and x0-in-I: x0 ∈ {−R <..< R}
    and 0 < R
  shows DERIV (λx. (∑ n. f n * x^(Suc n))) x0 :> (∑ n. f n * real (Suc n) *
x0^n)
    (is DERIV (λx. suminf (?f x)) x0 :> suminf (?f' x0))
proof −
  have for-subinterval: DERIV (λx. suminf (?f x)) x0 :> suminf (?f' x0)
    if 0 < R' and R' < R and −R' < x0 and x0 < R' for R'
  proof −
    from that have x0 ∈ {−R' <..< R'} and R' ∈ {−R <..< R} and x0 ∈ {−R
<..< R}
      by auto
    show ?thesis
    proof (rule DERIV-series')
      show summable (λ n. |f n * real (Suc n) * R'^n|)
      proof −
        have (R' + R) / 2 < R and 0 < (R' + R) / 2
          using ‹0 < R'› ‹0 < R› ‹R' < R› by (auto simp: field-simps)
        then have in-Rball: (R' + R) / 2 ∈ {−R <..< R}
          using ‹R' < R› by auto
        have norm R' < norm ((R' + R) / 2)
          using ‹0 < R'› ‹0 < R› ‹R' < R› by (auto simp: field-simps)
        from powser-insidea[OF converges[OF in-Rball] this] show ?thesis
          by auto
      qed
    next
      fix n x y
      assume x ∈ {−R' <..< R'} and y ∈ {−R' <..< R'}
      show |?f x n − ?f y n| ≤ |f n * real (Suc n) * R'^n| * |x−y|
      proof −
        have |f n * x ^ (Suc n) − f n * y ^ (Suc n)| =
          (|f n| * |x−y|) * |∑ p<Suc n. x ^ p * y ^ (n − p)|
```

```
      unfolding right-diff-distrib[symmetric] diff-power-eq-sum abs-mult
      by auto
    also have ... ≤ (|f n| * |x-y|) * (|real (Suc n)| * |R' ^ n|)
    proof (rule mult-left-mono)
     have |∑ p<Suc n. x ^ p * y ^ (n - p)| ≤ (∑ p<Suc n. |x ^ p * y ^ (n -
p)|)
       by (rule sum-abs)
     also have ... ≤ (∑ p<Suc n. R' ^ n)
     proof (rule sum-mono)
      fix p
      assume p ∈ {..<Suc n}
      then have p ≤ n by auto
      have |x^n| ≤ R'^n if  x ∈ {-R'<..<R'} for n and x :: real
      proof -
       from that have |x| ≤ R' by auto
       then show ?thesis
        unfolding power-abs by (rule power-mono) auto
      qed
       from mult-mono[OF this[OF ‹x ∈ {-R'<..<R'}›, of p] this[OF ‹y ∈
{-R'<..<R'}›, of n-p]]
        and ‹0 < R'›
      have |x^p * y^(n - p)| ≤ R'^p * R'^(n - p)
       unfolding abs-mult by auto
      then show |x^p * y^(n - p)| ≤ R'^n
       unfolding power-add[symmetric] using ‹p ≤ n› by auto
     qed
     also have ... = real (Suc n) * R' ^ n
      unfolding sum-constant card-atLeastLessThan by auto
    finally show |∑ p<Suc n. x ^ p * y ^ (n - p)| ≤ |real (Suc n)| * |R' ^ n|
     unfolding abs-of-nonneg[OF zero-le-power[OF less-imp-le[OF ‹0 < R'›]]]
      by linarith
     show 0 ≤ |f n| * |x - y|
      unfolding abs-mult[symmetric] by auto
   qed
   also have ... = |f n * real (Suc n) * R' ^ n| * |x - y|
    unfolding abs-mult mult.assoc[symmetric] by algebra
   finally show ?thesis .
  qed
 next
  show DERIV (λx. ?f x n) x0 :> ?f' x0 n for n
   by (auto intro!: derivative-eq-intros simp del: power-Suc)
 next
  fix x
  assume x ∈ {-R' <..< R'}
  then have R' ∈ {-R <..< R} and norm x < norm R'
   using assms ‹R' < R› by auto
  have summable (λn. f n * x^n)
  proof (rule summable-comparison-test, intro exI allI impI)
   fix n
```

```
      have le: |f n| * 1 ≤ |f n| * real (Suc n)
        by (rule mult-left-mono) auto
      show norm (f n * x^n) ≤ norm (f n * real (Suc n) * x^n)
        unfolding real-norm-def abs-mult
        using le mult-right-mono by fastforce
    qed (rule powser-insidea[OF converges[OF ‹R' ∈ {−R <..< R}›] ‹norm x <
norm R'›])
    from this[THEN summable-mult2[where c=x], simplified mult.assoc, simpli-
fied mult.commute]
    show summable (?f x) by auto
  next
    show summable (?f' x0)
      using converges[OF ‹x0 ∈ {−R <..< R}›] .
    show x0 ∈ {−R' <..< R'}
      using ‹x0 ∈ {−R' <..< R'}› .
  qed
 qed
 let ?R = (R + |x0|) / 2
 have |x0| < ?R
  using assms by (auto simp: field-simps)
 then have − ?R < x0
 proof (cases x0 < 0)
  case True
  then have − x0 < ?R
   using ‹|x0| < ?R› by auto
  then show ?thesis
   unfolding neg-less-iff-less[symmetric, of − x0] by auto
 next
  case False
  have − ?R < 0 using assms by auto
  also have ... ≤ x0 using False by auto
  finally show ?thesis .
 qed
 then have 0 < ?R ?R < R − ?R < x0 and x0 < ?R
  using assms by (auto simp: field-simps)
 from for-subinterval[OF this] show ?thesis .
qed

lemma geometric-deriv-sums:
 fixes z :: 'a :: {real-normed-field,banach}
 assumes norm z < 1
 shows  (λn. of-nat (Suc n) * z ^ n) sums (1 / (1 − z)^2)
proof −
 have (λn. diffs (λn. 1) n * z^n) sums (1 / (1 − z)^2)
 proof (rule termdiffs-sums-strong)
  fix z :: 'a assume norm z < 1
  thus (λn. 1 * z^n) sums (1 / (1 − z)) by (simp add: geometric-sums)
 qed (insert assms, auto intro!: derivative-eq-intros simp: power2-eq-square)
 thus ?thesis unfolding diffs-def by simp
```

qed

lemma *isCont-pochhammer* [*continuous-intros*]: *isCont* (λz. *pochhammer z n*) *z*
for *z* :: *'a*::*real-normed-field*
by (*induct n*) (*auto simp*: *pochhammer-rec'*)

lemma *continuous-on-pochhammer* [*continuous-intros*]: *continuous-on A* (λz. *pochhammer z n*)
for *A* :: *'a*::*real-normed-field set*
by (*intro continuous-at-imp-continuous-on ballI isCont-pochhammer*)

lemmas *continuous-on-pochhammer'* [*continuous-intros*] =
continuous-on-compose2[*OF continuous-on-pochhammer - subset-UNIV*]

112.6 Exponential Function

definition *exp* :: *'a* $\Rightarrow$ *'a*::{*real-normed-algebra-1*,*banach*}
where *exp* = (λx. $\sum n$. *x*^*n* /$_R$ *fact n*)

lemma *summable-exp-generic*:
fixes *x* :: *'a*::{*real-normed-algebra-1*,*banach*}
defines *S-def*: *S* $\equiv$ λn. *x*^*n* /$_R$ *fact n*
shows *summable S*
proof −
have *S-Suc*: $\bigwedge n$. *S* (*Suc n*) = (*x* * *S n*) /$_R$ (*Suc n*)
unfolding *S-def* **by** (*simp del*: *mult-Suc*)
obtain *r* :: *real* **where** *r0*: *0* < *r* **and** *r1*: *r* < *1*
using *dense* [*OF zero-less-one*] **by** *fast*
obtain *N* :: *nat* **where** *N*: *norm x* < *real N* * *r*
using *ex-less-of-nat-mult r0* **by** *auto*
from *r1* **show** *?thesis*
proof (*rule summable-ratio-test* [*rule-format*])
fix *n* :: *nat*
assume *n*: *N* ≤ *n*
have *norm x* ≤ *real N* * *r*
using *N* **by** (*rule order-less-imp-le*)
also have *real N* * *r* ≤ *real* (*Suc n*) * *r*
using *r0 n* **by** (*simp add*: *mult-right-mono*)
finally have *norm x* * *norm* (*S n*) ≤ *real* (*Suc n*) * *r* * *norm* (*S n*)
using *norm-ge-zero* **by** (*rule mult-right-mono*)
then have *norm* (*x* * *S n*) ≤ *real* (*Suc n*) * *r* * *norm* (*S n*)
by (*rule order-trans* [*OF norm-mult-ineq*])
then have *norm* (*x* * *S n*) / *real* (*Suc n*) ≤ *r* * *norm* (*S n*)
by (*simp add*: *pos-divide-le-eq ac-simps*)
then show *norm* (*S* (*Suc n*)) ≤ *r* * *norm* (*S n*)
by (*simp add*: *S-Suc inverse-eq-divide*)
qed
qed

lemma *summable-norm-exp*: *summable* (λ*n*. *norm* (*x*^*n* /$_R$ *fact n*))
for *x* :: *'a*::{*real-normed-algebra-1*,*banach*}
proof (*rule summable-norm-comparison-test* [*OF exI*, *rule-format*])
show *summable* (λ*n*. *norm x*^*n* /$_R$ *fact n*)
by (*rule summable-exp-generic*)
show *norm* (*x*^*n* /$_R$ *fact n*) ≤ *norm x*^*n* /$_R$ *fact n* **for** *n*
by (*simp add*: *norm-power-ineq*)
qed

lemma *summable-exp*: *summable* (λ*n*. *inverse* (*fact n*) ∗ *x*^*n*)
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *summable-exp-generic* [**where** *x*=*x*]
by (*simp add*: *scaleR-conv-of-real nonzero-of-real-inverse*)

lemma *exp-converges*: (λ*n*. *x*^*n* /$_R$ *fact n*) *sums exp x*
unfolding *exp-def* **by** (*rule summable-exp-generic* [*THEN summable-sums*])

lemma *exp-fdiffs*:
diffs (λ*n*. *inverse* (*fact n*)) = (λ*n*. *inverse* (*fact n* :: *'a*::{*real-normed-field*,*banach*}))
by (*simp add*: *diffs-def mult-ac nonzero-inverse-mult-distrib nonzero-of-real-inverse*
del: *mult-Suc of-nat-Suc*)

lemma *diffs-of-real*: *diffs* (λ*n*. *of-real* (*f n*)) = (λ*n*. *of-real* (*diffs f n*))
by (*simp add*: *diffs-def*)

lemma *DERIV-exp* [*simp*]: *DERIV exp x* :> *exp x*
unfolding *exp-def scaleR-conv-of-real*
proof (*rule DERIV-cong*)
have *sinv*: *summable* (λ*n*. *of-real* (*inverse* (*fact n*)) ∗ *x* ^ *n*) **for** *x*::*'a*
by (*rule exp-converges* [*THEN sums-summable*, *unfolded scaleR-conv-of-real*])
note *xx* = *exp-converges* [*THEN sums-summable*, *unfolded scaleR-conv-of-real*]
show ((λ*x*. ∑ *n*. *of-real* (*inverse* (*fact n*)) ∗ *x* ^ *n*) *has-field-derivative*
(∑ *n*. *diffs* (λ*n*. *of-real* (*inverse* (*fact n*))) *n* ∗ *x* ^ *n*)) (*at x*)
by (*rule termdiffs* [**where** *K*=*of-real* (*1* + *norm x*)]) (*simp-all only*: *diffs-of-real exp-fdiffs sinv norm-of-real*)
show (∑ *n*. *diffs* (λ*n*. *of-real* (*inverse* (*fact n*))) *n* ∗ *x* ^ *n*) = (∑ *n*. *of-real* (*inverse* (*fact n*)) ∗ *x* ^ *n*)
by (*simp add*: *diffs-of-real exp-fdiffs*)
qed

declare *DERIV-exp*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-exp*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemmas *has-derivative-exp*[*derivative-intros*] = *DERIV-exp*[*THEN DERIV-compose-FDERIV*]

lemma *norm-exp*: *norm* (*exp x*) ≤ *exp* (*norm x*)
proof −
from *summable-norm*[*OF summable-norm-exp*, *of x*]

have *norm* (*exp x*) ≤ (∑ *n*. *inverse* (*fact n*) ∗ *norm* (*x*^*n*))
by (*simp add*: *exp-def*)
also have ... ≤ *exp* (*norm x*)
using *summable-exp-generic*[*of norm x*] *summable-norm-exp*[*of x*]
by (*auto simp*: *exp-def intro*!: *suminf-le norm-power-ineq*)
finally show *?thesis* **.**
qed

lemma *isCont-exp*: *isCont exp x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
by (*rule DERIV-exp* [*THEN DERIV-isCont*])

lemma *isCont-exp'* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *exp* (*f x*)) *a*
for *f* :: - ⇒ *'a*::{*real-normed-field*,*banach*}
by (*rule isCont-o2* [*OF* - *isCont-exp*])

lemma *tendsto-exp* [*tendsto-intros*]: (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *exp* (*f x*)) ⟶ *exp a*) *F*
for *f*:: - ⇒ *'a*::{*real-normed-field*,*banach*}
by (*rule isCont-tendsto-compose* [*OF isCont-exp*])

lemma *continuous-exp* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *exp* (*f x*))
for *f* :: - ⇒ *'a*::{*real-normed-field*,*banach*}
unfolding *continuous-def* **by** (*rule tendsto-exp*)

lemma *continuous-on-exp* [*continuous-intros*]: *continuous-on s f* ⟹ *continuous-on s* (λ*x*. *exp* (*f x*))
for *f* :: - ⇒ *'a*::{*real-normed-field*,*banach*}
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-exp*)

112.6.1 Properties of the Exponential Function

lemma *exp-zero* [*simp*]: *exp 0* = *1*
unfolding *exp-def* **by** (*simp add*: *scaleR-conv-of-real*)

lemma *exp-series-add-commuting*:
fixes *x y* :: *'a*::{*real-normed-algebra-1*,*banach*}
defines *S-def*: *S* ≡ λ*x n*. *x*^*n* /$_R$ *fact n*
assumes *comm*: *x* ∗ *y* = *y* ∗ *x*
shows *S* (*x* + *y*) *n* = (∑ *i*≤*n*. *S x i* ∗ *S y* (*n* − *i*))
proof (*induct n*)
case *0*
show *?case*
unfolding *S-def* **by** *simp*
next
case (*Suc n*)
have *S-Suc*: ⋀*x n*. *S x* (*Suc n*) = (*x* ∗ *S x n*) /$_R$ *real* (*Suc n*)
unfolding *S-def* **by** (*simp del*: *mult-Suc*)

then have *times-S*: $\bigwedge x\ n.\ x * S\ x\ n = real\ (Suc\ n) *_R S\ x\ (Suc\ n)$
by *simp*
have *S-comm*: $\bigwedge n.\ S\ x\ n * y = y * S\ x\ n$
by (*simp add*: *power-commuting-commutes comm S-def*)

have $real\ (Suc\ n) *_R S\ (x + y)\ (Suc\ n) = (x + y) * (\sum i{\le}n.\ S\ x\ i * S\ y\ (n - i))$
by (*metis Suc.hyps times-S*)
also have $\ldots = x * (\sum i{\le}n.\ S\ x\ i * S\ y\ (n - i)) + y * (\sum i{\le}n.\ S\ x\ i * S\ y\ (n - i))$
by (*rule distrib-right*)
also have $\ldots = (\sum i{\le}n.\ x * S\ x\ i * S\ y\ (n - i)) + (\sum i{\le}n.\ S\ x\ i * y * S\ y\ (n - i))$
by (*simp add*: *sum-distrib-left ac-simps S-comm*)
also have $\ldots = (\sum i{\le}n.\ x * S\ x\ i * S\ y\ (n - i)) + (\sum i{\le}n.\ S\ x\ i * (y * S\ y\ (n - i)))$
by (*simp add*: *ac-simps*)
also have $\ldots = (\sum i{\le}n.\ real\ (Suc\ i) *_R (S\ x\ (Suc\ i) * S\ y\ (n - i)))$
$+ (\sum i{\le}n.\ real\ (Suc\ n - i) *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
by (*simp add*: *times-S Suc-diff-le*)
also have $(\sum i{\le}n.\ real\ (Suc\ i) *_R (S\ x\ (Suc\ i) * S\ y\ (n - i)))$
$= (\sum i{\le}Suc\ n.\ real\ i *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
by (*subst sum.atMost-Suc-shift*) *simp*
also have $(\sum i{\le}n.\ real\ (Suc\ n - i) *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
$= (\sum i{\le}Suc\ n.\ real\ (Suc\ n - i) *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
by *simp*
also have $(\sum i{\le}Suc\ n.\ real\ i *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
$+ (\sum i{\le}Suc\ n.\ real\ (Suc\ n - i) *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
$= (\sum i{\le}Suc\ n.\ real\ (Suc\ n) *_R (S\ x\ i * S\ y\ (Suc\ n - i)))$
by (*simp flip*: *sum.distrib scaleR-add-left of-nat-add*)
also have $\ldots = real\ (Suc\ n) *_R (\sum i{\le}Suc\ n.\ S\ x\ i * S\ y\ (Suc\ n - i))$
by (*simp only*: *scaleR-right.sum*)
finally show $S\ (x + y)\ (Suc\ n) = (\sum i{\le}Suc\ n.\ S\ x\ i * S\ y\ (Suc\ n - i))$
by (*simp del*: *sum.cl-ivl-Suc*)
qed

lemma *exp-add-commuting*: $x * y = y * x \Longrightarrow exp\ (x + y) = exp\ x * exp\ y$
by (*simp only*: *exp-def Cauchy-product summable-norm-exp exp-series-add-commuting*)

lemma *exp-times-arg-commute*: $exp\ A * A = A * exp\ A$
by (*simp add*: *exp-def suminf-mult*[*symmetric*] *summable-exp-generic power-commutes suminf-mult2*)

lemma *exp-add*: $exp\ (x + y) = exp\ x * exp\ y$
for $x\ y :: 'a{::}\{real\text{-}normed\text{-}field, banach\}$
by (*rule exp-add-commuting*) (*simp add*: *ac-simps*)

lemma *exp-double*: $exp(2 * z) = exp\ z\ \widehat{}\ 2$
by (*simp add*: *exp-add-commuting mult-2 power2-eq-square*)

lemmas *mult-exp-exp* = *exp-add* [*symmetric*]

lemma *exp-of-real*: *exp* (*of-real x*) = *of-real* (*exp x*)
unfolding *exp-def*
apply (*subst suminf-of-real* [*OF summable-exp-generic*])
apply (*simp add*: *scaleR-conv-of-real*)
done

lemmas *of-real-exp* = *exp-of-real*[*symmetric*]

corollary *exp-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow$ *exp z* $\in \mathbb{R}$
by (*metis Reals-cases Reals-of-real exp-of-real*)

lemma *exp-not-eq-zero* [*simp*]: *exp x* $\neq$ *0*
proof
have *exp x* $*$ *exp* ($-$ *x*) = *1*
by (*simp add*: *exp-add-commuting*[*symmetric*])
also assume *exp x* = *0*
finally show *False* **by** *simp*
qed

lemma *has-log-derivative-exp* [*derivative-intros*]:
assumes ((λ*x*. *f x*) *has-field-derivative f* ′) (*at x within A*)
shows ((λ*x*. *exp* (*f x*)) *has-log-derivative f* ′) (*at x within A*)
using *assms*
by (*cases at x within A* = *bot*)
(*auto simp*: *has-log-derivative-def Lim-ident-at intro*!: *derivative-eq-intros*)

lemma *exp-minus-inverse*: *exp x* $*$ *exp* ($-$ *x*) = *1*
by (*simp add*: *exp-add-commuting*[*symmetric*])

lemma *exp-minus*: *exp* ($-$ *x*) = *inverse* (*exp x*)
for *x* :: ′*a*::{*real-normed-field*,*banach*}
by (*intro inverse-unique* [*symmetric*] *exp-minus-inverse*)

lemma *exp-diff*: *exp* (*x* $-$ *y*) = *exp x* / *exp y*
for *x* :: ′*a*::{*real-normed-field*,*banach*}
using *exp-add* [*of x* $-$ *y*] **by** (*simp add*: *exp-minus divide-inverse*)

lemma *exp-of-nat-mult*: *exp* (*of-nat n* $*$ *x*) = *exp x* ^ *n*
for *x* :: ′*a*::{*real-normed-field*,*banach*}
by (*induct n*) (*auto simp*: *distrib-left exp-add mult.commute*)

corollary *exp-of-nat2-mult*: *exp* (*x* $*$ *of-nat n*) = *exp x* ^ *n*
for *x* :: ′*a*::{*real-normed-field*,*banach*}
by (*metis exp-of-nat-mult mult-of-nat-commute*)

lemma *exp-sum*: *finite I* $\Longrightarrow$ *exp* (*sum f I*) = *prod* (λ*x*. *exp* (*f x*)) *I*

by (*induct I rule*: *finite-induct*) (*auto simp*: *exp-add-commuting mult.commute*)

lemma *exp-divide-power-eq*:
fixes $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
assumes $n > 0$
shows $exp\ (x\ /\ of\text{-}nat\ n)\ \hat{}\ n = exp\ x$
using *assms*
proof (*induction n arbitrary*: *x*)
case (*Suc n*)
show *?case*
proof (*cases n = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have [*simp*]: $1 + (of\text{-}nat\ n * of\text{-}nat\ n + of\text{-}nat\ n * 2) \neq (0::'a)$
using *of-nat-eq-iff* [*of 1 + n * n + n * 2 0*]
by *simp*
from *False* **have** [*simp*]: $x * of\text{-}nat\ n\ /\ (1 + of\text{-}nat\ n)\ /\ of\text{-}nat\ n = x\ /\ (1 + of\text{-}nat\ n)$
by *simp*
have [*simp*]: $x\ /\ (1 + of\text{-}nat\ n) + x * of\text{-}nat\ n\ /\ (1 + of\text{-}nat\ n) = x$
using *of-nat-neq-0*
by (*auto simp add*: *field-split-simps*)
show *?thesis*
using *Suc.IH* [*of x * of-nat n / (1 + of-nat n)*] *False*
by (*simp add*: *exp-add* [*symmetric*])
qed
qed *simp*

lemma *exp-power-int*:
fixes $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $exp\ x\ powi\ n = exp\ (of\text{-}int\ n * x)$
proof (*cases* $n \geq 0$)
case *True*
have $exp\ x\ powi\ n = exp\ x\ \hat{}\ nat\ n$
using *True* **by** (*simp add*: *power-int-def*)
thus *?thesis*
using *True* **by** (*subst* (*asm*) *exp-of-nat-mult* [*symmetric*]) *auto*
next
case *False*
have $exp\ x\ powi\ n = inverse\ (exp\ x\ \hat{}\ nat\ (-n))$
using *False* **by** (*simp add*: *power-int-def field-simps*)
also have $exp\ x\ \hat{}\ nat\ (-n) = exp\ (of\text{-}nat\ (nat\ (-n)) * x)$
using *False* **by** (*subst exp-of-nat-mult*) *auto*
also have $inverse\ \ldots = exp\ (-(of\text{-}nat\ (nat\ (-n)) * x))$
by (*subst exp-minus*) (*auto simp*: *field-simps*)
also have $-(of\text{-}nat\ (nat\ (-n)) * x) = of\text{-}int\ n * x$
using *False* **by** *simp*

finally show *?thesis* **.**
qed

lemma
fixes $z :: 'a :: \{banach, real\text{-}normed\text{-}field\}$
assumes $norm\ z \leq 1\ /\ 2$
shows *norm-exp-bounds-lemma*: $norm\ (exp\ z - 1 - z) \leq norm\ z\ /\ 2$
and *norm-exp-bounds*: $norm\ (exp\ z - 1) \geq 1\ /\ 2 * norm\ z$
$norm\ (exp\ z - 1) \leq 3\ /\ 2 * norm\ z$
proof −
show ∗: $norm\ (exp\ z - 1 - z) \leq norm\ z\ /\ 2$
proof (*cases* $z = 0$)
case *False*
have *sums*: $(\lambda k.\ z\ \hat{}\ (k + 2)\ /_R\ fact\ (k + 2))\ sums\ (exp\ z - (\sum k{<}2.\ z\ \hat{}\ k\ /_R\ fact\ k))$
by (*intro sums-split-initial-segment exp-converges*)

have *summable* $(\lambda k.\ norm\ z\ \hat{}\ (k + 2)\ /_R\ fact\ (k + 2))$
using *summable-norm-exp*[*of z*]
by (*intro summable-norm summable-ignore-initial-segment*)
(*auto simp*: *norm-power norm-divide divide-simps*)
also have *?this* $\longleftrightarrow$ *summable* $(\lambda k.\ norm\ z\ \hat{}\ 2 * (norm\ z\ \hat{}\ k\ /\ fact\ (k + 2)))$
by (*simp add*: *power-add mult-ac divide-simps power2-eq-square del*: *of-nat-Suc of-nat-add*)
also have $\ldots \longleftrightarrow$ *summable* $(\lambda k.\ norm\ z\ \hat{}\ k\ /\ fact\ (k + 2))$
by (*subst summable-cmult-iff*) (*use* ‹$z \neq 0$› **in** *auto*)
finally have *summable*: *summable* $(\lambda k.\ norm\ z\ \hat{}\ k\ /\ fact\ (k + 2))$ **.**

have $exp\ z - 1 - z = (\sum k.\ z\ \hat{}\ (k + 2)\ /\ fact\ (k + 2))$
using *sums* **by** (*simp add*: *sums-iff scaleR-conv-of-real divide-simps eval-nat-numeral*)
also have $norm\ \ldots \leq (\sum k.\ norm\ (z\ \hat{}\ (k + 2)\ /\ fact\ (k + 2)))$
using *summable-norm-exp*[*of z*]
by (*intro summable-norm summable-ignore-initial-segment*)
(*auto simp*: *norm-power norm-divide divide-simps*)
also have $\ldots = (\sum k.\ norm\ z\ \hat{}\ 2 * (norm\ z\ \hat{}\ k\ /\ fact\ (k + 2)))$
by (*simp add*: *power-add norm-power norm-divide mult-ac norm-mult power2-eq-square del*: *of-nat-Suc*)
also have $\ldots = norm\ z\ \hat{}\ 2 * (\sum k.\ norm\ z\ \hat{}\ k\ /\ fact\ (k + 2))$
using *summable* **by** (*rule suminf-mult*)
also have $\ldots \leq norm\ z\ \hat{}\ 2 * (1\ /\ (1 - norm\ z)\ /\ 2)$
proof (*intro mult-left-mono*, *rule sums-le*)
show $(\lambda k.\ norm\ z\ \hat{}\ k\ /\ fact\ (k + 2))\ sums\ (\sum k.\ norm\ z\ \hat{}\ k\ /\ fact\ (k + 2))$
using *summable* **by** *blast*
show $(\lambda k.\ norm\ z\ \hat{}\ k\ /\ 2)\ sums\ (1\ /\ (1 - norm\ z)\ /\ 2)$
using *assms* **by** (*intro geometric-sums sums-divide*) *auto*
next
fix $k :: nat$
have $norm\ z\ \hat{}\ k\ /\ fact\ (k + 2) \leq norm\ z\ \hat{}\ k\ /\ fact\ 2$
by (*intro divide-left-mono fact-mono*) *auto*

```
    thus norm z ^ k / fact (k + 2) ≤ norm z ^ k / 2
      by simp
  qed (auto simp: divide-simps)
  also have ... = norm z * (norm z / (1 - norm z)) / 2
    by (simp add: power2-eq-square)
  also have ... ≤ norm z * ((1 / 2) / (1 - (1 / 2))) / 2
    using assms by (intro mult-mono frac-le diff-mono) auto
  also have ... = norm z / 2
    by simp
  finally show norm (exp z - 1 - z) ≤ norm z / 2 .
qed auto

have norm (exp z - 1) ≤ norm z + norm (exp z - 1 - z)
  by (rule norm-triangle-sub)
with * show norm (exp z - 1) ≤ 3 / 2 * norm z
  by simp

have norm z - norm (exp z - 1 - z) ≤ norm (exp z - 1)
  using norm-triangle-ineq3[of exp z - 1 - z -z] by simp
with * show norm (exp z - 1) ≥ 1 / 2 * norm z
  by simp
qed
```

112.6.2 Properties of the Exponential Function on Reals

Comparisons of *exp x* with zero.

Proof: because every exponential can be seen as a square.

```
lemma exp-ge-zero [simp]: 0 ≤ exp x
  for x :: real
proof -
  have 0 ≤ exp (x/2) * exp (x/2)
    by simp
  then show ?thesis
    by (simp add: exp-add [symmetric])
qed

lemma exp-gt-zero [simp]: 0 < exp x
  for x :: real
  by (simp add: order-less-le)

lemma not-exp-less-zero [simp]: ¬ exp x < 0
  for x :: real
  by (simp add: not-less)

lemma not-exp-le-zero [simp]: ¬ exp x ≤ 0
  for x :: real
  by (simp add: not-le)
```

lemma *abs-exp-cancel* [*simp*]: $|exp\ x| = exp\ x$
for x :: *real*
by *simp*

Strict monotonicity of exponential.

lemma *exp-ge-add-one-self-aux*:
fixes x :: *real*
assumes $0 \leq x$
shows $1 + x \leq exp\ x$
using *order-le-imp-less-or-eq* [*OF assms*]
proof
assume $0 < x$
have $1 + x \leq (\sum n{<}2.\ inverse\ (fact\ n) * x\hat{}n)$
by (*auto simp*: *numeral-2-eq-2*)
also have $\ldots \leq (\sum n.\ inverse\ (fact\ n) * x\hat{}n)$
using ‹$0 < x$› **by** (*auto simp add*: *zero-le-mult-iff intro*: *sum-le-suminf* [*OF summable-exp*])
finally show $1 + x \leq exp\ x$
by (*simp add*: *exp-def*)
qed *auto*

lemma *exp-gt-one*: $0 < x \Longrightarrow 1 < exp\ x$
for x :: *real*
proof −
assume x: $0 < x$
then have $1 < 1 + x$ **by** *simp*
also from x **have** $1 + x \leq exp\ x$
by (*simp add*: *exp-ge-add-one-self-aux*)
finally show *?thesis* **.**
qed

lemma *exp-less-mono*:
fixes $x\ y$ :: *real*
assumes $x < y$
shows $exp\ x < exp\ y$
proof −
from ‹$x < y$› **have** $0 < y - x$ **by** *simp*
then have $1 < exp\ (y - x)$ **by** (*rule exp-gt-one*)
then have $1 < exp\ y\ /\ exp\ x$ **by** (*simp only*: *exp-diff*)
then show $exp\ x < exp\ y$ **by** *simp*
qed

lemma *exp-less-cancel*: $exp\ x < exp\ y \Longrightarrow x < y$
for $x\ y$ :: *real*
unfolding *linorder-not-le* [*symmetric*]
by (*auto simp*: *order-le-less exp-less-mono*)

lemma *exp-less-cancel-iff* [*iff*]: $exp\ x < exp\ y \longleftrightarrow x < y$
for $x\ y$ :: *real*

by (*auto intro*: *exp-less-mono exp-less-cancel*)

lemma *exp-le-cancel-iff* [*iff*]: $exp\ x \leq exp\ y \longleftrightarrow x \leq y$
for $x\ y$ :: *real*
by (*auto simp*: *linorder-not-less* [*symmetric*])

lemma *exp-mono*:
fixes $x\ y$ :: *real*
assumes $x \leq y$
shows $exp\ x \leq exp\ y$
using *assms exp-le-cancel-iff* **by** *fastforce*

lemma *exp-minus'*: $exp\ (-x) = 1\ /\ (exp\ x)$
for x :: $'a$::{*real-normed-field*,*banach*}
by (*simp add*: *exp-minus inverse-eq-divide*)

lemma *exp-inj-iff* [*iff*]: $exp\ x = exp\ y \longleftrightarrow x = y$
for $x\ y$ :: *real*
by (*simp add*: *order-eq-iff*)

Comparisons of $exp\ x$ with one.

lemma *one-less-exp-iff* [*simp*]: $1 < exp\ x \longleftrightarrow 0 < x$
for x :: *real*
using *exp-less-cancel-iff* [**where** $x = 0$ **and** $y = x$] **by** *simp*

lemma *exp-less-one-iff* [*simp*]: $exp\ x < 1 \longleftrightarrow x < 0$
for x :: *real*
using *exp-less-cancel-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *one-le-exp-iff* [*simp*]: $1 \leq exp\ x \longleftrightarrow 0 \leq x$
for x :: *real*
using *exp-le-cancel-iff* [**where** $x = 0$ **and** $y = x$] **by** *simp*

lemma *exp-le-one-iff* [*simp*]: $exp\ x \leq 1 \longleftrightarrow x \leq 0$
for x :: *real*
using *exp-le-cancel-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *exp-eq-one-iff* [*simp*]: $exp\ x = 1 \longleftrightarrow x = 0$
for x :: *real*
using *exp-inj-iff* [**where** $x = x$ **and** $y = 0$] **by** *simp*

lemma *lemma-exp-total*: $1 \leq y \Longrightarrow \exists x.\ 0 \leq x \wedge x \leq y - 1 \wedge exp\ x = y$
for y :: *real*
proof (*rule IVT*)
assume $1 \leq y$
then have $0 \leq y - 1$ **by** *simp*
then have $1 + (y - 1) \leq exp\ (y - 1)$
by (*rule exp-ge-add-one-self-aux*)
then show $y \leq exp\ (y - 1)$ **by** *simp*

qed (*simp-all add*: *le-diff-eq*)

lemma *exp-total*: $0 < y \Longrightarrow \exists\, x.\ exp\ x = y$
  for y :: *real*
proof (*rule linorder-le-cases* [*of 1 y*])
  assume $1 \leq y$
  then show $\exists\, x.\ exp\ x = y$
    by (*fast dest*: *lemma-exp-total*)
next
  assume $0 < y$ **and** $y \leq 1$
  then have $1 \leq inverse\ y$
    by (*simp add*: *one-le-inverse-iff*)
  then obtain x **where** $exp\ x = inverse\ y$
    by (*fast dest*: *lemma-exp-total*)
  then have $exp\ (-\ x) = y$
    by (*simp add*: *exp-minus*)
  then show $\exists\, x.\ exp\ x = y$ **..**
qed

112.7 Natural Logarithm

class *ln* = *real-normed-algebra-1* + *banach* +
  fixes *ln* :: $'a \Rightarrow 'a$
  assumes *ln-one* [*simp*]: *ln 1* = *0*

definition *powr* :: $'a \Rightarrow 'a \Rightarrow 'a$::*ln* (**infixr** ‹*powr*› *80*)
  — exponentation via ln and exp
  where *x powr a* ≡ *if x* = *0 then 0 else exp* (*a* ∗ *ln x*)

lemma *powr-0* [*simp*]: *0 powr z* = *0*
  by (*simp add*: *powr-def*)

We totalise *ln* over all reals exactly as done in Mathlib

instantiation *real* :: *ln*
begin

definition *raw-ln-real* :: *real* ⇒ *real*
  where *raw-ln-real x* ≡ (*THE u. exp u* = *x*)

definition *ln-real* :: *real* ⇒ *real*
  where *ln-real* ≡ λ*x. if x*=*0 then 0 else raw-ln-real* |*x*|

instance
  by *intro-classes* (*simp add*: *ln-real-def raw-ln-real-def*)

end

lemma *powr-eq-0-iff* [*simp*]: *w powr z* = *0* ⟷ *w* = *0*
  by (*simp add*: *powr-def*)

lemma *raw-ln-exp* [*simp*]: *raw-ln-real* (*exp x*) = *x*
by (*simp add*: *raw-ln-real-def*)

lemma *exp-raw-ln* [*simp*]: *0* < *x* ⟹ *exp* (*raw-ln-real x*) = *x*
by (*auto dest*: *exp-total*)

lemma *raw-ln-unique*: *exp y* = *x* ⟹ *raw-ln-real x* = *y*
by *auto*

lemma *abs-raw-ln*: *x* ≠ *0* ⟹ *raw-ln-real*|*x*| = *ln x*
by (*simp add*: *ln-real-def*)

lemma *ln-0* [*simp*]: *ln* (*0*::*real*) = *0*
by (*simp add*: *ln-real-def*)

lemma *ln-minus*: *ln* (−*x*) = *ln x*
for *x* :: *real*
by (*simp add*: *ln-real-def*)

lemma *ln-exp* [*simp*]: *ln* (*exp x*) = *x*
for *x* :: *real*
by (*simp add*: *ln-real-def*)

lemma *exp-ln-abs*:
fixes *x*::*real*
shows *x* ≠ *0* ⟹ *exp* (*ln x*) = |*x*|
by (*simp add*: *ln-real-def*)

lemma *exp-ln* [*simp*]: *0* < *x* ⟹ *exp* (*ln x*) = *x*
for *x* :: *real*
using *exp-ln-abs* **by** *fastforce*

lemma *exp-ln-iff* [*simp*]: *exp* (*ln x*) = *x* ⟷ *0* < *x*
for *x* :: *real*
by (*metis exp-gt-zero exp-ln*)

lemma *ln-unique*: *exp y* = *x* ⟹ *ln x* = *y*
for *x* :: *real*
by *auto*

lemma *ln-unique'*: *exp y* = |*x*| ⟹ *ln x* = *y*
for *x* :: *real*
by (*metis abs-raw-ln abs-zero exp-not-eq-zero raw-ln-exp*)

lemma *raw-ln-mult*: *x*>*0* ⟹ *y*>*0* ⟹ *raw-ln-real* (*x* ∗ *y*) = *raw-ln-real x* + *raw-ln-real y*
by (*metis exp-add exp-ln raw-ln-exp*)

lemma *ln-mult*: *ln* $(x * y) = (if\ x{\neq}0 \wedge y{\neq}0\ then\ ln\ x + ln\ y\ else\ 0)$
for x :: *real*
by (*simp add*: *ln-real-def abs-mult raw-ln-mult*)

lemma *ln-mult-pos*: $x{>}0 \Longrightarrow y{>}0 \Longrightarrow ln\ (x * y) = ln\ x + ln\ y$
for x :: *real*
by (*simp add*: *ln-mult*)

lemma *ln-prod*: *finite* $I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow f\ i \neq 0) \Longrightarrow ln\ (prod\ f\ I) = sum\ (\lambda x.\ ln(f\ x))\ I$
for f :: $'a \Rightarrow real$
by (*induct I rule*: *finite-induct*) (*auto simp*: *ln-mult prod-pos*)

lemma *ln-inverse*: *ln* (*inverse* x) $= -\ ln\ x$
for x :: *real*
by (*smt* (*verit*) *inverse-nonzero-iff-nonzero ln-mult ln-one ln-real-def right-inverse*)

lemma *ln-div*: *ln* $(x/y) = (if\ x{\neq}0 \wedge y{\neq}0\ then\ ln\ x - ln\ y\ else\ 0)$
for x :: *real*
by (*simp add*: *divide-inverse ln-inverse ln-mult*)

lemma *ln-divide-pos*: $x{>}0 \Longrightarrow y{>}0 \Longrightarrow ln\ (x/y) = ln\ x - ln\ y$
for x :: *real*
by (*simp add*: *divide-inverse ln-inverse ln-mult*)

lemma *ln-realpow*: *ln* $(x\hat{}n) = real\ n * ln\ x$
proof (*cases* $x{=}0$)
case *True*
then show *?thesis* **by** (*auto simp*: *power-0-left*)
next
case *False*
then show *?thesis*
by (*induction n*) (*auto simp*: *ln-mult distrib-right*)
qed

lemma *ln-less-cancel-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x < ln\ y \longleftrightarrow x < y$
for x :: *real*
by (*subst exp-less-cancel-iff* [*symmetric*]) *simp*

lemma *ln-le-cancel-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x \leq ln\ y \longleftrightarrow x \leq y$
for x :: *real*
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *ln-mono*: $\bigwedge x{::}real.\ [\![x \leq y;\ 0 < x]\!] \Longrightarrow ln\ x \leq ln\ y$
by *simp*

lemma *ln-strict-mono*: $\bigwedge x{::}real.\ [\![x < y;\ 0 < x]\!] \Longrightarrow ln\ x < ln\ y$
by *simp*

lemma *ln-inj-iff* [*simp*]: $0 < x \Longrightarrow 0 < y \Longrightarrow ln\ x = ln\ y \longleftrightarrow x = y$
for x :: *real*
by (*simp add*: *order-eq-iff*)

lemma *ln-add-one-self-le-self*: $0 \leq x \Longrightarrow ln\ (1 + x) \leq x$
for x :: *real*
by (*rule exp-le-cancel-iff* [*THEN iffD1*]) (*simp add*: *exp-ge-add-one-self-aux*)

lemma *ln-less-self* [*simp*]: $0 < x \Longrightarrow ln\ x < x$
for x :: *real*
by (*rule order-less-le-trans* [**where** $y = ln\ (1 + x)$]) (*simp-all add*: *ln-add-one-self-le-self*)

lemma *ln-ge-iff*: $\bigwedge x$::*real*. $0 < x \Longrightarrow y \leq ln\ x \longleftrightarrow exp\ y \leq x$
using *exp-le-cancel-iff exp-total* **by** *force*

lemma *ln-ge-zero* [*simp*]: $1 \leq x \Longrightarrow 0 \leq ln\ x$
for x :: *real*
using *ln-le-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-ge-zero-imp-ge-one*: $0 \leq ln\ x \Longrightarrow 0 < x \Longrightarrow 1 \leq x$
for x :: *real*
using *ln-le-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-ge-zero-iff* [*simp*]: $0 < x \Longrightarrow 0 \leq ln\ x \longleftrightarrow 1 \leq x$
for x :: *real*
using *ln-le-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-less-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x < 0 \longleftrightarrow x < 1$
for x :: *real*
using *ln-less-cancel-iff* [*of x 1*] **by** *simp*

lemma *ln-le-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x \leq 0 \longleftrightarrow x \leq 1$
for x :: *real*
by (*metis less-numeral-extra*(*1*) *ln-le-cancel-iff ln-one*)

lemma *ln-gt-zero*: $1 < x \Longrightarrow 0 < ln\ x$
for x :: *real*
using *ln-less-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-gt-zero-imp-gt-one*: $0 < ln\ x \Longrightarrow 0 < x \Longrightarrow 1 < x$
for x :: *real*
using *ln-less-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-gt-zero-iff* [*simp*]: $0 < x \Longrightarrow 0 < ln\ x \longleftrightarrow 1 < x$
for x :: *real*
using *ln-less-cancel-iff* [*of 1 x*] **by** *simp*

lemma *ln-eq-zero-iff* [*simp*]: $0 < x \Longrightarrow ln\ x = 0 \longleftrightarrow x = 1$
for x :: *real*

using *ln-inj-iff* [*of x 1*] **by** *simp*

lemma *ln-less-zero*: $0 < x \Longrightarrow x < 1 \Longrightarrow ln\ x < 0$
for x :: *real*
by *simp*

lemma *powr-eq-one-iff* [*simp*]:
$a\ powr\ x = 1 \longleftrightarrow x = 0$ **if** $a > 1$ **for** $a\ x$ :: *real*
using *that* **by** (*auto simp*: *powr-def split*: *if-splits*)

A consequence of our "totalising" of ln

lemma *uminus-powr-eq*: $(-a)\ powr\ x = a\ powr\ x$ **for** x::*real*
by (*simp add*: *powr-def ln-minus*)

lemma *isCont-ln-pos*:
fixes x :: *real*
assumes $x > 0$
shows *isCont ln x*
by (*metis assms exp-ln isCont-exp isCont-inverse-function ln-exp*)

lemma *isCont-ln*:
fixes x :: *real*
assumes $x \neq 0$
shows *isCont ln x*
proof (*cases* $0 < x$)
case *False*
then have *isCont* (*ln o uminus*) *x*
using *isCont-minus* [*OF continuous-ident*] *assms continuous-at-compose isCont-ln-pos*
by *force*
then show *?thesis*
by (*simp add*: *comp-def ln-minus*)
qed (*simp add*: *isCont-ln-pos*)

lemma *tendsto-ln* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow a \neq 0 \Longrightarrow ((\lambda x.\ ln\ (f\ x)) \longrightarrow ln\ a)\ F$
for a :: *real*
by (*rule isCont-tendsto-compose* [*OF isCont-ln*])

lemma *continuous-ln*:
continuous $F\ f \Longrightarrow f\ (Lim\ F\ (\lambda x.\ x)) \neq 0 \Longrightarrow$ *continuous* $F\ (\lambda x.\ ln\ (f\ x :: real))$
unfolding *continuous-def* **by** (*rule tendsto-ln*)

lemma *isCont-ln'* [*continuous-intros*]:
continuous $(at\ x)\ f \Longrightarrow f\ x \neq 0 \Longrightarrow$ *continuous* $(at\ x)\ (\lambda x.\ ln\ (f\ x :: real))$
unfolding *continuous-at* **by** (*rule tendsto-ln*)

lemma *continuous-within-ln* [*continuous-intros*]:
continuous $(at\ x\ within\ s)\ f \Longrightarrow f\ x \neq 0 \Longrightarrow$ *continuous* $(at\ x\ within\ s)\ (\lambda x.\ ln$

```
(f x :: real))
  unfolding continuous-within by (rule tendsto-ln)

lemma continuous-on-ln [continuous-intros]:
  continuous-on s f ⟹ (∀ x∈s. f x ≠ 0) ⟹ continuous-on s (λx. ln (f x :: real))
  unfolding continuous-on-def by (auto intro: tendsto-ln)

lemma DERIV-ln: 0 < x ⟹ DERIV ln x :> inverse x
  for x :: real
  by (rule DERIV-inverse-function [where f=exp and a=0 and b=x+1])
    (auto intro: DERIV-cong [OF DERIV-exp exp-ln] isCont-ln)

lemma DERIV-ln-divide: 0 < x ⟹ DERIV ln x :> 1/x
  for x :: real
  by (rule DERIV-ln[THEN DERIV-cong]) (simp-all add: divide-inverse)

declare DERIV-ln-divide[THEN DERIV-chain2, derivative-intros]
  and DERIV-ln-divide[THEN DERIV-chain2, unfolded has-field-derivative-def,
derivative-intros]

lemmas has-derivative-ln[derivative-intros] = DERIV-ln[THEN DERIV-compose-FDERIV]

lemma ln-series:
  assumes 0 < x and x < 2
  shows ln x = (∑ n. (-1)^n * (1 / real (n + 1)) * (x - 1)^(Suc n))
    (is ln x = suminf (?f (x - 1)))
proof -
  let ?f' = λx n. (-1)^n * (x - 1)^n

  have ln x - suminf (?f (x - 1)) = ln 1 - suminf (?f (1 - 1))
  proof (rule DERIV-isconst3 [where x = x])
    fix x :: real
    assume x ∈ {0 <..< 2}
    then have 0 < x and x < 2 by auto
    have norm (1 - x) < 1
      using ‹0 < x› and ‹x < 2› by auto
    have 1/x = 1 / (1 - (1 - x)) by auto
    also have ... = (∑ n. (1 - x)^n)
      using geometric-sums[OF ‹norm (1 - x) < 1›] by (rule sums-unique)
    also have ... = suminf (?f' x)
      unfolding power-mult-distrib[symmetric]
      by (rule arg-cong[where f=suminf], rule arg-cong[where f=(^)], auto)
    finally have DERIV ln x :> suminf (?f' x)
      using DERIV-ln[OF ‹0 < x›] unfolding divide-inverse by auto
    moreover
    have repos: ⋀ h x :: real. h - 1 + x = h + x - 1 by auto
    have DERIV (λx. suminf (?f x)) (x - 1) :>
      (∑ n. (-1)^n * (1 / real (n + 1)) * real (Suc n) * (x - 1) ^ n)
    proof (rule DERIV-power-series')
```

```
    show x − 1 ∈ {− 1<..<1} and (0 :: real) < 1
      using ‹0 < x› ‹x < 2› by auto
  next
    fix x :: real
    assume x ∈ {− 1<..<1}
    then show summable (λn. (− 1) ^ n * (1 / real (n + 1)) * real (Suc n) *
x^n)
      by (simp add: abs-if flip: power-mult-distrib)
  qed
  then have DERIV (λx. suminf (?f x)) (x − 1) :> suminf (?f' x)
    unfolding One-nat-def by auto
  then have DERIV (λx. suminf (?f (x − 1))) x :> suminf (?f' x)
    unfolding DERIV-def repos .
  ultimately have DERIV (λx. ln x − suminf (?f (x − 1))) x :> suminf (?f'
x) − suminf (?f' x)
    by (rule DERIV-diff)
  then show DERIV (λx. ln x − suminf (?f (x − 1))) x :> 0 by auto
 qed (auto simp: assms)
 then show ?thesis by auto
qed

lemma exp-first-terms:
 fixes x :: 'a::{real-normed-algebra-1,banach}
 shows exp x = (∑ n<k. inverse(fact n) *R (x ^ n)) + (∑ n. inverse(fact (n +
k)) *R (x ^ (n + k)))
proof −
 have exp x = suminf (λn. inverse(fact n) *R (x^n))
   by (simp add: exp-def)
 also from summable-exp-generic have ... = (∑ n. inverse(fact(n+k)) *R (x ^
(n + k))) +
   (∑ n::nat<k. inverse(fact n) *R (x^n)) (is - = - + ?a)
   by (rule suminf-split-initial-segment)
 finally show ?thesis by simp
qed

lemma exp-first-term: exp x = 1 + (∑ n. inverse (fact (Suc n)) *R (x ^ Suc n))
 for x :: 'a::{real-normed-algebra-1,banach}
 using exp-first-terms[of x 1] by simp

lemma exp-first-two-terms: exp x = 1 + x + (∑ n. inverse (fact (n + 2)) *R (x
^ (n + 2)))
 for x :: 'a::{real-normed-algebra-1,banach}
 using exp-first-terms[of x 2] by (simp add: eval-nat-numeral)

lemma exp-bound:
 fixes x :: real
 assumes a: 0 ≤ x
   and b: x ≤ 1
 shows exp x ≤ 1 + x + x^2
```

proof −
have *suminf* (λ*n*. *inverse*(*fact* (*n*+2)) ∗ (*x* ^ (*n* + 2))) ≤ x^2
proof −
have (λ*n*. x^2 / 2 ∗ (1/2) ^ *n*) *sums* (x^2 / 2 ∗ (1 / (1 − 1/2)))
by (*intro sums-mult geometric-sums*) *simp*
then have *sumsx*: (λ*n*. x^2 / 2 ∗ (1/2) ^ *n*) *sums* x^2
by *simp*
have *suminf* (λ*n*. *inverse*(*fact* (*n*+2)) ∗ (*x* ^ (*n* + 2))) ≤ *suminf* (λ*n*. (x^2/2) ∗ ((1/2)^*n*))
proof (*intro suminf-le allI*)
show *inverse* (*fact* (*n* + 2)) ∗ *x* ^ (*n* + 2) ≤ (x^2/2) ∗ ((1/2)^*n*) **for** *n* :: *nat*
proof −
have (2::*nat*) ∗ 2 ^ *n* ≤ *fact* (*n* + 2)
by (*induct n*) *simp-all*
then have *real* ((2::*nat*) ∗ 2 ^ *n*) ≤ *real-of-nat* (*fact* (*n* + 2))
by (*simp only*: *of-nat-le-iff*)
then have ((2::*real*) ∗ 2 ^ *n*) ≤ *fact* (*n* + 2)
unfolding *of-nat-fact* **by** *simp*
then have *inverse* (*fact* (*n* + 2)) ≤ *inverse* ((2::*real*) ∗ 2 ^ *n*)
by (*rule le-imp-inverse-le*) *simp*
then have *inverse* (*fact* (*n* + 2)) ≤ 1/(2::*real*) ∗ (1/2)^*n*
by (*simp add*: *power-inverse* [*symmetric*])
then have *inverse* (*fact* (*n* + 2)) ∗ (*x*^*n* ∗ x^2) ≤ 1/2 ∗ (1/2)^*n* ∗ (1 ∗ x^2)
by (*rule mult-mono*) (*rule mult-mono*, *simp-all add*: *power-le-one a b*)
then show *?thesis*
unfolding *power-add* **by** (*simp add*: *ac-simps del*: *fact-Suc*)
qed
show *summable* (λ*n*. *inverse* (*fact* (*n* + 2)) ∗ *x* ^ (*n* + 2))
by (*rule summable-exp* [*THEN summable-ignore-initial-segment*])
show *summable* (λ*n*. x^2 / 2 ∗ (1/2) ^ *n*)
by (*rule sums-summable* [*OF sumsx*])
qed
also have ... = x^2
by (*rule sums-unique* [*THEN sym*]) (*rule sumsx*)
finally show *?thesis* **.**
qed
then show *?thesis*
unfolding *exp-first-two-terms* **by** *auto*
qed

corollary *exp-half-le2*: *exp*(1/2) ≤ (2::*real*)
using *exp-bound* [*of* 1/2]
by (*simp add*: *field-simps*)

corollary *exp-le*: *exp* 1 ≤ (3::*real*)
using *exp-bound* [*of* 1]
by (*simp add*: *field-simps*)

lemma *exp-bound-half*: *norm z* ≤ 1/2 ⟹ *norm* (*exp z*) ≤ 2

by (*blast intro*: *order-trans intro*!: *exp-half-le2 norm-exp*)

lemma *exp-bound-lemma*:
assumes $norm\ z \leq 1/2$
shows $norm\ (exp\ z) \leq 1 + 2 * norm\ z$
proof −
have ∗: $(norm\ z)^2 \leq norm\ z * 1$
unfolding *power2-eq-square*
by (*rule mult-left-mono*) (*use assms* **in** *auto*)
have $norm\ (exp\ z) \leq exp\ (norm\ z)$
by (*rule norm-exp*)
also have $\ldots \leq 1 + (norm\ z) + (norm\ z)^2$
using *assms exp-bound* **by** *auto*
also have $\ldots \leq 1 + 2 * norm\ z$
using ∗ **by** *auto*
finally show *?thesis* **.**
qed

lemma *real-exp-bound-lemma*: $0 \leq x \Longrightarrow x \leq 1/2 \Longrightarrow exp\ x \leq 1 + 2 * x$
for x :: *real*
using *exp-bound-lemma* [*of x*] **by** *simp*

lemma *ln-one-minus-pos-upper-bound*:
fixes x :: *real*
assumes *a*: $0 \leq x$ **and** *b*: $x < 1$
shows $ln\ (1 - x) \leq - x$
proof −
have $(1 - x) * (1 + x + x^2) = 1 - x\hat{}3$
by (*simp add*: *algebra-simps power2-eq-square power3-eq-cube*)
also have $\ldots \leq 1$
by (*auto simp*: *a*)
finally have $(1 - x) * (1 + x + x^2) \leq 1$ **.**
moreover have *c*: $0 < 1 + x + x^2$
by (*simp add*: *add-pos-nonneg a*)
ultimately have $1 - x \leq 1\ /\ (1 + x + x^2)$
by (*elim mult-imp-le-div-pos*)
also have $\ldots \leq 1\ /\ exp\ x$
by (*metis a abs-one b exp-bound exp-gt-zero frac-le less-eq-real-def real-sqrt-abs real-sqrt-pow2-iff real-sqrt-power*)
also have $\ldots = exp\ (-\ x)$
by (*auto simp*: *exp-minus divide-inverse*)
finally have $1 - x \leq exp\ (-\ x)$ **.**
also have $1 - x = exp\ (ln\ (1 - x))$
by (*metis b diff-0 exp-ln-iff less-iff-diff-less-0 minus-diff-eq*)
finally have $exp\ (ln\ (1 - x)) \leq exp\ (-\ x)$ **.**
then show *?thesis*
by (*auto simp only*: *exp-le-cancel-iff*)
qed

lemma *exp-ge-add-one-self* [*simp*]: $1 + x \leq exp\ x$
for x :: *real*
proof (*cases* $0 \leq x \vee x \leq -1$)
case *True*
then show *?thesis*
by (*meson exp-ge-add-one-self-aux exp-ge-zero order.trans real-add-le-0-iff*)
next
case *False*
then have *ln1*: $ln\ (1 + x) \leq x$
using *ln-one-minus-pos-upper-bound* [*of* $-x$] **by** *simp*
have $1 + x = exp\ (ln\ (1 + x))$
using *False* **by** *auto*
also have $\ldots \leq exp\ x$
by (*simp add*: *ln1*)
finally show *?thesis* **.**
qed

lemma *exp-gt-self*: $x < exp\ (x::real)$
using *exp-gt-zero ln-less-self* **by** *fastforce*

lemma *ln-one-plus-pos-lower-bound*:
fixes x :: *real*
assumes *a*: $0 \leq x$ **and** *b*: $x \leq 1$
shows $x - x^2 \leq ln\ (1 + x)$
proof −
have $exp\ (x - x^2) = exp\ x\ /\ exp\ (x^2)$
by (*rule exp-diff*)
also have $\ldots \leq (1 + x + x^2)\ /\ exp\ (x\ ^2)$
by (*metis a b divide-right-mono exp-bound exp-ge-zero*)
also have $\ldots \leq (1 + x + x^2)\ /\ (1 + x^2)$
by (*simp add*: *a divide-left-mono add-pos-nonneg*)
also from *a* **have** $\ldots \leq 1 + x$
by (*simp add*: *field-simps add-strict-increasing zero-le-mult-iff*)
finally have $exp\ (x - x^2) \leq 1 + x$ **.**
also have $\ldots = exp\ (ln\ (1 + x))$
proof −
from *a* **have** $0 < 1 + x$ **by** *auto*
then show *?thesis*
by (*auto simp only*: *exp-ln-iff* [*THEN sym*])
qed
finally have $exp\ (x - x^2) \leq exp\ (ln\ (1 + x))$ **.**
then show *?thesis*
by (*metis exp-le-cancel-iff*)
qed

lemma *ln-one-minus-pos-lower-bound*:
fixes x :: *real*
assumes *a*: $0 \leq x$ **and** *b*: $x \leq 1/2$
shows $-x - 2 * x^2 \leq ln\ (1 - x)$

proof −
from *b* **have** *c*: $x < 1$ **by** *auto*
then have $ln\ (1 - x) = -\ ln\ (1 + x\ /\ (1 - x))$
by (*auto simp*: *ln-inverse* [*symmetric*] *field-simps intro*: *arg-cong* [**where** *f*=*ln*])
also have $-\ (x\ /\ (1 - x)) \leq \ldots$
proof −
have $ln\ (1 + x\ /\ (1 - x)) \leq x\ /\ (1 - x)$
using *a c* **by** (*intro ln-add-one-self-le-self*) *auto*
then show *?thesis*
by *auto*
qed
also have $-\ (x\ /\ (1 - x)) = -\ x\ /\ (1 - x)$
by *auto*
finally have *d*: $-\ x\ /\ (1 - x) \leq ln\ (1 - x)$ **.**
have $0 < 1 - x$ **using** *a b* **by** *simp*
then have *e*: $-\ x - 2 * x^2 \leq -\ x\ /\ (1 - x)$
using *mult-right-le-one-le*[*of* $x * x\ 2 * x$] *a b*
by (*simp add*: *field-simps power2-eq-square*)
from *e d* **show** $-\ x - 2 * x^2 \leq ln\ (1 - x)$
by (*rule order-trans*)
qed

lemma *ln-add-one-self-le-self2*:
fixes x :: *real*
shows $-1 < x \Longrightarrow ln\ (1 + x) \leq x$
by (*metis diff-gt-0-iff-gt diff-minus-eq-add exp-ge-add-one-self exp-le-cancel-iff exp-ln minus-less-iff*)

lemma *abs-ln-one-plus-x-minus-x-bound-nonneg*:
fixes x :: *real*
assumes *x*: $0 \leq x$ **and** *x1*: $x \leq 1$
shows $|ln\ (1 + x) - x| \leq x^2$
proof −
from *x* **have** $ln\ (1 + x) \leq x$
by (*rule ln-add-one-self-le-self*)
then have $ln\ (1 + x) - x \leq 0$
by *simp*
then have $|ln(1 + x) - x| = -\ (ln(1 + x) - x)$
by (*rule abs-of-nonpos*)
also have $\ldots = x - ln\ (1 + x)$
by *simp*
also have $\ldots \leq x^2$
proof −
from *x x1* **have** $x - x^2 \leq ln\ (1 + x)$
by (*intro ln-one-plus-pos-lower-bound*)
then show *?thesis*
by *simp*
qed
finally show *?thesis* **.**

qed

lemma *abs-ln-one-plus-x-minus-x-bound-nonpos*:
fixes x :: *real*
assumes *a*: $-(1/2) \leq x$ **and** *b*: $x \leq 0$
shows $|ln\ (1 + x) - x| \leq 2 * x^2$
proof −
have ∗: $-\ (-x) - 2 * (-x)^2 \leq ln\ (1 - (-\ x))$
by (*metis a b diff-zero ln-one-minus-pos-lower-bound minus-diff-eq neg-le-iff-le*)

have $|ln\ (1 + x) - x| = x - ln\ (1 - (-\ x))$
using *a ln-add-one-self-le-self2* [*of x*] **by** (*simp add*: *abs-if*)
also have $\ldots \leq 2 * x^2$
using ∗ **by** (*simp add*: *algebra-simps*)
finally show *?thesis* **.**
qed

lemma *abs-ln-one-plus-x-minus-x-bound*:
fixes x :: *real*
assumes $|x| \leq 1/2$
shows $|ln\ (1 + x) - x| \leq 2 * x^2$
proof (*cases* $0 \leq x$)
case *True*
then show *?thesis*
using *abs-ln-one-plus-x-minus-x-bound-nonneg assms* **by** *fastforce*
next
case *False*
then show *?thesis*
using *abs-ln-one-plus-x-minus-x-bound-nonpos assms* **by** *auto*
qed

lemma *ln-x-over-x-mono*:
fixes x :: *real*
assumes *x*: $exp\ 1 \leq x\ x \leq y$
shows $ln\ y\ /\ y \leq ln\ x\ /\ x$
proof −
note x
moreover have $0 < exp\ (1::real)$ **by** *simp*
ultimately have *a*: $0 < x$ **and** *b*: $0 < y$
by (*fast intro*: *less-le-trans order-trans*)+
have $x * ln\ y - x * ln\ x = x * (ln\ y - ln\ x)$
by (*simp add*: *algebra-simps*)
also have $\ldots = x * ln\ (y\ /\ x)$
using *a b ln-div* **by** *force*
also have $y\ /\ x = (x + (y - x))\ /\ x$
by *simp*
also have $\ldots = 1 + (y - x)\ /\ x$
using *x a* **by** (*simp add*: *field-simps*)
also have $x * ln\ (1 + (y - x)\ /\ x) \leq x * ((y - x)\ /\ x)$

```
    using x a
    by (intro mult-left-mono ln-add-one-self-le-self) simp-all
  also have ... = y - x
    using a by simp
  also have ... = (y - x) * ln (exp 1) by simp
  also have ... ≤ (y - x) * ln x
    using a x exp-total of-nat-1 x(1)  by (fastforce intro: mult-left-mono)
  also have ... = y * ln x - x * ln x
    by (rule left-diff-distrib)
  finally have x * ln y ≤ y * ln x
    by arith
  then have ln y ≤ (y * ln x) / x
    using a by (simp add: field-simps)
  also have ... = y * (ln x / x) by simp
  finally show ?thesis
    using b by (simp add: field-simps)
qed

lemma ln-le-minus-one: 0 < x ⟹ ln x ≤ x - 1
  for x :: real
using exp-ge-add-one-self[of ln x] by simp

corollary ln-diff-le: 0 < x ⟹ 0 < y ⟹ ln x - ln y ≤ (x - y) / y
  for x :: real
by (metis diff-divide-distrib divide-pos-pos divide-self ln-divide-pos ln-le-minus-one
order-less-irrefl)

lemma ln-add1-ge:
  fixes t::real
  shows t≥0 ⟹ ln (t+1) ≥ t / (1+t)
using ln-diff-le [of 1 t+1] by (simp add: add.commute)

lemma ln-eq-minus-one:
  fixes x :: real
  assumes 0 < x ln x = x - 1
  shows x = 1
proof -
  let ?l = λy. ln y - y + 1
  have D: ⋀x::real. 0 < x ⟹ DERIV ?l x :> (1/x - 1)
    by (auto intro!: derivative-eq-intros)
  show ?thesis
  proof (cases rule: linorder-cases)
    assume x < 1
    from dense[OF ‹x < 1›] obtain a where x < a a < 1 by blast
    from ‹x < a› have ?l x < ?l a
    proof (rule DERIV-pos-imp-increasing)
      fix y
      assume x ≤ y y ≤ a
      with ‹0 < x› ‹a < 1› have 0 < 1 / y - 1 0 < y
```

```
      by (auto simp: field-simps)
    with D show ∃z. DERIV ?l y :> z ∧ 0 < z by blast
  qed
  also have ... ≤ 0
    using ln-le-minus-one ‹0 < x› ‹x < a› by (auto simp: field-simps)
  finally show x = 1 using assms by auto
 next
  assume 1 < x
  from dense[OF this] obtain a where 1 < a a < x by blast
  from ‹a < x› have ?l x < ?l a
  proof (rule DERIV-neg-imp-decreasing)
    fix y
    assume a ≤ y y ≤ x
    with ‹1 < a› have 1 / y - 1 < 0 0 < y
      by (auto simp: field-simps)
    with D show ∃z. DERIV ?l y :> z ∧ z < 0
      by blast
  qed
  also have ... ≤ 0
    using ln-le-minus-one ‹1 < a› by (auto simp: field-simps)
  finally show x = 1 using assms by auto
 next
  assume x = 1
  then show ?thesis by simp
 qed
qed

lemma has-log-derivative-imp-has-field-derivative-ln-real:
 assumes f x > (0::real) (f has-log-derivative D) (at x within A)
 shows   ((λx. ln (f x)) has-field-derivative D) (at x within A)
proof (cases at x within A = bot)
 case [simp]: False
 have [simp]: Lim (at x within A) (λx. x) = x
   by (simp add: Lim-ident-at)
 show ?thesis using assms
   by (auto intro!: derivative-eq-intros simp: has-log-derivative-def)
qed auto

corollary ln-diff-less: 0 < x ⟹ 0 < y ⟹ x ≠ y ⟹ ln x - ln y < (x - y) / y
for x :: real
using ln-eq-minus-one[of x/y] ln-diff-le[of x y]
by (fastforce simp: diff-divide-distrib ln-divide-pos)

lemma ln-add1-gt:
 fixes t::real
 shows t>0 ⟹ ln (t+1) > t / (1+t)
using ln-diff-less [of 1 t+1] ln-one by(simp add: diff-divide-distrib add.commute)

lemma ln-add-one-self-less-self:
```

fixes $x :: real$
assumes $x > 0$
shows $ln\ (1 + x) < x$
by (*smt* (*verit*, *best*) *assms ln-eq-minus-one ln-le-minus-one*)

lemma *ln-x-over-x-tendsto-0*: $((\lambda x{::}real.\ ln\ x\ /\ x) \longrightarrow 0)\ at\text{-}top$
proof (*rule lhospital-at-top-at-top*[**where** $f' = inverse$ **and** $g' = \lambda\text{-}.\ 1$])
from *eventually-gt-at-top*[*of* $0{::}real$]
show $\forall_F\ x\ in\ at\text{-}top.\ (ln\ has\text{-}real\text{-}derivative\ inverse\ x)\ (at\ x)$
by *eventually-elim* (*auto intro*!: *derivative-eq-intros simp*: *field-simps*)
qed (*use tendsto-inverse-0* **in**
‹*auto simp*: *filterlim-ident dest*!: *tendsto-mono*[*OF at-top-le-at-infinity*]›)

corollary *exp-1-gt-powr*:
assumes $x > (0{::}real)$
shows $exp\ 1 > (1 + 1/x)\ powr\ x$
proof −
have $ln\ (1 + 1/x) < 1/x$
using *ln-add-one-self-less-self assms* **by** *simp*
thus $exp\ 1 > (1 + 1/x)\ powr\ x$ **using** *assms*
by (*simp add*: *field-simps powr-def*)
qed

lemma *exp-ge-one-plus-x-over-n-power-n*:
assumes $x \geq -\ real\ n\ n > 0$
shows $(1 + x\ /\ of\text{-}nat\ n)\ \hat{}\ n \leq exp\ x$
proof (*cases* $x = -\ of\text{-}nat\ n$)
case *False*
from *assms False* **have** $(1 + x\ /\ of\text{-}nat\ n)\ \hat{}\ n = exp\ (of\text{-}nat\ n * ln\ (1 + x\ /\ of\text{-}nat\ n))$
by (*subst exp-of-nat-mult*, *subst exp-ln*) (*simp-all add*: *field-simps*)
also from *assms False* **have** $ln\ (1 + x\ /\ real\ n) \leq x\ /\ real\ n$
by (*intro ln-add-one-self-le-self2*) (*simp-all add*: *field-simps*)
with *assms* **have** $exp\ (of\text{-}nat\ n * ln\ (1 + x\ /\ of\text{-}nat\ n)) \leq exp\ x$
by (*simp add*: *field-simps*)
finally show *?thesis* **.**
next
case *True*
then show *?thesis* **by** (*simp add*: *zero-power*)
qed

lemma *exp-ge-one-minus-x-over-n-power-n*:
assumes $x \leq real\ n\ n > 0$
shows $(1 - x\ /\ of\text{-}nat\ n)\ \hat{}\ n \leq exp\ (-x)$
using *exp-ge-one-plus-x-over-n-power-n*[*of* $n\ -x$] *assms* **by** *simp*

lemma *exp-at-bot*: $(exp \longrightarrow (0{::}real))\ at\text{-}bot$
unfolding *tendsto-Zfun-iff*
proof (*rule ZfunI*, *simp add*: *eventually-at-bot-dense*)

```
  fix r :: real
  assume 0 < r
  have exp x < r if x < ln r for x
    by (metis ‹0 < r› exp-less-mono exp-ln that)
  then show ∃ k. ∀ n<k. exp n < r by auto
qed

lemma exp-at-top: LIM x at-top. exp x :: real :> at-top
  by (rule filterlim-at-top-at-top[where Q=λx. True and P=λx. 0 < x and g=ln])
     (auto intro: eventually-gt-at-top)

lemma lim-exp-minus-1: ((λz::'a. (exp(z) - 1) / z) ⟶ 1) (at 0)
  for x :: 'a::{real-normed-field,banach}
proof -
  have ((λz::'a. exp(z) - 1) has-field-derivative 1) (at 0)
    by (intro derivative-eq-intros | simp)+
  then show ?thesis
    by (simp add: Deriv.has-field-derivative-iff)
qed

lemma ln-at-0: LIM x at-right 0. ln (x::real) :> at-bot
   by (rule filterlim-at-bot-at-right[where Q=λx. 0 < x and P=λx. True and
g=exp])
     (auto simp: eventually-at-filter)

lemma ln-at-top: LIM x at-top. ln (x::real) :> at-top
   by (rule filterlim-at-top-at-top[where Q=λx. 0 < x and P=λx. True and
g=exp])
     (auto intro: eventually-gt-at-top)

lemma filtermap-ln-at-top: filtermap (ln::real ⇒ real) at-top = at-top
  by (intro filtermap-fun-inverse[of exp] exp-at-top ln-at-top) auto

lemma filtermap-exp-at-top: filtermap (exp::real ⇒ real) at-top = at-top
  by (intro filtermap-fun-inverse[of ln] exp-at-top ln-at-top)
     (auto simp: eventually-at-top-dense)

lemma filtermap-ln-at-right: filtermap ln (at-right (0::real)) = at-bot
  by (auto intro!: filtermap-fun-inverse[where g=λx. exp x] ln-at-0
      simp: filterlim-at exp-at-bot)

lemma tendsto-power-div-exp-0: ((λx. x ^ k / exp x) ⟶ (0::real)) at-top
proof (induct k)
  case 0
  show ((λx. x ^ 0 / exp x) ⟶ (0::real)) at-top
    by (simp add: inverse-eq-divide[symmetric])
       (metis filterlim-compose[OF tendsto-inverse-0] exp-at-top filterlim-mono
         at-top-le-at-infinity order-refl)
next
```

case (*Suc k*)
show *?case*
proof (*rule lhospital-at-top-at-top*)
show *eventually* (λ*x*. *DERIV* (λ*x*. *x* ^ *Suc k*) *x* :> (*real* (*Suc k*) ∗ *x*^*k*)) *at-top*
by *eventually-elim* (*intro derivative-eq-intros*, *auto*)
show *eventually* (λ*x*. *DERIV exp x* :> *exp x*) *at-top*
by *eventually-elim auto*
show *eventually* (λ*x*. *exp x* ≠ *0*) *at-top*
by *auto*
from *tendsto-mult*[*OF tendsto-const Suc*, *of real* (*Suc k*)]
show ((λ*x*. *real* (*Suc k*) ∗ *x* ^ *k* / *exp x*) ⟶ *0*) *at-top*
by *simp*
qed (*rule exp-at-top*)
qed

112.7.1 A couple of simple bounds

lemma *exp-plus-inverse-exp*:
fixes *x*::*real*
shows *2* ≤ *exp x* + *inverse* (*exp x*)
proof −
have *2* ≤ *exp x* + *exp* (−*x*)
using *exp-ge-add-one-self* [*of x*] *exp-ge-add-one-self* [*of* −*x*]
by *linarith*
then show *?thesis*
by (*simp add*: *exp-minus*)
qed

lemma *real-le-x-sinh*:
fixes *x*::*real*
assumes *0* ≤ *x*
shows *x* ≤ (*exp x* − *inverse*(*exp x*)) / *2*
proof −
have ∗: *exp a* − *inverse*(*exp a*) − *2*∗*a* ≤ *exp b* − *inverse*(*exp b*) − *2*∗*b* **if** *a* ≤ *b*
for *a b*::*real*
using *exp-plus-inverse-exp*
by (*fastforce intro*: *derivative-eq-intros DERIV-nonneg-imp-nondecreasing* [*OF that*])
show *?thesis*
using∗[*OF assms*] **by** *simp*
qed

lemma *real-le-abs-sinh*:
fixes *x*::*real*
shows *abs x* ≤ *abs*((*exp x* − *inverse*(*exp x*)) / *2*)
proof (*cases 0* ≤ *x*)
case *True*
show *?thesis*
using *real-le-x-sinh* [*OF True*] *True* **by** (*simp add*: *abs-if*)

next
case *False*
have $-x \leq (exp(-x) - inverse(exp(-x))) / 2$
by (*meson False linear neg-le-0-iff-le real-le-x-sinh*)
also have $\ldots \leq |(exp\ x - inverse\ (exp\ x)) / 2|$
by (*metis* (*no-types*, *opaque-lifting*) *abs-divide abs-le-iff abs-minus-cancel add.inverse-inverse exp-minus minus-diff-eq order-refl*)
finally show *?thesis*
using *False* **by** *linarith*
qed

112.8 The general logarithm

definition *log* :: *real* ⇒ *real* ⇒ *real*
— logarithm of x to base a
where *log a x* = *ln x* / *ln a*

lemma *log-exp* [*simp*]: *log b* (*exp x*) = *x* / *ln b*
by (*simp add*: *log-def*)

lemma *tendsto-log* [*tendsto-intros*]:
(*f* ⟶ *a*) *F* ⟹ (*g* ⟶ *b*) *F* ⟹ *0* < *a* ⟹ *a* ≠ *1* ⟹ *b*≠*0* ⟹
((λ*x*. *log* (*f x*) (*g x*)) ⟶ *log a b*) *F*
unfolding *log-def* **by** (*intro tendsto-intros*) *auto*

lemma *continuous-log*:
assumes *continuous F f*
and *continuous F g*
and *f* (*Lim F* (λ*x*. *x*)) > *0*
and *f* (*Lim F* (λ*x*. *x*)) ≠ *1*
and *g* (*Lim F* (λ*x*. *x*)) ≠ *0*
shows *continuous F* (λ*x*. *log* (*f x*) (*g x*))
using *assms* **by** (*simp add*: *continuous-def tendsto-log*)

lemma *continuous-at-within-log*[*continuous-intros*]:
assumes *continuous* (*at a within s*) *f*
and *continuous* (*at a within s*) *g*
and *0* < *f a*
and *f a* ≠ *1*
and *g a* ≠ *0*
shows *continuous* (*at a within s*) (λ*x*. *log* (*f x*) (*g x*))
using *assms* **unfolding** *continuous-within* **by** (*rule tendsto-log*)

lemma *continuous-on-log*[*continuous-intros*]:
assumes *continuous-on S f continuous-on S g*
and ∀ *x*∈*S*. *0* < *f x* ∀ *x*∈*S*. *f x* ≠ *1* ∀ *x*∈*S*. *g x* ≠ *0*
shows *continuous-on S* (λ*x*. *log* (*f x*) (*g x*))
using *assms* **unfolding** *continuous-on-def* **by** (*fast intro*: *tendsto-log*)

lemma *exp-powr-real*:
fixes *x*::*real* **shows** *exp x powr y* = *exp* (*x*∗*y*)
by (*simp add*: *powr-def*)

lemma *powr-one-eq-one* [*simp*]: *1 powr a* = *1*
by (*simp add*: *powr-def*)

lemma *powr-zero-eq-one* [*simp*]: *x powr 0* = (*if x* = *0 then 0 else 1*)
by (*simp add*: *powr-def*)

lemma *powr-eq-one-iff-gen*[*simp*]: *a powr x* = *1* ⟷ *x* = *0* **if** *a* > *0 a* ≠ *1* **for** *a x* :: *real*
using *that* **by** (*simp add*: *powr-def*)

lemma *powr-one-gt-zero-iff* [*simp*]: *x powr 1* = *x* ⟷ *0* ≤ *x*
for *x* :: *real*
by (*auto simp*: *powr-def*)
declare *powr-one-gt-zero-iff* [*THEN iffD2*, *simp*]

lemma *powr-diff*:
fixes *w*:: *'a*::{*ln*,*real-normed-field*}
shows *w powr* (*z1* − *z2*) = *w powr z1* / *w powr z2*
by (*simp add*: *powr-def algebra-simps exp-diff*)

lemma *powr-mult*: (*x* ∗ *y*) *powr a* = (*x powr a*) ∗ (*y powr a*)
for *a x y* :: *real*
by (*simp add*: *powr-def exp-add* [*symmetric*] *ln-mult distrib-left*)

lemma *prod-powr-distrib*:
fixes *x* :: *'a* ⇒ *real*
shows (*prod x I*) *powr r* = (∏ *i*∈*I*. *x i powr r*)
by (*induction I rule*: *infinite-finite-induct*) (*auto simp add*: *powr-mult prod-nonneg*)

lemma *powr-ge-zero* [*simp*]: *0* ≤ *x powr y*
for *x y* :: *real*
by (*simp add*: *powr-def*)

lemma *powr-non-neg*[*simp*]: ¬ *a powr x* < *0* **for** *a x*::*real*
using *powr-ge-zero*[*of a x*] **by** *arith*

lemma *inverse-powr*: ⋀*y*::*real*. *inverse y powr a* = *inverse* (*y powr a*)
by (*simp add*: *exp-minus ln-inverse powr-def*)

lemma *powr-divide*: (*x* / *y*) *powr a* = (*x powr a*) / (*y powr a*)
for *a b x* :: *real*
by (*simp add*: *divide-inverse powr-mult inverse-powr*)

lemma *powr-add*: *x powr* (*a* + *b*) = (*x powr a*) ∗ (*x powr b*)
for *a b x* :: *'a*::{*ln*,*real-normed-field*}

by (*simp add*: *powr-def exp-add* [*symmetric*] *distrib-right*)

lemma *powr-mult-base*: $0 \leq x \Longrightarrow x * x \text{ powr } y = x \text{ powr } (1 + y)$
for x :: *real*
by (*auto simp*: *powr-add*)

lemma *powr-mult-base'*: $\text{abs } x * x \text{ powr } y = x \text{ powr } (1 + y)$
for x :: *real*
by (*smt* (*verit*) *powr-mult-base uminus-powr-eq*)

lemma *powr-powr*: $(x \text{ powr } a) \text{ powr } b = x \text{ powr } (a * b)$
for $a\ b\ x$ :: *real*
by (*simp add*: *powr-def*)

lemma *powr-power*:
fixes z:: *'a*::{*real-normed-field*,*ln*}
shows $z \neq 0 \Longrightarrow (z \text{ powr } u) \hat{} n = z \text{ powr } (\text{of-nat } n * u)$
by (*induction n*) (*auto simp*: *algebra-simps powr-add*)

lemma *powr-powr-swap*: $(x \text{ powr } a) \text{ powr } b = (x \text{ powr } b) \text{ powr } a$
for $a\ b\ x$ :: *real*
by (*simp add*: *powr-powr mult.commute*)

lemma *powr-minus*: $x \text{ powr } (-a) = \text{inverse } (x \text{ powr } a)$
for $a\ x$ :: *'a*::{*ln*,*real-normed-field*}
by (*simp add*: *powr-def exp-minus* [*symmetric*])

lemma *powr-minus-divide*: $x \text{ powr } (-a) = 1/(x \text{ powr } a)$
for $a\ x$ :: *'a*::{*ln*,*real-normed-field*}
by (*simp add*: *divide-inverse powr-minus*)

lemma *powr-sum*:
assumes $x \neq 0$
shows $x \text{ powr sum } f\ A = (\prod y{\in}A.\ x \text{ powr } f\ y)$
proof (*cases finite A*)
case *True*
with *assms* **show** *?thesis*
by (*simp add*: *powr-def exp-sum sum-distrib-right*)
next
case *False*
with *assms* **show** *?thesis* **by** *auto*
qed

lemma *divide-powr-uminus*: $a\ /\ b \text{ powr } c = a * b \text{ powr } (-c)$
for $a\ b\ c$ :: *real*
by (*simp add*: *powr-minus-divide*)

lemma *powr-less-mono*: $a < b \Longrightarrow 1 < x \Longrightarrow x \text{ powr } a < x \text{ powr } b$
for $a\ b\ x$ :: *real*

by (*simp add*: *powr-def*)

lemma *powr-less-cancel*: x *powr* $a < x$ *powr* $b \Longrightarrow 1 < x \Longrightarrow a < b$
for a b x :: *real*
by (*simp add*: *powr-def*)

lemma *powr-less-cancel-iff* [*simp*]: $1 < x \Longrightarrow x$ *powr* $a < x$ *powr* $b \longleftrightarrow a < b$
for a b x :: *real*
by (*blast intro*: *powr-less-cancel powr-less-mono*)

lemma *powr-le-cancel-iff* [*simp*]: $1 < x \Longrightarrow x$ *powr* $a \leq x$ *powr* $b \longleftrightarrow a \leq b$
for a b x :: *real*
by (*simp add*: *linorder-not-less* [*symmetric*])

lemma *powr-realpow*: $0 < x \Longrightarrow x$ *powr* (*real* n) $= x \hat{} n$
by (*induction* n) (*simp-all add*: *ac-simps powr-add*)

lemma *powr-realpow'*: (z :: *real*) $\geq 0 \Longrightarrow n \neq 0 \Longrightarrow z$ *powr of-nat* $n = z \hat{} n$
by (*cases* $z = 0$) (*auto simp*: *powr-realpow*)

lemma *powr-real-of-int'*:
assumes $x \geq 0$ $x \neq 0 \vee n > 0$
shows x *powr real-of-int* $n =$ *power-int* x n
by (*metis assms exp-ln-iff exp-power-int nless-le power-int-eq-0-iff powr-def*)

lemma *exp-minus-ge*:
fixes x::*real* **shows** $1 - x \leq$ *exp* $(-x)$
by (*smt* (*verit*) *exp-ge-add-one-self*)

lemma *exp-minus-greater*:
fixes x::*real* **shows** $1 - x <$ *exp* $(-x) \longleftrightarrow x \neq 0$
by (*smt* (*verit*) *exp-minus-ge exp-eq-one-iff exp-gt-zero ln-eq-minus-one ln-exp*)

lemma *log-ln*: *ln* $x =$ *log* (*exp* 1) x
by (*simp add*: *log-def*)

lemma *DERIV-log*:
assumes $x > 0$
shows *DERIV* $(\lambda y.$ *log* b $y)$ $x :>$ 1 / (*ln* $b * x$)
proof −
define *lb* **where** *lb* $=$ 1 / *ln* b
moreover have *DERIV* $(\lambda y.$ *lb* $*$ *ln* $y)$ $x :>$ *lb* / x
using ‹$x > 0$› **by** (*auto intro*!: *derivative-eq-intros*)
ultimately show *?thesis*
by (*simp add*: *log-def*)
qed

lemmas *DERIV-log*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-log*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *deriva-*

tive-intros]

lemma *powr-log-cancel* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow 0 < x \Longrightarrow a\ powr\ (log\ a\ x) = x$
by (*simp add*: *powr-def log-def*)

lemma *log-powr-cancel* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow log\ a\ (a\ powr\ x) = x$
by (*simp add*: *log-def powr-def*)

lemma *powr-eq-iff*: $[\![y{>}0;\ a{>}1]\!] \Longrightarrow a\ powr\ x = y \longleftrightarrow log\ a\ y = x$
by *auto*

lemma *log-mult*:
$log\ a\ (x * y) = (if\ x{\neq}0 \wedge y{\neq}0\ then\ log\ a\ x + log\ a\ y\ else\ 0)$
by (*simp add*: *log-def ln-mult divide-inverse distrib-right*)

lemma *log-mult-pos*:
$x{>}0 \Longrightarrow y{>}0 \Longrightarrow log\ a\ (x * y) = log\ a\ x + log\ a\ y$
by (*simp add*: *log-def ln-mult divide-inverse distrib-right*)

lemma *log-eq-div-ln-mult-log*:
$0 < b \Longrightarrow b \neq 1 \Longrightarrow 0 < x \Longrightarrow log\ a\ x = (ln\ b/ln\ a) * log\ b\ x$
by (*simp add*: *log-def divide-inverse*)

Base 10 logarithms

lemma *log-base-10-eq1*: $0 < x \Longrightarrow log\ 10\ x = (ln\ (exp\ 1)\ /\ ln\ 10) * ln\ x$
by (*simp add*: *log-def*)

lemma *log-base-10-eq2*: $0 < x \Longrightarrow log\ 10\ x = (log\ 10\ (exp\ 1)) * ln\ x$
by (*simp add*: *log-def*)

lemma *log-one* [*simp*]: $log\ a\ 1 = 0$
by (*simp add*: *log-def*)

lemma *log-eq-one* [*simp*]: $0 < a \Longrightarrow a \neq 1 \Longrightarrow log\ a\ a = 1$
by (*simp add*: *log-def*)

lemma *log-inverse*: $log\ a\ (inverse\ x) = -\ log\ a\ x$
by (*simp add*: *ln-inverse log-def*)

lemma *log-recip*: $log\ a\ (1/x) = -\ log\ a\ x$
by (*simp add*: *divide-inverse log-inverse*)

lemma *log-divide*:
$log\ a\ (x\ /\ y) = (if\ x{\neq}0 \wedge y{\neq}0\ then\ log\ a\ x - log\ a\ y\ else\ 0)$
by (*simp add*: *diff-divide-distrib ln-div log-def*)

lemma *log-divide-pos*:
$x{>}0 \Longrightarrow y{>}0 \Longrightarrow log\ a\ (x\ /\ y) = log\ a\ x - log\ a\ y$

using *log-divide* **by** *auto*

lemma *powr-gt-zero* [*simp*]: $0 < x$ *powr* $a \longleftrightarrow x \neq 0$
for $a\ x$ :: *real*
by (*simp add*: *powr-def*)

lemma *powr-nonneg-iff*[*simp*]: a *powr* $x \leq 0 \longleftrightarrow a = 0$
for $a\ x$::*real*
by (*meson not-less powr-gt-zero*)

lemma *log-add-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow x{\neq}0 \Longrightarrow$ *log* $b\ x + y =$ *log* b ($x *$ b *powr* y)
and *add-log-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow x{\neq}0 \Longrightarrow y +$ *log* $b\ x =$ *log* b (b *powr* $y * x$)
and *log-minus-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow x{\neq}0 \Longrightarrow$ *log* $b\ x - y =$ *log* b ($x *$ b *powr* $-y$)
by (*simp-all add*: *log-mult log-divide*)

lemma *minus-log-eq-powr*: $0 < b \Longrightarrow b \neq 1 \Longrightarrow x{\neq}0 \Longrightarrow y -$ *log* $b\ x =$ *log* b (b *powr* y / x)
by (*simp add*: *diff-divide-eq-iff ln-div log-def powr-def*)

lemma *log-less-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow$ *log* $a\ x <$ *log* a $y \longleftrightarrow x < y$
using *powr-less-cancel-iff* [*of a*] *powr-log-cancel* [*of a x*] *powr-log-cancel* [*of a y*]
by (*metis less-eq-real-def less-trans not-le zero-less-one*)

lemma *log-inj*:
assumes $1 < b$
shows *inj-on* (*log* b) {$0 <..$}
proof (*rule inj-onI*, *simp*)
fix $x\ y$
assume *pos*: $0 < x\ 0 < y$ **and** $*$: *log* $b\ x =$ *log* $b\ y$
show $x = y$
proof (*cases rule*: *linorder-cases*)
assume $x = y$
then show *?thesis* **by** *simp*
next
assume $x < y$
then have *log* $b\ x <$ *log* $b\ y$
using *log-less-cancel-iff*[*OF* ‹$1 < b$›] *pos* **by** *simp*
then show *?thesis* **using** $*$ **by** *simp*
next
assume $y < x$
then have *log* $b\ y <$ *log* $b\ x$
using *log-less-cancel-iff*[*OF* ‹$1 < b$›] *pos* **by** *simp*
then show *?thesis* **using** $*$ **by** *simp*
qed
qed

lemma *log-le-cancel-iff* [*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow log\ a\ x \leq log\ a\ y \longleftrightarrow x \leq y$
by (*simp flip*: *linorder-not-less*)

lemma *log-mono*: $1 < a \Longrightarrow 0 < x \Longrightarrow x \leq y \Longrightarrow log\ a\ x \leq log\ a\ y$
by *simp*

lemma *log-less*: $1 < a \Longrightarrow 0 < x \Longrightarrow x < y \Longrightarrow log\ a\ x < log\ a\ y$
by *simp*

lemma *zero-less-log-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 < log\ a\ x \longleftrightarrow 1 < x$
using *log-less-cancel-iff*[*of a 1 x*] **by** *simp*

lemma *zero-le-log-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 0 \leq log\ a\ x \longleftrightarrow 1 \leq x$
using *log-le-cancel-iff*[*of a 1 x*] **by** *simp*

lemma *log-less-zero-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow log\ a\ x < 0 \longleftrightarrow x < 1$
using *log-less-cancel-iff*[*of a x 1*] **by** *simp*

lemma *log-le-zero-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow log\ a\ x \leq 0 \longleftrightarrow x \leq 1$
using *log-le-cancel-iff*[*of a x 1*] **by** *simp*

lemma *one-less-log-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 1 < log\ a\ x \longleftrightarrow a < x$
using *log-less-cancel-iff*[*of a a x*] **by** *simp*

lemma *one-le-log-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow 1 \leq log\ a\ x \longleftrightarrow a \leq x$
using *log-le-cancel-iff*[*of a a x*] **by** *simp*

lemma *log-less-one-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow log\ a\ x < 1 \longleftrightarrow x < a$
using *log-less-cancel-iff*[*of a x a*] **by** *simp*

lemma *log-le-one-cancel-iff*[*simp*]: $1 < a \Longrightarrow 0 < x \Longrightarrow log\ a\ x \leq 1 \longleftrightarrow x \leq a$
using *log-le-cancel-iff*[*of a x a*] **by** *simp*

lemma *le-log-iff*:
fixes *b x y* :: *real*
assumes $1 < b\ x > 0$
shows $y \leq log\ b\ x \longleftrightarrow b\ powr\ y \leq x$
using *assms*
by (*metis less-irrefl less-trans powr-le-cancel-iff powr-log-cancel zero-less-one*)

lemma *less-log-iff*:
assumes $1 < b\ x > 0$
shows $y < log\ b\ x \longleftrightarrow b\ powr\ y < x$
by (*metis assms dual-order.strict-trans less-irrefl powr-less-cancel-iff powr-log-cancel zero-less-one*)

lemma
assumes *1 < b x > 0*
shows *log-less-iff*: *log b x < y ⟷ x < b powr y*
and *log-le-iff*: *log b x ≤ y ⟷ x ≤ b powr y*
using *le-log-iff*[*OF assms*, *of y*] *less-log-iff*[*OF assms*, *of y*]
by *auto*

lemmas *powr-le-iff* = *le-log-iff*[*symmetric*]
and *powr-less-iff* = *less-log-iff*[*symmetric*]
and *less-powr-iff* = *log-less-iff*[*symmetric*]
and *le-powr-iff* = *log-le-iff*[*symmetric*]

lemma *le-log-of-power*:
assumes *b ^ n ≤ m 1 < b*
shows *n ≤ log b m*
proof −
from *assms* **have** *0 < m* **by** (*metis less-trans zero-less-power less-le-trans zero-less-one*)
thus *?thesis* **using** *assms* **by** (*simp add*: *le-log-iff powr-realpow*)
qed

lemma *le-log2-of-power*: *2 ^ n ≤ m ⟹ n ≤ log 2 m* **for** *m n* :: *nat*
using *le-log-of-power*[*of 2*] **by** *simp*

lemma *log-of-power-le*: ⟦ *m ≤ b ^ n*; *b > 1*; *m > 0* ⟧ ⟹ *log b* (*real m*) ≤ *n*
by (*simp add*: *log-le-iff powr-realpow*)

lemma *log2-of-power-le*: ⟦ *m ≤ 2 ^ n*; *m > 0* ⟧ ⟹ *log 2 m ≤ n* **for** *m n* :: *nat*
using *log-of-power-le*[*of - 2*] **by** *simp*

lemma *log-of-power-less*: ⟦ *m < b ^ n*; *b > 1*; *m > 0* ⟧ ⟹ *log b* (*real m*) < *n*
by (*simp add*: *log-less-iff powr-realpow*)

lemma *log2-of-power-less*: ⟦ *m < 2 ^ n*; *m > 0* ⟧ ⟹ *log 2 m < n* **for** *m n* :: *nat*
using *log-of-power-less*[*of - 2*] **by** *simp*

lemma *less-log-of-power*:
assumes *b ^ n < m 1 < b*
shows *n < log b m*
proof −
have *0 < m* **by** (*metis assms less-trans zero-less-power zero-less-one*)
thus *?thesis* **using** *assms* **by** (*simp add*: *less-log-iff powr-realpow*)
qed

lemma *less-log2-of-power*: *2 ^ n < m ⟹ n < log 2 m* **for** *m n* :: *nat*
using *less-log-of-power*[*of 2*] **by** *simp*

lemma *gr-one-powr*[*simp*]:
fixes *x y* :: *real* **shows** ⟦ *x > 1*; *y > 0* ⟧ ⟹ *1 < x powr y*

by(*simp add*: *less-powr-iff*)

lemma *log-pow-cancel* [*simp*]:
a > *0* ⟹ *a* ≠ *1* ⟹ *log a* (*a* ^ *b*) = *b*
by (*simp add*: *ln-realpow log-def*)

lemma *floor-log-eq-powr-iff*: *x* > *0* ⟹ *b* > *1* ⟹ ⌊*log b x*⌋ = *k* ⟷ *b powr k* ≤ *x* ∧ *x* < *b powr* (*k* + *1*)
by (*auto simp*: *floor-eq-iff powr-le-iff less-powr-iff*)

lemma *floor-log-nat-eq-powr-iff*:
fixes *b n k* :: *nat*
shows ⟦ *b* ≥ *2*; *k* > *0* ⟧ ⟹ *floor* (*log b* (*real k*)) = *n* ⟷ *b*^*n* ≤ *k* ∧ *k* < *b*^(*n*+*1*)
by (*auto simp*: *floor-log-eq-powr-iff powr-add powr-realpow*
of-nat-power[*symmetric*] *of-nat-mult*[*symmetric*] *ac-simps*
simp del: *of-nat-power of-nat-mult*)

lemma *floor-log-nat-eq-if*:
fixes *b n k* :: *nat*
assumes *b*^*n* ≤ *k k* < *b*^(*n*+*1*) *b* ≥ *2*
shows *floor* (*log b* (*real k*)) = *n*
proof −
have *k* ≥ *1*
using *assms linorder-le-less-linear* **by** *force*
with *assms* **show** *?thesis*
by(*simp add*: *floor-log-nat-eq-powr-iff*)
qed

lemma *ceiling-log-eq-powr-iff*:
⟦ *x* > *0*; *b* > *1* ⟧ ⟹ ⌈*log b x*⌉ = *int k* + *1* ⟷ *b powr k* < *x* ∧ *x* ≤ *b powr* (*k* + *1*)
by (*auto simp*: *ceiling-eq-iff powr-less-iff le-powr-iff*)

lemma *ceiling-log-nat-eq-powr-iff*:
fixes *b n k* :: *nat*
shows ⟦ *b* ≥ *2*; *k* > *0* ⟧ ⟹ ⌈*log b* (*real k*)⌉ = *int n* + *1* ⟷ (*b*^*n* < *k* ∧ *k* ≤ *b*^(*n*+*1*))
using *ceiling-log-eq-powr-iff*
by (*auto simp*: *powr-add powr-realpow of-nat-power*[*symmetric*] *of-nat-mult*[*symmetric*] *ac-simps*
simp del: *of-nat-power of-nat-mult*)

lemma *ceiling-log-nat-eq-if*:
fixes *b n k* :: *nat*
assumes *b*^*n* < *k k* ≤ *b*^(*n*+*1*) *b* ≥ *2*
shows ⌈*log* (*real b*) (*real k*)⌉ = *int n* + *1*
using *assms ceiling-log-nat-eq-powr-iff* **by** *force*

lemma *floor-log2-div2*:
fixes n :: *nat*
assumes $n \geq 2$
shows $\lfloor log\ 2\ (real\ n) \rfloor = \lfloor log\ 2\ (n\ div\ 2) \rfloor + 1$
proof *cases*
assume *n=2* **thus** *?thesis* **by** *simp*
next
let *?m = n div 2*
assume *n≠2*
hence *1 ≤ ?m* **using** *assms* **by** *arith*
then obtain *i* **where** *i*: $2 \hat{} i \leq ?m\ ?m < 2 \hat{} (i + 1)$
using *ex-power-ivl1*[*of 2 ?m*] **by** *auto*
have $2\hat{}(i+1) \leq 2*?m$ **using** *i*(*1*) **by** *simp*
also have $2*?m \leq n$ **by** *arith*
finally have $*$: $2\hat{}(i+1) \leq \ldots$ **.**
have $n < 2\hat{}(i+1+1)$ **using** *i*(*2*) **by** *simp*
from *floor-log-nat-eq-if*[*OF* $*$ *this*] *floor-log-nat-eq-if*[*OF i*]
show *?thesis* **by** *simp*
qed

lemma *ceiling-log2-div2*:
assumes $n \geq 2$
shows $\lceil log\ 2\ (real\ n) \rceil = \lceil log\ 2\ ((n-1)\ div\ 2 + 1) \rceil + 1$
proof *cases*
assume *n=2* **thus** *?thesis* **by** *simp*
next
let *?m* = $(n-1)$ *div 2 + 1*
assume *n≠2*
hence *2 ≤ ?m* **using** *assms* **by** *arith*
then obtain *i* **where** *i*: $2 \hat{} i < ?m\ ?m \leq 2 \hat{} (i + 1)$
using *ex-power-ivl2*[*of 2 ?m*] **by** *auto*
have $n \leq 2*?m$ **by** *arith*
also have $2*?m \leq 2 \hat{} ((i+1)+1)$ **using** *i*(*2*) **by** *simp*
finally have $*$: $n \leq \ldots$ **.**
have $2\hat{}(i+1) < n$ **using** *i*(*1*) **by** (*auto simp*: *less-Suc-eq-0-disj*)
from *ceiling-log-nat-eq-if*[*OF this* $*$] *ceiling-log-nat-eq-if*[*OF i*]
show *?thesis* **by** *simp*
qed

lemma *powr-real-of-int*:
$x > 0 \Longrightarrow x\ powr\ real\text{-}of\text{-}int\ n = (if\ n \geq 0\ then\ x \hat{} nat\ n\ else\ inverse\ (x \hat{} nat\ (-n)))$
using *powr-realpow*[*of x nat n*] *powr-realpow*[*of x nat* $(-n)$]
by (*auto simp*: *field-simps powr-minus*)

lemma *powr-numeral* [*simp*]: $0 \leq x \Longrightarrow x\ powr\ (numeral\ n :: real) = x \hat{} (numeral\ n)$
by (*metis less-le power-zero-numeral powr-0 of-nat-numeral powr-realpow*)

lemma *powr-int*:
assumes $x > 0$
shows $x\ powr\ i = (if\ i \geq 0\ then\ x\ \hat{}\ nat\ i\ else\ 1/x\ \hat{}\ nat\ (-i))$
by (*simp add*: *assms inverse-eq-divide powr-real-of-int*)

lemma *power-of-nat-log-ge*: $b > 1 \Longrightarrow b\ \hat{}\ nat\ \lceil log\ b\ x \rceil \geq x$
by (*smt* (*verit*) *less-log-of-power of-nat-ceiling*)

lemma *power-of-nat-log-le*:
assumes $b > 1\ x \geq 1$
shows $b\ \hat{}\ nat\ \lfloor log\ b\ x \rfloor \leq x$
proof −
have $\lfloor log\ b\ x \rfloor \geq 0$
using *assms* **by** *auto*
then show *?thesis*
by (*smt* (*verit*) *assms le-log-iff of-int-floor-le powr-int*)
qed

definition *powr-real* :: *real* ⇒ *real* ⇒ *real*
where [*code-abbrev*, *simp*]: *powr-real* = *Transcendental.powr*

lemma *compute-powr-real* [*code*]:
powr-real b i =
(*if b* ≤ *0 then Code.abort* (*STR ''powr-real with nonpositive base''*) (λ-. *powr-real b i*)
else if ⌊*i*⌋ = *i then* (*if* 0 ≤ *i then b* ^ *nat* ⌊*i*⌋ *else* 1 / *b* ^ *nat* ⌊− *i*⌋)
else Code.abort (*STR ''powr-real with non−integer exponent''*) (λ-. *powr-real b i*))
for *b i* :: *real*
by (*auto simp*: *powr-int*)

lemma *powr-one*: $0 \leq x \Longrightarrow x\ powr\ 1 = x$
for x :: *real*
using *powr-realpow* [*of x 1*] **by** *simp*

lemma *powr-one'* [*simp*]: $x\ powr\ 1 = |x|$
for x :: *real*
by (*simp add*: *ln-real-def powr-def*)

lemma *powr-neg-one*: $0 < x \Longrightarrow x\ powr\ -1 = 1/x$
for x :: *real*
using *powr-int* [*of x* − *1*] **by** *simp*

lemma *powr-neg-one'* [*simp*]: $x\ powr\ -1 = 1/|x|$
for x :: *real*
by (*simp add*: *powr-minus-divide*)

lemma *powr-neg-numeral*: $0 < x \Longrightarrow x\ powr\ -\ numeral\ n = 1/x\ \hat{}\ numeral\ n$
for x :: *real*

using *powr-int* [*of* x − *numeral* n] **by** *simp*

lemma *root-powr-inverse*: $0 < n \Longrightarrow 0 \leq x \Longrightarrow root\ n\ x = x\ powr\ (1/n)$
by (*simp add*: *exp-divide-power-eq powr-def real-root-pos-unique*)

lemma *powr-inverse-root*: $0 < n \Longrightarrow x\ powr\ (1/n) = |root\ n\ x|$
by (*metis abs-ge-zero mult-1 powr-one′ powr-powr real-root-abs root-powr-inverse*)

lemma *ln-powr* [*simp*]: $ln\ (x\ powr\ y) = y * ln\ x$
for x :: *real*
by (*simp add*: *powr-def*)

lemma *ln-root*: $n > 0 \Longrightarrow ln\ (root\ n\ b) = ln\ b\ /\ n$
by (*metis ln-powr mult-1 powr-inverse-root powr-one′ times-divide-eq-left*)

lemma *ln-sqrt*: $0 \leq x \Longrightarrow ln\ (sqrt\ x) = ln\ x\ /\ 2$
by (*metis* (*full-types*) *divide-inverse inverse-eq-divide ln-powr mult.commute of-nat-numeral pos2 root-powr-inverse sqrt-def*)

lemma *log-root*: $n > 0 \Longrightarrow a \geq 0 \Longrightarrow log\ b\ (root\ n\ a) = log\ b\ a\ /\ n$
by (*simp add*: *log-def ln-root*)

lemma *log-powr*: $log\ b\ (x\ powr\ y) = y * log\ b\ x$
by (*simp add*: *log-def*)

lemma *log-nat-power*: $0 \leq x \Longrightarrow log\ b\ (x\hat{}n) = real\ n * log\ b\ x$
by (*simp add*: *ln-realpow log-def*)

lemma *log-of-power-eq*:
assumes $m = b\ \hat{}\ n\ b > 1$
shows $n = log\ b\ (real\ m)$
proof −
have $n = log\ b\ (b\ \hat{}\ n)$ **using** *assms*(*2*) **by** (*simp add*: *log-nat-power*)
also have $\ldots = log\ b\ m$ **using** *assms* **by** *simp*
finally show *?thesis* **.**
qed

lemma *log2-of-power-eq*: $m = 2\ \hat{}\ n \Longrightarrow n = log\ 2\ m$ **for** $m\ n$ :: *nat*
using *log-of-power-eq*[*of - 2*] **by** *simp*

lemma *log-base-change*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow log\ b\ x = log\ a\ x\ /\ log\ a\ b$
by (*simp add*: *log-def*)

lemma *log-base-pow*: $0 < a \Longrightarrow log\ (a\ \hat{}\ n)\ x = log\ a\ x\ /\ n$
by (*simp add*: *log-def ln-realpow*)

lemma *log-base-powr*: $a \neq 0 \Longrightarrow log\ (a\ powr\ b)\ x = log\ a\ x\ /\ b$
by (*simp add*: *log-def ln-powr*)

lemma *log-base-root*: $n > 0 \Longrightarrow log\ (root\ n\ b)\ x = n * (log\ b\ x)$
by (*simp add*: *log-def ln-root*)

lemma *ln-bound*: $0 < x \Longrightarrow ln\ x \leq x$ **for** x :: *real*
using *ln-le-minus-one* **by** *force*

lemma *powr-less-one*:
fixes *x*::*real*
assumes $1 < x\ y < 0$
shows $x\ powr\ y < 1$
using *assms less-log-iff* **by** *force*

lemma *powr-le-one-le*: $\bigwedge x\ y$::*real*. $0 < x \Longrightarrow x \leq 1 \Longrightarrow 1 \leq y \Longrightarrow x\ powr\ y \leq x$
by (*smt* (*verit*) *ln-gt-zero-imp-gt-one ln-le-cancel-iff ln-powr mult-le-cancel-right2*)

lemma *powr-mono*:
fixes x :: *real*
assumes $a \leq b$ **and** $1 \leq x$ **shows** $x\ powr\ a \leq x\ powr\ b$
using *assms less-eq-real-def* **by** *auto*

lemma *ge-one-powr-ge-zero*: $1 \leq x \Longrightarrow 0 \leq a \Longrightarrow 1 \leq x\ powr\ a$
for x :: *real*
using *powr-mono* **by** *fastforce*

lemma *powr-less-mono2*: $0 < a \Longrightarrow 0 \leq x \Longrightarrow x < y \Longrightarrow x\ powr\ a < y\ powr\ a$
for x :: *real*
by (*simp add*: *powr-def*)

lemma *powr-less-mono2-neg*: $a < 0 \Longrightarrow 0 < x \Longrightarrow x < y \Longrightarrow y\ powr\ a < x\ powr\ a$
for x :: *real*
by (*simp add*: *powr-def*)

lemma *powr-mono2*: $x\ powr\ a \leq y\ powr\ a$ **if** $0 \leq a\ 0 \leq x\ x \leq y$
for x :: *real*
using *less-eq-real-def powr-less-mono2 that* **by** *auto*

lemma *powr-less-cancel2*: $0 < a \Longrightarrow 0 < x \Longrightarrow 0 < y \Longrightarrow x\ powr\ a < y\ powr\ a \Longrightarrow x < y$
for $a\ x\ y$ ::*real*
by (*metis less-le not-less-iff-gr-or-eq powr-less-mono2*)

lemma *powr01-less-one*:
fixes *x*::*real*
assumes $0 < x\ x < 1$
shows $x\ powr\ a < 1 \longleftrightarrow a>0$
proof
show $x\ powr\ a < 1 \Longrightarrow a>0$

using *assms not-less-iff-gr-or-eq powr-less-mono2-neg* **by** *fastforce*
show $a>0 \Longrightarrow x\ powr\ a < 1$
by (*metis assms less-eq-real-def powr-less-mono2 powr-one-eq-one*)
qed

lemma *powr-le1*: $0 \leq a \Longrightarrow |x| \leq 1 \Longrightarrow x\ powr\ a \leq 1$
for x :: *real*
by (*smt* (*verit*, *best*) *powr-mono2 powr-one-eq-one uminus-powr-eq*)

lemma *powr-mono2'*:
fixes $a\ x\ y$ :: *real*
assumes $a \leq 0\ x > 0\ x \leq y$
shows $x\ powr\ a \geq y\ powr\ a$
proof −
from *assms* **have** $x\ powr - a \leq y\ powr - a$
by (*intro powr-mono2*) *simp-all*
with *assms* **show** *?thesis*
by (*auto simp*: *powr-minus field-simps*)
qed

lemma *powr-mono'*: $a \leq (b::real) \Longrightarrow x \geq 0 \Longrightarrow x \leq 1 \Longrightarrow x\ powr\ b \leq x\ powr\ a$
using *powr-mono*[*of* $-b$ $-a$ *inverse x*] **by** (*auto simp*: *powr-def ln-inverse ln-div field-split-simps*)

lemma *powr-mono-both*:
fixes x :: *real*
assumes $0 \leq a\ a \leq b\ 1 \leq x\ x \leq y$
shows $x\ powr\ a \leq y\ powr\ b$
by (*meson assms order.trans powr-mono powr-mono2 zero-le-one*)

lemma *powr-mono-both'*:
fixes x :: *real*
assumes $a \geq b\ b{\geq}0\ 0 < x\ x \leq y\ y \leq 1$
shows $x\ powr\ a \leq y\ powr\ b$
by (*meson assms nless-le order.trans powr-mono' powr-mono2*)

lemma *powr-less-mono'*:
assumes $(x::real) > 0\ x < 1\ a < b$
shows $x\ powr\ b < x\ powr\ a$
by (*metis assms log-powr-cancel order.strict-iff-order powr-mono'*)

lemma *powr-inj*: $0 < a \Longrightarrow a \neq 1 \Longrightarrow a\ powr\ x = a\ powr\ y \longleftrightarrow x = y$
for x :: *real*
by (*metis log-powr-cancel*)

lemma *powr-half-sqrt*: $0 \leq x \Longrightarrow x\ powr\ (1/2) = sqrt\ x$
by (*simp add*: *powr-def root-powr-inverse sqrt-def*)

lemma *powr-half-sqrt-powr*: $0 \leq x \Longrightarrow x\ powr\ (a/2) = sqrt(x\ powr\ a)$

by (*metis divide-inverse mult.left-neutral powr-ge-zero powr-half-sqrt powr-powr*)

lemma *square-powr-half* [*simp*]:
fixes *x*::*real* **shows** x^2 *powr* (*1*/*2*) = |*x*|
by (*simp add*: *powr-half-sqrt*)

lemma *ln-powr-bound*: *1* ≤ *x* ⟹ *0* < *a* ⟹ *ln x* ≤ (*x powr a*) / *a*
for *x* :: *real*
by (*metis exp-gt-zero linear ln-eq-zero-iff ln-exp ln-less-self ln-powr mult.commute mult-imp-le-div-pos not-less powr-gt-zero*)

lemma *ln-powr-bound2*:
fixes *x* :: *real*
assumes *1* < *x* **and** *0* < *a*
shows (*ln x*) *powr a* ≤ (*a powr a*) ∗ *x*
proof −
from *assms* **have** *ln x* ≤ (*x powr* (*1* / *a*)) / (*1* / *a*)
by (*metis less-eq-real-def ln-powr-bound zero-less-divide-1-iff*)
also have ... = *a* ∗ (*x powr* (*1* / *a*))
by *simp*
finally have (*ln x*) *powr a* ≤ (*a* ∗ (*x powr* (*1* / *a*))) *powr a*
by (*metis assms less-imp-le ln-gt-zero powr-mono2*)
also have ... = (*a powr a*) ∗ ((*x powr* (*1* / *a*)) *powr a*)
using *assms powr-mult* **by** *auto*
also have (*x powr* (*1* / *a*)) *powr a* = *x powr* ((*1* / *a*) ∗ *a*)
by (*rule powr-powr*)
also have ... = *x* **using** *assms*
by *auto*
finally show *?thesis* **.**
qed

lemma *tendsto-powr*:
fixes *a b* :: *real*
assumes *f*: (*f* ⟶ *a*) *F*
and *g*: (*g* ⟶ *b*) *F*
and *a*: *a* ≠ *0*
shows ((λ*x*. *f x powr g x*) ⟶ *a powr b*) *F*
unfolding *powr-def*
proof (*rule filterlim-If*)
show ((λ*x*. *0*) ⟶ (*if a* = *0 then 0 else exp* (*b* ∗ *ln a*))) (*inf F* (*principal* {*x*. *f x* = *0*}))
using *tendsto-imp-eventually-ne* [*OF f*] *a*
by (*simp add*: *filterlim-iff eventually-inf-principal frequently-def*)
from *f g a* **show** ((λ*x*. *exp* (*g x* ∗ *ln* (*f x*))) ⟶ (*if a* = *0 then 0 else exp* (*b* ∗ *ln a*)))
(*inf F* (*principal* {*x*. *f x* ≠ *0*}))
by (*auto intro*!: *tendsto-intros intro*: *tendsto-mono inf-le1*)
qed

lemma *tendsto-powr'*[*tendsto-intros*]:
fixes *a* :: *real*
assumes *f*: (*f* ⟶ *a*) *F*
and *g*: (*g* ⟶ *b*) *F*
and *a*: *a* ≠ *0* ∨ (*b* > *0* ∧ *eventually* (λ*x*. *f x* ≥ *0*) *F*)
shows ((λ*x*. *f x powr g x*) ⟶ *a powr b*) *F*
proof −
from *a* **consider** *a* ≠ *0* | *a* = *0 b* > *0 eventually* (λ*x*. *f x* ≥ *0*) *F*
by *auto*
then show *?thesis*
proof *cases*
case *1*
with *f g* **show** *?thesis* **by** (*rule tendsto-powr*)
next
case *2*
have ((λ*x*. *if f x* = *0 then 0 else exp* (*g x* ∗ *ln* (*f x*))) ⟶ *0*) *F*
proof (*intro filterlim-If*)
have *filterlim f* (*principal* {*0*<..}) (*inf F* (*principal* {*z*. *f z* ≠ *0*}))
using ‹*eventually* (λ*x*. *f x* ≥ *0*) *F*›
by (*auto simp*: *filterlim-iff eventually-inf-principal*
eventually-principal elim: *eventually-mono*)
moreover have *filterlim f* (*nhds a*) (*inf F* (*principal* {*z*. *f z* ≠ *0*}))
by (*rule tendsto-mono*[*OF - f*]) *simp-all*
ultimately have *f*: *filterlim f* (*at-right 0*) (*inf F* (*principal* {*x*. *f x* ≠ *0*}))
by (*simp add*: *at-within-def filterlim-inf* ‹*a* = *0*›)
have *g*: (*g* ⟶ *b*) (*inf F* (*principal* {*z*. *f z* ≠ *0*}))
by (*rule tendsto-mono*[*OF - g*]) *simp-all*
show ((λ*x*. *exp* (*g x* ∗ *ln* (*f x*))) ⟶ *0*) (*inf F* (*principal* {*x*. *f x* ≠ *0*}))
by (*rule filterlim-compose*[*OF exp-at-bot*] *filterlim-tendsto-pos-mult-at-bot*
filterlim-compose[*OF ln-at-0*] *f g* ‹*b* > *0*›)+
qed *simp-all*
with ‹*a* = *0*› **show** *?thesis*
by (*simp add*: *powr-def*)
qed
qed

lemma *continuous-powr*:
assumes *continuous F f*
and *continuous F g*
and *f* (*Lim F* (λ*x*. *x*)) ≠ *0*
shows *continuous F* (λ*x*. (*f x*) *powr* (*g x* :: *real*))
using *assms* **unfolding** *continuous-def* **by** (*rule tendsto-powr*)

lemma *continuous-at-within-powr*[*continuous-intros*]:
fixes *f g* :: - ⇒ *real*
assumes *continuous* (*at a within s*) *f*
and *continuous* (*at a within s*) *g*
and *f a* ≠ *0*
shows *continuous* (*at a within s*) (λ*x*. (*f x*) *powr* (*g x*))

using *assms* **unfolding** *continuous-within* **by** (*rule tendsto-powr*)

lemma *continuous-on-powr*[*continuous-intros*]:
fixes *f g* :: - ⇒ *real*
assumes *continuous-on s f continuous-on s g* **and** ∀ *x*∈*s*. *f x* ≠ *0*
shows *continuous-on s* (λ*x*. (*f x*) *powr* (*g x*))
using *assms* **unfolding** *continuous-on-def* **by** (*fast intro*: *tendsto-powr*)

lemma *tendsto-powr2*:
fixes *a* :: *real*
assumes *f*: (*f* ⟶ *a*) *F*
and *g*: (*g* ⟶ *b*) *F*
and ∀ *F x in F*. *0* ≤ *f x*
and *b*: *0* < *b*
shows ((λ*x*. *f x powr g x*) ⟶ *a powr b*) *F*
using *tendsto-powr'*[*of f a F g b*] *assms* **by** *auto*

lemma *has-derivative-powr*[*derivative-intros*]:
assumes *g*[*derivative-intros*]: (*g has-derivative g'*) (*at x within X*)
and *f*[*derivative-intros*]:(*f has-derivative f'*) (*at x within X*)
assumes *pos*: *0* < *g x* **and** *x* ∈ *X*
shows ((λ*x*. *g x powr f x*::*real*) *has-derivative* (λ*h*. (*g x powr f x*) ∗ (*f' h* ∗ *ln* (*g x*) + *g' h* ∗ *f x* / *g x*))) (*at x within X*)
proof −
have ∀ *F x in at x within X*. *g x* > *0*
by (*rule order-tendstoD*[*OF* - *pos*])
(*rule has-derivative-continuous*[*OF g*, *unfolded continuous-within*])
then obtain *d* **where** *d* > *0* **and** *pos'*: ⋀*x'*. *x'* ∈ *X* ⟹ *dist x' x* < *d* ⟹ *0* < *g x'*
using *pos* **unfolding** *eventually-at* **by** *force*
have ((λ*x*. *exp* (*f x* ∗ *ln* (*g x*))) *has-derivative*
(λ*h*. (*g x powr f x*) ∗ (*f' h* ∗ *ln* (*g x*) + *g' h* ∗ *f x* / *g x*))) (*at x within X*)
using *pos*
by (*auto intro*!: *derivative-eq-intros simp*: *field-split-simps powr-def*)
then show *?thesis*
by (*rule has-derivative-transform-within*[*OF* - ‹*d* > *0*› ‹*x* ∈ *X*›]) (*auto simp*: *powr-def dest*: *pos'*)
qed

lemma *has-derivative-const-powr* [*derivative-intros*]:
fixes *a*::*real*
assumes ⋀*x*. (*f has-derivative f'*) (*at x*)
shows ((λ*x*. *a powr* (*f x*)) *has-derivative* (λ*y*. *f' y* ∗ *ln a* ∗ *a powr* (*f x*))) (*at x*)
using *assms*
apply (*simp add*: *powr-def*)
using *DERIV-compose-FDERIV DERIV-exp has-derivative-mult-left* **by** *blast*

lemma *has-real-derivative-const-powr* [*derivative-intros*]:
fixes *a*::*real*

assumes $\bigwedge x.$ (*f has-real-derivative f′ x*) (*at x*)
shows ((λ*x. a powr* (*f x*)) *has-real-derivative* (*f′ x* ∗ *ln a* ∗ *a powr* (*f x*))) (*at x*)
using *assms*
apply (*simp add*: *powr-def*)
apply (*rule assms impI derivative-eq-intros refl* | *simp*)+
done

lemma *DERIV-powr*:
fixes *r* :: *real*
assumes *g*: *DERIV g x* :> *m*
and *pos*: *g x* > *0*
and *f*: *DERIV f x* :> *r*
shows *DERIV* (λ*x. g x powr f x*) *x* :> (*g x powr f x*) ∗ (*r* ∗ *ln* (*g x*) + *m* ∗ *f x* / *g x*)
using *assms*
by (*auto intro*!: *derivative-eq-intros ext simp*: *has-field-derivative-def algebra-simps*)

lemma *DERIV-fun-powr*:
fixes *r* :: *real*
assumes *g*: *DERIV g x* :> *m*
and *pos*: *g x* > *0*
shows *DERIV* (λ*x.* (*g x*) *powr r*) *x* :> *r* ∗ (*g x*) *powr* (*r* − *of-nat 1*) ∗ *m*
using *DERIV-powr*[*OF g pos DERIV-const*, *of r*] *pos*
by (*simp add*: *powr-diff field-simps*)

lemma *has-real-derivative-powr*:
assumes *z* > *0*
shows ((λ*z. z powr r*) *has-real-derivative r* ∗ *z powr* (*r* − *1*)) (*at z*)
proof (*subst DERIV-cong-ev*[*OF refl - refl*])
from *assms* **have** *eventually* (λ*z. z* ≠ *0*) (*nhds z*)
by (*intro t1-space-nhds*) *auto*
then show *eventually* (λ*z. z powr r* = *exp* (*r* ∗ *ln z*)) (*nhds z*)
unfolding *powr-def* **by** *eventually-elim simp*
from *assms* **show** ((λ*z. exp* (*r* ∗ *ln z*)) *has-real-derivative r* ∗ *z powr* (*r* − *1*)) (*at z*)
by (*auto intro*!: *derivative-eq-intros simp*: *powr-def field-simps exp-diff*)
qed

declare *has-real-derivative-powr*[*THEN DERIV-chain2*, *derivative-intros*]

A more general version, by Johannes Hölzl

lemma *has-real-derivative-powr′*:
fixes *f g* :: *real* ⇒ *real*
assumes (*f has-real-derivative f′*) (*at x*)
assumes (*g has-real-derivative g′*) (*at x*)
assumes *f x* > *0*
defines *h* ≡ λ*x. f x powr g x* ∗ (*g′* ∗ *ln* (*f x*) + *f′* ∗ *g x* / *f x*)
shows ((λ*x. f x powr g x*) *has-real-derivative h x*) (*at x*)
proof (*subst DERIV-cong-ev*[*OF refl - refl*])

```
  from assms have isCont f x
    by (simp add: DERIV-continuous)
  hence f -x→ f x by (simp add: continuous-at)
  with ‹f x > 0› have eventually (λx. f x > 0) (nhds x)
    by (auto simp: tendsto-at-iff-tendsto-nhds dest: order-tendstoD)
  thus eventually (λx. f x powr g x = exp (g x * ln (f x))) (nhds x)
    by eventually-elim (simp add: powr-def)
next
  from assms show ((λx. exp (g x * ln (f x))) has-real-derivative h x) (at x)
    by (auto intro!: derivative-eq-intros simp: h-def powr-def)
qed

lemma tendsto-zero-powrI:
  assumes (f ⟶ (0::real)) F (g ⟶ b) F ∀⇩F x in F. 0 ≤ f x 0 < b
  shows ((λx. f x powr g x) ⟶ 0) F
  using tendsto-powr2[OF assms] by simp

lemma continuous-on-powr':
  fixes f g :: - ⇒ real
  assumes continuous-on s f continuous-on s g
    and ∀x∈s. f x ≥ 0 ∧ (f x = 0 ⟶ g x > 0)
  shows continuous-on s (λx. (f x) powr (g x))
  unfolding continuous-on-def
proof
  fix x
  assume x: x ∈ s
  from assms x show ((λx. f x powr g x) ⟶ f x powr g x) (at x within s)
  proof (cases f x = 0)
    case True
    from assms(3) have eventually (λx. f x ≥ 0) (at x within s)
      by (auto simp: at-within-def eventually-inf-principal)
    with True x assms show ?thesis
      by (auto intro!: tendsto-zero-powrI[of f - g g x] simp: continuous-on-def)
  next
    case False
    with assms x show ?thesis
      by (auto intro!: tendsto-powr' simp: continuous-on-def)
  qed
qed

lemma tendsto-neg-powr:
  assumes s < 0
    and f: LIM x F. f x :> at-top
  shows ((λx. f x powr s) ⟶ (0::real)) F
proof -
  have ((λx. exp (s * ln (f x))) ⟶ (0::real)) F (is ?X)
    by (auto intro!: filterlim-compose[OF exp-at-bot] filterlim-compose[OF ln-at-top]
        filterlim-tendsto-neg-mult-at-bot assms)
  also have ?X ⟷ ((λx. f x powr s) ⟶ (0::real)) F
```

using *f filterlim-at-top-dense*[*of f F*]
by (*intro filterlim-cong*[*OF refl refl*]) (*auto simp*: *neq-iff powr-def elim*: *eventually-mono*)
finally show *?thesis* **.**
qed

lemma *tendsto-exp-limit-at-right*: ((λ*y*. (*1* + *x* ∗ *y*) *powr* (*1* / *y*)) ⟶ *exp x*) (*at-right 0*)
for *x* :: *real*
proof (*cases x* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
have ((λ*y*. *ln* (*1* + *x* ∗ *y*)::*real*) *has-real-derivative 1* ∗ *x*) (*at 0*)
by (*auto intro*!: *derivative-eq-intros*)
then have ((λ*y*. *ln* (*1* + *x* ∗ *y*) / *y*) ⟶ *x*) (*at 0*)
by (*auto simp*: *has-field-derivative-def field-has-derivative-at*)
then have ∗: ((λ*y*. *exp* (*ln* (*1* + *x* ∗ *y*) / *y*)) ⟶ *exp x*) (*at 0*)
by (*rule tendsto-intros*)
then show *?thesis*
proof (*rule filterlim-mono-eventually*)
show *eventually* (λ*xa*. *exp* (*ln* (*1* + *x* ∗ *xa*) / *xa*) = (*1* + *x* ∗ *xa*) *powr* (*1* / *xa*)) (*at-right 0*)
unfolding *eventually-at-right*[*OF zero-less-one*]
using *False*
by (*intro exI*[*of - 1* / |*x*|]) (*auto simp*: *field-simps powr-def abs-if add-nonneg-eq-0-iff*)
qed (*simp-all add*: *at-eq-sup-left-right*)
qed

lemma *tendsto-exp-limit-at-top*: ((λ*y*. (*1* + *x* / *y*) *powr y*) ⟶ *exp x*) *at-top*
for *x* :: *real*
by (*simp add*: *filterlim-at-top-to-right inverse-eq-divide tendsto-exp-limit-at-right*)

lemma *tendsto-exp-limit-sequentially*: (λ*n*. (*1* + *x* / *n*) ^ *n*) ⟶ *exp x*
for *x* :: *real*
proof (*rule filterlim-mono-eventually*)
from *reals-Archimedean2* [*of* |*x*|] **obtain** *n* :: *nat* **where** ∗: *real n* > |*x*| **..**
then have *eventually* (λ*n* :: *nat*. *0* < *1* + *x* / *real n*) *at-top*
by (*intro eventually-sequentiallyI* [*of n*]) (*auto simp*: *field-split-simps*)
then show *eventually* (λ*n*. (*1* + *x* / *n*) *powr n* = (*1* + *x* / *n*) ^ *n*) *at-top*
by (*rule eventually-mono*) (*erule powr-realpow*)
show (λ*n*. (*1* + *x* / *real n*) *powr real n*) ⟶ *exp x*
by (*rule filterlim-compose* [*OF tendsto-exp-limit-at-top filterlim-real-sequentially*])
qed *auto*

112.9 Sine and Cosine

definition *sin-coeff* :: *nat* ⇒ *real*

where *sin-coeff* = (λ*n*. *if even n then 0 else* (− *1*) ^ ((*n* − *Suc 0*) *div 2*) / (*fact n*))

definition *cos-coeff* :: *nat* ⇒ *real*
where *cos-coeff* = (λ*n*. *if even n then* ((− *1*) ^ (*n div 2*)) / (*fact n*) *else 0*)

definition *sin* :: *'a* ⇒ *'a*::{*real-normed-algebra-1*,*banach*}
where *sin* = (λ*x*. ∑ *n*. *sin-coeff n* ∗$_R$ *x*^*n*)

definition *cos* :: *'a* ⇒ *'a*::{*real-normed-algebra-1*,*banach*}
where *cos* = (λ*x*. ∑ *n*. *cos-coeff n* ∗$_R$ *x*^*n*)

lemma *sin-coeff-0* [*simp*]: *sin-coeff 0* = *0*
unfolding *sin-coeff-def* **by** *simp*

lemma *cos-coeff-0* [*simp*]: *cos-coeff 0* = *1*
unfolding *cos-coeff-def* **by** *simp*

lemma *sin-coeff-Suc*: *sin-coeff* (*Suc n*) = *cos-coeff n* / *real* (*Suc n*)
unfolding *cos-coeff-def sin-coeff-def*
by (*simp del*: *mult-Suc*)

lemma *cos-coeff-Suc*: *cos-coeff* (*Suc n*) = − *sin-coeff n* / *real* (*Suc n*)
unfolding *cos-coeff-def sin-coeff-def*
by (*simp del*: *mult-Suc*) (*auto elim*: *oddE*)

lemma *summable-norm-sin*: *summable* (λ*n*. *norm* (*sin-coeff n* ∗$_R$ *x*^*n*))
for *x* :: *'a*::{*real-normed-algebra-1*,*banach*}
proof (*rule summable-comparison-test* [*OF - summable-norm-exp*])
show ∃ *N*. ∀ *n*≥*N*. *norm* (*norm* (*sin-coeff n* ∗$_R$ *x* ^ *n*)) ≤ *norm* (*x* ^ *n* /$_R$ *fact n*)
unfolding *sin-coeff-def*
by (*auto simp*: *divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)
qed

lemma *summable-norm-cos*: *summable* (λ*n*. *norm* (*cos-coeff n* ∗$_R$ *x*^*n*))
for *x* :: *'a*::{*real-normed-algebra-1*,*banach*}
proof (*rule summable-comparison-test* [*OF - summable-norm-exp*])
show ∃ *N*. ∀ *n*≥*N*. *norm* (*norm* (*cos-coeff n* ∗$_R$ *x* ^ *n*)) ≤ *norm* (*x* ^ *n* /$_R$ *fact n*)
unfolding *cos-coeff-def*
by (*auto simp*: *divide-inverse abs-mult power-abs* [*symmetric*] *zero-le-mult-iff*)
qed

lemma *sin-converges*: (λ*n*. *sin-coeff n* ∗$_R$ *x*^*n*) *sums sin x*
unfolding *sin-def*
by (*metis* (*full-types*) *summable-norm-cancel summable-norm-sin summable-sums*)

lemma *cos-converges*: $(\lambda n.\ \textit{cos-coeff}\ n *_R x\hat{}n)\ \textit{sums}\ \textit{cos}\ x$
unfolding *cos-def*
by (*metis* (*full-types*) *summable-norm-cancel summable-norm-cos summable-sums*)

lemma *sin-of-real*: $\textit{sin}\ (\textit{of-real}\ x) = \textit{of-real}\ (\textit{sin}\ x)$
for $x :: \textit{real}$
proof −
have $(\lambda n.\ \textit{of-real}\ (\textit{sin-coeff}\ n *_R\ x\hat{}n)) = (\lambda n.\ \textit{sin-coeff}\ n *_R\ (\textit{of-real}\ x)\hat{}n)$
proof
show $\textit{of-real}\ (\textit{sin-coeff}\ n *_R\ x\hat{}n) = \textit{sin-coeff}\ n *_R \textit{of-real}\ x\hat{}n$ **for** n
by (*simp add*: *scaleR-conv-of-real*)
qed
also have ... $\textit{sums}\ (\textit{sin}\ (\textit{of-real}\ x))$
by (*rule sin-converges*)
finally have $(\lambda n.\ \textit{of-real}\ (\textit{sin-coeff}\ n *_R x\hat{}n))\ \textit{sums}\ (\textit{sin}\ (\textit{of-real}\ x))$ **.**
then show *?thesis*
using *sums-unique2 sums-of-real* [*OF sin-converges*] **by** *blast*
qed

corollary *sin-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow \textit{sin}\ z \in \mathbb{R}$
by (*metis Reals-cases Reals-of-real sin-of-real*)

lemma *cos-of-real*: $\textit{cos}\ (\textit{of-real}\ x) = \textit{of-real}\ (\textit{cos}\ x)$
for $x :: \textit{real}$
proof −
have $(\lambda n.\ \textit{of-real}\ (\textit{cos-coeff}\ n *_R\ x\hat{}n)) = (\lambda n.\ \textit{cos-coeff}\ n *_R\ (\textit{of-real}\ x)\hat{}n)$
proof
show $\textit{of-real}\ (\textit{cos-coeff}\ n *_R\ x\hat{}n) = \textit{cos-coeff}\ n *_R \textit{of-real}\ x\hat{}n$ **for** n
by (*simp add*: *scaleR-conv-of-real*)
qed
also have ... $\textit{sums}\ (\textit{cos}\ (\textit{of-real}\ x))$
by (*rule cos-converges*)
finally have $(\lambda n.\ \textit{of-real}\ (\textit{cos-coeff}\ n *_R x\hat{}n))\ \textit{sums}\ (\textit{cos}\ (\textit{of-real}\ x))$ **.**
then show *?thesis*
using *sums-unique2 sums-of-real* [*OF cos-converges*]
by *blast*
qed

corollary *cos-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow \textit{cos}\ z \in \mathbb{R}$
by (*metis Reals-cases Reals-of-real cos-of-real*)

lemma *diffs-sin-coeff*: $\textit{diffs}\ \textit{sin-coeff} = \textit{cos-coeff}$
by (*simp add*: *diffs-def sin-coeff-Suc del*: *of-nat-Suc*)

lemma *diffs-cos-coeff*: $\textit{diffs}\ \textit{cos-coeff} = (\lambda n.\ -\ \textit{sin-coeff}\ n)$
by (*simp add*: *diffs-def cos-coeff-Suc del*: *of-nat-Suc*)

lemma *sin-int-times-real*: $\textit{sin}\ (\textit{of-int}\ m * \textit{of-real}\ x) = \textit{of-real}\ (\textit{sin}\ (\textit{of-int}\ m * x))$
by (*metis sin-of-real of-real-mult of-real-of-int-eq*)

lemma *cos-int-times-real*: *cos* (*of-int m* ∗ *of-real x*) = *of-real* (*cos* (*of-int m* ∗ *x*))
by (*metis cos-of-real of-real-mult of-real-of-int-eq*)

Now at last we can get the derivatives of exp, sin and cos.

lemma *DERIV-sin* [*simp*]: *DERIV sin x* :> *cos x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
unfolding *sin-def cos-def scaleR-conv-of-real*
apply (*rule DERIV-cong*)
apply (*rule termdiffs* [**where** *K*=*of-real* (*norm x*) + *1* :: *'a*])
apply (*simp-all add*: *norm-less-p1 diffs-of-real diffs-sin-coeff diffs-cos-coeff*
summable-minus-iff scaleR-conv-of-real [*symmetric*]
summable-norm-sin [*THEN summable-norm-cancel*]
summable-norm-cos [*THEN summable-norm-cancel*])
done

declare *DERIV-sin*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-sin*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemmas *has-derivative-sin*[*derivative-intros*] = *DERIV-sin*[*THEN DERIV-compose-FDERIV*]

lemma *DERIV-cos* [*simp*]: *DERIV cos x* :> − *sin x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
unfolding *sin-def cos-def scaleR-conv-of-real*
apply (*rule DERIV-cong*)
apply (*rule termdiffs* [**where** *K*=*of-real* (*norm x*) + *1* :: *'a*])
apply (*simp-all add*: *norm-less-p1 diffs-of-real diffs-minus suminf-minus*
diffs-sin-coeff diffs-cos-coeff
summable-minus-iff scaleR-conv-of-real [*symmetric*]
summable-norm-sin [*THEN summable-norm-cancel*]
summable-norm-cos [*THEN summable-norm-cancel*])
done

declare *DERIV-cos*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-cos*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemmas *has-derivative-cos*[*derivative-intros*] = *DERIV-cos*[*THEN DERIV-compose-FDERIV*]

lemma *isCont-sin*: *isCont sin x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
by (*rule DERIV-sin* [*THEN DERIV-isCont*])

lemma *continuous-on-sin-real*: *continuous-on* {*a*..*b*} *sin* **for** *a*::*real*
using *continuous-at-imp-continuous-on isCont-sin* **by** *blast*

lemma *isCont-cos*: *isCont cos x*
for *x* :: *'a*::{*real-normed-field*,*banach*}

by (*rule DERIV-cos* [*THEN DERIV-isCont*])

lemma *continuous-on-cos-real*: *continuous-on* {*a*..*b*} *cos* **for** *a*::*real*
using *continuous-at-imp-continuous-on isCont-cos* **by** *blast*

context
fixes *f* :: *'a*::*t2-space* ⇒ *'b*::{*real-normed-field*,*banach*}
begin

lemma *isCont-sin'* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *sin* (*f x*)) *a*
by (*rule isCont-o2* [*OF* - *isCont-sin*])

lemma *isCont-cos'* [*simp*]: *isCont f a* ⟹ *isCont* (λ*x*. *cos* (*f x*)) *a*
by (*rule isCont-o2* [*OF* - *isCont-cos*])

lemma *tendsto-sin* [*tendsto-intros*]: (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *sin* (*f x*)) ⟶ *sin a*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-sin*])

lemma *tendsto-cos* [*tendsto-intros*]: (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *cos* (*f x*)) ⟶ *cos a*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-cos*])

lemma *continuous-sin* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *sin* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-sin*)

lemma *continuous-on-sin* [*continuous-intros*]: *continuous-on s f* ⟹ *continuous-on s* (λ*x*. *sin* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-sin*)

lemma *continuous-cos* [*continuous-intros*]: *continuous F f* ⟹ *continuous F* (λ*x*. *cos* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-cos*)

lemma *continuous-on-cos* [*continuous-intros*]: *continuous-on s f* ⟹ *continuous-on s* (λ*x*. *cos* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-cos*)

end

lemma *continuous-within-sin*: *continuous* (*at z within s*) *sin*
for *z* :: *'a*::{*real-normed-field*,*banach*}
by (*simp add*: *continuous-within tendsto-sin*)

lemma *continuous-within-cos*: *continuous* (*at z within s*) *cos*
for *z* :: *'a*::{*real-normed-field*,*banach*}
by (*simp add*: *continuous-within tendsto-cos*)

112.10 Properties of Sine and Cosine

lemma *sin-zero* [*simp*]: *sin 0 = 0*
by (*simp add*: *sin-def sin-coeff-def scaleR-conv-of-real*)

lemma *cos-zero* [*simp*]: *cos 0 = 1*
by (*simp add*: *cos-def cos-coeff-def scaleR-conv-of-real*)

lemma *DERIV-fun-sin*: *DERIV g x :> m ⟹ DERIV* (*λx. sin* (*g x*)) *x :> cos* (*g x*) ∗ *m*
by (*fact derivative-intros*)

lemma *DERIV-fun-cos*: *DERIV g x :> m ⟹ DERIV* (*λx. cos*(*g x*)) *x :> − sin* (*g x*) ∗ *m*
by (*fact derivative-intros*)

112.11 Deriving the Addition Formulas

The product of two cosine series.

lemma *cos-x-cos-y*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
shows
(*λp*. ∑ *n*≤*p*.
if even p ∧ even n
then ((*−1*) ^ (*p div 2*) ∗ (*p choose n*) / (*fact p*)) ∗$_R$ (*x*^*n* ∗ *y*^(*p−n*)) *else 0*)
sums (*cos x* ∗ *cos y*)
proof −
have (*cos-coeff n* ∗ *cos-coeff* (*p* − *n*)) ∗$_R$ (*x*^*n* ∗ *y*^(*p* − *n*)) =
(*if even p ∧ even n then* ((*−1*) ^ (*p div 2*) ∗ (*p choose n*) / (*fact p*)) ∗$_R$ (*x*^*n*) ∗ *y*^(*p* − *n*)
else 0)
if *n* ≤ *p* **for** *n p* :: *nat*
proof −
from *that* **have** ∗: *even n ⟹ even p ⟹*
(*−1*) ^ (*n div 2*) ∗ (*−1*) ^ ((*p* − *n*) *div 2*) = (*−1* :: *real*) ^ (*p div 2*)
by (*metis div-add power-add le-add-diff-inverse odd-add*)
with *that* **show** *?thesis*
by (*auto simp*: *algebra-simps cos-coeff-def binomial-fact*)
qed
then have (*λp*. ∑ *n*≤*p*. *if even p ∧ even n*
then ((*−1*) ^ (*p div 2*) ∗ (*p choose n*) / (*fact p*)) ∗$_R$ (*x*^*n*) ∗ *y*^(*p−n*) *else 0*) =
(*λp*. ∑ *n*≤*p*. (*cos-coeff n* ∗ *cos-coeff* (*p* − *n*)) ∗$_R$ (*x*^*n* ∗ *y*^(*p−n*)))
by *simp*
also have ... = (*λp*. ∑ *n*≤*p*. (*cos-coeff n* ∗$_R$ *x*^*n*) ∗ (*cos-coeff* (*p* − *n*) ∗$_R$ *y*^(*p−n*)))
by (*simp add*: *algebra-simps*)
also have ... *sums* (*cos x* ∗ *cos y*)

```
    using summable-norm-cos
    by (auto simp: cos-def scaleR-conv-of-real intro!: Cauchy-product-sums)
  finally show ?thesis .
qed
```

The product of two sine series.

```
lemma sin-x-sin-y:
  fixes x :: 'a::{real-normed-field,banach}
  shows
    (λp. ∑ n≤p.
        if even p ∧ odd n
        then − ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
        else 0)
      sums (sin x * sin y)
proof −
  have (sin-coeff n * sin-coeff (p − n)) *R (x^n * y^(p−n)) =
    (if even p ∧ odd n
     then −((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
     else 0)
    if n ≤ p for n p :: nat
  proof −
    have (−1) ^ ((n − Suc 0) div 2) * (−1) ^ ((p − Suc n) div 2) = − ((−1 ::
real) ^ (p div 2))
      if np: odd n even p
    proof −
      have p > 0
        using ‹n ≤ p› neq0-conv that(1) by blast
      then have §: (− 1::real) ^ (p div 2 − Suc 0) = − ((− 1) ^ (p div 2))
        using ‹even p› by (auto simp add: dvd-def power-eq-if)
      from ‹n ≤ p› np have ∗: n − Suc 0 + (p − Suc n) = p − Suc (Suc 0)
        by (force simp: le-Suc-eq split: nat-diff-split)

      have (p − Suc (Suc 0)) div 2 = p div 2 − Suc 0
        by simp
      with ‹n ≤ p› np  § ∗ show ?thesis
        by (simp add: flip: div-add power-add)
    qed
    then show ?thesis
      using ‹n≤p› by (auto simp: algebra-simps sin-coeff-def binomial-fact)
  qed
  then have (λp. ∑ n≤p. if even p ∧ odd n
            then − ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
else 0) =
            (λp. ∑ n≤p. (sin-coeff n * sin-coeff (p − n)) *R (x^n * y^(p−n)))
    by simp
  also have ... = (λp. ∑ n≤p. (sin-coeff n *R x^n) * (sin-coeff (p − n) *R
y^(p−n)))
    by (simp add: algebra-simps)
  also have ... sums (sin x * sin y)
```

```
    using summable-norm-sin
    by (auto simp: sin-def scaleR-conv-of-real intro!: Cauchy-product-sums)
  finally show ?thesis .
qed

lemma sums-cos-x-plus-y:
  fixes x :: 'a::{real-normed-field,banach}
  shows
    (λp. ∑ n≤p.
        if even p
        then ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
        else 0)
      sums cos (x + y)
proof −
  have
    (∑ n≤p.
     if even p then ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
      else 0) = cos-coeff p *R ((x + y) ^ p)
    for p :: nat
  proof −
    have
      (∑ n≤p. if even p then ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n)
* y^(p−n) else 0) =
       (if even p then ∑ n≤p. ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n)
* y^(p−n) else 0)
      by simp
    also have ... =
       (if even p
        then of-real ((−1) ^ (p div 2) / (fact p)) * (∑ n≤p. (p choose n) *R (x^n)
* y^(p−n))
        else 0)
    by (auto simp: sum-distrib-left field-simps scaleR-conv-of-real nonzero-of-real-divide)
    also have ... = cos-coeff p *R ((x + y) ^ p)
    by (simp add: cos-coeff-def binomial-ring [of x y]  scaleR-conv-of-real atLeast0At-
Most)
    finally show ?thesis .
  qed
  then have
    (λp. ∑ n≤p.
        if even p
        then ((−1) ^ (p div 2) * (p choose n) / (fact p)) *R (x^n) * y^(p−n)
        else 0) = (λp. cos-coeff p *R ((x+y)^p))
    by simp
   also have ... sums cos (x + y)
    by (rule cos-converges)
   finally show ?thesis .
qed

theorem cos-add:
```

fixes $x :: 'a::\{real\text{-}normed\text{-}field,banach\}$
shows $cos\ (x + y) = cos\ x * cos\ y - sin\ x * sin\ y$
proof −
have
$(if\ even\ p \wedge even\ n$
$then\ ((-\ 1)\ \hat{}\ (p\ div\ 2) * int\ (p\ choose\ n)\ /\ (fact\ p))\ *_R\ (x\hat{}n) * y\hat{}(p-n)$
$else\ 0) -$
$(if\ even\ p \wedge odd\ n$
$then\ -\ ((-\ 1)\ \hat{}\ (p\ div\ 2) * int\ (p\ choose\ n)\ /\ (fact\ p))\ *_R\ (x\hat{}n) * y\hat{}(p-n)$
$else\ 0) =$
$(if\ even\ p\ then\ ((-1)\ \hat{}\ (p\ div\ 2) * (p\ choose\ n)\ /\ (fact\ p))\ *_R\ (x\hat{}n) * y\hat{}(p-n)$
$else\ 0)$
if $n \leq p$ **for** $n\ p :: nat$
by *simp*
then have
$(\lambda p.\ \sum n{\leq}p.\ (if\ even\ p\ then\ ((-1)\ \hat{}\ (p\ div\ 2) * (p\ choose\ n)\ /\ (fact\ p))\ *_R$
$(x\hat{}n) * y\hat{}(p-n)\ else\ 0))$
$sums\ (cos\ x * cos\ y - sin\ x * sin\ y)$
using *sums-diff* [*OF cos-x-cos-y* [*of x y*] *sin-x-sin-y* [*of x y*]]
by (*simp add*: *sum-subtractf* [*symmetric*])
then show *?thesis*
by (*blast intro*: *sums-cos-x-plus-y sums-unique2*)
qed

lemma *sin-minus-converges*: $(\lambda n.\ -\ (sin\text{-}coeff\ n\ *_R\ (-x)\hat{}n))\ sums\ sin\ x$
proof −
have [*simp*]: $\bigwedge n.\ -\ (sin\text{-}coeff\ n\ *_R\ (-x)\hat{}n) = (sin\text{-}coeff\ n\ *_R\ x\hat{}n)$
by (*auto simp*: *sin-coeff-def elim*!: *oddE*)
show *?thesis*
by (*simp add*: *sin-def summable-norm-sin* [*THEN summable-norm-cancel*,
THEN summable-sums])
qed

lemma *sin-minus* [*simp*]: $sin\ (-\ x) = -\ sin\ x$
for $x :: 'a::\{real\text{-}normed\text{-}algebra\text{-}1,banach\}$
using *sin-minus-converges* [*of x*]
by (*auto simp*: *sin-def summable-norm-sin* [*THEN summable-norm-cancel*]
suminf-minus sums-iff equation-minus-iff)

lemma *cos-minus-converges*: $(\lambda n.\ (cos\text{-}coeff\ n\ *_R\ (-x)\hat{}n))\ sums\ cos\ x$
proof −
have [*simp*]: $\bigwedge n.\ (cos\text{-}coeff\ n\ *_R\ (-x)\hat{}n) = (cos\text{-}coeff\ n\ *_R\ x\hat{}n)$
by (*auto simp*: *Transcendental.cos-coeff-def elim*!: *evenE*)
show *?thesis*
by (*simp add*: *cos-def summable-norm-cos* [*THEN summable-norm-cancel*,
THEN summable-sums])
qed

lemma *cos-minus* [*simp*]: $cos\ (-x) = cos\ x$

for $x :: 'a$::{*real-normed-algebra-1*,*banach*}
using *cos-minus-converges* [*of* x] **by** (*metis cos-def sums-unique*)

lemma *cos-abs-real* [*simp*]: *cos* $|x :: real| = cos\ x$
by (*simp add*: *abs-if*)

lemma *sin-cos-squared-add* [*simp*]: $(sin\ x)^2 + (cos\ x)^2 = 1$
for $x :: 'a$::{*real-normed-field*,*banach*}
using *cos-add* [*of* $x\ -x$]
by (*simp add*: *power2-eq-square algebra-simps*)

lemma *sin-cos-squared-add2* [*simp*]: $(cos\ x)^2 + (sin\ x)^2 = 1$
for $x :: 'a$::{*real-normed-field*,*banach*}
by (*subst add.commute*, *rule sin-cos-squared-add*)

lemma *sin-cos-squared-add3* [*simp*]: $cos\ x * cos\ x + sin\ x * sin\ x = 1$
for $x :: 'a$::{*real-normed-field*,*banach*}
using *sin-cos-squared-add2* [*unfolded power2-eq-square*] **.**

lemma *sin-squared-eq*: $(sin\ x)^2 = 1 - (cos\ x)^2$
for $x :: 'a$::{*real-normed-field*,*banach*}
unfolding *eq-diff-eq* **by** (*rule sin-cos-squared-add*)

lemma *cos-squared-eq*: $(cos\ x)^2 = 1 - (sin\ x)^2$
for $x :: 'a$::{*real-normed-field*,*banach*}
unfolding *eq-diff-eq* **by** (*rule sin-cos-squared-add2*)

lemma *abs-sin-le-one* [*simp*]: $|sin\ x| \leq 1$
for $x :: real$
by (*rule power2-le-imp-le*) (*simp-all add*: *sin-squared-eq*)

lemma *sin-ge-minus-one* [*simp*]: $-\ 1 \leq sin\ x$
for $x :: real$
using *abs-sin-le-one* [*of* x] **by** (*simp add*: *abs-le-iff*)

lemma *sin-le-one* [*simp*]: $sin\ x \leq 1$
for $x :: real$
using *abs-sin-le-one* [*of* x] **by** (*simp add*: *abs-le-iff*)

lemma *abs-cos-le-one* [*simp*]: $|cos\ x| \leq 1$
for $x :: real$
by (*rule power2-le-imp-le*) (*simp-all add*: *cos-squared-eq*)

lemma *cos-ge-minus-one* [*simp*]: $-\ 1 \leq cos\ x$
for $x :: real$
using *abs-cos-le-one* [*of* x] **by** (*simp add*: *abs-le-iff*)

lemma *cos-le-one* [*simp*]: $cos\ x \leq 1$
for $x :: real$

using *abs-cos-le-one* [*of x*] **by** (*simp add*: *abs-le-iff*)

lemma *cos-diff*: *cos* (*x* − *y*) = *cos x* ∗ *cos y* + *sin x* ∗ *sin y*
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *cos-add* [*of x* − *y*] **by** *simp*

lemma *cos-double*: *cos*(*2*∗*x*) = (*cos x*)^2 − (*sin x*)^2
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *cos-add* [**where** *x*=*x* **and** *y*=*x*] **by** (*simp add*: *power2-eq-square*)

lemma *sin-cos-le1*: |*sin x* ∗ *sin y* + *cos x* ∗ *cos y*| ≤ *1*
for *x* :: *real*
using *cos-diff* [*of x y*] **by** (*metis abs-cos-le-one add.commute*)

lemma *DERIV-fun-pow*: *DERIV g x* :> *m* ⟹ *DERIV* (λ*x*. (*g x*) ^ *n*) *x* :> *real n* ∗ (*g x*) ^ (*n* − *1*) ∗ *m*
by (*auto intro*!: *derivative-eq-intros simp*:)

lemma *DERIV-fun-exp*: *DERIV g x* :> *m* ⟹ *DERIV* (λ*x*. *exp* (*g x*)) *x* :> *exp* (*g x*) ∗ *m*
by (*auto intro*!: *derivative-intros*)

112.12 The Constant Pi

definition *pi* :: *real*
where *pi* = *2* ∗ (*THE x*. *0* ≤ *x* ∧ *x* ≤ *2* ∧ *cos x* = *0*)

Show that there's a least positive *x* with *cos x* = *0*; hence define pi.

lemma *sin-paired*: (λ*n*. (− *1*) ^ *n* / (*fact* (*2* ∗ *n* + *1*)) ∗ *x* ^ (*2* ∗ *n* + *1*)) *sums sin x*
for *x* :: *real*
proof −
have (λ*n*. ∑ *k* = *n*∗*2*..<*n* ∗ *2* + *2*. *sin-coeff k* ∗ *x* ^ *k*) *sums sin x*
by (*rule sums-group*) (*use sin-converges* [*of x*, *unfolded scaleR-conv-of-real*] **in** *auto*)
then show *?thesis*
by (*simp add*: *sin-coeff-def ac-simps*)
qed

lemma *sin-gt-zero-02*:
fixes *x* :: *real*
assumes *0* < *x* **and** *x* < *2*
shows *0* < *sin x*
proof −
let *?f* = λ*n*::*nat*. ∑ *k* = *n*∗*2*..<*n*∗*2*+*2*. (− *1*) ^ *k* / (*fact* (*2*∗*k*+*1*)) ∗ *x*^(*2*∗*k*+*1*)
have *pos*: ∀ *n*. *0* < *?f n*
proof
fix *n* :: *nat*
let *?k2* = *real* (*Suc* (*Suc* (*4* ∗ *n*)))

```
    let ?k3 = real (Suc (Suc (Suc (4 * n))))
    have x * x < ?k2 * ?k3
      using assms by (intro mult-strict-mono', simp-all)
    then have x * x * x * x ^ (n * 4) < ?k2 * ?k3 * x * x ^ (n * 4)
      by (intro mult-strict-right-mono zero-less-power ‹0 < x›)
    then show 0 < ?f n
      by (simp add: ac-simps divide-less-eq)
qed
  have sums: ?f sums sin x
    by (rule sin-paired [THEN sums-group]) simp
  show 0 < sin x
    unfolding sums-unique [OF sums] using sums-summable [OF sums] pos by
(simp add: suminf-pos)
qed

lemma cos-double-less-one: 0 < x ⟹ x < 2 ⟹ cos (2 * x) < 1
  for x :: real
  using sin-gt-zero-02 [where x = x] by (auto simp: cos-squared-eq cos-double)

lemma cos-paired: (λn. (− 1) ^ n / (fact (2 * n)) * x ^ (2 * n)) sums cos x
  for x :: real
proof −
  have (λn. ∑ k = n * 2..<n * 2 + 2. cos-coeff k * x ^ k) sums cos x
    by (rule sums-group) (use cos-converges [of x, unfolded scaleR-conv-of-real] in
auto)
  then show ?thesis
    by (simp add: cos-coeff-def ac-simps)
qed

lemma sum-pos-lt-pair:
  fixes f :: nat ⇒ real
  assumes f: summable f and fplus: ⋀d. 0 < f (k + (Suc(Suc 0) * d)) + f (k +
((Suc (Suc 0) * d) + 1))
  shows sum f {..<k} < suminf f
proof −
  have (λn. ∑ n = n * Suc (Suc 0)..<n * Suc (Suc 0) +  Suc (Suc 0). f (n +
k))
            sums (∑ n. f (n + k))
  proof (rule sums-group)
    show (λn. f (n + k)) sums (∑ n. f (n + k))
      by (simp add: f summable-iff-shift summable-sums)
  qed auto
  with fplus have 0 < (∑ n. f (n + k))
    apply (simp add: add.commute)
    apply (metis (no-types, lifting) suminf-pos summable-def sums-unique)
    done
  then show ?thesis
    by (simp add: f suminf-minus-initial-segment)
qed
```

lemma *cos-two-less-zero* [*simp*]: *cos 2 < (0::real)*
proof −
note *fact-Suc* [*simp del*]
from *sums-minus* [*OF cos-paired*]
have ∗: (λ*n*. − ((− *1*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / *fact* (*2* ∗ *n*))) *sums* − *cos* (*2::real*)
by *simp*
then have *sm*: *summable* (λ*n*. − ((− *1::real*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*))))
by (*rule sums-summable*)
have *0* < (∑ *n*<*Suc* (*Suc* (*Suc 0*)). − ((− *1::real*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*))))
by (*simp add*: *fact-num-eq-if power-eq-if*)
moreover have (∑ *n*<*Suc* (*Suc* (*Suc 0*)). − ((− *1::real*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*)))) <
(∑ *n*. − ((− *1*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*))))
proof −
{
fix *d*
let *?six4d* = *Suc* (*Suc* (*Suc* (*Suc* (*Suc* (*4* ∗ *d*))))))
have (*4::real*) ∗ (*fact* (*?six4d*)) < (*Suc* (*Suc* (*?six4d*)) ∗ *fact* (*Suc* (*?six4d*)))
unfolding *of-nat-mult* **by** (*rule mult-strict-mono*) (*simp-all add*: *fact-less-mono*)
then have (*4::real*) ∗ (*fact* (*?six4d*)) < (*fact* (*Suc* (*Suc* (*?six4d*))))
by (*simp only*: *fact-Suc* [*of Suc* (*?six4d*)] *of-nat-mult of-nat-fact*)
then have (*4::real*) ∗ *inverse* (*fact* (*Suc* (*Suc* (*?six4d*)))) < *inverse* (*fact* (*?six4d*))
by (*simp add*: *inverse-eq-divide less-divide-eq*)
}
then show *?thesis*
by (*force intro*!: *sum-pos-lt-pair* [*OF sm*] *simp add*: *divide-inverse algebra-simps*)
qed
ultimately have *0* < (∑ *n*. − ((− *1::real*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*))))
by (*rule order-less-trans*)
moreover from ∗ **have** − *cos 2* = (∑ *n*. − ((− *1::real*) ^ *n* ∗ *2* ^ (*2* ∗ *n*) / (*fact* (*2* ∗ *n*))))
by (*rule sums-unique*)
ultimately have (*0::real*) < − *cos 2* **by** *simp*
then show *?thesis* **by** *simp*
qed

lemmas *cos-two-neq-zero* [*simp*] = *cos-two-less-zero* [*THEN less-imp-neq*]
lemmas *cos-two-le-zero* [*simp*] = *cos-two-less-zero* [*THEN order-less-imp-le*]

lemma *cos-is-zero*: ∃!*x::real*. *0* ≤ *x* ∧ *x* ≤ *2* ∧ *cos x* = *0*
proof (*rule ex-ex1I*)
show ∃*x::real*. *0* ≤ *x* ∧ *x* ≤ *2* ∧ *cos x* = *0*
by (*rule IVT2*) *simp-all*
next

fix *a b* :: *real*
assume *ab*: $0 \leq a \wedge a \leq 2 \wedge \cos a = 0\ 0 \leq b \wedge b \leq 2 \wedge \cos b = 0$
have *cosd*: $\bigwedge x$::*real*. *cos differentiable* (*at x*)
unfolding *real-differentiable-def* **by** (*auto intro*: *DERIV-cos*)
show $a = b$
proof (*cases a b rule*: *linorder-cases*)
case *less*
then obtain *z* **where** $a < z\ z < b$ (*cos has-real-derivative 0*) (*at z*)
using *Rolle* **by** (*metis cosd continuous-on-cos-real ab*)
then have *sin z* = *0*
using *DERIV-cos DERIV-unique neg-equal-0-iff-equal* **by** *blast*
then show *?thesis*
by (*metis* ‹$a < z$› ‹$z < b$› *ab order-less-le-trans less-le sin-gt-zero-02*)
next
case *greater*
then obtain *z* **where** $b < z\ z < a$ (*cos has-real-derivative 0*) (*at z*)
using *Rolle* **by** (*metis cosd continuous-on-cos-real ab*)
then have *sin z* = *0*
using *DERIV-cos DERIV-unique neg-equal-0-iff-equal* **by** *blast*
then show *?thesis*
by (*metis* ‹$b < z$› ‹$z < a$› *ab order-less-le-trans less-le sin-gt-zero-02*)
qed *auto*
qed

lemma *pi-half*: *pi/2* = (*THE x*. $0 \leq x \wedge x \leq 2 \wedge \cos x = 0$)
by (*simp add*: *pi-def*)

lemma *cos-pi-half* [*simp*]: *cos* (*pi/2*) = *0*
by (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])

lemma *cos-of-real-pi-half* [*simp*]: *cos* ((*of-real pi/2*) :: *'a*) = *0*
if *SORT-CONSTRAINT*(*'a*::{*real-field*,*banach*,*real-normed-algebra-1*})
by (*metis cos-pi-half cos-of-real eq-numeral-simps*(*4*)
nonzero-of-real-divide of-real-0 of-real-numeral)

lemma *pi-half-gt-zero* [*simp*]: *0* < *pi/2*
proof −
have *0* ≤ *pi/2*
by (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])
then show *?thesis*
by (*metis cos-pi-half cos-zero less-eq-real-def one-neq-zero*)
qed

lemmas *pi-half-neq-zero* [*simp*] = *pi-half-gt-zero* [*THEN less-imp-neq*, *symmetric*]
lemmas *pi-half-ge-zero* [*simp*] = *pi-half-gt-zero* [*THEN order-less-imp-le*]

lemma *pi-half-less-two* [*simp*]: *pi/2* < *2*
proof −
have *pi/2* ≤ *2*

by (*simp add*: *pi-half cos-is-zero* [*THEN theI'*])
then show *?thesis*
by (*metis cos-pi-half cos-two-neq-zero le-less*)
qed

lemmas *pi-half-neq-two* [*simp*] = *pi-half-less-two* [*THEN less-imp-neq*]
lemmas *pi-half-le-two* [*simp*] = *pi-half-less-two* [*THEN order-less-imp-le*]

lemma *pi-gt-zero* [*simp*]: $0 < pi$
using *pi-half-gt-zero* **by** *simp*

lemma *pi-ge-zero* [*simp*]: $0 \leq pi$
by (*rule pi-gt-zero* [*THEN order-less-imp-le*])

lemma *pi-neq-zero* [*simp*]: $pi \neq 0$
by (*rule pi-gt-zero* [*THEN less-imp-neq*, *symmetric*])

lemma *pi-not-less-zero* [*simp*]: $\neg\ pi < 0$
by (*simp add*: *linorder-not-less*)

lemma *minus-pi-half-less-zero*: $-(pi/2) < 0$
by *simp*

lemma *m2pi-less-pi*: $-\ (2{*}pi) < pi$
by *simp*

lemma *sin-pi-half* [*simp*]: $sin(pi/2) = 1$
using *sin-cos-squared-add2* [**where** $x = pi/2$]
using *sin-gt-zero-02* [*OF pi-half-gt-zero pi-half-less-two*]
by (*simp add*: *power2-eq-1-iff*)

lemma *sin-of-real-pi-half* [*simp*]: $sin\ ((of\text{-}real\ pi/2) :: 'a) = 1$
if *SORT-CONSTRAINT*('a::{*real-field*,*banach*,*real-normed-algebra-1*})
using *sin-pi-half*
by (*metis sin-pi-half eq-numeral-simps*(4) *nonzero-of-real-divide of-real-1 of-real-numeral sin-of-real*)

lemma *sin-cos-eq*: $sin\ x = cos\ (of\text{-}real\ pi/2 - x)$
for $x :: 'a$::{*real-normed-field*,*banach*}
by (*simp add*: *cos-diff*)

lemma *minus-sin-cos-eq*: $-\ sin\ x = cos\ (x + of\text{-}real\ pi/2)$
for $x :: 'a$::{*real-normed-field*,*banach*}
by (*simp add*: *cos-add nonzero-of-real-divide*)

lemma *cos-sin-eq*: $cos\ x = sin\ (of\text{-}real\ pi/2 - x)$
for $x :: 'a$::{*real-normed-field*,*banach*}
using *sin-cos-eq* [*of of-real pi*/2 − *x*] **by** *simp*

lemma *sin-add*: *sin* (*x* + *y*) = *sin x* ∗ *cos y* + *cos x* ∗ *sin y*
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *cos-add* [*of of-real pi*/*2* − *x* −*y*]
by (*simp add*: *cos-sin-eq*) (*simp add*: *sin-cos-eq*)

lemma *sin-diff*: *sin* (*x* − *y*) = *sin x* ∗ *cos y* − *cos x* ∗ *sin y*
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *sin-add* [*of x* − *y*] **by** *simp*

lemma *sin-double*: *sin*(*2* ∗ *x*) = *2* ∗ *sin x* ∗ *cos x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *sin-add* [**where** *x*=*x* **and** *y*=*x*] **by** *simp*

lemma *cos-of-real-pi* [*simp*]: *cos* (*of-real pi*) = −*1*
using *cos-add* [**where** *x* = *pi*/*2* **and** *y* = *pi*/*2*]
by (*simp add*: *cos-of-real*)

lemma *sin-of-real-pi* [*simp*]: *sin* (*of-real pi*) = *0*
using *sin-add* [**where** *x* = *pi*/*2* **and** *y* = *pi*/*2*]
by (*simp add*: *sin-of-real*)

lemma *cos-pi* [*simp*]: *cos pi* = −*1*
using *cos-add* [**where** *x* = *pi*/*2* **and** *y* = *pi*/*2*] **by** *simp*

lemma *sin-pi* [*simp*]: *sin pi* = *0*
using *sin-add* [**where** *x* = *pi*/*2* **and** *y* = *pi*/*2*] **by** *simp*

lemma *sin-periodic-pi* [*simp*]: *sin* (*x* + *pi*) = − *sin x*
by (*simp add*: *sin-add*)

lemma *sin-periodic-pi2* [*simp*]: *sin* (*pi* + *x*) = − *sin x*
by (*simp add*: *sin-add*)

lemma *cos-periodic-pi* [*simp*]: *cos* (*x* + *pi*) = − *cos x*
by (*simp add*: *cos-add*)

lemma *cos-periodic-pi2* [*simp*]: *cos* (*pi* + *x*) = − *cos x*
by (*simp add*: *cos-add*)

lemma *sin-periodic* [*simp*]: *sin* (*x* + *2* ∗ *pi*) = *sin x*
by (*simp add*: *sin-add sin-double cos-double*)

lemma *cos-periodic* [*simp*]: *cos* (*x* + *2* ∗ *pi*) = *cos x*
by (*simp add*: *cos-add sin-double cos-double*)

lemma *cos-npi* [*simp*]: *cos* (*real n* ∗ *pi*) = (− *1*) ^ *n*
by (*induct n*) (*auto simp*: *distrib-right*)

lemma *cos-npi2* [*simp*]: *cos* (*pi* ∗ *real n*) = (− *1*) ^ *n*

by (*metis cos-npi mult.commute*)

lemma *sin-npi* [*simp*]: *sin* (*real n* ∗ *pi*) = *0*
for *n* :: *nat*
by (*induct n*) (*auto simp*: *distrib-right*)

lemma *sin-npi2* [*simp*]: *sin* (*pi* ∗ *real n*) = *0*
for *n* :: *nat*
by (*simp add*: *mult.commute* [*of pi*])

lemma *sin-npi-numeral* [*simp*]: *sin*(*Num.numeral n* ∗ *pi*) = *0*
by (*metis of-nat-numeral sin-npi*)

lemma *sin-npi2-numeral* [*simp*]: *sin* (*pi* ∗ *Num.numeral n*) = *0*
by (*metis of-nat-numeral sin-npi2*)

lemma *sin-npi-complex′* [*simp*]: *sin* (*of-nat n* ∗ *of-real pi*) = *0*
by (*metis of-real-0 of-real-mult of-real-of-nat-eq sin-npi sin-of-real*)

lemma *cos-npi-numeral* [*simp*]: *cos* (*Num.numeral n* ∗ *pi*) = (− *1*) ^*Num.numeral n*
by (*metis cos-npi of-nat-numeral*)

lemma *cos-npi2-numeral* [*simp*]: *cos* (*pi* ∗ *Num.numeral n*) = (− *1*) ^*Num.numeral n*
by (*metis cos-npi2 of-nat-numeral*)

lemma *cos-npi-complex′* [*simp*]: *cos* (*of-nat n* ∗ *of-real pi*) = (*−1*) ^ *n* **for** *n*
proof −
have *cos* (*of-nat n* ∗ *of-real pi* :: *′a*) = *of-real* (*cos* (*real n* ∗ *pi*))
by (*subst cos-of-real* [*symmetric*]) *simp*
also have *cos* (*real n* ∗ *pi*) = (*−1*) ^ *n*
by *simp*
finally show *?thesis* **by** *simp*
qed

lemma *cos-two-pi* [*simp*]: *cos* (*2* ∗ *pi*) = *1*
by (*simp add*: *cos-double*)

lemma *sin-two-pi* [*simp*]: *sin* (*2* ∗ *pi*) = *0*
by (*simp add*: *sin-double*)

context
fixes *w* :: *′a*::{*real-normed-field*,*banach*}

begin

lemma *sin-times-sin*: *sin w* ∗ *sin z* = (*cos* (*w* − *z*) − *cos* (*w* + *z*)) / *2*
by (*simp add*: *cos-diff cos-add*)

lemma *sin-times-cos*: *sin w * cos z = (sin (w + z) + sin (w − z)) / 2*
by (*simp add*: *sin-diff sin-add*)

lemma *cos-times-sin*: *cos w * sin z = (sin (w + z) − sin (w − z)) / 2*
by (*simp add*: *sin-diff sin-add*)

lemma *cos-times-cos*: *cos w * cos z = (cos (w − z) + cos (w + z)) / 2*
by (*simp add*: *cos-diff cos-add*)

lemma *cos-double-cos*: *cos (2 * w) = 2 * cos w ^ 2 − 1*
by (*simp add*: *cos-double sin-squared-eq*)

lemma *cos-double-sin*: *cos (2 * w) = 1 − 2 * sin w ^ 2*
by (*simp add*: *cos-double sin-squared-eq*)

end

lemma *sin-plus-sin*: *sin w + sin z = 2 * sin ((w + z) / 2) * cos ((w − z) / 2)*
for *w* :: *'a*::{*real-normed-field*,*banach*}
apply (*simp add*: *mult.assoc sin-times-cos*)
apply (*simp add*: *field-simps*)
done

lemma *sin-diff-sin*: *sin w − sin z = 2 * sin ((w − z) / 2) * cos ((w + z) / 2)*
for *w* :: *'a*::{*real-normed-field*,*banach*}
apply (*simp add*: *mult.assoc sin-times-cos*)
apply (*simp add*: *field-simps*)
done

lemma *cos-plus-cos*: *cos w + cos z = 2 * cos ((w + z) / 2) * cos ((w − z) / 2)*
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
apply (*simp add*: *mult.assoc cos-times-cos*)
apply (*simp add*: *field-simps*)
done

lemma *cos-diff-cos*: *cos w − cos z = 2 * sin ((w + z) / 2) * sin ((z − w) / 2)*
for *w* :: *'a*::{*real-normed-field*,*banach*,*field*}
apply (*simp add*: *mult.assoc sin-times-sin*)
apply (*simp add*: *field-simps*)
done

lemma *sin-pi-minus* [*simp*]: *sin (pi − x) = sin x*
by (*metis sin-minus sin-periodic-pi minus-minus uminus-add-conv-diff*)

lemma *cos-pi-minus* [*simp*]: *cos (pi − x) = − (cos x)*
by (*metis cos-minus cos-periodic-pi uminus-add-conv-diff*)

lemma *sin-minus-pi* [*simp*]: *sin (x − pi) = − (sin x)*

by (*simp add*: *sin-diff*)

lemma *cos-minus-pi* [*simp*]: *cos* (*x* − *pi*) = − (*cos x*)
by (*simp add*: *cos-diff*)

lemma *sin-2pi-minus* [*simp*]: *sin* (*2* ∗ *pi* − *x*) = − (*sin x*)
by (*metis sin-periodic-pi2 add-diff-eq mult-2 sin-pi-minus*)

lemma *cos-2pi-minus* [*simp*]: *cos* (*2* ∗ *pi* − *x*) = *cos x*
by (*metis* (*no-types*, *opaque-lifting*) *cos-add cos-minus cos-two-pi sin-minus sin-two-pi diff-0-right minus-diff-eq mult-1 mult-zero-left uminus-add-conv-diff*)

lemma *sin-gt-zero2*: *0* < *x* ⟹ *x* < *pi*/*2* ⟹ *0* < *sin x*
by (*metis sin-gt-zero-02 order-less-trans pi-half-less-two*)

lemma *sin-less-zero*:
assumes − *pi*/*2* < *x* **and** *x* < *0*
shows *sin x* < *0*
proof −
have *0* < *sin* (− *x*)
using *assms* **by** (*simp only*: *sin-gt-zero2*)
then show *?thesis* **by** *simp*
qed

lemma *pi-less-4*: *pi* < *4*
using *pi-half-less-two* **by** *auto*

lemma *cos-gt-zero*: *0* < *x* ⟹ *x* < *pi*/*2* ⟹ *0* < *cos x*
by (*simp add*: *cos-sin-eq sin-gt-zero2*)

lemma *cos-gt-zero-pi*: −(*pi*/*2*) < *x* ⟹ *x* < *pi*/*2* ⟹ *0* < *cos x*
using *cos-gt-zero* [*of x*] *cos-gt-zero* [*of* −*x*]
by (*cases rule*: *linorder-cases* [*of x 0*]) *auto*

lemma *cos-ge-zero*: −(*pi*/*2*) ≤ *x* ⟹ *x* ≤ *pi*/*2* ⟹ *0* ≤ *cos x*
by (*auto simp*: *order-le-less cos-gt-zero-pi*)
(*metis cos-pi-half eq-divide-eq eq-numeral-simps*(*4*))

lemma *sin-gt-zero*: *0* < *x* ⟹ *x* < *pi* ⟹ *0* < *sin x*
by (*simp add*: *sin-cos-eq cos-gt-zero-pi*)

lemma *sin-lt-zero*: *pi* < *x* ⟹ *x* < *2* ∗ *pi* ⟹ *sin x* < *0*
using *sin-gt-zero* [*of x* − *pi*]
by (*simp add*: *sin-diff*)

lemma *pi-ge-two*: *2* ≤ *pi*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *pi* < *2* **by** *auto*

```
  have ∃y > pi. y < 2 ∧ y < 2 * pi
  proof (cases 2 < 2 * pi)
    case True
    with dense[OF ‹pi < 2›] show ?thesis by auto
  next
    case False
    have pi < 2 * pi by auto
    from dense[OF this] and False show ?thesis by auto
  qed
  then obtain y where pi < y and y < 2 and y < 2 * pi
    by blast
  then have 0 < sin y
    using sin-gt-zero-02 by auto
  moreover have sin y < 0
    using sin-gt-zero[of y − pi] ‹pi < y› and ‹y < 2 * pi› sin-periodic-pi[of y −
pi]
    by auto
  ultimately show False by auto
qed

lemma sin-ge-zero: 0 ≤ x ⟹ x ≤ pi ⟹ 0 ≤ sin x
  by (auto simp: order-le-less sin-gt-zero)

lemma sin-le-zero: pi ≤ x ⟹ x < 2 * pi ⟹ sin x ≤ 0
  using sin-ge-zero [of x − pi] by (simp add: sin-diff)

lemma sin-pi-divide-n-ge-0 [simp]:
  assumes n ≠ 0
  shows 0 ≤ sin (pi/real n)
  by (rule sin-ge-zero) (use assms in ‹simp-all add: field-split-simps›)

lemma sin-pi-divide-n-gt-0:
  assumes 2 ≤ n
  shows 0 < sin (pi/real n)
  by (rule sin-gt-zero) (use assms in ‹simp-all add: field-split-simps›)
```

Proof resembles that of *cos-is-zero* but with *pi* for the upper bound

```
lemma cos-total:
  assumes y: −1 ≤ y y ≤ 1
  shows ∃!x. 0 ≤ x ∧ x ≤ pi ∧ cos x = y
proof (rule ex-ex1I)
  show ∃x::real. 0 ≤ x ∧ x ≤ pi ∧ cos x = y
    by (rule IVT2) (simp-all add: y)
next
  fix a b :: real
  assume ab: 0 ≤ a ∧ a ≤ pi ∧ cos a = y 0 ≤ b ∧ b ≤ pi ∧ cos b = y
  have cosd: ⋀x::real. cos differentiable (at x)
    unfolding real-differentiable-def by (auto intro: DERIV-cos)
  show a = b
```

```
  proof (cases a b rule: linorder-cases)
    case less
    then obtain z where a < z z < b (cos has-real-derivative 0) (at z)
      using Rolle by (metis cosd continuous-on-cos-real ab)
    then have sin z = 0
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show ?thesis
      by (metis ‹a < z› ‹z < b› ab order-less-le-trans less-le sin-gt-zero)
  next
    case greater
    then obtain z where b < z z < a (cos has-real-derivative 0) (at z)
      using Rolle by (metis cosd continuous-on-cos-real ab)
    then have sin z = 0
      using DERIV-cos DERIV-unique neg-equal-0-iff-equal by blast
    then show ?thesis
      by (metis ‹b < z› ‹z < a› ab order-less-le-trans less-le sin-gt-zero)
  qed auto
qed

lemma sin-total:
  assumes y: −1 ≤ y y ≤ 1
  shows ∃!x. − (pi/2) ≤ x ∧ x ≤ pi/2 ∧ sin x = y
proof −
  from cos-total [OF y]
  obtain x where x: 0 ≤ x x ≤ pi cos x = y
    and uniq: ⋀x'. 0 ≤ x' ⟹ x' ≤ pi ⟹ cos x' = y ⟹ x' = x
    by blast
  show ?thesis
    unfolding sin-cos-eq
  proof (rule ex1I [where a=pi/2 − x])
    show − (pi/2) ≤ z ∧ z ≤ pi/2 ∧ cos (of-real pi/2 − z) = y ⟹
          z = pi/2 − x for z
      using uniq [of pi/2 −z] by auto
  qed (use x in auto)
qed

lemma cos-zero-lemma:
  assumes 0 ≤ x cos x = 0
  shows ∃ n. odd n ∧ x = of-nat n * (pi/2)
proof −
  have xle: x < (1 + real-of-int ⌊x/pi⌋) * pi
    using floor-correct [of x/pi]
    by (simp add: add.commute divide-less-eq)
  obtain n where real n * pi ≤ x x < real (Suc n) * pi
  proof
    show real (nat ⌊x / pi⌋) * pi ≤ x
      using assms floor-divide-lower [of pi x] by auto
    show x < real (Suc (nat ⌊x / pi⌋)) * pi
      using assms floor-divide-upper [of pi x]  by (simp add: xle)
```

```
  qed
  then have x: 0 ≤ x − n ∗ pi (x − n ∗ pi) ≤ pi cos (x − n ∗ pi) = 0
    by (auto simp: algebra-simps cos-diff assms)
  then have ∃!x. 0 ≤ x ∧ x ≤ pi ∧ cos x = 0
    by (auto simp: intro!: cos-total)
  then obtain ϑ where ϑ: 0 ≤ ϑ ϑ ≤ pi cos ϑ = 0
    and uniq: ⋀φ. 0 ≤ φ ⟹ φ ≤ pi ⟹ cos φ = 0 ⟹ φ = ϑ
    by blast
  then have x − real n ∗ pi = ϑ
    using x by blast
  moreover have pi/2 = ϑ
    using pi-half-ge-zero uniq by fastforce
  ultimately show ?thesis
    by (rule-tac x = Suc (2 ∗ n) in exI) (simp add: algebra-simps)
qed

lemma sin-zero-lemma:
  assumes 0 ≤ x sin x = 0
  shows ∃ n::nat. even n ∧ x = real n ∗ (pi/2)
proof −
  obtain n where odd n and n: x + pi/2 = of-nat n ∗ (pi/2) n > 0
    using cos-zero-lemma [of x + pi/2] assms by (auto simp add: cos-add)
  then have x = real (n − 1) ∗ (pi/2)
    by (simp add: algebra-simps of-nat-diff)
  then show ?thesis
    by (simp add: ‹odd n›)
qed

lemma cos-zero-iff:
  cos x = 0 ⟷ ((∃ n. odd n ∧ x = real n ∗ (pi/2)) ∨ (∃ n. odd n ∧ x = − (real
n ∗ (pi/2))))
  (is ?lhs = ?rhs)
proof −
  have ∗: cos (real n ∗ pi/2) = 0 if odd n for n :: nat
  proof −
    from that obtain m where n = 2 ∗ m + 1 ..
    then show ?thesis
      by (simp add: field-simps) (simp add: cos-add add-divide-distrib)
  qed
  show ?thesis
  proof
    show ?rhs if ?lhs
      using that cos-zero-lemma [of x] cos-zero-lemma [of −x] by force
    show ?lhs if ?rhs
      using that by (auto dest: ∗ simp del: eq-divide-eq-numeral1)
  qed
qed

lemma sin-zero-iff:
```

*sin x = 0 ⟷ ((∃ n. even n ∧ x = real n * (pi/2)) ∨ (∃ n. even n ∧ x = − (real n * (pi/2))))*
(**is** *?lhs = ?rhs*)
proof
show *?rhs* **if** *?lhs*
using *that sin-zero-lemma* [*of x*] *sin-zero-lemma* [*of −x*] **by** *force*
show *?lhs* **if** *?rhs*
using *that* **by** (*auto elim*: *evenE*)
qed

lemma *sin-zero-pi-iff*:
fixes *x*::*real*
assumes *|x| < pi*
shows *sin x = 0 ⟷ x = 0*
proof
show *x = 0* **if** *sin x = 0*
using *that assms* **by** (*auto simp*: *sin-zero-iff*)
qed *auto*

lemma *cos-zero-iff-int*: *cos x = 0 ⟷ (∃ i. odd i ∧ x = of-int i * (pi/2))*
proof −
have *1*: *⋀n. odd n ⟹ ∃ i. odd i ∧ int n = i*
by (*metis even-of-nat-iff*)
have *2*: *⋀n. odd n ⟹ ∃ i. odd i ∧ − (real n * pi) = real-of-int i * pi*
by (*metis even-minus even-of-nat-iff mult.commute mult-minus-right of-int-minus of-int-of-nat-eq*)
have *3*: *⟦odd i; ∀ n. even n ∨ i ≠ − (int n)⟧ ⟹ ∃ n. odd n ∧ i = int n* **for** *i*
by (*cases i rule*: *int-cases2*) *auto*
show *?thesis*
by (*force simp*: *of-nat-of-int-iff cos-zero-iff intro*!: *1 2 3*)
qed

lemma *sin-zero-iff-int*: *sin x = 0 ⟷ (∃ i. even i ∧ x = of-int i * (pi/2))* (**is** *?lhs = ?rhs*)
proof *safe*
assume *?lhs*
then consider (*plus*) *n* **where** *even n x = real n * (pi/2)* | (*minus*) *n* **where** *even n x = − (real n * (pi/2))*
using *sin-zero-iff* **by** *auto*
then show *∃ n. even n ∧ x = of-int n * (pi/2)*
proof *cases*
case *plus*
then show *?rhs*
by (*metis even-of-nat-iff of-int-of-nat-eq*)
next
case *minus*
then show *?thesis*
by (*rule-tac x=− (int n)* **in** *exI*) *simp*
qed

next
fix *i* :: *int*
assume *even i*
then show *sin* (*of-int i* ∗ (*pi*/*2*)) = *0*
by (*cases i rule*: *int-cases2*, *simp-all add*: *sin-zero-iff*)
qed

lemma *sin-zero-iff-int2*: *sin x* = *0* ⟷ (∃ *i*::*int*. *x* = *of-int i* ∗ *pi*)
proof −
have *sin x* = *0* ⟷ (∃ *i*. *even i* ∧ *x* = *real-of-int i* ∗ (*pi*/*2*))
by (*auto simp*: *sin-zero-iff-int*)
also have ... = (∃ *j*. *x* = *real-of-int* (*2*∗*j*) ∗ (*pi*/*2*))
using *dvd-triv-left* **by** *blast*
also have ... = (∃ *i*::*int*. *x* = *of-int i* ∗ *pi*)
by *auto*
finally show *?thesis* **.**
qed

lemma *cos-zero-iff-int2*:
fixes *x*::*real*
shows *cos x* = *0* ⟷ (∃ *n*::*int*. *x* = *n* ∗ *pi* + *pi*/*2*)
using *sin-zero-iff-int2*[*of x*−*pi*/*2*] **unfolding** *sin-cos-eq*
by (*auto simp add*: *algebra-simps*)

lemma *sin-npi-int* [*simp*]: *sin* (*pi* ∗ *of-int n*) = *0*
by (*simp add*: *sin-zero-iff-int2*)

lemma *cos-monotone-0-pi*:
assumes *0* ≤ *y* **and** *y* < *x* **and** *x* ≤ *pi*
shows *cos x* < *cos y*
proof −
have − (*x* − *y*) < *0* **using** *assms* **by** *auto*
from *MVT2*[*OF* ‹*y* < *x*› *DERIV-cos*]
obtain *z* **where** *y* < *z* **and** *z* < *x* **and** *cos-diff*: *cos x* − *cos y* = (*x* − *y*) ∗ − *sin z*
by *auto*
then have *0* < *z* **and** *z* < *pi*
using *assms* **by** *auto*
then have *0* < *sin z*
using *sin-gt-zero* **by** *auto*
then have *cos x* − *cos y* < *0*
unfolding *cos-diff minus-mult-commute*[*symmetric*]
using ‹− (*x* − *y*) < *0*›
using *mult-neg-pos* **by** *blast*
then show *?thesis* **by** *auto*
qed

lemma *cos-monotone-0-pi-le*:
assumes *0* ≤ *y* **and** *y* ≤ *x* **and** *x* ≤ *pi*

shows *cos x ≤ cos y*
proof (*cases y < x*)
case *True*
show *?thesis*
using *cos-monotone-0-pi*[*OF* ‹*0 ≤ y*› *True* ‹*x ≤ pi*›] **by** *auto*
next
case *False*
then have *y = x* **using** ‹*y ≤ x*› **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *cos-monotone-minus-pi-0*:
assumes *− pi ≤ y* **and** *y < x* **and** *x ≤ 0*
shows *cos y < cos x*
proof −
have *0 ≤ − x* **and** *− x < − y* **and** *− y ≤ pi*
using *assms* **by** *auto*
from *cos-monotone-0-pi*[*OF this*] **show** *?thesis*
unfolding *cos-minus* **.**
qed

lemma *cos-monotone-minus-pi-0′*:
assumes *− pi ≤ y* **and** *y ≤ x* **and** *x ≤ 0*
shows *cos y ≤ cos x*
proof (*cases y < x*)
case *True*
show *?thesis* **using** *cos-monotone-minus-pi-0*[*OF* ‹*−pi ≤ y*› *True* ‹*x ≤ 0*›]
by *auto*
next
case *False*
then have *y = x* **using** ‹*y ≤ x*› **by** *auto*
then show *?thesis* **by** *auto*
qed

lemma *sin-monotone-2pi*:
assumes *− (pi/2) ≤ y* **and** *y < x* **and** *x ≤ pi/2*
shows *sin y < sin x*
unfolding *sin-cos-eq*
using *assms* **by** (*auto intro*: *cos-monotone-0-pi*)

lemma *sin-monotone-2pi-le*:
assumes *− (pi/2) ≤ y* **and** *y ≤ x* **and** *x ≤ pi/2*
shows *sin y ≤ sin x*
by (*metis assms le-less sin-monotone-2pi*)

lemma *sin-x-le-x*:
fixes *x* :: *real*
assumes *x ≥ 0*
shows *sin x ≤ x*

proof −
let *?f* = λ*x*. *x* − *sin x*
have ⋀*u*. ⟦*0* ≤ *u*; *u* ≤ *x*⟧ ⟹ ∃ *y*. (*?f has-real-derivative 1* − *cos u*) (*at u*)
by (*auto intro*!: *derivative-eq-intros simp*: *field-simps*)
then have *?f x* ≥ *?f 0*
by (*metis cos-le-one diff-ge-0-iff-ge DERIV-nonneg-imp-nondecreasing* [*OF assms*])
then show *sin x* ≤ *x* **by** *simp*
qed

lemma *sin-x-ge-neg-x*:
fixes *x* :: *real*
assumes *x*: *x* ≥ *0*
shows *sin x* ≥ − *x*
proof −
let *?f* = λ*x*. *x* + *sin x*
have §: ⋀*u*. ⟦*0* ≤ *u*; *u* ≤ *x*⟧ ⟹ ∃ *y*. (*?f has-real-derivative 1* + *cos u*) (*at u*)
by (*auto intro*!: *derivative-eq-intros simp*: *field-simps*)
have *?f x* ≥ *?f 0*
by (*rule DERIV-nonneg-imp-nondecreasing* [*OF assms*]) (*use* § *real-0-le-add-iff* **in** *force*)
then show *sin x* ≥ −*x* **by** *simp*
qed

lemma *abs-sin-x-le-abs-x*: |*sin x*| ≤ |*x*|
for *x* :: *real*
using *sin-x-ge-neg-x* [*of x*] *sin-x-le-x* [*of x*] *sin-x-ge-neg-x* [*of* −*x*] *sin-x-le-x* [*of* −*x*]
by (*auto simp*: *abs-real-def*)

112.13 More Corollaries about Sine and Cosine

lemma *sin-cos-npi*: *sin* (*real* (*Suc* (*2* ∗ *n*)) ∗ *pi*/*2*) = (−*1*) ^ *n*
proof −
have *sin* ((*real n* + *1*/*2*) ∗ *pi*) = *cos* (*real n* ∗ *pi*)
by (*auto simp*: *algebra-simps sin-add*)
then show *?thesis*
by (*simp add*: *distrib-right add-divide-distrib add.commute mult.commute* [*of pi*])
qed

lemma *cos-2npi* [*simp*]: *cos* (*2* ∗ *real n* ∗ *pi*) = *1*
for *n* :: *nat*
by (*cases even n*) (*simp-all add*: *cos-double mult.assoc*)

lemma *cos-3over2-pi* [*simp*]: *cos* (*3*/*2*∗*pi*) = *0*
proof −
have *cos* (*3*/*2*∗*pi*) = *cos* (*pi* + *pi*/*2*)
by *simp*

also have ... = *0*
by (*subst cos-add*, *simp*)
finally show *?thesis* **.**
qed

lemma *sin-2npi* [*simp*]: *sin* (*2* ∗ *real n* ∗ *pi*) = *0*
for *n* :: *nat*
by (*auto simp*: *mult.assoc sin-double*)

lemma *sin-3over2-pi* [*simp*]: *sin* (*3*/*2*∗*pi*) = − *1*
proof −
have *sin* (*3*/*2*∗*pi*) = *sin* (*pi* + *pi*/*2*)
by *simp*
also have ... = −*1*
by (*subst sin-add*, *simp*)
finally show *?thesis* **.**
qed

lemma *cos-pi-eq-zero*: *cos* (*pi* ∗ *real* (*Suc* (*2* ∗ *m*)) / *2*) = *0*
by (*simp add*: *cos-add sin-add distrib-right distrib-left add-divide-distrib*)

lemma *DERIV-cos-add* [*simp*]: *DERIV* (λ*x*. *cos* (*x* + *k*)) *xa* :> − *sin* (*xa* + *k*)
by (*auto intro*!: *derivative-eq-intros*)

lemma *sin-zero-norm-cos-one*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
assumes *sin x* = *0*
shows *norm* (*cos x*) = *1*
using *sin-cos-squared-add* [*of x*, *unfolded assms*]
by (*simp add*: *square-norm-one*)

lemma *sin-zero-abs-cos-one*: *sin x* = *0* ⟹ |*cos x*| = (*1*::*real*)
using *sin-zero-norm-cos-one* **by** *fastforce*

lemma *cos-one-sin-zero*:
fixes *x* :: *'a*::{*real-normed-field*,*banach*}
assumes *cos x* = *1*
shows *sin x* = *0*
using *sin-cos-squared-add* [*of x*, *unfolded assms*]
by *simp*

lemma *sin-times-pi-eq-0*: *sin* (*x* ∗ *pi*) = *0* ⟷ *x* ∈ ℤ
by (*simp add*: *sin-zero-iff-int2*) (*metis Ints-cases Ints-of-int*)

lemma *cos-one-2pi*: *cos x* = *1* ⟷ (∃ *n*::*nat*. *x* = *n* ∗ *2* ∗ *pi*) ∨ (∃ *n*::*nat*. *x* = −(*n* ∗ *2* ∗ *pi*))
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*

```
  then have sin x = 0
    by (simp add: cos-one-sin-zero)
  then show ?rhs
  proof (simp only: sin-zero-iff, elim exE disjE conjE)
    fix n :: nat
    assume n: even n x = real n * (pi/2)
    then obtain m where m: n = 2 * m
      using dvdE by blast
    then have me: even m using ‹?lhs› n
    by (auto simp: field-simps) (metis one-neq-neg-one  power-minus-odd power-one)
    show ?rhs
      using m me n
      by (auto simp: field-simps elim!: evenE)
  next
    fix n :: nat
    assume n: even n x = - (real n * (pi/2))
    then obtain m where m: n = 2 * m
      using dvdE by blast
    then have me: even m using ‹?lhs› n
    by (auto simp: field-simps) (metis one-neq-neg-one  power-minus-odd power-one)
    show ?rhs
      using m me n
      by (auto simp: field-simps elim!: evenE)
  qed
next
  assume ?rhs
  then show cos x = 1
    by (metis cos-2npi cos-minus mult.assoc mult.left-commute)
qed

lemma cos-one-2pi-int: cos x = 1 ⟷ (∃ n::int. x = n * 2 * pi) (is ?lhs = ?rhs)
proof
  assume cos x = 1
  then show ?rhs
   by (metis cos-one-2pi mult.commute mult-minus-right of-int-minus of-int-of-nat-eq)
next
  assume ?rhs
  then obtain i where x = real-of-int i * 2 * pi
    by blast
  then show cos x = 1
    using int-cases2 [of i]
    unfolding cos-one-2pi by fastforce
qed

lemma cos-npi-int [simp]:
  fixes n::int shows cos (pi * of-int n) = (if even n then 1 else -1)
    by (auto simp: algebra-simps cos-one-2pi-int elim!: oddE evenE)

lemma sin-cos-sqrt: 0 ≤ sin x ⟹ sin x = sqrt (1 - (cos(x) ^ 2))
```

using *sin-squared-eq real-sqrt-unique* **by** *fastforce*

lemma *sin-eq-0-pi*: $-\ pi < x \Longrightarrow x < pi \Longrightarrow sin\ x = 0 \Longrightarrow x = 0$
by (*metis sin-gt-zero sin-minus minus-less-iff neg-0-less-iff-less not-less-iff-gr-or-eq*)

lemma *cos-treble-cos*: $cos\ (3 * x) = 4 * cos\ x\ \hat{}\ 3 - 3 * cos\ x$
for $x :: 'a$::{*real-normed-field*,*banach*}
proof −
have ∗: $(sin\ x * (sin\ x * 3)) = 3 - (cos\ x * (cos\ x * 3))$
by (*simp add*: *mult.assoc* [*symmetric*] *sin-squared-eq* [*unfolded power2-eq-square*])
have $cos(3 * x) = cos(2{*}x + x)$
by *simp*
also have $\ldots = 4 * cos\ x\ \hat{}\ 3 - 3 * cos\ x$
unfolding *cos-add cos-double sin-double*
by (*simp add*: ∗ *field-simps power2-eq-square power3-eq-cube*)
finally show *?thesis* **.**
qed

lemma *cos-45*: $cos\ (pi/4) = sqrt\ 2\ /\ 2$
proof −
let $?c = cos\ (pi/4)$
let $?s = sin\ (pi/4)$
have *nonneg*: $0 \leq ?c$
by (*simp add*: *cos-ge-zero*)
have $0 = cos\ (pi/4 + pi/4)$
by *simp*
also have $cos\ (pi/4 + pi/4) = ?c^2 - ?s^2$
by (*simp only*: *cos-add power2-eq-square*)
also have $\ldots = 2 * ?c^2 - 1$
by (*simp add*: *sin-squared-eq*)
finally have $?c^2 = (sqrt\ 2\ /\ 2)^2$
by (*simp add*: *power-divide*)
then show *?thesis*
using *nonneg* **by** (*rule power2-eq-imp-eq*) *simp*
qed

lemma *cos-30*: $cos\ (pi/6) = sqrt\ 3/2$
proof −
let $?c = cos\ (pi/6)$
let $?s = sin\ (pi/6)$
have *pos-c*: $0 < ?c$
by (*rule cos-gt-zero*) *simp-all*
have $0 = cos\ (pi/6 + pi/6 + pi/6)$
by *simp*
also have $\ldots = (?c * ?c - ?s * ?s) * ?c - (?s * ?c + ?c * ?s) * ?s$
by (*simp only*: *cos-add sin-add*)
also have $\ldots = ?c * (?c^2 - 3 * ?s^2)$
by (*simp add*: *algebra-simps power2-eq-square*)
finally have $?c^2 = (sqrt\ 3/2)^2$

using *pos-c* **by** (*simp add*: *sin-squared-eq power-divide*)
then show *?thesis*
using *pos-c* [*THEN order-less-imp-le*]
by (*rule power2-eq-imp-eq*) *simp*
qed

lemma *sin-45*: *sin* (*pi*/*4*) = *sqrt 2* / *2*
by (*simp add*: *sin-cos-eq cos-45*)

lemma *sin-60*: *sin* (*pi*/*3*) = *sqrt 3*/*2*
by (*simp add*: *sin-cos-eq cos-30*)

lemma *cos-60*: *cos* (*pi*/*3*) = *1*/*2*
proof −
have *0* ≤ *cos* (*pi*/*3*)
by (*rule cos-ge-zero*) (*use pi-half-ge-zero* **in** ‹*linarith*+›)
then show *?thesis*
by (*simp add*: *cos-squared-eq sin-60 power-divide power2-eq-imp-eq*)
qed

lemma *sin-30*: *sin* (*pi*/*6*) = *1*/*2*
by (*simp add*: *sin-cos-eq cos-60*)

lemma *cos-120*: *cos* (*2* ∗ *pi*/*3*) = −*1*/*2*
and *sin-120*: *sin* (*2* ∗ *pi*/*3*) = *sqrt 3* / *2*
using *sin-double*[*of pi*/*3*] *cos-double*[*of pi*/*3*]
by (*simp-all add*: *power2-eq-square sin-60 cos-60*)

lemma *cos-120′*: *cos* (*pi* ∗ *2* / *3*) = −*1*/*2*
using *cos-120* **by** (*subst mult.commute*)

lemma *sin-120′*: *sin* (*pi* ∗ *2* / *3*) = *sqrt 3* / *2*
using *sin-120* **by** (*subst mult.commute*)

lemma *cos-integer-2pi*: *n* ∈ ℤ ⟹ *cos*(*2* ∗ *pi* ∗ *n*) = *1*
by (*metis Ints-cases cos-one-2pi-int mult.assoc mult.commute*)

lemma *sin-integer-2pi*: *n* ∈ ℤ ⟹ *sin*(*2* ∗ *pi* ∗ *n*) = *0*
by (*metis sin-two-pi Ints-mult mult.assoc mult.commute sin-times-pi-eq-0*)

lemma *cos-int-2pin* [*simp*]: *cos* ((*2* ∗ *pi*) ∗ *of-int n*) = *1*
by (*simp add*: *cos-one-2pi-int*)

lemma *sin-int-2pin* [*simp*]: *sin* ((*2* ∗ *pi*) ∗ *of-int n*) = *0*
by (*metis Ints-of-int sin-integer-2pi*)

lemma *sin-cos-eq-iff*: *sin y* = *sin x* ∧ *cos y* = *cos x* ⟷ (∃ *n*::*int*. *y* = *x* + *2* ∗ *pi* ∗ *n*) (**is** *?L*=*?R*)
proof

assume *?L*
then have *cos* (*y*−*x*) = *1*
using *cos-add* [*of y* −*x*] **by** *simp*
then show *?R*
by (*metis cos-one-2pi-int add.commute diff-add-cancel mult.assoc mult.commute*)

next
assume *?R*
then show *?L*
by (*auto simp*: *sin-add cos-add*)
qed

lemma *sincos-principal-value*: ∃ *y*. (− *pi* < *y* ∧ *y* ≤ *pi*) ∧ (*sin y* = *sin x* ∧ *cos y* = *cos x*)
proof −
define *y* **where** *y* ≡ *pi* − (*2* ∗ *pi*) ∗ *frac* ((*pi* − *x*) / (*2* ∗ *pi*))
have −*pi* < *y y* ≤ *pi*
by (*auto simp*: *field-simps frac-lt-1 y-def*)
moreover
have *sin y* = *sin x cos y* = *cos x*
by (*simp-all add*: *y-def frac-def divide-simps sin-add cos-add mult-of-int-commute*)
ultimately
show *?thesis* **by** *metis*
qed

112.14 Tangent

definition *tan* :: *'a* ⇒ *'a*::{*real-normed-field*,*banach*}
where *tan* = (λ*x*. *sin x* / *cos x*)

lemma *tan-of-real*: *of-real* (*tan x*) = (*tan* (*of-real x*) :: *'a*::{*real-normed-field*,*banach*})
by (*simp add*: *tan-def sin-of-real cos-of-real*)

lemma *tan-in-Reals* [*simp*]: *z* ∈ ℝ ⟹ *tan z* ∈ ℝ
for *z* :: *'a*::{*real-normed-field*,*banach*}
by (*simp add*: *tan-def*)

lemma *tan-zero* [*simp*]: *tan 0* = *0*
by (*simp add*: *tan-def*)

lemma *tan-pi* [*simp*]: *tan pi* = *0*
by (*simp add*: *tan-def*)

lemma *tan-npi* [*simp*]: *tan* (*real n* ∗ *pi*) = *0*
for *n* :: *nat*
by (*simp add*: *tan-def*)

lemma *tan-pi-half* [*simp*]: *tan* (*pi* / *2*) = *0*
by (*simp add*: *tan-def*)

lemma *tan-minus* [*simp*]: *tan* (− *x*) = − *tan x*
by (*simp add*: *tan-def*)

lemma *tan-periodic* [*simp*]: *tan* (*x* + 2 ∗ *pi*) = *tan x*
by (*simp add*: *tan-def*)

lemma *lemma-tan-add1*: *cos x* ≠ 0 ⟹ *cos y* ≠ 0 ⟹ 1 − *tan x* ∗ *tan y* = *cos* (*x* + *y*)/(*cos x* ∗ *cos y*)
by (*simp add*: *tan-def cos-add field-simps*)

lemma *add-tan-eq*: *cos x* ≠ 0 ⟹ *cos y* ≠ 0 ⟹ *tan x* + *tan y* = *sin*(*x* + *y*)/(*cos x* ∗ *cos y*)
for *x* :: 'a::{*real-normed-field*,*banach*}
by (*simp add*: *tan-def sin-add field-simps*)

lemma *tan-eq-0-cos-sin*: *tan x* = 0 ⟷ *cos x* = 0 ∨ *sin x* = 0
by (*auto simp*: *tan-def*)

Note: half of these zeros would normally be regarded as undefined cases.

lemma *tan-eq-0-Ex*:
assumes *tan x* = 0
obtains *k*::*int* **where** *x* = (*k*/2) ∗ *pi*
using *assms*
by (*metis cos-zero-iff-int mult.commute sin-zero-iff-int tan-eq-0-cos-sin times-divide-eq-left*)

lemma *tan-add*:
cos x ≠ 0 ⟹ *cos y* ≠ 0 ⟹ *cos* (*x* + *y*) ≠ 0 ⟹ *tan* (*x* + *y*) = (*tan x* + *tan y*)/(1 − *tan x* ∗ *tan y*)
for *x* :: 'a::{*real-normed-field*,*banach*}
by (*simp add*: *add-tan-eq lemma-tan-add1 field-simps*) (*simp add*: *tan-def*)

lemma *tan-double*: *cos x* ≠ 0 ⟹ *cos* (2 ∗ *x*) ≠ 0 ⟹ *tan* (2 ∗ *x*) = (2 ∗ *tan x*) / (1 − (*tan x*)²)
for *x* :: 'a::{*real-normed-field*,*banach*}
using *tan-add* [*of x x*] **by** (*simp add*: *power2-eq-square*)

lemma *tan-gt-zero*: 0 < *x* ⟹ *x* < *pi*/2 ⟹ 0 < *tan x*
by (*simp add*: *tan-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi*)

lemma *tan-less-zero*:
assumes − *pi*/2 < *x* **and** *x* < 0
shows *tan x* < 0
proof −
have 0 < *tan* (− *x*)
using *assms* **by** (*simp only*: *tan-gt-zero*)
then show *?thesis* **by** *simp*
qed

lemma *tan-half*: *tan x = sin (2 * x) / (cos (2 * x) + 1)*
for *x* :: *'a*::{*real-normed-field*,*banach*,*field*}
unfolding *tan-def sin-double cos-double sin-squared-eq*
by (*simp add*: *power2-eq-square*)

lemma *tan-30*: *tan (pi/6) = 1 / sqrt 3*
unfolding *tan-def* **by** (*simp add*: *sin-30 cos-30*)

lemma *tan-45*: *tan (pi/4) = 1*
unfolding *tan-def* **by** (*simp add*: *sin-45 cos-45*)

lemma *tan-60*: *tan (pi/3) = sqrt 3*
unfolding *tan-def* **by** (*simp add*: *sin-60 cos-60*)

lemma *DERIV-tan* [*simp*]: *cos x ≠ 0 ⟹ DERIV tan x :> inverse ((cos x)²)*
for *x* :: *'a*::{*real-normed-field*,*banach*}
unfolding *tan-def*
by (*auto intro*!: *derivative-eq-intros*, *simp add*: *divide-inverse power2-eq-square*)

declare *DERIV-tan*[*THEN DERIV-chain2*, *derivative-intros*]
and *DERIV-tan*[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemmas *has-derivative-tan*[*derivative-intros*] = *DERIV-tan*[*THEN DERIV-compose-FDERIV*]

lemma *isCont-tan*: *cos x ≠ 0 ⟹ isCont tan x*
for *x* :: *'a*::{*real-normed-field*,*banach*}
by (*rule DERIV-tan* [*THEN DERIV-isCont*])

lemma *isCont-tan'* [*simp*,*continuous-intros*]:
fixes *a* :: *'a*::{*real-normed-field*,*banach*} **and** *f* :: *'a ⇒ 'a*
shows *isCont f a ⟹ cos (f a) ≠ 0 ⟹ isCont (λx. tan (f x)) a*
by (*rule isCont-o2* [*OF - isCont-tan*])

lemma *tendsto-tan* [*tendsto-intros*]:
fixes *f* :: *'a ⇒ 'a*::{*real-normed-field*,*banach*}
shows *(f ⟶ a) F ⟹ cos a ≠ 0 ⟹ ((λx. tan (f x)) ⟶ tan a) F*
by (*rule isCont-tendsto-compose* [*OF isCont-tan*])

lemma *continuous-tan*:
fixes *f* :: *'a ⇒ 'a*::{*real-normed-field*,*banach*}
shows *continuous F f ⟹ cos (f (Lim F (λx. x))) ≠ 0 ⟹ continuous F (λx. tan (f x))*
unfolding *continuous-def* **by** (*rule tendsto-tan*)

lemma *continuous-on-tan* [*continuous-intros*]:
fixes *f* :: *'a ⇒ 'a*::{*real-normed-field*,*banach*}
shows *continuous-on s f ⟹ (∀x∈s. cos (f x) ≠ 0) ⟹ continuous-on s (λx.*

tan (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-tan*)

lemma *continuous-within-tan* [*continuous-intros*]:
fixes *f* :: *'a* ⇒ *'a*::{*real-normed-field*,*banach*}
shows *continuous* (*at x within s*) *f* ⟹
cos (*f x*) ≠ *0* ⟹ *continuous* (*at x within s*) (λ*x*. *tan* (*f x*))
unfolding *continuous-within* **by** (*rule tendsto-tan*)

lemma *LIM-cos-div-sin*: (λ*x*. *cos*(*x*)/*sin*(*x*)) −*pi*/*2*→ *0*
by (*rule tendsto-cong-limit*, (*rule tendsto-intros*)+, *simp-all*)

lemma *lemma-tan-total*:
assumes *0* < *y* **shows** ∃ *x*. *0* < *x* ∧ *x* < *pi*/*2* ∧ *y* < *tan x*
proof −
obtain *s* **where** *0* < *s*
and *s*: ⋀*x*. ⟦*x* ≠ *pi*/*2*; *norm* (*x* − *pi*/*2*) < *s*⟧ ⟹ *norm* (*cos x* / *sin x* − *0*) < *inverse y*
using *LIM-D* [*OF LIM-cos-div-sin*, *of inverse y*] *that assms* **by** *force*
obtain *e* **where** *e*: *0* < *e e* < *s e* < *pi*/*2*
using ‹*0* < *s*› *field-lbound-gt-zero pi-half-gt-zero* **by** *blast*
show *?thesis*
proof (*intro exI conjI*)
have *0* < *sin e 0* < *cos e*
using *e* **by** (*auto intro*: *cos-gt-zero sin-gt-zero2 simp*: *mult.commute*)
then
show *y* < *tan* (*pi*/*2* − *e*)
using *s* [*of pi*/*2* − *e*] *e assms*
by (*simp add*: *tan-def sin-diff cos-diff*) (*simp add*: *field-simps split*: *if-split-asm*)
qed (*use e* **in** *auto*)
qed

lemma *tan-total-pos*:
assumes *0* ≤ *y* **shows** ∃ *x*. *0* ≤ *x* ∧ *x* < *pi*/*2* ∧ *tan x* = *y*
proof (*cases y* = *0*)
case *True*
then show *?thesis*
using *pi-half-gt-zero tan-zero* **by** *blast*
next
case *False*
with *assms* **have** *y* > *0*
by *linarith*
obtain *x* **where** *x*: *0* < *x x* < *pi*/*2 y* < *tan x*
using *lemma-tan-total* ‹*0* < *y*› **by** *blast*
have ∃ *u*≥*0*. *u* ≤ *x* ∧ *tan u* = *y*
proof (*intro IVT allI impI*)
show *isCont tan u* **if** *0* ≤ *u* ∧ *u* ≤ *x* **for** *u*
proof −
have *cos u* ≠ *0*

```
      using antisym-conv2 cos-gt-zero that x(2) by fastforce
    with assms show ?thesis
      by (auto intro!: DERIV-tan [THEN DERIV-isCont])
  qed
 qed (use assms x in auto)
 then show ?thesis
  using x(2) by auto
qed

lemma lemma-tan-total1: ∃ x. −(pi/2) < x ∧ x < (pi/2) ∧ tan x = y
proof (cases 0::real y rule: le-cases)
 case le
 then show ?thesis
  by (meson less-le-trans minus-pi-half-less-zero tan-total-pos)
next
 case ge
 with tan-total-pos [of −y] obtain x where 0 ≤ x x < pi/2 tan x = − y
  by force
 then show ?thesis
  by (rule-tac x=−x in exI) auto
qed

proposition tan-total: ∃! x. −(pi/2) < x ∧ x < (pi/2) ∧ tan x = y
proof −
 have u = v if u: − (pi/2) < u u < pi/2 and v: − (pi/2) < v v < pi/2
  and eq: tan u = tan v for u v
 proof (cases u v rule: linorder-cases)
  case less
  have ⋀x. u ≤ x ∧ x ≤ v ⟶ isCont tan x
   by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans u(1)
v(2))
  then have continuous-on {u..v} tan
   by (simp add: continuous-at-imp-continuous-on)
  moreover have ⋀x. u < x ∧ x < v ⟹ tan differentiable (at x)
   by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
order.strict-trans u(1) v(2))
  ultimately obtain z where u < z z < v DERIV tan z :> 0
   by (metis less Rolle eq)
  moreover have cos z ≠ 0
   by (metis (no-types) ‹u < z› ‹z < v› cos-gt-zero-pi less-le-trans linorder-not-less
not-less-iff-gr-or-eq u(1) v(2))
  ultimately show ?thesis
   using DERIV-unique [OF - DERIV-tan] by fastforce
 next
  case greater
  have ⋀x. v ≤ x ∧ x ≤ u ⟹ isCont tan x
   by (metis cos-gt-zero-pi isCont-tan le-less-trans less-irrefl less-le-trans u(2)
v(1))
  then have continuous-on {v..u} tan
```

```
      by (simp add: continuous-at-imp-continuous-on)
    moreover have ⋀x. v < x ∧ x < u ⟹ tan differentiable (at x)
    by (metis DERIV-tan cos-gt-zero-pi real-differentiable-def less-numeral-extra(3)
order.strict-trans u(2) v(1))
    ultimately obtain z where v < z z < u DERIV tan z :> 0
      by (metis greater Rolle eq)
    moreover have cos z ≠ 0
       by (metis ‹v < z› ‹z < u› cos-gt-zero-pi less-eq-real-def less-le-trans or-
der-less-irrefl u(2) v(1))
    ultimately show ?thesis
      using DERIV-unique [OF - DERIV-tan] by fastforce
  qed auto
  then have ∃!x. - (pi/2) < x ∧ x < pi/2 ∧ tan x = y
    if x: - (pi/2) < x x < pi/2 tan x = y for x
    using that by auto
  then show ?thesis
    using lemma-tan-total1 [where y = y]
    by auto
qed

lemma tan-monotone:
  assumes - (pi/2) < y and y < x and x < pi/2
  shows tan y < tan x
proof -
  have DERIV tan x' :> inverse ((cos x')^2) if y ≤ x' x' ≤ x for x'
  proof -
    have -(pi/2) < x' and x' < pi/2
      using that assms by auto
    with cos-gt-zero-pi have cos x' ≠ 0 by force
    then show DERIV tan x' :> inverse ((cos x')^2)
      by (rule DERIV-tan)
  qed
  from MVT2[OF ‹y < x› this]
  obtain z where y < z and z < x
    and tan-diff: tan x - tan y = (x - y) * inverse ((cos z)^2) by auto
  then have - (pi/2) < z and z < pi/2
    using assms by auto
  then have 0 < cos z
    using cos-gt-zero-pi by auto
  then have inv-pos: 0 < inverse ((cos z)^2)
    by auto
  have 0 < x - y using ‹y < x› by auto
  with inv-pos have 0 < tan x - tan y
    unfolding tan-diff by auto
  then show ?thesis by auto
qed

lemma tan-monotone':
  assumes - (pi/2) < y
```

```
    and y < pi/2
    and - (pi/2) < x
    and x < pi/2
  shows y < x ⟷ tan y < tan x
proof
  assume y < x
  then show tan y < tan x
    using tan-monotone and ‹- (pi/2) < y› and ‹x < pi/2› by auto
next
  assume tan y < tan x
  show y < x
  proof (rule ccontr)
    assume ¬ ?thesis
    then have x ≤ y by auto
    then have tan x ≤ tan y
    proof (cases x = y)
      case True
      then show ?thesis by auto
    next
      case False
      then have x < y using ‹x ≤ y› by auto
      from tan-monotone[OF ‹- (pi/2) < x› this ‹y < pi/2›] show ?thesis
        by auto
    qed
    then show False
      using ‹tan y < tan x› by auto
  qed
qed

lemma tan-inverse: 1 / (tan y) = tan (pi/2 - y)
  unfolding tan-def sin-cos-eq[of y] cos-sin-eq[of y] by auto

lemma tan-periodic-pi[simp]: tan (x + pi) = tan x
  by (simp add: tan-def)

lemma tan-periodic-nat[simp]: tan (x + real n * pi) = tan x
proof (induct n arbitrary: x)
  case 0
  then show ?case by simp
next
  case (Suc n)
  have split-pi-off: x + real (Suc n) * pi = (x + real n * pi) + pi
    unfolding Suc-eq-plus1 of-nat-add  distrib-right by auto
  show ?case
    unfolding split-pi-off using Suc by auto
qed

lemma tan-periodic-int[simp]: tan (x + of-int i * pi) = tan x
proof (cases 0 ≤ i)
```

case *False*
then have *i-nat*: *of-int i = − of-int* (*nat* (*− i*)) **by** *auto*
then show *?thesis*
by (*smt* (*verit*, *best*) *mult-minus-left of-int-of-nat-eq tan-periodic-nat*)
qed (*use zero-le-imp-eq-int* **in** *fastforce*)

lemma *tan-periodic-n*[*simp*]: *tan* (*x + numeral n ∗ pi*) = *tan x*
using *tan-periodic-int*[*of - numeral n*] **by** *simp*

lemma *tan-minus-45* [*simp*]: *tan* (*−*(*pi/4*)) = *−1*
unfolding *tan-def* **by** (*simp add*: *sin-45 cos-45*)

lemma *tan-diff*:
cos x ≠ 0 ⟹ cos y ≠ 0 ⟹ cos (*x − y*) *≠ 0 ⟹ tan* (*x − y*) = (*tan x − tan y*)/(*1 + tan x ∗ tan y*)
for *x* :: *'a*::{*real-normed-field*,*banach*}
using *tan-add* [*of x −y*] **by** *simp*

lemma *tan-pos-pi2-le*: *0 ≤ x ⟹ x < pi/2 ⟹ 0 ≤ tan x*
using *less-eq-real-def tan-gt-zero* **by** *auto*

lemma *cos-tan*: |*x*| *< pi/2 ⟹ cos x = 1 / sqrt* (*1 + tan x ^ 2*)
using *cos-gt-zero-pi* [*of x*]
by (*simp add*: *field-split-simps tan-def real-sqrt-divide abs-if split*: *if-split-asm*)

lemma *cos-tan-half*: *cos x ≠0 ⟹ cos* (*2∗x*) = (*1 − *(*tan x*)^2) / (*1 + *(*tan x*)^2)
unfolding *cos-double tan-def* **by** (*auto simp add*:*field-simps*)

lemma *sin-tan*: |*x*| *< pi/2 ⟹ sin x = tan x / sqrt* (*1 + tan x ^ 2*)
using *cos-gt-zero* [*of x*] *cos-gt-zero* [*of −x*]
by (*force simp*: *field-split-simps tan-def real-sqrt-divide abs-if split*: *if-split-asm*)

lemma *sin-tan-half*: *sin* (*2∗x*) = *2 ∗ tan x / *(*1 + *(*tan x*)^2)
unfolding *sin-double tan-def*
by (*cases cos x=0*) (*auto simp add*:*field-simps power2-eq-square*)

lemma *tan-mono-le*: *−*(*pi/2*) *< x ⟹ x ≤ y ⟹ y < pi/2 ⟹ tan x ≤ tan y*
using *less-eq-real-def tan-monotone* **by** *auto*

lemma *tan-mono-lt-eq*:
−(*pi/2*) *< x ⟹ x < pi/2 ⟹ −*(*pi/2*) *< y ⟹ y < pi/2 ⟹ tan x < tan y ⟷ x < y*
using *tan-monotone′* **by** *blast*

lemma *tan-mono-le-eq*:
−(*pi/2*) *< x ⟹ x < pi/2 ⟹ −*(*pi/2*) *< y ⟹ y < pi/2 ⟹ tan x ≤ tan y ⟷ x ≤ y*
by (*meson tan-mono-le not-le tan-monotone*)

lemma *tan-bound-pi2*: $|x| < pi/4 \Longrightarrow |tan\ x| < 1$
using *tan-45 tan-monotone* [*of x pi/4*] *tan-monotone* [*of* $-x$ *pi/4*]
by (*auto simp*: *abs-if split*: *if-split-asm*)

lemma *tan-cot*: $tan(pi/2 - x) = inverse(tan\ x)$
by (*simp add*: *tan-def sin-diff cos-diff*)

112.15 Cotangent

definition *cot* :: $'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
where $cot = (\lambda x.\ cos\ x\ /\ sin\ x)$

lemma *cot-of-real*: *of-real* (*cot x*) = (*cot* (*of-real x*) :: $'a$::{*real-normed-field*,*banach*})
by (*simp add*: *cot-def sin-of-real cos-of-real*)

lemma *cot-in-Reals* [*simp*]: $z \in \mathbb{R} \Longrightarrow cot\ z \in \mathbb{R}$
for z :: $'a$::{*real-normed-field*,*banach*}
by (*simp add*: *cot-def*)

lemma *cot-zero* [*simp*]: $cot\ 0 = 0$
by (*simp add*: *cot-def*)

lemma *cot-pi* [*simp*]: $cot\ pi = 0$
by (*simp add*: *cot-def*)

lemma *cot-npi* [*simp*]: $cot\ (real\ n * pi) = 0$
for n :: *nat*
by (*simp add*: *cot-def*)

lemma *cot-minus* [*simp*]: $cot\ (-x) = -\ cot\ x$
by (*simp add*: *cot-def*)

lemma *cot-periodic* [*simp*]: $cot\ (x + 2 * pi) = cot\ x$
by (*simp add*: *cot-def*)

lemma *cot-altdef*: $cot\ x = inverse\ (tan\ x)$
by (*simp add*: *cot-def tan-def*)

lemma *tan-altdef*: $tan\ x = inverse\ (cot\ x)$
by (*simp add*: *cot-def tan-def*)

lemma *tan-cot'*: $tan\ (pi/2 - x) = cot\ x$
by (*simp add*: *tan-cot cot-altdef*)

lemma *cot-gt-zero*: $0 < x \Longrightarrow x < pi/2 \Longrightarrow 0 < cot\ x$
by (*simp add*: *cot-def zero-less-divide-iff sin-gt-zero2 cos-gt-zero-pi*)

lemma *cot-less-zero*:

assumes *lb*: $-\ pi/2 < x$ **and** $x < 0$
shows $cot\ x < 0$
by (*smt* (*verit*) *assms cot-gt-zero cot-minus divide-minus-left*)

lemma *DERIV-cot* [*simp*]: $sin\ x \neq 0 \Longrightarrow DERIV\ cot\ x :> -inverse\ ((sin\ x)^2)$
for $x :: 'a$::{*real-normed-field*,*banach*}
unfolding *cot-def* **using** *cos-squared-eq*[*of x*]
by (*auto intro*!: *derivative-eq-intros*) (*simp add*: *divide-inverse power2-eq-square*)

lemma *isCont-cot*: $sin\ x \neq 0 \Longrightarrow isCont\ cot\ x$
for $x :: 'a$::{*real-normed-field*,*banach*}
by (*rule DERIV-cot* [*THEN DERIV-isCont*])

lemma *isCont-cot'* [*simp*,*continuous-intros*]:
$isCont\ f\ a \Longrightarrow sin\ (f\ a) \neq 0 \Longrightarrow isCont\ (\lambda x.\ cot\ (f\ x))\ a$
for $a :: 'a$::{*real-normed-field*,*banach*} **and** $f :: 'a \Rightarrow 'a$
by (*rule isCont-o2* [*OF - isCont-cot*])

lemma *tendsto-cot* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow sin\ a \neq 0 \Longrightarrow ((\lambda x.\ cot\ (f\ x)) \longrightarrow cot\ a)\ F$
for $f :: 'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
by (*rule isCont-tendsto-compose* [*OF isCont-cot*])

lemma *continuous-cot*:
$continuous\ F\ f \Longrightarrow sin\ (f\ (Lim\ F\ (\lambda x.\ x))) \neq 0 \Longrightarrow continuous\ F\ (\lambda x.\ cot\ (f\ x))$
for $f :: 'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
unfolding *continuous-def* **by** (*rule tendsto-cot*)

lemma *continuous-on-cot* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
shows $continuous\text{-}on\ s\ f \Longrightarrow (\forall x{\in}s.\ sin\ (f\ x) \neq 0) \Longrightarrow continuous\text{-}on\ s\ (\lambda x.\ cot\ (f\ x))$
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-cot*)

lemma *continuous-within-cot* [*continuous-intros*]:
fixes $f :: 'a \Rightarrow 'a$::{*real-normed-field*,*banach*}
shows $continuous\ (at\ x\ within\ s)\ f \Longrightarrow sin\ (f\ x) \neq 0 \Longrightarrow continuous\ (at\ x\ within\ s)\ (\lambda x.\ cot\ (f\ x))$
unfolding *continuous-within* **by** (*rule tendsto-cot*)

112.16 Inverse Trigonometric Functions

definition $arcsin :: real \Rightarrow real$
where $arcsin\ y = (THE\ x.\ -(pi/2) \leq x \wedge x \leq pi/2 \wedge sin\ x = y)$

definition $arccos :: real \Rightarrow real$
where $arccos\ y = (THE\ x.\ 0 \leq x \wedge x \leq pi \wedge cos\ x = y)$

definition $arctan :: real \Rightarrow real$

where *arctan y* = (*THE x*. −(*pi*/*2*) < *x* ∧ *x* < *pi*/*2* ∧ *tan x* = *y*)

lemma *arcsin*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ − (*pi*/*2*) ≤ *arcsin y* ∧ *arcsin y* ≤ *pi*/*2* ∧ *sin* (*arcsin y*) = *y*
unfolding *arcsin-def* **by** (*rule theI'* [*OF sin-total*])

lemma *arcsin-pi*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ − (*pi*/*2*) ≤ *arcsin y* ∧ *arcsin y* ≤ *pi* ∧ *sin* (*arcsin y*) = *y*
by (*drule* (*1*) *arcsin*) (*force intro*: *order-trans*)

lemma *sin-arcsin* [*simp*]: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *sin* (*arcsin y*) = *y*
by (*blast dest*: *arcsin*)

lemma *arcsin-bounded*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ − (*pi*/*2*) ≤ *arcsin y* ∧ *arcsin y* ≤ *pi*/*2*
by (*blast dest*: *arcsin*)

lemma *arcsin-lbound*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ − (*pi*/*2*) ≤ *arcsin y*
by (*blast dest*: *arcsin*)

lemma *arcsin-ubound*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *arcsin y* ≤ *pi*/*2*
by (*blast dest*: *arcsin*)

lemma *arcsin-lt-bounded*:
assumes − *1* < *y y* < *1*
shows − (*pi*/*2*) < *arcsin y* ∧ *arcsin y* < *pi*/*2*
proof −
have *arcsin y* ≠ *pi*/*2*
by (*metis arcsin assms not-less not-less-iff-gr-or-eq sin-pi-half*)
moreover have *arcsin y* ≠ − *pi*/*2*
by (*metis arcsin assms minus-divide-left not-less not-less-iff-gr-or-eq sin-minus sin-pi-half*)
ultimately show *?thesis*
using *arcsin-bounded* [*of y*] *assms* **by** *auto*
qed

lemma *arcsin-sin*: − (*pi*/*2*) ≤ *x* ⟹ *x* ≤ *pi*/*2* ⟹ *arcsin* (*sin x*) = *x*
unfolding *arcsin-def*
using *the1-equality* [*OF sin-total*] **by** *simp*

lemma *arcsin-unique*:
assumes −*pi*/*2* ≤ *x* **and** *x* ≤ *pi*/*2* **and** *sin x* = *y* **shows** *arcsin y* = *x*
using *arcsin-sin*[*of x*] *assms* **by** *force*

lemma *arcsin-0* [*simp*]: *arcsin 0* = *0*
using *arcsin-sin* [*of 0*] **by** *simp*

lemma *arcsin-1* [*simp*]: *arcsin 1* = *pi*/*2*
using *arcsin-sin* [*of pi*/*2*] **by** *simp*

lemma *arcsin-minus-1* [*simp*]: *arcsin* (− *1*) = − (*pi*/*2*)
using *arcsin-sin* [*of* − *pi*/*2*] **by** *simp*

lemma *arcsin-minus*: − *1* ≤ *x* ⟹ *x* ≤ *1* ⟹ *arcsin* (− *x*) = − *arcsin x*
by (*metis* (*no-types*, *opaque-lifting*) *arcsin arcsin-sin minus-minus neg-le-iff-le sin-minus*)

lemma *arcsin-one-half* [*simp*]: *arcsin* (*1*/*2*) = *pi* / *6*
and *arcsin-minus-one-half* [*simp*]: *arcsin* (−(*1*/*2*)) = −*pi* / *6*
by (*intro arcsin-unique*; *simp add*: *sin-30 field-simps*)+

lemma *arcsin-one-over-sqrt-2*: *arcsin* (*1* / *sqrt 2*) = *pi* / *4*
by (*rule arcsin-unique*) (*auto simp*: *sin-45 field-simps*)

lemma *arcsin-eq-iff*: |*x*| ≤ *1* ⟹ |*y*| ≤ *1* ⟹ *arcsin x* = *arcsin y* ⟷ *x* = *y*
by (*metis abs-le-iff arcsin minus-le-iff*)

lemma *cos-arcsin-nonzero*: − *1* < *x* ⟹ *x* < *1* ⟹ *cos* (*arcsin x*) ≠ *0*
using *arcsin-lt-bounded cos-gt-zero-pi* **by** *force*

lemma *arccos*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *0* ≤ *arccos y* ∧ *arccos y* ≤ *pi* ∧ *cos* (*arccos y*) = *y*
unfolding *arccos-def* **by** (*rule theI'* [*OF cos-total*])

lemma *cos-arccos* [*simp*]: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *cos* (*arccos y*) = *y*
by (*blast dest*: *arccos*)

lemma *arccos-bounded*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *0* ≤ *arccos y* ∧ *arccos y* ≤ *pi*
by (*blast dest*: *arccos*)

lemma *arccos-lbound*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *0* ≤ *arccos y*
by (*blast dest*: *arccos*)

lemma *arccos-ubound*: − *1* ≤ *y* ⟹ *y* ≤ *1* ⟹ *arccos y* ≤ *pi*
by (*blast dest*: *arccos*)

lemma *arccos-lt-bounded*:
assumes − *1* < *y y* < *1*
shows *0* < *arccos y* ∧ *arccos y* < *pi*
proof −
have *arccos y* ≠ *0*
by (*metis* (*no-types*) *arccos assms*(*1*) *assms*(*2*) *cos-zero less-eq-real-def less-irrefl*)
moreover have *arccos y* ≠ −*pi*
by (*metis arccos assms*(*1*) *assms*(*2*) *cos-minus cos-pi not-less not-less-iff-gr-or-eq*)
ultimately show *?thesis*
using *arccos-bounded* [*of y*] *assms*
by (*metis arccos cos-pi not-less not-less-iff-gr-or-eq*)
qed

lemma *arccos-cos*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow arccos\ (cos\ x) = x$
by (*auto simp*: *arccos-def intro*!: *the1-equality cos-total*)

lemma *arccos-cos2*: $x \leq 0 \Longrightarrow -\ pi \leq x \Longrightarrow arccos\ (cos\ x) = -x$
by (*auto simp*: *arccos-def intro*!: *the1-equality cos-total*)

lemma *arccos-unique*:
assumes $0 \leq x$ **and** $x \leq pi$ **and** $cos\ x = y$ **shows** $arccos\ y = x$
using *arccos-cos assms* **by** *blast*

lemma *cos-arcsin*:
assumes $-\ 1 \leq x\ x \leq 1$
shows $cos\ (arcsin\ x) = sqrt\ (1 - x^2)$
proof (*rule power2-eq-imp-eq*)
show $(cos\ (arcsin\ x))^2 = (sqrt\ (1 - x^2))^2$
by (*simp add*: *square-le-1 assms cos-squared-eq*)
show $0 \leq cos\ (arcsin\ x)$
using *arcsin assms cos-ge-zero* **by** *blast*
show $0 \leq sqrt\ (1 - x^2)$
by (*simp add*: *square-le-1 assms*)
qed

lemma *sin-arccos*:
assumes $-\ 1 \leq x\ x \leq 1$
shows $sin\ (arccos\ x) = sqrt\ (1 - x^2)$
proof (*rule power2-eq-imp-eq*)
show $(sin\ (arccos\ x))^2 = (sqrt\ (1 - x^2))^2$
by (*simp add*: *square-le-1 assms sin-squared-eq*)
show $0 \leq sin\ (arccos\ x)$
by (*simp add*: *arccos-bounded assms sin-ge-zero*)
show $0 \leq sqrt\ (1 - x^2)$
by (*simp add*: *square-le-1 assms*)
qed

lemma *arccos-0* [*simp*]: $arccos\ 0 = pi/2$
using *arccos-cos pi-half-ge-zero* **by** *fastforce*

lemma *arccos-1* [*simp*]: $arccos\ 1 = 0$
using *arccos-cos* **by** *force*

lemma *arccos-minus-1* [*simp*]: $arccos\ (-\ 1) = pi$
by (*metis arccos-cos cos-pi order-refl pi-ge-zero*)

lemma *arccos-minus*: $-1 \leq x \Longrightarrow x \leq 1 \Longrightarrow arccos\ (-\ x) = pi - arccos\ x$
by (*smt* (*verit*, *ccfv-threshold*) *arccos arccos-cos cos-minus cos-minus-pi*)

lemma *arccos-one-half* [*simp*]: $arccos\ (1/2) = pi\ /\ 3$
and *arccos-minus-one-half* [*simp*]: $arccos\ (-(1/2)) = 2 * pi\ /\ 3$

by (*intro arccos-unique*; *simp add*: *cos-60 cos-120*)+

lemma *arccos-one-over-sqrt-2*: *arccos* (*1 / sqrt 2*) = *pi / 4*
by (*rule arccos-unique*) (*auto simp*: *cos-45 field-simps*)

corollary *arccos-minus-abs*:
assumes $|x| \leq 1$
shows *arccos* (− *x*) = *pi* − *arccos x*
using *assms* **by** (*simp add*: *arccos-minus*)

lemma *sin-arccos-nonzero*: − *1* < *x* ⟹ *x* < *1* ⟹ *sin* (*arccos x*) ≠ *0*
using *arccos-lt-bounded sin-gt-zero* **by** *force*

lemma *arctan*: − (*pi/2*) < *arctan y* ∧ *arctan y* < *pi/2* ∧ *tan* (*arctan y*) = *y*
unfolding *arctan-def* **by** (*rule theI'* [*OF tan-total*])

lemma *tan-arctan*: *tan* (*arctan y*) = *y*
by (*simp add*: *arctan*)

lemma *arctan-bounded*: − (*pi/2*) < *arctan y* ∧ *arctan y* < *pi/2*
by (*auto simp only*: *arctan*)

lemma *arctan-lbound*: − (*pi/2*) < *arctan y*
by (*simp add*: *arctan*)

lemma *arctan-ubound*: *arctan y* < *pi/2*
by (*auto simp only*: *arctan*)

lemma *arctan-unique*:
assumes −(*pi/2*) < *x*
and *x* < *pi/2*
and *tan x* = *y*
shows *arctan y* = *x*
using *assms arctan* [*of y*] *tan-total* [*of y*] **by** (*fast elim*: *ex1E*)

lemma *arctan-tan*: −(*pi/2*) < *x* ⟹ *x* < *pi/2* ⟹ *arctan* (*tan x*) = *x*
by (*rule arctan-unique*) *simp-all*

lemma *arctan-zero-zero* [*simp*]: *arctan 0* = *0*
by (*rule arctan-unique*) *simp-all*

lemma *arctan-minus*: *arctan* (− *x*) = − *arctan x*
using *arctan* [*of x*] **by** (*auto simp*: *arctan-unique*)

lemma *cos-arctan-not-zero* [*simp*]: *cos* (*arctan x*) ≠ *0*
by (*intro less-imp-neq* [*symmetric*] *cos-gt-zero-pi arctan-lbound arctan-ubound*)

lemma *tan-eq-arctan-Ex*:
shows *tan x* = *y* ⟷ (∃ *k*::*int*. *x* = *arctan y* + *k*∗*pi* ∨ (*x* = *pi/2* + *k*∗*pi* ∧

y=0))
proof
assume *lhs*: *tan x = y*
obtain *k*::*int* **where** *k*:−*pi/2 < x*−*k*∗*pi x*−*k*∗*pi ≤ pi/2*
proof
define *k* **where** *k ≡ ceiling* (*x/pi − 1/2*)
show *− pi / 2 < x − real-of-int k ∗ pi*
using *ceiling-divide-lower* [*of pi∗2* (*x ∗ 2 − pi*)] **by** (*auto simp*: *k-def field-simps*)
show *x*−*k*∗*pi ≤ pi/2*
using *ceiling-divide-upper* [*of pi∗2* (*x ∗ 2 − pi*)] **by** (*auto simp*: *k-def field-simps*)
qed
have *x = arctan y + of-int k ∗ pi* **when** *x ≠ pi/2 + k∗pi*
proof −
have *tan* (*x − k ∗ pi*) *= y* **using** *lhs tan-periodic-int*[*of - −k*] **by** *auto*
then have *arctan y = x − real-of-int k ∗ pi*
by (*smt* (*verit*) *arctan-tan lhs divide-minus-left k mult-minus-left of-int-minus tan-periodic-int that*)
then show *?thesis* **by** *auto*
qed
then show ∃ *k*. *x = arctan y + of-int k ∗ pi* ∨ (*x = pi/2 + k∗pi* ∧ *y=0*)
using *lhs k* **by** *force*
qed (*auto simp*: *arctan*)

lemma *arctan-tan-eq-abs-pi*:
assumes *cos ϑ ≠ 0*
obtains *k* **where** *arctan* (*tan ϑ*) *= ϑ − of-int k ∗ pi*
by (*metis add.commute assms cos-zero-iff-int2 eq-diff-eq tan-eq-arctan-Ex*)

lemma *tan-eq*:
assumes *tan x = tan y tan x ≠ 0*
obtains *k*::*int* **where** *x = y + k ∗ pi*
proof −
obtain *k0* **where** *k0*: *x = arctan* (*tan y*) *+ real-of-int k0 ∗ pi*
using *assms tan-eq-arctan-Ex*[*of x tan y*] **by** *auto*
obtain *k1* **where** *k1*: *arctan* (*tan y*) *= y − of-int k1 ∗ pi*
using *arctan-tan-eq-abs-pi assms tan-eq-0-cos-sin* **by** *auto*
have *x = y + (k0−k1)∗pi*
using *k0 k1* **by** (*auto simp*: *algebra-simps*)
with *that* **show** *?thesis*
by *blast*
qed

lemma *cos-arctan*: *cos* (*arctan x*) *= 1 / sqrt* (*1 + x*^{2})
proof (*rule power2-eq-imp-eq*)
have *0 < 1 + x*^{2} **by** (*simp add*: *add-pos-nonneg*)
show *0 ≤ 1 / sqrt* (*1 + x*^{2}) **by** *simp*
show *0 ≤ cos* (*arctan x*)

```
    by (intro less-imp-le cos-gt-zero-pi arctan-lbound arctan-ubound)
  have (cos (arctan x))^2 * (1 + (tan (arctan x))^2) = 1
    unfolding tan-def by (simp add: distrib-left power-divide)
  then show (cos (arctan x))^2 = (1 / sqrt (1 + x^2))^2
    using ‹0 < 1 + x^2› by (simp add: arctan power-divide eq-divide-eq)
qed

lemma sin-arctan: sin (arctan x) = x / sqrt (1 + x^2)
  using add-pos-nonneg [OF zero-less-one zero-le-power2 [of x]]
  using tan-arctan [of x] unfolding tan-def cos-arctan
  by (simp add: eq-divide-eq)

lemma tan-sec: cos x ≠ 0 ⟹ 1 + (tan x)^2 = (inverse (cos x))^2
  for x :: 'a::{real-normed-field,banach,field}
  by (simp add: add-divide-eq-iff inverse-eq-divide power2-eq-square tan-def)

lemma arctan-less-iff: arctan x < arctan y ⟷ x < y
  by (metis tan-monotone' arctan-lbound arctan-ubound tan-arctan)

lemma arctan-le-iff: arctan x ≤ arctan y ⟷ x ≤ y
  by (simp only: not-less [symmetric] arctan-less-iff)

lemma arctan-eq-iff: arctan x = arctan y ⟷ x = y
  by (simp only: eq-iff [where 'a=real] arctan-le-iff)

lemma zero-less-arctan-iff [simp]: 0 < arctan x ⟷ 0 < x
  using arctan-less-iff [of 0 x] by simp

lemma arctan-less-zero-iff [simp]: arctan x < 0 ⟷ x < 0
  using arctan-less-iff [of x 0] by simp

lemma zero-le-arctan-iff [simp]: 0 ≤ arctan x ⟷ 0 ≤ x
  using arctan-le-iff [of 0 x] by simp

lemma arctan-le-zero-iff [simp]: arctan x ≤ 0 ⟷ x ≤ 0
  using arctan-le-iff [of x 0] by simp

lemma arctan-eq-zero-iff [simp]: arctan x = 0 ⟷ x = 0
  using arctan-eq-iff [of x 0] by simp

lemma continuous-on-arcsin': continuous-on {-1 .. 1} arcsin
proof -
  have continuous-on (sin ` {- pi/2 .. pi/2}) arcsin
    by (rule continuous-on-inv) (auto intro: continuous-intros simp: arcsin-sin)
  also have sin ` {- pi/2 .. pi/2} = {-1 .. 1}
  proof safe
    fix x :: real
    assume x ∈ {-1..1}
    then show x ∈ sin ` {- pi/2..pi/2}
```

using *arcsin-lbound arcsin-ubound*
by (*intro image-eqI*[**where** *x*=*arcsin x*]) *auto*
qed *simp*
finally show *?thesis* **.**
qed

lemma *continuous-on-arcsin* [*continuous-intros*]:
continuous-on s f ⟹ (∀ *x*∈*s*. −*1* ≤ *f x* ∧ *f x* ≤ *1*) ⟹ *continuous-on s* (λ*x*. *arcsin* (*f x*))
using *continuous-on-compose*[*of s f*, *OF* - *continuous-on-subset*[*OF* *continuous-on-arcsin'*]]
by (*auto simp*: *comp-def subset-eq*)

lemma *isCont-arcsin*: −*1* < *x* ⟹ *x* < *1* ⟹ *isCont arcsin x*
using *continuous-on-arcsin'*[*THEN continuous-on-subset*, *of* { −*1* <..< *1* }]
by (*auto simp*: *continuous-on-eq-continuous-at subset-eq*)

lemma *continuous-on-arccos'*: *continuous-on* {−*1* .. *1*} *arccos*
proof −
have *continuous-on* (*cos* ' {*0* .. *pi*}) *arccos*
by (*rule continuous-on-inv*) (*auto intro*: *continuous-intros simp*: *arccos-cos*)
also have *cos* ' {*0* .. *pi*} = {−*1* .. *1*}
proof *safe*
fix *x* :: *real*
assume *x* ∈ {−*1*..*1*}
then show *x* ∈ *cos* ' {*0*..*pi*}
using *arccos-lbound arccos-ubound*
by (*intro image-eqI*[**where** *x*=*arccos x*]) *auto*
qed *simp*
finally show *?thesis* **.**
qed

lemma *continuous-on-arccos* [*continuous-intros*]:
continuous-on s f ⟹ (∀ *x*∈*s*. −*1* ≤ *f x* ∧ *f x* ≤ *1*) ⟹ *continuous-on s* (λ*x*. *arccos* (*f x*))
using *continuous-on-compose*[*of s f*, *OF* - *continuous-on-subset*[*OF* *continuous-on-arccos'*]]
by (*auto simp*: *comp-def subset-eq*)

lemma *isCont-arccos*: −*1* < *x* ⟹ *x* < *1* ⟹ *isCont arccos x*
using *continuous-on-arccos'*[*THEN continuous-on-subset*, *of* { −*1* <..< *1* }]
by (*auto simp*: *continuous-on-eq-continuous-at subset-eq*)

lemma *isCont-arctan*: *isCont arctan x*
proof −
obtain *u* **where** *u*: − (*pi*/*2*) < *u* *u* < *arctan x*
by (*meson arctan arctan-less-iff linordered-field-no-lb*)
obtain *v* **where** *v*: *arctan x* < *v* *v* < *pi*/*2*
by (*meson arctan-less-iff arctan-ubound linordered-field-no-ub*)

have *isCont arctan* (*tan* (*arctan x*))
proof (*rule isCont-inverse-function2* [*of u arctan x v*])
show $\bigwedge z.\ [\![u \leq z;\ z \leq v]\!] \Longrightarrow$ *arctan* (*tan z*) = *z*
using *arctan-unique u*(*1*) *v*(*2*) **by** *auto*
then show $\bigwedge z.\ [\![u \leq z;\ z \leq v]\!] \Longrightarrow$ *isCont tan z*
by (*metis arctan cos-gt-zero-pi isCont-tan less-irrefl*)
qed (*use u v* **in** *auto*)
then show *?thesis*
by (*simp add*: *arctan*)
qed

lemma *tendsto-arctan* [*tendsto-intros*]: (*f* $\longrightarrow$ *x*) *F* $\Longrightarrow$ ((λ*x*. *arctan* (*f x*)) $\longrightarrow$ *arctan x*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-arctan*])

lemma *continuous-arctan* [*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* (λ*x*. *arctan* (*f x*))
unfolding *continuous-def* **by** (*rule tendsto-arctan*)

lemma *continuous-on-arctan* [*continuous-intros*]:
continuous-on s f $\Longrightarrow$ *continuous-on s* (λ*x*. *arctan* (*f x*))
unfolding *continuous-on-def* **by** (*auto intro*: *tendsto-arctan*)

lemma *DERIV-arcsin*:
assumes − *1* < *x x* < *1*
shows *DERIV arcsin x* :> *inverse* (*sqrt* (*1* − x^2))
proof (*rule DERIV-inverse-function*)
show (*sin has-real-derivative sqrt* (*1* − x^2)) (*at* (*arcsin x*))
by (*rule derivative-eq-intros* | *use assms cos-arcsin* **in** *force*)+
show *sqrt* (*1* − x^2) ≠ *0*
using *abs-square-eq-1 assms* **by** *force*
qed (*use assms isCont-arcsin* **in** *auto*)

lemma *DERIV-arccos*:
assumes − *1* < *x x* < *1*
shows *DERIV arccos x* :> *inverse* (− *sqrt* (*1* − x^2))
proof (*rule DERIV-inverse-function*)
show (*cos has-real-derivative* − *sqrt* (*1* − x^2)) (*at* (*arccos x*))
by (*rule derivative-eq-intros* | *use assms sin-arccos* **in** *force*)+
show − *sqrt* (*1* − x^2) ≠ *0*
using *abs-square-eq-1 assms* **by** *force*
qed (*use assms isCont-arccos* **in** *auto*)

lemma *DERIV-arctan*: *DERIV arctan x* :> *inverse* (*1* + x^2)
proof (*rule DERIV-inverse-function*)
have *inverse* ((*cos* (*arctan x*))^2) = *1* + x^2
by (*metis arctan cos-arctan-not-zero power-inverse tan-sec*)
then show (*tan has-real-derivative 1* + x^2) (*at* (*arctan x*))
by (*auto intro*!: *derivative-eq-intros*)

show $\bigwedge y.\ [\![x - 1 < y;\ y < x + 1]\!] \Longrightarrow tan\ (arctan\ y) = y$
using *tan-arctan* **by** *blast*
show $1 + x^2 \neq 0$
by (*metis power-one sum-power2-eq-zero-iff zero-neq-one*)
qed (*use isCont-arctan* **in** *auto*)

declare
DERIV-arcsin[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arcsin[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]
DERIV-arccos[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arccos[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]
DERIV-arctan[*THEN DERIV-chain2*, *derivative-intros*]
DERIV-arctan[*THEN DERIV-chain2*, *unfolded has-field-derivative-def*, *derivative-intros*]

lemmas *has-derivative-arctan*[*derivative-intros*] = *DERIV-arctan*[*THEN DERIV-compose-FDERIV*]
and *has-derivative-arccos*[*derivative-intros*] = *DERIV-arccos*[*THEN DERIV-compose-FDERIV*]
and *has-derivative-arcsin*[*derivative-intros*] = *DERIV-arcsin*[*THEN DERIV-compose-FDERIV*]

lemma *filterlim-tan-at-right*: *filterlim tan at-bot* (*at-right* (− (*pi*/2)))
by (*rule filterlim-at-bot-at-right*[**where** *Q*=λ*x*. − *pi*/2 < *x* ∧ *x* < *pi*/2 **and** *P*=λ*x*. *True* **and** *g*=*arctan*])
(*auto simp*: *arctan le-less eventually-at dist-real-def simp del*: *less-divide-eq-numeral1*
intro!: *tan-monotone exI*[*of* - *pi*/2])

lemma *filterlim-tan-at-left*: *filterlim tan at-top* (*at-left* (*pi*/2))
by (*rule filterlim-at-top-at-left*[**where** *Q*=λ*x*. − *pi*/2 < *x* ∧ *x* < *pi*/2 **and** *P*=λ*x*. *True* **and** *g*=*arctan*])
(*auto simp*: *arctan le-less eventually-at dist-real-def simp del*: *less-divide-eq-numeral1*
intro!: *tan-monotone exI*[*of* - *pi*/2])

lemma *tendsto-arctan-at-top*: (*arctan* ⟶ (*pi*/2)) *at-top*
proof (*rule tendstoI*)
fix *e* :: *real*
assume 0 < *e*
define *y* **where** *y* = *pi*/2 − *min* (*pi*/2) *e*
then have *y*: 0 ≤ *y* *y* < *pi*/2 *pi*/2 ≤ *e* + *y*
using ‹0 < *e*› **by** *auto*
show *eventually* (λ*x*. *dist* (*arctan x*) (*pi*/2) < *e*) *at-top*
proof (*intro eventually-at-top-dense*[*THEN iffD2*] *exI allI impI*)
fix *x*
assume *tan y* < *x*
then have *arctan* (*tan y*) < *arctan x*
by (*simp add*: *arctan-less-iff*)
with *y* **have** *y* < *arctan x*
by (*subst* (*asm*) *arctan-tan*) *simp-all*
with *arctan-ubound*[*of x*, *arith*] *y* ‹0 < *e*›

```
    show dist (arctan x) (pi/2) < e
      by (simp add: dist-real-def)
  qed
qed

lemma tendsto-arctan-at-bot: (arctan ⟶ − (pi/2)) at-bot
  unfolding filterlim-at-bot-mirror arctan-minus
  by (intro tendsto-minus tendsto-arctan-at-top)

lemma sin-multiple-reduce:
  sin (x * numeral n :: 'a :: {real-normed-field, banach}) =
    sin x * cos (x * of-nat (pred-numeral n)) + cos x * sin (x * of-nat (pred-numeral
n))
proof −
  have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
    by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
  also have sin (x * ...) = sin (x * of-nat (pred-numeral n) + x)
    unfolding of-nat-Suc by (simp add: ring-distribs)
  finally show ?thesis
    by (simp add: sin-add)
qed

lemma cos-multiple-reduce:
  cos (x * numeral n :: 'a :: {real-normed-field, banach}) =
     cos (x * of-nat (pred-numeral n)) * cos x − sin (x * of-nat (pred-numeral n))
* sin x
proof −
  have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
    by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
  also have cos (x * ...) = cos (x * of-nat (pred-numeral n) + x)
    unfolding of-nat-Suc by (simp add: ring-distribs)
  finally show ?thesis
    by (simp add: cos-add)
qed

lemma arccos-eq-pi-iff: x ∈ {−1..1} ⟹ arccos x = pi ⟷ x = −1
  by (metis arccos arccos-minus-1 atLeastAtMost-iff cos-pi)

lemma arccos-eq-0-iff: x ∈ {−1..1} ⟹ arccos x = 0 ⟷ x = 1
  by (metis arccos arccos-1 atLeastAtMost-iff cos-zero)
```

112.17 Prove Totality of the Trigonometric Functions

```
lemma cos-arccos-abs: |y| ≤ 1 ⟹ cos (arccos y) = y
  by (simp add: abs-le-iff)

lemma sin-arccos-abs: |y| ≤ 1 ⟹ sin (arccos y) = sqrt (1 − y²)
  by (simp add: sin-arccos abs-le-iff)
```

lemma *sin-mono-less-eq*:
$-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow -(pi/2) \leq y \Longrightarrow y \leq pi/2 \Longrightarrow sin\ x < sin\ y \longleftrightarrow x < y$
by (*metis not-less-iff-gr-or-eq sin-monotone-2pi*)

lemma *sin-mono-le-eq*:
$-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow -(pi/2) \leq y \Longrightarrow y \leq pi/2 \Longrightarrow sin\ x \leq sin\ y \longleftrightarrow x \leq y$
by (*meson leD le-less-linear sin-monotone-2pi sin-monotone-2pi-le*)

lemma *sin-inj-pi*:
$-(pi/2) \leq x \Longrightarrow x \leq pi/2 \Longrightarrow -(pi/2) \leq y \Longrightarrow y \leq pi/2 \Longrightarrow sin\ x = sin\ y \Longrightarrow x = y$
by (*metis arcsin-sin*)

lemma *arcsin-le-iff*:
assumes $x \geq -1\ x \leq 1\ y \geq -pi/2\ y \leq pi/2$
shows $arcsin\ x \leq y \longleftrightarrow x \leq sin\ y$
proof −
have $arcsin\ x \leq y \longleftrightarrow sin\ (arcsin\ x) \leq sin\ y$
using *arcsin-bounded*[*of x*] *assms* **by** (*subst sin-mono-le-eq*) *auto*
also from *assms* **have** $sin\ (arcsin\ x) = x$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *le-arcsin-iff*:
assumes $x \geq -1\ x \leq 1\ y \geq -pi/2\ y \leq pi/2$
shows $arcsin\ x \geq y \longleftrightarrow x \geq sin\ y$
proof −
have $arcsin\ x \geq y \longleftrightarrow sin\ (arcsin\ x) \geq sin\ y$
using *arcsin-bounded*[*of x*] *assms* **by** (*subst sin-mono-le-eq*) *auto*
also from *assms* **have** $sin\ (arcsin\ x) = x$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *cos-mono-less-eq*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x < cos\ y \longleftrightarrow y < x$
by (*meson cos-monotone-0-pi cos-monotone-0-pi-le leD le-less-linear*)

lemma *cos-mono-le-eq*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x \leq cos\ y \longleftrightarrow y \leq x$
by (*metis arccos-cos cos-monotone-0-pi-le eq-iff linear*)

lemma *cos-inj-pi*: $0 \leq x \Longrightarrow x \leq pi \Longrightarrow 0 \leq y \Longrightarrow y \leq pi \Longrightarrow cos\ x = cos\ y \Longrightarrow x = y$
by (*metis arccos-cos*)

lemma *arccos-le-pi2*: $[\![0 \leq y;\ y \leq 1]\!] \Longrightarrow arccos\ y \leq pi/2$
by (*metis* (*mono-tags*) *arccos-0 arccos cos-le-one cos-monotone-0-pi-le*

cos-pi cos-pi-half pi-half-ge-zero antisym-conv less-eq-neg-nonpos linear minus-minus order.trans order-refl)

lemma *sincos-total-pi-half*:
assumes $0 \leq x$ $0 \leq y$ $x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq pi/2 \wedge x = cos\ t \wedge y = sin\ t$
proof −
have *x1*: $x \leq 1$
using *assms* **by** (*metis le-add-same-cancel1 power2-le-imp-le power-one zero-le-power2*)
with *assms* **have** ∗: $0 \leq arccos\ x$ $cos\ (arccos\ x) = x$
by (*auto simp*: *arccos*)
from *assms* **have** $y = sqrt\ (1 - x^2)$
by (*metis abs-of-nonneg add.commute add-diff-cancel real-sqrt-abs*)
with *x1* ∗ *assms arccos-le-pi2* [*of x*] **show** *?thesis*
by (*rule-tac x*=*arccos x* **in** *exI*) (*auto simp*: *sin-arccos*)
qed

lemma *sincos-total-pi*:
assumes $0 \leq y$ $x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq pi \wedge x = cos\ t \wedge y = sin\ t$
proof (*cases rule*: *le-cases* [*of 0 x*])
case *le*
from *sincos-total-pi-half* [*OF le*] **show** *?thesis*
by (*metis pi-ge-two pi-half-le-two add.commute add-le-cancel-left add-mono assms*)
next
case *ge*
then have $0 \leq -x$
by *simp*
then obtain *t* **where** *t*: $t \geq 0$ $t \leq pi/2$ $-x = cos\ t$ $y = sin\ t$
using *sincos-total-pi-half assms*
by *auto* (*metis* ‹$0 \leq -x$› *power2-minus*)
show *?thesis*
by (*rule exI* [**where** $x = pi - t$]) (*use t* **in** *auto*)
qed

lemma *sincos-total-2pi-le*:
assumes $x^2 + y^2 = 1$
shows $\exists t.\ 0 \leq t \wedge t \leq 2 * pi \wedge x = cos\ t \wedge y = sin\ t$
proof (*cases rule*: *le-cases* [*of 0 y*])
case *le*
from *sincos-total-pi* [*OF le*] **show** *?thesis*
by (*metis assms le-add-same-cancel1 mult.commute mult-2-right order.trans*)
next
case *ge*
then have $0 \leq -y$
by *simp*
then obtain *t* **where** *t*: $t \geq 0$ $t \leq pi$ $x = cos\ t$ $-y = sin\ t$
using *sincos-total-pi assms*

by *auto* (*metis* ‹*0* ≤ − *y*› *power2-minus*)
show *?thesis*
by (*rule exI* [**where** *x* = *2* ∗ *pi* − *t*]) (*use t* **in** *auto*)
qed

lemma *sincos-total-2pi*:
assumes $x^2 + y^2 = 1$
obtains *t* **where** *0* ≤ *t t* < *2*∗*pi x* = *cos t y* = *sin t*
proof −
from *sincos-total-2pi-le* [*OF assms*]
obtain *t* **where** *t*: *0* ≤ *t t* ≤ *2*∗*pi x* = *cos t y* = *sin t*
by *blast*
show *?thesis*
by (*cases t* = *2* ∗ *pi*) (*use t that* **in** ‹*force*+›)
qed

lemma *arcsin-less-mono*: |*x*| ≤ *1* ⟹ |*y*| ≤ *1* ⟹ *arcsin x* < *arcsin y* ⟷ *x* < *y*
by (*rule trans* [*OF sin-mono-less-eq* [*symmetric*]]) (*use arcsin-ubound arcsin-lbound*
in *auto*)

lemma *arcsin-le-mono*: |*x*| ≤ *1* ⟹ |*y*| ≤ *1* ⟹ *arcsin x* ≤ *arcsin y* ⟷ *x* ≤ *y*
using *arcsin-less-mono not-le* **by** *blast*

lemma *arcsin-less-arcsin*: − *1* ≤ *x* ⟹ *x* < *y* ⟹ *y* ≤ *1* ⟹ *arcsin x* < *arcsin y*
using *arcsin-less-mono* **by** *auto*

lemma *arcsin-le-arcsin*: − *1* ≤ *x* ⟹ *x* ≤ *y* ⟹ *y* ≤ *1* ⟹ *arcsin x* ≤ *arcsin y*
using *arcsin-le-mono* **by** *auto*

lemma *arcsin-nonneg*: *x* ∈ {*0*..*1*} ⟹ *arcsin x* ≥ *0*
using *arcsin-le-arcsin*[*of 0 x*] **by** *simp*

lemma *arccos-less-mono*: |*x*| ≤ *1* ⟹ |*y*| ≤ *1* ⟹ *arccos x* < *arccos y* ⟷ *y* < *x*
by (*rule trans* [*OF cos-mono-less-eq* [*symmetric*]]) (*use arccos-ubound arccos-lbound*
in *auto*)

lemma *arccos-le-mono*: |*x*| ≤ *1* ⟹ |*y*| ≤ *1* ⟹ *arccos x* ≤ *arccos y* ⟷ *y* ≤ *x*
using *arccos-less-mono* [*of y x*] **by** (*simp add*: *not-le* [*symmetric*])

lemma *arccos-less-arccos*: − *1* ≤ *x* ⟹ *x* < *y* ⟹ *y* ≤ *1* ⟹ *arccos y* < *arccos x*
using *arccos-less-mono* **by** *auto*

lemma *arccos-le-arccos*: − *1* ≤ *x* ⟹ *x* ≤ *y* ⟹ *y* ≤ *1* ⟹ *arccos y* ≤ *arccos x*
using *arccos-le-mono* **by** *auto*

lemma *arccos-eq-iff*: |*x*| ≤ *1* ∧ |*y*| ≤ *1* ⟹ *arccos x* = *arccos y* ⟷ *x* = *y*
using *cos-arccos-abs* **by** *fastforce*

lemma *arccos-cos-eq-abs*:
assumes $|\vartheta| \leq pi$
shows *arccos* (*cos* ϑ) = $|\vartheta|$
unfolding *arccos-def*
proof (*intro the-equality conjI*; *clarify?*)
show *cos* $|\vartheta|$ = *cos* ϑ
by (*simp add*: *abs-real-def*)
show $x = |\vartheta|$ **if** *cos* x = *cos* ϑ $0 \leq x$ $x \leq pi$ **for** x
by (*simp add*: ‹*cos* $|\vartheta|$ = *cos* ϑ› *assms cos-inj-pi that*)
qed (*use assms* **in** *auto*)

lemma *arccos-cos-eq-abs-2pi*:
obtains k **where** *arccos* (*cos* ϑ) = $|\vartheta$ − *of-int* k ∗ (2 ∗ *pi*)|
proof −
define k **where** $k \equiv \lfloor(\vartheta + pi)$ / (2 ∗ *pi*)$\rfloor$
have *lepi*: $|\vartheta$ − *of-int* k ∗ (2 ∗ *pi*)| ≤ *pi*
using *floor-divide-lower* [*of* 2∗*pi* ϑ + *pi*] *floor-divide-upper* [*of* 2∗*pi* ϑ + *pi*]
by (*auto simp*: *k-def abs-if algebra-simps*)
have *arccos* (*cos* ϑ) = *arccos* (*cos* (ϑ − *of-int* k ∗ (2 ∗ *pi*)))
using *cos-int-2pin sin-int-2pin* **by** (*simp add*: *cos-diff mult.commute*)
also have ... = $|\vartheta$ − *of-int* k ∗ (2 ∗ *pi*)|
using *arccos-cos-eq-abs lepi* **by** *blast*
finally show *?thesis*
using *that* **by** *metis*
qed

lemma *arccos-arctan*:
assumes −1 < x x < 1
shows *arccos* x = *pi*/2 − *arctan*(x / *sqrt*(1 − x^2))
proof −
have *arctan*(x / *sqrt*(1 − x^2)) − (*pi*/2 − *arccos* x) = 0
proof (*rule sin-eq-0-pi*)
show − *pi* < *arctan* (x / *sqrt* (1 − x^2)) − (*pi*/2 − *arccos* x)
using *arctan-lbound* [*of* x / *sqrt*(1 − x^2)] *arccos-bounded* [*of* x] *assms*
by (*simp add*: *algebra-simps*)
next
show *arctan* (x / *sqrt* (1 − x^2)) − (*pi*/2 − *arccos* x) < *pi*
using *arctan-ubound* [*of* x / *sqrt*(1 − x^2)] *arccos-bounded* [*of* x] *assms*
by (*simp add*: *algebra-simps*)
next
show *sin* (*arctan* (x / *sqrt* (1 − x^2)) − (*pi*/2 − *arccos* x)) = 0
using *assms*
by (*simp add*: *algebra-simps sin-diff cos-add sin-arccos sin-arctan cos-arctan*
power2-eq-square square-eq-1-iff)
qed
then show *?thesis*
by *simp*
qed

lemma *arcsin-plus-arccos*:
assumes $-1 \leq x\ x \leq 1$
shows $\arcsin\ x + \arccos\ x = pi/2$
proof −
have $\arcsin\ x = pi/2 - \arccos\ x$
apply (*rule sin-inj-pi*)
using *assms arcsin* [*OF assms*] *arccos* [*OF assms*]
by (*auto simp*: *algebra-simps sin-diff*)
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

lemma *arcsin-arccos-eq*: $-1 \leq x \Longrightarrow x \leq 1 \Longrightarrow \arcsin\ x = pi/2 - \arccos\ x$
using *arcsin-plus-arccos* **by** *force*

lemma *arccos-arcsin-eq*: $-1 \leq x \Longrightarrow x \leq 1 \Longrightarrow \arccos\ x = pi/2 - \arcsin\ x$
using *arcsin-plus-arccos* **by** *force*

lemma *arcsin-arctan*: $-1 < x \Longrightarrow x < 1 \Longrightarrow \arcsin\ x = \arctan(x\ /\ sqrt(1 - x^2))$
by (*simp add*: *arccos-arctan arcsin-arccos-eq*)

lemma *arcsin-arccos-sqrt-pos*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow \arcsin\ x = \arccos(sqrt(1 - x^2))$
by (*smt* (*verit*, *del-insts*) *arccos-cos arcsin-0 arcsin-le-arcsin arcsin-pi cos-arcsin*)

lemma *arcsin-arccos-sqrt-neg*: $-1 \leq x \Longrightarrow x \leq 0 \Longrightarrow \arcsin\ x = -\arccos(sqrt(1 - x^2))$
using *arcsin-arccos-sqrt-pos* [*of* $-x$]
by (*simp add*: *arcsin-minus*)

lemma *arccos-arcsin-sqrt-pos*: $0 \leq x \Longrightarrow x \leq 1 \Longrightarrow \arccos\ x = \arcsin(sqrt(1 - x^2))$
by (*smt* (*verit*, *del-insts*) *arccos-lbound arccos-le-pi2 arcsin-sin sin-arccos*)

lemma *arccos-arcsin-sqrt-neg*: $-1 \leq x \Longrightarrow x \leq 0 \Longrightarrow \arccos\ x = pi - \arcsin(sqrt(1 - x^2))$
using *arccos-arcsin-sqrt-pos* [*of* $-x$]
by (*simp add*: *arccos-minus*)

lemma *cos-limit-1*:
assumes $(\lambda j.\ \cos\ (\vartheta\ j)) \longrightarrow 1$
shows $\exists k.\ (\lambda j.\ \vartheta\ j - \textit{of-int}\ (k\ j) * (2 * pi)) \longrightarrow 0$
proof −
have $\forall_F\ j$ *in sequentially*. $\cos\ (\vartheta\ j) \in \{-1..1\}$
by *auto*
then have $(\lambda j.\ \arccos\ (\cos\ (\vartheta\ j))) \longrightarrow \arccos\ 1$
using *continuous-on-tendsto-compose* [*OF continuous-on-arccos′ assms*] **by** *auto*
moreover have $\bigwedge j.\ \exists k.\ \arccos\ (\cos\ (\vartheta\ j)) = |\vartheta\ j - \textit{of-int}\ k * (2 * pi)|$
using *arccos-cos-eq-abs-2pi* **by** *metis*

then have $\exists k.\ \forall j.\ arccos\ (cos\ (\vartheta\ j)) = |\vartheta\ j - of\text{-}int\ (k\ j) * (2 * pi)|$
by *metis*
ultimately have $\exists k.\ (\lambda j.\ |\vartheta\ j - of\text{-}int\ (k\ j) * (2 * pi)|) \longrightarrow 0$
by *auto*
then show *?thesis*
by (*simp add*: *tendsto-rabs-zero-iff*)
qed

lemma *cos-diff-limit-1*:
assumes $(\lambda j.\ cos\ (\vartheta\ j - \Theta)) \longrightarrow 1$
obtains k **where** $(\lambda j.\ \vartheta\ j - of\text{-}int\ (k\ j) * (2 * pi)) \longrightarrow \Theta$
proof −
obtain k **where** $(\lambda j.\ (\vartheta\ j - \Theta) - of\text{-}int\ (k\ j) * (2 * pi)) \longrightarrow 0$
using *cos-limit-1* [*OF assms*] **by** *auto*
then have $(\lambda j.\ \Theta + ((\vartheta\ j - \Theta) - of\text{-}int\ (k\ j) * (2 * pi))) \longrightarrow \Theta + 0$
by (*rule tendsto-add* [*OF tendsto-const*])
with *that* **show** *?thesis*
by *auto*
qed

112.18 Machin's formula

lemma *arctan-one*: $arctan\ 1 = pi/4$
by (*rule arctan-unique*) (*simp-all add*: *tan-45 m2pi-less-pi*)

lemma *tan-total-pi4*:
assumes $|x| < 1$
shows $\exists z.\ -(pi/4) < z \wedge z < pi/4 \wedge tan\ z = x$
proof
show $-(pi/4) < arctan\ x \wedge arctan\ x < pi/4 \wedge tan\ (arctan\ x) = x$
unfolding *arctan-one* [*symmetric*] *arctan-minus* [*symmetric*]
unfolding *arctan-less-iff*
using *assms* **by** (*auto simp*: *arctan*)
qed

lemma *arctan-add*:
assumes $|x| \leq 1\ |y| < 1$
shows $arctan\ x + arctan\ y = arctan\ ((x + y)\ /\ (1 - x * y))$
proof (*rule arctan-unique* [*symmetric*])
have $-(pi/4) \leq arctan\ x - (pi/4) < arctan\ y$
unfolding *arctan-one* [*symmetric*] *arctan-minus* [*symmetric*]
unfolding *arctan-le-iff arctan-less-iff*
using *assms* **by** *auto*
from *add-le-less-mono* [*OF this*] **show** *1*: $-(pi/2) < arctan\ x + arctan\ y$
by *simp*
have $arctan\ x \leq pi/4\ arctan\ y < pi/4$
unfolding *arctan-one* [*symmetric*]
unfolding *arctan-le-iff arctan-less-iff*
using *assms* **by** *auto*

from *add-le-less-mono* [*OF this*] **show** *2*: *arctan x + arctan y < pi/2*
by *simp*
show *tan* (*arctan x + arctan y*) = (*x + y*) / (*1 − x ∗ y*)
using *cos-gt-zero-pi* [*OF 1 2*] **by** (*simp add*: *arctan tan-add*)
qed

lemma *arctan-double*: |*x*| < *1* ⟹ *2 ∗ arctan x = arctan* ((*2 ∗ x*) / (*1 − x*^{2}))
by (*metis arctan-add linear mult-2 not-less power2-eq-square*)

theorem *machin*: *pi/4 = 4 ∗ arctan* (*1 / 5*) − *arctan* (*1/239*)
proof −
have |*1 / 5*| < (*1* :: *real*)
by *auto*
from *arctan-add*[*OF less-imp-le*[*OF this*] *this*] **have** *2 ∗ arctan* (*1 / 5*) = *arctan* (*5 / 12*)
by *auto*
moreover
have |*5 / 12*| < (*1* :: *real*)
by *auto*
from *arctan-add*[*OF less-imp-le*[*OF this*] *this*] **have** *2 ∗ arctan* (*5 / 12*) = *arctan* (*120 / 119*)
by *auto*
moreover
have |*1*| ≤ (*1*::*real*) **and** |*1/239*| < (*1*::*real*)
by *auto*
from *arctan-add*[*OF this*] **have** *arctan 1 + arctan* (*1/239*) = *arctan* (*120 / 119*)
by *auto*
ultimately have *arctan 1 + arctan* (*1/239*) = *4 ∗ arctan* (*1 / 5*)
by *auto*
then show *?thesis*
unfolding *arctan-one* **by** *algebra*
qed

lemma *machin-Euler*: *5 ∗ arctan* (*1 / 7*) + *2 ∗ arctan* (*3 / 79*) = *pi/4*
proof −
have *17*: |*1 / 7*| < (*1* :: *real*) **by** *auto*
with *arctan-double* **have** *2 ∗ arctan* (*1 / 7*) = *arctan* (*7 / 24*)
by *simp* (*simp add*: *field-simps*)
moreover
have |*7 / 24*| < (*1* :: *real*) **by** *auto*
with *arctan-double* **have** *2 ∗ arctan* (*7 / 24*) = *arctan* (*336 / 527*)
by *simp* (*simp add*: *field-simps*)
moreover
have |*336 / 527*| < (*1* :: *real*) **by** *auto*
from *arctan-add*[*OF less-imp-le*[*OF 17*] *this*]
have *arctan*(*1/7*) + *arctan* (*336 / 527*) = *arctan* (*2879 / 3353*)
by *auto*
ultimately have *I*: *5 ∗ arctan* (*1 / 7*) = *arctan* (*2879 / 3353*) **by** *auto*

have *379*: $|3 / 79| < (1 :: real)$ **by** *auto*
with *arctan-double* **have** *II*: $2 * arctan\ (3 / 79) = arctan\ (237 / 3116)$
by *simp* (*simp add*: *field-simps*)
have $*$: $|2879 / 3353| < (1 :: real)$ **by** *auto*
have $|237 / 3116| < (1 :: real)$ **by** *auto*
from *arctan-add*[*OF less-imp-le*[*OF* $*$] *this*] **have** $arctan\ (2879/3353) + arctan\ (237/3116) = pi/4$
by (*simp add*: *arctan-one*)
with *I II* **show** *?thesis* **by** *auto*
qed

112.19 Introducing the inverse tangent power series

lemma *monoseq-arctan-series*:
fixes $x :: real$
assumes $|x| \leq 1$
shows *monoseq* $(\lambda n.\ 1 / real\ (n * 2 + 1) * x\hat{}(n * 2 + 1))$
(**is** *monoseq ?a*)
proof (*cases* $x = 0$)
case *True*
then show *?thesis* **by** (*auto simp*: *monoseq-def*)
next
case *False*
have $norm\ x \leq 1$ **and** $x \leq 1$ **and** $-1 \leq x$
using *assms* **by** *auto*
show *monoseq ?a*
proof −
have *mono*: $1 / real\ (Suc\ (Suc\ n * 2)) * x\ \hat{}\ Suc\ (Suc\ n * 2) \leq 1 / real\ (Suc\ (n * 2)) * x\ \hat{}\ Suc\ (n * 2)$
if $0 \leq x$ **and** $x \leq 1$ **for** n **and** $x :: real$
proof (*rule mult-mono*)
show $1 / real\ (Suc\ (Suc\ n * 2)) \leq 1 / real\ (Suc\ (n * 2))$
by (*rule frac-le*) *simp-all*
show $0 \leq 1 / real\ (Suc\ (n * 2))$
by *auto*
show $x\ \hat{}\ Suc\ (Suc\ n * 2) \leq x\ \hat{}\ Suc\ (n * 2)$
by (*rule power-decreasing*) (*simp-all add*: ‹$0 \leq x$› ‹$x \leq 1$›)
show $0 \leq x\ \hat{}\ Suc\ (Suc\ n * 2)$
by (*rule zero-le-power*) (*simp add*: ‹$0 \leq x$›)
qed
show *?thesis*
proof (*cases* $0 \leq x$)
case *True*
from *mono*[*OF this* ‹$x \leq 1$›, *THEN allI*]
show *?thesis*
unfolding *Suc-eq-plus1*[*symmetric*] **by** (*rule mono-SucI2*)
next
case *False*
then have $0 \leq -\ x$ **and** $-\ x \leq 1$

```
      using ‹−1 ≤ x› by auto
    from mono[OF this]
    have 1 / real (Suc (Suc n * 2)) * x ^ Suc (Suc n * 2) ≥
       1 / real (Suc (n * 2)) * x ^ Suc (n * 2) for n
      using ‹0 ≤ −x› by auto
    then show ?thesis
      unfolding Suc-eq-plus1[symmetric] by (rule mono-SucI1[OF allI])
  qed
 qed
qed

lemma zeroseq-arctan-series:
 fixes x :: real
 assumes |x| ≤ 1
 shows (λn. 1 / real (n * 2 + 1) * x^(n * 2 + 1)) ⟶ 0
  (is ?a ⟶ 0)
proof (cases x = 0)
 case True
 then show ?thesis by simp
next
 case False
 have norm x ≤ 1 and x ≤ 1 and −1 ≤ x
  using assms by auto
 show ?a ⟶ 0
 proof (cases |x| < 1)
  case True
  then have norm x < 1 by auto
   from tendsto-mult[OF LIMSEQ-inverse-real-of-nat LIMSEQ-power-zero[OF
‹norm x < 1›, THEN LIMSEQ-Suc]]
  have (λn. 1 / real (n + 1) * x ^ (n + 1)) ⟶ 0
   unfolding inverse-eq-divide Suc-eq-plus1 by simp
  then show ?thesis
   using pos2 by (rule LIMSEQ-linear)
 next
  case False
  then have x = −1 ∨ x = 1
   using ‹|x| ≤ 1› by auto
  then have n-eq: ⋀ n. x ^ (n * 2 + 1) = x
   unfolding One-nat-def by auto
  from tendsto-mult[OF LIMSEQ-inverse-real-of-nat[THEN LIMSEQ-linear, OF
pos2, unfolded inverse-eq-divide] tendsto-const[of x]]
  show ?thesis
   unfolding n-eq Suc-eq-plus1 by auto
 qed
qed

lemma summable-arctan-series:
 fixes n :: nat
 assumes |x| ≤ 1
```

shows *summable* (λ *k*. (−*1*)^*k* ∗ (*1* / *real* (*k*∗*2*+*1*) ∗ *x* ^ (*k*∗*2*+*1*)))
(**is** *summable* (*?c x*))
by (*rule summable-Leibniz*(*1*),
rule zeroseq-arctan-series[*OF assms*],
rule monoseq-arctan-series[*OF assms*])

lemma *DERIV-arctan-series*:
assumes |*x*| < *1*
shows *DERIV* (λ*x'*. ∑ *k*. (−*1*)^*k* ∗ (*1* / *real* (*k* ∗ *2* + *1*) ∗ *x'* ^ (*k* ∗ *2* + *1*)))
x :>
(∑ *k*. (−*1*)^*k* ∗ *x*^(*k* ∗ *2*))
(**is** *DERIV ?arctan - :> ?Int*)
proof −
let *?f* = λ*n*. *if even n then* (−*1*)^(*n div 2*) ∗ *1* / *real* (*Suc n*) *else 0*

have *n-even*: *even n* ⟹ *2* ∗ (*n div 2*) = *n* **for** *n* :: *nat*
by *presburger*
then have *if-eq*: *?f n* ∗ *real* (*Suc n*) ∗ *x'*^*n* =
(*if even n then* (−*1*)^(*n div 2*) ∗ *x'*^(*2* ∗ (*n div 2*)) *else 0*)
for *n x'*
by *auto*

have *summable-Integral*: *summable* (λ *n*. (− *1*) ^ *n* ∗ *x*^(*2* ∗ *n*)) **if** |*x*| < *1* **for**
x :: *real*
proof −
from *that* **have** $x^2 < 1$
by (*simp add*: *abs-square-less-1*)
have *summable* (λ *n*. (− *1*) ^ *n* ∗ (x^2) ^*n*)
by (*rule summable-Leibniz*(*1*))
(*auto intro*!: *LIMSEQ-realpow-zero monoseq-realpow* ‹$x^2 < 1$› *order-less-imp-le*[*OF*
‹$x^2 < 1$›])
then show *?thesis*
by (*simp only*: *power-mult*)
qed

have *sums-even*: (*sums*) *f* = (*sums*) (λ *n*. *if even n then f* (*n div 2*) *else 0*)
for *f* :: *nat* ⇒ *real*
proof −
have *f sums x* = (λ *n*. *if even n then f* (*n div 2*) *else 0*) *sums x* **for** *x* :: *real*
proof
assume *f sums x*
from *sums-if*[*OF sums-zero this*] **show** (λ*n*. *if even n then f* (*n div 2*) *else*
0) *sums x*
by *auto*
next
assume (λ *n*. *if even n then f* (*n div 2*) *else 0*) *sums x*
from *LIMSEQ-linear*[*OF this*[*simplified sums-def*] *pos2*, *simplified sum-split-even-odd*[*simplified*
mult.commute]]
show *f sums x*

```
      unfolding sums-def by auto
    qed
    then show ?thesis ..
  qed

  have Int-eq: (∑ n. ?f n * real (Suc n) * x^n) = ?Int
    unfolding if-eq mult.commute[of - 2]
      suminf-def sums-even[of λ n. (− 1) ^ n * x ^ (2 * n), symmetric]
    by auto

  have arctan-eq: (∑ n. ?f n * x^(Suc n)) = ?arctan x for x
  proof −
    have if-eq': ⋀n. (if even n then (− 1) ^ (n div 2) * 1 / real (Suc n) else 0) *
x ^ Suc n =
      (if even n then (− 1) ^ (n div 2) * (1 / real (Suc (2 * (n div 2))) * x ^ Suc
(2 * (n div 2))) else 0)
      using n-even by auto
    have idx-eq: ⋀n. n * 2 + 1 = Suc (2 * n)
      by auto
    then show ?thesis
      unfolding if-eq' idx-eq suminf-def
        sums-even[of λ n. (− 1) ^ n * (1 / real (Suc (2 * n)) * x ^ Suc (2 * n)),
symmetric]
      by auto
  qed

  have DERIV (λ x. ∑ n. ?f n * x^(Suc n)) x :> (∑ n. ?f n * real (Suc n) *
x^n)
  proof (rule DERIV-power-series')
    show x ∈ {− 1 <..< 1}
      using ‹| x | < 1› by auto
    show summable (λ n. ?f n * real (Suc n) * x'^n)
      if x'-bounds: x' ∈ {− 1 <..< 1} for x' :: real
    proof −
      from that have |x'| < 1 by auto
      then show ?thesis
        using that sums-summable sums-if [OF sums-0 [of λx. 0] summable-sums
[OF summable-Integral]]
        by (auto simp add: if-distrib [of λx. x * y for y] cong: if-cong)
    qed
  qed auto
  then show ?thesis
    by (simp only: Int-eq arctan-eq)
qed

lemma arctan-series:
  assumes |x| ≤ 1
  shows arctan x = (∑ k. (−1)^k * (1 / real (k * 2 + 1) * x ^ (k * 2 + 1)))
    (is - = suminf (λ n. ?c x n))
```

proof −
let *?c′* = λ*x n*. (−*1*)^*n* ∗ *x*^(*n*∗*2*)

have *DERIV-arctan-suminf*: *DERIV* (λ *x*. *suminf* (*?c x*)) *x* :> (*suminf* (*?c′ x*))
if *0* < *r* **and** *r* < *1* **and** |*x*| < *r* **for** *r x* :: *real*
proof (*rule DERIV-arctan-series*)
from *that* **show** |*x*| < *1*
using ‹*r* < *1*› **and** ‹|*x*| < *r*› **by** *auto*
qed

{
fix *x* :: *real*
assume |*x*| ≤ *1*
note *summable-Leibniz*[*OF zeroseq-arctan-series*[*OF this*] *monoseq-arctan-series*[*OF this*]]
} **note** *arctan-series-borders* = *this*

have *when-less-one*: *arctan x* = (∑ *k*. *?c x k*) **if** |*x*| < *1* **for** *x* :: *real*
proof −
obtain *r* **where** |*x*| < *r* **and** *r* < *1*
using *dense*[*OF* ‹|*x*| < *1*›] **by** *blast*
then have *0* < *r* **and** − *r* < *x* **and** *x* < *r* **by** *auto*

have *suminf-eq-arctan-bounded*: *suminf* (*?c x*) − *arctan x* = *suminf* (*?c a*) − *arctan a*
if −*r* < *a* **and** *b* < *r* **and** *a* < *b* **and** *a* ≤ *x* **and** *x* ≤ *b* **for** *x a b*
proof −
from *that* **have** |*x*| < *r* **by** *auto*
show *suminf* (*?c x*) − *arctan x* = *suminf* (*?c a*) − *arctan a*
proof (*rule DERIV-isconst2*[*of a b*])
show *a* < *b* **and** *a* ≤ *x* **and** *x* ≤ *b*
using ‹*a* < *b*› ‹*a* ≤ *x*› ‹*x* ≤ *b*› **by** *auto*
have ∀ *x*. − *r* < *x* ∧ *x* < *r* ⟶ *DERIV* (λ *x*. *suminf* (*?c x*) − *arctan x*) *x* :> *0*
proof (*rule allI*, *rule impI*)
fix *x*
assume −*r* < *x* ∧ *x* < *r*
then have |*x*| < *r* **by** *auto*
with ‹*r* < *1*› **have** |*x*| < *1* **by** *auto*
have |− (*x*^2)| < *1* **using** *abs-square-less-1* ‹|*x*| < *1*› **by** *auto*
then have (λ*n*. (− (*x*^2)) ^ *n*) *sums* (*1* / (*1* − (− (*x*^2))))
unfolding *real-norm-def*[*symmetric*] **by** (*rule geometric-sums*)
then have (*?c′ x*) *sums* (*1* / (*1* − (− (*x*^2))))
unfolding *power-mult-distrib*[*symmetric*] *power-mult mult.commute*[*of - 2*] **by** *auto*
then have *suminf-c′-eq-geom*: *inverse* (*1* + *x*^2) = *suminf* (*?c′ x*)
using *sums-unique* **unfolding** *inverse-eq-divide* **by** *auto*
have *DERIV* (λ *x*. *suminf* (*?c x*)) *x* :> (*inverse* (*1* + *x*^2))
unfolding *suminf-c′-eq-geom*

```
        by (rule DERIV-arctan-suminf[OF ‹0 < r› ‹r < 1› ‹|x| < r›])
      from DERIV-diff [OF this DERIV-arctan] show DERIV (λx. suminf (?c
x) − arctan x) x :> 0
        by auto
      qed
      then have DERIV-in-rball: ∀ y. a ≤ y ∧ y ≤ b ⟶ DERIV (λx. suminf
(?c x) − arctan x) y :> 0
        using ‹−r < a› ‹b < r› by auto
      then show ⋀y. ⟦a < y; y < b⟧ ⟹ DERIV (λx. suminf (?c x) − arctan
x) y :> 0
        using ‹|x| < r› by auto
      show continuous-on {a..b} (λx. suminf (?c x) − arctan x)
        using DERIV-in-rball DERIV-atLeastAtMost-imp-continuous-on by blast
    qed
  qed

  have suminf-arctan-zero: suminf (?c 0) − arctan 0 = 0
  unfolding Suc-eq-plus1[symmetric] power-Suc2 mult-zero-right arctan-zero-zero
suminf-zero
    by auto

  have suminf (?c x) − arctan x = 0
  proof (cases x = 0)
    case True
    then show ?thesis
      using suminf-arctan-zero by auto
  next
    case False
    then have 0 < |x| and − |x| < |x|
      by auto
    have suminf (?c (− |x|)) − arctan (− |x|) = suminf (?c 0) − arctan 0
      by (rule suminf-eq-arctan-bounded[where x1=0 and a1=−|x| and b1=|x|,
symmetric])
        (simp-all only: ‹|x| < r› ‹−|x| < |x|› neg-less-iff-less)
    moreover
    have suminf (?c x) − arctan x = suminf (?c (− |x|)) − arctan (− |x|)
      by (rule suminf-eq-arctan-bounded[where x1=x and a1=− |x| and b1=|x|])
        (simp-all only: ‹|x| < r› ‹− |x| < |x|› neg-less-iff-less)
    ultimately show ?thesis
      using suminf-arctan-zero by auto
  qed
  then show ?thesis by auto
qed

show arctan x = suminf (λn. ?c x n)
proof (cases |x| < 1)
  case True
  then show ?thesis by (rule when-less-one)
next
```

case *False*
then have $|x| = 1$ **using** ‹$|x| \leq 1$› **by** *auto*
let *?a* = $\lambda x\ n.\ |1\ /\ real\ (n * 2 + 1) * x\hat{}(n * 2 + 1)|$
let *?diff* = $\lambda x\ n.\ |arctan\ x - (\sum i{<}n.\ ?c\ x\ i)|$
have *?diff 1 n* $\leq$ *?a 1 n* **for** n :: *nat*
proof −
have $0 < (1$ :: *real*$)$ **by** *auto*
moreover
have *?diff x n* $\leq$ *?a x n* **if** $0 < x$ **and** $x < 1$ **for** x :: *real*
proof −
from *that* **have** $|x| \leq 1$ **and** $|x| < 1$
by *auto*
from ‹$0 < x$› **have** $0 < 1\ /\ real\ (0 * 2 + (1{::}nat)) * x\ \hat{}\ (0 * 2 + 1)$
by *auto*
note *bounds* = *mp*[*OF arctan-series-borders*(*2*)[*OF* ‹$|x| \leq 1$›] *this*, *unfolded when-less-one*[*OF* ‹$|x| < 1$›, *symmetric*], *THEN spec*]
have $0 < 1\ /\ real\ (n{*}2{+}1) * x\hat{}(n{*}2{+}1)$
by (*rule mult-pos-pos*) (*simp-all only*: *zero-less-power*[*OF* ‹$0 < x$›], *auto*)
then have *a-pos*: *?a x n* = $1\ /\ real\ (n{*}2{+}1) * x\hat{}(n{*}2{+}1)$
by (*rule abs-of-pos*)
show *?thesis*
proof (*cases even n*)
case *True*
then have *sgn-pos*: $(-1)\hat{}n = (1{::}real)$ **by** *auto*
from ‹*even n*› **obtain** m **where** $n = 2 * m$ **..**
then have $2 * m = n$ **..**
from *bounds*[*of m*, *unfolded this atLeastAtMost-iff*]
have $|arctan\ x - (\sum i{<}n.\ (?c\ x\ i))| \leq (\sum i{<}n + 1.\ (?c\ x\ i)) - (\sum i{<}n.\ (?c\ x\ i))$
by *auto*
also have ... = *?c x n* **by** *auto*
also have ... = *?a x n* **unfolding** *sgn-pos a-pos* **by** *auto*
finally show *?thesis* **.**
next
case *False*
then have *sgn-neg*: $(-1)\hat{}n = (-1{::}real)$ **by** *auto*
from ‹*odd n*› **obtain** m **where** $n = 2 * m + 1$ **..**
then have *m-def*: $2 * m + 1 = n$ **..**
then have *m-plus*: $2 * (m + 1) = n + 1$ **by** *auto*
from *bounds*[*of m + 1*, *unfolded this atLeastAtMost-iff*, *THEN conjunct1*] *bounds*[*of m*, *unfolded m-def atLeastAtMost-iff*, *THEN conjunct2*]
have $|arctan\ x - (\sum i{<}n.\ (?c\ x\ i))| \leq (\sum i{<}n.\ (?c\ x\ i)) - (\sum i{<}n+1.\ (?c\ x\ i))$ **by** *auto*
also have ... = − *?c x n* **by** *auto*
also have ... = *?a x n* **unfolding** *sgn-neg a-pos* **by** *auto*
finally show *?thesis* **.**
qed
qed
hence $\forall x \in \{\ 0 <..< 1\ \}.\ 0 \leq$ *?a x n* − *?diff x n* **by** *auto*

```
    moreover have isCont (λ x. ?a x n − ?diff x n) x for x
      unfolding diff-conv-add-uminus divide-inverse
        by (auto intro!: isCont-add isCont-rabs continuous-ident isCont-minus
isCont-arctan
         continuous-at-within-inverse isCont-mult isCont-power continuous-const
isCont-sum
        simp del: add-uminus-conv-diff)
    ultimately have 0 ≤ ?a 1 n − ?diff 1 n
      by (rule LIM-less-bound)
    then show ?thesis by auto
  qed
  have ?a 1 ⟶ 0
    unfolding tendsto-rabs-zero-iff power-one divide-inverse One-nat-def
   by (auto intro!: tendsto-mult LIMSEQ-linear LIMSEQ-inverse-real-of-nat simp
del: of-nat-Suc)
  have ?diff 1 ⟶ 0
  proof (rule LIMSEQ-I)
    fix r :: real
    assume 0 < r
    obtain N :: nat where N-I: N ≤ n ⟹ ?a 1 n < r for n
      using LIMSEQ-D[OF ‹?a 1 ⟶ 0› ‹0 < r›] by auto
    have norm (?diff 1 n − 0) < r if N ≤ n for n
      using ‹?diff 1 n ≤ ?a 1 n› N-I[OF that] by auto
    then show ∃ N. ∀ n ≥ N. norm (?diff 1 n − 0) < r by blast
  qed
 from this [unfolded tendsto-rabs-zero-iff, THEN tendsto-add [OF - tendsto-const],
of − arctan 1, THEN tendsto-minus]
  have (?c 1) sums (arctan 1) unfolding sums-def by auto
  then have arctan 1 = (∑ i. ?c 1 i) by (rule sums-unique)

  show ?thesis
  proof (cases x = 1)
    case True
    then show ?thesis by (simp add: ‹arctan 1 = (∑ i. ?c 1 i)›)
  next
    case False
    then have x = −1 using ‹|x| = 1› by auto

    have − (pi/2) < 0 using pi-gt-zero by auto
    have − (2 * pi) < 0 using pi-gt-zero by auto

    have c-minus-minus: ?c (− 1) i = − ?c 1 i for i by auto

    have arctan (− 1) = arctan (tan (−(pi/4)))
      unfolding tan-45 tan-minus ..
    also have ... = − (pi/4)
        by (rule arctan-tan) (auto simp: order-less-trans[OF ‹− (pi/2) < 0›
pi-gt-zero])
    also have ... = − (arctan (tan (pi/4)))
```

```
      unfolding neg-equal-iff-equal
     by (rule arctan-tan[symmetric]) (auto simp: order-less-trans[OF ‹− (2 * pi)
< 0› pi-gt-zero])
    also have ... = − (arctan 1)
      unfolding tan-45 ..
    also have ... = − (∑ i. ?c 1 i)
      using ‹arctan 1 = (∑ i. ?c 1 i)› by auto
    also have ... = (∑ i. ?c (− 1) i)
      using suminf-minus[OF sums-summable[OF ‹(?c 1) sums (arctan 1)›]]
      unfolding c-minus-minus by auto
    finally show ?thesis using ‹x = −1› by auto
   qed
  qed
qed

lemma arctan-half: arctan x = 2 * arctan (x / (1 + sqrt(1 + x^2)))
  for x :: real
proof −
  obtain y where low: − (pi/2) < y and high: y < pi/2 and y-eq: tan y = x
    using tan-total by blast
  then have low2: − (pi/2) < y / 2 and high2: y / 2 < pi/2
    by auto

  have 0 < cos y by (rule cos-gt-zero-pi[OF low high])
  then have cos y ≠ 0 and cos-sqrt: sqrt ((cos y)^2) = cos y
    by auto

  have 1 + (tan y)^2 = 1 + (sin y)^2 / (cos y)^2
    unfolding tan-def power-divide ..
  also have ... = (cos y)^2 / (cos y)^2 + (sin y)^2 / (cos y)^2
    using ‹cos y ≠ 0› by auto
  also have ... = 1 / (cos y)^2
    unfolding add-divide-distrib[symmetric] sin-cos-squared-add2 ..
  finally have 1 + (tan y)^2 = 1 / (cos y)^2 .

  have sin y / (cos y + 1) = tan y / ((cos y + 1) / cos y)
    unfolding tan-def using ‹cos y ≠ 0› by (simp add: field-simps)
  also have ... = tan y / (1 + 1 / cos y)
    using ‹cos y ≠ 0› unfolding add-divide-distrib by auto
  also have ... = tan y / (1 + 1 / sqrt ((cos y)^2))
    unfolding cos-sqrt ..
  also have ... = tan y / (1 + sqrt (1 / (cos y)^2))
    unfolding real-sqrt-divide by auto
  finally have eq: sin y / (cos y + 1) = tan y / (1 + sqrt(1 + (tan y)^2))
    unfolding ‹1 + (tan y)^2 = 1 / (cos y)^2› .

  have arctan x = y
    using arctan-tan low high y-eq by auto
  also have ... = 2 * (arctan (tan (y/2)))
```

using *arctan-tan*[*OF low2 high2*] **by** *auto*
also have ... = *2* ∗ (*arctan* (*sin y* / (*cos y* + *1*)))
unfolding *tan-half* **by** *auto*
finally show *?thesis*
unfolding *eq* ‹*tan y* = *x*› **.**
qed

lemma *arctan-monotone*: *x* < *y* ⟹ *arctan x* < *arctan y*
by (*simp only*: *arctan-less-iff*)

lemma *arctan-monotone'*: *x* ≤ *y* ⟹ *arctan x* ≤ *arctan y*
by (*simp only*: *arctan-le-iff*)

lemma *arctan-inverse*:
assumes *x* ≠ *0*
shows *arctan* (*1*/*x*) = *sgn x* ∗ *pi*/*2* − *arctan x*
proof (*rule arctan-unique*)
have §: *x* > *0* ⟹ *arctan x* < *pi*
using *arctan-bounded* [*of x*] **by** *linarith*
show − (*pi*/*2*) < *sgn x* ∗ *pi*/*2* − *arctan x*
using *assms* **by** (*auto simp*: *sgn-real-def arctan algebra-simps* §)
show *sgn x* ∗ *pi*/*2* − *arctan x* < *pi*/*2*
using *arctan-bounded* [*of* − *x*] *assms*
by (*auto simp*: *algebra-simps sgn-real-def arctan-minus*)
show *tan* (*sgn x* ∗ *pi*/*2* − *arctan x*) = *1*/*x*
unfolding *tan-inverse* [*of arctan x*, *unfolded tan-arctan*] *sgn-real-def*
by (*simp add*: *tan-def cos-arctan sin-arctan sin-diff cos-diff*)
qed

theorem *pi-series*: *pi*/*4* = (∑ *k*. (−*1*) ^ *k* ∗ *1* / *real* (*k* ∗ *2* + *1*))
(**is** - = *?SUM*)
proof −
have *pi*/*4* = *arctan 1*
using *arctan-one* **by** *auto*
also have ... = *?SUM*
using *arctan-series*[*of 1*] **by** *auto*
finally show *?thesis* **by** *auto*
qed

112.20 Existence of Polar Coordinates

lemma *cos-x-y-le-one*: |*x* / *sqrt* (*x*² + *y*²)| ≤ *1*
by (*rule power2-le-imp-le* [*OF - zero-le-one*])
(*simp add*: *power-divide divide-le-eq not-sum-power2-lt-zero*)

lemma *polar-Ex*: ∃ *r*::*real*. ∃ *a*. *x* = *r* ∗ *cos a* ∧ *y* = *r* ∗ *sin a*
proof −
have *polar-ex1*: ∃ *r a*. *x* = *r* ∗ *cos a* ∧ *y* = *r* ∗ *sin a* **if** *0* < *y* **for** *y*
proof −

have $x = sqrt\ (x^2 + y^2) * cos\ (arccos\ (x\ /\ sqrt\ (x^2 + y^2)))$
by (*simp add*: *cos-arccos-abs* [*OF cos-x-y-le-one*])
moreover have $y = sqrt\ (x^2 + y^2) * sin\ (arccos\ (x\ /\ sqrt\ (x^2 + y^2)))$
using *that*
by (*simp add*: *sin-arccos-abs* [*OF cos-x-y-le-one*] *power-divide right-diff-distrib flip*: *real-sqrt-mult*)
ultimately show *?thesis*
by *blast*
qed
show *?thesis*
proof (*cases 0*::*real y rule*: *linorder-cases*)
case *less*
then show *?thesis*
by (*rule polar-ex1*)
next
case *equal*
then show *?thesis*
by (*force simp*: *intro*!: *cos-zero sin-zero*)
next
case *greater*
with *polar-ex1* [**where** $y{=}{-}y$] **show** *?thesis*
by *auto* (*metis cos-minus minus-minus minus-mult-right sin-minus*)
qed
qed

112.21 Basics about polynomial functions: products, extremal behaviour and root counts

lemma *polynomial-product-nat*:
fixes x :: *nat*
assumes *m*: $\bigwedge i.\ i > m \Longrightarrow int\ (a\ i) = 0$
and *n*: $\bigwedge j.\ j > n \Longrightarrow int\ (b\ j) = 0$
shows $(\sum i{\leq}m.\ (a\ i) * x\ \hat{}\ i) * (\sum j{\leq}n.\ (b\ j) * x\ \hat{}\ j) =$
$(\sum r{\leq}m + n.\ (\sum k{\leq}r.\ (a\ k) * (b\ (r - k))) * x\ \hat{}\ r)$
using *polynomial-product* [*of m a n b x*] *assms*
by (*simp only*: *of-nat-mult* [*symmetric*] *of-nat-power* [*symmetric*] *of-nat-eq-iff Int.int-sum* [*symmetric*])

lemma *polyfun-diff*:
fixes x :: *'a*::*idom*
assumes $1 \leq n$
shows $(\sum i{\leq}n.\ a\ i * x\hat{}i) - (\sum i{\leq}n.\ a\ i * y\hat{}i) =$
$(x - y) * (\sum j{<}n.\ (\sum i{=}Suc\ j..n.\ a\ i * y\hat{}(i - j - 1)) * x\hat{}j)$
proof −
have *h*: *bij-betw* $(\lambda(i,j).\ (j,i))$ $((SIGMA\ i : atMost\ n.\ lessThan\ i))$ $(SIGMA\ j : lessThan\ n.\ \{Suc\ j..n\})$
by (*auto simp*: *bij-betw-def inj-on-def*)
have $(\sum i{\leq}n.\ a\ i * x\hat{}i) - (\sum i{\leq}n.\ a\ i * y\hat{}i) = (\sum i{\leq}n.\ a\ i * (x\hat{}i - y\hat{}i))$
by (*simp add*: *right-diff-distrib sum-subtractf*)

also have ... = ($\sum i{\le}n.\ a\ i * (x - y) * (\sum j{<}i.\ y\hat{}(i - Suc\ j) * x\hat{}j)$)
by (*simp add*: *power-diff-sumr2 mult.assoc*)
also have ... = ($\sum i{\le}n.\ \sum j{<}i.\ a\ i * (x - y) * (y\hat{}(i - Suc\ j) * x\hat{}j)$)
by (*simp add*: *sum-distrib-left*)
also have ... = ($\sum (i,j) \in (SIGMA\ i : atMost\ n.\ lessThan\ i).\ a\ i * (x - y) * (y\hat{}(i - Suc\ j) * x\hat{}j)$)
by (*simp add*: *sum.Sigma*)
also have ... = ($\sum (j,i) \in (SIGMA\ j : lessThan\ n.\ \{Suc\ j..n\}).\ a\ i * (x - y) * (y\hat{}(i - Suc\ j) * x\hat{}j)$)
by (*auto simp*: *sum.reindex-bij-betw* [*OF h*, *symmetric*] *intro*: *sum.cong-simp*)
also have ... = ($\sum j{<}n.\ \sum i{=}Suc\ j..n.\ a\ i * (x - y) * (y\hat{}(i - Suc\ j) * x\hat{}j)$)
by (*simp add*: *sum.Sigma*)
also have ... = ($(x - y) * (\sum j{<}n.\ (\sum i{=}Suc\ j..n.\ a\ i * y\hat{}(i - j - 1)) * x\hat{}j)$)
by (*simp add*: *sum-distrib-left mult-ac*)
finally show *?thesis* **.**
qed

lemma *polyfun-diff-alt*:
fixes x :: *'a*::*idom*
assumes $1 \le n$
shows ($\sum i{\le}n.\ a\ i * x\hat{}i) - (\sum i{\le}n.\ a\ i * y\hat{}i) =$
$(x - y) * ((\sum j{<}n.\ \sum k{<}n{-}j.\ a(j + k + 1) * y\hat{}k * x\hat{}j))$
proof −
have ($\sum i{=}Suc\ j..n.\ a\ i * y\hat{}(i - j - 1)) = (\sum k{<}n{-}j.\ a(j{+}k{+}1) * y\hat{}k)$
if $j < n$ **for** j :: *nat*
proof −
have $\bigwedge k.\ k < n - j \Longrightarrow k \in (\lambda i.\ i - Suc\ j)\ `\ \{Suc\ j..n\}$
by (*rule-tac x*=*k* + *Suc j* **in** *image-eqI*, *auto*)
then have *h*: *bij-betw* ($\lambda i.\ i - (j + 1)$) $\{Suc\ j..n\}$ (*lessThan* ($n{-}j$))
by (*auto simp*: *bij-betw-def inj-on-def*)
then show *?thesis*
by (*auto simp*: *sum.reindex-bij-betw* [*OF h*, *symmetric*] *intro*: *sum.cong-simp*)
qed
then show *?thesis*
by (*simp add*: *polyfun-diff* [*OF assms*] *sum-distrib-right*)
qed

lemma *polyfun-linear-factor*:
fixes a :: *'a*::*idom*
shows $\exists b.\ \forall z.\ (\sum i{\le}n.\ c(i) * z\hat{}i) = (z - a) * (\sum i{<}n.\ b(i) * z\hat{}i) + (\sum i{\le}n.\ c(i) * a\hat{}i)$
proof (*cases* $n = 0$)
case *True* **then show** *?thesis*
by *simp*
next
case *False*
have ($\exists b.\ \forall z.\ (\sum i{\le}n.\ c\ i * z\hat{}i) = (z - a) * (\sum i{<}n.\ b\ i * z\hat{}i) + (\sum i{\le}n.\ c\ i * a\hat{}i)) \longleftrightarrow$
($\exists b.\ \forall z.\ (\sum i{\le}n.\ c\ i * z\hat{}i) - (\sum i{\le}n.\ c\ i * a\hat{}i) = (z - a) * (\sum i{<}n.\ b\ i$

$* z\hat{}i))$
by (*simp add*: *algebra-simps*)
also have $\ldots \longleftrightarrow$
$(\exists b.\ \forall z.\ (z - a) * (\sum j{<}n.\ (\sum i = Suc\ j..n.\ c\ i * a\hat{}(i - Suc\ j)) * z\hat{}j) =$
$(z - a) * (\sum i{<}n.\ b\ i * z\hat{}i))$
using *False* **by** (*simp add*: *polyfun-diff*)
also have $\ldots = True$ **by** *auto*
finally show *?thesis*
by *simp*
qed

lemma *polyfun-linear-factor-root*:
fixes $a :: 'a{::}idom$
assumes $(\sum i{\leq}n.\ c(i) * a\hat{}i) = 0$
obtains b **where** $\bigwedge z.\ (\sum i{\leq}n.\ c\ i * z\hat{}i) = (z - a) * (\sum i{<}n.\ b\ i * z\hat{}i)$
using *polyfun-linear-factor* [*of c n a*] *assms* **by** *auto*

lemma *isCont-polynom*: *isCont* $(\lambda w.\ \sum i{\leq}n.\ c\ i * w\hat{}i)\ a$
for $c :: nat \Rightarrow 'a{::}real\text{-}normed\text{-}div\text{-}algebra$
by *simp*

lemma *zero-polynom-imp-zero-coeffs*:
fixes $c :: nat \Rightarrow 'a{::}\{ab\text{-}semigroup\text{-}mult,real\text{-}normed\text{-}div\text{-}algebra\}$
assumes $\bigwedge w.\ (\sum i{\leq}n.\ c\ i * w\hat{}i) = 0\ \ k \leq n$
shows $c\ k = 0$
using *assms*
proof (*induction n arbitrary*: *c k*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n c k*)
have [*simp*]: $c\ 0 = 0$ **using** *Suc.prems*(*1*) [*of 0*]
by *simp*
have $(\sum i{\leq}Suc\ n.\ c\ i * w\hat{}i) = w * (\sum i{\leq}n.\ c\ (Suc\ i) * w\hat{}i)$ **for** w
proof −
have $(\sum i{\leq}Suc\ n.\ c\ i * w\hat{}i) = (\sum i{\leq}n.\ c\ (Suc\ i) * w\ \hat{}\ Suc\ i)$
unfolding *Set-Interval.sum.atMost-Suc-shift*
by *simp*
also have $\ldots = w * (\sum i{\leq}n.\ c\ (Suc\ i) * w\hat{}i)$
by (*simp add*: *sum-distrib-left ac-simps*)
finally show *?thesis* **.**
qed
then have *w*: $\bigwedge w.\ w \neq 0 \Longrightarrow (\sum i{\leq}n.\ c\ (Suc\ i) * w\hat{}i) = 0$
using *Suc* **by** *auto*
then have $(\lambda h.\ \sum i{\leq}n.\ c\ (Suc\ i) * h\hat{}i) -0{\to}\ 0$
by (*simp cong*: *LIM-cong*) — the case $w = 0$ by continuity

```
  then have (∑ i≤n. c (Suc i) * 0^i) = 0
    using isCont-polynom [of 0 λi. c (Suc i) n] LIM-unique
    by (force simp: Limits.isCont-iff)
  then have ⋀w. (∑ i≤n. c (Suc i) * w^i) = 0
    using w by metis
  then have ⋀i. i ≤ n ⟹ c (Suc i) = 0
    using Suc.IH [of λi. c (Suc i)] by blast
  then show ?case using ‹k ≤ Suc n›
    by (cases k) auto
qed

lemma polyfun-rootbound:
  fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
  assumes c k ≠ 0 k≤n
  shows finite {z. (∑ i≤n. c(i) * z^i) = 0} ∧ card {z. (∑ i≤n. c(i) * z^i) = 0}
≤ n
  using assms
proof (induction n arbitrary: c k)
  case 0
  then show ?case
    by simp
next
  case (Suc m c k)
  let ?succase = ?case
  show ?case
  proof (cases {z. (∑ i≤Suc m. c(i) * z^i) = 0} = {})
    case True
    then show ?succase
      by simp
  next
    case False
    then obtain z0 where z0: (∑ i≤Suc m. c(i) * z0^i) = 0
      by blast
    then obtain b where b: ⋀w. (∑ i≤Suc m. c i * w^i) = (w − z0) * (∑ i≤m.
b i * w^i)
      using polyfun-linear-factor-root [OF z0, unfolded lessThan-Suc-atMost]
      by blast
    then have eq: {z. (∑ i≤Suc m. c i * z^i) = 0} = insert z0 {z. (∑ i≤m. b i
* z^i) = 0}
      by auto
    have ¬ (∀ k≤m. b k = 0)
    proof
      assume [simp]: ∀ k≤m. b k = 0
      then have ⋀w. (∑ i≤m. b i * w^i) = 0
        by simp
      then have ⋀w. (∑ i≤Suc m. c i * w^i) = 0
        using b by simp
      then have ⋀k. k ≤ Suc m ⟹ c k = 0
        using zero-polynom-imp-zero-coeffs by blast
```

```
    then show False using Suc.prems by blast
  qed
  then obtain k' where bk': b k' ≠ 0 k' ≤ m
    by blast
  show ?succase
    using Suc.IH [of b k'] bk'
    by (simp add: eq card-insert-if del: sum.atMost-Suc)
  qed
qed

lemma
  fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
  assumes c k ≠ 0 k≤n
  shows polyfun-roots-finite: finite {z. (∑ i≤n. c(i) * z^i) = 0}
    and polyfun-roots-card: card {z. (∑ i≤n. c(i) * z^i) = 0} ≤ n
  using polyfun-rootbound assms by auto

lemma polyfun-finite-roots:
  fixes c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
  shows finite {x. (∑ i≤n. c i * x^i) = 0} ⟷ (∃ i≤n. c i ≠ 0)
    (is ?lhs = ?rhs)
proof
  assume ?lhs
  moreover have ¬ finite {x. (∑ i≤n. c i * x^i) = 0} if ∀ i≤n. c i = 0
  proof −
    from that have ⋀x. (∑ i≤n. c i * x^i) = 0
      by simp
    then show ?thesis
      using ex-new-if-finite [OF infinite-UNIV-char-0 [where 'a='a]]
      by auto
  qed
  ultimately show ?rhs by metis
next
  assume ?rhs
  with polyfun-rootbound show ?lhs by blast
qed

lemma polyfun-eq-0: (∀ x. (∑ i≤n. c i * x^i) = 0) ⟷ (∀ i≤n. c i = 0)
  for c :: nat ⇒ 'a::{idom,real-normed-div-algebra}

  using zero-polynom-imp-zero-coeffs by auto

lemma polyfun-eq-coeffs: (∀ x. (∑ i≤n. c i * x^i) = (∑ i≤n. d i * x^i)) ⟷
(∀ i≤n. c i = d i)
  for c :: nat ⇒ 'a::{idom,real-normed-div-algebra}
proof −
  have (∀ x. (∑ i≤n. c i * x^i) = (∑ i≤n. d i * x^i)) ⟷ (∀ x. (∑ i≤n. (c i −
d i) * x^i) = 0)
    by (simp add: left-diff-distrib Groups-Big.sum-subtractf)
```

also have ... $\longleftrightarrow$ $(\forall\, i{\leq}n.\ c\ i - d\ i = 0)$
by (*rule polyfun-eq-0*)
finally show *?thesis*
by *simp*
qed

lemma *polyfun-eq-const*:
fixes c :: *nat* $\Rightarrow$ 'a::{*idom*,*real-normed-div-algebra*}
shows $(\forall\, x.\ (\sum i{\leq}n.\ c\ i * x\hat{}i) = k) \longleftrightarrow c\ 0 = k \wedge (\forall\, i \in \{1..n\}.\ c\ i = 0)$
(**is** *?lhs* = *?rhs*)
proof −
have ∗: $\forall\, x.\ (\sum i{\leq}n.\ (\mathit{if}\ i{=}0\ \mathit{then}\ k\ \mathit{else}\ 0) * x\hat{}i) = k$
by (*induct n*) *auto*
show *?thesis*
proof
assume *?lhs*
with ∗ **have** $(\forall\, i{\leq}n.\ c\ i = (\mathit{if}\ i{=}0\ \mathit{then}\ k\ \mathit{else}\ 0))$
by (*simp add*: *polyfun-eq-coeffs* [*symmetric*])
then show *?rhs* **by** *simp*
next
assume *?rhs*
then show *?lhs* **by** (*induct n*) *auto*
qed
qed

lemma *root-polyfun*:
fixes z :: 'a::*idom*
assumes $1 \leq n$
shows $z\hat{}n = a \longleftrightarrow (\sum i{\leq}n.\ (\mathit{if}\ i = 0\ \mathit{then} -a\ \mathit{else\ if}\ i{=}n\ \mathit{then}\ 1\ \mathit{else}\ 0) * z\hat{}i) = 0$
using *assms* **by** (*cases n*) (*simp-all add*: *sum.atLeast-Suc-atMost atLeast0AtMost* [*symmetric*])

lemma
assumes *SORT-CONSTRAINT*('a::{*idom*,*real-normed-div-algebra*})
and $1 \leq n$
shows *finite-roots-unity*: *finite* $\{z{::}'a.\ z\hat{}n = 1\}$
and *card-roots-unity*: *card* $\{z{::}'a.\ z\hat{}n = 1\} \leq n$
using *polyfun-rootbound* [*of* $\lambda i.\ \mathit{if}\ i = 0\ \mathit{then} -1\ \mathit{else\ if}\ i{=}n\ \mathit{then}\ 1\ \mathit{else}\ 0\ n\ n$] *assms*(*2*)
by (*auto simp*: *root-polyfun* [*OF assms*(*2*)])

112.22 Hyperbolic functions

definition *sinh* :: 'a :: {*banach*, *real-normed-algebra-1*} $\Rightarrow$ 'a **where**
sinh x = (*exp x* − *exp* (−*x*)) /$_R$ *2*

definition *cosh* :: 'a :: {*banach*, *real-normed-algebra-1*} $\Rightarrow$ 'a **where**
cosh x = (*exp x* + *exp* (−*x*)) /$_R$ *2*

definition *tanh* :: *'a* :: {*banach*, *real-normed-field*} ⇒ *'a* **where**
tanh x = sinh x / cosh x

definition *arsinh* :: *'a* :: {*banach*, *real-normed-algebra-1*, *ln*} ⇒ *'a* **where**
arsinh x = ln (x + (x^2 + 1) powr of-real (1/2))

definition *arcosh* :: *'a* :: {*banach*, *real-normed-algebra-1*, *ln*} ⇒ *'a* **where**
arcosh x = ln (x + (x^2 − 1) powr of-real (1/2))

definition *artanh* :: *'a* :: {*banach*, *real-normed-field*, *ln*} ⇒ *'a* **where**
artanh x = ln ((1 + x) / (1 − x)) / 2

lemma *arsinh-0* [*simp*]: *arsinh 0 = 0*
by (*simp add*: *arsinh-def*)

lemma *arcosh-1* [*simp*]: *arcosh 1 = 0*
by (*simp add*: *arcosh-def*)

lemma *artanh-0* [*simp*]: *artanh 0 = 0*
by (*simp add*: *artanh-def*)

lemma *tanh-altdef*:
tanh x = (exp x − exp (−x)) / (exp x + exp (−x))
proof −
have *tanh x = (2 *$_R$ sinh x) / (2 *$_R$ cosh x)*
by (*simp add*: *tanh-def scaleR-conv-of-real*)
also have *2 *$_R$ sinh x = exp x − exp (−x)*
by (*simp add*: *sinh-def*)
also have *2 *$_R$ cosh x = exp x + exp (−x)*
by (*simp add*: *cosh-def*)
finally show *?thesis* **.**
qed

lemma *tanh-real-altdef*: *tanh (x*::*real) = (1 − exp (− 2 ∗ x)) / (1 + exp (− 2 ∗ x))*
proof −
have [*simp*]: *exp (2 ∗ x) = exp x ∗ exp x exp (x ∗ 2) = exp x ∗ exp x*
by (*subst exp-add* [*symmetric*]; *simp*)+
have *tanh x = (2 ∗ exp (−x) ∗ sinh x) / (2 ∗ exp (−x) ∗ cosh x)*
by (*simp add*: *tanh-def*)
also have *2 ∗ exp (−x) ∗ sinh x = 1 − exp (−2∗x)*
by (*simp add*: *exp-minus field-simps sinh-def*)
also have *2 ∗ exp (−x) ∗ cosh x = 1 + exp (−2∗x)*
by (*simp add*: *exp-minus field-simps cosh-def*)
finally show *?thesis* **.**
qed

lemma *sinh-converges*: $(\lambda n.\ \textit{if even } n \textit{ then } 0 \textit{ else } x \mathbin{\hat{}} n /_R \textit{ fact } n)$ *sums sinh x*
proof −
  have $(\lambda n.\ (x \mathbin{\hat{}} n /_R \textit{ fact } n - (-x) \mathbin{\hat{}} n /_R \textit{ fact } n) /_R 2)$ *sums sinh x*
    unfolding *sinh-def* **by** (*intro sums-scaleR-right sums-diff exp-converges*)
  also have $(\lambda n.\ (x \mathbin{\hat{}} n /_R \textit{ fact } n - (-x) \mathbin{\hat{}} n /_R \textit{ fact } n) /_R 2) =$
    $(\lambda n.\ \textit{if even } n \textit{ then } 0 \textit{ else } x \mathbin{\hat{}} n /_R \textit{ fact } n)$ **by** *auto*
  finally show *?thesis* **.**
qed

lemma *cosh-converges*: $(\lambda n.\ \textit{if even } n \textit{ then } x \mathbin{\hat{}} n /_R \textit{ fact } n \textit{ else } 0)$ *sums cosh x*
proof −
  have $(\lambda n.\ (x \mathbin{\hat{}} n /_R \textit{ fact } n + (-x) \mathbin{\hat{}} n /_R \textit{ fact } n) /_R 2)$ *sums cosh x*
    unfolding *cosh-def* **by** (*intro sums-scaleR-right sums-add exp-converges*)
  also have $(\lambda n.\ (x \mathbin{\hat{}} n /_R \textit{ fact } n + (-x) \mathbin{\hat{}} n /_R \textit{ fact } n) /_R 2) =$
    $(\lambda n.\ \textit{if even } n \textit{ then } x \mathbin{\hat{}} n /_R \textit{ fact } n \textit{ else } 0)$ **by** *auto*
  finally show *?thesis* **.**
qed

lemma *sinh-0* [*simp*]: *sinh 0 = 0*
  by (*simp add*: *sinh-def*)

lemma *cosh-0* [*simp*]: *cosh 0 = 1*
proof −
  have *cosh 0* $= (1/2) *_R (1 + 1)$ **by** (*simp add*: *cosh-def*)
  also have $\ldots = 1$ **by** (*rule scaleR-half-double*)
  finally show *?thesis* **.**
qed

lemma *tanh-0* [*simp*]: *tanh 0 = 0*
  by (*simp add*: *tanh-def*)

lemma *sinh-minus* [*simp*]: $\textit{sinh } (-x) = -\textit{sinh } x$
  by (*simp add*: *sinh-def algebra-simps*)

lemma *cosh-minus* [*simp*]: $\textit{cosh } (-x) = \textit{cosh } x$
  by (*simp add*: *cosh-def algebra-simps*)

lemma *tanh-minus* [*simp*]: $\textit{tanh } (-x) = -\textit{tanh } x$
  by (*simp add*: *tanh-def*)

lemma *sinh-ln-real*: $x > 0 \Longrightarrow \textit{sinh } (\textit{ln } x :: \textit{real}) = (x - \textit{inverse } x) / 2$
  by (*simp add*: *sinh-def exp-minus*)

lemma *cosh-ln-real*: $x > 0 \Longrightarrow \textit{cosh } (\textit{ln } x :: \textit{real}) = (x + \textit{inverse } x) / 2$
  by (*simp add*: *cosh-def exp-minus*)

lemma *tanh-ln-real*:
  $\textit{tanh } (\textit{ln } x :: \textit{real}) = (x \mathbin{\hat{}} 2 - 1) / (x \mathbin{\hat{}} 2 + 1)$ **if** $x > 0$
proof −

from *that* **have** $(x * 2 - inverse\ x * 2) * (x^2 + 1) =$
$(x^2 - 1) * (2 * x + 2 * inverse\ x)$
by (*simp add*: *field-simps power2-eq-square*)
moreover have $x^2 + 1 > 0$
using *that* **by** (*simp add*: *ac-simps add-pos-nonneg*)
moreover have $2 * x + 2 * inverse\ x > 0$
using *that* **by** (*simp add*: *add-pos-pos*)
ultimately have $(x * 2 - inverse\ x * 2)\ /$
$(2 * x + 2 * inverse\ x) =$
$(x^2 - 1)\ /\ (x^2 + 1)$
by (*simp add*: *frac-eq-eq*)
with *that* **show** *?thesis*
by (*simp add*: *tanh-def sinh-ln-real cosh-ln-real*)
qed

lemma *has-field-derivative-scaleR-right* [*derivative-intros*]:
$(f\ has\text{-}field\text{-}derivative\ D)\ F \Longrightarrow ((\lambda x.\ c *_R f\ x)\ has\text{-}field\text{-}derivative\ (c *_R D))\ F$
unfolding *has-field-derivative-def*
using *has-derivative-scaleR-right*[*of f* $\lambda x.\ D * x\ F\ c$]
by (*simp add*: *mult-scaleR-left* [*symmetric*] *del*: *mult-scaleR-left*)

lemma *has-field-derivative-sinh* [*THEN DERIV-chain2*, *derivative-intros*]:
$(sinh\ has\text{-}field\text{-}derivative\ cosh\ x)\ (at\ (x :: 'a :: \{banach, real\text{-}normed\text{-}field\}))$
unfolding *sinh-def cosh-def* **by** (*auto intro*!: *derivative-eq-intros*)

lemma *has-field-derivative-cosh* [*THEN DERIV-chain2*, *derivative-intros*]:
$(cosh\ has\text{-}field\text{-}derivative\ sinh\ x)\ (at\ (x :: 'a :: \{banach, real\text{-}normed\text{-}field\}))$
unfolding *sinh-def cosh-def* **by** (*auto intro*!: *derivative-eq-intros*)

lemma *has-field-derivative-tanh* [*THEN DERIV-chain2*, *derivative-intros*]:
$cosh\ x \neq 0 \Longrightarrow (tanh\ has\text{-}field\text{-}derivative\ 1 - tanh\ x\ \hat{}\ 2)$
$(at\ (x :: 'a :: \{banach, real\text{-}normed\text{-}field\}))$
unfolding *tanh-def* **by** (*auto intro*!: *derivative-eq-intros simp*: *power2-eq-square field-split-simps*)

lemma *has-derivative-sinh* [*derivative-intros*]:
fixes $g :: 'a \Rightarrow ('a :: \{banach, real\text{-}normed\text{-}field\})$
assumes $(g\ has\text{-}derivative\ (\lambda x.\ Db * x))\ (at\ x\ within\ s)$
shows $((\lambda x.\ sinh\ (g\ x))\ has\text{-}derivative\ (\lambda y.\ (cosh\ (g\ x) * Db) * y))\ (at\ x\ within\ s)$
proof −
have $((\lambda x.\ -g\ x)\ has\text{-}derivative\ (\lambda y.\ -(Db * y)))\ (at\ x\ within\ s)$
using *assms* **by** (*intro derivative-intros*)
also have $(\lambda y.\ -(Db * y)) = (\lambda x.\ (-Db) * x)$ **by** (*simp add*: *fun-eq-iff*)
finally have $((\lambda x.\ sinh\ (g\ x))\ has\text{-}derivative$
$(\lambda y.\ (exp\ (g\ x) * Db * y - exp\ (-g\ x) * (-Db) * y)\ /_R\ 2))\ (at\ x\ within\ s)$
unfolding *sinh-def* **by** (*intro derivative-intros assms*)
also have $(\lambda y.\ (exp\ (g\ x) * Db * y - exp\ (-g\ x) * (-Db) * y)\ /_R\ 2) = (\lambda y.\ (cosh\ (g\ x) * Db) * y)$

by (*simp add*: *fun-eq-iff cosh-def algebra-simps*)
finally show *?thesis* **.**
qed

lemma *has-derivative-cosh* [*derivative-intros*]:
fixes $g :: 'a \Rightarrow ('a :: \{banach, real\text{-}normed\text{-}field\})$
assumes $(g\ has\text{-}derivative\ (\lambda y.\ Db * y))\ (at\ x\ within\ s)$
shows $((\lambda x.\ cosh\ (g\ x))\ has\text{-}derivative\ (\lambda y.\ (sinh\ (g\ x) * Db) * y))\ (at\ x\ within\ s)$
proof −
have $((\lambda x.\ -\ g\ x)\ has\text{-}derivative\ (\lambda y.\ -(Db * y)))\ (at\ x\ within\ s)$
using *assms* **by** (*intro derivative-intros*)
also have $(\lambda y.\ -(Db * y)) = (\lambda y.\ (-Db) * y)$ **by** (*simp add*: *fun-eq-iff*)
finally have $((\lambda x.\ cosh\ (g\ x))\ has\text{-}derivative$
$(\lambda y.\ (exp\ (g\ x) * Db * y + exp\ (-g\ x) * (-Db) * y)\ /_R\ 2))\ (at\ x\ within\ s)$
unfolding *cosh-def* **by** (*intro derivative-intros assms*)
also have $(\lambda y.\ (exp\ (g\ x) * Db * y + exp\ (-g\ x) * (-Db) * y)\ /_R\ 2) = (\lambda y.\ (sinh\ (g\ x) * Db) * y)$
by (*simp add*: *fun-eq-iff sinh-def algebra-simps*)
finally show *?thesis* **.**
qed

lemma *sinh-plus-cosh*: $sinh\ x + cosh\ x = exp\ x$
proof −
have $sinh\ x + cosh\ x = (1/2) *_R (exp\ x + exp\ x)$
by (*simp add*: *sinh-def cosh-def algebra-simps*)
also have $\ldots = exp\ x$ **by** (*rule scaleR-half-double*)
finally show *?thesis* **.**
qed

lemma *cosh-plus-sinh*: $cosh\ x + sinh\ x = exp\ x$
by (*subst add.commute*) (*rule sinh-plus-cosh*)

lemma *cosh-minus-sinh*: $cosh\ x - sinh\ x = exp\ (-x)$
proof −
have $cosh\ x - sinh\ x = (1/2) *_R (exp\ (-x) + exp\ (-x))$
by (*simp add*: *sinh-def cosh-def algebra-simps*)
also have $\ldots = exp\ (-x)$ **by** (*rule scaleR-half-double*)
finally show *?thesis* **.**
qed

lemma *sinh-minus-cosh*: $sinh\ x - cosh\ x = -exp\ (-x)$
using *cosh-minus-sinh*[*of x*] **by** (*simp add*: *algebra-simps*)

context
fixes $x :: 'a :: \{real\text{-}normed\text{-}field, banach\}$
begin

lemma *sinh-zero-iff*: *sinh x = 0 ⟷ exp x ∈ {1, −1}*
by (*auto simp*: *sinh-def field-simps exp-minus power2-eq-square square-eq-1-iff*)

lemma *cosh-zero-iff*: *cosh x = 0 ⟷ exp x ^ 2 = −1*
by (*auto simp*: *cosh-def exp-minus field-simps power2-eq-square eq-neg-iff-add-eq-0*)

lemma *cosh-square-eq*: *cosh x ^ 2 = sinh x ^ 2 + 1*
by (*simp add*: *cosh-def sinh-def algebra-simps power2-eq-square exp-add* [*symmetric*]
scaleR-conv-of-real)

lemma *sinh-square-eq*: *sinh x ^ 2 = cosh x ^ 2 − 1*
by (*simp add*: *cosh-square-eq*)

lemma *hyperbolic-pythagoras*: *cosh x ^ 2 − sinh x ^ 2 = 1*
by (*simp add*: *cosh-square-eq*)

lemma *sinh-add*: *sinh (x + y) = sinh x * cosh y + cosh x * sinh y*
by (*simp add*: *sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add* [*symmetric*])

lemma *sinh-diff*: *sinh (x − y) = sinh x * cosh y − cosh x * sinh y*
by (*simp add*: *sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add* [*symmetric*])

lemma *cosh-add*: *cosh (x + y) = cosh x * cosh y + sinh x * sinh y*
by (*simp add*: *sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add* [*symmetric*])

lemma *cosh-diff*: *cosh (x − y) = cosh x * cosh y − sinh x * sinh y*
by (*simp add*: *sinh-def cosh-def algebra-simps scaleR-conv-of-real exp-add* [*symmetric*])

lemma *tanh-add*:
*tanh (x + y) = (tanh x + tanh y) / (1 + tanh x * tanh y)*
if *cosh x ≠ 0 cosh y ≠ 0*
proof −
have (*sinh x * cosh y + cosh x * sinh y*) * (*1 + sinh x * sinh y / (cosh x * cosh y*)) =
(*cosh x * cosh y + sinh x * sinh y*) * ((*sinh x * cosh y + sinh y * cosh x*) / (*cosh y * cosh x*))
using *that* **by** (*simp add*: *field-split-simps*)
also have (*sinh x * cosh y + sinh y * cosh x*) / (*cosh y * cosh x*) = *sinh x / cosh x + sinh y / cosh y*
using *that* **by** (*simp add*: *field-split-simps*)
finally have (*sinh x * cosh y + cosh x * sinh y*) * (*1 + sinh x * sinh y / (cosh x * cosh y*)) =
(*sinh x / cosh x + sinh y / cosh y*) * (*cosh x * cosh y + sinh x * sinh y*)
by *simp*
then show *?thesis*
using *that* **by** (*auto simp add*: *tanh-def sinh-add cosh-add eq-divide-eq*)
(*simp-all add*: *field-split-simps*)
qed

lemma *sinh-double*: *sinh* (*2* ∗ *x*) = *2* ∗ *sinh x* ∗ *cosh x*
using *sinh-add*[*of x*] **by** *simp*

lemma *cosh-double*: *cosh* (*2* ∗ *x*) = *cosh x* ^ *2* + *sinh x* ^ *2*
using *cosh-add*[*of x*] **by** (*simp add*: *power2-eq-square*)

end

lemma *sinh-field-def*: *sinh z* = (*exp z* − *exp* (−*z*)) / (*2* :: *'a* :: {*banach*, *real-normed-field*})
by (*simp add*: *sinh-def scaleR-conv-of-real*)

lemma *cosh-field-def*: *cosh z* = (*exp z* + *exp* (−*z*)) / (*2* :: *'a* :: {*banach*, *real-normed-field*})
by (*simp add*: *cosh-def scaleR-conv-of-real*)

112.22.1 More specific properties of the real functions

lemma *plus-inverse-ge-2*:
fixes *x* :: *real*
assumes *x* > *0*
shows *x* + *inverse x* ≥ *2*
proof −
have *0* ≤ (*x* − *1*) ^ *2* **by** *simp*
also have ... = *x*^*2* − *2*∗*x* + *1* **by** (*simp add*: *power2-eq-square algebra-simps*)
finally show *?thesis* **using** *assms* **by** (*simp add*: *field-simps power2-eq-square*)
qed

lemma *sinh-real-nonneg-iff* [*simp*]: *sinh* (*x* :: *real*) ≥ *0* ⟷ *x* ≥ *0*
by (*simp add*: *sinh-def*)

lemma *sinh-real-pos-iff* [*simp*]: *sinh* (*x* :: *real*) > *0* ⟷ *x* > *0*
by (*simp add*: *sinh-def*)

lemma *sinh-real-nonpos-iff* [*simp*]: *sinh* (*x* :: *real*) ≤ *0* ⟷ *x* ≤ *0*
by (*simp add*: *sinh-def*)

lemma *sinh-real-neg-iff* [*simp*]: *sinh* (*x* :: *real*) < *0* ⟷ *x* < *0*
by (*simp add*: *sinh-def*)

lemma *cosh-real-ge-1*: *cosh* (*x* :: *real*) ≥ *1*
using *plus-inverse-ge-2*[*of exp x*] **by** (*simp add*: *cosh-def exp-minus*)

lemma *cosh-real-pos* [*simp*]: *cosh* (*x* :: *real*) > *0*
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *cosh-real-nonneg*[*simp*]: *cosh* (*x* :: *real*) ≥ *0*
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *cosh-real-nonzero* [*simp*]: *cosh* (*x* :: *real*) ≠ *0*
using *cosh-real-ge-1*[*of x*] **by** *simp*

lemma *arsinh-real-def*: *arsinh* (*x*::*real*) = *ln* (*x* + *sqrt* (*x*^2 + *1*))
by (*simp add*: *arsinh-def powr-half-sqrt*)

lemma *arcosh-real-def*: *x* ≥ *1* ⟹ *arcosh* (*x*::*real*) = *ln* (*x* + *sqrt* (*x*^2 − *1*))
by (*simp add*: *arcosh-def powr-half-sqrt*)

lemma *arsinh-real-aux*: *0* < *x* + *sqrt* (*x* ^ *2* + *1* :: *real*)
proof (*cases x* < *0*)
case *True*
have (−*x*) ^ *2* = *x* ^ *2* **by** *simp*
also have *x* ^ *2* < *x* ^ *2* + *1* **by** *simp*
finally have *sqrt* ((−*x*) ^ *2*) < *sqrt* (*x* ^ *2* + *1*)
by (*rule real-sqrt-less-mono*)
thus *?thesis* **using** *True* **by** *simp*
qed (*auto simp*: *add-nonneg-pos*)

lemma *arsinh-minus-real* [*simp*]: *arsinh* (−*x*::*real*) = −*arsinh x*
proof −
have *arsinh* (−*x*) = *ln* (*sqrt* (*x*^2 + *1*) − *x*)
by (*simp add*: *arsinh-real-def*)
also have *sqrt* (*x*^2 + *1*) − *x* = *inverse* (*sqrt* (*x*^2 + *1*) + *x*)
using *arsinh-real-aux*[*of x*] **by** (*simp add*: *field-split-simps algebra-simps power2-eq-square*)
also have *ln* ... = −*arsinh x*
using *arsinh-real-aux*[*of x*] **by** (*simp add*: *arsinh-real-def ln-inverse*)
finally show *?thesis* **.**
qed

lemma *artanh-minus-real* [*simp*]:
assumes *abs x* < *1*
shows *artanh* (−*x*::*real*) = −*artanh x*
by (*smt* (*verit*) *artanh-def assms field-sum-of-halves ln-div*)

lemma *sinh-less-cosh-real*: *sinh* (*x* :: *real*) < *cosh x*
by (*simp add*: *sinh-def cosh-def*)

lemma *sinh-le-cosh-real*: *sinh* (*x* :: *real*) ≤ *cosh x*
by (*simp add*: *sinh-def cosh-def*)

lemma *tanh-real-lt-1*: *tanh* (*x* :: *real*) < *1*
by (*simp add*: *tanh-def sinh-less-cosh-real*)

lemma *tanh-real-gt-neg1*: *tanh* (*x* :: *real*) > −*1*
proof −
have − *cosh x* < *sinh x* **by** (*simp add*: *sinh-def cosh-def field-split-simps*)
thus *?thesis* **by** (*simp add*: *tanh-def field-simps*)
qed

lemma *tanh-real-bounds*: *tanh* (*x* :: *real*) ∈ {−*1*<..<*1*}

using *tanh-real-lt-1 tanh-real-gt-neg1* **by** *simp*

context
fixes x :: *real*
begin

lemma *arsinh-sinh-real*: *arsinh* (*sinh* x) = x
by (*simp add*: *arsinh-real-def powr-def sinh-square-eq sinh-plus-cosh*)

lemma *arcosh-cosh-real*: $x \geq 0 \Longrightarrow$ *arcosh* (*cosh* x) = x
by (*simp add*: *arcosh-real-def powr-def cosh-square-eq cosh-real-ge-1 cosh-plus-sinh*)

lemma *artanh-tanh-real*: *artanh* (*tanh* x) = x
proof −
have *artanh* (*tanh* x) = *ln* (*cosh* x ∗ (*cosh* x + *sinh* x) / (*cosh* x ∗ (*cosh* x − *sinh* x))) / 2
by (*simp add*: *artanh-def tanh-def field-split-simps*)
also have *cosh* x ∗ (*cosh* x + *sinh* x) / (*cosh* x ∗ (*cosh* x − *sinh* x)) =
(*cosh* x + *sinh* x) / (*cosh* x − *sinh* x) **by** *simp*
also have ... = (*exp* x)^2
by (*simp add*: *cosh-plus-sinh cosh-minus-sinh exp-minus field-simps power2-eq-square*)
also have *ln* ((*exp* x)^2) / 2 = x **by** (*simp add*: *ln-realpow*)
finally show *?thesis* **.**
qed

lemma *sinh-real-zero-iff* [*simp*]: *sinh* x = 0 ⟷ x = 0
by (*metis arsinh-0 arsinh-sinh-real sinh-0*)

lemma *cosh-real-one-iff* [*simp*]: *cosh* x = 1 ⟷ x = 0
by (*smt* (*verit*, *best*) *Transcendental.arcosh-cosh-real cosh-0 cosh-minus*)

lemma *tanh-real-nonneg-iff* [*simp*]: *tanh* $x \geq 0$ ⟷ $x \geq 0$
by (*simp add*: *tanh-def field-simps*)

lemma *tanh-real-pos-iff* [*simp*]: *tanh* $x > 0$ ⟷ $x > 0$
by (*simp add*: *tanh-def field-simps*)

lemma *tanh-real-nonpos-iff* [*simp*]: *tanh* $x \leq 0$ ⟷ $x \leq 0$
by (*simp add*: *tanh-def field-simps*)

lemma *tanh-real-neg-iff* [*simp*]: *tanh* $x < 0$ ⟷ $x < 0$
by (*simp add*: *tanh-def field-simps*)

lemma *tanh-real-zero-iff* [*simp*]: *tanh* x = 0 ⟷ x = 0
by (*simp add*: *tanh-def field-simps*)

end

lemma *sinh-real-strict-mono*: *strict-mono* (*sinh* :: *real* ⇒ *real*)

by (*force intro*: *strict-monoI DERIV-pos-imp-increasing* [**where** *f*=*sinh*] *derivative-intros*)

lemma *cosh-real-strict-mono*:
assumes $0 \leq x$ **and** $x < (y::real)$
shows $cosh\ x < cosh\ y$
proof −
from *assms* **have** $\exists z{>}x.\ z < y \wedge cosh\ y - cosh\ x = (y - x) * sinh\ z$
by (*intro MVT2*) (*auto dest*: *connectedD-interval intro*!: *derivative-eq-intros*)
then obtain *z* **where** *z*: $z > x\ z < y\ cosh\ y - cosh\ x = (y - x) * sinh\ z$ **by** *blast*
note ‹$cosh\ y - cosh\ x = (y - x) * sinh\ z$›
also from ‹$z > x$› **and** *assms* **have** $(y - x) * sinh\ z > 0$ **by** (*intro mult-pos-pos*) *auto*
finally show $cosh\ x < cosh\ y$ **by** *simp*
qed

lemma *tanh-real-strict-mono*: *strict-mono* $(tanh :: real \Rightarrow real)$
proof −
have $tanh\ x\ \hat{}\ 2 < 1$ **for** $x :: real$
using *tanh-real-bounds*[*of x*] **by** (*simp add*: *abs-square-less-1 abs-if*)
then show *?thesis*
by (*force intro*!: *strict-monoI DERIV-pos-imp-increasing* [**where** *f*=*tanh*] *derivative-intros*)
qed

lemma *sinh-real-abs* [*simp*]: $sinh\ (abs\ x :: real) = abs\ (sinh\ x)$
by (*simp add*: *abs-if*)

lemma *cosh-real-abs* [*simp*]: $cosh\ (abs\ x :: real) = cosh\ x$
by (*simp add*: *abs-if*)

lemma *tanh-real-abs* [*simp*]: $tanh\ (abs\ x :: real) = abs\ (tanh\ x)$
by (*auto simp*: *abs-if*)

lemma *sinh-real-eq-iff* [*simp*]: $sinh\ x = sinh\ y \longleftrightarrow x = (y :: real)$
using *sinh-real-strict-mono* **by** (*simp add*: *strict-mono-eq*)

lemma *tanh-real-eq-iff* [*simp*]: $tanh\ x = tanh\ y \longleftrightarrow x = (y :: real)$
using *tanh-real-strict-mono* **by** (*simp add*: *strict-mono-eq*)

lemma *cosh-real-eq-iff* [*simp*]: $cosh\ x = cosh\ y \longleftrightarrow abs\ x = abs\ (y :: real)$
proof −
have $cosh\ x = cosh\ y \longleftrightarrow x = y$ **if** $x \geq 0\ y \geq 0$ **for** $x\ y :: real$
using *cosh-real-strict-mono*[*of x y*] *cosh-real-strict-mono*[*of y x*] *that*
by (*cases x y rule*: *linorder-cases*) *auto*
from *this*[*of abs x abs y*] **show** *?thesis* **by** *simp*
qed

lemma *sinh-real-le-iff* [*simp*]: *sinh x* ≤ *sinh y* ⟷ *x* ≤ (*y*::*real*)
using *sinh-real-strict-mono* **by** (*simp add*: *strict-mono-less-eq*)

lemma *cosh-real-nonneg-le-iff*: *x* ≥ *0* ⟹ *y* ≥ *0* ⟹ *cosh x* ≤ *cosh y* ⟷ *x* ≤ (*y*::*real*)
using *cosh-real-strict-mono*[*of x y*] *cosh-real-strict-mono*[*of y x*]
by (*cases x y rule*: *linorder-cases*) *auto*

lemma *cosh-real-nonpos-le-iff*: *x* ≤ *0* ⟹ *y* ≤ *0* ⟹ *cosh x* ≤ *cosh y* ⟷ *x* ≥ (*y*::*real*)
using *cosh-real-nonneg-le-iff*[*of* −*x* −*y*] **by** *simp*

lemma *tanh-real-le-iff* [*simp*]: *tanh x* ≤ *tanh y* ⟷ *x* ≤ (*y*::*real*)
using *tanh-real-strict-mono* **by** (*simp add*: *strict-mono-less-eq*)

lemma *sinh-real-less-iff* [*simp*]: *sinh x* < *sinh y* ⟷ *x* < (*y*::*real*)
using *sinh-real-strict-mono* **by** (*simp add*: *strict-mono-less*)

lemma *cosh-real-nonneg-less-iff*: *x* ≥ *0* ⟹ *y* ≥ *0* ⟹ *cosh x* < *cosh y* ⟷ *x* < (*y*::*real*)
using *cosh-real-strict-mono*[*of x y*] *cosh-real-strict-mono*[*of y x*]
by (*cases x y rule*: *linorder-cases*) *auto*

lemma *cosh-real-nonpos-less-iff*: *x* ≤ *0* ⟹ *y* ≤ *0* ⟹ *cosh x* < *cosh y* ⟷ *x* > (*y*::*real*)
using *cosh-real-nonneg-less-iff*[*of* −*x* −*y*] **by** *simp*

lemma *tanh-real-less-iff* [*simp*]: *tanh x* < *tanh y* ⟷ *x* < (*y*::*real*)
using *tanh-real-strict-mono* **by** (*simp add*: *strict-mono-less*)

112.22.2 Limits

lemma *sinh-real-at-top*: *filterlim* (*sinh* :: *real* ⇒ *real*) *at-top at-top*
proof −
have ∗: ((λ*x*. − *exp* (− *x*)) ⟶ (−*0*::*real*)) *at-top*
by (*intro tendsto-minus filterlim-compose*[*OF exp-at-bot*] *filterlim-uminus-at-bot-at-top*)
have *filterlim* (λ*x*. (*1*/*2*) ∗ (−*exp* (−*x*) + *exp x*) :: *real*) *at-top at-top*
by (*rule filterlim-tendsto-pos-mult-at-top*[*OF* - -
filterlim-tendsto-add-at-top[*OF* ∗]] *tendsto-const*)+ (*auto simp*: *exp-at-top*)
also have (λ*x*. (*1*/*2*) ∗ (−*exp* (−*x*) + *exp x*) :: *real*) = *sinh*
by (*simp add*: *fun-eq-iff sinh-def*)
finally show *?thesis* **.**
qed

lemma *sinh-real-at-bot*: *filterlim* (*sinh* :: *real* ⇒ *real*) *at-bot at-bot*
proof −
have *filterlim* (λ*x*. −*sinh x* :: *real*) *at-bot at-top*

by (*simp add*: *filterlim-uminus-at-top* [*symmetric*] *sinh-real-at-top*)
also have (λ*x*. −*sinh x* :: *real*) = (λ*x*. *sinh* (−*x*)) **by** *simp*
finally show *?thesis* **by** (*subst filterlim-at-bot-mirror*)
qed

lemma *cosh-real-at-top*: *filterlim* (*cosh* :: *real* ⇒ *real*) *at-top at-top*
proof −
have ∗: ((λ*x*. *exp* (− *x*)) ⟶ (*0*::*real*)) *at-top*
by (*intro filterlim-compose*[*OF exp-at-bot*] *filterlim-uminus-at-bot-at-top*)
have *filterlim* (λ*x*. (*1*/*2*) ∗ (*exp* (−*x*) + *exp x*) :: *real*) *at-top at-top*
by (*rule filterlim-tendsto-pos-mult-at-top*[*OF* - -
filterlim-tendsto-add-at-top[*OF* ∗]] *tendsto-const*)+ (*auto simp*: *exp-at-top*)
also have (λ*x*. (*1*/*2*) ∗ (*exp* (−*x*) + *exp x*) :: *real*) = *cosh*
by (*simp add*: *fun-eq-iff cosh-def*)
finally show *?thesis* **.**
qed

lemma *cosh-real-at-bot*: *filterlim* (*cosh* :: *real* ⇒ *real*) *at-top at-bot*
proof −
have *filterlim* (λ*x*. *cosh* (−*x*) :: *real*) *at-top at-top*
by (*simp add*: *cosh-real-at-top*)
thus *?thesis* **by** (*subst filterlim-at-bot-mirror*)
qed

lemma *tanh-real-at-top*: (*tanh* ⟶ (*1*::*real*)) *at-top*
proof −
have ((λ*x*::*real*. (*1* − *exp* (− *2* ∗ *x*)) / (*1* + *exp* (− *2* ∗ *x*))) ⟶ (*1* − *0*) / (*1* + *0*)) *at-top*
by (*intro tendsto-intros filterlim-compose*[*OF exp-at-bot*]
filterlim-tendsto-neg-mult-at-bot[*OF tendsto-const*] *filterlim-ident*) *auto*
also have (λ*x*::*real*. (*1* − *exp* (− *2* ∗ *x*)) / (*1* + *exp* (− *2* ∗ *x*))) = *tanh*
by (*rule ext*) (*simp add*: *tanh-real-altdef*)
finally show *?thesis* **by** *simp*
qed

lemma *tanh-real-at-bot*: (*tanh* ⟶ (−*1*::*real*)) *at-bot*
proof −
have ((λ*x*::*real*. −*tanh x*) ⟶ −*1*) *at-top*
by (*intro tendsto-minus tanh-real-at-top*)
also have (λ*x*. −*tanh x* :: *real*) = (λ*x*. *tanh* (−*x*)) **by** *simp*
finally show *?thesis* **by** (*subst filterlim-at-bot-mirror*)
qed

112.22.3 Properties of the inverse hyperbolic functions

lemma *isCont-sinh*: *isCont sinh* (*x* :: *'a* :: {*real-normed-field*, *banach*})
unfolding *sinh-def* [*abs-def*] **by** (*auto intro*!: *continuous-intros*)

lemma *isCont-cosh*: *isCont cosh* (*x* :: *'a* :: {*real-normed-field*, *banach*})
unfolding *cosh-def* [*abs-def*] **by** (*auto intro*!: *continuous-intros*)

lemma *isCont-tanh*: *cosh x* ≠ *0* ⟹ *isCont tanh* (*x* :: *'a* :: {*real-normed-field*, *banach*})
unfolding *tanh-def* [*abs-def*]
by (*auto intro*!: *continuous-intros isCont-divide isCont-sinh isCont-cosh*)

lemma *continuous-on-sinh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous-on A f*
shows *continuous-on A* (λ*x*. *sinh* (*f x*))
unfolding *sinh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-on-cosh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous-on A f*
shows *continuous-on A* (λ*x*. *cosh* (*f x*))
unfolding *cosh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-sinh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous F f*
shows *continuous F* (λ*x*. *sinh* (*f x*))
unfolding *sinh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-cosh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous F f*
shows *continuous F* (λ*x*. *cosh* (*f x*))
unfolding *cosh-def* **using** *assms* **by** (*intro continuous-intros*)

lemma *continuous-on-tanh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous-on A f* ⋀*x*. *x* ∈ *A* ⟹ *cosh* (*f x*) ≠ *0*
shows *continuous-on A* (λ*x*. *tanh* (*f x*))
unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros*) *auto*

lemma *continuous-at-within-tanh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous* (*at x within A*) *f cosh* (*f x*) ≠ *0*
shows *continuous* (*at x within A*) (λ*x*. *tanh* (*f x*))
unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros continuous-divide*) *auto*

lemma *continuous-tanh* [*continuous-intros*]:
fixes *f* :: - ⇒*'a*::{*real-normed-field*,*banach*}
assumes *continuous F f cosh* (*f* (*Lim F* (λ*x*. *x*))) ≠ *0*
shows *continuous F* (λ*x*. *tanh* (*f x*))

unfolding *tanh-def* **using** *assms* **by** (*intro continuous-intros continuous-divide*) *auto*

lemma *tendsto-sinh* [*tendsto-intros*]:
fixes *f* :: - ⇒'*a*::{*real-normed-field*,*banach*}
shows (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *sinh* (*f x*)) ⟶ *sinh a*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-sinh*])

lemma *tendsto-cosh* [*tendsto-intros*]:
fixes *f* :: - ⇒'*a*::{*real-normed-field*,*banach*}
shows (*f* ⟶ *a*) *F* ⟹ ((λ*x*. *cosh* (*f x*)) ⟶ *cosh a*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-cosh*])

lemma *tendsto-tanh* [*tendsto-intros*]:
fixes *f* :: - ⇒'*a*::{*real-normed-field*,*banach*}
shows (*f* ⟶ *a*) *F* ⟹ *cosh a* ≠ *0* ⟹ ((λ*x*. *tanh* (*f x*)) ⟶ *tanh a*) *F*
by (*rule isCont-tendsto-compose* [*OF isCont-tanh*])

lemma *arsinh-real-has-field-derivative* [*derivative-intros*]:
fixes *x* :: *real*
shows (*arsinh has-field-derivative* (*1* / (*sqrt* (*x* ^ *2* + *1*)))) (*at x within A*)
proof −
have *pos*: *1* + *x* ^ *2* > *0* **by** (*intro add-pos-nonneg*) *auto*
from *pos arsinh-real-aux*[*of x*] **show** *?thesis* **unfolding** *arsinh-def* [*abs-def*]
by (*auto intro*!: *derivative-eq-intros simp*: *powr-minus powr-half-sqrt field-split-simps*)
qed

lemma *arcosh-real-has-field-derivative* [*derivative-intros*]:
fixes *x* :: *real*
assumes *x* > *1*
shows (*arcosh has-field-derivative* (*1* / (*sqrt* (*x* ^ *2* − *1*)))) (*at x within A*)
proof −
from *assms* **have** *x* + *sqrt* (*x*^{2} − *1*) > *0* **by** (*simp add*: *add-pos-pos*)
thus *?thesis* **using** *assms* **unfolding** *arcosh-def* [*abs-def*]
by (*auto intro*!: *derivative-eq-intros*
simp: *powr-minus powr-half-sqrt field-split-simps power2-eq-1-iff*)
qed

lemma *artanh-real-has-field-derivative* [*derivative-intros*]:
(*artanh has-field-derivative* (*1* / (*1* − *x* ^ *2*))) (*at x within A*) **if**
|*x*| < *1* **for** *x* :: *real*
proof −
from *that* **have** − *1* < *x x* < *1* **by** *linarith*+
hence (*artanh has-field-derivative* (*4* − *4* ∗ *x*) / ((*1* + *x*) ∗ (*1* − *x*) ∗ (*1* − *x*) ∗ *4*))
(*at x within A*) **unfolding** *artanh-def* [*abs-def*]
by (*auto intro*!: *derivative-eq-intros simp*: *powr-minus powr-half-sqrt*)
also have (*4* − *4* ∗ *x*) / ((*1* + *x*) ∗ (*1* − *x*) ∗ (*1* − *x*) ∗ *4*) = *1* / ((*1* + *x*) ∗

```
(1 − x))
    using ‹−1 < x› ‹x < 1› by (simp add: frac-eq-eq)
  also have (1 + x) * (1 − x) = 1 − x ^ 2
    by (simp add: algebra-simps power2-eq-square)
  finally show ?thesis .
qed

lemma cosh-double-cosh: cosh (2 * x :: 'a :: {banach, real-normed-field}) = 2 *
(cosh x)^2 − 1
  using cosh-double[of x] by (simp add: sinh-square-eq)

lemma sinh-multiple-reduce:
  sinh (x * numeral n :: 'a :: {real-normed-field, banach}) =
      sinh x * cosh (x * of-nat (pred-numeral n)) + cosh x * sinh (x * of-nat
(pred-numeral n))
proof −
  have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
    by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
  also have sinh (x * ...) = sinh (x * of-nat (pred-numeral n) + x)
    unfolding of-nat-Suc by (simp add: ring-distribs)
  finally show ?thesis
    by (simp add: sinh-add)
qed

lemma cosh-multiple-reduce:
  cosh (x * numeral n :: 'a :: {real-normed-field, banach}) =
     cosh (x * of-nat (pred-numeral n)) * cosh x + sinh (x * of-nat (pred-numeral
n)) * sinh x
proof −
  have numeral n = of-nat (pred-numeral n) + (1 :: 'a)
    by (metis add.commute numeral-eq-Suc of-nat-Suc of-nat-numeral)
  also have cosh (x * ...) = cosh (x * of-nat (pred-numeral n) + x)
    unfolding of-nat-Suc by (simp add: ring-distribs)
  finally show ?thesis
    by (simp add: cosh-add)
qed

lemma cosh-arcosh-real [simp]:
  assumes x ≥ (1 :: real)
  shows   cosh (arcosh x) = x
proof −
  have eventually (λt::real. cosh t ≥ x) at-top
    using cosh-real-at-top by (simp add: filterlim-at-top)
  then obtain t where t ≥ 1 cosh t ≥ x
    by (metis eventually-at-top-linorder linorder-not-le order-le-less)
  moreover have isCont cosh (y :: real) for y
    by (intro continuous-intros)
  ultimately obtain y where y ≥ 0 x = cosh y
    using IVT[of cosh 0 x t] assms by auto
```

thus *?thesis*
by (*simp add*: *arcosh-cosh-real*)
qed

lemma *arcosh-eq-0-iff-real* [*simp*]: $x \geq 1 \Longrightarrow arcosh\ x = 0 \longleftrightarrow x = (1 :: real)$
using *cosh-arcosh-real* **by** *fastforce*

lemma *arcosh-nonneg-real* [*simp*]:
assumes $x \geq 1$
shows $arcosh\ (x :: real) \geq 0$
proof −
have $1 + 0 \leq x + (x^2 - 1)\ powr\ (1 / 2)$
using *assms* **by** (*intro add-mono*) *auto*
thus *?thesis* **unfolding** *arcosh-def* **by** *simp*
qed

lemma *arcosh-real-strict-mono*:
fixes $x\ y :: real$
assumes $1 \leq x\ x < y$
shows $arcosh\ x < arcosh\ y$
proof −
have $cosh\ (arcosh\ x) < cosh\ (arcosh\ y)$
by (*subst* (*1 2*) *cosh-arcosh-real*) (*use assms* **in** *auto*)
thus *?thesis*
using *assms* **by** (*subst* (*asm*) *cosh-real-nonneg-less-iff*) *auto*
qed

lemma *arcosh-less-iff-real* [*simp*]:
fixes $x\ y :: real$
assumes $1 \leq x\ 1 \leq y$
shows $arcosh\ x < arcosh\ y \longleftrightarrow x < y$
using *arcosh-real-strict-mono*[*of x y*] *arcosh-real-strict-mono*[*of y x*] *assms*
by (*cases x y rule*: *linorder-cases*) *auto*

lemma *arcosh-real-gt-1-iff* [*simp*]: $x \geq 1 \Longrightarrow arcosh\ x > 0 \longleftrightarrow x \neq (1 :: real)$
using *arcosh-less-iff-real*[*of 1 x*] **by** (*auto simp del*: *arcosh-less-iff-real*)

lemma *sinh-arcosh-real*: $x \geq 1 \Longrightarrow sinh\ (arcosh\ x) = sqrt\ (x^2 - 1)$
by (*rule sym*, *rule real-sqrt-unique*) (*auto simp*: *sinh-square-eq*)

lemma *sinh-arsinh-real* [*simp*]: $sinh\ (arsinh\ x :: real) = x$
proof −
have *eventually* (λ*t*::*real*. *sinh t* ≥ *x*) *at-top*
using *sinh-real-at-top* **by** (*simp add*: *filterlim-at-top*)
then obtain *t* **where** *sinh t* ≥ *x*
by (*metis eventually-at-top-linorder linorder-not-le order-le-less*)
moreover have *eventually* (λ*t*::*real*. *sinh t* ≤ *x*) *at-bot*
using *sinh-real-at-bot* **by** (*simp add*: *filterlim-at-bot*)

then obtain t' **where** $t' \leq t$ *sinh* $t' \leq x$
by (*metis eventually-at-bot-linorder nle-le*)
moreover have *isCont sinh* (*y* :: *real*) **for** *y*
by (*intro continuous-intros*)
ultimately obtain *y* **where** $x = sinh\ y$
using *IVT*[*of sinh t' x t*] **by** *auto*
thus *?thesis*
by (*simp add*: *arsinh-sinh-real*)
qed

lemma *arsinh-real-strict-mono*:
fixes *x y* :: *real*
assumes $x < y$
shows $arsinh\ x < arsinh\ y$
proof −
have $sinh\ (arsinh\ x) < sinh\ (arsinh\ y)$
by (*subst* (*1 2*) *sinh-arsinh-real*) (*use assms* **in** *auto*)
thus *?thesis*
using *assms* **by** (*subst* (*asm*) *sinh-real-less-iff*) *auto*
qed

lemma *arsinh-less-iff-real* [*simp*]:
fixes *x y* :: *real*
shows $arsinh\ x < arsinh\ y \longleftrightarrow x < y$
using *arsinh-real-strict-mono*[*of x y*] *arsinh-real-strict-mono*[*of y x*]
by (*cases x y rule*: *linorder-cases*) *auto*

lemma *arsinh-real-eq-0-iff* [*simp*]: $arsinh\ x = 0 \longleftrightarrow x = (0 :: real)$
by (*metis arsinh-0 sinh-arsinh-real*)

lemma *arsinh-real-pos-iff* [*simp*]: $arsinh\ x > 0 \longleftrightarrow x > (0 :: real)$
using *arsinh-less-iff-real*[*of 0 x*] **by** (*simp del*: *arsinh-less-iff-real*)

lemma *arsinh-real-neg-iff* [*simp*]: $arsinh\ x < 0 \longleftrightarrow x < (0 :: real)$
using *arsinh-less-iff-real*[*of x 0*] **by** (*simp del*: *arsinh-less-iff-real*)

lemma *cosh-arsinh-real*: $cosh\ (arsinh\ x) = sqrt\ (x^2 + 1)$
by (*rule sym*, *rule real-sqrt-unique*) (*auto simp*: *cosh-square-eq*)

lemma *continuous-on-arsinh* [*continuous-intros*]: *continuous-on A* (*arsinh* :: *real* ⇒ *real*)
by (*rule DERIV-continuous-on derivative-intros*)+

lemma *continuous-on-arcosh* [*continuous-intros*]:
assumes $A \subseteq \{1..\}$
shows *continuous-on A* (*arcosh* :: *real* ⇒ *real*)
proof −
have *pos*: $x + sqrt\ (x\ \hat{}\ 2 - 1) > 0$ **if** $x \geq 1$ **for** *x*
using *that* **by** (*intro add-pos-nonneg*) *auto*

show *?thesis*
unfolding *arcosh-def* [*abs-def*]
by (*intro continuous-on-subset* [*OF - assms*] *continuous-on-ln continuous-on-add*
continuous-on-id continuous-on-powr')
(*auto dest*: *pos simp*: *powr-half-sqrt intro*!: *continuous-intros*)
qed

lemma *continuous-on-artanh* [*continuous-intros*]:
assumes $A \subseteq \{-1<..<1\}$
shows *continuous-on A* (*artanh* :: *real* ⇒ *real*)
unfolding *artanh-def* [*abs-def*]
by (*intro continuous-on-subset* [*OF - assms*]) (*auto intro*!: *continuous-intros*)

lemma *continuous-on-arsinh'* [*continuous-intros*]:
fixes *f* :: *real* ⇒ *real*
assumes *continuous-on A f*
shows *continuous-on A* (λ*x*. *arsinh* (*f x*))
by (*rule continuous-on-compose2*[*OF continuous-on-arsinh assms*]) *auto*

lemma *continuous-on-arcosh'* [*continuous-intros*]:
fixes *f* :: *real* ⇒ *real*
assumes *continuous-on A f* ⋀*x*. *x* ∈ *A* ⟹ *f x* ≥ *1*
shows *continuous-on A* (λ*x*. *arcosh* (*f x*))
by (*rule continuous-on-compose2*[*OF continuous-on-arcosh assms*(*1*) *order.refl*])
(*use assms*(*2*) **in** *auto*)

lemma *continuous-on-artanh'* [*continuous-intros*]:
fixes *f* :: *real* ⇒ *real*
assumes *continuous-on A f* ⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ {−*1*<..<*1*}
shows *continuous-on A* (λ*x*. *artanh* (*f x*))
by (*rule continuous-on-compose2*[*OF continuous-on-artanh assms*(*1*) *order.refl*])
(*use assms*(*2*) **in** *auto*)

lemma *isCont-arsinh* [*continuous-intros*]: *isCont arsinh* (*x* :: *real*)
using *continuous-on-arsinh*[*of UNIV*] **by** (*auto simp*: *continuous-on-eq-continuous-at*)

lemma *isCont-arcosh* [*continuous-intros*]:
assumes *x* > *1*
shows *isCont arcosh* (*x* :: *real*)
proof −
have *continuous-on* {*1*::*real*<..} *arcosh*
by (*rule continuous-on-arcosh*) *auto*
with *assms* **show** *?thesis* **by** (*auto simp*: *continuous-on-eq-continuous-at*)
qed

lemma *isCont-artanh* [*continuous-intros*]:
assumes *x* > −*1 x* < *1*
shows *isCont artanh* (*x* :: *real*)
proof −

have *continuous-on* $\{-1<..<(1::real)\}$ *artanh*
by (*rule continuous-on-artanh*) *auto*
with *assms* **show** *?thesis* **by** (*auto simp*: *continuous-on-eq-continuous-at*)
qed

lemma *tendsto-arsinh* [*tendsto-intros*]: $(f \longrightarrow a)\ F \Longrightarrow ((\lambda x.\ arsinh\ (f\ x)) \longrightarrow arsinh\ a)\ F$
for $f :: - \Rightarrow real$
by (*rule isCont-tendsto-compose* [*OF isCont-arsinh*])

lemma *tendsto-arcosh-strong* [*tendsto-intros*]:
fixes $f :: - \Rightarrow real$
assumes $(f \longrightarrow a)\ F\ a \geq 1$ *eventually* $(\lambda x.\ f\ x \geq 1)\ F$
shows $((\lambda x.\ arcosh\ (f\ x)) \longrightarrow arcosh\ a)\ F$
by (*rule continuous-on-tendsto-compose*[*OF continuous-on-arcosh*[*OF order.refl*]])
(*use assms* **in** *auto*)

lemma *tendsto-arcosh*:
fixes $f :: - \Rightarrow real$
assumes $(f \longrightarrow a)\ F\ a > 1$
shows $((\lambda x.\ arcosh\ (f\ x)) \longrightarrow arcosh\ a)\ F$
by (*rule isCont-tendsto-compose* [*OF isCont-arcosh*]) (*use assms* **in** *auto*)

lemma *tendsto-arcosh-at-left-1*: $(arcosh \longrightarrow 0)\ (at\text{-}right\ (1::real))$
proof −
have $(arcosh \longrightarrow arcosh\ 1)\ (at\text{-}right\ (1::real))$
by (*rule tendsto-arcosh-strong*) (*auto simp*: *eventually-at intro*!: *exI*[*of - 1*])
thus *?thesis* **by** *simp*
qed

lemma *tendsto-artanh* [*tendsto-intros*]:
fixes $f :: 'a \Rightarrow real$
assumes $(f \longrightarrow a)\ F\ a > -1\ a < 1$
shows $((\lambda x.\ artanh\ (f\ x)) \longrightarrow artanh\ a)\ F$
by (*rule isCont-tendsto-compose* [*OF isCont-artanh*]) (*use assms* **in** *auto*)

lemma *continuous-arsinh* [*continuous-intros*]:
$continuous\ F\ f \Longrightarrow continuous\ F\ (\lambda x.\ arsinh\ (f\ x :: real))$
unfolding *continuous-def* **by** (*rule tendsto-arsinh*)

lemma *continuous-arcosh-strong* [*continuous-intros*]:
assumes *continuous F f eventually* $(\lambda x.\ f\ x \geq 1)\ F$
shows *continuous F* $(\lambda x.\ arcosh\ (f\ x :: real))$
proof (*cases* $F = bot$)
case *False*
show *?thesis*
unfolding *continuous-def*
proof (*intro tendsto-arcosh-strong*)

show $1 \leq f\ (Lim\ F\ (\lambda x.\ x))$
using *assms False* **unfolding** *continuous-def* **by** (*rule tendsto-lowerbound*)
qed (*insert assms*, *auto simp*: *continuous-def*)
qed *auto*

lemma *continuous-arcosh*:
$continuous\ F\ f \Longrightarrow f\ (Lim\ F\ (\lambda x.\ x)) > 1 \Longrightarrow continuous\ F\ (\lambda x.\ arcosh\ (f\ x :: real))$
unfolding *continuous-def* **by** (*rule tendsto-arcosh*) *auto*

lemma *continuous-artanh* [*continuous-intros*]:
$continuous\ F\ f \Longrightarrow f\ (Lim\ F\ (\lambda x.\ x)) \in \{-1<..<1\} \Longrightarrow continuous\ F\ (\lambda x.\ artanh\ (f\ x :: real))$
unfolding *continuous-def* **by** (*rule tendsto-artanh*) *auto*

lemma *arsinh-real-at-top*:
filterlim (*arsinh* :: *real* ⇒ *real*) *at-top at-top*
proof (*subst filterlim-cong*[*OF refl refl*])
show *filterlim* $(\lambda x.\ ln\ (x + sqrt\ (1 + x^2)))$ *at-top at-top*
by (*intro filterlim-compose*[*OF ln-at-top filterlim-at-top-add-at-top*] *filterlim-ident filterlim-compose*[*OF sqrt-at-top*] *filterlim-tendsto-add-at-top*[*OF tendsto-const*] *filterlim-pow-at-top*) *auto*
qed (*auto intro*!: *eventually-mono*[*OF eventually-ge-at-top*[*of 1*]] *simp*: *arsinh-real-def add-ac*)

lemma *arsinh-real-at-bot*:
filterlim (*arsinh* :: *real* ⇒ *real*) *at-bot at-bot*
proof −
have *filterlim* $(\lambda x{::}real.\ -arsinh\ x)$ *at-bot at-top*
by (*subst filterlim-uminus-at-top* [*symmetric*]) (*rule arsinh-real-at-top*)
also have $(\lambda x{::}real.\ -arsinh\ x) = (\lambda x.\ arsinh\ (-x))$ **by** *simp*
finally show *?thesis*
by (*subst filterlim-at-bot-mirror*)
qed

lemma *arcosh-real-at-top*:
filterlim (*arcosh* :: *real* ⇒ *real*) *at-top at-top*
proof (*subst filterlim-cong*[*OF refl refl*])
show *filterlim* $(\lambda x.\ ln\ (x + sqrt\ (-1 + x^2)))$ *at-top at-top*
by (*intro filterlim-compose*[*OF ln-at-top filterlim-at-top-add-at-top*] *filterlim-ident filterlim-compose*[*OF sqrt-at-top*] *filterlim-tendsto-add-at-top*[*OF tendsto-const*] *filterlim-pow-at-top*) *auto*
qed (*auto intro*!: *eventually-mono*[*OF eventually-ge-at-top*[*of 1*]] *simp*: *arcosh-real-def*)

lemma *artanh-real-at-left-1*:
filterlim (*artanh* :: *real* ⇒ *real*) *at-top* (*at-left 1*)
proof −

have $*$: *filterlim* (λx::*real*. (*1* + *x*) / (*1* − *x*)) *at-top* (*at-left 1*)
by (*rule LIM-at-top-divide*)
(*auto intro*!: *tendsto-eq-intros eventually-mono*[*OF eventually-at-left-real*[*of 0*]])
have *filterlim* (λx::*real*. (*1/2*) * *ln* ((*1* + *x*) / (*1* − *x*))) *at-top* (*at-left 1*)
by (*intro filterlim-tendsto-pos-mult-at-top*[*OF tendsto-const*] *
filterlim-compose[*OF ln-at-top*]) *auto*
also have (λx::*real*. (*1/2*) * *ln* ((*1* + *x*) / (*1* − *x*))) = *artanh*
by (*simp add*: *artanh-def* [*abs-def*])
finally show *?thesis* **.**
qed

lemma *artanh-real-at-right-1*:
filterlim (*artanh* :: *real* ⇒ *real*) *at-bot* (*at-right* (−*1*))
proof −
have *?thesis* ⟷ *filterlim* (λx::*real*. −*artanh x*) *at-top* (*at-right* (−*1*))
by (*simp add*: *filterlim-uminus-at-bot*)
also have ... ⟷ *filterlim* (λx::*real*. *artanh* (−*x*)) *at-top* (*at-right* (−*1*))
by (*intro filterlim-cong refl eventually-mono*[*OF eventually-at-right-real*[*of −1 1*]]) *auto*
also have ... ⟷ *filterlim* (*artanh* :: *real* ⇒ *real*) *at-top* (*at-left 1*)
by (*simp add*: *filterlim-at-left-to-right*)
also have ... **by** (*rule artanh-real-at-left-1*)
finally show *?thesis* **.**
qed

112.23 Simprocs for root and power literals

lemma *numeral-powr-numeral-real* [*simp*]:
numeral m powr numeral n = (*numeral m* ^ *numeral n* :: *real*)
by (*simp add*: *powr-numeral*)

context
begin

private lemma *sqrt-numeral-simproc-aux*:
assumes *m* * *m* ≡ *n*
shows *sqrt* (*numeral n* :: *real*) ≡ *numeral m*
proof −
have *numeral n* ≡ *numeral m* * (*numeral m* :: *real*) **by** (*simp add*: *assms* [*symmetric*])
moreover have *sqrt* ... ≡ *numeral m* **by** (*subst real-sqrt-abs2*) *simp*
ultimately show *sqrt* (*numeral n* :: *real*) ≡ *numeral m* **by** *simp*
qed

private lemma *root-numeral-simproc-aux*:
assumes *Num.pow m n* ≡ *x*
shows *root* (*numeral n*) (*numeral x* :: *real*) ≡ *numeral m*
by (*subst assms* [*symmetric*], *subst numeral-pow*, *subst real-root-pos2*) *simp-all*

private lemma *powr-numeral-simproc-aux*:
assumes *Num.pow y n = x*
shows *numeral x powr (m / numeral n :: real) ≡ numeral y powr m*
by (*subst assms* [*symmetric*], *subst numeral-pow*, *subst powr-numeral* [*symmetric*])
(*simp*, *subst powr-powr*, *simp-all*)

private lemma *numeral-powr-inverse-eq*:
numeral x powr (inverse (numeral n)) = numeral x powr (1 / numeral n :: real)
by *simp*

ML ‹

```
signature ROOT-NUMERAL-SIMPROC = sig

val sqrt : int option -> int -> int option
val sqrt' : int option -> int -> int option
val nth-root : int option -> int -> int -> int option
val nth-root' : int option -> int -> int -> int option
val sqrt-proc : Simplifier.proc
val root-proc : int * int -> Simplifier.proc
val powr-proc : int * int -> Simplifier.proc

end

structure Root-Numeral-Simproc : ROOT-NUMERAL-SIMPROC = struct

fun iterate NONE p f x =
      let
        fun go x = if p x then x else go (f x)
      in
        SOME (go x)
      end
  | iterate (SOME threshold) p f x =
      let
        fun go (threshold, x) =
          if p x then SOME x else if threshold = 0 then NONE else go (threshold -
1, f x)
      in
        go (threshold, x)
      end


fun nth-root - 1 x = SOME x
  | nth-root - - 0 = SOME 0
  | nth-root - - 1 = SOME 1
  | nth-root threshold n x =
  let
```

```
    fun newton-step y = ((n - 1) * y + x div Integer.pow (n - 1) y) div n
    fun is-root y = Integer.pow n y <= x andalso x < Integer.pow n (y + 1)
  in
    if x < n then
      SOME 1
    else if x < Integer.pow n 2 then
      SOME 1
    else
      let
       val y = Real.floor (Math.pow (Real.fromInt x, Real.fromInt 1 / Real.fromInt
n))
      in
        if is-root y then
          SOME y
        else
          iterate threshold is-root newton-step ((x + n - 1) div n)
      end
  end

fun nth-root' - 1 x = SOME x
  | nth-root' - - 0 = SOME 0
  | nth-root' - - 1 = SOME 1
  | nth-root' threshold n x = if x < n then NONE else if x < Integer.pow n 2 then
NONE else
      case nth-root threshold n x of
        NONE => NONE
      | SOME y => if Integer.pow n y = x then SOME y else NONE

fun sqrt - 0 = SOME 0
  | sqrt - 1 = SOME 1
  | sqrt threshold n =
    let
      fun aux (a, b) = if n >= b * b then aux (b, b * b) else (a, b)
      val (lower-root, lower-n) = aux (1, 2)
      fun newton-step x = (x + n div x) div 2
      fun is-sqrt r = r*r <= n andalso n < (r+1)*(r+1)
      val y = Real.floor (Math.sqrt (Real.fromInt n))
    in
      if is-sqrt y then
        SOME y
      else
          Option.mapPartial (iterate threshold is-sqrt newton-step o (fn x => x *
lower-root))
          (sqrt threshold (n div lower-n))
    end

fun sqrt' threshold x =
  case sqrt threshold x of
    NONE => NONE
```

```
  | SOME y => if y * y = x then SOME y else NONE

fun sqrt-proc ctxt ct =
  let
    val n = ct |> Thm.term-of |> dest-comb |> snd |> dest-comb |> snd |>
HOLogic.dest-numeral
  in
    case sqrt' (SOME 10000) n of
      NONE => NONE
    | SOME m =>
        SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HO-
Logic.mk-numeral) [m, n])
          @{thm sqrt-numeral-simproc-aux})
  end
    handle TERM - => NONE

fun root-proc (threshold1, threshold2) ctxt ct =
  let
    val [n, x] =
      ct |> Thm.term-of |> strip-comb |> snd |> map (dest-comb #> snd #>
HOLogic.dest-numeral)
  in
    if n > threshold1 orelse x > threshold2 then NONE else
      case nth-root' (SOME 100) n x of
        NONE => NONE
      | SOME m =>
          SOME (Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt o HO-
Logic.mk-numeral) [m, n, x])
            @{thm root-numeral-simproc-aux})
  end
    handle TERM - => NONE
         | Match => NONE

fun powr-proc (threshold1, threshold2) ctxt ct =
  let
    val eq-thm = Conv.try-conv (Conv.rewr-conv @{thm numeral-powr-inverse-eq})
ct
    val ct = Thm.dest-equals-rhs (Thm.cprop-of eq-thm)
    val (-, [x, t]) = strip-comb (Thm.term-of ct)
    val (-, [m, n]) = strip-comb t
    val [x, n] = map (dest-comb #> snd #> HOLogic.dest-numeral) [x, n]
  in
    if n > threshold1 orelse x > threshold2 then NONE else
      case nth-root' (SOME 100) n x of
        NONE => NONE
      | SOME y =>
          let
            val [y, n, x] = map HOLogic.mk-numeral [y, n, x]
            val thm = Thm.instantiate' [] (map (SOME o Thm.cterm-of ctxt) [y, n,
```

```
x, m])
            @{thm powr-numeral-simproc-aux}
        in
          SOME (@{thm transitive} OF [eq-thm, thm])
        end
  end
    handle TERM - => NONE
         | Match => NONE

end
›

end

simproc-setup sqrt-numeral (sqrt (numeral n)) =
  ‹K Root-Numeral-Simproc.sqrt-proc›

simproc-setup root-numeral (root (numeral n) (numeral x)) =
  ‹K (Root-Numeral-Simproc.root-proc (200, Integer.pow 200 2))›

simproc-setup powr-divide-numeral
  (numeral x powr (m / numeral n :: real) | numeral x powr (inverse (numeral n)
:: real)) =
    ‹K (Root-Numeral-Simproc.powr-proc (200, Integer.pow 200 2))›


lemma root 100 1267650600228229401496703205376 = 2
  by simp

lemma sqrt 196 = 14
  by simp

lemma 256 powr (7 / 4 :: real) = 16384
  by simp

lemma 27 powr (inverse 3) = (3::real)
  by simp

end
```

113 Complex Numbers: Rectangular and Polar Representations

```
theory Complex
imports Transcendental Real-Vector-Spaces
begin
```

We use the **codatatype** command to define the type of complex numbers.

This allows us to use **primcorec** to define complex functions by defining their real and imaginary result separately.

codatatype *complex* = *Complex* (*Re*: *real*) (*Im*: *real*)

lemma *complex-surj*: *Complex* (*Re z*) (*Im z*) = *z*
by (*rule complex.collapse*)

lemma *complex-eqI* [*intro?*]: *Re x* = *Re y* ⟹ *Im x* = *Im y* ⟹ *x* = *y*
by (*rule complex.expand*) *simp*

lemma *complex-eq-iff*: *x* = *y* ⟷ *Re x* = *Re y* ∧ *Im x* = *Im y*
by (*auto intro*: *complex.expand*)

113.1 Addition and Subtraction

instantiation *complex* :: *ab-group-add*
begin

primcorec *zero-complex*
where
Re 0 = *0*
| *Im 0* = *0*

primcorec *plus-complex*
where
Re (*x* + *y*) = *Re x* + *Re y*
| *Im* (*x* + *y*) = *Im x* + *Im y*

primcorec *uminus-complex*
where
Re (− *x*) = − *Re x*
| *Im* (− *x*) = − *Im x*

primcorec *minus-complex*
where
Re (*x* − *y*) = *Re x* − *Re y*
| *Im* (*x* − *y*) = *Im x* − *Im y*

instance
by *standard* (*simp-all add*: *complex-eq-iff*)

end

113.2 Multiplication and Division

instantiation *complex* :: *field*
begin

primcorec *one-complex*

where
  $Re\ 1 = 1$
| $Im\ 1 = 0$

primcorec *times-complex*
where
  $Re\ (x * y) = Re\ x * Re\ y - Im\ x * Im\ y$
| $Im\ (x * y) = Re\ x * Im\ y + Im\ x * Re\ y$

primcorec *inverse-complex*
where
  $Re\ (inverse\ x) = Re\ x\ /\ ((Re\ x)^2 + (Im\ x)^2)$
| $Im\ (inverse\ x) = -\ Im\ x\ /\ ((Re\ x)^2 + (Im\ x)^2)$

definition $x\ div\ y = x * inverse\ y$ **for** $x\ y :: complex$

instance
by *standard*
  (*simp-all add*: *complex-eq-iff divide-complex-def*
   distrib-left distrib-right right-diff-distrib left-diff-distrib
   power2-eq-square add-divide-distrib [*symmetric*])

end

lemma *Re-divide*: $Re\ (x\ /\ y) = (Re\ x * Re\ y + Im\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2)$
by (*simp add*: *divide-complex-def add-divide-distrib*)

lemma *Im-divide*: $Im\ (x\ /\ y) = (Im\ x * Re\ y - Re\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2)$
by (*simp add*: *divide-complex-def diff-divide-distrib*)

lemma *Complex-divide*:
  $(x\ /\ y) = Complex\ ((Re\ x * Re\ y + Im\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2))$
    $((Im\ x * Re\ y - Re\ x * Im\ y)\ /\ ((Re\ y)^2 + (Im\ y)^2))$
by (*metis Im-divide Re-divide complex-surj*)

lemma *Re-power2*: $Re\ (x\ \hat{}\ 2) = (Re\ x)\hat{}2 - (Im\ x)\hat{}2$
by (*simp add*: *power2-eq-square*)

lemma *Im-power2*: $Im\ (x\ \hat{}\ 2) = 2 * Re\ x * Im\ x$
by (*simp add*: *power2-eq-square*)

lemma *Re-power-real* [*simp*]: $Im\ x = 0 \Longrightarrow Re\ (x\ \hat{}\ n) = Re\ x\ \hat{}\ n$
by (*induct n*) *simp-all*

lemma *Im-power-real* [*simp*]: $Im\ x = 0 \Longrightarrow Im\ (x\ \hat{}\ n) = 0$
by (*induct n*) *simp-all*

113.3 Scalar Multiplication

instantiation *complex* :: *real-field*
begin

primcorec *scaleR-complex*
  where
    Re (*scaleR r x*) = *r* ∗ *Re x*
  | *Im* (*scaleR r x*) = *r* ∗ *Im x*

instance
proof
  fix *a b* :: *real* **and** *x y* :: *complex*
  show *scaleR a* (*x* + *y*) = *scaleR a x* + *scaleR a y*
    by (*simp add*: *complex-eq-iff distrib-left*)
  show *scaleR* (*a* + *b*) *x* = *scaleR a x* + *scaleR b x*
    by (*simp add*: *complex-eq-iff distrib-right*)
  show *scaleR a* (*scaleR b x*) = *scaleR* (*a* ∗ *b*) *x*
    by (*simp add*: *complex-eq-iff mult.assoc*)
  show *scaleR 1 x* = *x*
    by (*simp add*: *complex-eq-iff*)
  show *scaleR a x* ∗ *y* = *scaleR a* (*x* ∗ *y*)
    by (*simp add*: *complex-eq-iff algebra-simps*)
  show *x* ∗ *scaleR a y* = *scaleR a* (*x* ∗ *y*)
    by (*simp add*: *complex-eq-iff algebra-simps*)
qed

end

113.4 Numerals, Arithmetic, and Embedding from R

declare [[*coercion of-real* :: *real* ⇒ *complex*]]
declare [[*coercion of-rat* :: *rat* ⇒ *complex*]]
declare [[*coercion of-int* :: *int* ⇒ *complex*]]
declare [[*coercion of-nat* :: *nat* ⇒ *complex*]]

abbreviation *complex-of-nat*::*nat* ⇒ *complex*
  where *complex-of-nat* ≡ *of-nat*

abbreviation *complex-of-int*::*int* ⇒ *complex*
  where *complex-of-int* ≡ *of-int*

abbreviation *complex-of-rat*::*rat* ⇒ *complex*
  where *complex-of-rat* ≡ *of-rat*

abbreviation *complex-of-real* :: *real* ⇒ *complex*
  where *complex-of-real* ≡ *of-real*

lemma *complex-Re-of-nat* [*simp*]: *Re* (*of-nat n*) = *of-nat n*
  by (*induct n*) *simp-all*

lemma *complex-Im-of-nat* [*simp*]: *Im* (*of-nat n*) = *0*
by (*induct n*) *simp-all*

lemma *complex-Re-of-int* [*simp*]: *Re* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *complex-Im-of-int* [*simp*]: *Im* (*of-int z*) = *0*
by (*cases z rule*: *int-diff-cases*) *simp*

lemma *complex-Re-numeral* [*simp*]: *Re* (*numeral v*) = *numeral v*
using *complex-Re-of-int* [*of numeral v*] **by** *simp*

lemma *complex-Im-numeral* [*simp*]: *Im* (*numeral v*) = *0*
using *complex-Im-of-int* [*of numeral v*] **by** *simp*

lemma *Re-complex-of-real* [*simp*]: *Re* (*complex-of-real z*) = *z*
by (*simp add*: *of-real-def*)

lemma *Im-complex-of-real* [*simp*]: *Im* (*complex-of-real z*) = *0*
by (*simp add*: *of-real-def*)

lemma *Re-divide-numeral* [*simp*]: *Re* (*z* / *numeral w*) = *Re z* / *numeral w*
by (*simp add*: *Re-divide sqr-conv-mult*)

lemma *Im-divide-numeral* [*simp*]: *Im* (*z* / *numeral w*) = *Im z* / *numeral w*
by (*simp add*: *Im-divide sqr-conv-mult*)

lemma *Re-divide-of-nat* [*simp*]: *Re* (*z* / *of-nat n*) = *Re z* / *of-nat n*
by (*cases n*) (*simp-all add*: *Re-divide field-split-simps power2-eq-square del*: *of-nat-Suc*)

lemma *Im-divide-of-nat* [*simp*]: *Im* (*z* / *of-nat n*) = *Im z* / *of-nat n*
by (*cases n*) (*simp-all add*: *Im-divide field-split-simps power2-eq-square del*: *of-nat-Suc*)

lemma *Re-inverse* [*simp*]: $r \in \mathbb{R} \Longrightarrow$ *Re* (*inverse r*) = *inverse* (*Re r*)
by (*metis Re-complex-of-real Reals-cases of-real-inverse*)

lemma *Im-inverse* [*simp*]: $r \in \mathbb{R} \Longrightarrow$ *Im* (*inverse r*) = *0*
by (*metis Im-complex-of-real Reals-cases of-real-inverse*)

lemma *of-real-Re* [*simp*]: $z \in \mathbb{R} \Longrightarrow$ *of-real* (*Re z*) = *z*
by (*auto simp*: *Reals-def*)

lemma *complex-Re-fact* [*simp*]: *Re* (*fact n*) = *fact n*
proof −
have (*fact n* :: *complex*) = *of-real* (*fact n*)
by *simp*
also have *Re* ... = *fact n*
by (*subst Re-complex-of-real*) *simp-all*

finally show *?thesis* **.**
qed

lemma *surj-Re*: *surj Re*
by (*metis Re-complex-of-real surj-def*)

lemma *surj-Im*: *surj Im*
by (*metis complex.sel*(*2*) *surj-def*)

lemma *complex-Im-fact* [*simp*]: *Im* (*fact n*) = *0*
by (*metis complex-Im-of-nat of-nat-fact*)

lemma *Re-prod-Reals*: (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ ℝ) ⟹ *Re* (*prod f A*) = *prod* (λ*x*. *Re* (*f x*)) *A*
proof (*induction A rule*: *infinite-finite-induct*)
case (*insert x A*)
hence *Re* (*prod f* (*insert x A*)) = *Re* (*f x*) ∗ *Re* (*prod f A*) − *Im* (*f x*) ∗ *Im* (*prod f A*)
by *simp*
also from *insert.prems* **have** *f x* ∈ ℝ **by** *simp*
hence *Im* (*f x*) = *0* **by** (*auto elim*!: *Reals-cases*)
also have *Re* (*prod f A*) = (∏ *x*∈*A*. *Re* (*f x*))
by (*intro insert.IH insert.prems*) *auto*
finally show *?case* **using** *insert.hyps* **by** *simp*
qed *auto*

113.5 The Complex Number *i*

primcorec *imaginary-unit* :: *complex* (‹i›)
where
Re i = *0*
| *Im* i = *1*

lemma *Complex-eq*: *Complex a b* = *a* + i ∗ *b*
by (*simp add*: *complex-eq-iff*)

lemma *complex-eq*: *a* = *Re a* + i ∗ *Im a*
by (*simp add*: *complex-eq-iff*)

lemma *fun-complex-eq*: *f* = (λ*x*. *Re* (*f x*) + i ∗ *Im* (*f x*))
by (*simp add*: *fun-eq-iff complex-eq*)

lemma *i-squared* [*simp*]: i ∗ i = −*1*
by (*simp add*: *complex-eq-iff*)

lemma *power2-i* [*simp*]: i^2 = −*1*
by (*simp add*: *power2-eq-square*)

lemma *inverse-i* [*simp*]: *inverse* i = − i

by (*rule inverse-unique*) *simp*

lemma *divide-i* [*simp*]: x / i = − i ∗ x
by (*simp add*: *divide-complex-def*)

lemma *complex-i-mult-minus* [*simp*]: i ∗ (i ∗ x) = − x
by (*simp add*: *mult.assoc* [*symmetric*])

lemma *complex-i-not-zero* [*simp*]: i ≠ *0*
by (*simp add*: *complex-eq-iff*)

lemma *complex-i-not-one* [*simp*]: i ≠ *1*
by (*simp add*: *complex-eq-iff*)

lemma *complex-i-not-numeral* [*simp*]: i ≠ *numeral w*
by (*simp add*: *complex-eq-iff*)

lemma *complex-i-not-neg-numeral* [*simp*]: i ≠ − *numeral w*
by (*simp add*: *complex-eq-iff*)

lemma *complex-split-polar*: ∃ *r a*. z = *complex-of-real r* ∗ (*cos a* + i ∗ *sin a*)
by (*simp add*: *complex-eq-iff polar-Ex*)

lemma *i-even-power* [*simp*]: i ^ (*n* ∗ *2*) = (*−1*) ^ *n*
by (*metis mult.commute power2-i power-mult*)

lemma *i-even-power'* [*simp*]: *even n* ⟹ i ^ *n* = (*−1*) ^ (*n div 2*)
by (*metis dvd-mult-div-cancel power2-i power-mult*)

lemma *Re-i-times* [*simp*]: *Re* (i ∗ z) = − *Im z*
by *simp*

lemma *Im-i-times* [*simp*]: *Im* (i ∗ z) = *Re z*
by *simp*

lemma *i-times-eq-iff*: i ∗ w = z ⟷ w = − (i ∗ z)
by *auto*

lemma *divide-numeral-i* [*simp*]: z / (*numeral n* ∗ i) = − (i ∗ z) / *numeral n*
by (*metis divide-divide-eq-left divide-i mult.commute mult-minus-right*)

lemma *imaginary-eq-real-iff* [*simp*]:
assumes y ∈ *Reals x* ∈ *Reals*
shows i ∗ y = x ⟷ *x=0* ∧ *y=0*
by (*metis Im-complex-of-real Im-i-times assms mult-zero-right of-real-0 of-real-Re*)

lemma *real-eq-imaginary-iff* [*simp*]:
assumes y ∈ *Reals x* ∈ *Reals*
shows x = i ∗ y ⟷ *x=0* ∧ *y=0*

using *assms imaginary-eq-real-iff* **by** *fastforce*

113.6 Vector Norm

instantiation *complex* :: *real-normed-field*
begin

definition *norm z* = *sqrt* ((*Re z*)^2 + (*Im z*)^2)

abbreviation *cmod* :: *complex* ⇒ *real*
where *cmod* ≡ *norm*

definition *complex-sgn-def*: *sgn x* = *x* /$_R$ *cmod x*

definition *dist-complex-def*: *dist x y* = *cmod* (*x* − *y*)

definition *uniformity-complex-def*:
(*uniformity* :: (*complex* × *complex*) *filter*) = (*INF e*∈{*0* <..}. *principal* {(*x*, *y*). *dist x y* < *e*})

definition *open-complex-def*:
open (*U* :: *complex set*) ⟷ (∀ *x*∈*U*. *eventually* (λ(*x'*, *y*). *x'* = *x* ⟶ *y* ∈ *U*) *uniformity*)

instance
proof
fix *r* :: *real* **and** *x y* :: *complex* **and** *S* :: *complex set*
show (*norm x* = *0*) = (*x* = *0*)
by (*simp add*: *norm-complex-def complex-eq-iff*)
show *norm* (*x* + *y*) ≤ *norm x* + *norm y*
by (*simp add*: *norm-complex-def complex-eq-iff real-sqrt-sum-squares-triangle-ineq*)
show *norm* (*scaleR r x*) = |*r*| * *norm x*
by (*simp add*: *norm-complex-def complex-eq-iff power-mult-distrib distrib-left* [*symmetric*]
real-sqrt-mult)
show *norm* (*x* * *y*) = *norm x* * *norm y*
by (*simp add*: *norm-complex-def complex-eq-iff real-sqrt-mult* [*symmetric*]
power2-eq-square algebra-simps)
qed (*rule complex-sgn-def dist-complex-def open-complex-def uniformity-complex-def*)+

end

lemma *Re-divide'*: *Re* (*x* / *y*) = (*Re x* * *Re y* + *Im x* * *Im y*) / (*norm y*)^2
by (*simp add*: *Re-divide norm-complex-def*)

lemma *Im-divide'*: *Im* (*x* / *y*) = (*Im x* * *Re y* − *Re x* * *Im y*) / (*norm y*)^2
by (*simp add*: *Im-divide norm-complex-def*)

lemma *norm-ii* [*simp*]: *norm* i = *1*

by (*simp add*: *norm-complex-def*)

lemma *cmod-unit-one*: *cmod* (*cos a* + i ∗ *sin a*) = *1*
by (*simp add*: *norm-complex-def*)

lemma *cmod-complex-polar*: *cmod* (*r* ∗ (*cos a* + i ∗ *sin a*)) = |*r*|
by (*simp add*: *norm-mult cmod-unit-one*)

lemma *complex-Re-le-cmod*: *Re x* ≤ *cmod x*
unfolding *norm-complex-def* **by** (*rule real-sqrt-sum-squares-ge1*)

lemma *complex-mod-minus-le-complex-mod*: − *cmod x* ≤ *cmod x*
by (*rule order-trans* [*OF - norm-ge-zero*]) *simp*

lemma *complex-mod-triangle-ineq2*: *cmod* (*b* + *a*) − *cmod b* ≤ *cmod a*
by (*rule ord-le-eq-trans* [*OF norm-triangle-ineq2*]) *simp*

lemma *abs-Re-le-cmod*: |*Re x*| ≤ *cmod x*
by (*simp add*: *norm-complex-def*)

lemma *abs-Im-le-cmod*: |*Im x*| ≤ *cmod x*
by (*simp add*: *norm-complex-def*)

lemma *cmod-le*: *cmod z* ≤ |*Re z*| + |*Im z*|
using *norm-complex-def sqrt-sum-squares-le-sum-abs* **by** *presburger*

lemma *cmod-eq-Re*: *Im z* = *0* ⟹ *cmod z* = |*Re z*|
by (*simp add*: *norm-complex-def*)

lemma *cmod-eq-Im*: *Re z* = *0* ⟹ *cmod z* = |*Im z*|
by (*simp add*: *norm-complex-def*)

lemma *cmod-power2*: (*cmod z*)^2 = (*Re z*)^2 + (*Im z*)^2
by (*simp add*: *norm-complex-def*)

lemma *cmod-plus-Re-le-0-iff*: *cmod z* + *Re z* ≤ *0* ⟷ *Re z* = − *cmod z*
using *abs-Re-le-cmod*[*of z*] **by** *auto*

lemma *cmod-Re-le-iff*: *Im x* = *Im y* ⟹ *cmod x* ≤ *cmod y* ⟷ |*Re x*| ≤ |*Re y*|
by (*metis add.commute add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff*)

lemma *cmod-Im-le-iff*: *Re x* = *Re y* ⟹ *cmod x* ≤ *cmod y* ⟷ |*Im x*| ≤ |*Im y*|
by (*metis add-le-cancel-left norm-complex-def real-sqrt-abs real-sqrt-le-iff*)

lemma *Im-eq-0*: |*Re z*| = *cmod z* ⟹ *Im z* = *0*
by (*subst* (*asm*) *power-eq-iff-eq-base*[*symmetric*, **where** *n=2*]) (*auto simp add*: *norm-complex-def*)

lemma *abs-sqrt-wlog*: (⋀*x*. *x* ≥ *0* ⟹ *P x* (*x*^{2})) ⟹ *P* |*x*| (*x*^{2})

for *x*::*'a*::*linordered-idom*
by (*metis abs-ge-zero power2-abs*)

lemma *complex-abs-le-norm*: $|Re\ z| + |Im\ z| \leq sqrt\ 2 * norm\ z$
unfolding *norm-complex-def*
apply (*rule abs-sqrt-wlog* [**where** *x*=*Re z*])
apply (*rule abs-sqrt-wlog* [**where** *x*=*Im z*])
apply (*rule power2-le-imp-le*)
apply (*simp-all add*: *power2-sum add.commute sum-squares-bound real-sqrt-mult* [*symmetric*])
done

lemma *complex-unit-circle*: $z \neq 0 \Longrightarrow (Re\ z\ /\ cmod\ z)^2 + (Im\ z\ /\ cmod\ z)^2 = 1$
by (*simp add*: *norm-complex-def complex-eq-iff power2-eq-square add-divide-distrib* [*symmetric*])

Properties of complex signum.

lemma *sgn-eq*: *sgn z* = *z* / *complex-of-real* (*cmod z*)
by (*simp add*: *sgn-div-norm divide-inverse scaleR-conv-of-real mult.commute*)

lemma *Re-sgn* [*simp*]: *Re*(*sgn z*) = *Re*(*z*)/*cmod z*
by (*simp add*: *complex-sgn-def divide-inverse*)

lemma *Im-sgn* [*simp*]: *Im*(*sgn z*) = *Im*(*z*)/*cmod z*
by (*simp add*: *complex-sgn-def divide-inverse*)

113.7 Absolute value

instantiation *complex* :: *field-abs-sgn*
begin

definition *abs-complex* :: *complex* ⇒ *complex*
where *abs-complex* = *of-real* ∘ *norm*

instance
proof qed (*auto simp add*: *abs-complex-def complex-sgn-def norm-divide norm-mult scaleR-conv-of-real field-simps*)
end

113.8 Completeness of the Complexes

lemma *bounded-linear-Re*: *bounded-linear Re*
by (*rule bounded-linear-intro* [**where** *K*=*1*]) (*simp-all add*: *norm-complex-def*)

lemma *bounded-linear-Im*: *bounded-linear Im*
by (*rule bounded-linear-intro* [**where** *K*=*1*]) (*simp-all add*: *norm-complex-def*)

lemmas *Cauchy-Re* = *bounded-linear.Cauchy* [*OF bounded-linear-Re*]
lemmas *Cauchy-Im* = *bounded-linear.Cauchy* [*OF bounded-linear-Im*]

lemmas *tendsto-Re* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-Re*]
lemmas *tendsto-Im* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-Im*]
lemmas *isCont-Re* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-Re*]
lemmas *isCont-Im* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-Im*]
lemmas *continuous-Re* [*simp*] = *bounded-linear.continuous* [*OF bounded-linear-Re*]
lemmas *continuous-Im* [*simp*] = *bounded-linear.continuous* [*OF bounded-linear-Im*]
lemmas *continuous-on-Re* [*continuous-intros*] = *bounded-linear.continuous-on*[*OF bounded-linear-Re*]
lemmas *continuous-on-Im* [*continuous-intros*] = *bounded-linear.continuous-on*[*OF bounded-linear-Im*]
lemmas *has-derivative-Re* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Re*]
lemmas *has-derivative-Im* [*derivative-intros*] = *bounded-linear.has-derivative*[*OF bounded-linear-Im*]
lemmas *sums-Re* = *bounded-linear.sums* [*OF bounded-linear-Re*]
lemmas *sums-Im* = *bounded-linear.sums* [*OF bounded-linear-Im*]
lemmas *Re-suminf* = *bounded-linear.suminf*[*OF bounded-linear-Re*]
lemmas *Im-suminf* = *bounded-linear.suminf*[*OF bounded-linear-Im*]

lemma *continuous-on-Complex* [*continuous-intros*]:
continuous-on A f $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ *continuous-on A* (λx. *Complex* (*f x*) (*g x*))
unfolding *Complex-eq* **by** (*intro continuous-intros*)

lemma *tendsto-Complex* [*tendsto-intros*]:
(*f* $\longrightarrow$ *a*) *F* $\Longrightarrow$ (*g* $\longrightarrow$ *b*) *F* $\Longrightarrow$ ((λx. *Complex* (*f x*) (*g x*)) $\longrightarrow$ *Complex a b*) *F*
unfolding *Complex-eq* **by** (*auto intro*!: *tendsto-intros*)

lemma *tendsto-complex-iff*:
(*f* $\longrightarrow$ *x*) *F* $\longleftrightarrow$ (((λx. *Re* (*f x*)) $\longrightarrow$ *Re x*) *F* $\wedge$ ((λx. *Im* (*f x*)) $\longrightarrow$ *Im x*) *F*)
proof *safe*
assume ((λx. *Re* (*f x*)) $\longrightarrow$ *Re x*) *F* ((λx. *Im* (*f x*)) $\longrightarrow$ *Im x*) *F*
from *tendsto-Complex*[*OF this*] **show** (*f* $\longrightarrow$ *x*) *F*
unfolding *complex.collapse* **.**
qed (*auto intro*: *tendsto-intros*)

lemma *continuous-complex-iff*:
continuous F f $\longleftrightarrow$ *continuous F* (λx. *Re* (*f x*)) $\wedge$ *continuous F* (λx. *Im* (*f x*))
by (*simp only*: *continuous-def tendsto-complex-iff*)

lemma *continuous-on-of-real-o-iff* [*simp*]:
continuous-on S (λx. *complex-of-real* (*g x*)) = *continuous-on S g*
using *continuous-on-Re continuous-on-of-real* **by** *fastforce*

lemma *continuous-on-of-real-id* [*simp*]:
continuous-on S (*of-real* :: *real* $\Rightarrow$ *'a*::*real-normed-algebra-1*)
by (*rule continuous-on-of-real* [*OF continuous-on-id*])

lemma *has-vector-derivative-complex-iff*: (*f has-vector-derivative x*) *F* ⟷
((λ*x*. *Re* (*f x*)) *has-field-derivative* (*Re x*)) *F* ∧
((λ*x*. *Im* (*f x*)) *has-field-derivative* (*Im x*)) *F*
by (*simp add*: *has-vector-derivative-def has-field-derivative-def has-derivative-def tendsto-complex-iff algebra-simps bounded-linear-scaleR-left bounded-linear-mult-right*)

lemma *has-field-derivative-Re*[*derivative-intros*]:
(*f has-vector-derivative D*) *F* ⟹ ((λ*x*. *Re* (*f x*)) *has-field-derivative* (*Re D*)) *F*
unfolding *has-vector-derivative-complex-iff* **by** *safe*

lemma *has-field-derivative-Im*[*derivative-intros*]:
(*f has-vector-derivative D*) *F* ⟹ ((λ*x*. *Im* (*f x*)) *has-field-derivative* (*Im D*)) *F*
unfolding *has-vector-derivative-complex-iff* **by** *safe*

instance *complex* :: *banach*
proof
fix *X* :: *nat* ⇒ *complex*
assume *X*: *Cauchy X*
then have (λ*n*. *Complex* (*Re* (*X n*)) (*Im* (*X n*))) ⟶
Complex (*lim* (λ*n*. *Re* (*X n*))) (*lim* (λ*n*. *Im* (*X n*)))
by (*intro tendsto-Complex convergent-LIMSEQ-iff*[*THEN iffD1*]
Cauchy-convergent-iff[*THEN iffD1*] *Cauchy-Re Cauchy-Im*)
then show *convergent X*
unfolding *complex.collapse* **by** (*rule convergentI*)
qed

declare *DERIV-power*[**where** *'a*=*complex*, *unfolded of-nat-def*[*symmetric*], *derivative-intros*]

113.9 Complex Conjugation

primcorec *cnj* :: *complex* ⇒ *complex*
where
Re (*cnj z*) = *Re z*
| *Im* (*cnj z*) = − *Im z*

lemma *complex-cnj-cancel-iff* [*simp*]: *cnj x* = *cnj y* ⟷ *x* = *y*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-cnj* [*simp*]: *cnj* (*cnj z*) = *z*
by (*simp add*: *complex-eq-iff*)

lemma *in-image-cnj-iff*: *z* ∈ *cnj* ‘ *A* ⟷ *cnj z* ∈ *A*
by (*metis complex-cnj-cnj image-iff*)

lemma *image-cnj-conv-vimage-cnj*: *cnj* ‘ *A* = *cnj* −‘ *A*
using *in-image-cnj-iff* **by** *blast*

lemma *complex-cnj-zero* [*simp*]: *cnj 0 = 0*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-zero-iff* [*iff*]: *cnj z = 0 ⟷ z = 0*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-one-iff* [*simp*]: *cnj z = 1 ⟷ z = 1*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-add* [*simp*]: *cnj (x + y) = cnj x + cnj y*
by (*simp add*: *complex-eq-iff*)

lemma *cnj-sum* [*simp*]: *cnj (sum f s) = (∑ x∈s. cnj (f x))*
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *complex-cnj-diff* [*simp*]: *cnj (x − y) = cnj x − cnj y*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-minus* [*simp*]: *cnj (− x) = − cnj x*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-one* [*simp*]: *cnj 1 = 1*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-mult* [*simp*]: *cnj (x * y) = cnj x * cnj y*
by (*simp add*: *complex-eq-iff*)

lemma *cnj-prod* [*simp*]: *cnj (prod f s) = (∏ x∈s. cnj (f x))*
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *complex-cnj-inverse* [*simp*]: *cnj (inverse x) = inverse (cnj x)*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-divide* [*simp*]: *cnj (x / y) = cnj x / cnj y*
by (*simp add*: *divide-complex-def*)

lemma *complex-cnj-power* [*simp*]: *cnj (x ^ n) = cnj x ^ n*
by (*induct n*) *simp-all*

lemma *complex-cnj-of-nat* [*simp*]: *cnj (of-nat n) = of-nat n*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-of-int* [*simp*]: *cnj (of-int z) = of-int z*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-numeral* [*simp*]: *cnj (numeral w) = numeral w*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-neg-numeral* [*simp*]: *cnj (− numeral w) = − numeral w*

by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-scaleR* [*simp*]: *cnj* (*scaleR r x*) = *scaleR r* (*cnj x*)
by (*simp add*: *complex-eq-iff*)

lemma *complex-mod-cnj* [*simp*]: *cmod* (*cnj z*) = *cmod z*
by (*simp add*: *norm-complex-def*)

lemma *complex-cnj-complex-of-real* [*simp*]: *cnj* (*of-real x*) = *of-real x*
by (*simp add*: *complex-eq-iff*)

lemma *complex-cnj-i* [*simp*]: *cnj* i = − i
by (*simp add*: *complex-eq-iff*)

lemma *complex-add-cnj*: *z* + *cnj z* = *complex-of-real* (2 ∗ *Re z*)
by (*simp add*: *complex-eq-iff*)

lemma *complex-diff-cnj*: *z* − *cnj z* = *complex-of-real* (2 ∗ *Im z*) ∗ i
by (*simp add*: *complex-eq-iff*)

lemma *dist-cnj* [*simp*]: *dist* (*cnj a*) (*cnj b*) = *dist a b*
by (*metis complex-cnj-diff complex-mod-cnj dist-norm*)

lemma *Ints-cnj* [*intro*]: *x* ∈ ℤ ⟹ *cnj x* ∈ ℤ
by (*auto elim*!: *Ints-cases*)

lemma *cnj-in-Ints-iff* [*simp*]: *cnj x* ∈ ℤ ⟷ *x* ∈ ℤ
using *Ints-cnj*[*of x*] *Ints-cnj*[*of cnj x*] **by** *auto*

lemma *complex-mult-cnj*: *z* ∗ *cnj z* = *complex-of-real* ((*Re z*)^2 + (*Im z*)^2)
by (*simp add*: *complex-eq-iff power2-eq-square*)

lemma *cnj-add-mult-eq-Re*: *z* ∗ *cnj w* + *cnj z* ∗ *w* = 2 ∗ *Re* (*z* ∗ *cnj w*)
by (*rule complex-eqI*) *auto*

lemma *complex-mod-mult-cnj*: *cmod* (*z* ∗ *cnj z*) = (*cmod z*)^2
by (*simp add*: *norm-mult power2-eq-square*)

lemma *complex-mod-sqrt-Re-mult-cnj*: *cmod z* = *sqrt* (*Re* (*z* ∗ *cnj z*))
by (*simp add*: *norm-complex-def power2-eq-square*)

lemma *complex-In-mult-cnj-zero* [*simp*]: *Im* (*z* ∗ *cnj z*) = 0
by *simp*

lemma *complex-cnj-fact* [*simp*]: *cnj* (*fact n*) = *fact n*
by (*subst of-nat-fact* [*symmetric*], *subst complex-cnj-of-nat*) *simp*

lemma *complex-cnj-pochhammer* [*simp*]: *cnj* (*pochhammer z n*) = *pochhammer* (*cnj z*) *n*

by (*induct n arbitrary*: *z*) (*simp-all add*: *pochhammer-rec*)

lemma *bounded-linear-cnj*: *bounded-linear cnj*
using *complex-cnj-add complex-cnj-scaleR* **by** (*rule bounded-linear-intro* [**where** *K=1*]) *simp*

lemma *linear-cnj*: *linear cnj*
using *bounded-linear.linear*[*OF bounded-linear-cnj*] **.**

lemmas *tendsto-cnj* [*tendsto-intros*] = *bounded-linear.tendsto* [*OF bounded-linear-cnj*]
and *isCont-cnj* [*simp*] = *bounded-linear.isCont* [*OF bounded-linear-cnj*]
and *continuous-cnj* [*simp*, *continuous-intros*] = *bounded-linear.continuous* [*OF bounded-linear-cnj*]
and *continuous-on-cnj* [*simp*, *continuous-intros*] = *bounded-linear.continuous-on* [*OF bounded-linear-cnj*]
and *has-derivative-cnj* [*simp*, *derivative-intros*] = *bounded-linear.has-derivative* [*OF bounded-linear-cnj*]

lemma *lim-cnj*: ((λ*x*. *cnj*(*f x*)) ⟶ *cnj l*) *F* ⟷ (*f* ⟶ *l*) *F*
by (*simp add*: *tendsto-iff dist-complex-def complex-cnj-diff* [*symmetric*] *del*: *complex-cnj-diff*)

lemma *sums-cnj*: ((λ*x*. *cnj*(*f x*)) *sums cnj l*) ⟷ (*f sums l*)
by (*simp add*: *sums-def lim-cnj cnj-sum* [*symmetric*] *del*: *cnj-sum*)

lemma *differentiable-cnj-iff*:
(λ*z*. *cnj* (*f z*)) *differentiable at x within A* ⟷ *f differentiable at x within A*
proof
assume (λ*z*. *cnj* (*f z*)) *differentiable at x within A*
then obtain *D* **where** ((λ*z*. *cnj* (*f z*)) *has-derivative D*) (*at x within A*)
by (*auto simp*: *differentiable-def*)
from *has-derivative-cnj*[*OF this*] **show** *f differentiable at x within A*
by (*auto simp*: *differentiable-def*)
next
assume *f differentiable at x within A*
then obtain *D* **where** (*f has-derivative D*) (*at x within A*)
by (*auto simp*: *differentiable-def*)
from *has-derivative-cnj*[*OF this*] **show** (λ*z*. *cnj* (*f z*)) *differentiable at x within A*
by (*auto simp*: *differentiable-def*)
qed

lemma *has-vector-derivative-cnj* [*derivative-intros*]:
assumes (*f has-vector-derivative f′*) (*at z within A*)
shows ((λ*z*. *cnj* (*f z*)) *has-vector-derivative cnj f′*) (*at z within A*)
using *assms* **by** (*auto simp*: *has-vector-derivative-complex-iff intro*: *derivative-intros*)

lemma *has-field-derivative-cnj-cnj*:
assumes (*f has-field-derivative F*) (*at* (*cnj z*))

shows $((cnj \circ f \circ cnj)\ has\text{-}field\text{-}derivative\ cnj\ F)\ (at\ z)$
proof −
have $cnj\ -0\rightarrow\ cnj\ 0$
by (*subst lim-cnj*) *auto*
also have $cnj\ 0 = 0$
by *simp*
finally have $*$: *filterlim cnj* (*at 0*) (*at 0*)
by (*auto simp*: *filterlim-at eventually-at-filter*)
have $(\lambda h.\ (f\ (cnj\ z + cnj\ h) - f\ (cnj\ z))\ /\ cnj\ h)\ -0\rightarrow\ F$
by (*rule filterlim-compose*[*OF* - $*$]) (*use assms* **in** ‹*auto simp*: *DERIV-def*›)
thus *?thesis*
by (*subst* (*asm*) *lim-cnj* [*symmetric*]) (*simp add*: *DERIV-def*)
qed

113.10 Basic Lemmas

lemma *complex-of-real-code*[*code-unfold*]: *of-real* = $(\lambda x.\ Complex\ x\ 0)$
by (*intro ext*, *auto simp*: *complex-eq-iff*)

lemma *complex-eq-0*: $z{=}0 \longleftrightarrow (Re\ z)^2 + (Im\ z)^2 = 0$
by (*metis zero-complex.sel complex-eqI sum-power2-eq-zero-iff*)

lemma *complex-neq-0*: $z{\neq}0 \longleftrightarrow (Re\ z)^2 + (Im\ z)^2 > 0$
by (*metis complex-eq-0 less-numeral-extra*(*3*) *sum-power2-gt-zero-iff*)

lemma *complex-norm-square*: *of-real* $((norm\ z)^2) = z * cnj\ z$
by (*cases z*)
(*auto simp*: *complex-eq-iff norm-complex-def power2-eq-square*[*symmetric*] *of-real-power*[*symmetric*]
simp del: *of-real-power*)

lemma *complex-div-cnj*: $a\ /\ b = (a * cnj\ b)\ /\ (norm\ b)^2$
using *complex-norm-square* **by** *auto*

lemma *Re-complex-div-eq-0*: $Re\ (a\ /\ b) = 0 \longleftrightarrow Re\ (a * cnj\ b) = 0$
by (*auto simp add*: *Re-divide*)

lemma *Im-complex-div-eq-0*: $Im\ (a\ /\ b) = 0 \longleftrightarrow Im\ (a * cnj\ b) = 0$
by (*auto simp add*: *Im-divide*)

lemma *complex-div-gt-0*: $(Re\ (a\ /\ b) > 0 \longleftrightarrow Re\ (a * cnj\ b) > 0) \wedge (Im\ (a\ /\ b)$
$> 0 \longleftrightarrow Im\ (a * cnj\ b) > 0)$
proof (*cases* $b = 0$)
case *True*
then show *?thesis* **by** *auto*
next
case *False*
then have $0 < (Re\ b)^2 + (Im\ b)^2$
by (*simp add*: *complex-eq-iff sum-power2-gt-zero-iff*)
then show *?thesis*

by (*simp add*: *Re-divide Im-divide zero-less-divide-iff*)
qed

lemma *Re-complex-div-gt-0*: $Re\ (a\ /\ b) > 0 \longleftrightarrow Re\ (a * cnj\ b) > 0$
and *Im-complex-div-gt-0*: $Im\ (a\ /\ b) > 0 \longleftrightarrow Im\ (a * cnj\ b) > 0$
using *complex-div-gt-0* **by** *auto*

lemma *Re-complex-div-ge-0*: $Re\ (a\ /\ b) \geq 0 \longleftrightarrow Re\ (a * cnj\ b) \geq 0$
by (*metis le-less Re-complex-div-eq-0 Re-complex-div-gt-0*)

lemma *Im-complex-div-ge-0*: $Im\ (a\ /\ b) \geq 0 \longleftrightarrow Im\ (a * cnj\ b) \geq 0$
by (*metis Im-complex-div-eq-0 Im-complex-div-gt-0 le-less*)

lemma *Re-complex-div-lt-0*: $Re\ (a\ /\ b) < 0 \longleftrightarrow Re\ (a * cnj\ b) < 0$
by (*metis less-asym neq-iff Re-complex-div-eq-0 Re-complex-div-gt-0*)

lemma *Im-complex-div-lt-0*: $Im\ (a\ /\ b) < 0 \longleftrightarrow Im\ (a * cnj\ b) < 0$
by (*metis Im-complex-div-eq-0 Im-complex-div-gt-0 less-asym neq-iff*)

lemma *Re-complex-div-le-0*: $Re\ (a\ /\ b) \leq 0 \longleftrightarrow Re\ (a * cnj\ b) \leq 0$
by (*metis not-le Re-complex-div-gt-0*)

lemma *Im-complex-div-le-0*: $Im\ (a\ /\ b) \leq 0 \longleftrightarrow Im\ (a * cnj\ b) \leq 0$
by (*metis Im-complex-div-gt-0 not-le*)

lemma *Re-divide-of-real* [*simp*]: $Re\ (z\ /\ of\text{-}real\ r) = Re\ z\ /\ r$
by (*simp add*: *Re-divide power2-eq-square*)

lemma *Im-divide-of-real* [*simp*]: $Im\ (z\ /\ of\text{-}real\ r) = Im\ z\ /\ r$
by (*simp add*: *Im-divide power2-eq-square*)

lemma *Re-divide-Reals* [*simp*]: $r \in \mathbb{R} \Longrightarrow Re\ (z\ /\ r) = Re\ z\ /\ Re\ r$
by (*metis Re-divide-of-real of-real-Re*)

lemma *Im-divide-Reals* [*simp*]: $r \in \mathbb{R} \Longrightarrow Im\ (z\ /\ r) = Im\ z\ /\ Re\ r$
by (*metis Im-divide-of-real of-real-Re*)

lemma *Re-sum*[*simp*]: $Re\ (sum\ f\ s) = (\sum x{\in}s.\ Re\ (f\ x))$
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *Im-sum*[*simp*]: $Im\ (sum\ f\ s) = (\sum x{\in}s.\ Im(f\ x))$
by (*induct s rule*: *infinite-finite-induct*) *auto*

lemma *Rats-complex-of-real-iff* [*iff*]: $complex\text{-}of\text{-}real\ x \in \mathbb{Q} \longleftrightarrow x \in \mathbb{Q}$
proof −
have $\bigwedge a\ b.\ [\![0 < b;\ x = complex\text{-}of\text{-}int\ a\ /\ complex\text{-}of\text{-}int\ b]\!] \Longrightarrow x \in \mathbb{Q}$
by (*metis Rats-divide Rats-of-int Re-complex-of-real Re-divide-of-real of-real-of-int-eq*)
then show *?thesis*
by (*auto simp*: *elim*!: *Rats-cases'*)

qed

lemma *sum-Re-le-cmod*: $(\sum i{\in}I.\ Re\ (z\ i)) \leq cmod\ (\sum i{\in}I.\ z\ i)$
by (*metis Re-sum complex-Re-le-cmod*)

lemma *sum-Im-le-cmod*: $(\sum i{\in}I.\ Im\ (z\ i)) \leq cmod\ (\sum i{\in}I.\ z\ i)$
by (*smt* (*verit*, *best*) *Im-sum abs-Im-le-cmod sum.cong*)

lemma *sums-complex-iff*: $f\ sums\ x \longleftrightarrow ((\lambda x.\ Re\ (f\ x))\ sums\ Re\ x) \wedge ((\lambda x.\ Im\ (f\ x))\ sums\ Im\ x)$
unfolding *sums-def tendsto-complex-iff Im-sum Re-sum* **..**

lemma *summable-complex-iff*: $summable\ f \longleftrightarrow summable\ (\lambda x.\ Re\ (f\ x)) \wedge\ summable\ (\lambda x.\ Im\ (f\ x))$
unfolding *summable-def sums-complex-iff*[*abs-def*] **by** (*metis complex.sel*)

lemma *summable-complex-of-real* [*simp*]: $summable\ (\lambda n.\ complex\text{-}of\text{-}real\ (f\ n)) \longleftrightarrow summable\ f$
unfolding *summable-complex-iff* **by** *simp*

lemma *summable-Re*: $summable\ f \Longrightarrow summable\ (\lambda x.\ Re\ (f\ x))$
unfolding *summable-complex-iff* **by** *blast*

lemma *summable-Im*: $summable\ f \Longrightarrow summable\ (\lambda x.\ Im\ (f\ x))$
unfolding *summable-complex-iff* **by** *blast*

lemma *complex-is-Nat-iff*: $z \in \mathbb{N} \longleftrightarrow Im\ z = 0 \wedge (\exists i.\ Re\ z = of\text{-}nat\ i)$
by (*auto simp*: *Nats-def complex-eq-iff*)

lemma *complex-is-Int-iff*: $z \in \mathbb{Z} \longleftrightarrow Im\ z = 0 \wedge (\exists i.\ Re\ z = of\text{-}int\ i)$
by (*auto simp*: *Ints-def complex-eq-iff*)

lemma *complex-is-Real-iff*: $z \in \mathbb{R} \longleftrightarrow Im\ z = 0$
by (*auto simp*: *Reals-def complex-eq-iff*)

lemma *sgn-complex-iff*: $sgn\ x = sgn\ (Re\ x) \longleftrightarrow x \in \mathbb{R}$
by (*metis Im-complex-of-real Im-sgn Reals-0 complex-is-Real-iff divide-eq-0-iff norm-eq-zero of-real-Re sgn-of-real*)

lemma *Reals-cnj-iff*: $z \in \mathbb{R} \longleftrightarrow cnj\ z = z$
by (*auto simp*: *complex-is-Real-iff complex-eq-iff*)

lemma *in-Reals-norm*: $z \in \mathbb{R} \Longrightarrow norm\ z = |Re\ z|$
by (*simp add*: *complex-is-Real-iff norm-complex-def*)

lemma *Re-Reals-divide*: $r \in \mathbb{R} \Longrightarrow Re\ (r\ /\ z) = Re\ r * Re\ z\ /\ (norm\ z)^2$
by (*simp add*: *Re-divide complex-is-Real-iff cmod-power2*)

lemma *Im-Reals-divide*: $r \in \mathbb{R} \Longrightarrow Im\ (r\ /\ z) = -Re\ r * Im\ z\ /\ (norm\ z)^2$

by (*simp add*: *Im-divide complex-is-Real-iff cmod-power2*)

lemma *series-comparison-complex*:
fixes f:: *nat* ⇒ *'a*::*banach*
assumes *sg*: *summable g*
and ⋀*n*. *g n* ∈ ℝ ⋀*n*. *Re* (*g n*) ≥ *0*
and *fg*: ⋀*n*. *n* ≥ *N* ⟹ *norm*(*f n*) ≤ *norm*(*g n*)
shows *summable f*
proof −
have *g*: ⋀*n*. *cmod* (*g n*) = *Re* (*g n*)
using *assms* **by** (*metis abs-of-nonneg in-Reals-norm*)
show *?thesis*
by (*metis fg g sg summable-comparison-test summable-complex-iff*)
qed

113.11 Polar Form for Complex Numbers

lemma *complex-unimodular-polar*:
assumes *norm z* = *1*
obtains *t* **where** *0* ≤ *t t* < *2* ∗ *pi z* = *Complex* (*cos t*) (*sin t*)
by (*metis cmod-power2 one-power2 complex-surj sincos-total-2pi* [*of Re z Im z*] *assms*)

113.11.1 $\cos\theta + i\sin\theta$

primcorec *cis* :: *real* ⇒ *complex*
where
Re (*cis a*) = *cos a*
| *Im* (*cis a*) = *sin a*

lemma *cis-zero* [*simp*]: *cis 0* = *1*
by (*simp add*: *complex-eq-iff*)

lemma *norm-cis* [*simp*]: *norm* (*cis a*) = *1*
by (*simp add*: *norm-complex-def*)

lemma *sgn-cis* [*simp*]: *sgn* (*cis a*) = *cis a*
by (*simp add*: *sgn-div-norm*)

lemma *cis-2pi* [*simp*]: *cis* (*2* ∗ *pi*) = *1*
by (*simp add*: *cis.ctr complex-eq-iff*)

lemma *cis-neq-zero* [*simp*]: *cis a* ≠ *0*
by (*metis norm-cis norm-zero zero-neq-one*)

lemma *cis-cnj*: *cnj* (*cis t*) = *cis* (−*t*)
by (*simp add*: *complex-eq-iff*)

lemma *cis-mult*: *cis a* ∗ *cis b* = *cis* (*a* + *b*)
by (*simp add*: *complex-eq-iff cos-add sin-add*)

lemma *DeMoivre*: $(cis\ a)\ \hat{}\ n = cis\ (real\ n * a)$
by (*induct n*) (*simp-all add*: *algebra-simps cis-mult*)

lemma *cis-inverse* [*simp*]: $inverse\ (cis\ a) = cis\ (-a)$
by (*simp add*: *complex-eq-iff*)

lemma *cis-divide*: $cis\ a\ /\ cis\ b = cis\ (a - b)$
by (*simp add*: *divide-complex-def cis-mult*)

lemma *cis-power-int*: $cis\ x\ powi\ n = cis\ (of\text{-}int\ n * x)$
by (*auto simp*: *power-int-def DeMoivre*)

lemma *complex-cnj-power-int* [*simp*]: $cnj\ (x\ powi\ n) = cnj\ x\ powi\ n$
by (*auto simp*: *power-int-def*)

lemma *divide-conv-cnj*: $norm\ z = 1 \Longrightarrow x\ /\ z = x * cnj\ z$
by (*metis complex-div-cnj div-by-1 mult-1 of-real-1 power2-eq-square*)

lemma *i-not-in-Reals* [*simp*, *intro*]: $\mathrm{i} \notin \mathbb{R}$
by (*auto simp*: *complex-is-Real-iff*)

lemma *cos-n-Re-cis-pow-n*: $cos\ (real\ n * a) = Re\ (cis\ a\ \hat{}\ n)$
by (*auto simp add*: *DeMoivre*)

lemma *sin-n-Im-cis-pow-n*: $sin\ (real\ n * a) = Im\ (cis\ a\ \hat{}\ n)$
by (*auto simp add*: *DeMoivre*)

lemma *cis-pi* [*simp*]: $cis\ pi = -1$
by (*simp add*: *complex-eq-iff*)

lemma *cis-pi-half*[*simp*]: $cis\ (pi\ /\ 2) = \mathrm{i}$
by (*simp add*: *cis.ctr complex-eq-iff*)

lemma *cis-minus-pi-half*[*simp*]: $cis\ (-(pi\ /\ 2)) = -\mathrm{i}$
by (*simp add*: *cis.ctr complex-eq-iff*)

lemma *cis-multiple-2pi*[*simp*]: $n \in \mathbb{Z} \Longrightarrow cis\ (2 * pi * n) = 1$
by (*auto elim*!: *Ints-cases simp*: *cis.ctr one-complex.ctr*)

lemma *minus-cis*: $-cis\ x = cis\ (x + pi)$
by (*simp flip*: *cis-mult*)

lemma *minus-cis'*: $-cis\ x = cis\ (x - pi)$
by (*simp flip*: *cis-divide*)

113.11.2 $r(\cos\theta + i\sin\theta)$

definition *rcis* :: *real* ⇒ *real* ⇒ *complex*

where *rcis r a = complex-of-real r * cis a*

lemma *Re-rcis* [*simp*]: *Re(rcis r a) = r * cos a*
by (*simp add*: *rcis-def*)

lemma *Im-rcis* [*simp*]: *Im(rcis r a) = r * sin a*
by (*simp add*: *rcis-def*)

lemma *rcis-Ex*: *∃ r a. z = rcis r a*
by (*simp add*: *complex-eq-iff polar-Ex*)

lemma *complex-mod-rcis* [*simp*]: *cmod (rcis r a) = |r|*
by (*simp add*: *rcis-def norm-mult*)

lemma *cis-rcis-eq*: *cis a = rcis 1 a*
by (*simp add*: *rcis-def*)

lemma *rcis-mult*: *rcis r1 a * rcis r2 b = rcis (r1 * r2) (a + b)*
by (*simp add*: *rcis-def cis-mult*)

lemma *rcis-zero-mod* [*simp*]: *rcis 0 a = 0*
by (*simp add*: *rcis-def*)

lemma *rcis-zero-arg* [*simp*]: *rcis r 0 = complex-of-real r*
by (*simp add*: *rcis-def*)

lemma *rcis-eq-zero-iff* [*simp*]: *rcis r a = 0 ⟷ r = 0*
by (*simp add*: *rcis-def*)

lemma *DeMoivre2*: *(rcis r a) ^ n = rcis (r ^ n) (real n * a)*
by (*simp add*: *rcis-def power-mult-distrib DeMoivre*)

lemma *rcis-inverse*: *inverse(rcis r a) = rcis (1 / r) (− a)*
by (*simp add*: *divide-inverse rcis-def*)

lemma *rcis-divide*: *rcis r1 a / rcis r2 b = rcis (r1 / r2) (a − b)*
by (*simp add*: *rcis-def cis-divide* [*symmetric*])

113.11.3 Complex exponential

lemma *exp-Reals-eq*:
assumes *z ∈ ℝ*
shows *exp z = of-real (exp (Re z))*
using *assms* **by** (*auto elim*!: *Reals-cases simp*: *exp-of-real*)

lemma *cis-conv-exp*: *cis b = exp (i * b)*
proof −
have *(i * complex-of-real b) ^ n /$_R$ fact n =*
*of-real (cos-coeff n * b^n) + i * of-real (sin-coeff n * b^n)*

for n :: *nat*
proof −
have i $\hat{}$ n = *fact* n $*_R$ (*cos-coeff* n + i $*$ *sin-coeff* n)
by (*induct* n)
(*simp-all add*: *sin-coeff-Suc cos-coeff-Suc complex-eq-iff Re-divide Im-divide field-simps*
power2-eq-square add-nonneg-eq-0-iff)
then show *?thesis*
by (*simp add*: *field-simps*)
qed
then show *?thesis*
using *sin-converges* [*of b*] *cos-converges* [*of b*]
by (*auto simp add*: *Complex-eq cis.ctr exp-def simp del*: *of-real-mult intro*!: *sums-unique sums-add sums-mult sums-of-real*)
qed

lemma *exp-eq-polar*: *exp* z = *exp* (*Re* z) $*$ *cis* (*Im* z)
unfolding *cis-conv-exp exp-of-real* [*symmetric*] *mult-exp-exp*
by (*cases* z) (*simp add*: *Complex-eq*)

lemma *Re-exp*: *Re* (*exp* z) = *exp* (*Re* z) $*$ *cos* (*Im* z)
unfolding *exp-eq-polar* **by** *simp*

lemma *Im-exp*: *Im* (*exp* z) = *exp* (*Re* z) $*$ *sin* (*Im* z)
unfolding *exp-eq-polar* **by** *simp*

lemma *norm-cos-sin* [*simp*]: *norm* (*Complex* (*cos* t) (*sin* t)) = *1*
by (*simp add*: *norm-complex-def*)

lemma *norm-exp-eq-Re* [*simp*]: *norm* (*exp* z) = *exp* (*Re* z)
by (*simp add*: *cis.code cmod-complex-polar exp-eq-polar Complex-eq*)

lemma *complex-exp-exists*: ∃ a r. z = *complex-of-real* r $*$ *exp* a
using *cis-conv-exp rcis-Ex rcis-def* **by** *force*

lemma *exp-pi-i* [*simp*]: *exp* (*of-real pi* $*$ i) = −*1*
by (*metis cis-conv-exp cis-pi mult.commute*)

lemma *exp-pi-i′* [*simp*]: *exp* (i $*$ *of-real pi*) = −*1*
using *cis-conv-exp cis-pi* **by** *auto*

lemma *exp-two-pi-i* [*simp*]: *exp* (*2* $*$ *of-real pi* $*$ i) = *1*
by (*simp add*: *exp-eq-polar complex-eq-iff*)

lemma *exp-two-pi-i′* [*simp*]: *exp* (i $*$ (*of-real pi* $*$ *2*)) = *1*
by (*metis exp-two-pi-i mult.commute*)

lemma *continuous-on-cis* [*continuous-intros*]:
continuous-on A f ⟹ *continuous-on* A (λx. *cis* (f x))

by (*auto simp*: *cis-conv-exp intro*!: *continuous-intros*)

lemma *tendsto-exp-0-Re-at-bot*: (*exp* ⟶ *0*) (*filtercomap Re at-bot*)
proof −
have ((λ*z*. *cmod* (*exp z*)) ⟶ *0*) (*filtercomap Re at-bot*)
by (*auto intro*!: *filterlim-filtercomapI exp-at-bot*)
thus *?thesis*
using *tendsto-norm-zero-iff* **by** *blast*
qed

lemma *filterlim-exp-at-infinity-Re-at-top*: *filterlim exp at-infinity* (*filtercomap Re at-top*)
proof −
have *filterlim* (λ*z*. *norm* (*exp z*)) *at-top* (*filtercomap Re at-top*)
by (*auto intro*!: *filterlim-filtercomapI exp-at-top*)
thus *?thesis*
using *filterlim-norm-at-top-imp-at-infinity* **by** *blast*
qed

lemma *tendsto-cis* [*tendsto-intros*]:
assumes (*f* ⟶ *x*) *F*
shows ((λ*u*. *cis* (*f u*)) ⟶ *cis x*) *F*
unfolding *cis-conv-exp* **by** (*intro tendsto-intros assms*)

lemma *tendsto-rcis* [*tendsto-intros*]:
assumes (*f* ⟶ *r*) *F* (*g* ⟶ *x*) *F*
shows ((λ*u*. *rcis* (*f u*) (*g u*)) ⟶ *rcis r x*) *F*
unfolding *rcis-def* **by** (*intro tendsto-intros assms*)

lemma *continuous-on-rcis* [*continuous-intros*]:
continuous-on A f ⟹ *continuous-on A g* ⟹ *continuous-on A* (λ*x*. *rcis* (*f x*) (*g x*))
unfolding *rcis-def* **by** (*intro continuous-intros*)

lemma *has-derivative-cis* [*derivative-intros*]:
assumes (*f has-derivative d*) (*at x within A*)
shows ((λ*x*. *cis* (*f x*)) *has-derivative* (λ*t*. *d t* ∗$_R$ (i ∗ *cis* (*f x*)))) (*at x within A*)
proof (*rule has-derivative-compose*[*OF assms*])
have *cis-eq*: *cis* = (λ*x*. *cos x* + i ∗ *sin x*)
by (*auto simp*: *complex-eq-iff cos-of-real sin-of-real*)
have (*cis has-vector-derivative* (i ∗ *cis* (*f x*))) (*at* (*f x*))
unfolding *cis-eq* **by** (*auto intro*!: *derivative-eq-intros simp*: *cos-of-real sin-of-real algebra-simps*)
thus (*cis has-derivative* (λ*a*. *a* ∗$_R$ (i ∗ *cis* (*f x*)))) (*at* (*f x*))
by (*simp add*: *has-vector-derivative-def*)
qed

113.11.4 Complex argument

definition *Arg* :: *complex* ⇒ *real*
where *Arg z* = (*if z* = *0 then 0 else* (*SOME a. sgn z* = *cis a* ∧ − *pi* < *a* ∧ *a* ≤ *pi*))

lemma *Arg-zero*: *Arg 0* = *0*
by (*simp add*: *Arg-def*)

lemma *cis-Arg-unique*:
assumes *sgn z* = *cis x* **and** −*pi* < *x* **and** *x* ≤ *pi*
shows *Arg z* = *x*
proof −
from *assms* **have** *z* ≠ *0* **by** *auto*
have (*SOME a. sgn z* = *cis a* ∧ −*pi* < *a* ∧ *a* ≤ *pi*) = *x*
proof
fix *a*
define *d* **where** *d* = *a* − *x*
assume *a*: *sgn z* = *cis a* ∧ − *pi* < *a* ∧ *a* ≤ *pi*
from *a assms* **have** − (*2*∗*pi*) < *d* ∧ *d* < *2*∗*pi*
unfolding *d-def* **by** *simp*
moreover
from *a assms* **have** *cos a* = *cos x* **and** *sin a* = *sin x*
by (*simp-all add*: *complex-eq-iff*)
then have *cos*: *cos d* = *1*
by (*simp add*: *d-def cos-diff*)
moreover from *cos* **have** *sin d* = *0*
by (*rule cos-one-sin-zero*)
ultimately have *d* = *0*
by (*auto simp*: *sin-zero-iff elim*!: *evenE dest*!: *less-2-cases*)
then show *a* = *x*
by (*simp add*: *d-def*)
qed (*simp add*: *assms del*: *Re-sgn Im-sgn*)
with ‹*z* ≠ *0*› **show** *Arg z* = *x*
by (*simp add*: *Arg-def*)
qed

lemma *Arg-correct*:
assumes *z* ≠ *0*
shows *sgn z* = *cis* (*Arg z*) ∧ −*pi* < *Arg z* ∧ *Arg z* ≤ *pi*
proof (*simp add*: *Arg-def assms*, *rule someI-ex*)
obtain *r a* **where** *z*: *z* = *rcis r a*
using *rcis-Ex* **by** *fast*
with *assms* **have** *r* ≠ *0* **by** *auto*
define *b* **where** *b* = (*if 0* < *r then a else a* + *pi*)
have *b*: *sgn z* = *cis b*
using ‹*r* ≠ *0*› **by** (*simp add*: *z b-def rcis-def of-real-def sgn-scaleR sgn-if complex-eq-iff*)
have *cis-2pi-nat*: *cis* (*2* ∗ *pi* ∗ *real-of-nat n*) = *1* **for** *n*
by (*induct n*) (*simp-all add*: *distrib-left cis-mult* [*symmetric*] *complex-eq-iff*)

have *cis-2pi-int*: *cis* (*2* ∗ *pi* ∗ *real-of-int x*) = *1* **for** *x*
by (*cases x rule*: *int-diff-cases*)
(*simp add*: *right-diff-distrib cis-divide* [*symmetric*] *cis-2pi-nat*)
define *c* **where** *c* = *b* − *2* ∗ *pi* ∗ *of-int* ⌈(*b* − *pi*) / (*2* ∗ *pi*)⌉
have *sgn z* = *cis c*
by (*simp add*: *b c-def cis-divide* [*symmetric*] *cis-2pi-int*)
moreover have − *pi* < *c* ∧ *c* ≤ *pi*
using *ceiling-correct* [*of* (*b* − *pi*) / (*2*∗*pi*)]
by (*simp add*: *c-def less-divide-eq divide-le-eq algebra-simps del*: *le-of-int-ceiling*)
ultimately show ∃ *a*. *sgn z* = *cis a* ∧ −*pi* < *a* ∧ *a* ≤ *pi*
by *fast*
qed

lemma *Arg-bounded*: − *pi* < *Arg z* ∧ *Arg z* ≤ *pi*
by (*cases z* = *0*) (*simp-all add*: *Arg-zero Arg-correct*)

lemma *cis-Arg*: *z* ≠ *0* ⟹ *cis* (*Arg z*) = *sgn z*
by (*simp add*: *Arg-correct*)

lemma *rcis-cmod-Arg*: *rcis* (*cmod z*) (*Arg z*) = *z*
by (*cases z* = *0*) (*simp-all add*: *rcis-def cis-Arg sgn-div-norm of-real-def*)

lemma *rcis-cnj*:
shows *cnj a* = *rcis* (*cmod a*) (− *Arg a*)
by (*metis cis-cnj complex-cnj-complex-of-real complex-cnj-mult rcis-cmod-Arg rcis-def*)

lemma *cos-Arg-i-mult-zero* [*simp*]: *y* ≠ *0* ⟹ *Re y* = *0* ⟹ *cos* (*Arg y*) = *0*
using *cis-Arg* [*of y*] **by** (*simp add*: *complex-eq-iff*)

lemma *Arg-ii* [*simp*]: *Arg* i = *pi*/*2*
by (*rule cis-Arg-unique*; *simp add*: *sgn-eq*)

lemma *Arg-minus-ii* [*simp*]: *Arg* (−i) = −*pi*/*2*
proof (*rule cis-Arg-unique*)
show *sgn* (− i) = *cis* (− *pi* / *2*)
by (*simp add*: *sgn-eq*)
show − *pi* / *2* ≤ *pi*
using *pi-not-less-zero* **by** *linarith*
qed *auto*

lemma *cos-Arg*: *z* ≠ *0* ⟹ *cos* (*Arg z*) = *Re z* / *norm z*
by (*metis Re-sgn cis.sel*(*1*) *cis-Arg*)

lemma *sin-Arg*: *z* ≠ *0* ⟹ *sin* (*Arg z*) = *Im z* / *norm z*
by (*metis Im-sgn cis.sel*(*2*) *cis-Arg*)

113.12 Complex n-th roots

lemma *bij-betw-roots-unity*:
assumes $n > 0$
shows *bij-betw* ($\lambda k.$ *cis* ($2 * pi * real\ k\ /\ real\ n$)) $\{..<n\}$ $\{z.\ z\ \hat{}\ n = 1\}$
(**is** *bij-betw ?f - -*)
unfolding *bij-betw-def*
proof (*intro conjI*)
show *inj*: *inj-on ?f* $\{..<n\}$ **unfolding** *inj-on-def*
proof (*safe*, *goal-cases*)
case (*1 k l*)
hence *kl*: $k < n\ l < n$ **by** *simp-all*
from *1* **have** *1* = *?f k* / *?f l* **by** *simp*
also have ... = *cis* (*2*pi*(*real k* − *real l*)/*n*)
using *assms* **by** (*simp add*: *field-simps cis-divide*)
finally have *cos* (*2*pi*(*real k* − *real l*) / *n*) = *1*
by (*simp add*: *complex-eq-iff*)
then obtain *m* :: *int* **where** *2* * *pi* * (*real k* − *real l*) / *real n* = *real-of-int m* * *2* * *pi*
by (*subst* (*asm*) *cos-one-2pi-int*) *blast*
hence *real-of-int* (*int k* − *int l*) = *real-of-int* (*m* * *int n*)
unfolding *of-int-diff of-int-mult* **using** *assms*
by (*simp add*: *nonzero-divide-eq-eq*)
also note *of-int-eq-iff*
finally have *: *abs m* * *n* = *abs* (*int k* − *int l*) **by** (*simp add*: *abs-mult*)
also have ... < *int n* **using** *kl* **by** *linarith*
finally have *m* = *0* **using** *assms* **by** *simp*
with * **show** *k* = *l* **by** *simp*
qed

have *subset*: *?f* ‘ $\{..<n\}$ ⊆ $\{z.\ z\ \hat{}\ n = 1\}$
proof *safe*
fix *k* :: *nat*
have *cis* (*2* * *pi* * *real k* / *real n*) ^ *n* = *cis* (*2* * *pi*) ^ *k*
using *assms* **by** (*simp add*: *DeMoivre mult-ac*)
also have *cis* (*2* * *pi*) = *1* **by** (*simp add*: *complex-eq-iff*)
finally show *?f k* ^ *n* = *1* **by** *simp*
qed

have *n* = *card* $\{..<n\}$ **by** *simp*
also from *assms* **and** *subset* **have** ... ≤ *card* $\{z$::*complex*. $z\ \hat{}\ n = 1\}$
by (*intro card-inj-on-le*[*OF inj*]) (*auto simp*: *finite-roots-unity*)
finally have *card*: *card* $\{z$::*complex*. $z\ \hat{}\ n = 1\}$ = *n*
using *assms* **by** (*intro antisym card-roots-unity*) *auto*

have *card* (*?f* ‘ $\{..<n\}$) = *card* $\{z$::*complex*. $z\ \hat{}\ n = 1\}$
using *card inj* **by** (*subst card-image*) *auto*
with *subset* **and** *assms* **show** *?f* ‘ $\{..<n\}$ = $\{z$::*complex*. $z\ \hat{}\ n = 1\}$
by (*intro card-subset-eq finite-roots-unity*) *auto*
qed

lemma *card-roots-unity-eq*:
assumes $n > 0$
shows *card* $\{z::complex.\ z \hat{} n = 1\} = n$
using *bij-betw-same-card* [*OF bij-betw-roots-unity* [*OF assms*]] **by** *simp*

lemma *bij-betw-nth-root-unity*:
fixes $c :: complex$ **and** $n :: nat$
assumes *c*: $c \neq 0$ **and** *n*: $n > 0$
defines $c' \equiv root\ n\ (norm\ c) * cis\ (Arg\ c\ /\ n)$
shows *bij-betw* $(\lambda z.\ c' * z)\ \{z.\ z \hat{} n = 1\}\ \{z.\ z \hat{} n = c\}$
proof −
have $c' \hat{} n = of\text{-}real\ (root\ n\ (norm\ c)) \hat{} n * cis\ (Arg\ c)$
unfolding *of-real-power* **using** *n* **by** (*simp add*: *c'-def power-mult-distrib DeMoivre*)
also from *n* **have** $root\ n\ (norm\ c) \hat{} n = norm\ c$ **by** *simp*
also from *c* **have** $of\text{-}real \ldots * cis\ (Arg\ c) = c$ **by** (*simp add*: *cis-Arg Complex.sgn-eq*)
finally have [*simp*]: $c' \hat{} n = c$ **.**

show *?thesis* **unfolding** *bij-betw-def inj-on-def*
proof *safe*
fix $z :: complex$ **assume** $z \hat{} n = 1$
hence $(c' * z) \hat{} n = c' \hat{} n$ **by** (*simp add*: *power-mult-distrib*)
also have $c' \hat{} n = of\text{-}real\ (root\ n\ (norm\ c)) \hat{} n * cis\ (Arg\ c)$
unfolding *of-real-power* **using** *n* **by** (*simp add*: *c'-def power-mult-distrib DeMoivre*)
also from *n* **have** $root\ n\ (norm\ c) \hat{} n = norm\ c$ **by** *simp*
also from *c* **have** $\ldots * cis\ (Arg\ c) = c$ **by** (*simp add*: *cis-Arg Complex.sgn-eq*)
finally show $(c' * z) \hat{} n = c$ **.**
next
fix *z* **assume** *z*: $c = z \hat{} n$
define z' **where** $z' = z\ /\ c'$
from *c* **and** *n* **have** $c' \neq 0$ **by** (*auto simp*: *c'-def*)
with *n c* **have** $z = c' * z'$ **and** $z' \hat{} n = 1$
by (*auto simp*: *z'-def power-divide z*)
thus $z \in (\lambda z.\ c' * z)\ `\ \{z.\ z \hat{} n = 1\}$ **by** *blast*
qed (*insert c n*, *auto simp*: *c'-def*)
qed

lemma *finite-nth-roots* [*intro*]:
assumes $n > 0$
shows *finite* $\{z::complex.\ z \hat{} n = c\}$
proof (*cases* $c = 0$)
case *True*
with *assms* **have** $\{z::complex.\ z \hat{} n = c\} = \{0\}$ **by** *auto*
thus *?thesis* **by** *simp*
next
case *False*

from *assms* **have** *finite* {*z*::*complex*. *z* ^ *n* = *1*} **by** (*intro finite-roots-unity*) *simp-all*
also have *?this* ⟷ *?thesis*
by (*rule bij-betw-finite*, *rule bij-betw-nth-root-unity*) *fact+*
finally show *?thesis* **.**
qed

lemma *card-nth-roots*:
assumes *c* ≠ *0* *n* > *0*
shows *card* {*z*::*complex*. *z* ^ *n* = *c*} = *n*
proof −
have *card* {*z*. *z* ^ *n* = *c*} = *card* {*z*::*complex*. *z* ^ *n* = *1*}
by (*rule sym*, *rule bij-betw-same-card*, *rule bij-betw-nth-root-unity*) *fact+*
also have ... = *n* **by** (*rule card-roots-unity-eq*) *fact+*
finally show *?thesis* **.**
qed

lemma *sum-roots-unity*:
assumes *n* > *1*
shows ∑{*z*::*complex*. *z* ^ *n* = *1*} = *0*
proof −
define ω **where** ω = *cis* (*2* ∗ *pi* / *real n*)
have [*simp*]: ω ≠ *1*
proof
assume ω = *1*
with *assms* **obtain** *k* :: *int* **where** *2* ∗ *pi* / *real n* = *2* ∗ *pi* ∗ *of-int k*
by (*auto simp*: ω*-def complex-eq-iff cos-one-2pi-int*)
with *assms* **have** *real n* ∗ *of-int k* = *1* **by** (*simp add*: *field-simps*)
also have *real n* ∗ *of-int k* = *of-int* (*int n* ∗ *k*) **by** *simp*
also have *1* = (*of-int 1* :: *real*) **by** *simp*
also note *of-int-eq-iff*
finally show *False* **using** *assms* **by** (*auto simp*: *zmult-eq-1-iff*)
qed

have (∑ *z* | *z* ^ *n* = *1*. *z* :: *complex*) = (∑ *k*<*n*. *cis* (*2* ∗ *pi* ∗ *real k* / *real n*))
using *assms* **by** (*intro sum.reindex-bij-betw* [*symmetric*] *bij-betw-roots-unity*) *auto*
also have ... = (∑ *k*<*n*. ω ^ *k*)
by (*intro sum.cong refl*) (*auto simp*: ω*-def DeMoivre mult-ac*)
also have ... = (ω ^ *n* − *1*) / (ω − *1*)
by (*subst geometric-sum*) *auto*
also have ω ^ *n* − *1* = *cis* (*2* ∗ *pi*) − *1* **using** *assms* **by** (*auto simp*: ω*-def DeMoivre*)
also have ... = *0* **by** (*simp add*: *complex-eq-iff*)
finally show *?thesis* **by** *simp*
qed

lemma *sum-nth-roots*:
assumes *n* > *1*

shows $\sum \{z::complex.\ z \mathbin{\hat{}} n = c\} = 0$
proof (*cases* $c = 0$)
case *True*
with *assms* **have** $\{z::complex.\ z \mathbin{\hat{}} n = c\} = \{0\}$ **by** *auto*
also have $\sum \ldots = 0$ **by** *simp*
finally show *?thesis* **.**
next
case *False*
define c' **where** $c' = root\ n\ (norm\ c) * cis\ (Arg\ c\ /\ n)$
from *False* **and** *assms* **have** $(\sum \{z.\ z \mathbin{\hat{}} n = c\}) = (\sum z \mid z \mathbin{\hat{}} n = 1.\ c' * z)$
by (*subst sum.reindex-bij-betw* [*OF bij-betw-nth-root-unity*, *symmetric*])
(*auto simp*: *sum-distrib-left finite-roots-unity c'-def*)
also from *assms* **have** $\ldots = 0$
by (*simp add*: *sum-distrib-left* [*symmetric*] *sum-roots-unity*)
finally show *?thesis* **.**
qed

113.13 Square root of complex numbers

primcorec *csqrt* :: *complex* ⇒ *complex*
where
Re (*csqrt z*) = *sqrt* ((*cmod z* + *Re z*) / *2*)
| *Im* (*csqrt z*) = (*if Im z* = *0 then 1 else sgn* (*Im z*)) * *sqrt* ((*cmod z* − *Re z*) / *2*)

lemma *csqrt-of-real-nonneg* [*simp*]: *Im x* = *0* ⟹ *Re x* ≥ *0* ⟹ *csqrt x* = *sqrt* (*Re x*)
by (*simp add*: *complex-eq-iff norm-complex-def*)

lemma *csqrt-of-real-nonpos* [*simp*]: *Im x* = *0* ⟹ *Re x* ≤ *0* ⟹ *csqrt x* = i * *sqrt* |*Re x*|
by (*simp add*: *complex-eq-iff norm-complex-def*)

lemma *of-real-sqrt*: *x* ≥ *0* ⟹ *of-real* (*sqrt x*) = *csqrt* (*of-real x*)
by (*simp add*: *complex-eq-iff norm-complex-def*)

lemma *csqrt-0* [*simp*]: *csqrt 0* = *0*
by *simp*

lemma *csqrt-1* [*simp*]: *csqrt 1* = *1*
by *simp*

lemma *csqrt-ii* [*simp*]: *csqrt* i = (*1* + i) / *sqrt 2*
by (*simp add*: *complex-eq-iff Re-divide Im-divide real-sqrt-divide real-div-sqrt*)

lemma *power2-csqrt*[*simp*,*algebra*]: $(csqrt\ z)^2 = z$
proof (*cases Im z* = *0*)
case *True*
then show *?thesis*

using *real-sqrt-pow2*[*of Re z*] *real-sqrt-pow2*[*of − Re z*]
by (*cases 0*::*real Re z rule*: *linorder-cases*)
(*simp-all add*: *complex-eq-iff Re-power2 Im-power2 power2-eq-square cmod-eq-Re*)
next
case *False*
moreover have *cmod z ∗ cmod z − Re z ∗ Re z = Im z ∗ Im z*
by (*simp add*: *norm-complex-def power2-eq-square*)
moreover have $|Re\ z| \leq cmod\ z$
by (*simp add*: *norm-complex-def*)
ultimately show *?thesis*
by (*simp add*: *Re-power2 Im-power2 complex-eq-iff real-sgn-eq*
field-simps real-sqrt-mult[*symmetric*] *real-sqrt-divide*)
qed

lemma *csqrt-power-even*:
assumes *even n*
shows *csqrt z ^ n = z ^ (n div 2)*
by (*metis assms dvd-mult-div-cancel power2-csqrt power-mult*)

lemma *norm-csqrt* [*simp*]: *norm* (*csqrt z*) = *sqrt* (*norm z*)
by (*metis abs-of-nonneg norm-ge-zero norm-mult power2-csqrt power2-eq-square real-sqrt-abs*)

lemma *csqrt-eq-0* [*simp*]: *csqrt z = 0 ⟷ z = 0*
by *auto* (*metis power2-csqrt power-eq-0-iff*)

lemma *csqrt-eq-1* [*simp*]: *csqrt z = 1 ⟷ z = 1*
by *auto* (*metis power2-csqrt power2-eq-1-iff*)

lemma *csqrt-principal*: *0 < Re* (*csqrt z*) ∨ *Re* (*csqrt z*) = *0* ∧ *0* ≤ *Im* (*csqrt z*)
by (*auto simp add*: *not-less cmod-plus-Re-le-0-iff Im-eq-0*)

lemma *Re-csqrt*: *0* ≤ *Re* (*csqrt z*)
by (*metis csqrt-principal le-less*)

lemma *csqrt-square*:
assumes *0 < Re b* ∨ (*Re b = 0* ∧ *0* ≤ *Im b*)
shows *csqrt* (*b^2*) = *b*
proof −
have *csqrt* (*b^2*) = *b* ∨ *csqrt* (*b^2*) = − *b*
by (*simp add*: *power2-eq-iff*[*symmetric*])
moreover have *csqrt* (*b^2*) ≠ −*b* ∨ *b = 0*
using *csqrt-principal*[*of b ^ 2*] *assms*
by (*intro disjCI notI*) (*auto simp*: *complex-eq-iff*)
ultimately show *?thesis*
by *auto*
qed

lemma *csqrt-unique*: $w^2 = z \Longrightarrow 0 < Re\ w \vee Re\ w = 0 \wedge 0 \leq Im\ w \Longrightarrow csqrt\ z$

$= w$
by (*auto simp*: *csqrt-square*)

lemma *csqrt-minus* [*simp*]:
assumes $Im\ x < 0 \vee (Im\ x = 0 \wedge 0 \leq Re\ x)$
shows $csqrt\ (-x) = \mathrm{i} * csqrt\ x$
proof −
have $csqrt\ ((\mathrm{i} * csqrt\ x)\hat{}2) = \mathrm{i} * csqrt\ x$
proof (*rule csqrt-square*)
have $Im\ (csqrt\ x) \leq 0$
using *assms* **by** (*auto simp add*: *cmod-eq-Re mult-le-0-iff field-simps complex-Re-le-cmod*)
then show $0 < Re\ (\mathrm{i} * csqrt\ x) \vee Re\ (\mathrm{i} * csqrt\ x) = 0 \wedge 0 \leq Im\ (\mathrm{i} * csqrt\ x)$
by (*auto simp add*: *Re-csqrt simp del*: *csqrt.simps*)
qed
also have $(\mathrm{i} * csqrt\ x)\hat{}2 = - x$
by (*simp add*: *power-mult-distrib*)
finally show *?thesis* **.**
qed

lemma *csqrt-neq-neg-real*:
assumes $Im\ x = 0\ Re\ x < 0$
shows $csqrt\ z \neq x$
using *csqrt-principal*[*of z*] *assms* **by** *auto*

lemma *csqrt-of-real*: $x \geq 0 \Longrightarrow csqrt\ (of\text{-}real\ x) = of\text{-}real\ (sqrt\ x)$
by (*rule csqrt-unique*) (*auto simp flip*: *of-real-power*)

lemma *csqrt-of-real′*: $csqrt\ (of\text{-}real\ x) = of\text{-}real\ (sqrt\ |x|) * (if\ x \geq 0\ then\ 1\ else\ \mathrm{i})$
by (*rule csqrt-unique*) (*auto simp flip*: *of-real-power simp*: *power-mult-distrib*)

lemma *csqrt-minus-Reals*:
assumes $x \in \mathbb{R}$
shows $csqrt\ (-x) = sgn\ (Re\ x) * \mathrm{i} * csqrt\ x$
proof (*cases* $Re\ x \geq 0$)
case *True*
then show *?thesis*
using *assms complex-is-Real-iff sgn-1-pos* **by** *force*
next
case *False*
then obtain $Im\ x = 0\ sgn\ (Re\ x) = -1$
using *assms complex-is-Real-iff* **by** *auto*
with *False* **show** *?thesis*
by *auto*
qed

lemmas *cmod-def* = *norm-complex-def*

lemma *Complex-simps*:

shows *Complex-eq-0*: $Complex\ a\ b = 0 \longleftrightarrow a = 0 \wedge b = 0$
and *complex-add*: $Complex\ a\ b + Complex\ c\ d = Complex\ (a + c)\ (b + d)$
and *complex-minus*: $-\ (Complex\ a\ b) = Complex\ (-\ a)\ (-\ b)$
and *complex-diff*: $Complex\ a\ b - Complex\ c\ d = Complex\ (a - c)\ (b - d)$
and *Complex-eq-1*: $Complex\ a\ b = 1 \longleftrightarrow a = 1 \wedge b = 0$
and *Complex-eq-neg-1*: $Complex\ a\ b = -\ 1 \longleftrightarrow a = -\ 1 \wedge b = 0$
and *complex-mult*: $Complex\ a\ b * Complex\ c\ d = Complex\ (a * c - b * d)\ (a * d + b * c)$
and *complex-inverse*: $inverse\ (Complex\ a\ b) = Complex\ (a\ /\ (a^2 + b^2))\ (-\ b\ /\ (a^2 + b^2))$
and *Complex-eq-numeral*: $Complex\ a\ b = numeral\ w \longleftrightarrow a = numeral\ w \wedge b = 0$
and *Complex-eq-neg-numeral*: $Complex\ a\ b = -\ numeral\ w \longleftrightarrow a = -\ numeral\ w \wedge b = 0$
and *complex-scaleR*: $scaleR\ r\ (Complex\ a\ b) = Complex\ (r * a)\ (r * b)$
and *Complex-eq-i*: $Complex\ x\ y = \mathrm{i} \longleftrightarrow x = 0 \wedge y = 1$
and *i-mult-Complex*: $\mathrm{i} * Complex\ a\ b = Complex\ (-\ b)\ a$
and *Complex-mult-i*: $Complex\ a\ b * \mathrm{i} = Complex\ (-\ b)\ a$
and *i-complex-of-real*: $\mathrm{i} * complex\text{-}of\text{-}real\ r = Complex\ 0\ r$
and *complex-of-real-i*: $complex\text{-}of\text{-}real\ r * \mathrm{i} = Complex\ 0\ r$
and *Complex-add-complex-of-real*: $Complex\ x\ y + complex\text{-}of\text{-}real\ r = Complex\ (x+r)\ y$
and *complex-of-real-add-Complex*: $complex\text{-}of\text{-}real\ r + Complex\ x\ y = Complex\ (r+x)\ y$
and *Complex-mult-complex-of-real*: $Complex\ x\ y * complex\text{-}of\text{-}real\ r = Complex\ (x*r)\ (y*r)$
and *complex-of-real-mult-Complex*: $complex\text{-}of\text{-}real\ r * Complex\ x\ y = Complex\ (r*x)\ (r*y)$
and *complex-eq-cancel-iff2*: $(Complex\ x\ y = complex\text{-}of\text{-}real\ xa) = (x = xa \wedge y = 0)$
and *complex-cnj*: $cnj\ (Complex\ a\ b) = Complex\ a\ (-\ b)$
and *Complex-sum'*: $sum\ (\lambda x.\ Complex\ (f\ x)\ 0)\ s = Complex\ (sum\ f\ s)\ 0$
and *Complex-sum*: $Complex\ (sum\ f\ s)\ 0 = sum\ (\lambda x.\ Complex\ (f\ x)\ 0)\ s$
and *complex-of-real-def*: $complex\text{-}of\text{-}real\ r = Complex\ r\ 0$
and *complex-norm*: $cmod\ (Complex\ x\ y) = sqrt\ (x^2 + y^2)$
by (*simp-all add*: *norm-complex-def field-simps complex-eq-iff Re-divide Im-divide*)

lemma *Complex-in-Reals*: $Complex\ x\ 0 \in \mathbb{R}$
by (*metis Reals-of-real complex-of-real-def*)

lemma *Complex-divide-complex-of-real*: $Complex\ x\ y\ /\ of\text{-}real\ r = Complex\ (x/r)\ (y/r)$
by (*metis complex-of-real-mult-Complex divide-inverse mult.commute of-real-inverse*)

lemma *cmod-neg-real*: $cmod\ (Complex\ (-x)\ y) = cmod\ (Complex\ x\ y)$
by (*metis complex-cnj complex-minus complex-mod-cnj norm-minus-cancel*)

Express a complex number as a linear combination of two others, not collinear with the origin

lemma *complex-axes*:
assumes *Im* $(y/x) \neq 0$
obtains a b **where** $z = \textit{of-real}\ a * x + \textit{of-real}\ b * y$
proof −
define *dd* **where** $dd \equiv \textit{Re}\ y * \textit{Im}\ x - \textit{Im}\ y * \textit{Re}\ x$
define a **where** $a = (\textit{Im}\ z * \textit{Re}\ y - \textit{Re}\ z * \textit{Im}\ y)\ /\ dd$
define b **where** $b = (\textit{Re}\ z * \textit{Im}\ x - \textit{Im}\ z * \textit{Re}\ x)\ /\ dd$
have $dd \neq 0$
using *assms* **by** (*auto simp*: *dd-def Im-complex-div-eq-0*)
have $a * \textit{Re}\ x + b * \textit{Re}\ y = \textit{Re}\ z$
using ‹$dd \neq 0$›
apply (*simp add*: *a-def b-def field-simps*)
by (*metis dd-def diff-add-cancel distrib-right mult.assoc mult.commute*)
moreover have $a * \textit{Im}\ x + b * \textit{Im}\ y = \textit{Im}\ z$
using ‹$dd \neq 0$›
apply (*simp add*: *a-def b-def field-simps*)
by (*metis* (*no-types*) *dd-def diff-add-cancel distrib-right mult.assoc mult.commute*)
ultimately have $z = \textit{of-real}\ a * x + \textit{of-real}\ b * y$
by (*simp add*: *complex-eqI*)
then show *?thesis* **using** *that* **by** *simp*
qed

end

114 MacLaurin and Taylor Series

theory *MacLaurin*
imports *Transcendental*
begin

114.1 Maclaurin's Theorem with Lagrange Form of Remainder

lemma *fact-diff-Suc*: $n < \textit{Suc}\ m \Longrightarrow \textit{fact}\ (\textit{Suc}\ m - n) = (\textit{Suc}\ m - n) * \textit{fact}\ (m - n)$
by (*subst fact-reduce*) *auto*

lemma *Maclaurin*:
assumes *h*: $0 < h$
and *n*: $0 < n$
and *diff-0*: *diff* $0 = f$
and *diff-Suc*: $\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq h \longrightarrow \textit{DERIV}\ (\textit{diff}\ m)\ t :> \textit{diff}\ (\textit{Suc}\ m)\ t$
shows
$\exists\, t{::}\textit{real}.\ 0 < t \wedge t < h \wedge$
$f\ h = (\sum m{<}n.\ \textit{diff}\ m\ 0\ /\ \textit{fact}\ m * h\hat{}m) + \textit{diff}\ n\ t\ /\ \textit{fact}\ n * h\hat{}n$
proof −

```
  define B where B = (f h − (∑ m<n. diff m 0 / fact m * h^m)) * fact n / h^n


  define difg where [abs-def]: difg m t =
    diff m t − (∑ p<n−m. diff (m + p) 0 / fact p * t^p) −
    B * (t ^ (n − m) / fact (n − m)) for m t

  have difg-Suc: DERIV (difg m) t :> difg (Suc m) t
    if mn: m < n and t0: 0 ≤ t and th: t ≤ h for m t
  proof−
    have DERIV (difg m) t :>
      diff (Suc m) t −
      (∑ x ∈ {..< n − m}. real x * t ^ (x − 1) * diff (m + x) 0 / fact x) −
      real (n − m) * t ^ (n − Suc m) * B / fact (n − m)
      by (auto simp: difg-def intro!: derivative-eq-intros diff-Suc[rule-format] mn t0
th)
    also from mn have {..< n − m} = insert 0 (Suc ' {..< n − Suc m})
      unfolding atLeast0LessThan[symmetric] by auto
    also have (∑ x ∈ .... real x * t ^ (x − 1) * diff (m + x) 0 / fact x) =
        (∑ x < n − Suc m. (1 + real x) * t ^ x * diff (m + 1 + x) 0 / fact (Suc
x))
      by (subst sum.insert) (auto simp: sum.reindex)
    also have ... = (∑ x < n − Suc m. diff (m + 1 + x) 0 * t^x / fact x)
      by (rule sum.cong) (simp-all add: nonzero-divide-eq-eq)
    also have (n − m) * t ^ (n − Suc m) * B / fact (n − m) = B * (t ^ (n −
Suc m) / fact (n − Suc m))
      using mn by (simp add: field-split-simps fact-reduce)
    finally show DERIV (difg m) t :> difg (Suc m) t
      by (simp add: difg-def)
  qed


  have myRolle: ∃ t. 0 < t ∧ t < h' ∧ DERIV (difg m) t :> 0
    if mn: m < n and eq: difg m 0 = difg m h' and h'0: 0 < h' and h'h: h' ≤ h
for m h'
  proof (intro Rolle[OF h'0 eq])
    show continuous-on {0..h'} (difg m)
    using mn h'h by (auto intro!: continuous-at-imp-continuous-on DERIV-isCont[OF
difg-Suc])
    fix x assume x0: 0 < x and xh': x < h'
    with h'h have 0 ≤ x x ≤ h by auto
    from difg-Suc[OF mn this]
    show difg m differentiable (at x)
      by (auto simp: real-differentiable-def)
  qed
  from n obtain m where nm: n = Suc m by (auto simp: gr0-conv-Suc)
  then have m < n by simp
  then have ∃ t. 0 < t ∧ t < h ∧ DERIV (difg m) t :> 0
  proof (induct m)
    case 0
```

```
    from n have difg 0 0 = difg 0 h by (auto simp: difg-def B-def diff-0)
    with n h show ?case
      by (auto intro!: myRolle)
  next
    case (Suc m)
    from Suc have dSm0: difg (Suc m) 0 = 0
      by (auto simp: difg-def)
    from Suc obtain t where t0: 0 < t and th: t < h and mt: DERIV (difg m)
t :> 0
      by auto
    from t0 th DERIV-unique[OF difg-Suc mt] Suc
    have difg (Suc m) 0 = difg (Suc m) t
      by (auto simp: dSm0)
    with t0 th ‹Suc m < n›
    have ∃ t'. 0 < t' ∧ t' < t ∧ DERIV (difg (Suc m)) t' :> 0
      by (auto intro!: myRolle)
    with th show ?case
      by auto
  qed
  from this nm DERIV-unique[OF difg-Suc]
  obtain t where 0 < t t < h difg n t = 0
    by auto
  then show ?thesis by (auto simp: B-def difg-def)
qed

lemma Maclaurin2:
  fixes n :: nat
    and h :: real
  assumes INIT1: 0 < h
    and INIT2: diff 0 = f
    and DERIV: ∀ m t. m < n ∧ 0 ≤ t ∧ t ≤ h ⟶ DERIV (diff m) t :> diff
(Suc m) t
  shows ∃ t. 0 < t ∧ t ≤ h ∧ f h = (∑ m<n. diff m 0 / fact m * h ^ m) + diff
n t / fact n * h ^ n
proof (cases n)
  case 0
  with INIT1 INIT2 show ?thesis by fastforce
next
  case Suc
  then have n > 0 by simp
  from Maclaurin [OF INIT1 this INIT2 DERIV]
  show ?thesis by fastforce
qed

lemma Maclaurin-minus:
  fixes n :: nat and h :: real
  assumes h < 0 0 < n diff 0 = f
    and DERIV: ∀ m t. m < n ∧ h ≤ t ∧ t ≤ 0 ⟶ DERIV (diff m) t :> diff
(Suc m) t
```

shows $\exists t.\ h < t \wedge t < 0 \wedge f\ h = (\sum m{<}n.\ diff\ m\ 0\ /\ fact\ m * h\ \hat{}\ m) + diff\ n\ t\ /\ fact\ n * h\ \hat{}\ n$
proof −

Transform *ABL′* into *derivative-intros* format.

note *DERIV′* = *DERIV-chain′*[*OF - DERIV*[*rule-format*], *THEN DERIV-cong*]
let *?sum* = $\lambda t.$
($\sum m{<}n.\ (-1)\ \hat{}\ m * diff\ m\ (-0)\ /\ (fact\ m) * (-h)\ \hat{}\ m) +$
$(-1)\ \hat{}\ n * diff\ n\ (-t)\ /\ (fact\ n) * (-h)\ \hat{}\ n$
from *assms* **have** $\exists t{>}0.\ t < -h \wedge f\ (-(-h)) =$ *?sum t*
by (*intro Maclaurin*) (*auto intro*!: *derivative-eq-intros DERIV′*)
then obtain *t* **where** $0 < t\ t < -h\ f\ (-(-h)) =$ *?sum t*
by *blast*
moreover have $(-1)\ \hat{}\ n * diff\ n\ (-t) * (-h)\ \hat{}\ n\ /\ fact\ n = diff\ n\ (-t) * h\ \hat{}\ n\ /\ fact\ n$
by (*auto simp*: *power-mult-distrib*[*symmetric*])
moreover
have ($\sum m{<}n.\ (-1)\ \hat{}\ m * diff\ m\ 0 * (-h)\ \hat{}\ m\ /\ fact\ m) = (\sum m{<}n.\ diff\ m\ 0 * h\ \hat{}\ m\ /\ fact\ m)$
by (*auto intro*: *sum.cong simp add*: *power-mult-distrib*[*symmetric*])
ultimately have $h < -t \wedge -t < 0 \wedge$
$f\ h = (\sum m{<}n.\ diff\ m\ 0\ /\ (fact\ m) * h\ \hat{}\ m) + diff\ n\ (-t)\ /\ (fact\ n) * h\ \hat{}\ n$
by *auto*
then show *?thesis* **..**
qed

114.2 More Convenient "Bidirectional" Version.

lemma *Maclaurin-bi-le*:
fixes *n* :: *nat* **and** *x* :: *real*
assumes *diff 0* = *f*
and *DERIV* : $\forall m\ t.\ m < n \wedge |t| \leq |x| \longrightarrow$ *DERIV* (*diff m*) *t* :> *diff* (*Suc m*) *t*
shows $\exists t.\ |t| \leq |x| \wedge f\ x = (\sum m{<}n.\ diff\ m\ 0\ /\ fact\ m * x\ \hat{}\ m) + diff\ n\ t\ /\ fact\ n * x\ \hat{}\ n$
(**is** $\exists t.$ - $\wedge f\ x =$ *?f x t*)
proof (*cases n* = *0*)
case *True*
with ‹*diff 0* = *f*› **show** *?thesis* **by** *force*
next
case *False*
show *?thesis*
proof (*cases rule*: *linorder-cases*)
assume *x* = *0*
with ‹*n* ≠ *0*› ‹*diff 0* = *f*› *DERIV* **have** $|0| \leq |x| \wedge f\ x =$ *?f x 0*
by *auto*
then show *?thesis* **..**
next
assume *x* < *0*
with ‹*n* ≠ *0*› *DERIV* **have** $\exists t{>}x.\ t < 0 \wedge$ *diff 0 x* = *?f x t*

```
    by (intro Maclaurin-minus) auto
  then obtain t where x < t t < 0
    diff 0 x = (∑ m<n. diff m 0 / fact m * x ^ m) + diff n t / fact n * x ^ n
    by blast
  with ‹x < 0› ‹diff 0 = f› show ?thesis by force
 next
  assume x > 0
  with ‹n ≠ 0› ‹diff 0 = f› DERIV have ∃ t>0. t < x ∧ diff 0 x = ?f x t
    by (intro Maclaurin) auto
  then obtain t where 0 < t t < x
    diff 0 x = (∑ m<n. diff m 0 / fact m * x ^ m) + diff n t / fact n * x ^ n
    by blast
  with ‹x > 0› ‹diff 0 = f› have |t| ≤ |x| ∧ f x = ?f x t by simp
  then show ?thesis ..
 qed
qed

lemma Maclaurin-all-lt:
 fixes x :: real
 assumes INIT1: diff 0 = f
  and INIT2: 0 < n
  and INIT3: x ≠ 0
  and DERIV: ∀ m x. DERIV (diff m) x :> diff(Suc m) x
 shows ∃ t. 0 < |t| ∧ |t| < |x| ∧ f x =
    (∑ m<n. (diff m 0 / fact m) * x ^ m) + (diff n t / fact n) * x ^ n
  (is ∃ t. - ∧ - ∧ f x = ?f x t)
proof (cases rule: linorder-cases)
 assume x = 0
 with INIT3 show ?thesis ..
next
 assume x < 0
 with assms have ∃ t>x. t < 0 ∧ f x = ?f x t
   by (intro Maclaurin-minus) auto
 then show ?thesis by force
next
 assume x > 0
 with assms have ∃ t>0. t < x ∧ f x = ?f x t
   by (intro Maclaurin) auto
 then show ?thesis by force
qed

lemma Maclaurin-zero: x = 0 ⟹ n ≠ 0 ⟹ (∑ m<n. (diff m 0 / fact m) * x
^ m) = diff 0 0
 for x :: real and n :: nat
 by simp


lemma Maclaurin-all-le:
 fixes x :: real and n :: nat
```

assumes *INIT*: *diff 0 = f*
and *DERIV*: $\forall m\ x.$ *DERIV (diff m) x :> diff (Suc m) x*
shows $\exists t.\ |t| \leq |x| \wedge f\ x =$ ($\sum$ *m<n. (diff m 0 / fact m) * x ^ m) + (diff n t / fact n) * x ^ n*
(**is** $\exists t.$ *- ∧ f x = ?f x t*)
proof (*cases n = 0*)
case *True*
with *INIT* **show** *?thesis* **by** *force*
next
case *False*
show *?thesis*
using *DERIV INIT Maclaurin-bi-le* **by** *auto*
qed

lemma *Maclaurin-all-le-objl*:
diff 0 = f ∧ (∀ m x. DERIV (diff m) x :> diff (Suc m) x) ⟶
($\exists t$::*real*. $|t| \leq |x| \wedge f\ x =$ ($\sum$ *m<n. (diff m 0 / fact m) * x ^ m) + (diff n t / fact n) * x ^ n*)
for *x* :: *real* **and** *n* :: *nat*
by (*blast intro*: *Maclaurin-all-le*)

114.3 Version for Exponential Function

lemma *Maclaurin-exp-lt*:
fixes *x* :: *real* **and** *n* :: *nat*
shows
x ≠ 0 ⟹ n > 0 ⟹
($\exists t.\ 0 < |t| \wedge |t| < |x| \wedge$ *exp x =* ($\sum$ *m<n. x ^ m / fact m) + exp t / fact n * x ^ n*)
using *Maclaurin-all-lt* [**where** *diff = λn. exp* **and** *f = exp* **and** *x = x* **and** *n = n*] **by** *auto*

lemma *Maclaurin-exp-le*:
fixes *x* :: *real* **and** *n* :: *nat*
shows $\exists t.\ |t| \leq |x| \wedge$ *exp x =* ($\sum$ *m<n. x ^ m / fact m) + exp t / fact n * x ^ n*
using *Maclaurin-all-le-objl* [**where** *diff = λn. exp* **and** *f = exp* **and** *x = x* **and** *n = n*] **by** *auto*

corollary *exp-lower-Taylor-quadratic*: *0 ≤ x ⟹ 1 + x + x²* / *2 ≤ exp x*
for *x* :: *real*
using *Maclaurin-exp-le* [*of x 3*] **by** (*auto simp*: *numeral-3-eq-3 power2-eq-square*)

corollary *ln-2-less-1*: *ln 2 <* (*1*::*real*)
by (*smt* (*verit*) *ln-eq-minus-one ln-le-minus-one*)

114.4 Version for Sine Function

lemma *mod-exhaust-less-4*: *m mod 4 = 0 ∨ m mod 4 = 1 ∨ m mod 4 = 2 ∨ m mod 4 = 3*

for $m :: nat$
by *auto*

lemma *sin-coeff-lemma*: $sin\text{-}coeff\ m = sin\ (0 + 1/2 * real\ m * pi) / fact\ m$
by (*auto simp*: *sin-add sin-zero-iff sin-coeff-def algebra-simps add-divide-distrib elim*!: *oddE*)

lemma *Maclaurin-sin-expansion2*:
$\exists t.\ |t| \leq |x| \wedge$
$sin\ x = (\sum m{<}n.\ sin\text{-}coeff\ m * x \hat{} m) + sin\ (t + 1/2 * real\ n * pi) / fact\ n * x \hat{} n$
unfolding *sin-coeff-lemma*
apply (*rule Maclaurin-all-le*)
by (*auto simp*: *ring-distribs add-divide-distrib sin-add cos-add intro*!: *derivative-eq-intros*)

lemma *Maclaurin-sin-expansion*:
$\exists t.\ sin\ x = (\sum m{<}n.\ sin\text{-}coeff\ m * x \hat{} m) + sin\ (t + 1/2 * real\ n * pi) / fact\ n * x \hat{} n$
using *Maclaurin-sin-expansion2* [*of x n*] **by** *blast*

lemma *Maclaurin-sin-expansion3*:
assumes $n > 0\ x > 0$
shows $\exists t.\ 0 < t \wedge t < x \wedge$
$sin\ x = (\sum m{<}n.\ sin\text{-}coeff\ m * x \hat{} m) + sin\ (t + 1/2 * real\ n * pi) / fact\ n * x \hat{} n$
unfolding *sin-coeff-lemma*
apply (*rule Maclaurin*)
by (*auto intro*!: *derivative-eq-intros simp*: *assms ring-distribs sin-add cos-add*)

lemma *Maclaurin-sin-expansion4*:
assumes $0 < x$
shows $\exists t.\ 0 < t \wedge t \leq x \wedge sin\ x = (\sum m{<}n.\ sin\text{-}coeff\ m * x \hat{} m) + sin\ (t + 1/2 * real\ n * pi) / fact\ n * x \hat{} n$
unfolding *sin-coeff-lemma*
apply (*rule Maclaurin2*)
by (*auto intro*!: *derivative-eq-intros simp*: *assms ring-distribs sin-add cos-add*)

114.5 Maclaurin Expansion for Cosine Function

lemma *sumr-cos-zero-one* [*simp*]: $(\sum m{<}Suc\ n.\ cos\text{-}coeff\ m * 0 \hat{} m) = 1$
by (*induct n*) *auto*

lemma *cos-add-pi-half-add*: $cos\ (x + (pi/2 + y)) = -\ sin\ (x + y)$
by (*auto simp*: *cos-add sin-add*)

lemma *cos-coeff-lemma*: $cos\text{-}coeff\ m = cos\ (0 + 1/2 * real\ m * pi) / fact\ m$
by (*cases even m*) (*auto simp add*: *cos-coeff-def cos-zero-iff*)

lemma *Maclaurin-cos-expansion*:
  $\exists\, t$::*real*. $|t| \leq |x| \wedge$
    cos x = ($\sum$ m<n. *cos-coeff* $m * x \hat{} m$) + *cos*(t + *1/2* * *real* n * *pi*) / *fact* n * $x \hat{} n$
  unfolding *cos-coeff-lemma*
  apply (*rule Maclaurin-all-le*)
  by (*simp-all add*: *ring-distribs cos-add-pi-half-add*)

lemma *Maclaurin-cos-expansion2*:
  assumes $x > 0$ $n > 0$
  shows $\exists\, t.\ 0 < t \wedge t < x \wedge$
      cos x = ($\sum$ m<n. *cos-coeff* $m * x \hat{} m$) + *cos* (t + *1/2* * *real* n * *pi*) / *fact* $n * x \hat{} n$
  unfolding *cos-coeff-lemma*
  apply (*rule Maclaurin*)
  by (*simp-all add*: *assms ring-distribs cos-add-pi-half-add*)

lemma *Maclaurin-minus-cos-expansion*:
  assumes $n > 0$ $x < 0$
  shows $\exists\, t.\ x < t \wedge t < 0 \wedge$
        cos x = ($\sum$ m<n. *cos-coeff* $m * x \hat{} m$) + *cos* (t + *1/2* * *real* n * *pi*) / *fact* $n * x \hat{} n$
  unfolding *cos-coeff-lemma*
  apply (*rule Maclaurin-minus*)
  by (*simp-all add*: *assms ring-distribs cos-add-pi-half-add*)

lemma *sin-bound-lemma*: $x = y \Longrightarrow |u| \leq v \Longrightarrow |(x + u) - y| \leq v$
  for x y u v :: *real*
  by *auto*

lemma *Maclaurin-sin-bound*: |*sin* x − ($\sum$ m<n. *sin-coeff* $m * x \hat{} m$)| ≤ *inverse* (*fact* n) * |x| $\hat{}$ n
proof −
  have *est*: $x \leq 1 \Longrightarrow 0 \leq y \Longrightarrow x * y \leq 1 * y$ **for** x y :: *real*
    by (*rule mult-right-mono*) *simp-all*
  let *?diff* = λ(n::*nat*) (x::*real*).
    if n *mod* *4* = *0* *then* *sin* x
    else if n *mod* *4* = *1* *then* *cos* x
    else if n *mod* *4* = *2* *then* − *sin* x
    else − *cos* x
  have *diff-0*: *?diff* *0* = *sin* **by** *simp*
  have *DERIV* (*?diff* m) x :> *?diff* (*Suc* m) x **for** m **and** x
    using *mod-exhaust-less-4* [*of* m]
    by (*auto simp*: *mod-Suc intro*!: *derivative-eq-intros*)

then have *DERIV-diff*: $\forall\, m\ x.\ DERIV\ (?diff\ m)\ x :> ?diff\ (Suc\ m)\ x$
by *blast*
from *Maclaurin-all-le* [*OF diff-0 DERIV-diff*]
obtain *t* **where** *t1*: $|t| \leq |x|$
and *t2*: $sin\ x = (\sum m{<}n.\ ?diff\ m\ 0\ /\ (fact\ m) * x\ \hat{}\ m) + ?diff\ n\ t\ /\ (fact\ n) * x\ \hat{}\ n$
by *fast*
have *diff-m-0*: $?diff\ m\ 0 = (if\ even\ m\ then\ 0\ else\ (-\ 1)\ \hat{}\ ((m - Suc\ 0)\ div\ 2))$
for *m*
using *mod-exhaust-less-4* [*of m*]
by (*auto simp*: *minus-one-power-iff even-even-mod-4-iff* [*of m*] *dest*: *even-mod-4-div-2 odd-mod-4-div-2*)
show *?thesis*
apply (*subst t2*)
apply (*rule sin-bound-lemma*)
apply (*rule sum.cong*[*OF refl*])
apply (*simp add*: *diff-m-0 sin-coeff-def*)
using *est*
apply (*auto intro*: *mult-right-mono* [**where** *b=1*, *simplified*] *mult-right-mono*
simp: *ac-simps divide-inverse power-abs* [*symmetric*] *abs-mult*)
done
qed

115 Taylor series

We use MacLaurin and the translation of the expansion point *c* to *0* to prove Taylor's theorem.

lemma *Taylor-up*:
assumes *INIT*: $n > 0\ diff\ 0 = f$
and *DERIV*: $\forall\, m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow DERIV\ (diff\ m)\ t :> (diff\ (Suc\ m)\ t)$
and *INTERV*: $a \leq c\ c < b$
shows $\exists\, t{::}real.\ c < t \wedge t < b \wedge$
$f\ b = (\sum m{<}n.\ (diff\ m\ c\ /\ fact\ m) * (b - c)\ \hat{}\ m) + (diff\ n\ t\ /\ fact\ n) * (b - c)\ \hat{}\ n$
proof −
from *INTERV* **have** $0 < b - c$ **by** *arith*
moreover from *INIT* **have** $n > 0\ (\lambda m\ x.\ diff\ m\ (x + c))\ 0 = (\lambda x.\ f\ (x + c))$
by *auto*
moreover
have $\forall\, m\ t.\ m < n \wedge 0 \leq t \wedge t \leq b - c \longrightarrow DERIV\ (\lambda x.\ diff\ m\ (x + c))\ t :> diff\ (Suc\ m)\ (t + c)$
proof (*intro strip*)
fix *m t*
assume $m < n \wedge 0 \leq t \wedge t \leq b - c$
with *DERIV* **and** *INTERV* **have** $DERIV\ (diff\ m)\ (t + c) :> diff\ (Suc\ m)\ (t + c)$
by *auto*

moreover from *DERIV-ident* **and** *DERIV-const* **have** *DERIV* $(\lambda x.\ x + c)\ t$ *:> 1 + 0*
by (*rule DERIV-add*)
ultimately have *DERIV* $(\lambda x.\ diff\ m\ (x + c))\ t$ *:> diff* $(Suc\ m)\ (t + c) * (1 + 0)$
by (*rule DERIV-chain2*)
then show *DERIV* $(\lambda x.\ diff\ m\ (x + c))\ t$ *:> diff* $(Suc\ m)\ (t + c)$
by *simp*
qed
ultimately obtain x **where**
$0 < x \wedge x < b - c \wedge$
$f\ (b - c + c) =$
$(\sum m<n.\ diff\ m\ (0 + c)\ /\ fact\ m * (b - c)\ \hat{}\ m) + diff\ n\ (x + c)\ /\ fact\ n * (b - c)\ \hat{}\ n$
by (*rule Maclaurin* [*THEN exE*])
then show *?thesis*
by (*smt* (*verit*) *sum.cong*)
qed

lemma *Taylor-down*:
fixes a :: *real* **and** n :: *nat*
assumes *INIT*: $n > 0$ *diff 0* $= f$
and *DERIV*: $(\forall m\ t.\ m < n \wedge a \leq t \wedge t \leq b \longrightarrow DERIV\ (diff\ m)\ t$ *:> diff* $(Suc\ m)\ t)$
and *INTERV*: $a < c\ c \leq b$
shows $\exists t.\ a < t \wedge t < c \wedge$
$f\ a = (\sum m<n.\ (diff\ m\ c\ /\ fact\ m) * (a - c)\hat{}m) + (diff\ n\ t\ /\ fact\ n) * (a - c)\hat{}n$
proof −
from *INTERV* **have** $a-c < 0$ **by** *arith*
moreover from *INIT* **have** $n > 0\ (\lambda m\ x.\ diff\ m\ (x + c))\ 0 = (\lambda x.\ f\ (x + c))$
by *auto*
moreover
have $\forall m\ t.\ m < n \wedge a - c \leq t \wedge t \leq 0 \longrightarrow DERIV\ (\lambda x.\ diff\ m\ (x + c))\ t$ *:> diff* $(Suc\ m)\ (t + c)$
proof (*rule allI impI*)+
fix $m\ t$
assume $m < n \wedge a - c \leq t \wedge t \leq 0$
with *DERIV* **and** *INTERV* **have** *DERIV* $(diff\ m)\ (t + c)$ *:> diff* $(Suc\ m)\ (t + c)$
by *auto*
moreover from *DERIV-ident* **and** *DERIV-const* **have** *DERIV* $(\lambda x.\ x + c)\ t$ *:> 1 + 0*
by (*rule DERIV-add*)
ultimately show *DERIV* $(\lambda x.\ diff\ m\ (x + c))\ t$ *:> diff* $(Suc\ m)\ (t + c)$
using *DERIV-chain2 DERIV-shift* **by** *blast*
qed
ultimately obtain x **where**
$a - c < x \wedge x < 0 \wedge$

```
    f (a − c + c) =
      (∑ m<n. diff m (0 + c) / fact m * (a − c) ^ m) + diff n (x + c) / fact n
* (a − c) ^ n
    by (rule Maclaurin-minus [THEN exE])
  then have a < x + c ∧ x + c < c ∧
    f a = (∑ m<n. diff m c / fact m * (a − c) ^ m) + diff n (x + c) / fact n *
(a − c) ^ n
    by fastforce
  then show ?thesis by fastforce
qed

theorem Taylor:
  fixes a :: real and n :: nat
  assumes INIT: n > 0 diff 0 = f
    and DERIV: ∀ m t. m < n ∧ a ≤ t ∧ t ≤ b ⟶ DERIV (diff m) t :> diff
(Suc m) t
    and INTERV: a ≤ c  c ≤ b a ≤ x x ≤ b x ≠ c
  shows ∃ t.
    (if x < c then x < t ∧ t < c else c < t ∧ t < x) ∧
    f x = (∑ m<n. (diff m c / fact m) * (x − c)^m) + (diff n t / fact n) * (x −
c)^n
proof (cases x < c)
  case True
  note INIT
  moreover have ∀ m t. m < n ∧ x ≤ t ∧ t ≤ b ⟶ DERIV (diff m) t :> diff
(Suc m) t
    using DERIV and INTERV by fastforce
  ultimately show ?thesis
    using True INTERV Taylor-down by simp
next
  case False
  note INIT
  moreover have ∀ m t. m < n ∧ a ≤ t ∧ t ≤ x ⟶ DERIV (diff m) t :> diff
(Suc m) t
    using DERIV and INTERV by fastforce
  ultimately show ?thesis
    using Taylor-up INTERV False by simp
qed

end
```

116 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier. Thanks to Alexander Maletzky among others.

```
theory Binomial-Plus
imports Real
begin
```

116.1 More facts about binomial coefficients

These facts could have been proven before, but having real numbers makes the proofs a lot easier.

lemma *central-binomial-odd*:
odd n $\Longrightarrow$ *n choose* (*Suc* (*n div 2*)) = *n choose* (*n div 2*)
proof −
assume *odd n*
hence *Suc* (*n div 2*) ≤ *n* **by** *presburger*
hence *n choose* (*Suc* (*n div 2*)) = *n choose* (*n* − *Suc* (*n div 2*))
by (*rule binomial-symmetric*)
also from ‹*odd n*› **have** *n* − *Suc* (*n div 2*) = *n div 2* **by** *presburger*
finally show *?thesis* **.**
qed

lemma *binomial-less-binomial-Suc*:
assumes *k*: *k* < *n div 2*
shows *n choose k* < *n choose* (*Suc k*)
proof −
from *k* **have** *k'*: *k* ≤ *n Suc k* ≤ *n* **by** *simp-all*
from *k'* **have** *real* (*n choose k*) = *fact n* / (*fact k* ∗ *fact* (*n* − *k*))
by (*simp add*: *binomial-fact*)
also from *k'* **have** *n* − *k* = *Suc* (*n* − *Suc k*) **by** *simp*
also from *k'* **have** *fact* ... = (*real n* − *real k*) ∗ *fact* (*n* − *Suc k*)
by (*subst fact-Suc*) (*simp-all add*: *of-nat-diff*)
also from *k* **have** *fact k* = *fact* (*Suc k*) / (*real k* + *1*) **by** (*simp add*: *field-simps*)
also have *fact n* / (*fact* (*Suc k*) / (*real k* + *1*) ∗ ((*real n* − *real k*) ∗ *fact* (*n* − *Suc k*))) =
(*n choose* (*Suc k*)) ∗ ((*real k* + *1*) / (*real n* − *real k*))
using *k* **by** (*simp add*: *field-split-simps binomial-fact*)
also from *assms* **have** (*real k* + *1*) / (*real n* − *real k*) < *1* **by** *simp*
finally show *?thesis* **using** *k* **by** (*simp add*: *mult-less-cancel-left*)
qed

lemma *binomial-strict-mono*:
assumes *k* < *k'* *2*∗*k'* ≤ *n*
shows *n choose k* < *n choose k'*
proof −
from *assms* **have** *k* ≤ *k'* − *1* **by** *simp*
thus *?thesis*
proof (*induction rule*: *inc-induct*)
case *base*
with *assms binomial-less-binomial-Suc*[*of k'* − *1 n*]
show *?case* **by** *simp*
next
case (*step k*)
from *step.prems step.hyps assms* **have** *n choose k* < *n choose* (*Suc k*)
by (*intro binomial-less-binomial-Suc*) *simp-all*
also have ... < *n choose k'* **by** (*rule step.IH*)

```
    finally show ?case .
  qed
qed

lemma binomial-mono:
  assumes k ≤ k' 2∗k' ≤ n
  shows   n choose k ≤ n choose k'
  using assms binomial-strict-mono[of k k' n] by (cases k = k') simp-all

lemma binomial-strict-antimono:
  assumes k < k' 2 ∗ k ≥ n k' ≤ n
  shows   n choose k > n choose k'
proof −
  from assms have n choose (n − k) > n choose (n − k')
    by (intro binomial-strict-mono) (simp-all add: algebra-simps)
  with assms show ?thesis by (simp add: binomial-symmetric [symmetric])
qed

lemma binomial-antimono:
  assumes k ≤ k' k ≥ n div 2 k' ≤ n
  shows   n choose k ≥ n choose k'
proof (cases k = k')
  case False
  note not-eq = False
  show ?thesis
  proof (cases k = n div 2 ∧ odd n)
    case False
    with assms(2) have 2∗k ≥ n by presburger
    with not-eq assms binomial-strict-antimono[of k k' n]
      show ?thesis by simp
  next
    case True
    have n choose k' ≤ n choose (Suc (n div 2))
    proof (cases k' = Suc (n div 2))
      case False
      with assms True not-eq have Suc (n div 2) < k' by simp
      with assms binomial-strict-antimono[of Suc (n div 2) k' n] True
        show ?thesis by auto
    qed simp-all
    also from True have ... = n choose k by (simp add: central-binomial-odd)
    finally show ?thesis .
  qed
qed simp-all

lemma binomial-maximum: n choose k ≤ n choose (n div 2)
proof −
  have k ≤ n div 2 ⟷ 2∗k ≤ n by linarith
  consider 2∗k ≤ n | 2∗k ≥ n k ≤ n | k > n by linarith
  thus ?thesis
```

proof *cases*
case *1*
thus *?thesis* **by** (*intro binomial-mono*) *linarith+*
next
case *2*
thus *?thesis* **by** (*intro binomial-antimono*) *simp-all*
qed (*simp-all add*: *binomial-eq-0*)
qed

lemma *binomial-maximum'*: (*2*∗*n*) *choose k* ≤ (*2*∗*n*) *choose n*
using *binomial-maximum*[*of 2*∗*n*] **by** *simp*

lemma *central-binomial-lower-bound*:
assumes *n* > *0*
shows *4*^*n* / (*2*∗*real n*) ≤ *real* ((*2*∗*n*) *choose n*)
proof −
from *binomial*[*of 1 1 2*∗*n*]
have *4* ^ *n* = (∑ *k*≤*2*∗*n*. (*2*∗*n*) *choose k*)
by (*simp add*: *power-mult power2-eq-square One-nat-def* [*symmetric*] *del*: *One-nat-def*)
also have {..*2*∗*n*} = {*0*<..<*2*∗*n*} ∪ {*0*,*2*∗*n*} **by** *auto*
also have (∑ *k*∈.... (*2*∗*n*) *choose k*) =
(∑ *k*∈{*0*<..<*2*∗*n*}. (*2*∗*n*) *choose k*) + (∑ *k*∈{*0*,*2*∗*n*}. (*2*∗*n*) *choose k*)
by (*subst sum.union-disjoint*) *auto*
also have (∑ *k*∈{*0*,*2*∗*n*}. (*2*∗*n*) *choose k*) ≤ (∑ *k*≤*1*. (*n choose k*)^2)
by (*cases n*) *simp-all*
also from *assms* **have** ... ≤ (∑ *k*≤*n*. (*n choose k*)^2)
by (*intro sum-mono2*) *auto*
also have ... = (*2*∗*n*) *choose n* **by** (*rule choose-square-sum*)
also have (∑ *k*∈{*0*<..<*2*∗*n*}. (*2*∗*n*) *choose k*) ≤ (∑ *k*∈{*0*<..<*2*∗*n*}. (*2*∗*n*) *choose n*)
by (*intro sum-mono binomial-maximum'*)
also have ... = *card* {*0*<..<*2*∗*n*} ∗ ((*2*∗*n*) *choose n*) **by** *simp*
also have *card* {*0*<..<*2*∗*n*} ≤ *2*∗*n* − *1* **by** (*cases n*) *simp-all*
also have (*2* ∗ *n* − *1*) ∗ (*2* ∗ *n choose n*) + (*2* ∗ *n choose n*) = ((*2*∗*n*) *choose n*) ∗ (*2*∗*n*)
using *assms* **by** (*simp add*: *algebra-simps*)
finally have *4* ^ *n* ≤ (*2* ∗ *n choose n*) ∗ (*2* ∗ *n*) **by** *simp-all*
hence *real* (*4* ^ *n*) ≤ *real* ((*2* ∗ *n choose n*) ∗ (*2* ∗ *n*))
by (*subst of-nat-le-iff*)
with *assms* **show** *?thesis* **by** (*simp add*: *field-simps*)
qed

lemma *upper-le-binomial*:
assumes *0* < *k* **and** *k* < *n*
shows *n* ≤ *n choose k*
proof −
from *assms* **have** *1* ≤ *n* **by** *simp*
define *k'* **where** *k'* = (*if n div 2* ≤ *k then k else n* − *k*)
from *assms* **have** *1*: *k'* ≤ *n* − *1* **and** *2*: *n div 2* ≤ *k'* **by** (*auto simp*: *k'-def*)

from *assms*(*2*) **have** $k \leq n$ **by** *simp*
have *n choose k* = *n choose k'* **by** (*simp add*: *k'-def binomial-symmetric*[*OF* ‹*k* ≤ *n*›])
have *n* = *n choose 1* **by** (*simp only*: *choose-one*)
also from ‹*1* ≤ *n*› **have** ... = *n choose* (*n* − *1*) **by** (*rule binomial-symmetric*)
also from *1 2* **have** ... ≤ *n choose k'* **by** (*rule binomial-antimono*) *simp*
also have ... = *n choose k* **by** (*simp add*: *k'-def binomial-symmetric*[*OF* ‹*k* ≤ *n*›])
finally show *?thesis* **.**
qed

116.2 Results about binomials and integers, thanks to Alexander Maletzky

Restore original sort constraints: semidom rather than field of char 0

setup ‹*Sign.add-const-constraint* (@{*const-name gbinomial*}, *SOME* @{*typ* 'a::{*semidom-divide*,*semiring-char*
⇒ *nat* ⇒ 'a})›

lemma *gbinomial-eq-0-int*:
assumes *n* < *k*
shows (*int n*) *gchoose k* = *0*
by (*simp add*: *assms gbinomial-prod-rev prod-zero*)

corollary *gbinomial-eq-0*: *0* ≤ *a* ⟹ *a* < *int k* ⟹ *a gchoose k* = *0*
by (*metis nat-eq-iff2 nat-less-iff gbinomial-eq-0-int*)

lemma *gbinomial-mono*:
fixes *k*::*nat* **and** *a*::*real*
assumes *of-nat k* ≤ *a a* ≤ *b* **shows** *a gchoose k* ≤ *b gchoose k*
using *assms*
by (*force simp*: *gbinomial-prod-rev intro*!: *divide-right-mono prod-mono*)

lemma *int-binomial*: *int* (*n choose k*) = (*int n*) *gchoose k*
proof (*cases k* ≤ *n*)
case *True*
from *refl* **have** *eq*: (∏ *i* = *0*..<*k*. *int* (*n* − *i*)) = (∏ *i* = *0*..<*k*. *int n* − *int i*)
proof (*rule prod.cong*)
fix *i*
assume *i* ∈ {*0*..<*k*}
with *True* **show** *int* (*n* − *i*) = *int n* − *int i* **by** *simp*
qed
show *?thesis*
by (*simp add*: *gbinomial-binomial*[*symmetric*] *gbinomial-prod-rev zdiv-int eq*)
next
case *False*
thus *?thesis* **by** (*simp add*: *gbinomial-eq-0-int*)
qed

lemma *falling-fact-pochhammer*: *prod* (λ*i*. *a* − *int i*) {*0*..<*k*} = (− *1*) ^ *k* ∗ *pochhammer* (− *a*) *k*
proof −
have *eq*: *z* ^ *Suc n* ∗ *prod f* {*0*..*n*} = *prod* (λ*x*. *z* ∗ *f x*) {*0*..*n*} **for** *z*::*int* **and** *n f*
by (*induct n*) (*simp-all add*: *ac-simps*)
show *?thesis*
proof (*cases k*)
case *0*
thus *?thesis* **by** (*simp add*: *pochhammer-minus*)
next
case (*Suc n*)
thus *?thesis*
by (*simp only*: *pochhammer-prod atLeastLessThanSuc-atLeastAtMost*
prod.atLeast-Suc-atMost-Suc-shift eq flip: *power-mult-distrib*) (*simp add*: *of-nat-diff*)
qed
qed

lemma *falling-fact-pochhammer′*: *prod* (λ*i*. *a* − *int i*) {*0*..<*k*} = *pochhammer* (*a* − *int k* + *1*) *k*
by (*simp add*: *falling-fact-pochhammer pochhammer-minus′*)

lemma *gbinomial-int-pochhammer*: (*a*::*int*) *gchoose k* = (− *1*) ^ *k* ∗ *pochhammer* (− *a*) *k div fact k*
by (*simp only*: *gbinomial-prod-rev falling-fact-pochhammer*)

lemma *gbinomial-int-pochhammer′*: *a gchoose k* = *pochhammer* (*a* − *int k* + *1*) *k div fact k*
by (*simp only*: *gbinomial-prod-rev falling-fact-pochhammer′*)

lemma *fact-dvd-pochhammer*: *fact k dvd pochhammer* (*a*::*int*) *k*
proof −
have *dvd*: *y* ≠ *0* ⟹ ((*of-int* (*x div y*))::′*a*::*field-char-0*) = *of-int x* / *of-int y* ⟹ *y dvd x*
for *x y* :: *int*
by (*metis dvd-triv-right nonzero-eq-divide-eq of-int-0-eq-iff of-int-eq-iff of-int-mult*)
show *?thesis*
proof (*cases 0* < *a*)
case *True*
moreover define *n* **where** *n* = *nat* (*a* − *1*) + *k*
ultimately have *a*: *a* = *int n* − *int k* + *1* **by** *simp*
from *fact-nonzero* **show** *?thesis* **unfolding** *a*
proof (*rule dvd*)
have *of-int* (*pochhammer* (*int n* − *int k* + *1*) *k div fact k*) = (*of-int* (*int n gchoose k*)::*rat*)
by (*simp only*: *gbinomial-int-pochhammer′*)
also have ... = *of-nat* (*n choose k*)
by (*metis int-binomial of-int-of-nat-eq*)
also have ... = (*of-nat n*) *gchoose k* **by** (*fact binomial-gbinomial*)

```
    also have ... = pochhammer (of-nat n − of-nat k + 1) k / fact k
      by (fact gbinomial-pochhammer′)
    also have ... = pochhammer (of-int (int n − int k + 1)) k / fact k by simp
    also have ... = (of-int (pochhammer (int n − int k + 1) k)) / (of-int (fact
k))
      by (simp only: of-int-fact pochhammer-of-int)
    finally show of-int (pochhammer (int n − int k + 1) k div fact k) =
                 of-int (pochhammer (int n − int k + 1) k) / rat-of-int (fact k) .
  qed
 next
  case False
  moreover define n where n = nat (− a)
  ultimately have a: a = − int n by simp
  from fact-nonzero have fact k dvd (−1)^k * pochhammer (− int n) k
  proof (rule dvd)
    have of-int ((−1)^k * pochhammer (− int n) k div fact k) = (of-int (int n
gchoose k)::rat)
      by (metis falling-fact-pochhammer gbinomial-prod-rev)
    also have ... = of-int (int (n choose k)) by (simp only: int-binomial)
    also have ... = of-nat (n choose k) by simp
    also have ... = (of-nat n) gchoose k by (fact binomial-gbinomial)
    also have ... = (−1)^k * pochhammer (− of-nat n) k / fact k
      by (fact gbinomial-pochhammer)
    also have ... = (−1)^k * pochhammer (of-int (− int n)) k / fact k by simp
    also have ... = (−1)^k * (of-int (pochhammer (− int n) k)) / (of-int (fact
k))
      by (simp only: of-int-fact pochhammer-of-int)
    also have ... = (of-int ((−1)^k * pochhammer (− int n) k)) / (of-int (fact
k)) by simp
    finally show of-int ((− 1) ^ k * pochhammer (− int n) k div fact k) =
                 of-int ((− 1) ^ k * pochhammer (− int n) k) / rat-of-int (fact k) .
  qed
  thus ?thesis unfolding a by (metis dvdI dvd-mult-unit-iff′ minus-one-mult-self)
 qed
qed

lemma gbinomial-int-negated-upper: (a gchoose k) = (−1) ^ k * ((int k − a − 1)
gchoose k)
 by (simp add: gbinomial-int-pochhammer pochhammer-minus algebra-simps fact-dvd-pochhammer
div-mult-swap)

lemma gbinomial-int-mult-fact: fact k * (a gchoose k) = (∏ i = 0..<k. a − int i)
 by (simp only: gbinomial-int-pochhammer′ fact-dvd-pochhammer dvd-mult-div-cancel
falling-fact-pochhammer′)

corollary gbinomial-int-mult-fact′: (a gchoose k) * fact k = (∏ i = 0..<k. a − int
i)
 using gbinomial-int-mult-fact[of k a] by (simp add: ac-simps)
```

lemma *gbinomial-int-binomial*:
*a gchoose k = (if 0 ≤ a then int ((nat a) choose k) else (−1::int)^k * int ((k + (nat (− a)) − 1) choose k))*
by (*auto simp*: *int-binomial gbinomial-int-negated-upper*[*of a*] *int-ops*(*6*))

corollary *gbinomial-nneg*: *0 ≤ a ⟹ a gchoose k = int ((nat a) choose k)*
by (*simp add*: *gbinomial-int-binomial*)

corollary *gbinomial-neg*: *a < 0 ⟹ a gchoose k = (−1::int)^k * int ((k + (nat (− a)) − 1) choose k)*
by (*simp add*: *gbinomial-int-binomial*)

lemma *of-int-gbinomial*: *of-int (a gchoose k) = (of-int a :: 'a::field-char-0) gchoose k*
proof −
have *of-int-div*: *y dvd x ⟹ of-int (x div y) = of-int x / (of-int y :: 'a)* **for** *x y :: int* **by** *auto*
show *?thesis*
by (*simp add*: *gbinomial-int-pochhammer' gbinomial-pochhammer' of-int-div fact-dvd-pochhammer*
pochhammer-of-int[*symmetric*])
qed

lemma *uminus-one-gbinomial* [*simp*]: *(− 1::int) gchoose k = (− 1) ^ k*
by (*simp add*: *gbinomial-int-binomial*)

lemma *gbinomial-int-Suc-Suc*: *(x + 1::int) gchoose (Suc k) = (x gchoose k) + (x gchoose (Suc k))*
proof (*rule linorder-cases*)
assume *1*: *x + 1 < 0*
hence *2*: *x < 0* **by** *simp*
then obtain *n* **where** *3*: *nat (− x) = Suc n* **using** *not0-implies-Suc* **by** *fastforce*
hence *4*: *nat (− x − 1) = n* **by** *simp*
show *?thesis*
proof (*cases k*)
case *0*
show *?thesis* **by** (*simp add*: ‹*k = 0*›)
next
case (*Suc k'*)
from *1 2 3 4* **show** *?thesis* **by** (*simp add*: ‹*k = Suc k'*› *gbinomial-int-binomial int-distrib*(*2*))
qed
next
assume *x + 1 = 0*
hence *x = − 1* **by** *simp*
thus *?thesis* **by** *simp*
next
assume *0 < x + 1*
hence *0 ≤ x + 1* **and** *0 ≤ x* **and** *nat (x + 1) = Suc (nat x)* **by** *simp-all*

thus *?thesis* **by** (*simp add*: *gbinomial-int-binomial*)
qed

corollary *plus-Suc-gbinomial*:
(*x* + (*1* + *int k*)) *gchoose* (*Suc k*) = ((*x* + *int k*) *gchoose k*) + ((*x* + *int k*) *gchoose* (*Suc k*))
(**is** *?l* = *?r*)
proof −
have *?l* = (*x* + *int k* + *1*) *gchoose* (*Suc k*) **by** (*simp only*: *ac-simps*)
also have ... = *?r* **by** (*fact gbinomial-int-Suc-Suc*)
finally show *?thesis* **.**
qed

lemma *gbinomial-int-n-n* [*simp*]: (*int n*) *gchoose n* = *1*
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
have *int* (*Suc n*) *gchoose Suc n* = (*int n* + *1*) *gchoose Suc n* **by** (*simp add*: *add.commute*)
also have ... = (*int n gchoose n*) + (*int n gchoose* (*Suc n*)) **by** (*fact gbinomial-int-Suc-Suc*)
finally show *?case* **by** (*simp add*: *Suc gbinomial-eq-0*)
qed

lemma *gbinomial-int-Suc-n* [*simp*]: (*1* + *int n*) *gchoose n* = *1* + *int n*
proof (*induct n*)
case *0*
show *?case* **by** *simp*
next
case (*Suc n*)
have *1* + *int* (*Suc n*) *gchoose Suc n* = (*1* + *int n*) + *1 gchoose Suc n* **by** *simp*
also have ... = (*1* + *int n gchoose n*) + (*1* + *int n gchoose* (*Suc n*)) **by** (*fact gbinomial-int-Suc-Suc*)
also have ... = *1* + *int n* + (*int* (*Suc n*) *gchoose* (*Suc n*)) **by** (*simp add*: *Suc*)
also have ... = *1* + *int* (*Suc n*) **by** (*simp only*: *gbinomial-int-n-n*)
finally show *?case* **.**
qed

lemma *zbinomial-eq-0-iff* [*simp*]: *a gchoose k* = *0* ⟷ (*0* ≤ *a* ∧ *a* < *int k*)
proof
assume *a*: *a gchoose k* = *0*
have *1*: *b* < *int k* **if** *b gchoose k* = *0* **for** *b*
proof (*rule ccontr*)
assume ¬ *b* < *int k*
hence *0* ≤ *b* **and** *k* ≤ *nat b* **by** *simp-all*
from *this*(*1*) **have** *int* ((*nat b*) *choose k*) = *b gchoose k* **by** (*simp add*: *gbinomial-int-binomial*)

```
    also have ... = 0 by (fact that)
    finally show False using ‹k ≤ nat b› by simp
  qed
  show 0 ≤ a ∧ a < int k
  proof
    show 0 ≤ a
    proof (rule ccontr)
      assume ¬ 0 ≤ a
      hence (−1) ^ k ∗ ((int k − a − 1) gchoose k) = a gchoose k
        by (simp add: gbinomial-int-negated-upper[of a])
      also have ... = 0 by (fact a)
      finally have (int k − a − 1) gchoose k = 0 by simp
      hence int k − a − 1 < int k by (rule 1)
      with ‹¬ 0 ≤ a› show False by simp
    qed
  next
    from a show a < int k by (rule 1)
  qed
qed (auto intro: gbinomial-eq-0)
```

116.3 Sums

```
lemma gchoose-rising-sum-nat: (∑j≤n. int j + int k gchoose k) = (int n + int k
+ 1) gchoose (Suc k)
proof −
  have (∑j≤n. int j + int k gchoose k) = int (∑j≤n. k + j choose k)
    by (simp add: int-binomial add.commute)
  also have (∑j≤n. k + j choose k) = (k + n + 1) choose (k + 1) by (fact
choose-rising-sum(1))
  also have int ... = (int n + int k + 1) gchoose (Suc k)
    by (simp add: int-binomial ac-simps del: binomial-Suc-Suc)
  finally show ?thesis .
qed

lemma gchoose-rising-sum:
  assumes 0 ≤ n  — Necessary condition.
  shows (∑j=0..n. j + int k gchoose k) = (n + int k + 1) gchoose (Suc k)
proof −
  from - refl have (∑j=0..n. j + int k gchoose k) = (∑j∈int ‘ {0..nat n}. j +
int k gchoose k)
  proof (rule sum.cong)
   from assms show {0..n} = int ‘ {0..nat n} by (simp add: image-int-atLeastAtMost)
  qed
  also have ... = (∑j≤nat n. int j + int k gchoose k) by (simp add: sum.reindex
atMost-atLeast0)
 also have ... = (int (nat n) + int k + 1) gchoose (Suc k) by (fact gchoose-rising-sum-nat)
  also from assms have ... = (n + int k + 1) gchoose (Suc k) by (simp add:
add.assoc add.commute)
  finally show ?thesis .
```

qed

end
theory *Alethe-Arith-Real-Rewrites*
imports *HOL.Real*
begin

named-theorems *rewrite-arith-geq-norm1-real* ‹›

lemma [*rewrite-arith-geq-norm1-real*]:
fixes *t*::*real* **and** *s*::*real*
shows *NO-MATCH cvc-a* (*undefined t s*)
⟹ ((*t* ≥ *s*) = ((*t* − *s*) ≥ *0/1*))
by *simp*

named-theorems *rewrite-arith-eq-elim-real* ‹›

lemma [*rewrite-arith-eq-elim-real*]:
fixes *t*::*'a*::*linordered-idom* **and** *s*::*'a*::*linordered-idom*
shows *NO-MATCH cvc-a* (*undefined t s*)
⟹ ((*t* = *s*) = (*t* ≥ *s* ∧ *t* ≤ *s*))
by *auto*

named-theorems *rewrite-arith-to-int-to-real* ‹›

lemma [*rewrite-arith-to-int-to-real*]:
fixes *x*::*int*
shows *NO-MATCH cvc-a* (*undefined x*) ⟹ *floor* (*of-int x* ::*real*) = *x*
by *simp*

named-theorems *rewrite-arith-int-eq-conflict* ‹›

lemma [*rewrite-arith-int-eq-conflict*]:
fixes *t*::*int* **and** *c*::*real*
shows *NO-MATCH cvc-a* (*undefined t c*)
⟹ ¬((*of-int* (*floor c*)::*real*) = *c*) ⟹ (((*of-int t* ::*real*) = *c*) = *False*)
by *auto*

named-theorems *rewrite-arith-int-geq-tighten* ‹›

lemma [*rewrite-arith-int-geq-tighten*]:
fixes *t*::*int* **and** *c*::*real* **and** *cc*::*int*
shows *NO-MATCH cvc-a* (*undefined t c cc*)
⟹ (¬((*of-int* (*floor c*)) = *c*) ∧ (*cc* = (*floor c*) + *1*)) ⟹ (((*of-int t* ::*real*) ≥ *c*)

$= (t \geq cc))$
by *linarith*

end
theory *SMT-CVC-Real*
imports *Alethe-Arith-Real-Rewrites HOL.Real*
begin

cvc5-rare *Alethe-Arith-Real-Rewrites.rewrite-arith-geq-norm1-real*
cvc5-rare *Alethe-Arith-Real-Rewrites.rewrite-arith-eq-elim-real*
cvc5-rare *Alethe-Arith-Real-Rewrites.rewrite-arith-to-int-to-real*
cvc5-rare *Alethe-Arith-Real-Rewrites.rewrite-arith-int-eq-conflict*
cvc5-rare *Alethe-Arith-Real-Rewrites.rewrite-arith-int-geq-tighten*

lemma *alethe-push-assumption-in-goal*[*no-atp*]:
$A \Longrightarrow (B \equiv C) \Longrightarrow (B \equiv (A \wedge C))$
by *simp*

lemmas *cvc-evaluate*[*no-atp*] = *of-rat-add of-rat-minus of-rat-diff of-rat-mult of-rat-divide of-rat-eq-iff*
of-rat-divide nonzero-of-rat-divide of-rat-neg-numeral-eq alethe-push-assumption-in-goal

lemmas [*alethe-poly-norm*] =
ab-group-add-class.minus-add-distrib
add-uminus-conv-diff add.inverse-inverse
uminus-add-conv-diff right-diff-distrib-numeral
mult-minus-left distrib-left-numeral mult-num-simps add-num-simps
numeral-mult[*symmetric*] *minus-diff-eq of-int-diff of-int-numeral of-int-add*
of-int-mult of-int-numeral mult-minus-left of-int-1 add.inverse-neutral
diff-0 minus-diff-eq add.left-neutral mult.left-neutral of-int-0 divide-minus-left
times-divide-eq-left add-divide-distrib divide-eq-eq-numeral1 eq-numeral-simps
if-True if-False distrib-right-numeral left-diff-distrib-numeral times-divide-eq-left
of-int-neg-numeral ring-1-class.of-int-diff ring-1-class.of-int-numeral ring-1-class.of-int-mult
mult-numeral-left-semiring-numeral mult-num-simps of-int-minus mult-minus-right
times-divide-eq-right
of-int-of-nat-eq diff-divide-distrib add.right-neutral diff-cancel mult-minus-right
mult-zero-left mult-zero-right div-0 div-by-1 add-uminus-conv-diff numeral-times-numeral
mult-numeral-left-semiring-numeral mult-num-simps mult-numeral-left uminus-add-conv-diff
div-by-1 divide-numeral-1 divide-divide-eq-right neg-equal-iff-equal numeral-One
divide-divide-eq-left
divide-self-if if-True if-False one-neq-zero more-arith-simps
division-ring-class.times-divide-eq-right divide-cancel-left

117 Lemmas to reconstruct `alethe_poly_simp_rel`

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2 cx cy*::*real*
shows $cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (x1 - x2)) = (cy * (y1 - y2))) \longrightarrow ((x1 = x2) = (y1 = y2))$

by *force*

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2* ::*int* **and** *cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (real\text{-}of\text{-}int\ (x1 - x2))) = (cy * ((real\text{-}of\text{-}int\ y1) - (real\text{-}of\text{-}int\ y2)))) \longrightarrow$
$((x1 = x2) = ((real\text{-}of\text{-}int\ y1) = (real\text{-}of\text{-}int\ y2)))$
by *auto*

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2 cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (x1 - x2)) = (cy * (y1 - y2))) \longrightarrow ((x1 < x2) = (y1 < y2))$
by (*metis less-iff-diff-less-0 mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2* ::*int* **and** *cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (real\text{-}of\text{-}int\ (x1 - x2))) = (cy * ((real\text{-}of\text{-}int\ y1) - (real\text{-}of\text{-}int\ y2)))) \longrightarrow$
$((x1 < x2) = ((real\text{-}of\text{-}int\ y1) < (real\text{-}of\text{-}int\ y2)))$
by (*metis less-iff-diff-less-0 mult-less-0-iff not-less-iff-gr-or-eq of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y2* ::*int* **and** *cx cy y1*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (real\text{-}of\text{-}int\ (x1 - x2))) = (cy * (y1 - (real\text{-}of\text{-}int\ y2)))) \longrightarrow ((x1 < x2)$
$= (y1 < (real\text{-}of\text{-}int\ y2)))$
by (*metis less-iff-diff-less-0 mult-less-0-iff not-less-iff-gr-or-eq of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1* ::*int* **and** *cx cy y2*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (real\text{-}of\text{-}int\ (x1 - x2))) = (cy * ((real\text{-}of\text{-}int\ y1) - y2))) \longrightarrow ((x1 < x2)$
$= ((real\text{-}of\text{-}int\ y1) < y2))$
by (*metis less-iff-diff-less-0 mult-less-0-iff not-less-iff-gr-or-eq of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2 cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (x1 - x2)) = (cy * (y1 - y2))) \longrightarrow ((x1 \leq x2) = (y1 \leq y2))$
by (*metis diff-gt-0-iff-gt linorder-not-le mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:

fixes *x1 x2 y1 y2* ::*int* **and** *cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * real\text{-}of\text{-}int\ (x1 - x2)) = (cy * ((real\text{-}of\text{-}int\ y1) - (real\text{-}of\text{-}int\ y2)))) \longrightarrow$
$((x1 \leq x2) = (real\text{-}of\text{-}int\ y1 \leq real\text{-}of\text{-}int\ y2))$
apply (*cases x1* $\leq$ *x2*)
apply (*cases* $(0 < cx)$)
apply *simp-all*
apply (*metis le-iff-diff-le-0 mult-le-cancel-left mult-zero-right not-less-iff-gr-or-eq of-int-le-iff*)
apply (*metis diff-ge-0-iff-ge eq-iff-diff-eq-0 linorder-not-le mult-zero-right no-zero-divisors of-int-diff of-int-less-0-iff order-le-less zero-less-mult-iff*)
by (*metis* (*no-types*, *opaque-lifting*) *cvc-arith-rewrite-defs*(*5*) *diff-gt-0-iff-gt mult-less-0-iff not-less-iff-gr-or-eq of-int-less-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y2* ::*int* **and** *cx cy y1*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * (real\text{-}of\text{-}int\ (x2 - x1))) = (cy * ((real\text{-}of\text{-}int\ y2) - y1))) \longrightarrow ((x1 \leq x2)$
$= (y1 \leq (real\text{-}of\text{-}int\ y2)))$
apply (*cases x1* $\leq$ *x2*)
apply (*case-tac* [!] $(0 < cx)$)
apply *simp-all*
apply (*metis diff-ge-0-iff-ge mult.commute mult-left-le-imp-le mult-zero-left of-int-0-le-iff of-int-diff split-mult-pos-le zero-le-square*)
apply (*metis cvc-arith-rewrite-defs*(*5*) *diff-ge-0-iff-ge mult-le-0-iff of-int-0-le-iff of-int-diff order-le-less*)
apply (*metis* (*no-types*, *lifting*) *diff-ge-0-iff-ge linorder-not-less mult-eq-0-iff mult-le-0-iff nle-le of-int-le-iff*)
by (*metis diff-ge-0-iff-ge leI mult-le-0-iff of-int-le-iff order-antisym-conv*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1* ::*int* **and** *cx cy y2*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * real\text{-}of\text{-}int\ (x1 - x2)) = (cy * ((real\text{-}of\text{-}int\ y1) - y2))) \longrightarrow ((x1 \leq x2) =$
$(real\text{-}of\text{-}int\ y1 \leq y2))$
apply (*cases x1* $\leq$ *x2*)
apply (*cases* $(0 < cx)$)
apply *simp-all*
apply (*metis le-iff-diff-le-0 mult-le-cancel-left mult-zero-right not-less-iff-gr-or-eq of-int-le-iff*)
apply (*metis diff-ge-0-iff-ge eq-iff-diff-eq-0 linorder-not-le mult-zero-right no-zero-divisors of-int-diff of-int-less-0-iff order-le-less zero-less-mult-iff*)
by (*metis* (*no-types*, *opaque-lifting*) *cvc-arith-rewrite-defs*(*5*) *diff-gt-0-iff-gt mult-less-0-iff not-less-iff-gr-or-eq of-int-less-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:

fixes *x1 x2 y1 y2 cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow ((cx * (x1 - x2)) = (cy * (y1 - y2))) \longrightarrow ((x1 > x2) = (y1 > y2))$
by (*metis le-iff-diff-le-0 linorder-not-le mult-less-0-iff nless-le zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2* ::*int* **and** *cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow ((cx * real\text{-}of\text{-}int\ (x1 - x2)) = (cy * (real\text{-}of\text{-}int\ y1 - real\text{-}of\text{-}int\ y2))) \longrightarrow ((x1 > x2) = ((real\text{-}of\text{-}int\ y1) > (real\text{-}of\text{-}int\ y2)))$
apply (*cases x1 > x2*)
apply (*cases* $(0 < cx)$)
apply (*metis diff-gt-0-iff-gt of-int-diff of-int-less-iff order.asym zero-less-mult-iff*)
apply (*metis diff-gt-0-iff-gt linorder-neqE-linordered-idom mult-less-0-iff of-int-0-less-iff*)
by (*metis diff-gt-0-iff-gt mult-less-0-iff not-less-iff-gr-or-eq of-int-0 of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1* ::*int* **and** *cx cy y2*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow ((cx * real\text{-}of\text{-}int\ (x1 - x2)) = (cy * (real\text{-}of\text{-}int\ y1 - y2))) \longrightarrow ((x1 > x2) = ((real\text{-}of\text{-}int\ y1) > y2))$
apply (*cases x1 > x2*)
apply (*cases* $(0 < cx)$)
apply (*metis of-int-0-less-iff diff-gt-0-iff-gt zero-less-mult-pos mult-pos-pos*)
apply (*metis diff-gt-0-iff-gt mult-eq-0-iff not-less-iff-gr-or-eq of-int-0-less-iff zero-less-mult-iff*)
by (*metis diff-gt-0-iff-gt mult-less-0-iff not-less-iff-gr-or-eq of-int-0 of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y2* ::*int* **and** *cx cy y1*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow ((cx * real\text{-}of\text{-}int\ (x1 - x2)) = (cy * (y1 - real\text{-}of\text{-}int\ y2))) \longrightarrow ((x1 > x2) = (y1 > (real\text{-}of\text{-}int\ y2)))$
apply (*cases x1 > x2*)
apply (*cases* $(0 < cx)$)
apply (*metis diff-gt-0-iff-gt of-int-diff of-int-less-iff order.asym zero-less-mult-iff*)
apply (*metis diff-gt-0-iff-gt linorder-neqE-linordered-idom mult-less-0-iff of-int-0-less-iff*)
by (*metis diff-gt-0-iff-gt mult-less-0-iff not-less-iff-gr-or-eq of-int-0 of-int-less-0-iff zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2 cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow ((cx * (x1 - x2)) = (cy * (y1 - y2))) \longrightarrow ((x1 \geq x2) = (y1 \geq y2))$

by (*metis diff-ge-0-iff-ge linorder-not-le mult-less-0-iff not-less-iff-gr-or-eq zero-less-mult-iff*)

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1 y2* ::*int* **and** *cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * \text{real-of-int } (x1 - x2)) = (cy * (\text{real-of-int } y1 - \text{real-of-int } y2))) \longrightarrow ((x1 \geq x2) = ((\text{real-of-int } y1) \geq (\text{real-of-int } y2)))$
using *SMT.alethe-poly-simp-rel*(*14*) **by** *blast*

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y1* ::*int* **and** *cx cy y2*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * \text{real-of-int } (x1 - x2)) = (cy * (\text{real-of-int } y1 - y2))) \longrightarrow ((x1 \geq x2) = ((\text{real-of-int } y1) \geq y2))$
using *SMT.alethe-poly-simp-rel*(*14*) **by** *blast*

lemma [*alethe-poly-simp-rel*]:
fixes *x1 x2 y2* ::*int* **and** *y1 cx cy*::*real*
shows $((cx > 0) = (cy > 0)) \Longrightarrow cx \neq 0 \Longrightarrow cy \neq 0 \Longrightarrow$
$((cx * \text{real-of-int } (x1 - x2)) = (cy * (y1 - \text{real-of-int } y2))) \longrightarrow ((x1 \geq x2) = (y1 \geq (\text{real-of-int } y2)))$
apply (*cases x1* $\geq$ *x2*)
apply (*cases* (*0* $<$ *cx*))
apply (*metis diff-ge-0-iff-ge less-eq-real-def not-less-iff-gr-or-eq of-int-nonneg zero-le-mult-iff*)
apply (*metis cvc-arith-rewrite-defs*(*5*) *diff-ge-0-iff-ge mult-le-0-iff of-int-nonneg order-antisym-conv*)
by (*metis cvc-arith-rewrite-defs*(*5*) *diff-ge-0-iff-ge linorder-le-cases of-int-0-le-iff order-antisym-conv zero-le-mult-iff zero-less-mult-iff*)

end

118 Comprehensive Complex Theory

theory *Complex-Main*
imports
Complex
MacLaurin
Binomial-Plus
SMT-CVC-Real
begin

end

References

[1] H. Davenport. *The Higher Arithmetic.* Cambridge University Press, 1992.

[2] M. Gordon. HOL: A machine oriented formulation of higher-order logic. Technical Report 68, University of Cambridge, Computer Laboratory, 1985.

[3] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science.* Addison–Wesley, Boston, MA, USA, 2nd edition, 1994.

[4] M. Holz, K. Steffens, and E. Weitz. *Introduction to Cardinal Arithmetic.* Birkhäuser, 1999.

[5] D. Leijen. Division and modulus for computer scientists. 2001.

[6] C. Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J. Groote, editors, *Typed Lambda Calculi and Applications*, LNCS 664, pages 328–345. Springer, 1993.