The Supplemental Isabelle/HOL Library

October 24, 2026

Contents

1 Implementation of Association Lists **21**
- 1.1 *update* and *updates* . . . 21
- 1.2 *delete* . . . 24
- 1.3 *update-with-aux* and *delete-aux* . . . 25
- 1.4 *restrict* . . . 27
- 1.5 *clearjunk* . . . 28
- 1.6 *map-ran* . . . 30
- 1.7 *merge* . . . 31
- 1.8 *compose* . . . 32
- 1.9 *map-entry* . . . 36
- 1.10 *map-default* . . . 36

2 Axiomatic Declaration of Bounded Natural Functors **37**

3 Generalized Corecursor Sugar (corec and friends) **37**
- 3.1 Coinduction . . . 38

4 A general "while" combinator **41**
- 4.1 *while-option* . . . 42
- 4.2 *while* . . . 45
- 4.3 Termination, *lfp* and *gfp* . . . 46
- 4.4 *T-while-option* . . . 49
- 4.5 *while-Some* and *while-saturate* . . . 50
- 4.6 Reflexive, transitive closure . . . 52

5 The Bourbaki-Witt tower construction for transfinite iteration **54**
- 5.1 Connect with the while combinator for executability on chain-finite lattices. . . . 59

6 Division with modulus centered towards zero. **63**

7 Order on characters **66**

8 A generic phantom type **67**

9 Cardinality of types **68**
9.1 Cardinalities of types 68
9.2 Classes with at least 1 and 2 70
9.3 A type class for deciding finiteness of types 71
9.4 A type class for computing the cardinality of types 71
9.5 Instantiations for *card-UNIV* 72

10 Code setup for sets with cardinality type information **76**

11 Eliminating pattern matches **79**

12 Lazy types in generated code **79**
12.1 The type *lazy* 80
12.2 Implementation 82

13 Test infrastructure for the code generator **82**
13.1 YXML encoding for *term* 82
13.2 Test engine and drivers 85

14 A combinator to build partial equivalence relations from a predicate and an equivalence relation **85**

15 Formalisation of chain-complete partial orders, continuity and admissibility **86**
15.1 Continuity 93
15.1.1 Theorem collection *cont-intro* 93
15.2 Admissibility 103
15.3 (=) as order 108
15.4 ccpo for products 109
15.5 Complete lattices as ccpo 115
15.6 Parallel fixpoint induction 120

16 Confluence **125**

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums **131**
17.1 The datatype universe 131
17.2 Freeness: Distinctness of Constructors 133
17.3 Set Constructions 137

18 Bijections between natural numbers and other types **141**
18.1 Type *nat* × *nat* . . . 141
18.2 Type *nat* + *nat* . . . 143
18.3 Type *int* . . . 144
18.4 Type *nat list* . . . 145
18.5 Finite sets of naturals . . . 146
18.5.1 Preliminaries . . . 146
18.5.2 From sets to naturals . . . 147
18.5.3 From naturals to sets . . . 148
18.5.4 Proof of isomorphism . . . 148

19 Encoding (almost) everything into natural numbers **150**
19.1 The class of countable types . . . 150
19.2 Conversion functions . . . 150
19.3 Finite types are countable . . . 150
19.4 Automatically proving countability of old-style datatypes . . . 151
19.5 Automatically proving countability of datatypes . . . 153
19.6 More Countable types . . . 154
19.7 The rationals are countably infinite . . . 155

20 Infinite Sets and Related Concepts **156**
20.1 The set of natural numbers is infinite . . . 156
20.2 The set of integers is also infinite . . . 157
20.3 Infinitely Many and Almost All . . . 158
20.4 Enumeration of an Infinite Set . . . 161
20.5 Properties of *enumerate* on finite sets . . . 165

21 Countable sets **169**
21.1 Predicate for countable sets . . . 169
21.2 Enumerate a countable set . . . 170
21.3 Closure properties of countability . . . 173
21.4 Misc lemmas . . . 176
21.5 Uncountable . . . 178

22 Countable Complete Lattices **179**
22.0.1 Instances of countable complete lattices . . . 185

23 Type of (at Most) Countable Sets **185**
23.1 Cardinal stuff . . . 185
23.2 The type of countable sets . . . 186
23.3 Additional lemmas . . . 193
23.3.1 *cempty* . . . 193
23.3.2 *cinsert* . . . 193
23.3.3 *cimage* . . . 193

23.3.4 bounded quantification . . . 193
23.3.5 *cUnion* . . . 194
23.4 Setup for Lifting/Transfer . . . 194
23.4.1 Relator and predicator properties . . . 194
23.4.2 Transfer rules for the Transfer package . . . 194
23.5 Registration as BNF . . . 196

24 Debugging facilities for code generated towards Isabelle/ML 198

25 Sequence of Properties on Subsequences 199

26 Common discrete functions 201
26.1 Discrete logarithm . . . 201
26.2 Discrete square root . . . 205

27 Pi and Function Sets 210
27.1 Basic Properties of *Pi* . . . 211
27.2 Composition With a Restricted Domain: *compose* . . . 214
27.3 Bounded Abstraction: *restrict* . . . 214
27.4 Bijections Between Sets . . . 215
27.5 Extensionality . . . 216
27.6 Cardinality . . . 217
27.7 Extensional Function Spaces . . . 217
27.7.1 Injective Extensional Function Spaces . . . 223
27.7.2 Misc properties of functions, composition and restriction from HOL Light . . . 224
27.7.3 Cardinality . . . 224
27.8 The pigeonhole principle . . . 227
27.9 Products of sums . . . 227

28 Partitions and Disjoint Sets 228
28.1 Set of Disjoint Sets . . . 228
28.1.1 Family of Disjoint Sets . . . 230
28.2 Construct Disjoint Sequences . . . 233
28.3 Partitions . . . 234
28.4 Constructions of partitions . . . 235
28.5 Finiteness of partitions . . . 235
28.6 Equivalence of partitions and equivalence classes . . . 236
28.7 Refinement of partitions . . . 238
28.8 The coarsest common refinement of a set of partitions . . . 239

29 Type of finite sets defined as a subtype of sets 242
29.1 Definition of the type . . . 242
29.2 Basic operations and type class instantiations . . . 242
29.3 Other operations . . . 245

29.4 Transferred lemmas from Set.thy . . . 248
29.5 Additional lemmas . . . 264
29.5.1 *ffUnion* . . . 264
29.5.2 *fbind* . . . 264
29.5.3 *fsingleton* . . . 264
29.5.4 *femepty* . . . 264
29.5.5 *fset* . . . 264
29.5.6 *ffilter* . . . 265
29.5.7 *fset-of-list* . . . 265
29.5.8 *finsert* . . . 265
29.5.9 *fimage* . . . 266
29.5.10 bounded quantification . . . 266
29.5.11 *fcard* . . . 267
29.5.12 *sorted-list-of-fset* . . . 269
29.5.13 *ffold* . . . 269
29.5.14 (|⊂|) . . . 270
29.5.15 Group operations . . . 270
29.5.16 Semilattice operations . . . 271
29.6 Choice in fsets . . . 273
29.7 Induction and Cases rules for fsets . . . 273
29.8 Lemmas depending on induction . . . 275
29.9 Setup for Lifting/Transfer . . . 275
29.9.1 Relator and predicator properties . . . 275
29.9.2 Transfer rules for the Transfer package . . . 275
29.10 BNF setup . . . 278
29.11 Size setup . . . 279
29.12 Advanced relator customization . . . 280
29.12.1 Countability . . . 281
29.13 Quickcheck setup . . . 281
29.14 Code Generation Setup . . . 283

30 Type of finite maps defined as a subtype of maps 284
30.1 Auxiliary constants and lemmas over *map* . . . 284
30.2 Abstract characterisation . . . 286
30.3 Operations . . . 287
30.4 BNF setup . . . 301
30.5 *size* setup . . . 306
30.6 Additional operations . . . 307
30.7 Additional properties . . . 308
30.8 Lifting/transfer setup . . . 308
30.9 View as datatype . . . 309
30.10 Code setup . . . 311
30.11 Instances . . . 313
30.12 Tests . . . 314

31 Disjoint FSets **314**

32 Lists with elements distinct as canonical example for datatype invariants **316**
32.1 The type of distinct lists . . . 316
32.2 Executable version obeying invariant . . . 318
32.3 Induction principle and case distinction . . . 319
32.4 Functorial structure . . . 320
32.5 Quickcheck generators . . . 320
32.6 BNF instance . . . 320

33 Type of dual ordered lattices **322**
33.1 Pointwise ordering . . . 324
33.2 Binary infimum and supremum . . . 325
33.3 Top and bottom elements . . . 326
33.4 Complement . . . 327
33.5 Complete lattice operations . . . 328

34 Equipollence and Other Relations Connected with Cardinality **330**
34.1 Eqpoll . . . 330
34.2 The strict relation . . . 334
34.3 Mapping by an injection . . . 335
34.4 Inserting elements into sets . . . 336
34.5 Binary sums and unions . . . 337
34.6 Binary Cartesian products . . . 338
34.7 General Unions . . . 340
34.8 General Cartesian products (Pi) . . . 342
34.9 Misc other resultd . . . 346

35 Continuity and iterations **353**
35.1 Continuity for complete lattices . . . 353
35.1.1 Least fixed points in countable complete lattices . . . 361

36 Extended natural numbers (i.e. with infinity) **362**
36.1 Type definition . . . 363
36.2 Constructors and numbers . . . 364
36.3 Addition . . . 365
36.4 Multiplication . . . 366
36.5 Numerals . . . 367
36.6 Subtraction . . . 368
36.7 Ordering . . . 368
36.8 Cancellation simprocs . . . 372
36.9 Well-ordering . . . 374

36.10Complete Lattice . . . 374
36.11Traditional theorem names . . . 375

37 Liminf and Limsup on conditionally complete lattices 376
37.0.1 *Liminf* and *Limsup* . . . 377
37.1 More Limits . . . 388

38 Extended real number line 389
38.1 Definition and basic properties . . . 394
38.1.1 Addition . . . 396
38.1.2 Linear order on *ereal* . . . 399
38.1.3 Multiplication . . . 406
38.1.4 Power . . . 414
38.1.5 Subtraction . . . 414
38.1.6 Division . . . 418
38.2 Complete lattice . . . 423
38.3 Extended real intervals . . . 426
38.4 Topological space . . . 429
38.5 Relation to *enat* . . . 439
38.6 Limits on *ereal* . . . 441
38.6.1 Convergent sequences . . . 444
38.6.2 Sums . . . 455
38.6.3 Continuity . . . 465
38.6.4 liminf and limsup . . . 469
38.6.5 Tests for code generator . . . 473

39 Indicator Function 473

40 The type of non-negative extended real numbers 478
40.1 Defining the extended non-negative reals . . . 481
40.2 Cancellation simprocs . . . 486
40.3 Order with top . . . 487
40.4 Arithmetic . . . 490
40.5 Coercion from *real* to *ennreal* . . . 494
40.6 Coercion from *ennreal* to *real* . . . 500
40.7 Coercion from *enat* to *ennreal* . . . 501
40.8 Topology on *ennreal* . . . 502
40.9 Approximation lemmas . . . 514
40.10*ennreal* theorems . . . 517

41 Logarithm of Natural Numbers 521
41.1 Preliminaries . . . 521
41.2 Floorlog . . . 522
41.3 . . . 525

41.4 Bitlen 529

42 Various algebraic structures combined with a lattice 531
42.1 Positive Part, Negative Part, Absolute Value 533

43 Floating-Point Numbers 541
43.1 Real operations preserving the representation as floating point number 542
43.2 Arithmetic operations on floating point numbers 545
43.3 Quickcheck 547
43.4 Represent floats as unique mantissa and exponent 548
43.5 Compute arithmetic operations 552
43.6 Lemmas for types *real*, *nat*, *int* 554
43.7 Rounding Real Numbers 554
43.8 Rounding Floats 557
43.9 Truncating Real Numbers 560
43.10Truncating Floats 563
43.11Approximation of positive rationals 567
43.12Division 571
43.13Approximate Addition 572
43.14Approximate Multiplication 580
43.15Approximate Power 581
43.16Lemmas needed by Approximate 586

44 Pointwise instantiation of functions to algebra type classes 591

45 Pointwise instantiation of functions to division 596
45.1 Syntactic with division 596

46 Functions as a real vector spaces 597

47 Lexicographic order on functions 597

48 The *going-to* filter 599

49 Big sum and product over function bodies 602
49.1 Abstract product 602
49.2 Concrete sum 606
49.3 Concrete product 607

50 Infinite Type Class 608

51 Algebraic operations on sets 609

52 Interval Type **616**
52.1 Membership . . . 621
52.2 Quickcheck . . . 633

53 Approximate Operations on Intervals of Floating Point Numbers **634**
53.1 Intervals with Floating Point Bounds . . . 635
53.2 intros for *real-interval* . . . 636
53.3 bounds for lists . . . 637
53.4 constants for code generation . . . 642

54 Immutable Arrays with Code Generation **642**
54.1 Fundamental operations . . . 642
54.2 Generic code equations . . . 643
54.3 Auxiliary operations for code generation . . . 644
54.4 Code Generation for SML . . . 645
54.5 Code Generation for Haskell . . . 646

55 Definition of Landau symbols **647**
55.1 Definition of Landau symbols . . . 647
55.2 Landau symbols and limits . . . 669
55.3 Flatness of real functions . . . 682
55.4 Asymptotic Equivalence . . . 684

56 Values extended by a bottom element **696**
56.1 Values extended by a top element . . . 698
56.2 Values extended by a top and a bottom element . . . 701

57 Infinite Streams **706**
57.1 prepend list to stream . . . 707
57.2 set of streams with elements in some fixed set . . . 707
57.3 nth, take, drop for streams . . . 709
57.4 unary predicates lifted to streams . . . 712
57.5 recurring stream out of a list . . . 712
57.6 iterated application of a function . . . 713
57.7 stream repeating a single element . . . 714
57.8 stream of natural numbers . . . 715
57.9 flatten a stream of lists . . . 715
57.10merge a stream of streams . . . 716
57.11product of two streams . . . 717
57.12interleave two streams . . . 717
57.13zip . . . 718
57.14zip via function . . . 718

58 List prefixes, suffixes, and homeomorphic embedding 719
58.1 Prefix order on lists . . . 719
58.2 Basic properties of prefixes . . . 720
58.3 Prefixes . . . 725
58.4 Longest Common Prefix . . . 726
58.5 Parallel lists . . . 729
58.6 Suffix order on lists . . . 729
58.7 Suffixes . . . 735
58.8 Homeomorphic embedding on lists . . . 737
58.9 Subsequences (special case of homeomorphic embedding) . . . 740
58.10 Appending elements . . . 742
58.11 Relation to standard list operations . . . 743
58.12 Contiguous sublists . . . 744
58.12.1 *sublist* . . . 744
58.12.2 *sublists* . . . 748
58.13 Parametricity . . . 749

59 Linear Temporal Logic on Streams 751

60 Preliminaries 751

61 Linear temporal logic 751

62 Weak vs. strong until (contributed by Michael Foster, University of Sheffield) 767

63 Lists as vectors 769
63.1 + and − . . . 769
63.2 Inner product . . . 770

64 Definitions of Least Upper Bounds and Greatest Lower Bounds 772
64.1 Rules for the Relations ∗<= and <=∗ . . . 772
64.2 Rules about the Operators *leastP*, *ub* and *lub* . . . 773
64.3 Rules about the Operators *greatestP*, *isLb* and *isGlb* . . . 774

65 An abstract view on maps for code generation. 777
65.1 Parametricity transfer rules . . . 777
65.2 Type definition and primitive operations . . . 779
65.3 Functorial structure . . . 780
65.4 Derived operations . . . 780
65.5 Properties . . . 782
65.5.1 *entries*, *ordered-entries*, and *fold* . . . 792
65.6 Code generator setup . . . 798

66 Monad notation for arbitrary types 798

67 Less common functions on lists **799**

68 (Finite) Multisets **809**
68.1 The type of multisets . . . 809
68.2 Representing multisets . . . 810
68.3 Basic operations . . . 812
68.3.1 Conversion to set and membership . . . 812
68.3.2 Union . . . 814
68.3.3 Difference . . . 815
68.3.4 Min and Max . . . 817
68.3.5 Equality of multisets . . . 817
68.3.6 Pointwise ordering induced by count . . . 820
68.3.7 Intersection and bounded union . . . 824
68.3.8 Additional intersection facts . . . 825
68.3.9 Additional bounded union facts . . . 827
68.4 Replicate and repeat operations . . . 828
68.4.1 Simprocs . . . 829
68.4.2 Conditionally complete lattice . . . 831
68.4.3 Filter (with comprehension syntax) . . . 835
68.4.4 Size . . . 838
68.5 Induction and case splits . . . 840
68.5.1 Strong induction and subset induction for multisets . 842
68.6 Least and greatest elements . . . 843
68.7 The fold combinator . . . 843
68.8 Image . . . 845
68.9 Further conversions . . . 851
68.10 More properties of the replicate, repeat, and image operations 859
68.11 Big operators . . . 865
68.12 Multiset as order-ignorant lists . . . 874
68.13 The multiset order . . . 879
68.13.1 Well-foundedness . . . 880
68.13.2 Closure-free presentation . . . 882
68.13.3 Monotonicity . . . 885
68.13.4 The multiset extension is cancellative for multiset union 888
68.13.5 Strict partial-order properties . . . 889
68.13.6 Strict total-order properties . . . 892
68.14 Quasi-executable version of the multiset extension . . . 893
68.14.1 Monotonicity of multiset union . . . 895
68.14.2 Termination proofs with multiset orders . . . 896
68.15 Legacy theorem bindings . . . 899
68.16 Naive implementation using lists . . . 901
68.17 BNF setup . . . 904
68.18 Size setup . . . 911
68.19 Lemmas about Size . . . 911

68.20The set of multisets of a given size 912

69 More Theorems about the Multiset Order 916
69.1 Alternative Characterizations 916
69.1.1 The Dershowitz–Manna Ordering 916
69.1.2 The Huet–Oppen Ordering 917
69.1.3 Monotonicity 920
69.1.4 Properties of Orders 921
69.1.5 Simplifications 931
69.2 Simprocs 932
69.3 Additional facts and instantiations 932

70 Fixed Length Lists 935

71 Non-negative, non-positive integers and reals 938
71.1 Non-positive integers 938
71.2 Non-negative reals 941
71.3 Non-positive reals 942

72 Numeral Syntax for Types 946
72.1 Numeral Types 946
72.2 *num1* 947
72.3 Locales for for modular arithmetic subtypes 948
72.4 Ring class instances 950
72.5 Order instances 952
72.6 Code setup and type classes for code generation 953
72.7 Syntax 956
72.8 Examples 957

73 ω-words 957
73.1 Type declaration and elementary operations 958
73.2 Subsequence, Prefix, and Suffix 959
73.3 Prepending 963
73.4 The limit set of an ω-word 965
73.5 Index sequences and piecewise definitions 972

74 Combinator syntax for generic, open state monads (single-threaded monads) 976
74.1 Motivation 977
74.2 State transformations and combinators 977
74.3 Monad laws 978
74.4 Do-syntax 978

75 Canonical order on option type 979

76 Futures and parallel lists for code generated towards Isabelle/ML **989**
76.1 Futures . . . 989
76.2 Parallel lists . . . 989

77 Input syntax for pattern aliases (or "as-patterns" in Haskell) **990**
77.1 Definition . . . 991
77.2 Usage . . . 994

78 Periodic Functions **994**

79 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view **998**
79.1 Preliminary: auxiliary operations for *almost everywhere zero* . 998
79.2 Type definition . . . 1002
79.3 Additive structure . . . 1004
79.4 Multiplicative structure . . . 1006
79.5 Single-point mappings . . . 1011
79.6 Integral domains . . . 1014
79.7 Mapping order . . . 1015
79.8 Fundamental mapping notions . . . 1017
79.9 Degree . . . 1019
79.10 Inductive structure . . . 1021
79.11 Quasi-functorial structure . . . 1022
79.12 Canonical dense representation of $nat \Rightarrow_0 'a$ . . . 1024
79.13 Canonical sparse representation of $'a \Rightarrow_0 'b$ . . . 1027
79.14 Size estimation . . . 1028
79.15 Further mapping operations and properties . . . 1030
79.16 Free Abelian Groups Over a Type . . . 1030

80 Exponentiation by Squaring **1036**

81 Preorders with explicit equivalence relation **1037**

82 Additive group operations on product types **1039**
82.1 Operations . . . 1039
82.2 Class instances . . . 1040

83 Roots of real quadratics **1041**

84 Pretty syntax for Quotient operations **1044**

85 Quotient infrastructure for the set type **1044**
85.1 Contravariant set map (vimage) and set relator, rules for the Quotient package . . . 1044

86 Quotient infrastructure for the product type **1046**
86.1 Rules for the Quotient package 1046

87 Quotient infrastructure for the option type **1048**
87.1 Rules for the Quotient package 1049

88 Quotient infrastructure for the list type **1050**
88.1 Rules for the Quotient package 1050

89 Quotient infrastructure for the sum type **1054**
89.1 Rules for the Quotient package 1054

90 Quotient types **1056**
90.1 Equivalence relations and quotient types 1056
90.2 Equality on quotients 1058
90.3 Picking representing elements 1058

91 Ramsey's Theorem **1060**
91.1 Preliminary definitions 1060
91.1.1 The n-element subsets of a set A 1060
91.1.2 Further properties, involving equipollence 1064
91.1.3 Partition predicates 1065
91.2 Finite versions of Ramsey's theorem 1068
91.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers 1068
91.2.2 Trivial cases 1069
91.2.3 Ramsey's theorem with TWO colours and arbitrary exponents (hypergraph version) 1069
91.2.4 Full Ramsey's theorem with multiple colours and arbitrary exponents 1075
91.2.5 Simple graph version 1077
91.3 Preliminaries for the infinitary version 1078
91.3.1 "Axiom" of Dependent Choice 1078
91.3.2 Partition functions 1079
91.4 Ramsey's Theorem: Infinitary Version 1079
91.5 Disjunctive Well-Foundedness 1084

92 Modulo and congruence on the reals **1085**

93 Generic reflection and reification **1092**

94 Assigning lengths to types by type classes **1094**

95 Saturated arithmetic **1096**
95.1 The type of saturated naturals . . . 1096
95.2 Enumeration . . . 1101

96 Set Idioms **1102**
96.1 Idioms for being a suitable union/intersection of something . 1102
96.2 The "Relative to" operator . . . 1109

97 Signed division: negative results rounded towards zero rather than minus infinity. **1117**

98 State monad **1122**

99 Comparators on linear quasi-orders **1127**
99.1 Basic properties . . . 1127
99.2 Fundamental comparator combinators . . . 1131
99.3 Direct implementations for linear orders on selected types . . 1133

100 Stably sorted lists **1134**

101 Alternative sorting algorithms **1142**
101.1 Quicksort . . . 1142
101.2 Mergesort . . . 1143
101.3 Lexicographic products . . . 1148

102 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming **1150**

103 Time functions for various standard library operations **1151**
103.1 List . . . 1151

104 A table-based implementation of the reflexive transitive closure **1154**

105 Binary Tree **1159**
105.1 *map-tree* . . . 1161
105.2 *size* . . . 1161
105.3 *set-tree* . . . 1161
105.4 *subtrees* . . . 1162
105.5 *height* and *min-height* . . . 1163
105.6 *complete* . . . 1164
105.7 *acomplete* . . . 1165
105.8 *wbalanced* . . . 1166
105.9 *ipl* . . . 1166

105.10 List of entries . . . 1166
105.11 Binary Search Tree . . . 1167
105.12 *heap* . . . 1168
105.13 *mirror* . . . 1168

106 Multiset of Elements of Binary Tree 1168

107 Unordered pairs 1172

108 A type of finite bit strings 1177
108.1 Preliminaries . . . 1177
108.2 Fundamentals . . . 1177
108.2.1 Type definition . . . 1177
108.2.2 Basic arithmetic . . . 1177
108.2.3 Basic tool setup . . . 1179
108.2.4 Basic code generation setup . . . 1179
108.2.5 Basic conversions . . . 1180
108.3 Elementary case distinctions . . . 1187
108.3.1 Basic ordering . . . 1188
108.4 Enumeration . . . 1190
108.5 Bit-wise operations . . . 1191
108.6 Conversions including casts . . . 1201
108.6.1 Generic unsigned conversion . . . 1201
108.6.2 Generic signed conversion . . . 1204
108.6.3 More . . . 1205
108.7 Arithmetic operations . . . 1209
108.8 Ordering . . . 1212
108.9 Bit-wise operations . . . 1214
108.10 More shift operations . . . 1217
108.11 Single-bit operations . . . 1218
108.12 Rotation . . . 1218
108.13 Split and cat operations . . . 1221
108.14 More on conversions . . . 1222
108.15 Testing bits . . . 1225
108.16 Word Arithmetic . . . 1229
108.17 Transferring goals from words to ints . . . 1233
108.18 Order on fixed-length words . . . 1235
108.19 Conditions for the addition (etc) of two words to overflow . . . 1238
108.20 Some proof tool support . . . 1240
108.21 More on overflows and monotonicity . . . 1242
108.22 Arithmetic type class instantiations . . . 1246
108.23 Word and nat . . . 1247
108.24 Cardinality, finiteness of set of words . . . 1251
108.25 Bitwise Operations on Words . . . 1252

108.25.1 shift functions in terms of lists of bools 1256
108.25.2 Mask 1259
108.25.3 Slices 1261
108.25.4 Revcast 1262
108.26 Split and cat 1263
108.26.1 Split and slice 1263
108.27 Rotation 1265
108.27.1 Word rotation commutes with bit-wise operations . . 1267
108.28 Maximum machine word 1267
108.29 Recursion combinator for words 1271
108.30 Some more naive computations rules 1273
108.31 Executable intervals 1274
108.32 Tool support 1274

109 The Field of Integers mod 2 1275

110 Pointwise order on product types 1279
110.1 Pointwise ordering 1279
110.2 Binary infimum and supremum 1280
110.3 Top and bottom elements 1281
110.4 Complete lattice operations 1283
110.5 Complete distributive lattices 1284
110.6 Bekic's Theorem 1284

111 Finite Lattices 1286
111.1 Finite Complete Lattices 1286
111.2 Finite Distributive Lattices 1289
111.3 Linear Orders 1290
111.4 Finite Linear Orders 1291

112 Lexicographic order on lists 1291

113 Lexicographic order on lists 1293

114 Prefix order on lists as order class instance 1296

115 Lexicographic order on product types 1296

116 Subsequence Ordering 1299
116.1 Definitions and basic lemmas 1299

117 Records based on BNF/datatype machinery 1301

118 Implementation of mappings with Association Lists 1302

119 Avoidance of pattern matching on natural numbers 1310
119.1 Case analysis . . . 1310
119.2 Preprocessors . . . 1310
119.3 Candidates which need special treatment . . . 1312

120 Implementation of natural numbers as binary numerals 1312
120.1 Representation . . . 1312
120.2 Basic arithmetic . . . 1313
120.3 Conversions . . . 1314

121 Code generation of prolog programs 1315

122 Setup for Numerals 1315

123 Implementation of integer numbers by target-language integers 1315

124 Implementation of natural numbers by target-language integers 1323
124.1 Implementation for *nat* . . . 1323

125 Implementation of natural and integer numbers by target-language integers 1328

126 Preprocessor setup for floats implemented by target language numerals 1328

127 Abstract type of association lists with unique keys 1329
127.1 Preliminaries . . . 1329
127.2 Type (*'key*, *'value*) *alist* . . . 1330
127.3 Primitive operations . . . 1330
127.4 Abstract operation properties . . . 1331
127.5 Further operations . . . 1331
127.5.1 Equality . . . 1331
127.5.2 Size . . . 1331
127.6 Quickcheck generators . . . 1332

128 alist is a BNF 1334

129 Multisets partially implemented by association lists 1334

130 Implementation of Red-Black Trees 1344
130.1 Datatype of RB trees . . . 1344
130.2 Tree properties . . . 1344
130.2.1 Content of a tree . . . 1344
130.2.2 Search tree properties . . . 1345

130.2.3 Tree lookup . . . 1346
130.2.4 Red-black properties . . . 1349
130.3 Insertion . . . 1350
130.4 Deletion . . . 1355
130.5 Modifying existing entries . . . 1365
130.6 Mapping all entries . . . 1366
130.7 Folding over entries . . . 1367
130.8 Bulkloading a tree . . . 1367
130.9 Building a RBT from a sorted list . . . 1368
130.10 union and intersection of sorted associative lists . . . 1381
130.11 Code generator setup . . . 1409

131 Abstract type of RBT trees 1411
131.1 Type definition . . . 1411
131.2 Primitive operations . . . 1412
131.3 Derived operations . . . 1413
131.4 Abstract lookup properties . . . 1413
131.5 Quickcheck generators . . . 1416
131.6 Hide implementation details . . . 1416

132 Implementation of mappings with Red-Black Trees 1416
132.1 Data type and invariant . . . 1416
132.2 Operations . . . 1417
132.3 Invariant preservation . . . 1417
132.4 Map Semantics . . . 1418

133 Implementation of sets using RBT trees 1418

134 Definition of code datatype constructors 1418

135 Lemmas 1418
135.1 Auxiliary lemmas . . . 1418
135.2 fold and filter . . . 1419
135.3 foldi and Ball . . . 1420
135.4 foldi and Bex . . . 1420
135.5 folding over non empty trees and selecting the minimal and maximal element . . . 1420
135.5.1 concrete . . . 1420
135.5.2 abstract . . . 1425

136 Code equations 1427

137 Introduction 1435

138 Termination 1436

139 Partial Functions **1436**

140 Higher-Order Functions **1437**
140.1 Limitations 1438

141 Predefined Functions **1438**

142 Locales **1439**

143 Fine Points **1439**

144 Common constants **1442**

145 Pairs **1442**

146 Filters **1442**

147 Bounded quantifiers **1442**

148 Operations on Predicates **1442**

149 Setup for Numerals **1443**

150 Arithmetic operations **1443**
150.1 Arithmetic on naturals and integers 1443
150.2 Inductive definitions for ordering on naturals 1445

151 Alternative list definitions **1446**
151.1 Alternative rules for *length* 1446
151.2 Alternative rules for *list-all2* 1446
151.3 Alternative rules for membership in lists 1446

152 Setup for String.literal **1447**

153 Simplification rules for optimisation **1447**

154 A Prototype of Quickcheck based on the Predicate Compiler **1447**

155 TFL: recursive function definitions **1447**
155.1 Lemmas for TFL 1448
155.2 Rule setup 1449

156 Program extraction from proofs involving datatypes and inductive predicates **1449**

157 Refute **1449**

1 Implementation of Association Lists

theory *AList*
imports *Main*
begin

context
begin

The operations preserve distinctness of keys and function *clearjunk* distributes over them. Since *clearjunk* enforces distinctness of keys it can be used to establish the invariant, e.g. for inductive proofs.

1.1 *update* and *updates*

qualified primrec *update* :: $'key \Rightarrow 'val \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
update k v [] = [(*k*, *v*)]
| *update k v* (*p* # *ps*) = (*if fst p* = *k then* (*k*, *v*) # *ps else p* # *update k v ps*)

lemma *update-conv'*: *map-of* (*update k v al*) = (*map-of al*)(*k*↦*v*)
by (*induct al*) (*auto simp add*: *fun-eq-iff*)

corollary *update-conv*: *map-of* (*update k v al*) *k'* = ((*map-of al*)(*k*↦*v*)) *k'*
by (*simp add*: *update-conv'*)

lemma *dom-update*: *fst* ' *set* (*update k v al*) = {*k*} ∪ *fst* ' *set al*
by (*induct al*) *auto*

lemma *update-keys*:
map fst (*update k v al*) =
(*if k* ∈ *set* (*map fst al*) *then map fst al else map fst al* @ [*k*])
by (*induct al*) *simp-all*

lemma *distinct-update*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*update k v al*))
using *assms* **by** (*simp add*: *update-keys*)

lemma *update-filter*:
a ≠ *k* ⟹ *update k v* [*q*←*ps*. *fst q* ≠ *a*] = [*q*←*update k v ps*. *fst q* ≠ *a*]
by (*induct ps*) *auto*

lemma *update-triv*: *map-of al k* = *Some v* ⟹ *update k v al* = *al*
by (*induct al*) *auto*

lemma *update-nonempty* [*simp*]: *update k v al* ≠ []
by (*induct al*) *auto*

lemma *update-eqD*: *update k v al = update k v′ al′ ⟹ v = v′*
proof (*induct al arbitrary*: *al′*)
  case *Nil*
  then show *?case*
    by (*cases al′*) (*auto split*: *if-split-asm*)
next
  case *Cons*
  then show *?case*
    by (*cases al′*) (*auto split*: *if-split-asm*)
qed

lemma *update-last* [*simp*]: *update k v* (*update k v′ al*) = *update k v al*
  by (*induct al*) *auto*

Note that the lists are not necessarily the same: *update k v* (*update k′ v′* []) = [(*k′*, *v′*), (*k*, *v*)] and *update k′ v′* (*update k v* []) = [(*k*, *v*), (*k′*, *v′*)].

lemma *update-swap*:
  k ≠ k′ ⟹ map-of (*update k v* (*update k′ v′ al*)) = *map-of* (*update k′ v′* (*update k v al*))
  by (*simp add*: *update-conv′ fun-eq-iff*)

lemma *update-Some-unfold*:
  map-of (*update k v al*) *x* = *Some y* ⟷
    x = *k* ∧ *v* = *y* ∨ *x ≠ k* ∧ *map-of al x* = *Some y*
  by (*simp add*: *update-conv′ map-upd-Some-unfold*)

lemma *image-update* [*simp*]: *x ∉ A ⟹ map-of* (*update x y al*) ‘ *A* = *map-of al* ‘ *A*
  by (*auto simp add*: *update-conv′*)

qualified definition *updates* ::
    ′key list ⇒ ′val list ⇒ (*′key* × *′val*) *list ⇒* (*′key* × *′val*) *list*
  where *updates ks vs* = *fold* (*case-prod update*) (*zip ks vs*)

lemma *updates-simps* [*simp*]:
  updates [] *vs ps* = *ps*
  updates ks [] *ps* = *ps*
  updates (*k*#*ks*) (*v*#*vs*) *ps* = *updates ks vs* (*update k v ps*)
  by (*simp-all add*: *updates-def*)

lemma *updates-key-simp* [*simp*]:
  updates (*k* # *ks*) *vs ps* =
    (*case vs of* [] ⇒ *ps* | *v* # *vs* ⇒ *updates ks vs* (*update k v ps*))
  by (*cases vs*) *simp-all*

lemma *updates-conv′*: *map-of* (*updates ks vs al*) = (*map-of al*)(*ks*[↦]*vs*)
proof −
  have *map-of* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
      fold (λ(*k*, *v*) *f*. *f*(*k* ↦ *v*)) (*zip ks vs*) ∘ *map-of*

by (*rule fold-commute*) (*auto simp add*: *fun-eq-iff update-conv'*)
then show *?thesis*
by (*auto simp add*: *updates-def fun-eq-iff map-upds-fold-map-upd foldl-conv-fold split-def*)
qed

lemma *updates-conv*: *map-of* (*updates ks vs al*) *k* = ((*map-of al*)(*ks*[↦]*vs*)) *k*
by (*simp add*: *updates-conv'*)

lemma *distinct-updates*:
assumes *distinct* (*map fst al*)
shows *distinct* (*map fst* (*updates ks vs al*))
proof −
have *distinct* (*fold*
(λ(*k*, *v*) *al*. *if k* ∈ *set al then al else al* @ [*k*])
(*zip ks vs*) (*map fst al*))
by (*rule fold-invariant* [*of zip ks vs* λ-. *True*]) (*auto intro*: *assms*)
moreover have *map fst* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
fold (λ(*k*, *v*) *al*. *if k* ∈ *set al then al else al* @ [*k*]) (*zip ks vs*) ∘ *map fst*
by (*rule fold-commute*) (*simp add*: *update-keys split-def case-prod-beta comp-def*)
ultimately show *?thesis*
by (*simp add*: *updates-def fun-eq-iff*)
qed

lemma *updates-append1*[*simp*]: *size ks* < *size vs* ⟹
updates (*ks*@[*k*]) *vs al* = *update k* (*vs*!*size ks*) (*updates ks vs al*)
by (*induct ks arbitrary*: *vs al*) (*auto split*: *list.splits*)

lemma *updates-list-update-drop*[*simp*]:
size ks ≤ *i* ⟹ *i* < *size vs* ⟹
updates ks (*vs*[*i*:=*v*]) *al* = *updates ks vs al*
by (*induct ks arbitrary*: *al vs i*) (*auto split*: *list.splits nat.splits*)

lemma *update-updates-conv-if*:
map-of (*updates xs ys* (*update x y al*)) =
map-of
(*if x* ∈ *set* (*take* (*length ys*) *xs*)
then updates xs ys al
else (*update x y* (*updates xs ys al*)))
by (*simp add*: *updates-conv' update-conv' map-upd-upds-conv-if*)

lemma *updates-twist* [*simp*]:
k ∉ *set ks* ⟹
map-of (*updates ks vs* (*update k v al*)) = *map-of* (*update k v* (*updates ks vs al*))
by (*simp add*: *updates-conv' update-conv'*)

lemma *updates-apply-notin* [*simp*]:
k ∉ *set ks* ⟹ *map-of* (*updates ks vs al*) *k* = *map-of al k*
by (*simp add*: *updates-conv*)

lemma *updates-append-drop* [*simp*]:
size xs = size ys ⟹ updates (xs @ zs) ys al = updates xs ys al
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

lemma *updates-append2-drop* [*simp*]:
size xs = size ys ⟹ updates xs (ys @ zs) al = updates xs ys al
by (*induct xs arbitrary*: *ys al*) (*auto split*: *list.splits*)

1.2 *delete*

qualified definition *delete* :: *'key ⇒ ('key × 'val) list ⇒ ('key × 'val) list*
where *delete-eq*: *delete k = filter (λ(k', -). k ≠ k')*

lemma *delete-simps* [*simp*]:
delete k [] = []
delete k (p # ps) = (if fst p = k then delete k ps else p # delete k ps)
by (*auto simp add*: *delete-eq*)

lemma *delete-conv'*: *map-of (delete k al) = (map-of al)(k := None)*
by (*induct al*) (*auto simp add*: *fun-eq-iff*)

corollary *delete-conv*: *map-of (delete k al) k' = ((map-of al)(k := None)) k'*
by (*simp add*: *delete-conv'*)

lemma *delete-keys*: *map fst (delete k al) = removeAll k (map fst al)*
by (*simp add*: *delete-eq removeAll-filter-not-eq filter-map split-def comp-def*)

lemma *distinct-delete*:
assumes *distinct (map fst al)*
shows *distinct (map fst (delete k al))*
using *assms* **by** (*simp add*: *delete-keys distinct-removeAll*)

lemma *delete-id* [*simp*]: *k ∉ fst ` set al ⟹ delete k al = al*
by (*auto simp add*: *image-iff delete-eq filter-id-conv*)

lemma *delete-idem*: *delete k (delete k al) = delete k al*
by (*simp add*: *delete-eq*)

lemma *map-of-delete* [*simp*]: *k' ≠ k ⟹ map-of (delete k al) k' = map-of al k'*
by (*simp add*: *delete-conv'*)

lemma *delete-notin-dom*: *k ∉ fst ` set (delete k al)*
by (*auto simp add*: *delete-eq*)

lemma *dom-delete-subset*: *fst ` set (delete k al) ⊆ fst ` set al*
by (*auto simp add*: *delete-eq*)

lemma *delete-update-same*: *delete k (update k v al) = delete k al*

by (*induct al*) *simp-all*

lemma *delete-update*: $k \neq l \Longrightarrow$ *delete l* (*update k v al*) = *update k v* (*delete l al*)
by (*induct al*) *simp-all*

lemma *delete-twist*: *delete x* (*delete y al*) = *delete y* (*delete x al*)
by (*simp add*: *delete-eq conj-commute*)

lemma *length-delete-le*: *length* (*delete k al*) $\leq$ *length al*
by (*simp add*: *delete-eq*)

1.3 *update-with-aux* and *delete-aux*

qualified primrec *update-with-aux* ::
$'val \Rightarrow 'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val)$ *list* $\Rightarrow ('key \times 'val)$ *list*
where
update-with-aux v k f [] = [(*k*, *f v*)]
| *update-with-aux v k f* (*p* # *ps*) =
(*if* (*fst p* = *k*) *then* (*k*, *f* (*snd p*)) # *ps else p* # *update-with-aux v k f ps*)

The above *delete* traverses all the list even if it has found the key. This one does not have to keep going because is assumes the invariant that keys are distinct.

qualified fun *delete-aux* :: $'key \Rightarrow ('key \times 'val)$ *list* $\Rightarrow ('key \times 'val)$ *list*
where
delete-aux k [] = []
| *delete-aux k* ((*k'*, *v*) # *xs*) = (*if k* = *k' then xs else* (*k'*, *v*) # *delete-aux k xs*)

lemma *map-of-update-with-aux'*:
map-of (*update-with-aux v k f ps*) *k'* =
((*map-of ps*)(*k* $\mapsto$ (*case map-of ps k of None* $\Rightarrow$ *f v* | *Some v* $\Rightarrow$ *f v*))) *k'*
by (*induct ps*) *auto*

lemma *map-of-update-with-aux*:
map-of (*update-with-aux v k f ps*) =
(*map-of ps*)(*k* $\mapsto$ (*case map-of ps k of None* $\Rightarrow$ *f v* | *Some v* $\Rightarrow$ *f v*))
by (*simp add*: *fun-eq-iff map-of-update-with-aux'*)

lemma *dom-update-with-aux*: *fst* ' *set* (*update-with-aux v k f ps*) = {*k*} $\cup$ *fst* ' *set ps*
by (*induct ps*) *auto*

lemma *distinct-update-with-aux* [*simp*]:
distinct (*map fst* (*update-with-aux v k f ps*)) = *distinct* (*map fst ps*)
by (*induct ps*) (*auto simp add*: *dom-update-with-aux*)

lemma *set-update-with-aux*:
distinct (*map fst xs*) $\Longrightarrow$
set (*update-with-aux v k f xs*) =

(*set xs* − {*k*} × *UNIV* ∪ {(*k*, *f* (*case map-of xs k of None* ⇒ *v* | *Some v* ⇒ *v*))})
by (*induct xs*) (*auto intro*: *rev-image-eqI*)

lemma *set-delete-aux*: *distinct* (*map fst xs*) ⟹ *set* (*delete-aux k xs*) = *set xs* − {*k*} × *UNIV*
apply (*induct xs*)
apply *simp-all*
apply *clarsimp*
apply (*fastforce intro*: *rev-image-eqI*)
done

lemma *dom-delete-aux*: *distinct* (*map fst ps*) ⟹ *fst* ‘ *set* (*delete-aux k ps*) = *fst* ‘ *set ps* − {*k*}
by (*auto simp add*: *set-delete-aux*)

lemma *distinct-delete-aux* [*simp*]: *distinct* (*map fst ps*) ⟹ *distinct* (*map fst* (*delete-aux k ps*))
proof (*induct ps*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a ps*)
obtain *k′ v* **where** *a*: *a* = (*k′*, *v*)
by (*cases a*)
show *?case*
proof (*cases k′* = *k*)
case *True*
with *Cons a* **show** *?thesis* **by** *simp*
next
case *False*
with *Cons a* **have** *k′* ∉ *fst* ‘ *set ps distinct* (*map fst ps*)
by *simp-all*
with *False a* **have** *k′* ∉ *fst* ‘ *set* (*delete-aux k ps*)
by (*auto dest*!: *dom-delete-aux*[**where** *k*=*k*])
with *Cons a* **show** *?thesis*
by *simp*
qed
qed

lemma *map-of-delete-aux′*:
distinct (*map fst xs*) ⟹ *map-of* (*delete-aux k xs*) = (*map-of xs*)(*k* := *None*)
apply (*induct xs*)
apply (*fastforce simp add*: *map-of-eq-None-iff fun-upd-twist*)
apply (*auto intro*!: *ext*)
apply (*simp add*: *map-of-eq-None-iff*)
done

lemma *map-of-delete-aux*:

distinct (*map fst xs*) $\Longrightarrow$ *map-of* (*delete-aux k xs*) $k' = ((\textit{map-of}\ xs)(k := None))\ k'$
by (*simp add*: *map-of-delete-aux'*)

lemma *delete-aux-eq-Nil-conv*: *delete-aux k ts* $= [] \longleftrightarrow ts = [] \vee (\exists v.\ ts = [(k, v)])$
by (*cases ts*) (*auto split*: *if-split-asm*)

1.4 *restrict*

qualified definition *restrict* :: *'key set* $\Rightarrow$ (*'key* $\times$ *'val*) *list* $\Rightarrow$ (*'key* $\times$ *'val*) *list*
where *restrict-eq*: *restrict A* = *filter* ($\lambda(k, v).\ k \in A$)

lemma *restr-simps* [*simp*]:
restrict A [] = []
restrict A (*p*#*ps*) = (*if fst p* $\in$ *A then p* # *restrict A ps else restrict A ps*)
by (*auto simp add*: *restrict-eq*)

lemma *restr-conv'*: *map-of* (*restrict A al*) = ((*map-of al*)|‘ *A*)
proof
show *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k* **for** *k*
apply (*induct al*)
apply *simp*
apply (*cases k* $\in$ *A*)
apply *auto*
done
qed

corollary *restr-conv*: *map-of* (*restrict A al*) *k* = ((*map-of al*)|‘ *A*) *k*
by (*simp add*: *restr-conv'*)

lemma *distinct-restr*: *distinct* (*map fst al*) $\Longrightarrow$ *distinct* (*map fst* (*restrict A al*))
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-empty* [*simp*]:
restrict {} *al* = []
restrict A [] = []
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-in* [*simp*]: *x* $\in$ *A* $\Longrightarrow$ *map-of* (*restrict A al*) *x* = *map-of al x*
by (*simp add*: *restr-conv'*)

lemma *restr-out* [*simp*]: *x* $\notin$ *A* $\Longrightarrow$ *map-of* (*restrict A al*) *x* = *None*
by (*simp add*: *restr-conv'*)

lemma *dom-restr* [*simp*]: *fst* ‘ *set* (*restrict A al*) = *fst* ‘ *set al* $\cap$ *A*
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-upd-same* [*simp*]: *restrict* ($-\{x\}$) (*update x y al*) = *restrict* ($-\{x\}$) *al*

by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-restr* [*simp*]: *restrict A* (*restrict B al*) = *restrict* (*A*∩*B*) *al*
by (*induct al*) (*auto simp add*: *restrict-eq*)

lemma *restr-update*[*simp*]:
map-of (*restrict D* (*update x y al*)) =
map-of ((*if x* ∈ *D then* (*update x y* (*restrict* (*D*−{*x*}) *al*)) *else restrict D al*))
by (*simp add*: *restr-conv*′ *update-conv*′)

lemma *restr-delete* [*simp*]:
delete x (*restrict D al*) = (*if x* ∈ *D then restrict* (*D* − {*x*}) *al else restrict D al*)
apply (*simp add*: *delete-eq restrict-eq*)
apply (*auto simp add*: *split-def*)
proof −
have *y* ≠ *x* ⟷ *x* ≠ *y* **for** *y*
by *auto*
then show [*p* ← *al*. *fst p* ∈ *D* ∧ *x* ≠ *fst p*] = [*p* ← *al*. *fst p* ∈ *D* ∧ *fst p* ≠ *x*]
by *simp*
assume *x* ∉ *D*
then have *y* ∈ *D* ⟷ *y* ∈ *D* ∧ *x* ≠ *y* **for** *y*
by *auto*
then show [*p* ← *al* . *fst p* ∈ *D* ∧ *x* ≠ *fst p*] = [*p* ← *al* . *fst p* ∈ *D*]
by *simp*
qed

lemma *update-restr*:
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
by (*simp add*: *update-conv*′ *restr-conv*′) (*rule fun-upd-restrict*)

lemma *update-restr-conv* [*simp*]:
x ∈ *D* ⟹
map-of (*update x y* (*restrict D al*)) = *map-of* (*update x y* (*restrict* (*D* − {*x*}) *al*))
by (*simp add*: *update-conv*′ *restr-conv*′)

lemma *restr-updates* [*simp*]:
length xs = *length ys* ⟹ *set xs* ⊆ *D* ⟹
map-of (*restrict D* (*updates xs ys al*)) =
map-of (*updates xs ys* (*restrict* (*D* − *set xs*) *al*))
by (*simp add*: *updates-conv*′ *restr-conv*′)

lemma *restr-delete-twist*: (*restrict A* (*delete a ps*)) = *delete a* (*restrict A ps*)
by (*induct ps*) *auto*

1.5 *clearjunk*

qualified function *clearjunk* :: ('*key* × '*val*) *list* ⇒ ('*key* × '*val*) *list*

where
clearjunk [] = []
| *clearjunk* (*p*#*ps*) = *p* # *clearjunk* (*delete* (*fst p*) *ps*)
by *pat-completeness auto*
termination
by (*relation measure length*) (*simp-all add*: *less-Suc-eq-le length-delete-le*)

lemma *map-of-clearjunk*: *map-of* (*clearjunk al*) = *map-of al*
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *fun-eq-iff*)

lemma *clearjunk-keys-set*: *set* (*map fst* (*clearjunk al*)) = *set* (*map fst al*)
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *delete-keys*)

lemma *dom-clearjunk*: *fst* ' *set* (*clearjunk al*) = *fst* ' *set al*
using *clearjunk-keys-set* **by** *simp*

lemma *distinct-clearjunk* [*simp*]: *distinct* (*map fst* (*clearjunk al*))
by (*induct al rule*: *clearjunk.induct*) (*simp-all del*: *set-map add*: *clearjunk-keys-set delete-keys*)

lemma *ran-clearjunk*: *ran* (*map-of* (*clearjunk al*)) = *ran* (*map-of al*)
by (*simp add*: *map-of-clearjunk*)

lemma *ran-map-of*: *ran* (*map-of al*) = *snd* ' *set* (*clearjunk al*)
proof −
have *ran* (*map-of al*) = *ran* (*map-of* (*clearjunk al*))
by (*simp add*: *ran-clearjunk*)
also have ... = *snd* ' *set* (*clearjunk al*)
by (*simp add*: *ran-distinct*)
finally show *?thesis* **.**
qed

lemma *graph-map-of*: *Map.graph* (*map-of al*) = *set* (*clearjunk al*)
by (*metis distinct-clearjunk graph-map-of-if-distinct-dom map-of-clearjunk*)

lemma *clearjunk-update*: *clearjunk* (*update k v al*) = *update k v* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*simp-all add*: *delete-update*)

lemma *clearjunk-updates*: *clearjunk* (*updates ks vs al*) = *updates ks vs* (*clearjunk al*)
proof −
have *clearjunk* ∘ *fold* (*case-prod update*) (*zip ks vs*) =
fold (*case-prod update*) (*zip ks vs*) ∘ *clearjunk*
by (*rule fold-commute*) (*simp add*: *clearjunk-update case-prod-beta o-def*)
then show *?thesis*
by (*simp add*: *updates-def fun-eq-iff*)
qed

lemma *clearjunk-delete*: *clearjunk* (*delete x al*) = *delete x* (*clearjunk al*)

by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *delete-idem delete-twist*)

lemma *clearjunk-restrict*: *clearjunk* (*restrict A al*) = *restrict A* (*clearjunk al*)
by (*induct al rule*: *clearjunk.induct*) (*auto simp add*: *restr-delete-twist*)

lemma *distinct-clearjunk-id* [*simp*]: *distinct* (*map fst al*) ⟹ *clearjunk al* = *al*
by (*induct al rule*: *clearjunk.induct*) *auto*

lemma *clearjunk-idem*: *clearjunk* (*clearjunk al*) = *clearjunk al*
by *simp*

lemma *length-clearjunk*: *length* (*clearjunk al*) ≤ *length al*
proof (*induct al rule*: *clearjunk.induct* [*case-names Nil Cons*])
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons kv al*)
moreover have *length* (*delete* (*fst kv*) *al*) ≤ *length al*
by (*fact length-delete-le*)
ultimately have *length* (*clearjunk* (*delete* (*fst kv*) *al*)) ≤ *length al*
by (*rule order-trans*)
then show *?case*
by *simp*
qed

lemma *delete-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *delete k* (*map f ps*) = *map f* (*delete k ps*)
by (*simp add*: *delete-eq filter-map comp-def split-def assms*)

lemma *clearjunk-map*:
assumes ⋀*kv*. *fst* (*f kv*) = *fst kv*
shows *clearjunk* (*map f ps*) = *map f* (*clearjunk ps*)
by (*induct ps rule*: *clearjunk.induct* [*case-names Nil Cons*])
(*simp-all add*: *clearjunk-delete delete-map assms*)

1.6 *map-ran*

definition *map-ran* :: ('*key* ⇒ '*val1* ⇒ '*val2*) ⇒ ('*key* × '*val1*) *list* ⇒ ('*key* × '*val2*) *list*
where *map-ran f* = *map* (λ(*k*, *v*). (*k*, *f k v*))

lemma *map-ran-simps* [*simp*]:
map-ran f [] = []
map-ran f ((*k*, *v*) # *ps*) = (*k*, *f k v*) # *map-ran f ps*
by (*simp-all add*: *map-ran-def*)

lemma *map-ran-Cons-sel*: *map-ran f* (*p* # *ps*) = (*fst p*, *f* (*fst p*) (*snd p*)) # *map-ran f ps*

by (*simp add*: *map-ran-def case-prod-beta*)

lemma *length-map-ran*[*simp*]: *length* (*map-ran f al*) = *length al*
by (*simp add*: *map-ran-def*)

lemma *map-fst-map-ran*[*simp*]: *map fst* (*map-ran f al*) = *map fst al*
by (*simp add*: *map-ran-def case-prod-beta*)

lemma *dom-map-ran*: *fst ‘ set* (*map-ran f al*) = *fst ‘ set al*
by (*simp add*: *map-ran-def image-image split-def*)

lemma *map-ran-conv*: *map-of* (*map-ran f al*) *k* = *map-option* (*f k*) (*map-of al k*)
by (*induct al*) *auto*

lemma *distinct-map-ran*: *distinct* (*map fst al*) ⟹ *distinct* (*map fst* (*map-ran f al*))
by *simp*

lemma *map-ran-filter*: *map-ran f* [*p*←*ps*. *fst p* ≠ *a*] = [*p*←*map-ran f ps*. *fst p* ≠ *a*]
by (*simp add*: *map-ran-def filter-map split-def comp-def*)

lemma *clearjunk-map-ran*: *clearjunk* (*map-ran f al*) = *map-ran f* (*clearjunk al*)
by (*simp add*: *map-ran-def split-def clearjunk-map*)

1.7 *merge*

qualified definition *merge* :: (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list* ⇒ (*'key* × *'val*) *list*
where *merge qs ps* = *foldr* (λ(*k*, *v*). *update k v*) *ps qs*

lemma *merge-simps* [*simp*]:
merge qs [] = *qs*
merge qs (*p*#*ps*) = *update* (*fst p*) (*snd p*) (*merge qs ps*)
by (*simp-all add*: *merge-def split-def*)

lemma *merge-updates*: *merge qs ps* = *updates* (*rev* (*map fst ps*)) (*rev* (*map snd ps*)) *qs*
by (*simp add*: *merge-def updates-def foldr-conv-fold zip-rev zip-map-fst-snd*)

lemma *dom-merge*: *fst ‘ set* (*merge xs ys*) = *fst ‘ set xs* ∪ *fst ‘ set ys*
by (*induct ys arbitrary*: *xs*) (*auto simp add*: *dom-update*)

lemma *distinct-merge*: *distinct* (*map fst xs*) ⟹ *distinct* (*map fst* (*merge xs ys*))
by (*simp add*: *merge-updates distinct-updates*)

lemma *clearjunk-merge*: *clearjunk* (*merge xs ys*) = *merge* (*clearjunk xs*) *ys*
by (*simp add*: *merge-updates clearjunk-updates*)

lemma *merge-conv'*: *map-of* (*merge xs ys*) = *map-of xs* ++ *map-of ys*
proof −
have *map-of* ∘ *fold* (*case-prod update*) (*rev ys*) =
fold (λ(*k*, *v*) *m*. *m*(*k* ↦ *v*)) (*rev ys*) ∘ *map-of*
by (*rule fold-commute*) (*simp add*: *update-conv'* *case-prod-beta split-def fun-eq-iff*)
then show *?thesis*
by (*simp add*: *merge-def map-add-map-of-foldr foldr-conv-fold fun-eq-iff*)
qed

corollary *merge-conv*: *map-of* (*merge xs ys*) *k* = (*map-of xs* ++ *map-of ys*) *k*
by (*simp add*: *merge-conv'*)

lemma *merge-empty*: *map-of* (*merge* [] *ys*) = *map-of ys*
by (*simp add*: *merge-conv'*)

lemma *merge-assoc* [*simp*]: *map-of* (*merge m1* (*merge m2 m3*)) = *map-of* (*merge* (*merge m1 m2*) *m3*)
by (*simp add*: *merge-conv'*)

lemma *merge-Some-iff*:
map-of (*merge m n*) *k* = *Some x* ⟷
map-of n k = *Some x* ∨ *map-of n k* = *None* ∧ *map-of m k* = *Some x*
by (*simp add*: *merge-conv'* *map-add-Some-iff*)

lemmas *merge-SomeD* [*dest*!] = *merge-Some-iff* [*THEN iffD1*]

lemma *merge-find-right* [*simp*]: *map-of n k* = *Some v* ⟹ *map-of* (*merge m n*) *k* = *Some v*
by (*simp add*: *merge-conv'*)

lemma *merge-None* [*iff*]: (*map-of* (*merge m n*) *k* = *None*) = (*map-of n k* = *None* ∧ *map-of m k* = *None*)
by (*simp add*: *merge-conv'*)

lemma *merge-upd* [*simp*]: *map-of* (*merge m* (*update k v n*)) = *map-of* (*update k v* (*merge m n*))
by (*simp add*: *update-conv'* *merge-conv'*)

lemma *merge-updatess* [*simp*]:
map-of (*merge m* (*updates xs ys n*)) = *map-of* (*updates xs ys* (*merge m n*))
by (*simp add*: *updates-conv'* *merge-conv'*)

lemma *merge-append*: *map-of* (*xs* @ *ys*) = *map-of* (*merge ys xs*)
by (*simp add*: *merge-conv'*)

1.8 *compose*

qualified function *compose* :: ('*key* × '*a*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*key* × '*b*) *list*
where

```
    compose [] ys = []
  | compose (x # xs) ys =
      (case map-of ys (snd x) of
        None ⇒ compose (delete (fst x) xs) ys
      | Some v ⇒ (fst x, v) # compose xs ys)
  by pat-completeness auto
termination
  by (relation measure (length ∘ fst)) (simp-all add: less-Suc-eq-le length-delete-le)

lemma compose-first-None [simp]: map-of xs k = None ⟹ map-of (compose xs
ys) k = None
  by (induct xs ys rule: compose.induct) (auto split: option.splits if-split-asm)

lemma compose-conv: map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: map-of (compose (delete (fst x) xs) ys) k =
        (map-of ys ∘m map-of (delete (fst x) xs)) k
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      from True delete-notin-dom [of k xs]
      have map-of (delete (fst x) xs) k = None
        by (simp add: map-of-eq-None-iff)
      with hyp show ?thesis
        using True None
        by simp
    next
      case False
      from False have map-of (delete (fst x) xs) k = map-of xs k
        by simp
      with hyp show ?thesis
        using False None by (simp add: map-comp-def)
    qed
  next
    case (Some v)
    with 2
    have map-of (compose xs ys) k = (map-of ys ∘m map-of xs) k
      by simp
    with Some show ?thesis
      by (auto simp add: map-comp-def)
  qed
```

qed

lemma *compose-conv'*: *map-of* (*compose xs ys*) = (*map-of ys* $\circ_m$ *map-of xs*)
by (*rule ext*) (*rule compose-conv*)

lemma *compose-first-Some* [*simp*]: *map-of xs k* = *Some v* $\Longrightarrow$ *map-of* (*compose xs ys*) *k* = *map-of ys v*
by (*simp add*: *compose-conv*)

lemma *dom-compose*: *fst* ' *set* (*compose xs ys*) $\subseteq$ *fst* ' *set xs*
proof (*induct xs ys rule*: *compose.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 x xs ys*)
show *?case*
proof (*cases map-of ys* (*snd x*))
case *None*
with *2.hyps* **have** *fst* ' *set* (*compose* (*delete* (*fst x*) *xs*) *ys*) $\subseteq$ *fst* ' *set* (*delete* (*fst x*) *xs*)
by *simp*
also have ... $\subseteq$ *fst* ' *set xs*
by (*rule dom-delete-subset*)
finally show *?thesis*
using *None* **by** *auto*
next
case (*Some v*)
with *2.hyps* **have** *fst* ' *set* (*compose xs ys*) $\subseteq$ *fst* ' *set xs*
by *simp*
with *Some* **show** *?thesis*
by *auto*
qed
qed

lemma *distinct-compose*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*compose xs ys*))
using *assms*
proof (*induct xs ys rule*: *compose.induct*)
case *1*
then show *?case* **by** *simp*
next
case (*2 x xs ys*)
show *?case*
proof (*cases map-of ys* (*snd x*))
case *None*
with *2* **show** *?thesis* **by** *simp*
next
case (*Some v*)

```
    with 2 dom-compose [of xs ys] show ?thesis
      by auto
  qed
qed

lemma compose-delete-twist: compose (delete k xs) ys = delete k (compose xs ys)
proof (induct xs ys rule: compose.induct)
  case 1
  then show ?case by simp
next
  case (2 x xs ys)
  show ?case
  proof (cases map-of ys (snd x))
    case None
    with 2 have hyp: compose (delete k (delete (fst x) xs)) ys =
        delete k (compose (delete (fst x) xs) ys)
      by simp
    show ?thesis
    proof (cases fst x = k)
      case True
      with None hyp show ?thesis
        by (simp add: delete-idem)
    next
      case False
      from None False hyp show ?thesis
        by (simp add: delete-twist)
    qed
  next
    case (Some v)
    with 2 have hyp: compose (delete k xs) ys = delete k (compose xs ys)
      by simp
    with Some show ?thesis
      by simp
  qed
qed

lemma compose-clearjunk: compose xs (clearjunk ys) = compose xs ys
  by (induct xs ys rule: compose.induct)
    (auto simp add: map-of-clearjunk split: option.splits)

lemma clearjunk-compose: clearjunk (compose xs ys) = compose (clearjunk xs) ys
  by (induct xs rule: clearjunk.induct)
   (auto split: option.splits simp add: clearjunk-delete delete-idem compose-delete-twist)

lemma compose-empty [simp]: compose xs [] = []
  by (induct xs) (auto simp add: compose-delete-twist)

lemma compose-Some-iff:
  (map-of (compose xs ys) k = Some v) ⟷
```

$(\exists k'.\ map\text{-}of\ xs\ k = Some\ k' \wedge map\text{-}of\ ys\ k' = Some\ v)$
by (*simp add*: *compose-conv map-comp-Some-iff*)

lemma *map-comp-None-iff*:
$map\text{-}of\ (compose\ xs\ ys)\ k = None \longleftrightarrow$
$(map\text{-}of\ xs\ k = None \vee (\exists k'.\ map\text{-}of\ xs\ k = Some\ k' \wedge map\text{-}of\ ys\ k' = None))$
by (*simp add*: *compose-conv map-comp-None-iff*)

1.9 *map-entry*

qualified fun *map-entry* :: $'key \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
$map\text{-}entry\ k\ f\ [] = []$
| $map\text{-}entry\ k\ f\ (p\ \#\ ps) =$
$(if\ fst\ p = k\ then\ (k,\ f\ (snd\ p))\ \#\ ps\ else\ p\ \#\ map\text{-}entry\ k\ f\ ps)$

lemma *map-of-map-entry*:
$map\text{-}of\ (map\text{-}entry\ k\ f\ xs) =$
$(map\text{-}of\ xs)(k := case\ map\text{-}of\ xs\ k\ of\ None \Rightarrow None \mid Some\ v' \Rightarrow Some\ (f\ v'))$
by (*induct xs*) *auto*

lemma *dom-map-entry*: $fst\ `\ set\ (map\text{-}entry\ k\ f\ xs) = fst\ `\ set\ xs$
by (*induct xs*) *auto*

lemma *distinct-map-entry*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-entry k f xs*))
using *assms* **by** (*induct xs*) (*auto simp add*: *dom-map-entry*)

1.10 *map-default*

fun *map-default* :: $'key \Rightarrow 'val \Rightarrow ('val \Rightarrow 'val) \Rightarrow ('key \times 'val)\ list \Rightarrow ('key \times 'val)\ list$
where
$map\text{-}default\ k\ v\ f\ [] = [(k,\ v)]$
| $map\text{-}default\ k\ v\ f\ (p\ \#\ ps) =$
$(if\ fst\ p = k\ then\ (k,\ f\ (snd\ p))\ \#\ ps\ else\ p\ \#\ map\text{-}default\ k\ v\ f\ ps)$

lemma *map-of-map-default*:
$map\text{-}of\ (map\text{-}default\ k\ v\ f\ xs) =$
$(map\text{-}of\ xs)(k := case\ map\text{-}of\ xs\ k\ of\ None \Rightarrow Some\ v \mid Some\ v' \Rightarrow Some\ (f\ v'))$
by (*induct xs*) *auto*

lemma *dom-map-default*: $fst\ `\ set\ (map\text{-}default\ k\ v\ f\ xs) = insert\ k\ (fst\ `\ set\ xs)$
by (*induct xs*) *auto*

lemma *distinct-map-default*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*map-default k v f xs*))

using *assms* **by** (*induct xs*) (*auto simp add*: *dom-map-default*)

end

end

2 Axiomatic Declaration of Bounded Natural Functors

theory *BNF-Axiomatization*
imports *Main*
keywords
bnf-axiomatization :: *thy-decl*
begin

ML-file ‹../*Tools*/*BNF*/*bnf-axiomatization.ML*›

end

3 Generalized Corecursor Sugar (corec and friends)

theory *BNF-Corec*
imports *Main*
keywords
corec :: *thy-defn* **and**
corecursive :: *thy-goal-defn* **and**
friend-of-corec :: *thy-goal-defn* **and**
coinduction-upto :: *thy-decl*
begin

lemma *obj-distinct-prems*: $P \longrightarrow P \longrightarrow Q \Longrightarrow P \Longrightarrow Q$
by *auto*

lemma *inject-refine*: $g\ (f\ x) = x \Longrightarrow g\ (f\ y) = y \Longrightarrow f\ x = f\ y \longleftrightarrow x = y$
by (*metis* (*no-types*))

lemma *convol-apply*: *BNF-Def.convol f g x* = (*f x*, *g x*)
unfolding *convol-def* **..**

lemma *Grp-UNIV-id*: *BNF-Def.Grp UNIV id* = (=)
unfolding *BNF-Def.Grp-def* **by** *auto*

lemma *sum-comp-cases*:
assumes $f \circ Inl = g \circ Inl$ **and** $f \circ Inr = g \circ Inr$
shows $f = g$
proof (*rule ext*)
fix *a* **show** *f a* = *g a*
using *assms* **unfolding** *comp-def fun-eq-iff* **by** (*cases a*) *auto*

qed

lemma *case-sum-Inl-Inr-L*: *case-sum* $(f \circ Inl)$ $(f \circ Inr) = f$
by (*metis case-sum-expand-Inr'*)

lemma *eq-o-InrI*: $[\![g \circ Inl = h;$ *case-sum* $h\ f = g]\!] \Longrightarrow f = g \circ Inr$
by (*auto simp*: *fun-eq-iff split*: *sum.splits*)

lemma *id-bnf-o*: *BNF-Composition.id-bnf* $\circ\ f = f$
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *o-id-bnf*: $f \circ$ *BNF-Composition.id-bnf* $= f$
unfolding *BNF-Composition.id-bnf-def* **by** (*rule o-def*)

lemma *if-True-False*:
(*if P then True else Q*) $\longleftrightarrow P \vee Q$
(*if P then False else Q*) $\longleftrightarrow \neg P \wedge Q$
(*if P then Q else True*) $\longleftrightarrow \neg P \vee Q$
(*if P then Q else False*) $\longleftrightarrow P \wedge Q$
by *auto*

lemma *if-distrib-fun*: (*if c then f else g*) $x =$ (*if c then f x else g x*)
by *simp*

3.1 Coinduction

lemma *eq-comp-compI*: $a \circ b = f \circ x \Longrightarrow x \circ c = id \Longrightarrow f = a \circ (b \circ c)$
unfolding *fun-eq-iff* **by** *simp*

lemma *self-bounded-weaken-left*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ a\ b \Longrightarrow a \leq b$
by (*erule le-infE*)

lemma *self-bounded-weaken-right*: $(a :: 'a :: semilattice\text{-}inf) \leq inf\ b\ a \Longrightarrow a \leq b$
by (*erule le-infE*)

lemma *symp-iff*: *symp* $R \longleftrightarrow R = R^{-1-1}$
by (*metis antisym conversep.cases conversep-le-swap predicate2I symp-def*)

lemma *equivp-inf*: $[\![equivp\ R;$ *equivp* $S]\!] \Longrightarrow$ *equivp* $(inf\ R\ S)$
unfolding *equivp-def inf-fun-def inf-bool-def* **by** *metis*

lemma *vimage2p-rel-prod*:
$(\lambda x\ y.$ *rel-prod R S* (*BNF-Def.convol f1 g1 x*) (*BNF-Def.convol f2 g2 y*)) =
(*inf* (*BNF-Def.vimage2p f1 f2 R*) (*BNF-Def.vimage2p g1 g2 S*))
unfolding *vimage2p-def rel-prod.simps convol-def* **by** *auto*

lemma *predicate2I-obj*: $(\forall x\ y.\ P\ x\ y \longrightarrow Q\ x\ y) \Longrightarrow P \leq Q$
by *auto*

lemma *predicate2D-obj*: $P \leq Q \Longrightarrow P\ x\ y \longrightarrow Q\ x\ y$
by *auto*

locale *cong* =
fixes *rel* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool)$
and *eval* :: $'b \Rightarrow 'a$
and *retr* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool)$
assumes *rel-mono*: $\bigwedge R\ S.\ R \leq S \Longrightarrow rel\ R \leq rel\ S$
and *equivp-retr*: $\bigwedge R.\ equivp\ R \Longrightarrow equivp\ (retr\ R)$
and *retr-eval*: $\bigwedge R\ x\ y.\ [\![(rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval;\ rel\ (inf\ R\ (retr\ R))\ x\ y]\!]$
$\Longrightarrow$
$retr\ R\ (eval\ x)\ (eval\ y)$
begin

definition *cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow bool$ **where**
$cong\ R \equiv equivp\ R \wedge (rel\text{-}fun\ (rel\ R)\ R)\ eval\ eval$

lemma *cong-retr*: $cong\ R \Longrightarrow cong\ (inf\ R\ (retr\ R))$
unfolding *cong-def*
by (*auto simp*: *rel-fun-def dest*: *predicate2D*[*OF rel-mono*, *rotated*]
intro: *equivp-inf equivp-retr retr-eval*)

lemma *cong-equivp*: $cong\ R \Longrightarrow equivp\ R$
unfolding *cong-def* **by** *simp*

definition *gen-cong* :: $('a \Rightarrow 'a \Rightarrow bool) \Rightarrow 'a \Rightarrow 'a \Rightarrow bool$ **where**
$gen\text{-}cong\ R\ j1\ j2 \equiv \forall R'.\ R \leq R' \wedge cong\ R' \longrightarrow R'\ j1\ j2$

lemma *gen-cong-reflp*[*intro*, *simp*]: $x = y \Longrightarrow gen\text{-}cong\ R\ x\ y$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-reflp*)

lemma *gen-cong-symp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ x$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-symp*)

lemma *gen-cong-transp*[*intro*]: $gen\text{-}cong\ R\ x\ y \Longrightarrow gen\text{-}cong\ R\ y\ z \Longrightarrow gen\text{-}cong\ R\ x\ z$
unfolding *gen-cong-def* **by** (*auto dest*: *cong-equivp equivp-transp*)

lemma *equivp-gen-cong*: $equivp\ (gen\text{-}cong\ R)$
by (*intro equivpI reflpI sympI transpI*) *auto*

lemma *leq-gen-cong*: $R \leq gen\text{-}cong\ R$
unfolding *gen-cong-def*[*abs-def*] **by** *auto*

lemmas *imp-gen-cong*[*intro*] = *predicate2D*[*OF leq-gen-cong*]

lemma *gen-cong-minimal*: $[\![R \leq R';\ cong\ R']\!] \Longrightarrow gen\text{-}cong\ R \leq R'$
unfolding *gen-cong-def*[*abs-def*] **by** (*rule predicate2I*) *metis*

lemma *congdd-base-gen-congdd-base-aux*:
rel (*gen-cong R*) *x y* ⟹ *R* ≤ *R'* ⟹ *cong R'* ⟹ *R'* (*eval x*) (*eval y*)
by (*force simp*: *rel-fun-def gen-cong-def cong-def dest*: *spec*[*of* - *R'*] *predicate2D*[*OF rel-mono*, *rotated* −*1*, *of* - - - *R'*])

lemma *cong-gen-cong*: *cong* (*gen-cong R*)
proof −
have *rel* (*gen-cong R*) *x y* ⟹ *R* ≤ *R'* ⟹ *cong R'* ⟹ *R'* (*eval x*) (*eval y*) **for** *R' x y*
by (*force simp*: *rel-fun-def gen-cong-def cong-def dest*: *spec*[*of* - *R'*]
predicate2D[*OF rel-mono*, *rotated* −*1*, *of* - - - *R'*])
then show *cong* (*gen-cong R*) **by** (*auto simp*: *equivp-gen-cong rel-fun-def gen-cong-def cong-def*)
qed

lemma *gen-cong-eval-rel-fun*:
(*rel-fun* (*rel* (*gen-cong R*)) (*gen-cong R*)) *eval eval*
using *cong-gen-cong*[*of R*] **unfolding** *cong-def* **by** *simp*

lemma *gen-cong-eval*:
rel (*gen-cong R*) *x y* ⟹ *gen-cong R* (*eval x*) (*eval y*)
by (*erule rel-funD*[*OF gen-cong-eval-rel-fun*])

lemma *gen-cong-idem*: *gen-cong* (*gen-cong R*) = *gen-cong R*
by (*simp add*: *antisym cong-gen-cong gen-cong-minimal leq-gen-cong*)

lemma *gen-cong-rho*:
ϱ = *eval* ∘ *f* ⟹ *rel* (*gen-cong R*) (*f x*) (*f y*) ⟹ *gen-cong R* (ϱ *x*) (ϱ *y*)
by (*simp add*: *gen-cong-eval*)
lemma *coinduction*:
assumes *coind*: ∀ *R*. *R* ≤ *retr R* ⟶ *R* ≤ (=)
assumes *cih*: *R* ≤ *retr* (*gen-cong R*)
shows *R* ≤ (=)
apply (*rule order-trans*[*OF leq-gen-cong mp*[*OF spec*[*OF coind*]]])
apply (*rule self-bounded-weaken-left*[*OF gen-cong-minimal*])
apply (*rule inf-greatest*[*OF leq-gen-cong cih*])
apply (*rule cong-retr*[*OF cong-gen-cong*])
done

end

lemma *rel-sum-case-sum*:
rel-fun (*rel-sum R S*) *T* (*case-sum f1 g1*) (*case-sum f2 g2*) = (*rel-fun R T f1 f2* ∧ *rel-fun S T g1 g2*)
by (*auto simp*: *rel-fun-def rel-sum.simps split*: *sum.splits*)

context
fixes *rel eval rel' eval' retr emb*
assumes *base*: *cong rel eval retr*

and *step*: *cong rel′ eval′ retr*
and *emb*: *eval′* ∘ *emb* = *eval*
and *emb-transfer*: *rel-fun* (*rel R*) (*rel′ R*) *emb emb*
begin

interpretation *base*: *cong rel eval retr* **by** (*rule base*)
interpretation *step*: *cong rel′ eval′ retr* **by** (*rule step*)

lemma *gen-cong-emb*: *base.gen-cong R* ≤ *step.gen-cong R*
proof (*rule base.gen-cong-minimal*[*OF step.leq-gen-cong*])
note *step.gen-cong-eval-rel-fun*[*transfer-rule*] *emb-transfer*[*transfer-rule*]
have (*rel-fun* (*rel* (*step.gen-cong R*)) (*step.gen-cong R*)) *eval eval*
unfolding *emb*[*symmetric*] **by** *transfer-prover*
then show *base.cong* (*step.gen-cong R*)
by (*auto simp*: *base.cong-def step.equivp-gen-cong*)
qed

end

named-theorems *friend-of-corec-simps*

ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec-tactics.ML*›
ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec.ML*›
ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec-sugar-util.ML*›
ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec-sugar-tactics.ML*›
ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec-sugar.ML*›
ML-file ‹../*Tools*/*BNF*/*bnf-gfp-grec-unique-sugar.ML*›

method-setup *transfer-prover-eq* = ‹
Scan.succeed (*SIMPLE-METHOD′ o BNF-GFP-Grec-Tactics.transfer-prover-eq-tac*)
› *apply transfer-prover after folding relator-eq*

method-setup *corec-unique* = ‹
Scan.succeed (*SIMPLE-METHOD′ o BNF-GFP-Grec-Unique-Sugar.corec-unique-tac*)
› *prove uniqueness of corecursive equation*

end

4 A general "while" combinator

theory *While-Combinator*
imports *Main*
begin

Defining partial functions in HOL is tricky. This theory provides a while-combinator that facilitates the definition of (potentially) partial tail-recursive functions.

The theory provides the function *while-option b f s* that iterates *f* on *s* while *b* is true. If iteration terminates with *t*, *Some t* is returned, *None*

otherwise. Thus termination can be shown by proving that *Some* is always returned (for some subset of inputs).

Convenient variations include *while-Some* (for more efficient code) and *while-saturate* (for saturating a set).

4.1 *while-option*

definition *while-option* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow 'a \Rightarrow 'a\ option$ **where**
while-option b c s = (*if* $(\exists k.\ \neg\ b\ ((c\ ^^\ k)\ s))$
then Some $((c\ ^^\ (LEAST\ k.\ \neg\ b\ ((c\ ^^\ k)\ s)))\ s)$
else None)

theorem *while-option-unfold*[*code*]:
while-option b c s = (*if b s then while-option b c* (*c s*) *else Some s*)
proof *cases*
assume *b s*
show *?thesis*
proof (*cases* $\exists k.\ \neg\ b\ ((c\ ^^\ k)\ s)$)
case *True*
then obtain *k* **where** *1*: $\neg\ b\ ((c\ ^^\ k)\ s)$ **..**
with ‹*b s*› **obtain** *l* **where** *k* = *Suc l* **by** (*cases k*) *auto*
with *1* **have** $\neg\ b\ ((c\ ^^\ l)\ (c\ s))$ **by** (*auto simp*: *funpow-swap1*)
then have *2*: $\exists l.\ \neg\ b\ ((c\ ^^\ l)\ (c\ s))$ **..**
from *1*
have $(LEAST\ k.\ \neg\ b\ ((c\ ^^\ k)\ s)) = Suc\ (LEAST\ l.\ \neg\ b\ ((c\ ^^\ Suc\ l)\ s))$
by (*rule Least-Suc*) (*simp add*: ‹*b s*›)
also have ... = $Suc\ (LEAST\ l.\ \neg\ b\ ((c\ ^^\ l)\ (c\ s)))$
by (*simp add*: *funpow-swap1*)
finally
show *?thesis*
using *True 2* ‹*b s*› **by** (*simp add*: *funpow-swap1 while-option-def*)
next
case *False*
then have $\neg\ (\exists l.\ \neg\ b\ ((c\ ^^\ Suc\ l)\ s))$ **by** *blast*
then have $\neg\ (\exists l.\ \neg\ b\ ((c\ ^^\ l)\ (c\ s)))$
by (*simp add*: *funpow-swap1*)
with *False* ‹*b s*› **show** *?thesis* **by** (*simp add*: *while-option-def*)
qed
next
assume [*simp*]: $\neg\ b\ s$
have *least*: $(LEAST\ k.\ \neg\ b\ ((c\ ^^\ k)\ s)) = 0$
by (*rule Least-equality*) *auto*
moreover
have $\exists k.\ \neg\ b\ ((c\ ^^\ k)\ s)$ **by** (*rule exI*[*of - 0*::*nat*]) *auto*
ultimately show *?thesis* **unfolding** *while-option-def* **by** *auto*
qed

lemma *while-option-stop2*:
while-option b c s = *Some t* $\Longrightarrow \exists k.\ t = (c\hat{}\hat{}k)\ s \wedge \neg\ b\ t$

apply(*simp add*: *while-option-def split*: *if-splits*)
by (*metis* (*lifting*) *LeastI-ex*)

lemma *while-option-stop*: *while-option b c s* = *Some t* ⟹ ¬ *b t*
by(*metis while-option-stop2*)

theorem *while-option-rule*:
  assumes *step*: ⋀*s*. *P s* ⟹ *b s* ⟹ *P* (*c s*)
    and *result*: *while-option b c s* = *Some t*
    and *init*: *P s*
  shows *P t*
proof −
  define *k* **where** *k* = (*LEAST k*. ¬ *b* ((*c* ^^ *k*) *s*))
  from *assms* **have** *t*: *t* = (*c* ^^ *k*) *s*
    by (*simp add*: *while-option-def k-def split*: *if-splits*)
  have *1*: ∀ *i*<*k*. *b* ((*c* ^^ *i*) *s*)
    by (*auto simp*: *k-def dest*: *not-less-Least*)
  have *i* ≤ *k* ⟹ *P* ((*c* ^^ *i*) *s*) **for** *i*
    by (*induct i*) (*auto simp*: *init step 1*)
  thus *P t* **by** (*auto simp*: *t*)
qed

lemma *funpow-commute*:
  ⟦∀ *k'* < *k*. *f* (*c* ((*c*^^*k'*) *s*)) = *c'* (*f* ((*c*^^*k'*) *s*))⟧ ⟹ *f* ((*c*^^*k*) *s*) = (*c'*^^*k*) (*f s*)
by (*induct k arbitrary*: *s*) *auto*

lemma *while-option-commute-invariant*:
assumes *Invariant*: ⋀*s*. *P s* ⟹ *b s* ⟹ *P* (*c s*)
assumes *TestCommute*: ⋀*s*. *P s* ⟹ *b s* = *b'* (*f s*)
assumes *BodyCommute*: ⋀*s*. *P s* ⟹ *b s* ⟹ *f* (*c s*) = *c'* (*f s*)
assumes *Initial*: *P s*
shows *map-option f* (*while-option b c s*) = *while-option b' c'* (*f s*)
unfolding *while-option-def*
proof (*rule trans*[*OF if-distrib if-cong*], *safe*, *unfold option.inject*)
  fix *k*
  assume ¬ *b* ((*c* ^^ *k*) *s*)
  with *Initial* **show** ∃ *k*. ¬ *b'* ((*c'* ^^ *k*) (*f s*))
  proof (*induction k arbitrary*: *s*)
    case *0* **thus** *?case* **by** (*auto simp*: *TestCommute intro*: *exI*[*of* - *0*])
  next
    case (*Suc k*) **thus** *?case*
    proof (*cases b s*)
      assume *b s*
      with *Suc.IH*[*of c s*] *Suc.prems* **show** *?thesis*
      by (*metis BodyCommute Invariant comp-apply funpow.simps*(*2*) *funpow-swap1*)
    next
      assume ¬ *b s*
      with *Suc* **show** *?thesis* **by** (*auto simp*: *TestCommute intro*: *exI* [*of* - *0*])
    qed

```
  qed
next
  fix k
  assume ¬ b' ((c' ^^ k) (f s))
  with Initial show ∃k. ¬ b ((c ^^ k) s)
  proof (induction k arbitrary: s)
    case 0 thus ?case by (auto simp: TestCommute intro: exI[of - 0])
  next
    case (Suc k) thus ?case
    proof (cases b s)
      assume b s
      with Suc.IH[of c s] Suc.prems show ?thesis
      by (metis BodyCommute Invariant comp-apply funpow.simps(2) funpow-swap1)
    next
      assume ¬ b s
      with Suc show ?thesis by (auto simp: TestCommute intro: exI [of - 0])
    qed
  qed
next
  fix k
  assume k: ¬ b' ((c' ^^ k) (f s))
  have *: (LEAST k. ¬ b' ((c' ^^ k) (f s))) = (LEAST k. ¬ b ((c ^^ k) s))
        (is ?k' = ?k)
  proof (cases ?k')
    case 0
    have ¬ b' ((c' ^^ 0) (f s))
      unfolding 0[symmetric] by (rule LeastI[of - k]) (rule k)
    hence ¬ b s by (auto simp: TestCommute Initial)
    hence ?k = 0 by (intro Least-equality) auto
    with 0 show ?thesis by auto
  next
    case (Suc k')
    have ¬ b' ((c' ^^ Suc k') (f s))
      unfolding Suc[symmetric] by (rule LeastI) (rule k)
    moreover
    have b': b' ((c' ^^ k) (f s)) if asm: k ≤ k' for k
    proof −
      from asm have k < ?k' unfolding Suc by simp
      thus ?thesis by (rule iffD1[OF not-not, OF not-less-Least])
    qed
    have b: b ((c ^^ k) s)
      and body: f ((c ^^ k) s) = (c' ^^ k) (f s)
      and inv: P ((c ^^ k) s)
      if asm: k ≤ k' for k
    proof −
      from asm have f ((c ^^ k) s) = (c' ^^ k) (f s)
        and b ((c ^^ k) s) = b' ((c' ^^ k) (f s))
        and P ((c ^^ k) s)
        by (induct k) (auto simp: b' assms)
```

```
      with ‹k ≤ k'›
      show b ((c ^^ k) s)
        and f ((c ^^ k) s) = (c' ^^ k) (f s)
        and P ((c ^^ k) s)
        by (auto simp: b')
    qed
    hence k': f ((c ^^ k') s) = (c' ^^ k') (f s) by auto
    ultimately show ?thesis unfolding Suc using b
    proof (intro Least-equality[symmetric], goal-cases)
      case 1
      hence Test: ¬ b' (f ((c ^^ Suc k') s))
        by (auto simp: BodyCommute inv b)
      have P ((c ^^ Suc k') s) by (auto simp: Invariant inv b)
      with Test show ?case by (auto simp: TestCommute)
    next
      case 2
      thus ?case by (metis not-less-eq-eq)
    qed
  qed
  have f ((c ^^ ?k) s) = (c' ^^ ?k') (f s) unfolding *
  proof (rule funpow-commute, clarify)
    fix k assume k < ?k
    hence TestTrue: b ((c ^^ k) s) by (auto dest: not-less-Least)
    from ‹k < ?k› have P ((c ^^ k) s)
    proof (induct k)
      case 0 thus ?case by (auto simp: assms)
    next
      case (Suc h)
      hence P ((c ^^ h) s) by auto
      with Suc show ?case
        by (auto, metis (lifting, no-types) Invariant Suc-lessD not-less-Least)
    qed
    with TestTrue show f (c ((c ^^ k) s)) = c' (f ((c ^^ k) s))
      by (metis BodyCommute)
  qed
  thus ∃ z. (c ^^ ?k) s = z ∧ f z = (c' ^^ ?k') (f s) by blast
qed

lemma while-option-commute:
  assumes ⋀s. b s = b' (f s) ⋀s. ⟦b s⟧ ⟹ f (c s) = c' (f s)
  shows map-option f (while-option b c s) = while-option b' c' (f s)
by(rule while-option-commute-invariant[where P = λ-. True])
  (auto simp add: assms)
```

4.2 *while*

```
definition while :: ('a ⇒ bool) ⇒ ('a ⇒ 'a) ⇒ 'a ⇒ 'a
where while b c s = the (while-option b c s)
```

lemma *while-unfold* [*code*]:
while b c s = (*if b s then while b c* (*c s*) *else s*)
unfolding *while-def* **by** (*subst while-option-unfold*) *simp*

lemma *def-while-unfold*:
assumes *fdef*: *f* == *while test do*
shows *f x* = (*if test x then f*(*do x*) *else x*)
unfolding *fdef* **by** (*fact while-unfold*)

The proof rule for *while*, where P is the invariant.

theorem *while-rule-lemma*:
assumes *invariant*: $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P\ (c\ s)$
and *terminate*: $\bigwedge s.\ P\ s \Longrightarrow \neg\ b\ s \Longrightarrow Q\ s$
and *wf*: *wf* $\{(t,\ s).\ P\ s \wedge b\ s \wedge t = c\ s\}$
shows $P\ s \Longrightarrow Q$ (*while b c s*)
using *wf*
apply (*induct s*)
apply *simp*
apply (*subst while-unfold*)
apply (*simp add*: *invariant terminate*)
done

theorem *while-rule*:
$[\![P\ s;$
$\bigwedge s.\ [\![P\ s;\ b\ s]\!] \Longrightarrow P\ (c\ s);$
$\bigwedge s.\ [\![P\ s;\ \neg\ b\ s]\!] \Longrightarrow Q\ s;$
wf r;
$\bigwedge s.\ [\![P\ s;\ b\ s]\!] \Longrightarrow (c\ s,\ s) \in r]\!] \Longrightarrow$
Q (*while b c s*)
apply (*rule while-rule-lemma*)
prefer *4* **apply** *assumption*
apply *blast*
apply *blast*
apply (*erule wf-subset*)
apply *blast*
done

Combine invariant preservation and variant decrease in one goal:

theorem *while-rule2*:
$[\![P\ s;$
$\bigwedge s.\ [\![P\ s;\ b\ s]\!] \Longrightarrow P\ (c\ s) \wedge (c\ s,\ s) \in r;$
$\bigwedge s.\ [\![P\ s;\ \neg\ b\ s]\!] \Longrightarrow Q\ s;$
wf r$]\!] \Longrightarrow$
Q (*while b c s*)
using *while-rule*[*of P*] **by** *metis*

4.3 Termination, *lfp* and *gfp*

theorem *wf-while-option-Some*:
assumes *wf* $\{(t,\ s).\ (P\ s \wedge b\ s) \wedge t = c\ s\}$

and $\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s)$ **and** $P\ s$
shows $\exists t.$ *while-option* $b\ c\ s = Some\ t$
using *assms*(*1*,*3*)
proof (*induction s*)
case *less* **thus** *?case* **using** *assms*(*2*)
by (*subst while-option-unfold*) *simp*
qed

lemma *wf-rel-while-option-Some*:
assumes *wf*: *wf R*
assumes *smaller*: $\bigwedge s.\ P\ s \wedge b\ s \Longrightarrow (c\ s,\ s) \in R$
assumes *inv*: $\bigwedge s.\ P\ s \wedge b\ s \Longrightarrow P(c\ s)$
assumes *init*: $P\ s$
shows $\exists t.$ *while-option* $b\ c\ s = Some\ t$
proof −
from *smaller* **have** $\{(t,s).\ P\ s \wedge b\ s \wedge t = c\ s\} \subseteq R$ **by** *auto*
with *wf* **have** *wf* $\{(t,s).\ P\ s \wedge b\ s \wedge t = c\ s\}$ **by** (*auto simp*: *wf-subset*)
with *inv init* **show** *?thesis* **by** (*auto simp*: *wf-while-option-Some*)
qed

theorem *measure-while-option-Some*: **fixes** $f :: 's \Rightarrow nat$
shows $(\bigwedge s.\ P\ s \Longrightarrow b\ s \Longrightarrow P(c\ s) \wedge f(c\ s) < f\ s)$
$\Longrightarrow P\ s \Longrightarrow \exists t.$ *while-option* $b\ c\ s = Some\ t$
by(*blast intro*: *wf-while-option-Some*[*OF wf-if-measure*, *of P b f*])

Kleene iteration starting from the empty set and assuming some finite bounding set:

lemma *while-option-finite-subset-Some*: **fixes** $C :: 'a\ set$
assumes *mono f* **and** $\bigwedge X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows $\exists P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ \{\} = Some\ P$
proof(*rule measure-while-option-Some*[**where**
$f = \%A::'a\ set.\ card\ C - card\ A$ **and** $P = \%A.\ A \subseteq C \wedge A \subseteq f\ A$ **and** $s = \{\}$])
fix *A* **assume** *A*: $A \subseteq C \wedge A \subseteq f\ A\ f\ A \neq A$
show $(f\ A \subseteq C \wedge f\ A \subseteq f\ (f\ A)) \wedge card\ C - card\ (f\ A) < card\ C - card\ A$
(**is** *?L* $\wedge$ *?R*)
proof
show *?L* **by** (*metis A*(*1*) *assms*(*2*) *monoD*[*OF* ‹*mono f*›])
show *?R* **by** (*metis A assms*(*2*,*3*) *card-seteq diff-less-mono2 equalityI linorder-le-less-linear rev-finite-subset*)
qed
qed *simp*

lemma *lfp-the-while-option*:
assumes *mono f* **and** $\bigwedge X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows *lfp f* = *the*(*while-option* $(\lambda A.\ f\ A \neq A)\ f\ \{\}$)
proof−
obtain *P* **where** *while-option* $(\lambda A.\ f\ A \neq A)\ f\ \{\} = Some\ P$
using *while-option-finite-subset-Some*[*OF assms*] **by** *blast*

with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

lemma *lfp-while*:
assumes *mono f* **and** $\bigwedge X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C*
shows *lfp f* = *while* ($\lambda A.\ f\ A \neq A$) *f* {}
unfolding *while-def* **using** *assms* **by** (*rule lfp-the-while-option*) *blast*

lemma *wf-finite-less*:
assumes *finite* (*C* :: *'a*::*order set*)
shows *wf* {(*x*, *y*). {*x*, *y*} ⊆ *C* ∧ *x* < *y*}
by (*rule wf-measure*[**where** *f*=λ*b*. *card* {*a*. *a* ∈ *C* ∧ *a* < *b*}, *THEN wf-subset*])
(*fastforce simp*: *less-eq assms intro*: *psubset-card-mono*)

lemma *wf-finite-greater*:
assumes *finite* (*C* :: *'a*::*order set*)
shows *wf* {(*x*, *y*). {*x*, *y*} ⊆ *C* ∧ *y* < *x*}
by (*rule wf-measure*[**where** *f*=λ*b*. *card* {*a*. *a* ∈ *C* ∧ *b* < *a*}, *THEN wf-subset*])
(*fastforce simp*: *less-eq assms intro*: *psubset-card-mono*)

lemma *while-option-finite-increasing-Some*:
fixes *f* :: *'a*::*order* ⇒ *'a*
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*) **and** *s* ≤ *f s*
shows ∃ *P*. *while-option* (λ*A*. *f A* ≠ *A*) *f s* = *Some P*
by (*rule wf-rel-while-option-Some*[**where** *R*={(*x*, *y*). *y* < *x*} **and** *P*=λ*A*. *A* ≤ *f A* **and** *s*=*s*])
(*auto simp*: *assms monoD intro*: *wf-finite-greater*[**where** *C*=*UNIV*::*'a set*, *simplified*])

lemma *lfp-the-while-option-lattice*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*)
shows *lfp f* = *the* (*while-option* (λ*A*. *f A* ≠ *A*) *f bot*)
proof −
obtain *P* **where** *while-option* (λ*A*. *f A* ≠ *A*) *f bot* = *Some P*
using *while-option-finite-increasing-Some*[*OF assms*, **where** *s*=*bot*] **by** *simp blast*
with *while-option-stop2*[*OF this*] *lfp-Kleene-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

lemma *lfp-while-lattice*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*)
shows *lfp f* = *while* (λ*A*. *f A* ≠ *A*) *f bot*
unfolding *while-def* **using** *assms* **by** (*rule lfp-the-while-option-lattice*)

lemma *while-option-finite-decreasing-Some*:

fixes $f :: 'a::order \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*) **and** $f\ s \leq s$
shows $\exists P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ s =$ *Some P*
by (*rule wf-rel-while-option-Some*[**where** $R=\{(x,\ y).\ x < y\}$ **and** $P=\lambda A.\ f\ A \leq A$ **and** $s=s$])
(*auto simp add*: *assms monoD intro*: *wf-finite-less*[**where** $C=UNIV::'a$ *set*, *simplified*])

lemma *gfp-the-while-option-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*)
shows *gfp f* = *the*(*while-option* $(\lambda A.\ f\ A \neq A)$ *f top*)
proof −
obtain *P* **where** *while-option* $(\lambda A.\ f\ A \neq A)$ *f top* = *Some P*
using *while-option-finite-decreasing-Some*[*OF assms*, **where** *s*=*top*] **by** *simp blast*
with *while-option-stop2*[*OF this*] *gfp-Kleene-iter*[*OF assms*(*1*)]
show *?thesis* **by** *auto*
qed

lemma *gfp-while-lattice*:
fixes $f :: 'a::complete\text{-}lattice \Rightarrow 'a$
assumes *mono f* **and** *finite* (*UNIV* :: *'a set*)
shows *gfp f* = *while* $(\lambda A.\ f\ A \neq A)$ *f top*
unfolding *while-def* **using** *assms* **by** (*rule gfp-the-while-option-lattice*)

4.4 *T-while-option*

Step counting for while-loops: add a counter.

definition *T-while-option2*
:: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow nat) \Rightarrow ('a \Rightarrow nat) \Rightarrow 'a \times nat \Rightarrow ('a \times nat)$ *option* **where**
T-while-option2 b f T-b T-f = *while-option* $(\lambda(x,n).\ b\ x)$ $(\lambda(x,n).\ (f\ x,\ n + T\text{-}b\ x + T\text{-}f\ x))$

definition *T-while-option* :: $('a \Rightarrow bool) \Rightarrow ('a \Rightarrow 'a) \Rightarrow ('a \Rightarrow nat) \Rightarrow ('a \Rightarrow nat) \Rightarrow 'a \Rightarrow nat$ **where**
T-while-option b f T-b T-f x = *snd* (*the* (*T-while-option2 b f T-b T-f* $(x,0)$))

lemma *T-while-option-eq-T-while-option2*:
while-option b f s = *map-option fst* (*T-while-option2 b f tb tf* $(s,n0)$)
unfolding *T-while-option2-def*
using *while-option-commute*[*of* $\lambda(s,\text{-}).\ b\ s$ *b fst* $(\lambda(s,\ n).\ (f\ s,\ n + tb\ s + tf\ s))$ *f* $(s,n0)$]
by (*simp add*: *split-def*)

lemma *T-while-option-if-T-while-option2*:
T-while-option2 b f tb tf $(s,n0)$ = *Some* $(t,n) \Longrightarrow$ *while-option b f s* = *Some t*
by(*simp add*: *T-while-option-eq-T-while-option2*[*of b f s tb tf n0*])

lemma *T-while-option2-if-T-while-option*:
while-option b f s = Some t $\Longrightarrow$ $\exists$ *n. T-while-option2 b f tb tf* (*s*,*n0*) = *Some* (*t*,*n*)
by(*simp add*: *T-while-option-eq-T-while-option2*[*of b f s tb tf n0*])

4.5 *while-Some* and *while-saturate*

A variation intended for efficient code. The problem with *while-option b c*: the computations of *b* and *c* may share subcomputations but they need to be performed twice.

definition *while-Some* :: ('*s* $\Rightarrow$ '*s option*) $\Rightarrow$ '*s* $\Rightarrow$ '*s option* **where**
while-Some f = *while-option* (λ*s*. *f s* $\neq$ *None*) (*the o f*)

lemma *while-Some-rec*[*code*]:
while-Some f x = (*case f x of None* $\Rightarrow$ *Some x* | *Some y* $\Rightarrow$ *while-Some f y*)
unfolding *while-Some-def while-option-unfold*[*of - - x*] **by** *auto*

A frequent special case: saturation of a set.

definition *while-saturate* :: ('*a set* $\Rightarrow$ '*a set*) $\Rightarrow$ '*a set* $\Rightarrow$ '*a set option* **where**
while-saturate f = *while-option* (λ*M*. $\neg$ *f M* $\subseteq$ *M*) (λ*M*. *M* $\cup$ *f M*)

lemma *while-option-cong*: ($\bigwedge$*s*. *b s* $\Longrightarrow$ *c s* = *c' s*) $\Longrightarrow$ *while-option b c s* = *while-option b c' s*
using *while-option-commute*[*of b b id c c'*]
by (*simp add*: *option.map-id*)

lemma *while-saturate-code*[*code*]: *while-saturate f M* =
while-Some (λ*M*. *let M'* = *f M in if M'* $\subseteq$ *M then None else Some* (*M* $\cup$ *M'*)) *M*
unfolding *while-saturate-def Let-def while-Some-def*
by (*auto intro*!: *while-option-cong split*: *if-splits*)

lemma *while-saturate-saturated*: *while-saturate F S0* = *Some S* $\Longrightarrow$ *F S* $\subseteq$ *S*
by (*metis* (*mono-tags*, *lifting*) *while-option-stop while-saturate-def*)

Termination:

lemma *while-option-sat-finite-subset-Some*: **fixes** *C* :: '*a set*
assumes *mono f* **and** $\bigwedge$*X*. *X* $\subseteq$ *C* $\Longrightarrow$ *f X* $\subseteq$ *C* **and** *finite C* **and** *M* $\subseteq$ *C*
shows $\exists$ *S*. *while-option* (λ*M*. $\neg$ *f M* $\subseteq$ *M*) (λ*M*. *M* $\cup$ *f M*) *M* = *Some S*
proof(*rule measure-while-option-Some*[**where**
f= %*A*::'*a set*. *card C* $-$ *card A* **and** *P*= %*A*. *M* $\subseteq$ *A* $\wedge$ *A* $\subseteq$ *C* **and** *s*= *M*])
fix *A* **assume** *A*: *M* $\subseteq$ *A* $\wedge$ *A* $\subseteq$ *C* $\neg$ *f A* $\subseteq$ *A*
show (*M* $\subseteq$ *A* $\cup$ *f A* $\wedge$ *A* $\cup$ *f A* $\subseteq$ *C*) $\wedge$ *card C* $-$ *card* (*A* $\cup$ *f A*) $<$ *card C* $-$ *card A*
(**is** *?L* $\wedge$ *?R*)
proof
show *?L* **by** (*metis assms*(*2*) *A*(*1*) *sup.coboundedI1 le-sup-iff*)
show *?R* **using** *A assms*(*2*,*3*) *card-seteq finite-subset*
by (*metis diff-less-mono2 finite-Un linorder-not-le sup-ge1 sup-ge2*)

qed
next
show $M \subseteq M \wedge M \subseteq C$ **using** ‹$M \subseteq C$› **by** *blast*
qed

corollary *while-saturate-finite-subset-Some*:
assumes *mono f* **and** $\bigwedge X.\ X \subseteq C \Longrightarrow f\ X \subseteq C$ **and** *finite C* **and** $M \subseteq C$
shows $\exists S.$ *while-saturate f M* = *Some S*
unfolding *while-saturate-def*
using *while-option-sat-finite-subset-Some assms* **by** *blast*

Hoare-like proof rules, for the whole set and for all elements of the set:

theorem *while-saturate-rule*:
assumes $\bigwedge S.\ P\ S \Longrightarrow P\ (S \cup F\ S)$
and *while-saturate F S* = *Some T*
and *P S*
shows *P T*
using *assms*[*unfolded while-saturate-def*] *while-option-rule*[*of P*]
by (*metis* (*lifting*))

theorem *while-saturate-rule1*:
assumes $\bigwedge S\ y.\ (\bigwedge x.\ x \in S \Longrightarrow P\ x) \Longrightarrow y \in F\ S \Longrightarrow P\ y$
and *while-saturate F S* = *Some T*
and $\bigwedge x.\ x \in S \Longrightarrow P\ x$
and $y \in T$
shows *P y*
using *while-saturate-rule*[*OF - assms*(*2*), *of* ‹$\lambda S.\ \forall s{\in}S.\ P\ s$›]
by (*metis Un-iff assms*(*1*,*3*,*4*))

lemma *while-saturate-incr*: *while-saturate F S0* = *Some S* $\Longrightarrow$ *S0* $\subseteq$ *S*
by (*simp add*: *le-supI1 while-saturate-rule*)

Correctness: finds the least saturated/closed set above M

lemma *while-option-sat-prefix*: **assumes** *mono f*
and *while-option* ($\lambda M.\ \neg\ f\ M \subseteq M$) ($\lambda M.\ M \cup f\ M$) *M* = *Some S*
and $M \subseteq P$ **and** $f\ P \subseteq P$
shows $S \subseteq P$
proof −
have $((\lambda M.\ M \cup f\ M)\ \widehat{\ }\widehat{\ }\ k)\ M \subseteq P$ **for** *k*
proof (*induction k*)
case *0* **thus** *?case* **using** ‹$M \subseteq P$› **by** *simp*
next
case (*Suc k*) **thus** *?case*
by *simp* (*meson* ‹*f P* $\subseteq$ *P*› *monoD*[*OF* ‹*mono f*›] *order.trans*)
qed
thus *?thesis* **by** (*metis assms*(*2*) *while-option-stop2*)
qed

corollary *while-saturate-prefix*:

⟦ *mono f*; *while-saturate f M = Some S*; *M ⊆ P*; *f P ⊆ P* ⟧ ⟹ *S ⊆ P*
using *while-option-sat-prefix* **unfolding** *while-saturate-def* **by** *blast*

4.6 Reflexive, transitive closure

Computing the reflexive, transitive closure by iterating a successor function. Stops when an element is found that dos not satisfy the test.

More refined (and hence more efficient) versions can be found in ITP 2011 paper by Nipkow (the theories are in the AFP entry Flyspeck by Nipkow) and the AFP article Executable Transitive Closures by René Thiemann.

context
fixes *p* :: *'a* ⇒ *bool*
and *f* :: *'a* ⇒ *'a list*
and *x* :: *'a*
begin

qualified fun *rtrancl-while-test* :: *'a list* × *'a set* ⇒ *bool*
where *rtrancl-while-test* (*ws*,-) = (*ws* ≠ [] ∧ *p*(*hd ws*))

qualified fun *rtrancl-while-step* :: *'a list* × *'a set* ⇒ *'a list* × *'a set*
where *rtrancl-while-step* (*ws*, *Z*) =
(*let x* = *hd ws*; *new* = *remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f x*))
in (*new* @ *tl ws*, *set new* ∪ *Z*))

definition *rtrancl-while* :: (*'a list* ∗ *'a set*) *option*
where *rtrancl-while* = *while-option rtrancl-while-test rtrancl-while-step* ([*x*],{*x*})

qualified fun *rtrancl-while-invariant* :: *'a list* × *'a set* ⇒ *bool*
where *rtrancl-while-invariant* (*ws*, *Z*) =
(*x* ∈ *Z* ∧ *set ws* ⊆ *Z* ∧ *distinct ws* ∧ {(*x*,*y*). *y* ∈ *set*(*f x*)} `` (*Z* − *set ws*) ⊆ *Z*
∧
Z ⊆ {(*x*,*y*). *y* ∈ *set*(*f x*)}∗ `` {*x*} ∧ (∀ *z*∈*Z* − *set ws*. *p z*))

qualified lemma *rtrancl-while-invariant*:
assumes *inv*: *rtrancl-while-invariant st* **and** *test*: *rtrancl-while-test st*
shows *rtrancl-while-invariant* (*rtrancl-while-step st*)
proof (*cases st*)
fix *ws Z*
assume *st*: *st* = (*ws*, *Z*)
with *test* **obtain** *h t* **where** *ws* = *h* # *t p h* **by** (*cases ws*) *auto*
with *inv st* **show** *?thesis* **by** (*auto intro*: *rtrancl.rtrancl-into-rtrancl*)
qed

lemma *rtrancl-while-Some*:
assumes *rtrancl-while* = *Some*(*ws*,*Z*)
shows *if ws* = []
then Z = {(*x*,*y*). *y* ∈ *set*(*f x*)}∗ `` {*x*} ∧ (∀ *z*∈*Z*. *p z*)
else ¬*p*(*hd ws*) ∧ *hd ws* ∈ {(*x*,*y*). *y* ∈ *set*(*f x*)}∗ `` {*x*}

proof −
have *rtrancl-while-invariant* ([*x*],{*x*}) **by** *simp*
with *rtrancl-while-invariant* **have** *I*: *rtrancl-while-invariant* (*ws*,*Z*)
by (*rule while-option-rule*[*OF - assms*[*unfolded rtrancl-while-def*]])
show *?thesis*
proof (*cases ws* = [])
case *True*
thus *?thesis* **using** *I*
by (*auto simp del*:*Image-Collect-case-prod dest*: *Image-closed-trancl*)
next
case *False*
thus *?thesis* **using** *I while-option-stop*[*OF assms*[*unfolded rtrancl-while-def*]]
by (*simp add*: *subset-iff*)
qed
qed

lemma *rtrancl-while-finite-Some*:
assumes *finite* ({(*x*, *y*). *y* ∈ *set* (*f x*)}* `` {*x*}) (**is** *finite ?Cl*)
shows ∃ *y*. *rtrancl-while* = *Some y*
proof −
let *?R* = (λ(-, *Z*). *card* (*?Cl* − *Z*)) <∗*mlex*∗> (λ(*ws*, -). *length ws*) <∗*mlex*∗>
{}
have *wf ?R* **by** (*blast intro*: *wf-mlex*)
then show *?thesis* **unfolding** *rtrancl-while-def*
proof (*rule wf-rel-while-option-Some*[*of ?R rtrancl-while-invariant*])
fix *st*
assume ∗: *rtrancl-while-invariant st* ∧ *rtrancl-while-test st*
hence *I*: *rtrancl-while-invariant* (*rtrancl-while-step st*)
by (*blast intro*: *rtrancl-while-invariant*)
show (*rtrancl-while-step st*, *st*) ∈ *?R*
proof (*cases st*)
fix *ws Z*
let *?ws* = *fst* (*rtrancl-while-step st*)
let *?Z* = *snd* (*rtrancl-while-step st*)
assume *st*: *st* = (*ws*, *Z*)
with ∗ **obtain** *h t* **where** *ws*: *ws* = *h* # *t p h* **by** (*cases ws*) *auto*
show *?thesis*
proof (*cases remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f h*)) = [])
case *False*
then obtain *z* **where** *z* ∈ *set* (*remdups* (*filter* (λ*y*. *y* ∉ *Z*) (*f h*))) **by** *fastforce*
with *st ws I* **have** *Z* ⊂ *?Z Z* ⊆ *?Cl ?Z* ⊆ *?Cl* **by** *auto*
with *assms* **have** *card* (*?Cl* − *?Z*) < *card* (*?Cl* − *Z*) **by** (*blast intro*: *psubset-card-mono*)
with *st ws* **show** *?thesis* **unfolding** *mlex-prod-def* **by** *simp*
next
case *True*
with *st ws* **have** *?Z* = *Z ?ws* = *t* **by** (*auto simp*: *filter-empty-conv*)
with *st ws* **show** *?thesis* **unfolding** *mlex-prod-def* **by** *simp*

qed
qed
qed (*simp-all add*: *rtrancl-while-invariant*)
qed

end

end

5 The Bourbaki-Witt tower construction for transfinite iteration

theory *Bourbaki-Witt-Fixpoint*
imports *While-Combinator*
begin

lemma *ChainsI* [*intro?*]:
$(\bigwedge a\ b.\ [\![\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow (a,\ b) \in r \vee (b,\ a) \in r) \Longrightarrow Y \in Chains\ r$
unfolding *Chains-def* **by** *blast*

lemma *in-Chains-subset*: $[\![\ M \in Chains\ r;\ M' \subseteq M\ ]\!] \Longrightarrow M' \in Chains\ r$
by(*auto simp add*: *Chains-def*)

lemma *in-ChainsD*: $[\![\ M \in Chains\ r;\ x \in M;\ y \in M\ ]\!] \Longrightarrow (x,\ y) \in r \vee (y,\ x) \in r$
unfolding *Chains-def* **by** *fast*

lemma *Chains-FieldD*: $[\![\ M \in Chains\ r;\ x \in M\ ]\!] \Longrightarrow x \in Field\ r$
by(*auto simp add*: *Chains-def intro*: *FieldI1 FieldI2*)

lemma *in-Chains-conv-chain*: $M \in Chains\ r \longleftrightarrow Complete\text{-}Partial\text{-}Order.chain\ (\lambda x\ y.\ (x,\ y) \in r)\ M$
by(*simp add*: *Chains-def chain-def*)

lemma *partial-order-on-trans*:
$[\![\ partial\text{-}order\text{-}on\ A\ r;\ (x,\ y) \in r;\ (y,\ z) \in r\ ]\!] \Longrightarrow (x,\ z) \in r$
by(*auto simp add*: *order-on-defs dest*: *transD*)

locale *bourbaki-witt-fixpoint* =
fixes *lub* :: $'a\ set \Rightarrow 'a$
and *leq* :: $('a \times 'a)\ set$
and *f* :: $'a \Rightarrow 'a$
assumes *po*: *Partial-order leq*
and *lub-least*: $[\![\ M \in Chains\ leq;\ M \neq \{\};\ \bigwedge x.\ x \in M \Longrightarrow (x,\ z) \in leq\ ]\!] \Longrightarrow (lub\ M,\ z) \in leq$
and *lub-upper*: $[\![\ M \in Chains\ leq;\ x \in M\ ]\!] \Longrightarrow (x,\ lub\ M) \in leq$
and *lub-in-Field*: $[\![\ M \in Chains\ leq;\ M \neq \{\}\ ]\!] \Longrightarrow lub\ M \in Field\ leq$
and *increasing*: $\bigwedge x.\ x \in Field\ leq \Longrightarrow (x,\ f\ x) \in leq$
begin

lemma *leq-trans*: ⟦ (*x*, *y*) ∈ *leq*; (*y*, *z*) ∈ *leq* ⟧ ⟹ (*x*, *z*) ∈ *leq*
by(*rule partial-order-on-trans*[*OF po*])

lemma *leq-refl*: *x* ∈ *Field leq* ⟹ (*x*, *x*) ∈ *leq*
using *po* **by**(*simp add*: *order-on-defs refl-on-def*)

lemma *leq-antisym*: ⟦ (*x*, *y*) ∈ *leq*; (*y*, *x*) ∈ *leq* ⟧ ⟹ *x* = *y*
using *po* **by**(*simp add*: *order-on-defs antisym-def*)

inductive-set *iterates-above* :: *'a* ⇒ *'a set*
  for *a*
where
  base: *a* ∈ *iterates-above a*
| *step*: *x* ∈ *iterates-above a* ⟹ *f x* ∈ *iterates-above a*
| *Sup*: ⟦ *M* ∈ *Chains leq*; *M* ≠ {}; ⋀*x*. *x* ∈ *M* ⟹ *x* ∈ *iterates-above a* ⟧ ⟹ *lub M* ∈ *iterates-above a*

definition *fixp-above* :: *'a* ⇒ *'a*
where *fixp-above a* = (*if a* ∈ *Field leq then lub* (*iterates-above a*) *else a*)

lemma *fixp-above-outside*: *a* ∉ *Field leq* ⟹ *fixp-above a* = *a*
by(*simp add*: *fixp-above-def*)

lemma *fixp-above-inside*: *a* ∈ *Field leq* ⟹ *fixp-above a* = *lub* (*iterates-above a*)
by(*simp add*: *fixp-above-def*)

context
  notes *leq-refl* [*intro*!, *simp*]
  and *base* [*intro*]
  and *step* [*intro*]
  and *Sup* [*intro*]
  and *leq-trans* [*trans*]
begin

lemma *iterates-above-le-f*: ⟦ *x* ∈ *iterates-above a*; *a* ∈ *Field leq* ⟧ ⟹ (*x*, *f x*) ∈ *leq*
by(*induction x rule*: *iterates-above.induct*)(*blast intro*: *increasing FieldI2 lub-in-Field*)+

lemma *iterates-above-Field*: ⟦ *x* ∈ *iterates-above a*; *a* ∈ *Field leq* ⟧ ⟹ *x* ∈ *Field leq*
by(*drule* (*1*) *iterates-above-le-f*)(*rule FieldI1*)

lemma *iterates-above-ge*:
  assumes *y*: *y* ∈ *iterates-above a*
  and *a*: *a* ∈ *Field leq*
  shows (*a*, *y*) ∈ *leq*
using *y* **by**(*induction*)(*auto intro*: *a increasing iterates-above-le-f leq-trans leq-trans*[*OF* - *lub-upper*])

lemma *iterates-above-lub*:
assumes *M*: $M \in Chains\ leq$
and *nempty*: $M \neq \{\}$
and *upper*: $\bigwedge y.\ y \in M \Longrightarrow \exists z \in M.\ (y, z) \in leq \wedge z \in iterates\text{-}above\ a$
shows $lub\ M \in iterates\text{-}above\ a$
proof −
let *?M* = $M \cap iterates\text{-}above\ a$
from *M* **have** *M'*: $?M \in Chains\ leq$ **by**(*rule in-Chains-subset*)*simp*
have $?M \neq \{\}$ **using** *nempty* **by**(*auto dest*: *upper*)
with *M'* **have** $lub\ ?M \in iterates\text{-}above\ a$ **by**(*rule Sup*) *blast*
also have $lub\ ?M = lub\ M$ **using** *nempty*
by(*intro leq-antisym*)(*blast intro*!: *lub-least*[*OF M*] *lub-least*[*OF M'*] *intro*: *lub-upper*[*OF M'*] *lub-upper*[*OF M*] *leq-trans dest*: *upper*)+
finally show *?thesis* **.**
qed

lemma *iterates-above-successor*:
assumes *y*: $y \in iterates\text{-}above\ a$
and *a*: $a \in Field\ leq$
shows $y = a \vee y \in iterates\text{-}above\ (f\ a)$
using *y*
proof *induction*
case *base* **thus** *?case* **by** *simp*
next
case (*step x*) **thus** *?case* **by** *auto*
next
case (*Sup M*)
show *?case*
proof(*cases* $\exists x.\ M \subseteq \{x\}$)
case *True*
with ‹$M \neq \{\}$› **obtain** *y* **where** *M*: $M = \{y\}$ **by** *auto*
have $lub\ M = y$
by(*rule leq-antisym*)(*auto intro*!: *lub-upper Sup lub-least ChainsI simp add*: *a M Sup.hyps*(*3*)[*of y*, *THEN iterates-above-Field*] *dest*: *iterates-above-Field*)
with *Sup.IH*[*of y*] *M* **show** *?thesis* **by** *simp*
next
case *False*
from *Sup*(*1*−*2*) **have** $lub\ M \in iterates\text{-}above\ (f\ a)$
proof(*rule iterates-above-lub*)
fix *y*
assume *y*: $y \in M$
from *Sup.IH*[*OF this*] **show** $\exists z{\in}M.\ (y, z) \in leq \wedge z \in iterates\text{-}above\ (f\ a)$
proof
assume $y = a$
from *y False* **obtain** *z* **where** *z*: $z \in M$ **and** *neq*: $y \neq z$ **by** (*metis insertI1 subsetI*)
with *Sup.IH*[*OF z*] ‹$y = a$› *Sup.hyps*(*3*)[*OF z*]
show *?thesis* **by**(*auto dest*: *iterates-above-ge intro*: *a*)
next

```
      assume ∗: y ∈ iterates-above (f a)
      with increasing[OF a] have y ∈ Field leq
        by(auto dest!: iterates-above-Field intro: FieldI2)
      with ∗ show ?thesis using y by auto
    qed
  qed
  thus ?thesis by simp
 qed
qed

lemma iterates-above-Sup-aux:
 assumes M: M ∈ Chains leq M ≠ {}
 and M': M' ∈ Chains leq M' ≠ {}
 and comp: ⋀x. x ∈ M ⟹ x ∈ iterates-above (lub M') ∨ lub M' ∈ iterates-above
x
 shows (lub M, lub M') ∈ leq ∨ lub M ∈ iterates-above (lub M')
proof(cases ∃ x ∈ M. x ∈ iterates-above (lub M'))
 case True
 then obtain x where x: x ∈ M x ∈ iterates-above (lub M') by blast
 have lub-M': lub M' ∈ Field leq using M' by(rule lub-in-Field)
 have lub M ∈ iterates-above (lub M') using M
 proof(rule iterates-above-lub)
  fix y
  assume y: y ∈ M
  from comp[OF y] show ∃ z∈M. (y, z) ∈ leq ∧ z ∈ iterates-above (lub M')
  proof
   assume y ∈ iterates-above (lub M')
   from this iterates-above-Field[OF this] y lub-M' show ?thesis by blast
  next
   assume lub M' ∈ iterates-above y
    hence (y, lub M') ∈ leq using Chains-FieldD[OF M(1) y] by(rule iter-
ates-above-ge)
   also have (lub M', x) ∈ leq using x(2) lub-M' by(rule iterates-above-ge)
   finally show ?thesis using x by blast
  qed
 qed
 thus ?thesis ..
next
 case False
 have (lub M, lub M') ∈ leq using M
 proof(rule lub-least)
  fix x
  assume x: x ∈ M
  from comp[OF x] x False have lub M' ∈ iterates-above x by auto
  moreover from M(1) x have x ∈ Field leq by(rule Chains-FieldD)
  ultimately show (x, lub M') ∈ leq by(rule iterates-above-ge)
 qed
 thus ?thesis ..
qed
```

lemma *iterates-above-triangle*:
assumes *x*: $x \in$ *iterates-above a*
and *y*: $y \in$ *iterates-above a*
and *a*: $a \in$ *Field leq*
shows $x \in$ *iterates-above y* $\vee$ $y \in$ *iterates-above x*
using *x y*
proof(*induction arbitrary*: *y*)
case *base* **then show** *?case* **by** *simp*
next
case (*step x*) **thus** *?case* **using** *a*
by(*auto dest*: *iterates-above-successor intro*: *iterates-above-Field*)
next
case *x*: (*Sup M*)
hence *lub*: *lub M* $\in$ *iterates-above a* **by** *blast*
from ‹*y* $\in$ *iterates-above a*› **show** *?case*
proof(*induction*)
case *base* **show** *?case* **using** *lub* **by** *simp*
next
case (*step y*) **thus** *?case* **using** *a*
by(*auto dest*: *iterates-above-successor intro*: *iterates-above-Field*)
next
case *y*: (*Sup M'*)
hence *lub'*: *lub M'* $\in$ *iterates-above a* **by** *blast*
have $*$: *x* $\in$ *iterates-above* (*lub M'*) $\vee$ *lub M'* $\in$ *iterates-above x* **if** $x \in M$ **for** *x*
using *that lub'* **by**(*rule x.IH*)
with *x*(*1*−*2*) *y*(*1*−*2*) **have** (*lub M*, *lub M'*) $\in$ *leq* $\vee$ *lub M* $\in$ *iterates-above* (*lub M'*)
by(*rule iterates-above-Sup-aux*)
moreover from *y*(*1*−*2*) *x*(*1*−*2*) **have** (*lub M'*, *lub M*) $\in$ *leq* $\vee$ *lub M'* $\in$ *iterates-above* (*lub M*)
by(*rule iterates-above-Sup-aux*)(*blast dest*: *y.IH*)
ultimately show *?case* **by**(*auto 4 3 dest*: *leq-antisym*)
qed
qed

lemma *chain-iterates-above*:
assumes *a*: $a \in$ *Field leq*
shows *iterates-above a* $\in$ *Chains leq* (**is** *?C* $\in$ -)
proof (*rule ChainsI*)
fix *x y*
assume *x* $\in$ *?C y* $\in$ *?C*
hence *x* $\in$ *iterates-above y* $\vee$ *y* $\in$ *iterates-above x* **using** *a* **by**(*rule iterates-above-triangle*)
moreover from ‹*x* $\in$ *?C*› *a* **have** *x* $\in$ *Field leq* **by**(*rule iterates-above-Field*)
moreover from ‹*y* $\in$ *?C*› *a* **have** *y* $\in$ *Field leq* **by**(*rule iterates-above-Field*)
ultimately show (*x*, *y*) $\in$ *leq* $\vee$ (*y*, *x*) $\in$ *leq* **by**(*auto dest*: *iterates-above-ge*)
qed

lemma *fixp-iterates-above*: *fixp-above a* $\in$ *iterates-above a*

by(*auto intro*: *chain-iterates-above simp add*: *fixp-above-def*)

lemma *fixp-above-Field*: *a* ∈ *Field leq* ⟹ *fixp-above a* ∈ *Field leq*
using *fixp-iterates-above* **by**(*rule iterates-above-Field*)

lemma *fixp-above-unfold*:
assumes *a*: *a* ∈ *Field leq*
shows *fixp-above a* = *f* (*fixp-above a*) (**is** *?a* = *f ?a*)
proof(*rule leq-antisym*)
show (*?a*, *f ?a*) ∈ *leq* **using** *fixp-above-Field*[*OF a*] **by**(*rule increasing*)

have *f ?a* ∈ *iterates-above a* **using** *fixp-iterates-above* **by**(*rule iterates-above.step*)
with *chain-iterates-above*[*OF a*] **show** (*f ?a*, *?a*) ∈ *leq*
by(*simp add*: *fixp-above-inside assms lub-upper*)
qed

end

lemma *fixp-above-induct* [*case-names adm base step*]:
assumes *adm*: *ccpo.admissible lub* (λ*x y*. (*x*, *y*) ∈ *leq*) *P*
and *base*: *P a*
and *step*: ⋀*x*. *P x* ⟹ *P* (*f x*)
shows *P* (*fixp-above a*)
proof(*cases a* ∈ *Field leq*)
case *True*
from *adm chain-iterates-above*[*OF True*]
show *?thesis* **unfolding** *fixp-above-inside*[*OF True*] *in-Chains-conv-chain*
proof(*rule ccpo.admissibleD*)
have *a* ∈ *iterates-above a* **..**
then show *iterates-above a* ≠ {} **by**(*auto*)
show *P x* **if** *x* ∈ *iterates-above a* **for** *x* **using** *that*
by *induction*(*auto intro*: *base step simp add*: *in-Chains-conv-chain dest*: *ccpo.admissibleD*[*OF adm*])
qed
qed(*simp add*: *fixp-above-outside base*)

end

5.1 Connect with the while combinator for executability on chain-finite lattices.

context *bourbaki-witt-fixpoint* **begin**

lemma *in-Chains-finite*: — Translation from ⟦*Complete-Partial-Order.chain* (≤) *?A*; *finite ?A*; *?A* ≠ {}⟧ ⟹ *Sup ?A* ∈ *?A*.
assumes *M* ∈ *Chains leq*
and *M* ≠ {}
and *finite M*
shows *lub M* ∈ *M*

```
using assms(3,1,2)
proof induction
  case empty thus ?case by simp
next
  case (insert x M)
  note chain = ‹insert x M ∈ Chains leq›
  show ?case
  proof(cases M = {})
    case True thus ?thesis
      using chain in-ChainsD leq-antisym lub-least lub-upper by fastforce
  next
    case False
    from chain have chain': M ∈ Chains leq
      using in-Chains-subset subset-insertI by blast
    hence lub M ∈ M using False by(rule insert.IH)
    show ?thesis
    proof(cases (x, lub M) ∈ leq)
      case True
      have (lub (insert x M), lub M) ∈ leq using chain
        by (rule lub-least) (auto simp: True intro: lub-upper[OF chain'])
      with False have lub (insert x M) = lub M
       using lub-upper[OF chain] lub-least[OF chain'] by (blast intro: leq-antisym)
      with ‹lub M ∈ M› show ?thesis by simp
    next
      case False
      with in-ChainsD[OF chain, of x lub M] ‹lub M ∈ M›
      have lub (insert x M) = x
        by − (rule leq-antisym, (blast intro: FieldI2 chain chain' insert.prems(2)
leq-refl leq-trans lub-least lub-upper)+)
      thus ?thesis by simp
    qed
  qed
qed

lemma fun-pow-iterates-above: (f ^^ k) a ∈ iterates-above a
using iterates-above.base iterates-above.step by (induct k) simp-all

lemma chfin-iterates-above-fun-pow:
  assumes x ∈ iterates-above a
  assumes ∀ M ∈ Chains leq. finite M
  shows ∃ j. x = (f ^^ j) a
using assms(1)
proof induct
  case base then show ?case by (simp add: exI[where x=0])
next
  case (step x) then obtain j where x = (f ^^ j) a by blast
  with step(1) show ?case by (simp add: exI[where x=Suc j])
next
  case (Sup M) with in-Chains-finite assms(2) show ?case by blast
```

qed

lemma *Chain-finite-iterates-above-fun-pow-iff*:
assumes $\forall M \in$ *Chains leq. finite M*
shows $x \in$ *iterates-above a* $\longleftrightarrow (\exists j.\ x = (f \frown\frown j)\ a)$
using *chfin-iterates-above-fun-pow fun-pow-iterates-above assms* **by** *blast*

lemma *fixp-above-Kleene-iter-ex*:
assumes $(\forall M \in$ *Chains leq. finite M*$)$
obtains *k* **where** *fixp-above a* $= (f \frown\frown k)\ a$
using *assms* **by** *atomize-elim* (*simp add*: *chfin-iterates-above-fun-pow fixp-iterates-above*)

context fixes *a* **assumes** *a*: $a \in$ *Field leq* **begin**

lemma *funpow-Field-leq*: $(f \frown\frown k)\ a \in$ *Field leq*
using *a* **by** (*induct k*) (*auto intro*: *increasing FieldI2*)

lemma *funpow-prefix*: $j < k \Longrightarrow ((f \frown\frown j)\ a, (f \frown\frown k)\ a) \in$ *leq*
proof(*induct k*)
case (*Suc k*)
with *leq-trans*[*OF - increasing*[*OF funpow-Field-leq*]] *funpow-Field-leq increasing a*
show *?case* **by** *simp* (*metis less-antisym*)
qed *simp*

lemma *funpow-suffix*: $(f \frown\frown$ *Suc k*$)\ a = (f \frown\frown k)\ a \Longrightarrow ((f \frown\frown (j + k))\ a, (f \frown\frown k)\ a) \in$ *leq*
using *funpow-Field-leq*
by (*induct j*) (*simp-all del*: *funpow.simps add*: *funpow-Suc-right funpow-add leq-refl*)

lemma *funpow-stability*: $(f \frown\frown$ *Suc k*$)\ a = (f \frown\frown k)\ a \Longrightarrow ((f \frown\frown j)\ a, (f \frown\frown k)\ a) \in$ *leq*
using *funpow-prefix funpow-suffix*[**where** *j*=$j - k$ **and** *k*=*k*] **by** (*cases* $j < k$) *simp-all*

lemma *funpow-in-Chains*: $\{(f \frown\frown k)\ a\ |k.\ True\} \in$ *Chains leq*
using *chain-iterates-above*[*OF a*] *fun-pow-iterates-above*
by (*blast intro*: *ChainsI dest*: *in-ChainsD*)

lemma *fixp-above-Kleene-iter*:
assumes $\forall M \in$ *Chains leq. finite M* — convenient but surely not necessary
assumes $(f \frown\frown$ *Suc k*$)\ a = (f \frown\frown k)\ a$
shows *fixp-above a* $= (f \frown\frown k)\ a$
proof(*rule leq-antisym*)
show (*fixp-above a*, $(f \frown\frown k)\ a) \in$ *leq* **using** *assms a*
by(*auto simp add*: *fixp-above-def chain-iterates-above Chain-finite-iterates-above-fun-pow-iff funpow-stability*[*OF assms*(*2*)] *intro*!: *lub-least intro*: *iterates-above.base*)
show $((f \frown\frown k)\ a$, *fixp-above a*$) \in$ *leq* **using** *a*
by(*auto simp add*: *fixp-above-def chain-iterates-above fun-pow-iterates-above*

intro!: *lub-upper*)
qed

context assumes *chfin*: $\forall M \in Chains\ leq.\ finite\ M$ **begin**

lemma *Chain-finite-wf*: *wf* $\{(f\ ((f\ ^{\frown\frown}\ k)\ a),\ (f\ ^{\frown\frown}\ k)\ a)\ |k.\ f\ ((f\ ^{\frown\frown}\ k)\ a) \neq (f\ ^{\frown\frown}\ k)\ a\}$
apply(*rule wf-measure*[**where** $f{=}\lambda b.\ card\ \{(f\ ^{\frown\frown}\ j)\ a\ |j.\ (b,\ (f\ ^{\frown\frown}\ j)\ a) \in leq\}$, *THEN wf-subset*])
apply(*auto simp*: *set-eq-iff intro*!: *psubset-card-mono*[*OF finite-subset*[*OF* - *bspec*[*OF chfin funpow-in-Chains*]]])
apply(*metis funpow-Field-leq increasing leq-antisym leq-trans leq-refl*)+
done

lemma *while-option-finite-increasing*: $\exists P.$ *while-option* $(\lambda A.\ f\ A \neq A)\ f\ a = Some\ P$
by(*rule wf-rel-while-option-Some*[*OF Chain-finite-wf*, **where** $P{=}\lambda A.\ (\exists k.\ A = (f\ ^{\frown\frown}\ k)\ a) \wedge (A,\ f\ A) \in leq$ **and** $s{=}a$])
(*auto simp*: *a increasing chfin FieldI2 chfin-iterates-above-fun-pow fun-pow-iterates-above iterates-above.step intro*: *exI*[**where** $x{=}0$])

lemma *fixp-above-the-while-option*: *fixp-above a* = *the* (*while-option* $(\lambda A.\ f\ A \neq A)\ f\ a$)
proof −
obtain *P* **where** *while-option* $(\lambda A.\ f\ A \neq A)\ f\ a = Some\ P$
using *while-option-finite-increasing* **by** *blast*
with *while-option-stop2*[*OF this*] *fixp-above-Kleene-iter*[*OF chfin*]
show *?thesis* **by** *fastforce*
qed

lemma *fixp-above-conv-while*: *fixp-above a* = *while* $(\lambda A.\ f\ A \neq A)\ f\ a$
unfolding *while-def* **by** (*rule fixp-above-the-while-option*)

end

end

end

lemma *bourbaki-witt-fixpoint-complete-latticeI*:
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
assumes $\bigwedge x.\ x \leq f\ x$
shows *bourbaki-witt-fixpoint Sup* $\{(x,\ y).\ x \leq y\}\ f$
by *unfold-locales* (*auto simp*: *assms Sup-upper order-on-defs Field-def intro*: *refl-onI transI antisymI Sup-least*)

end

6 Division with modulus centered towards zero.

theory *Centered-Division*
imports *Main*
begin

lemma *off-iff-abs-mod-2-eq-one*:
‹*odd l* ⟷ |*l*| *mod* 2 = 1› **for** *l* :: *int*
by (*simp flip*: *odd-iff-mod-2-eq-one*)

The following specification of division on integers centers the modulus around zero. This is useful e.g. to define division on Gauss numbers. N.b.: This is not mentioned [2].

definition *centered-divide* :: ‹*int* ⇒ *int* ⇒ *int›* (**infixl** ‹*cdiv*› 70)
where ‹*k cdiv l* = *sgn l* ∗ ((*k* + |*l*| *div* 2) *div* |*l*|)›

definition *centered-modulo* :: ‹*int* ⇒ *int* ⇒ *int›* (**infixl** ‹*cmod*› 70)
where ‹*k cmod l* = (*k* + |*l*| *div* 2) *mod* |*l*| − |*l*| *div* 2›

Example: *k cmod* 5 ∈ {− 2, − 1, 0, 1, 2}

lemma *signed-take-bit-eq-cmod*:
‹*signed-take-bit n k* = *k cmod* (2 ^ *Suc n*)›
by (*simp only*: *centered-modulo-def power-abs abs-numeral flip*: *take-bit-eq-mod*)
(*simp add*: *signed-take-bit-eq-take-bit-shift*)

Property *signed-take-bit n k* = *k cmod* $2^{Suc\ n}$ is the key to generalize centered division to arbitrary structures satisfying *ring-bit-operations*, but so far it is not clear what practical relevance that would have.

lemma *cdiv-mult-cmod-eq*:
‹*k cdiv l* ∗ *l* + *k cmod l* = *k*›
proof −
have ∗: ‹*l* ∗ (*sgn l* ∗ *j*) = |*l*| ∗ *j*› **for** *j*
by (*simp add*: *ac-simps abs-sgn*)
show *?thesis*
by (*simp add*: *centered-divide-def centered-modulo-def algebra-simps* ∗)
qed

lemma *mult-cdiv-cmod-eq*:
‹*l* ∗ (*k cdiv l*) + *k cmod l* = *k*›
using *cdiv-mult-cmod-eq* [*of k l*] **by** (*simp add*: *ac-simps*)

lemma *cmod-cdiv-mult-eq*:
‹*k cmod l* + *k cdiv l* ∗ *l* = *k*›
using *cdiv-mult-cmod-eq* [*of k l*] **by** (*simp add*: *ac-simps*)

lemma *cmod-mult-cdiv-eq*:
‹*k cmod l* + *l* ∗ (*k cdiv l*) = *k*›
using *cdiv-mult-cmod-eq* [*of k l*] **by** (*simp add*: *ac-simps*)

lemma *minus-cdiv-mult-eq-cmod*:
‹*k − k cdiv l ∗ l = k cmod l*›
by (*rule add-implies-diff* [*symmetric*]) (*fact cmod-cdiv-mult-eq*)

lemma *minus-mult-cdiv-eq-cmod*:
‹*k − l ∗ (k cdiv l) = k cmod l*›
by (*rule add-implies-diff* [*symmetric*]) (*fact cmod-mult-cdiv-eq*)

lemma *minus-cmod-eq-cdiv-mult*:
‹*k − k cmod l = k cdiv l ∗ l*›
by (*rule add-implies-diff* [*symmetric*]) (*fact cdiv-mult-cmod-eq*)

lemma *minus-cmod-eq-mult-cdiv*:
‹*k − k cmod l = l ∗ (k cdiv l)*›
by (*rule add-implies-diff* [*symmetric*]) (*fact mult-cdiv-cmod-eq*)

lemma *cdiv-0-eq* [*simp*]:
‹*k cdiv 0 = 0*›
by (*simp add*: *centered-divide-def*)

lemma *cmod-0-eq* [*simp*]:
‹*k cmod 0 = k*›
by (*simp add*: *centered-modulo-def*)

lemma *cdiv-1-eq* [*simp*]:
‹*k cdiv 1 = k*›
by (*simp add*: *centered-divide-def*)

lemma *cmod-1-eq* [*simp*]:
‹*k cmod 1 = 0*›
by (*simp add*: *centered-modulo-def*)

lemma *zero-cdiv-eq* [*simp*]:
‹*0 cdiv k = 0*›
by (*auto simp add*: *centered-divide-def not-less zdiv-eq-0-iff*)

lemma *zero-cmod-eq* [*simp*]:
‹*0 cmod k = 0*›
by (*auto simp add*: *centered-modulo-def not-less zmod-trivial-iff*)

lemma *cdiv-minus-eq*:
‹*k cdiv − l = − (k cdiv l)*›
by (*simp add*: *centered-divide-def*)

lemma *cmod-minus-eq* [*simp*]:
‹*k cmod − l = k cmod l*›
by (*simp add*: *centered-modulo-def*)

lemma *cdiv-abs-eq*:

```
  ‹k cdiv |l| = sgn l * (k cdiv l)›
  by (simp add: centered-divide-def)

lemma cmod-abs-eq [simp]:
  ‹k cmod |l| = k cmod l›
  by (simp add: centered-modulo-def)

lemma nonzero-mult-cdiv-cancel-right:
  ‹k * l cdiv l = k› if ‹l ≠ 0›
proof −
  have ‹sgn l * k * |l| cdiv l = k›
    using that by (simp add: centered-divide-def)
  with that show ?thesis
    by (simp add: ac-simps abs-sgn)
qed

lemma cdiv-self-eq [simp]:
  ‹k cdiv k = 1› if ‹k ≠ 0›
  using that nonzero-mult-cdiv-cancel-right [of k 1] by simp

lemma cmod-self-eq [simp]:
  ‹k cmod k = 0›
proof −
  have ‹(sgn k * |k| + |k| div 2) mod |k| = |k| div 2›
    by (auto simp add: zmod-trivial-iff)
  also have ‹sgn k * |k| = k›
    by (simp add: abs-sgn)
  finally show ?thesis
    by (simp add: centered-modulo-def algebra-simps)
qed

lemma cmod-less-divisor:
  ‹k cmod l < |l| − |l| div 2› if ‹l ≠ 0›
  using that pos-mod-bound [of ‹|l|›] by (simp add: centered-modulo-def)

lemma cmod-less-equal-divisor:
  ‹k cmod l ≤ |l| div 2› if ‹l ≠ 0›
proof −
  from that cmod-less-divisor [of l k]
  have ‹k cmod l < |l| − |l| div 2›
    by simp
  also have ‹|l| − |l| div 2 = |l| div 2 + of-bool (odd l)›
    by auto
  finally show ?thesis
    by (cases ‹even l›) simp-all
qed

lemma divisor-less-equal-cmod′:
  ‹|l| div 2 − |l| ≤ k cmod l› if ‹l ≠ 0›
```

proof −
have ‹*0* ≤ (*k* + |*l*| *div* *2*) *mod* |*l*|›
using *that pos-mod-sign* [*of* ‹|*l*|›] **by** *simp*
then show *?thesis*
by (*simp-all add*: *centered-modulo-def*)
qed

lemma *divisor-less-equal-cmod*:
‹− (|*l*| *div* *2*) ≤ *k* *cmod* *l*› **if** ‹*l* ≠ *0*›
using *that divisor-less-equal-cmod'* [*of* *l* *k*]
by (*simp add*: *centered-modulo-def*)

lemma *abs-cmod-less-equal*:
‹|*k* *cmod* *l*| ≤ |*l*| *div* *2*› **if** ‹*l* ≠ *0*›
using *that divisor-less-equal-cmod* [*of* *l* *k*]
by (*simp add*: *abs-le-iff cmod-less-equal-divisor*)

end

7 Order on characters

theory *Char-ord*
imports *Main*
begin

instantiation *char* :: *linorder*
begin

definition *less-eq-char* :: ‹*char* ⇒ *char* ⇒ *bool*›
where ‹*c1* ≤ *c2* ⟷ *of-char* *c1* ≤ (*of-char* *c2* :: *nat*)›

definition *less-char* :: ‹*char* ⇒ *char* ⇒ *bool*›
where ‹*c1* < *c2* ⟷ *of-char* *c1* < (*of-char* *c2* :: *nat*)›

instance
by *standard* (*auto simp add*: *less-eq-char-def less-char-def*)

end

lemma *less-eq-char-simp* [*simp*, *code*]:
‹*Char b0 b1 b2 b3 b4 b5 b6 b7* ≤ *Char c0 c1 c2 c3 c4 c5 c6 c7*
⟷ *lexordp-eq* [*b7*, *b6*, *b5*, *b4*, *b3*, *b2*, *b1*, *b0*] [*c7*, *c6*, *c5*, *c4*, *c3*, *c2*, *c1*, *c0*]›
by (*simp only*: *less-eq-char-def of-char-def char.sel horner-sum-less-eq-iff-lexordp-eq list.size*) *simp*

lemma *less-char-simp* [*simp*, *code*]:
‹*Char b0 b1 b2 b3 b4 b5 b6 b7* < *Char c0 c1 c2 c3 c4 c5 c6 c7*
⟷ *ord-class.lexordp* [*b7*, *b6*, *b5*, *b4*, *b3*, *b2*, *b1*, *b0*] [*c7*, *c6*, *c5*, *c4*, *c3*, *c2*,

c1, *c0*]›
by (*simp only*: *less-char-def of-char-def char.sel horner-sum-less-iff-lexordp list.size*) *simp*

instantiation *char* :: *distrib-lattice*
begin

definition ‹(*inf* :: *char* ⇒ -) = *min*›
definition ‹(*sup* :: *char* ⇒ -) = *max*›

instance
by *standard* (*auto simp add*: *inf-char-def sup-char-def max-min-distrib2*)

end

code-identifier
code-module *Char-ord* ⇀
(*SML*) *Str* **and** (*OCaml*) *Str* **and** (*Haskell*) *Str* **and** (*Scala*) *Str*

end

8 A generic phantom type

theory *Phantom-Type*
imports *Main*
begin

datatype ('*a*, '*b*) *phantom* = *phantom* (*of-phantom*: '*b*)

lemma *type-definition-phantom'*: *type-definition of-phantom phantom UNIV*
by(*unfold-locales*) *simp-all*

lemma *phantom-comp-of-phantom* [*simp*]: *phantom* ∘ *of-phantom* = *id*
and *of-phantom-comp-phantom* [*simp*]: *of-phantom* ∘ *phantom* = *id*
by(*simp-all add*: *o-def id-def*)

syntax -*Phantom* :: *type* ⇒ *logic* (‹(‹*indent=1 notation=‹mixfix Phantom››Phantom*/(*1* '(-')))›)
syntax-consts -*Phantom* == *phantom*
translations
Phantom('*t*) => *CONST phantom* :: - ⇒ ('*t*, -) *phantom*

typed-print-translation ‹
let
fun phantom-tr' ctxt (*Type* (***type-name*** ‹*fun*›, [-, *Type* (***type-name*** ‹*phantom*›,
[*T*, -])])) *ts* =
list-comb
(*Syntax.const* ***syntax-const*** ‹-*Phantom*› $ *Syntax-Phases.term-of-typ ctxt*
T, *ts*)
| *phantom-tr'* - - - = *raise Match*;

in [(***const-syntax*** ‹*phantom*›, *phantom-tr'*)] *end*
›

lemma *of-phantom-inject* [*simp*]:
of-phantom x = *of-phantom y* ⟷ *x* = *y*
by(*cases x y rule*: *phantom.exhaust*[*case-product phantom.exhaust*]) *simp*

end

9 Cardinality of types

theory *Cardinality*
imports *Phantom-Type*
begin

9.1 Cardinalities of types

syntax *-type-card* :: *type* => *nat* (‹(‹*indent=1 notation=‹mixfix CARD›› CARD/(1'(-'))*)›)

syntax-consts *-type-card* == *card*

translations *CARD('t)* => *CONST card* (*CONST UNIV* :: *'t set*)

print-translation ‹
let
fun card-univ-tr' ctxt [*Const* (***const-syntax*** ‹*UNIV*›, *Type* (-, [*T*]))] =
Syntax.const ***syntax-const*** ‹*-type-card*› $ *Syntax-Phases.term-of-typ ctxt T*
in [(***const-syntax*** ‹*card*›, *card-univ-tr'*)] *end*
›

lemma *card-prod* [*simp*]: *CARD('a* × *'b)* = *CARD('a)* ∗ *CARD('b)*
unfolding *UNIV-Times-UNIV* [*symmetric*] **by** (*simp only*: *card-cartesian-product*)

lemma *card-UNIV-sum*: *CARD('a* + *'b)* = (*if CARD('a)* ≠ *0* ∧ *CARD('b)* ≠ *0*
then CARD('a) + *CARD('b) else 0*)
unfolding *UNIV-Plus-UNIV*[*symmetric*]
by(*auto simp add*: *card-eq-0-iff card-Plus simp del*: *UNIV-Plus-UNIV*)

lemma *card-sum* [*simp*]: *CARD('a* + *'b)* = *CARD('a*::*finite)* + *CARD('b*::*finite)*
by(*simp add*: *card-UNIV-sum*)

lemma *card-UNIV-option*: *CARD('a option)* = (*if CARD('a)* = *0 then 0 else*
CARD('a) + *1*)
proof −
have (*None* :: *'a option*) ∉ *range Some* **by** *clarsimp*
thus *?thesis*
by (*simp add*: *UNIV-option-conv card-eq-0-iff finite-range-Some card-image*)
qed

lemma *card-option* [*simp*]: $CARD('a\ option) = Suc\ CARD('a::finite)$
by(*simp add*: *card-UNIV-option*)

lemma *card-UNIV-set*: $CARD('a\ set) = (if\ CARD('a) = 0\ then\ 0\ else\ 2\ ^\frown CARD('a))$
by(*simp add*: *card-eq-0-iff card-Pow flip*: *Pow-UNIV*)

lemma *card-set* [*simp*]: $CARD('a\ set) = 2\ ^\frown CARD('a::finite)$
by(*simp add*: *card-UNIV-set*)

lemma *card-nat* [*simp*]: $CARD(nat) = 0$
by (*simp add*: *card-eq-0-iff*)

lemma *card-fun*: $CARD('a \Rightarrow 'b) = (if\ CARD('a) \neq 0 \wedge CARD('b) \neq 0 \vee CARD('b) = 1\ then\ CARD('b)\ ^\frown CARD('a)\ else\ 0)$
proof −
have $CARD('a \Rightarrow 'b) = CARD('b)\ ^\frown CARD('a)$ **if** $0 < CARD('a)$ **and** $0 < CARD('b)$
proof −
from *that* **have** *fina*: *finite* (*UNIV* :: $'a\ set$) **and** *finb*: *finite* (*UNIV* :: $'b\ set$)
by(*simp-all only*: *card-ge-0-finite*)
from *finite-distinct-list*[*OF finb*] **obtain** *bs*
where *bs*: $set\ bs = (UNIV :: 'b\ set)$ **and** *distb*: *distinct bs* **by** *blast*
from *finite-distinct-list*[*OF fina*] **obtain** *as*
where *as*: $set\ as = (UNIV :: 'a\ set)$ **and** *dista*: *distinct as* **by** *blast*
have *cb*: $CARD('b) = length\ bs$
unfolding *bs*[*symmetric*] *distinct-card*[*OF distb*] **..**
have *ca*: $CARD('a) = length\ as$
unfolding *as*[*symmetric*] *distinct-card*[*OF dista*] **..**
let *?xs* = $map\ (\lambda ys.\ the \circ map\text{-}of\ (zip\ as\ ys))\ (List.n\text{-}lists\ (length\ as)\ bs)$
have $UNIV = set\ ?xs$
proof(*rule UNIV-eq-I*)
fix $f :: 'a \Rightarrow 'b$
from *as* **have** $f = the \circ map\text{-}of\ (zip\ as\ (map\ f\ as))$
by(*auto simp add*: *map-of-zip-map*)
thus $f \in set\ ?xs$ **using** *bs* **by**(*auto simp add*: *set-n-lists*)
qed
moreover have *distinct ?xs* **unfolding** *distinct-map*
proof(*intro conjI distinct-n-lists distb inj-onI*)
fix *xs ys* :: *'b list*
assume *xs*: $xs \in set\ (List.n\text{-}lists\ (length\ as)\ bs)$
and *ys*: $ys \in set\ (List.n\text{-}lists\ (length\ as)\ bs)$
and *eq*: $the \circ map\text{-}of\ (zip\ as\ xs) = the \circ map\text{-}of\ (zip\ as\ ys)$
from *xs ys* **have** [*simp*]: $length\ xs = length\ as\ length\ ys = length\ as$
by(*simp-all add*: *length-n-lists-elem*)
have $map\text{-}of\ (zip\ as\ xs) = map\text{-}of\ (zip\ as\ ys)$
proof
fix *x*
from *as bs* **have** $\exists y.\ map\text{-}of\ (zip\ as\ xs)\ x = Some\ y\ \exists y.\ map\text{-}of\ (zip\ as\ ys)\ x = Some\ y$

```
        by(simp-all add: map-of-zip-is-Some[symmetric])
      with eq show map-of (zip as xs) x = map-of (zip as ys) x
        by(auto dest: fun-cong[where x=x])
    qed
    with dista show xs = ys by(simp add: map-of-zip-inject)
   qed
   hence card (set ?xs) = length ?xs by(simp only: distinct-card)
   moreover have length ?xs = length bs ⌢ length as by(simp add: length-n-lists)
   ultimately show ?thesis using cb ca by simp
  qed
  moreover have CARD('a ⇒ 'b) = 1 if CARD('b) = 1
  proof −
   from that obtain b where b: UNIV = {b :: 'b} by(auto simp add: card-Suc-eq)
    have eq: UNIV = {λx :: 'a. b ::'b}
    proof(rule UNIV-eq-I)
     fix x :: 'a ⇒ 'b
     have x y = b for y
     proof −
      have x y ∈ UNIV ..
      thus ?thesis unfolding b by simp
     qed
     thus x ∈ {λx. b} by(auto)
    qed
    show ?thesis unfolding eq by simp
  qed
  ultimately show ?thesis
   by(auto simp del: One-nat-def)(auto simp add: card-eq-0-iff dest: finite-fun-UNIVD2
finite-fun-UNIVD1)
qed

corollary finite-UNIV-fun:
  finite (UNIV :: ('a ⇒ 'b) set) ⟷
   finite (UNIV :: 'a set) ∧ finite (UNIV :: 'b set) ∨ CARD('b) = 1
  (is ?lhs ⟷ ?rhs)
proof −
  have ?lhs ⟷ CARD('a ⇒ 'b) > 0 by(simp add: card-gt-0-iff)
  also have ... ⟷ CARD('a) > 0 ∧ CARD('b) > 0 ∨ CARD('b) = 1
    by(simp add: card-fun)
  also have ... = ?rhs by(simp add: card-gt-0-iff)
  finally show ?thesis .
qed

lemma card-literal: CARD(String.literal) = 0
by(simp add: card-eq-0-iff infinite-literal)
```

9.2 Classes with at least 1 and 2

Class finite already captures "at least 1"

```
lemma zero-less-card-finite [simp]: 0 < CARD('a::finite)
```

unfolding *neq0-conv* [*symmetric*] **by** *simp*

lemma *one-le-card-finite* [*simp*]: *Suc 0* ≤ *CARD*('*a*::*finite*)
by (*simp add*: *less-Suc-eq-le* [*symmetric*])

class *CARD-1* =
assumes *CARD-1*: *CARD* ('*a*) = *1*
begin

subclass *finite*
proof
from *CARD-1* **show** *finite* (*UNIV* :: '*a set*)
using *finite-UNIV-fun* **by** *fastforce*
qed

end

Class for cardinality "at least 2"

class *card2* = *finite* +
assumes *two-le-card*: *2* ≤ *CARD*('*a*)

lemma *one-less-card*: *Suc 0* < *CARD*('*a*::*card2*)
using *two-le-card* [**where** '*a*='*a*] **by** *simp*

lemma *one-less-int-card*: *1* < *int CARD*('*a*::*card2*)
using *one-less-card* [**where** '*a*='*a*] **by** *simp*

9.3 A type class for deciding finiteness of types

type-synonym '*a finite-UNIV* = ('*a*, *bool*) *phantom*

class *finite-UNIV* =
fixes *finite-UNIV* :: ('*a*, *bool*) *phantom*
assumes *finite-UNIV*: *finite-UNIV* = *Phantom*('*a*) (*finite* (*UNIV* :: '*a set*))

lemma *finite-UNIV-code* [*code-unfold*]:
finite (*UNIV* :: '*a* :: *finite-UNIV set*)
⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)
by(*simp add*: *finite-UNIV*)

9.4 A type class for computing the cardinality of types

definition *is-list-UNIV* :: '*a list* ⇒ *bool*
where *is-list-UNIV xs* = (*let c* = *CARD*('*a*) *in if c* = *0 then False else size* (*remdups xs*) = *c*)

lemma *is-list-UNIV-iff*: *is-list-UNIV xs* ⟷ *set xs* = *UNIV*
by(*auto simp add*: *is-list-UNIV-def Let-def card-eq-0-iff List.card-set*[*symmetric*]
dest: *subst*[**where** *P*=*finite*, *OF - finite-set*] *card-eq-UNIV-imp-eq-UNIV*)

type-synonym *'a card-UNIV = ('a, nat) phantom*

class *card-UNIV = finite-UNIV +*
  fixes *card-UNIV :: 'a card-UNIV*
  assumes *card-UNIV: card-UNIV = Phantom('a) CARD('a)*

9.5 Instantiations for *card-UNIV*

instantiation *nat :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(nat) False*
definition *card-UNIV = Phantom(nat) 0*
instance by *intro-classes (simp-all add: finite-UNIV-nat-def card-UNIV-nat-def)*
end

instantiation *int :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(int) False*
definition *card-UNIV = Phantom(int) 0*
instance by *intro-classes (simp-all add: card-UNIV-int-def finite-UNIV-int-def)*
end

instantiation *natural :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(natural) False*
definition *card-UNIV = Phantom(natural) 0*
instance
proof
  show *finite-UNIV = Phantom(natural) (finite (UNIV::natural set))*
    by *(simp add: finite-UNIV-natural-def infinite-UNIV-char-0)*
  then show *card-UNIV-class.card-UNIV = Phantom(natural) CARD(natural)*
    by *(metis card-UNIV-natural-def card-eq-0-iff finite-UNIV-natural-def phantom.inject)*
qed
end

instantiation *integer :: card-UNIV* **begin**
definition *finite-UNIV = Phantom(integer) False*
definition *card-UNIV = Phantom(integer) 0*
instance
proof
  show *finite-UNIV = Phantom(integer) (finite (UNIV::integer set))*
    by *(simp add: finite-UNIV-integer-def infinite-UNIV-char-0)*
  then show *card-UNIV-class.card-UNIV = Phantom(integer) CARD(integer)*
    by *(metis card-UNIV-integer-def card-eq-0-iff finite-UNIV-integer-def phantom.inject)*
qed
end

instantiation *list :: (type) card-UNIV* **begin**
definition *finite-UNIV = Phantom('a list) False*

definition *card-UNIV* = *Phantom*('*a list*) *0*
instance by *intro-classes* (*simp-all add*: *card-UNIV-list-def finite-UNIV-list-def infinite-UNIV-listI*)
end

instantiation *unit* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*unit*) *True*
definition *card-UNIV* = *Phantom*(*unit*) *1*
instance by *intro-classes* (*simp-all add*: *card-UNIV-unit-def finite-UNIV-unit-def*)
end

instantiation *bool* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*bool*) *True*
definition *card-UNIV* = *Phantom*(*bool*) *2*
instance by(*intro-classes*)(*simp-all add*: *card-UNIV-bool-def finite-UNIV-bool-def*)
end

instantiation *char* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*char*) *True*
definition *card-UNIV* = *Phantom*(*char*) *256*
instance by *intro-classes* (*simp-all add*: *card-UNIV-char-def card-UNIV-char finite-UNIV-char-def*)
end

instantiation *prod* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: '*b finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-prod-def finite-UNIV finite-prod*)
end

instantiation *prod* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* × '*b*)
(*of-phantom* (*card-UNIV* :: '*a card-UNIV*) ∗ *of-phantom* (*card-UNIV* :: '*b card-UNIV*))
instance by *intro-classes* (*simp add*: *card-UNIV-prod-def card-UNIV*)
end

instantiation *sum* :: (*finite-UNIV*, *finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* + '*b*)
(*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *of-phantom* (*finite-UNIV* :: '*b finite-UNIV*))
instance
by *intro-classes* (*simp add*: *finite-UNIV-sum-def finite-UNIV*)
end

instantiation *sum* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* + '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)

in if ca ≠ 0 ∧ cb ≠ 0 then ca + cb else 0)
instance by *intro-classes* (*auto simp add*: *card-UNIV-sum-def card-UNIV card-UNIV-sum*)
end

instantiation *fun* :: (*finite-UNIV*, *card-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let cb* = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in cb = *1* ∨ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*) ∧ *cb* ≠ *0*)
instance
by *intro-classes* (*auto simp add*: *finite-UNIV-fun-def Let-def card-UNIV finite-UNIV finite-UNIV-fun card-gt-0-iff*)
end

instantiation *fun* :: (*card-UNIV*, *card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a* ⇒ '*b*)
(*let ca* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*);
cb = *of-phantom* (*card-UNIV* :: '*b card-UNIV*)
in if ca ≠ *0* ∧ *cb* ≠ *0* ∨ *cb* = *1 then cb* ^ *ca else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-fun-def card-UNIV Let-def card-fun*)
end

instantiation *option* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a option*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-option-def finite-UNIV*)
end

instantiation *option* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a option*)
(*let c* = *of-phantom* (*card-UNIV* :: '*a card-UNIV*) *in if c* ≠ *0 then Suc c else 0*)
instance by *intro-classes* (*simp add*: *card-UNIV-option-def card-UNIV card-UNIV-option*)
end

instantiation *String.literal* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*String.literal*) *False*
definition *card-UNIV* = *Phantom*(*String.literal*) *0*
instance
by *intro-classes* (*simp-all add*: *card-UNIV-literal-def finite-UNIV-literal-def infinite-literal card-literal*)
end

instantiation *set* :: (*finite-UNIV*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a set*) (*of-phantom* (*finite-UNIV* :: '*a finite-UNIV*))
instance by *intro-classes* (*simp add*: *finite-UNIV-set-def finite-UNIV Finite-Set.finite-set*)
end

instantiation *set* :: (*card-UNIV*) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a set*)

(*let c = of-phantom (card-UNIV :: 'a card-UNIV) in if c = 0 then 0 else 2 ^ c*)
instance by *intro-classes* (*simp add*: *card-UNIV-set-def card-UNIV-set card-UNIV*)
end

lemma *UNIV-finite-1*: *UNIV = set* [*finite-1*.a_1]
by(*auto intro*: *finite-1.exhaust*)

lemma *UNIV-finite-2*: *UNIV = set* [*finite-2*.a_1, *finite-2*.a_2]
by(*auto intro*: *finite-2.exhaust*)

lemma *UNIV-finite-3*: *UNIV = set* [*finite-3*.a_1, *finite-3*.a_2, *finite-3*.a_3]
by(*auto intro*: *finite-3.exhaust*)

lemma *UNIV-finite-4*: *UNIV = set* [*finite-4*.a_1, *finite-4*.a_2, *finite-4*.a_3, *finite-4*.a_4]
by(*auto intro*: *finite-4.exhaust*)

lemma *UNIV-finite-5*:
UNIV = set [*finite-5*.a_1, *finite-5*.a_2, *finite-5*.a_3, *finite-5*.a_4, *finite-5*.a_5]
by(*auto intro*: *finite-5.exhaust*)

instantiation *Enum.finite-1* :: *card-UNIV* **begin**
definition *finite-UNIV = Phantom*(*Enum.finite-1*) *True*
definition *card-UNIV = Phantom*(*Enum.finite-1*) *1*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-1 card-UNIV-finite-1-def finite-UNIV-finite-1-def*)
end

instantiation *Enum.finite-2* :: *card-UNIV* **begin**
definition *finite-UNIV = Phantom*(*Enum.finite-2*) *True*
definition *card-UNIV = Phantom*(*Enum.finite-2*) *2*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-2 card-UNIV-finite-2-def finite-UNIV-finite-2-def*)
end

instantiation *Enum.finite-3* :: *card-UNIV* **begin**
definition *finite-UNIV = Phantom*(*Enum.finite-3*) *True*
definition *card-UNIV = Phantom*(*Enum.finite-3*) *3*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-3 card-UNIV-finite-3-def finite-UNIV-finite-3-def*)
end

instantiation *Enum.finite-4* :: *card-UNIV* **begin**
definition *finite-UNIV = Phantom*(*Enum.finite-4*) *True*
definition *card-UNIV = Phantom*(*Enum.finite-4*) *4*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-4 card-UNIV-finite-4-def finite-UNIV-finite-4-def*)
end

instantiation *Enum.finite-5* :: *card-UNIV* **begin**

definition *finite-UNIV* = *Phantom*(*Enum.finite-5*) *True*
definition *card-UNIV* = *Phantom*(*Enum.finite-5*) *5*
instance
by *intro-classes* (*simp-all add*: *UNIV-finite-5 card-UNIV-finite-5-def finite-UNIV-finite-5-def*)
end

end

10 Code setup for sets with cardinality type information

theory *Code-Cardinality* **imports** *Cardinality* **begin**

Implement *CARD*('*a*) via *card-UNIV-class.card-UNIV* and provide implementations for *finite*, *card*, (⊆), and (=)if the calling context already provides *finite-UNIV* and *card-UNIV* instances. If we implemented the latter always via *card-UNIV-class.card-UNIV*, we would require instances of essentially all element types, i.e., a lot of instantiation proofs and – at run time – possibly slow dictionary constructions.

context
begin

qualified definition *card-UNIV'* :: '*a card-UNIV*
where *card-UNIV'* = *Phantom*('*a*) *CARD*('*a*)

lemma *CARD-code* [*code-unfold*]:
CARD('*a*) = *of-phantom* (*card-UNIV'* :: '*a card-UNIV*)
by(*simp add*: *card-UNIV'-def*)

lemma *card-UNIV'-code* [*code*]:
card-UNIV' = *card-UNIV*
by(*simp add*: *card-UNIV card-UNIV'-def*)

end

lemma *card-Compl*:
finite A ⟹ *card* (− *A*) = *card* (*UNIV* :: '*a set*) − *card* (*A* :: '*a set*)
by (*metis Compl-eq-Diff-UNIV card-Diff-subset top-greatest*)

context fixes *xs* :: '*a* :: *finite-UNIV list*
begin

qualified definition *finite'* :: '*a set* ⇒ *bool*
where [*simp*, *code-abbrev*]: *finite'* = *finite*

lemma *finite'-code* [*code*]:
finite' (*set xs*) ⟷ *True*
finite' (*List.coset xs*) ⟷ *of-phantom* (*finite-UNIV* :: '*a finite-UNIV*)

by(*simp-all add*: *card-gt-0-iff finite-UNIV*)

end

context fixes *xs* :: *'a* :: *card-UNIV list*
begin

qualified definition *card'* :: *'a set* ⇒ *nat*
where [*simp*, *code-abbrev*]: *card'* = *card*

lemma *card'-code* [*code*]:
card' (*set xs*) = *length* (*remdups xs*)
card' (*List.coset xs*) = *of-phantom* (*card-UNIV* :: *'a card-UNIV*) − *length* (*remdups xs*)
by(*simp-all add*: *List.card-set card-Compl card-UNIV*)

qualified definition *subset'* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code-abbrev*]: *subset'* = (⊆)

lemma *subset'-code* [*code*]:
subset' *A* (*List.coset ys*) ⟷ (∀ *y* ∈ *set ys*. *y* ∉ *A*)
subset' (*set ys*) *B* ⟷ (∀ *y* ∈ *set ys*. *y* ∈ *B*)
subset' (*List.coset xs*) (*set ys*) ⟷ (*let n* = *CARD*(*'a*) *in n* > *0* ∧ *card*(*set* (*xs* @ *ys*)) = *n*)
by(*auto simp add*: *Let-def card-gt-0-iff dest*: *card-eq-UNIV-imp-eq-UNIV intro*: *arg-cong*[**where** *f*=*card*])
(*metis finite-compl finite-set rev-finite-subset*)

qualified definition *eq-set* :: *'a set* ⇒ *'a set* ⇒ *bool*
where [*simp*, *code-abbrev*]: *eq-set* = (=)

lemma *eq-set-code* [*code*]:
fixes *ys*
defines *rhs* ≡
let n = *CARD*(*'a*)
in if n = *0 then False else*
let xs' = *remdups xs*; *ys'* = *remdups ys*
in length xs' + *length ys'* = *n* ∧ (∀ *x* ∈ *set xs'*. *x* ∉ *set ys'*) ∧ (∀ *y* ∈ *set ys'*. *y* ∉ *set xs'*)
shows *eq-set* (*List.coset xs*) (*set ys*) ⟷ *rhs*
and *eq-set* (*set ys*) (*List.coset xs*) ⟷ *rhs*
and *eq-set* (*set xs*) (*set ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
and *eq-set* (*List.coset xs*) (*List.coset ys*) ⟷ (∀ *x* ∈ *set xs*. *x* ∈ *set ys*) ∧ (∀ *y* ∈ *set ys*. *y* ∈ *set xs*)
proof *goal-cases*
{
case *1*

```
    show ?case (is ?lhs ⟷ ?rhs)
    proof
      show ?rhs if ?lhs
        using that
        by (auto simp add: rhs-def Let-def List.card-set[symmetric]
          card-Un-Int[where A=set xs and B=− set xs] card-UNIV
          Compl-partition card-gt-0-iff dest: sym)(metis finite-compl finite-set)
      show ?lhs if ?rhs
      proof −
        have ⟦ ∀ y∈set xs. y ∉ set ys; ∀ x∈set ys. x ∉ set xs ⟧ ⟹ set xs ∩ set ys =
{} by blast
        with that show ?thesis
          by (auto simp add: rhs-def Let-def List.card-set[symmetric]
            card-UNIV card-gt-0-iff card-Un-Int[where A=set xs and B=set ys]
            dest: card-eq-UNIV-imp-eq-UNIV split: if-split-asm)
      qed
    qed
  }
  moreover
  case 2
  ultimately show ?case unfolding eq-set-def by blast
next
  case 3
  show ?case unfolding eq-set-def List.coset-def by blast
next
  case 4
  show ?case unfolding eq-set-def List.coset-def by blast
qed

end
```

Provide more informative exceptions than Match for non-rewritten cases. If generated code raises one these exceptions, then a code equation calls the mentioned operator for an element type that is not an instance of *card-UNIV* and is therefore not implemented via *card-UNIV-class.card-UNIV*. Constrain the element type with sort *card-UNIV* to change this.

```
lemma card-code [code]:
  card (set xs) = length (remdups xs)
  card (List.coset xs) =
   Code.abort (STR ''card (List.coset -) requires type class instance card-UNIV'')
     (λ-. card (List.coset xs))
  by (simp-all add: length-remdups-card-conv)

lemma coset-subseteq-set-code [code]:
  set xs ⊆ B = list-all (λx. x ∈ B) xs
  A ⊆ List.coset ys = list-all (λy. y ∉ A) ys
  List.coset xs ⊆ set ys ⟷
  (if xs = [] ∧ ys = [] then False
   else Code.abort
```

(*STR ''subset-eq (List.coset -) (List.set -) requires type class instance card-UNIV''*)
(λ-. *List.coset xs* ⊆ *set ys*))
by (*auto simp add*: *list-all-iff*)

notepad begin — test code setup
have *List.coset* [*True*] = *set* [*False*] ∧
List.coset [] ⊆ *List.set* [*True*, *False*] ∧
finite (*List.coset* [*True*])
by *eval*

end

end

11 Eliminating pattern matches

theory *Case-Converter*
imports *Main*
begin

definition *missing-pattern-match* :: *String.literal* ⇒ (*unit* ⇒ *'a*) ⇒ *'a* **where**
missing-pattern-match m f = *f* ()

lemma *missing-pattern-match-cong* [*cong*]:
m = *m'* ⟹ *missing-pattern-match m f* = *missing-pattern-match m' f*
by(*rule arg-cong*)

lemma *missing-pattern-match-code* [*code*, *code-unfold*]:
missing-pattern-match = *Code.abort*
unfolding *missing-pattern-match-def Code.abort-def* **..**

ML-file ‹*case-converter.ML*›

end

12 Lazy types in generated code

theory *Code-Lazy*
imports *Case-Converter*
keywords
code-lazy-type
activate-lazy-type
deactivate-lazy-type
activate-lazy-types
deactivate-lazy-types
print-lazy-types :: *thy-decl*
begin

This theory and the CodeLazy tool described in [3].

It hooks into Isabelle's code generator such that the generated code evaluates a user-specified set of type constructors lazily, even in target languages with eager evaluation. The lazy type must be algebraic, i.e., values must be built from constructors and a corresponding case operator decomposes them. Every datatype and codatatype is algebraic and thus eligible for lazification.

12.1 The type *lazy*

typedef *'a lazy = UNIV :: 'a set* **..**
setup-lifting *type-definition-lazy*
lift-definition *delay :: (unit ⇒ 'a) ⇒ 'a lazy* **is** *λf. f ()* **.**
lift-definition *force :: 'a lazy ⇒ 'a* **is** *λx. x* **.**

code-datatype *delay*
lemma *force-delay* [*code*]: *force (delay f) = f ()* **by** *transfer (rule refl)*
lemma *delay-force*: *delay (λ-. force s) = s* **by** *transfer (rule refl)*

definition *termify-lazy2 :: 'a :: typerep lazy ⇒ term*
where *termify-lazy2 x =*
Code-Evaluation.App (Code-Evaluation.Const (STR ''Code-Lazy.delay'') (TYPEREP((unit ⇒ 'a) ⇒ 'a lazy)))
(Code-Evaluation.Const (STR ''Pure.dummy-pattern'') (TYPEREP((unit ⇒ 'a))))

definition *termify-lazy ::*
(String.literal ⇒ 'typerep ⇒ 'term) ⇒
('term ⇒ 'term ⇒ 'term) ⇒
(String.literal ⇒ 'typerep ⇒ 'term ⇒ 'term) ⇒
'typerep ⇒ ('typerep ⇒ 'typerep ⇒ 'typerep) ⇒ ('typerep ⇒ 'typerep) ⇒
('a ⇒ 'term) ⇒ 'typerep ⇒ 'a :: typerep lazy ⇒ 'term ⇒ term
where *termify-lazy - - - - - - - - - x - = termify-lazy2 x*

declare [[*code drop*: *Code-Evaluation.term-of :: - lazy ⇒ -*]]

lemma *term-of-lazy-code* [*code*]:
Code-Evaluation.term-of x ≡
termify-lazy
Code-Evaluation.Const Code-Evaluation.App Code-Evaluation.Abs
TYPEREP(unit) (λT U. typerep.Typerep (STR ''fun'') [T, U]) (λT. typerep.Typerep (STR ''Code-Lazy.lazy'') [T])
Code-Evaluation.term-of TYPEREP('a) x (Code-Evaluation.Const (STR '''') (TYPEREP(unit)))
for *x :: 'a :: {typerep, term-of} lazy*
by (*rule term-of-anything*)

The implementations of - *lazy* using language primitives cache forced values.

Term reconstruction for lazy looks into the lazy value and reconstructs it to the depth it has been evaluated. This is not done for Haskell as we

do not know of any portable way to inspect whether a lazy value has been evaluated to or not.

code-printing code-module *Lazy* ⇀ (*SML*) **file** ~~/*src*/*HOL*/*Library*/*Tools*/*lazy.ML*
  for type-constructor *lazy* **constant** *delay force termify-lazy*
| **type-constructor** *lazy* ⇀ (*SML*) - *Lazy.lazy*
| **constant** *delay* ⇀ (*SML*) *Lazy.lazy*
| **constant** *force* ⇀ (*SML*) *Lazy.force*
| **constant** *termify-lazy* ⇀ (*SML*) *Lazy.termify'-lazy*

code-reserved (*SML*) *Lazy*

code-printing — For code generation within the Isabelle environment, we reuse the thread-safe implementation of lazy from `~~/src/Pure/Concurrent/lazy.ML`
  code-module *Lazy* ⇀ (*Eval*) ‹› **for constant** *undefined*
| **type-constructor** *lazy* ⇀ (*Eval*) - *Lazy.lazy*
| **constant** *delay* ⇀ (*Eval*) *Lazy.lazy*
| **constant** *force* ⇀ (*Eval*) *Lazy.force*
| **code-module** *Termify-Lazy* ⇀ (*Eval*) **file** ~~/*src*/*HOL*/*Library*/*Tools*/*termify-lazy.ML*
  for constant *termify-lazy*
| **constant** *termify-lazy* ⇀ (*Eval*) *Termify'-Lazy.termify'-lazy*

code-reserved (*Eval*) *Termify-Lazy*

code-printing
  type-constructor *lazy* ⇀ (*OCaml*) - *Lazy.t*
| **constant** *delay* ⇀ (*OCaml*) *Lazy.from'-fun*
| **constant** *force* ⇀ (*OCaml*) *Lazy.force*
| **code-module** *Termify-Lazy* ⇀ (*OCaml*) **file** ~~/*src*/*HOL*/*Library*/*Tools*/*termify-lazy.ocaml*
  for constant *termify-lazy*
| **constant** *termify-lazy* ⇀ (*OCaml*) *Termify'-Lazy.termify'-lazy*

code-reserved (*OCaml*) *Lazy Termify-Lazy*

code-printing
  code-module *Lazy* ⇀ (*Haskell*) **file** ~~/*src*/*HOL*/*Library*/*Tools*/*lazy.hs*
  for type-constructor *lazy* **constant** *delay force*
| **type-constructor** *lazy* ⇀ (*Haskell*) *Lazy.Lazy* -
| **constant** *delay* ⇀ (*Haskell*) *Lazy.delay*
| **constant** *force* ⇀ (*Haskell*) *Lazy.force*

code-reserved (*Haskell*) *Lazy*

code-printing
  code-module *Lazy* ⇀ (*Scala*) **file** ~~/*src*/*HOL*/*Library*/*Tools*/*lazy.scala*
  for type-constructor *lazy* **constant** *delay force termify-lazy*
| **type-constructor** *lazy* ⇀ (*Scala*) *Lazy.Lazy*[-]
| **constant** *delay* ⇀ (*Scala*) *Lazy.delay*
| **constant** *force* ⇀ (*Scala*) *Lazy.force*
| **constant** *termify-lazy* ⇀ (*Scala*) *Lazy.termify'-lazy*

code-reserved (*Scala*) *Lazy*

Make evaluation with the simplifier respect *delay*s.

lemma *delay-lazy-cong*: *delay f* = *delay f* **by** *simp*
setup ‹*Code-Simp.map-ss* (*Simplifier.add-cong* @{*thm delay-lazy-cong*})›

12.2 Implementation

ML-file ‹*code-lazy.ML*›

setup ‹
Code-Preproc.add-functrans (*lazy-datatype*, *Code-Lazy.transform-code-eqs*)
›

end

13 Test infrastructure for the code generator

theory *Code-Test*
imports *Main*
keywords *test-code* :: *diag*
begin

13.1 YXML encoding for *term*

datatype (*plugins del*: *code size quickcheck*) *yxml-of-term* = *YXML*

lemma *yot-anything*: *x* = (*y* :: *yxml-of-term*)
by(*cases x y rule*: *yxml-of-term.exhaust*[*case-product yxml-of-term.exhaust*])(*simp*)

definition *yot-empty* :: *yxml-of-term* **where** [*code drop*]: *yot-empty* = *YXML*
definition *yot-literal* :: *String.literal* ⇒ *yxml-of-term*
where [*code drop*]: *yot-literal* - = *YXML*
definition *yot-append* :: *yxml-of-term* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
where [*code drop*]: *yot-append* - - = *YXML*
definition *yot-concat* :: *yxml-of-term list* ⇒ *yxml-of-term*
where [*code drop*]: *yot-concat* - = *YXML*

Serialise *yxml-of-term* to native string of target language

code-printing type-constructor *yxml-of-term*
⇀ (*SML*) *string*
and (*OCaml*) *string*
and (*Haskell*) *String*
and (*Scala*) *String*
| **constant** *yot-empty*
⇀ (*SML*)
and (*OCaml*)
and (*Haskell*)

```
  and (Scala)
| constant yot-literal
  ⇀ (SML) -
  and (OCaml) -
  and (Haskell) -
  and (Scala) -
| constant yot-append
  ⇀ (SML) String.concat [(-), (-)]
  and (OCaml) String.concat  [(-); (-)]
  and (Haskell) infixr 5 ++
  and (Scala) infixl 5 +
| constant yot-concat
  ⇀ (SML) String.concat
  and (OCaml) String.concat
  and (Haskell) Prelude.concat
  and (Scala) -.mkString()
```

Stripped-down implementations of Isabelle's XML tree with YXML encoding as defined in `~~/src/Pure/PIDE/xml.ML`, `~~/src/Pure/PIDE/yxml.ML` sufficient to encode *term* as in `~~/src/Pure/term_xml.ML`.

```
datatype (plugins del: code size quickcheck) xml-tree = XML-Tree

lemma xml-tree-anything: x = (y :: xml-tree)
by(cases x y rule: xml-tree.exhaust[case-product xml-tree.exhaust])(simp)

context begin
local-setup ‹Local-Theory.map-background-naming (Name-Space.mandatory-path
xml)›

type-synonym attributes = (String.literal × String.literal) list
type-synonym body = xml-tree list

definition Elem :: String.literal ⇒ attributes ⇒ xml-tree list ⇒ xml-tree
where [code drop]: Elem - - - = XML-Tree

definition Text :: String.literal ⇒ xml-tree
where [code drop]: Text - = XML-Tree

definition node :: xml-tree list ⇒ xml-tree
where node ts = Elem (STR '':'') [] ts

definition tagged :: String.literal ⇒ String.literal option ⇒ xml-tree list ⇒ xml-tree
where tagged tag x ts = Elem tag (case x of None ⇒ [] | Some x' ⇒ [(STR ''0'',
x')]) ts

definition list where list f xs = map (node ∘ f) xs

definition X :: yxml-of-term where X = yot-literal (STR 0x05)
definition Y :: yxml-of-term where Y = yot-literal (STR 0x06)
```

definition *XY* :: *yxml-of-term* **where** *XY* = *yot-append X Y*
definition *XYX* :: *yxml-of-term* **where** *XYX* = *yot-append XY X*

end

code-datatype *xml.Elem xml.Text*

definition *yxml-string-of-xml-tree* :: *xml-tree* ⇒ *yxml-of-term* ⇒ *yxml-of-term*
where *yxml-string-of-xml-tree - - = YXML*

lemma *yxml-string-of-xml-tree-code* [*code*]:
yxml-string-of-xml-tree (*xml.Elem name atts ts*) *rest* =
yot-append xml.XY (
yot-append (*yot-literal name*) (
foldr (λ(*a*, *x*) *rest*.
yot-append xml.Y (
yot-append (*yot-literal a*) (
yot-append (*yot-literal* (*STR* ''='')) (
yot-append (*yot-literal x*) *rest*)))) *atts* (
foldr yxml-string-of-xml-tree ts (
yot-append xml.XYX rest))))
yxml-string-of-xml-tree (*xml.Text s*) *rest* = *yot-append* (*yot-literal s*) *rest*
by(*rule yot-anything*)+

definition *yxml-string-of-body* :: *xml.body* ⇒ *yxml-of-term*
where *yxml-string-of-body ts* = *foldr yxml-string-of-xml-tree ts yot-empty*

Encoding *term* into XML trees as defined in `~~/src/Pure/term_xml.ML`.

definition *xml-of-typ* :: *Typerep.typerep* ⇒ *xml.body*
where *xml-of-typ - =* [*XML-Tree*]

definition *xml-of-term* :: *Code-Evaluation.term* ⇒ *xml.body*
where *xml-of-term - =* [*XML-Tree*]

lemma *xml-of-typ-code* [*code*]:
xml-of-typ (*typerep.Typerep t args*) = [*xml.tagged* (*STR* ''0'') (*Some t*) (*xml.list xml-of-typ args*)]
by(*simp add*: *xml-of-typ-def xml-tree-anything*)

lemma *xml-of-term-code* [*code*]:
xml-of-term (*Code-Evaluation.Const x ty*) = [*xml.tagged* (*STR* ''0'') (*Some x*) (*xml-of-typ ty*)]
xml-of-term (*Code-Evaluation.App t1 t2*) = [*xml.tagged* (*STR* ''5'') *None* [*xml.node* (*xml-of-term t1*), *xml.node* (*xml-of-term t2*)]]
xml-of-term (*Code-Evaluation.Abs x ty t*) = [*xml.tagged* (*STR* ''4'') (*Some x*) [*xml.node* (*xml-of-typ ty*), *xml.node* (*xml-of-term t*)]]
— FIXME: *Code-Evaluation.Free* is used only in *HOL.Quickcheck-Narrowing* to represent uninstantiated parameters in constructors. Here, we always translate them to `Free` variables.

xml-of-term (*Code-Evaluation.Free x ty*) = [*xml.tagged* (*STR ''1''*) (*Some x*) (*xml-of-typ ty*)]
by(*simp-all add*: *xml-of-term-def xml-tree-anything*)

definition *yxml-string-of-term* :: *Code-Evaluation.term* ⇒ *yxml-of-term*
where *yxml-string-of-term* = *yxml-string-of-body* ∘ *xml-of-term*

13.2 Test engine and drivers

ML-file ‹*code-test.ML*›

end

14 A combinator to build partial equivalence relations from a predicate and an equivalence relation

theory *Combine-PER*
imports *Main*
begin

unbundle *lattice-syntax*

definition *combine-per* :: ('*a* ⇒ *bool*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ '*a* ⇒ *bool*
where *combine-per P R* = (λ*x y*. *P x* ∧ *P y*) ⊓ *R*

lemma *combine-per-simp* [*simp*]:
combine-per P R x y ⟷ *P x* ∧ *P y* ∧ *x* ≈ *y* **for** *R* (**infixl** ‹≈› *50*)
by (*simp add*: *combine-per-def*)

lemma *combine-per-top* [*simp*]: *combine-per* ⊤ *R* = *R*
by (*simp add*: *fun-eq-iff*)

lemma *combine-per-eq* [*simp*]: *combine-per P HOL.eq* = *HOL.eq* ⊓ (λ*x y*. *P x*)
by (*auto simp add*: *fun-eq-iff*)

lemma *symp-combine-per*: *symp R* ⟹ *symp* (*combine-per P R*)
by (*auto simp add*: *symp-def sym-def combine-per-def*)

lemma *transp-combine-per*: *transp R* ⟹ *transp* (*combine-per P R*)
by (*auto simp add*: *transp-def trans-def combine-per-def*)

lemma *combine-perI*: *P x* ⟹ *P y* ⟹ *x* ≈ *y* ⟹ *combine-per P R x y* **for** *R* (**infixl** ‹≈› *50*)
by (*simp add*: *combine-per-def*)

lemma *symp-combine-per-symp*: *symp R* ⟹ *symp* (*combine-per P R*)
by (*auto intro*!: *sympI elim*: *sympE*)

lemma *transp-combine-per-transp*: *transp R* $\Longrightarrow$ *transp* (*combine-per P R*)
by (*auto intro*!: *transpI elim*: *transpE*)

lemma *equivp-combine-per-part-equivp* [*intro?*]:
fixes *R* (**infixl** ‹$\approx$› *50*)
assumes $\exists\, x.\ P\ x$ **and** *equivp R*
shows *part-equivp* (*combine-per P R*)
proof −
from ‹$\exists\, x.\ P\ x$› **obtain** *x* **where** *P x* **..**
moreover from ‹*equivp R*› **have** $x \approx x$
by (*rule equivp-reflp*)
ultimately have $\exists\, x.\ P\ x \wedge x \approx x$
by *blast*
with ‹*equivp R*› **show** *?thesis*
by (*auto intro*!: *part-equivpI symp-combine-per-symp transp-combine-per-transp*
elim: *equivpE*)
qed

end

15 Formalisation of chain-complete partial orders, continuity and admissibility

theory *Complete-Partial-Order2*
imports *Main*
begin

unbundle *lattice-syntax*

lemma *chain-transfer* [*transfer-rule*]:
includes *lifting-syntax*
shows ((*A* ===> *A* ===> (=)) ===> *rel-set A* ===> (=)) *Complete-Partial-Order.chain*
Complete-Partial-Order.chain
unfolding *chain-def*[*abs-def*] **by** *transfer-prover*

lemma *linorder-chain* [*simp*, *intro*!]:
fixes *Y* :: - :: *linorder set*
shows *Complete-Partial-Order.chain* (≤) *Y*
by(*auto intro*: *chainI*)

lemma *fun-lub-apply*: $\bigwedge$*Sup*. *fun-lub Sup Y x* = *Sup* ((λ*f*. *f x*) ‘ *Y*)
by(*simp add*: *fun-lub-def image-def*)

lemma *fun-lub-empty* [*simp*]: *fun-lub lub* {} = (λ-. *lub* {})
by(*rule ext*)(*simp add*: *fun-lub-apply*)

lemma *chain-fun-ordD*:

assumes *Complete-Partial-Order.chain* (*fun-ord le*) *Y*
shows *Complete-Partial-Order.chain le* ((λ*f*. *f x*) ‘ *Y*)
by(*rule chainI*)(*auto dest*: *chainD*[*OF assms*] *simp add*: *fun-ord-def*)

lemma *chain-Diff*:
Complete-Partial-Order.chain ord A
⟹ *Complete-Partial-Order.chain ord* (*A* − *B*)
by(*erule chain-subset*) *blast*

lemma *chain-rel-prodD1*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain orda* (*fst* ‘ *Y*)
by(*auto 4 3 simp add*: *chain-def*)

lemma *chain-rel-prodD2*:
Complete-Partial-Order.chain (*rel-prod orda ordb*) *Y*
⟹ *Complete-Partial-Order.chain ordb* (*snd* ‘ *Y*)
by(*auto 4 3 simp add*: *chain-def*)

context *ccpo* **begin**

lemma *ccpo-fun*: *class.ccpo* (*fun-lub Sup*) (*fun-ord* (≤)) (*mk-less* (*fun-ord* (≤)))
by *standard* (*auto 4 3 simp add*: *mk-less-def fun-ord-def fun-lub-apply*
intro: *order.trans order.antisym chain-imageI ccpo-Sup-upper ccpo-Sup-least*)

lemma *ccpo-Sup-below-iff*: *Complete-Partial-Order.chain* (≤) *Y* ⟹ *Sup Y* ≤ *x* ⟷ (∀ *y*∈*Y*. *y* ≤ *x*)
by (*meson local.ccpo-Sup-least local.ccpo-Sup-upper local.dual-order.trans*)

lemma *Sup-minus-bot*:
assumes *chain*: *Complete-Partial-Order.chain* (≤) *A*
shows ⨆(*A* − {⨆{}}) = ⨆*A*
(**is** *?lhs* = *?rhs*)
proof (*rule order.antisym*)
show *?lhs* ≤ *?rhs*
by (*blast intro*: *ccpo-Sup-least chain-Diff*[*OF chain*] *ccpo-Sup-upper*[*OF chain*])
show *?rhs* ≤ *?lhs*
proof (*rule ccpo-Sup-least* [*OF chain*])
show *x* ∈ *A* ⟹ *x* ≤ *?lhs* **for** *x*
by (*cases x* = ⨆{})
(*blast intro*: *ccpo-Sup-least chain-empty ccpo-Sup-upper*[*OF chain-Diff*[*OF chain*]])+
qed
qed

lemma *mono-lub*:
fixes *le-b* (**infix** ‹⊑› *60*)
assumes *chain*: *Complete-Partial-Order.chain* (*fun-ord* (≤)) *Y*

and *mono*: $\bigwedge f.\ f \in Y \Longrightarrow$ *monotone le-b* $(\leq)$ f
shows *monotone* $(\sqsubseteq)$ $(\leq)$ (*fun-lub Sup* Y)
proof(*rule monotoneI*)
fix x y
assume $x \sqsubseteq y$

have *chain''*: $\bigwedge x.$ *Complete-Partial-Order.chain* $(\leq)$ $((\lambda f.\ f\ x)\ `\ Y)$
using *chain* **by**(*rule chain-imageI*)(*simp add*: *fun-ord-def*)
then show *fun-lub Sup* $Y\ x \leq$ *fun-lub Sup* $Y\ y$ **unfolding** *fun-lub-apply*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in (\lambda f.\ f\ x)\ `\ Y$
then obtain f **where** $f \in Y\ x' = f\ x$ **by** *blast*
note $\langle x' = f\ x\rangle$ **also**
from $\langle f \in Y\rangle$ $\langle x \sqsubseteq y\rangle$ **have** $f\ x \leq f\ y$ **by**(*blast dest*: *mono monotoneD*)
also have $\ldots \leq \bigsqcup((\lambda f.\ f\ y)\ `\ Y)$ **using** *chain''*
by(*rule ccpo-Sup-upper*)(*simp add*: $\langle f \in Y\rangle$)
finally show $x' \leq \bigsqcup((\lambda f.\ f\ y)\ `\ Y)$ **.**
qed
qed

context
fixes *le-b* (**infix** $\langle\sqsubseteq\rangle$ *60*) **and** Y f
assumes *chain*: *Complete-Partial-Order.chain le-b* Y
and *mono1*: $\bigwedge y.\ y \in Y \Longrightarrow$ *monotone le-b* $(\leq)$ $(\lambda x.\ f\ x\ y)$
and *mono2*: $\bigwedge x\ a\ b.\ [\![\ x \in Y;\ a \sqsubseteq b;\ a \in Y;\ b \in Y\ ]\!] \Longrightarrow f\ x\ a \leq f\ x\ b$
begin

lemma *Sup-mono*:
assumes *le*: $x \sqsubseteq y$ **and** x: $x \in Y$ **and** y: $y \in Y$
shows $\bigsqcup(f\ x\ `\ Y) \leq \bigsqcup(f\ y\ `\ Y)$ (**is** *- ≤ ?rhs*)
proof(*rule ccpo-Sup-least*)
from *chain* **show** *chain'*: *Complete-Partial-Order.chain* $(\leq)$ $(f\ x\ `\ Y)$ **when** $x \in Y$ **for** x
by(*rule chain-imageI*) (*insert that*, *auto dest*: *mono2*)

fix x'
assume $x' \in f\ x\ `\ Y$
then obtain y' **where** $y' \in Y\ x' = f\ x\ y'$ **by** *blast* **note** *this*(*2*)
also from *mono1*[*OF* $\langle y' \in Y\rangle$] *le* **have** $\ldots \leq f\ y\ y'$ **by**(*rule monotoneD*)
also have $\ldots \leq$ *?rhs* **using** *chain'*[*OF y*]
by (*auto intro*!: *ccpo-Sup-upper simp add*: $\langle y' \in Y\rangle$)
finally show $x' \leq$ *?rhs* **.**
qed(*rule x*)

lemma *diag-Sup*: $\bigsqcup((\lambda x.\ \bigsqcup(f\ x\ `\ Y))\ `\ Y) = \bigsqcup((\lambda x.\ f\ x\ x)\ `\ Y)$ (**is** *?lhs = ?rhs*)
proof(*rule order.antisym*)
have *chain1*: *Complete-Partial-Order.chain* $(\leq)$ $((\lambda x.\ \bigsqcup(f\ x\ `\ Y))\ `\ Y)$
using *chain* **by**(*rule chain-imageI*)(*rule Sup-mono*)

have *chain2*: $\bigwedge y'.\ y' \in Y \Longrightarrow$ *Complete-Partial-Order.chain* $(\leq)$ $(f\ y'\ `\ Y)$ **using** *chain*
by(*rule chain-imageI*)(*auto dest*: *mono2*)
have *chain3*: *Complete-Partial-Order.chain* $(\leq)$ $((\lambda x.\ f\ x\ x)\ `\ Y)$
using *chain* **by**(*rule chain-imageI*)(*auto intro*: *monotoneD*[*OF mono1*] *mono2 order.trans*)

show *?lhs* $\leq$ *?rhs* **using** *chain1*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in (\lambda x.\ \bigsqcup(f\ x\ `\ Y))\ `\ Y$
then obtain y' **where** $y' \in Y\ x' = \bigsqcup(f\ y'\ `\ Y)$ **by** *blast* **note** *this*(*2*)
also have $\ldots \leq$ *?rhs* **using** *chain2*[*OF* $\langle y' \in Y\rangle$]
proof(*rule ccpo-Sup-least*)
fix x
assume $x \in f\ y'\ `\ Y$
then obtain y **where** $y \in Y$ **and** x: $x = f\ y'\ y$ **by** *blast*
define y'' **where** $y'' = (\mathit{if}\ y \sqsubseteq y'\ \mathit{then}\ y'\ \mathit{else}\ y)$
from *chain* $\langle y \in Y\rangle$ $\langle y' \in Y\rangle$ **have** $y \sqsubseteq y' \vee y' \sqsubseteq y$ **by**(*rule chainD*)
hence $f\ y'\ y \leq f\ y''\ y''$ **using** $\langle y \in Y\rangle$ $\langle y' \in Y\rangle$
by(*auto simp add*: *y''-def intro*: *mono2 monotoneD*[*OF mono1*])
also from $\langle y \in Y\rangle$ $\langle y' \in Y\rangle$ **have** $y'' \in Y$ **by**(*simp add*: *y''-def*)
from *chain3* **have** $f\ y''\ y'' \leq$ *?rhs* **by**(*rule ccpo-Sup-upper*)(*simp add*: $\langle y'' \in Y\rangle$)
finally show $x \leq$ *?rhs* **by**(*simp add*: x)
qed
finally show $x' \leq$ *?rhs* **.**
qed

show *?rhs* $\leq$ *?lhs* **using** *chain3*
proof(*rule ccpo-Sup-least*)
fix y
assume $y \in (\lambda x.\ f\ x\ x)\ `\ Y$
then obtain x **where** $x \in Y$ **and** $y = f\ x\ x$ **by** *blast*
then show $y \leq$ *?lhs*
by (*metis* (*no-types*, *lifting*) *chain1 chain2 imageI ccpo-Sup-upper order.trans*)
qed
qed

end

lemma *Sup-image-mono-le*:
fixes *le-b* (**infix** $\langle\sqsubseteq\rangle$ *60*) **and** *Sup-b* $(\langle\bigsqcup\rangle)$
assumes *ccpo*: *class.ccpo Sup-b* $(\sqsubseteq)$ *lt-b*
assumes *chain*: *Complete-Partial-Order.chain* $(\sqsubseteq)$ Y
and *mono*: $\bigwedge x\ y.\ [\![\ x \sqsubseteq y;\ x \in Y\ ]\!] \Longrightarrow f\ x \leq f\ y$
shows *Sup* $(f\ `\ Y) \leq f\ (\bigsqcup Y)$
proof(*rule ccpo-Sup-least*)
show *Complete-Partial-Order.chain* $(\leq)$ $(f\ `\ Y)$

using *chain* **by**(*rule chain-imageI*)(*rule mono*)

fix x
assume $x \in f \mathbin{`} Y$
then obtain y **where** $y \in Y$ **and** $x = f\ y$ **by** *blast* **note** *this*(*2*)
also have $y \sqsubseteq \bigsqcup Y$ **using** *ccpo chain* ‹$y \in Y$› **by**(*rule ccpo.ccpo-Sup-upper*)
hence $f\ y \leq f\ (\bigsqcup Y)$ **using** ‹$y \in Y$› **by**(*rule mono*)
finally show $x \leq \ldots$ **.**
qed

lemma *swap-Sup*:
fixes *le-b* (**infix** ‹$\sqsubseteq$› *60*)
assumes Y: *Complete-Partial-Order.chain* ($\sqsubseteq$) Y
and Z: *Complete-Partial-Order.chain* (*fun-ord* ($\leq$)) Z
and *mono*: $\bigwedge f.\ f \in Z \Longrightarrow$ *monotone* ($\sqsubseteq$) ($\leq$) f
shows $\bigsqcup((\lambda x.\ \bigsqcup(x \mathbin{`} Y)) \mathbin{`} Z) = \bigsqcup((\lambda x.\ \bigsqcup((\lambda f.\ f\ x) \mathbin{`} Z)) \mathbin{`} Y)$
(**is** *?lhs* = *?rhs*)
proof(*cases* $Y = \{\}$)
case *True*
then show *?thesis*
by (*simp add*: *image-constant-conv cong del*: *SUP-cong-simp*)
next
case *False*
have *chain1*: $\bigwedge f.\ f \in Z \Longrightarrow$ *Complete-Partial-Order.chain* ($\leq$) ($f \mathbin{`} Y$)
by(*rule chain-imageI*[*OF Y*])(*rule monotoneD*[*OF mono*])
have *chain2*: *Complete-Partial-Order.chain* ($\leq$) ($(\lambda x.\ \bigsqcup(x \mathbin{`} Y)) \mathbin{`} Z$) **using** Z
proof(*rule chain-imageI*)
fix $f\ g$
assume $f \in Z\ g \in Z$
and *fun-ord* ($\leq$) $f\ g$
from *chain1*[*OF* ‹$f \in Z$›] **show** $\bigsqcup(f \mathbin{`} Y) \leq \bigsqcup(g \mathbin{`} Y)$
proof(*rule ccpo-Sup-least*)
fix x
assume $x \in f \mathbin{`} Y$
then obtain y **where** $y \in Y\ x = f\ y$ **by** *blast* **note** *this*(*2*)
also have $\ldots \leq g\ y$ **using** ‹*fun-ord* ($\leq$) $f\ g$› **by**(*simp add*: *fun-ord-def*)
also have $\ldots \leq \bigsqcup(g \mathbin{`} Y)$ **using** *chain1*[*OF* ‹$g \in Z$›]
by(*rule ccpo-Sup-upper*)(*simp add*: ‹$y \in Y$›)
finally show $x \leq \bigsqcup(g \mathbin{`} Y)$ **.**
qed
qed
have *chain3*: $\bigwedge x.$ *Complete-Partial-Order.chain* ($\leq$) ($(\lambda f.\ f\ x) \mathbin{`} Z$)
using Z **by**(*rule chain-imageI*)(*simp add*: *fun-ord-def*)
have *chain4*: *Complete-Partial-Order.chain* ($\leq$) ($(\lambda x.\ \bigsqcup((\lambda f.\ f\ x) \mathbin{`} Z)) \mathbin{`} Y$)
using Y
proof(*rule chain-imageI*)
fix $f\ x\ y$
assume $x \sqsubseteq y$

```
    show ⨆((λf. f x) ` Z) ≤ ⨆((λf. f y) ` Z) (is - ≤ ?rhs) using chain3
    proof(rule ccpo-Sup-least)
      fix x'
      assume x' ∈ (λf. f x) ` Z
      then obtain f where f ∈ Z x' = f x by blast
      then show x' ≤ ?rhs
        by (metis (mono-tags, lifting) ‹x ⊑ y› chain3 imageI ccpo-Sup-upper
            order-trans mono monotoneD)
    qed
  qed

  from chain2 have ?lhs ≤ ?rhs
  proof(rule ccpo-Sup-least)
    fix x
    assume x ∈ (λx. ⨆(x ` Y)) ` Z
    then obtain f where f ∈ Z x = ⨆(f ` Y) by blast note this(2)
    also have ... ≤ ?rhs using chain1[OF ‹f ∈ Z›]
    proof(rule ccpo-Sup-least)
      fix x'
      assume x' ∈ f ` Y
      then obtain y where y ∈ Y x' = f y by blast
      then show x' ≤ ?rhs
      by (metis (mono-tags, lifting) ‹f ∈ Z› chain3 chain4 imageI local.ccpo-Sup-upper
          order.trans)
    qed
    finally show x ≤ ?rhs .
  qed
  moreover
  have ?rhs ≤ ?lhs using chain4
  proof(rule ccpo-Sup-least)
    fix x
    assume x ∈ (λx. ⨆((λf. f x) ` Z)) ` Y
    then obtain y where y ∈ Y x = ⨆((λf. f y) ` Z) by blast note this(2)
    also have ... ≤ ?lhs using chain3
    proof(rule ccpo-Sup-least)
      fix x'
      assume x' ∈ (λf. f y) ` Z
      then obtain f where f ∈ Z x' = f y by blast
      then show x' ≤ ?lhs
        by (metis (mono-tags, lifting) ‹y ∈ Y› ccpo-Sup-below-iff chain1 chain2
imageI
          ccpo-Sup-upper)
    qed
    finally show x ≤ ?lhs .
  qed
  ultimately show ?lhs = ?rhs
    by (rule order.antisym)
qed
```

lemma *fixp-mono*:
assumes *fg*: *fun-ord* (≤) *f g*
and *f*: *monotone* (≤) (≤) *f*
and *g*: *monotone* (≤) (≤) *g*
shows *ccpo-class.fixp f* ≤ *ccpo-class.fixp g*
unfolding *fixp-def*
proof(*rule ccpo-Sup-least*)
fix *x*
assume *x* ∈ *ccpo-class.iterates f*
thus *x* ≤ ⨆ *ccpo-class.iterates g*
proof *induction*
case (*step x*)
from *f step.IH* **have** *f x* ≤ *f* (⨆ *ccpo-class.iterates g*) **by**(*rule monotoneD*)
also have ... ≤ *g* (⨆ *ccpo-class.iterates g*) **using** *fg* **by**(*simp add*: *fun-ord-def*)
also have ... = ⨆ *ccpo-class.iterates g* **by**(*fold fixp-def fixp-unfold*[*OF g*]) *simp*
finally show *?case* **.**
qed(*blast intro*: *ccpo-Sup-least*)
qed(*rule chain-iterates*[*OF f*])

context fixes *ordb* :: *'b* ⇒ *'b* ⇒ *bool* (**infix** ‹⊑› *60*) **begin**

lemma *iterates-mono*:
assumes *f*: *f* ∈ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* (≤)) *F*
and *mono*: ⋀*f*. *monotone* (⊑) (≤) *f* ⟹ *monotone* (⊑) (≤) (*F f*)
shows *monotone* (⊑) (≤) *f*
using *f*
by(*induction rule*: *ccpo.iterates.induct*[*OF ccpo-fun*, *consumes 1*])(*blast intro*: *mono mono-lub*)+

lemma *fixp-preserves-mono*:
assumes *mono*: ⋀*x*. *monotone* (*fun-ord* (≤)) (≤) (λ*f*. *F f x*)
and *mono2*: ⋀*f*. *monotone* (⊑) (≤) *f* ⟹ *monotone* (⊑) (≤) (*F f*)
shows *monotone* (⊑) (≤) (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* (≤)) *F*)
(**is** *monotone - - ?fixp*)
proof(*rule monotoneI*)
have *mono*: *monotone* (*fun-ord* (≤)) (*fun-ord* (≤)) *F*
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def intro*: *monotoneD*[*OF mono*])
let *?iter* = *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* (≤)) *F*
have *chain*: ⋀*x*. *Complete-Partial-Order.chain* (≤) ((λ*f*. *f x*) ‘ *?iter*)
by(*rule chain-imageI*[*OF ccpo.chain-iterates*[*OF ccpo-fun mono*]])(*simp add*: *fun-ord-def*)
fix *x y*
assume *x* ⊑ *y*
have (⨆*f*∈*?iter*. *f x*) ≤ (⨆*f*∈*?iter*. *f y*)
using *chain*
proof(*rule ccpo-Sup-least*)
fix *x'*
assume *x'* ∈ (λ*f*. *f x*) ‘ *?iter*
then obtain *f* **where** *f*: *f* ∈ *?iter x'* = *f x* **by** *blast*

then have $f\ x \leq f\ y$
by (*metis* ‹$x \sqsubseteq y$› *iterates-mono mono2 monotoneD*)
also have $f\ y \leq \bigsqcup((\lambda f.\ f\ y)\ `\ ?iter)$
using *chain f local.ccpo-Sup-upper* **by** *auto*
finally show $x' \leq \ldots$
using *f*(*2*) **by** *blast*
qed
then show *?fixp x* ≤ *?fixp y*
unfolding *ccpo.fixp-def*[*OF ccpo-fun*] *fun-lub-apply* **.**
qed

end

end

lemma *monotone2monotone*:
assumes *2*: $\bigwedge x.$ *monotone ordb ordc* $(\lambda y.\ f\ x\ y)$
and *t*: *monotone orda ordb* $(\lambda x.\ t\ x)$
and *1*: $\bigwedge y.$ *monotone orda ordc* $(\lambda x.\ f\ x\ y)$
and *trans*: *transp ordc*
shows *monotone orda ordc* $(\lambda x.\ f\ x\ (t\ x))$
using *assms* **unfolding** *monotone-on-def* **by** (*metis UNIV-I transpE*)

15.1 Continuity

definition *cont* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where
cont luba orda lubb ordb f ⟷
$(\forall Y.$ *Complete-Partial-Order.chain orda Y* ⟶ $Y \neq \{\}$ ⟶ *f* (*luba Y*) = *lubb* (*f* ‘ *Y*))

definition *mcont* :: $('a\ set \Rightarrow 'a) \Rightarrow ('a \Rightarrow 'a \Rightarrow bool) \Rightarrow ('b\ set \Rightarrow 'b) \Rightarrow ('b \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
where
mcont luba orda lubb ordb f ⟷
monotone orda ordb f ∧ *cont luba orda lubb ordb f*

15.1.1 Theorem collection *cont-intro*

named-theorems *cont-intro continuity and admissibility intro rules*
ML ‹
(∗ *apply cont-intro rules as intro and try to solve*
the remaining of the emerging subgoals with simp ∗)
fun cont-intro-tac ctxt =
REPEAT-ALL-NEW (*resolve-tac ctxt* (*rev* (*Named-Theorems.get ctxt* ***named-theorems*** ‹*cont-intro*›)))
THEN-ALL-NEW (*SOLVED'* (*simp-tac ctxt*))

fun cont-intro-simproc ctxt ct =

```
  let
    fun mk-stmt t = t
      |> HOLogic.mk-Trueprop
      |> Thm.cterm-of ctxt
      |> Goal.init
    fun mk-thm t =
      if exists-subterm Term.is-Var t then
        NONE
      else
        case SINGLE (cont-intro-tac ctxt 1) (mk-stmt t) of
          SOME thm => SOME (Goal.finish ctxt thm RS @{thm Eq-TrueI})
        | NONE => NONE
  in
    case Thm.term-of ct of
      t as Const- ‹ccpo.admissible - for - - -› => mk-thm t
    | t as Const- ‹mcont - - for - - - - -› => mk-thm t
    | t as Const- ‹monotone-on - - for - - - -› => mk-thm t
    | - => NONE
  end
  handle THM - => NONE
  | TYPE - => NONE
›

simproc-setup cont-intro
  ( ccpo.admissible lub ord P
  | mcont lub ord lub' ord' f
  | monotone ord ord' f
  ) = ‹K cont-intro-simproc›

lemmas [cont-intro] =
  call-mono
  let-mono
  if-mono
  option.const-mono
  tailrec.const-mono
  bind-mono
```

experiment begin

The following proof by simplification diverges if variables are not handled properly.

lemma $(\bigwedge f.\ monotone\ R\ S\ f \Longrightarrow thesis) \Longrightarrow monotone\ R\ S\ g \Longrightarrow thesis$
by *simp*

end

declare *if-mono*[*simp*]

lemma *monotone-id'* [*cont-intro*]: *monotone ord ord* $(\lambda x.\ x)$

by(*simp add*: *monotone-def*)

lemma *monotone-applyI*:
monotone orda ordb F $\Longrightarrow$ *monotone* (*fun-ord orda*) *ordb* ($\lambda f.$ *F* (*f x*))
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def dest*: *monotoneD*)

lemma *monotone-if-fun* [*partial-function-mono*]:
⟦ *monotone* (*fun-ord orda*) (*fun-ord ordb*) *F*; *monotone* (*fun-ord orda*) (*fun-ord ordb*) *G* ⟧
$\Longrightarrow$ *monotone* (*fun-ord orda*) (*fun-ord ordb*) ($\lambda f\ n.$ *if c n then F f n else G f n*)
by(*simp add*: *monotone-def fun-ord-def*)

lemma *monotone-fun-apply-fun* [*partial-function-mono*]:
monotone (*fun-ord* (*fun-ord ord*)) (*fun-ord ord*) ($\lambda f\ n.$ *f t* (*g n*))
by(*rule monotoneI*)(*simp add*: *fun-ord-def*)

lemma *monotone-fun-ord-apply*:
monotone orda (*fun-ord ordb*) *f* $\longleftrightarrow$ ($\forall x.$ *monotone orda ordb* ($\lambda y.$ *f y x*))
by(*auto simp add*: *monotone-def fun-ord-def*)

context *preorder* **begin**

declare *transp-on-le*[*cont-intro*]

lemma *monotone-const* [*simp*, *cont-intro*]: *monotone ord* ($\leq$) (λ-. *c*)
by(*rule monotoneI*) *simp*

end

lemma *transp-le* [*cont-intro*, *simp*]:
class.preorder ord (*mk-less ord*) $\Longrightarrow$ *transp ord*
by(*rule preorder.transp-on-le*)

context *partial-function-definitions* **begin**

declare *const-mono* [*cont-intro*, *simp*]

lemma *transp-le* [*cont-intro*, *simp*]: *transp leq*
by(*rule transpI*)(*rule leq-trans*)

lemma *preorder* [*cont-intro*, *simp*]: *class.preorder leq* (*mk-less leq*)
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *leq-refl leq-trans*)

declare *ccpo*[*cont-intro*, *simp*]

end

lemma *contI* [*intro?*]:
($\bigwedge Y.$ ⟦ *Complete-Partial-Order.chain orda Y*; *Y* $\neq$ {} ⟧ $\Longrightarrow$ *f* (*luba Y*) = *lubb*

$(f \text{ ` } Y))$
$\Longrightarrow$ *cont luba orda lubb ordb f*
unfolding *cont-def* **by** *blast*

lemma *contD*:
$[\![$ *cont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; $Y \neq \{\}$ $]\!]$
$\Longrightarrow$ *f* (*luba Y*) = *lubb* (*f* ` *Y*)
unfolding *cont-def* **by** *blast*

lemma *cont-id* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord id*
by(*rule contI*) *simp*

lemma *cont-id'* [*simp*, *cont-intro*]: $\bigwedge$*Sup*. *cont Sup ord Sup ord* ($\lambda x.$ *x*)
by (*simp add*: *Inf.INF-identity-eq contI*)

lemma *cont-applyI* [*cont-intro*]:
assumes *cont*: *cont luba orda lubb ordb g*
shows *cont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* ($\lambda f.$ *g* (*f x*))
using *assms* **by** (*simp add*: *cont-def chain-fun-ordD fun-lub-apply image-image*)

lemma *call-cont*: *cont* (*fun-lub lub*) (*fun-ord ord*) *lub ord* ($\lambda f.$ *f t*)
by(*simp add*: *cont-def fun-lub-apply*)

lemma *cont-if* [*cont-intro*]:
$[\![$ *cont luba orda lubb ordb f*; *cont luba orda lubb ordb g* $]\!]$
$\Longrightarrow$ *cont luba orda lubb ordb* ($\lambda x.$ *if c then f x else g x*)
by(*cases c*) *simp-all*

lemma *mcontI* [*intro?*]:
$[\![$ *monotone orda ordb f*; *cont luba orda lubb ordb f* $]\!] \Longrightarrow$ *mcont luba orda lubb ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-mono*: *mcont luba orda lubb ordb f* $\Longrightarrow$ *monotone orda ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-cont* [*simp*]: *mcont luba orda lubb ordb f* $\Longrightarrow$ *cont luba orda lubb ordb f*
by(*simp add*: *mcont-def*)

lemma *mcont-monoD*:
$[\![$ *mcont luba orda lubb ordb f*; *orda x y* $]\!] \Longrightarrow$ *ordb* (*f x*) (*f y*)
by(*auto simp add*: *mcont-def dest*: *monotoneD*)

lemma *mcont-contD*:
$[\![$ *mcont luba orda lubb ordb f*; *Complete-Partial-Order.chain orda Y*; $Y \neq \{\}$ $]\!]$
$\Longrightarrow$ *f* (*luba Y*) = *lubb* (*f* ` *Y*)
by(*auto simp add*: *mcont-def dest*: *contD*)

lemma *mcont-call* [*cont-intro*, *simp*]:
mcont (*fun-lub lub*) (*fun-ord ord*) *lub ord* (λ*f*. *f t*)
by(*simp add*: *mcont-def call-mono call-cont*)

lemma *mcont-id'* [*cont-intro*, *simp*]: *mcont lub ord lub ord* (λ*x*. *x*)
by(*simp add*: *mcont-def monotone-id'*)

lemma *mcont-applyI*:
mcont luba orda lubb ordb (λ*x*. *F x*) ⟹ *mcont* (*fun-lub luba*) (*fun-ord orda*) *lubb ordb* (λ*f*. *F* (*f x*))
by(*simp add*: *mcont-def monotone-applyI cont-applyI*)

lemma *mcont-if* [*cont-intro*, *simp*]:
⟦ *mcont luba orda lubb ordb* (λ*x*. *f x*); *mcont luba orda lubb ordb* (λ*x*. *g x*) ⟧
⟹ *mcont luba orda lubb ordb* (λ*x*. *if c then f x else g x*)
by(*simp add*: *mcont-def cont-if*)

lemma *cont-fun-lub-apply*:
cont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟷ (∀ *x*. *cont luba orda lubb ordb* (λ*y*. *f y x*))
by(*simp add*: *cont-def fun-lub-def fun-eq-iff*)(*auto simp add*: *image-def*)

lemma *mcont-fun-lub-apply*:
mcont luba orda (*fun-lub lubb*) (*fun-ord ordb*) *f* ⟷ (∀ *x*. *mcont luba orda lubb ordb* (λ*y*. *f y x*))
by(*auto simp add*: *monotone-fun-ord-apply cont-fun-lub-apply mcont-def*)

context *ccpo* **begin**

lemma *cont-const* [*simp*, *cont-intro*]: *cont luba orda Sup* (≤) (λ*x*. *c*)
by (*rule contI*) (*simp add*: *image-constant-conv cong del*: *SUP-cong-simp*)

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda Sup (≤) (λ*x*. *c*)
by(*simp add*: *mcont-def*)

lemma *cont-apply*:
assumes *2*: ⋀*x*. *cont lubb ordb Sup* (≤) (λ*y*. *f x y*)
and *t*: *cont luba orda lubb ordb* (λ*x*. *t x*)
and *1*: ⋀*y*. *cont luba orda Sup* (≤) (λ*x*. *f x y*)
and *mono*: *monotone orda ordb* (λ*x*. *t x*)
and *mono2*: ⋀*x*. *monotone ordb* (≤) (λ*y*. *f x y*)
and *mono1*: ⋀*y*. *monotone orda* (≤) (λ*x*. *f x y*)
shows *cont luba orda Sup* (≤) (λ*x*. *f x* (*t x*))
proof
fix *Y*
assume *chain*: *Complete-Partial-Order.chain orda Y* **and** *Y* ≠ {}
moreover from *chain* **have** *chain'*: *Complete-Partial-Order.chain ordb* (*t* ‘ *Y*)
by(*rule chain-imageI*)(*rule monotoneD*[*OF mono*])

ultimately show *f (luba Y) (t (luba Y)) = ⨆((λx. f x (t x)) ‘ Y)*
by(*simp add*: *contD*[*OF 1*] *contD*[*OF t*] *contD*[*OF 2*] *image-image*)
(*rule diag-Sup*[*OF chain*], *auto intro*: *monotone2monotone*[*OF mono2 mono monotone-const transpI*] *monotoneD*[*OF mono1*])
qed

lemma *mcont2mcont'*:
⟦ ⋀*x*. *mcont lub' ord' Sup (≤) (λy. f x y)*;
⋀*y*. *mcont lub ord Sup (≤) (λx. f x y)*;
mcont lub ord lub' ord' (λy. t y) ⟧
⟹ *mcont lub ord Sup (≤) (λx. f x (t x))*
unfolding *mcont-def* **by**(*blast intro*: *transp-on-le monotone2monotone cont-apply*)

lemma *mcont2mcont*:
⟦*mcont lub' ord' Sup (≤) (λx. f x)*; *mcont lub ord lub' ord' (λx. t x)*⟧
⟹ *mcont lub ord Sup (≤) (λx. f (t x))*
by(*rule mcont2mcont'*[*OF - mcont-const*])

context
fixes *ord* :: *'b ⇒ 'b ⇒ bool* (**infix** ‹⊑› *60*)
and *lub* :: *'b set ⇒ 'b* (‹⋁›)
begin

lemma *cont-fun-lub-Sup*:
assumes *chainM*: *Complete-Partial-Order.chain* (*fun-ord* (≤)) *M*
and *mcont* [*rule-format*]: ∀*f*∈*M*. *mcont lub* (⊑) *Sup* (≤) *f*
shows *cont lub* (⊑) *Sup* (≤) (*fun-lub Sup M*)
proof(*rule contI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain* (⊑) *Y*
and *Y*: *Y* ≠ {}
from *swap-Sup*[*OF chain chainM mcont*[*THEN mcont-mono*]]
show *fun-lub Sup M* (⋁ *Y*) = ⨆(*fun-lub Sup M ‘ Y*)
by(*simp add*: *mcont-contD*[*OF mcont chain Y*] *fun-lub-apply cong*: *image-cong*)
qed

lemma *mcont-fun-lub-Sup*:
⟦ *Complete-Partial-Order.chain* (*fun-ord* (≤)) *M*;
∀*f*∈*M*. *mcont lub ord Sup* (≤) *f* ⟧
⟹ *mcont lub* (⊑) *Sup* (≤) (*fun-lub Sup M*)
by(*simp add*: *mcont-def cont-fun-lub-Sup mono-lub*)

lemma *iterates-mcont*:
assumes *f*: *f* ∈ *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* (≤)) *F*
and *mono*: ⋀*f*. *mcont lub* (⊑) *Sup* (≤) *f* ⟹ *mcont lub* (⊑) *Sup* (≤) (*F f*)
shows *mcont lub* (⊑) *Sup* (≤) *f*
using *f*
by(*induction rule*: *ccpo.iterates.induct*[*OF ccpo-fun*, *consumes 1*, *case-names step Sup*])(*blast intro*: *mono mcont-fun-lub-Sup*)+

lemma *fixp-preserves-mcont*:
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord* ($\leq$)) ($\leq$) ($\lambda f.$ *F f x*)
and *mcont*: $\bigwedge f.$ *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) *f* $\Longrightarrow$ *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) (*F f*)
shows *mcont lub* ($\sqsubseteq$) *Sup* ($\leq$) (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* ($\leq$)) *F*)
(**is** *mcont - - - - ?fixp*)
unfolding *mcont-def*
proof(*intro conjI monotoneI contI*)
have *mono*: *monotone* (*fun-ord* ($\leq$)) (*fun-ord* ($\leq$)) *F*
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def intro*: *monotoneD*[*OF mono*])
let *?iter* = *ccpo.iterates* (*fun-lub Sup*) (*fun-ord* ($\leq$)) *F*
have *chain*: $\bigwedge x.$ *Complete-Partial-Order.chain* ($\leq$) (($\lambda f.$ *f x*) ‘ *?iter*)
by(*rule chain-imageI*[*OF ccpo.chain-iterates*[*OF ccpo-fun mono*]])(*simp add*: *fun-ord-def*)

show *?fixp x* $\leq$ *?fixp y* **if** *x* $\sqsubseteq$ *y* **for** *x y*
proof −
have ($\bigsqcup f{\in}$*?iter*. *f x*)
$\leq$ ($\bigsqcup f{\in}$*?iter*. *f y*)
using *chain*
proof(*rule ccpo-Sup-least*)
fix x'
assume $x' \in$ ($\lambda f.$ *f x*) ‘ *?iter*
then obtain *f* **where** *f*: *f* $\in$ *?iter* $x' = f\ x$ **by** *blast*
then have *f x* $\leq$ *f y*
by (*metis iterates-mcont mcont mcont-monoD that*)
also have *f y* $\leq \bigsqcup$((($\lambda f.$ *f y*) ‘ *?iter*)
using *chain f local.ccpo-Sup-upper* **by** *auto*
finally show $x' \leq \ldots$
using *f*(*2*) **by** *blast*
qed
then show *?thesis*
by (*simp add*: *ccpo.fixp-def*[*OF ccpo-fun*] *fun-lub-apply*)
qed
show *?fixp* ($\bigvee Y$) = $\bigsqcup$(*?fixp* ‘ *Y*)
if *chain*: *Complete-Partial-Order.chain* ($\sqsubseteq$) *Y* **and** *Y*: *Y* $\neq$ {} **for** *Y*
proof −
have *f* ($\bigvee Y$) = $\bigsqcup$(*f* ‘ *Y*) **if** *f* $\in$ *?iter* **for** *f*
using *that mcont chain Y*
by (*rule mcont-contD*[*OF iterates-mcont*])
moreover have $\bigsqcup$((($\lambda f.$ $\bigsqcup$(*f* ‘ *Y*)) ‘ *?iter*) = $\bigsqcup$((($\lambda x.$ $\bigsqcup$((($\lambda f.$ *f x*) ‘ *?iter*)) ‘ *Y*)
using *chain ccpo.chain-iterates*[*OF ccpo-fun mono*]
by (*rule swap-Sup*)(*rule mcont-mono*[*OF iterates-mcont*[*OF - mcont*]])
ultimately show *?thesis*
unfolding *ccpo.fixp-def*[*OF ccpo-fun*]
by (*simp add*: *fun-lub-apply cong*: *image-cong*)
qed
qed

end

context
fixes $F :: 'c \Rightarrow 'c$ **and** $U :: 'c \Rightarrow 'b \Rightarrow 'a$ **and** $C :: ('b \Rightarrow 'a) \Rightarrow 'c$ **and** f
assumes *mono*: $\bigwedge x.$ *monotone* (*fun-ord* $(\leq)$) $(\leq)$ $(\lambda f.\ U\ (F\ (C\ f))\ x)$
and *eq*: $f \equiv C$ (*ccpo.fixp* (*fun-lub Sup*) (*fun-ord* $(\leq)$) $(\lambda f.\ U\ (F\ (C\ f))))$
and *inverse*: $\bigwedge f.\ U\ (C\ f) = f$
begin

lemma *fixp-preserves-mono-uc*:
assumes *mono2*: $\bigwedge f.$ *monotone ord* $(\leq)$ $(U\ f) \Longrightarrow$ *monotone ord* $(\leq)$ $(U\ (F\ f))$
shows *monotone ord* $(\leq)$ $(U\ f)$
using *fixp-preserves-mono*[*OF mono mono2*] **by**(*subst eq*)(*simp add*: *inverse*)

lemma *fixp-preserves-mcont-uc*:
assumes *mcont*: $\bigwedge f.$ *mcont lubb ordb Sup* $(\leq)$ $(U\ f) \Longrightarrow$ *mcont lubb ordb Sup* $(\leq)$ $(U\ (F\ f))$
shows *mcont lubb ordb Sup* $(\leq)$ $(U\ f)$
using *fixp-preserves-mcont*[*OF mono mcont*] **by**(*subst eq*)(*simp add*: *inverse*)

end

lemmas *fixp-preserves-mono1* = *fixp-preserves-mono-uc*[*of* $\lambda x.\ x$ - $\lambda x.\ x$, *OF* - - *refl*]
lemmas *fixp-preserves-mono2* =
fixp-preserves-mono-uc[*of case-prod* - *curry*, *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mono3* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mono4* =
fixp-preserves-mono-uc[*of* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]

lemmas *fixp-preserves-mcont1* = *fixp-preserves-mcont-uc*[*of* $\lambda x.\ x$ - $\lambda x.\ x$, *OF* - - *refl*]
lemmas *fixp-preserves-mcont2* =
fixp-preserves-mcont-uc[*of case-prod* - *curry*, *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mcont3* =
fixp-preserves-mcont-uc[*of* $\lambda f.$ *case-prod* (*case-prod f*) - $\lambda f.$ *curry* (*curry f*), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]
lemmas *fixp-preserves-mcont4* =
fixp-preserves-mcont-uc[*of* $\lambda f.$ *case-prod* (*case-prod* (*case-prod f*)) - $\lambda f.$ *curry* (*curry* (*curry f*)), *unfolded case-prod-curry curry-case-prod*, *OF* - - *refl*]

end

lemma (**in** *preorder*) *monotone-if-bot*:
fixes *bot*
assumes *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ (x \le bound)\ ]\!] \Longrightarrow ord\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow ord\ bot\ (f\ x)\ ord\ bot\ bot$
shows *monotone* $(\le)$ *ord* $(\lambda x.\ if\ x \le bound\ then\ bot\ else\ f\ x)$
by(*rule monotoneI*)(*auto intro*: *bot intro*: *mono order-trans*)

lemma (**in** *ccpo*) *mcont-if-bot*:
fixes *bot* **and** *lub* $(\langle\bigvee\rangle)$ **and** *ord* (**infix** $\langle\sqsubseteq\rangle$ *60*)
assumes *ccpo*: *class.ccpo lub* $(\sqsubseteq)$ *lt*
and *mono*: $\bigwedge x\ y.\ [\![\ x \le y;\ \neg\ x \le bound\ ]\!] \Longrightarrow f\ x \sqsubseteq f\ y$
and *cont*: $\bigwedge Y.\ [\![$ *Complete-Partial-Order.chain* $(\le)\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ x \le bound\ ]\!] \Longrightarrow f\ (\bigsqcup Y) = \bigvee(f\ `\ Y)$
and *bot*: $\bigwedge x.\ \neg\ x \le bound \Longrightarrow bot \sqsubseteq f\ x$
shows *mcont Sup* $(\le)$ *lub* $(\sqsubseteq)$ $(\lambda x.\ if\ x \le bound\ then\ bot\ else\ f\ x)$ (**is** *mcont - - - - ?g*)
proof(*intro mcontI contI*)
interpret *c*: *ccpo lub* $(\sqsubseteq)$ *lt* **by**(*fact ccpo*)
show *monotone* $(\le)$ $(\sqsubseteq)$ *?g* **by**(*rule monotone-if-bot*)(*simp-all add*: *mono bot*)

fix *Y*
assume *chain*: *Complete-Partial-Order.chain* $(\le)$ *Y* **and** *Y*: $Y \neq \{\}$
show *?g* $(\bigsqcup Y) = \bigvee(?g\ `\ Y)$
proof(*cases* $Y \subseteq \{x.\ x \le bound\}$)
case *True*
hence $\bigsqcup Y \le bound$ **using** *chain* **by**(*auto intro*: *ccpo-Sup-least*)
moreover have $Y \cap \{x.\ \neg\ x \le bound\} = \{\}$ **using** *True* **by** *auto*
ultimately show *?thesis* **using** *True Y*
by (*auto simp add*: *image-constant-conv cong del*: *c.SUP-cong-simp*)
next
case *False*
let $?Y = Y \cap \{x.\ \neg\ x \le bound\}$
have *chain'*: *Complete-Partial-Order.chain* $(\le)$ *?Y*
using *chain* **by**(*rule chain-subset*) *simp*

from *False* **obtain** *y* **where** *ybound*: $\neg\ y \le bound$ **and** *y*: $y \in Y$ **by** *blast*
hence $\neg \bigsqcup Y \le bound$ **by** (*metis ccpo-Sup-upper chain order.trans*)
hence *?g* $(\bigsqcup Y) = f\ (\bigsqcup Y)$ **by** *simp*
also have $\bigsqcup Y \le \bigsqcup ?Y$ **using** *chain*
proof(*rule ccpo-Sup-least*)
fix *x*
assume *x*: $x \in Y$
show $x \le \bigsqcup ?Y$
proof(*cases* $x \le bound$)
case *True*
with *chainD*[*OF chain x y*] **have** $x \le y$ **using** *ybound* **by**(*auto intro*: *order-trans*)
thus *?thesis* **by**(*rule order-trans*)(*auto intro*: *ccpo-Sup-upper*[*OF chain'*] *simp add*: *y ybound*)

qed(*auto intro*: *ccpo-Sup-upper*[*OF chain'*] *simp add*: *x*)
qed
hence $\bigsqcup Y = \bigsqcup ?Y$ **by**(*rule order.antisym*)(*blast intro*: *ccpo-Sup-least*[*OF chain'*] *ccpo-Sup-upper*[*OF chain*])
hence $f\ (\bigsqcup Y) = f\ (\bigsqcup ?Y)$ **by** *simp*
also have $f\ (\bigsqcup ?Y) = \bigsqcup(f\ `\ ?Y)$ **using** *chain'* **by**(*rule cont*)(*insert y ybound*, *auto*)
also have $\bigsqcup(f\ `\ ?Y) = \bigsqcup(?g\ `\ Y)$
proof(*cases* $Y \cap \{x.\ x \leq bound\} = \{\}$)
case *True*
hence $f\ `\ ?Y = ?g\ `\ Y$ **by** *auto*
thus *?thesis* **by**(*rule arg-cong*)
next
case *False*
have *chain''*: *Complete-Partial-Order.chain* ($\sqsubseteq$) (*insert bot* ($f\ `\ ?Y$))
using *chain* **by**(*auto intro*!: *chainI bot dest*: *chainD intro*: *mono*)
hence *chain'''*: *Complete-Partial-Order.chain* ($\sqsubseteq$) ($f\ `\ ?Y$) **by**(*rule chain-subset*) *blast*
have *bot* $\sqsubseteq \bigsqcup(f\ `\ ?Y)$ **using** *y ybound* **by**(*blast intro*: *c.order-trans*[*OF bot*] *c.ccpo-Sup-upper*[*OF chain''*])
hence $\bigsqcup(insert\ bot\ (f\ `\ ?Y)) \sqsubseteq \bigsqcup(f\ `\ ?Y)$ **using** *chain''*
by(*auto intro*: *c.ccpo-Sup-least c.ccpo-Sup-upper*[*OF chain''*])
with - **have** $\ldots = \bigsqcup(insert\ bot\ (f\ `\ ?Y))$
by(*rule c.order.antisym*)(*blast intro*: *c.ccpo-Sup-least*[*OF chain''*] *c.ccpo-Sup-upper*[*OF chain'*])
also have *insert bot* ($f\ `\ ?Y$) = $?g\ `\ Y$ **using** *False* **by** *auto*
finally show *?thesis* **.**
qed
finally show *?thesis* **.**
qed
qed

context *partial-function-definitions* **begin**

lemma *mcont-const* [*cont-intro*, *simp*]:
mcont luba orda lub leq ($\lambda x.\ c$)
by(*rule ccpo.mcont-const*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemmas [*cont-intro*, *simp*] =
ccpo.cont-const[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *mono2mono*:
assumes *monotone ordb leq* ($\lambda y.\ f\ y$) *monotone orda ordb* ($\lambda x.\ t\ x$)
shows *monotone orda leq* ($\lambda x.\ f\ (t\ x)$)
using *assms* **by**(*rule monotone2monotone*) *simp-all*

lemmas *mcont2mcont'* = *ccpo.mcont2mcont'*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *mcont2mcont* = *ccpo.mcont2mcont*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemmas *fixp-preserves-mono1 = ccpo.fixp-preserves-mono1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono2 = ccpo.fixp-preserves-mono2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono3 = ccpo.fixp-preserves-mono3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mono4 = ccpo.fixp-preserves-mono4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont1 = ccpo.fixp-preserves-mcont1*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont2 = ccpo.fixp-preserves-mcont2*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont3 = ccpo.fixp-preserves-mcont3*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
lemmas *fixp-preserves-mcont4 = ccpo.fixp-preserves-mcont4*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

lemma *monotone-if-bot*:
fixes *bot*
assumes *g*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)\ \mathit{ord}\ \mathit{bot}\ \mathit{bot}$
shows *monotone leq ord g*
unfolding *g*[*abs-def*] **using** *preorder mono bot* **by**(*rule preorder.monotone-if-bot*)

lemma *mcont-if-bot*:
fixes *bot*
assumes *ccpo*: *class.ccpo lub' ord* (*mk-less ord*)
and *bot*: $\bigwedge x.\ \neg\ \mathit{leq}\ x\ \mathit{bound} \Longrightarrow \mathit{ord}\ \mathit{bot}\ (f\ x)$
and *g*: $\bigwedge x.\ g\ x = (\mathit{if}\ \mathit{leq}\ x\ \mathit{bound}\ \mathit{then}\ \mathit{bot}\ \mathit{else}\ f\ x)$
and *mono*: $\bigwedge x\ y.\ [\![\ \mathit{leq}\ x\ y;\ \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow \mathit{ord}\ (f\ x)\ (f\ y)$
and *cont*: $\bigwedge Y.\ [\![\ \mathit{Complete\text{-}Partial\text{-}Order.chain}\ \mathit{leq}\ Y;\ Y \neq \{\};\ \bigwedge x.\ x \in Y \Longrightarrow \neg\ \mathit{leq}\ x\ \mathit{bound}\ ]\!] \Longrightarrow f\ (\mathit{lub}\ Y) = \mathit{lub'}\ (f\ `\ Y)$
shows *mcont lub leq lub' ord g*
unfolding *g*[*abs-def*] **using** *ccpo mono cont bot* **by**(*rule ccpo.mcont-if-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]])

end

15.2 Admissibility

lemma *admissible-subst*:
assumes *adm*: *ccpo.admissible luba orda* $(\lambda x.\ P\ x)$
and *mcont*: *mcont lubb ordb luba orda f*
shows *ccpo.admissible lubb ordb* $(\lambda x.\ P\ (f\ x))$
using *assms* **by** (*simp add*: *ccpo.admissible-def chain-imageI mcont-contD mcont-monoD*)

lemmas [*simp*, *cont-intro*] =

admissible-all
admissible-ball
admissible-const
admissible-conj

lemma *admissible-disj'* [*simp*, *cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*); *ccpo.admissible lub ord P*; *ccpo.admissible lub ord Q* ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ∨ *Q x*)
by(*rule ccpo.admissible-disj*)

lemma *admissible-imp'* [*cont-intro*]:
⟦ *class.ccpo lub ord* (*mk-less ord*);
ccpo.admissible lub ord (λ*x*. ¬ *P x*);
ccpo.admissible lub ord (λ*x*. *Q x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*)
unfolding *imp-conv-disj* **by**(*rule ccpo.admissible-disj*)

lemma *admissible-imp* [*cont-intro*]:
(*Q* ⟹ *ccpo.admissible lub ord* (λ*x*. *P x*))
⟹ *ccpo.admissible lub ord* (λ*x*. *Q* ⟶ *P x*)
by(*rule ccpo.admissibleI*)(*auto dest*: *ccpo.admissibleD*)

lemma *admissible-not-mem'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-not-mem*: *ccpo.admissible Union* (⊆) (λ*A*. *x* ∉ *A*)
by(*rule ccpo.admissibleI*) *auto*

lemma *admissible-eqI*:
assumes *f*: *cont luba orda lub ord* (λ*x*. *f x*)
and *g*: *cont luba orda lub ord* (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
by (*smt* (*verit*, *best*) *Sup.SUP-cong ccpo.admissible-def contD assms*)

corollary *admissible-eq-mcontI* [*cont-intro*]:
⟦ *mcont luba orda lub ord* (λ*x*. *f x*);
mcont luba orda lub ord (λ*x*. *g x*) ⟧
⟹ *ccpo.admissible luba orda* (λ*x*. *f x* = *g x*)
by(*rule admissible-eqI*)(*auto simp add*: *mcont-def*)

lemma *admissible-iff* [*cont-intro*, *simp*]:
⟦ *ccpo.admissible lub ord* (λ*x*. *P x* ⟶ *Q x*); *ccpo.admissible lub ord* (λ*x*. *Q x* ⟶ *P x*) ⟧
⟹ *ccpo.admissible lub ord* (λ*x*. *P x* ⟷ *Q x*)
by(*subst iff-conv-conj-imp*)(*rule admissible-conj*)

context *ccpo* **begin**

lemma *admissible-leI*:
assumes *f*: *mcont luba orda Sup* (≤) (λ*x*. *f x*)

and *g*: *mcont luba orda Sup* (≤) (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* ≤ *g x*)
proof(*rule ccpo.admissibleI*)
fix *A*
assume *chain*: *Complete-Partial-Order.chain orda A*
and *le*: ∀ *x*∈*A*. *f x* ≤ *g x*
and *False*: *A* ≠ {}
have *f* (*luba A*) = ⨆(*f* ‘ *A*) **by**(*simp add*: *mcont-contD*[*OF f*] *chain False*)
also have ... ≤ ⨆(*g* ‘ *A*)
proof(*rule ccpo-Sup-least*)
from *chain* **show** *Complete-Partial-Order.chain* (≤) (*f* ‘ *A*)
by(*rule chain-imageI*)(*rule mcont-monoD*[*OF f*])
fix *x*
assume *x* ∈ *f* ‘ *A*
then obtain *y* **where** *y* ∈ *A x* = *f y* **by** *blast* **note** *this*(*2*)
also have *f y* ≤ *g y* **using** *le* ‹*y* ∈ *A*› **by** *simp*
also have *Complete-Partial-Order.chain* (≤) (*g* ‘ *A*)
using *chain* **by**(*rule chain-imageI*)(*rule mcont-monoD*[*OF g*])
hence *g y* ≤ ⨆(*g* ‘ *A*) **by**(*rule ccpo-Sup-upper*)(*simp add*: ‹*y* ∈ *A*›)
finally show *x* ≤ ... **.**
qed
also have ... = *g* (*luba A*) **by**(*simp add*: *mcont-contD*[*OF g*] *chain False*)
finally show *f* (*luba A*) ≤ *g* (*luba A*) **.**
qed

end

lemma *admissible-leI*:
fixes *ord* (**infix** ‹⊑› *60*) **and** *lub* (‹⋁›)
assumes *class.ccpo lub* (⊑) (*mk-less* (⊑))
and *mcont luba orda lub* (⊑) (λ*x*. *f x*)
and *mcont luba orda lub* (⊑) (λ*x*. *g x*)
shows *ccpo.admissible luba orda* (λ*x*. *f x* ⊑ *g x*)
using *assms* **by**(*rule ccpo.admissible-leI*)

declare *ccpo-class.admissible-leI*[*cont-intro*]

context *ccpo* **begin**

lemma *admissible-not-below*: *ccpo.admissible Sup* (≤) (λ*x*. ¬ (≤) *x y*)
by(*rule ccpo.admissibleI*)(*simp add*: *ccpo-Sup-below-iff*)

end

lemma (**in** *preorder*) *preorder* [*cont-intro*, *simp*]: *class.preorder* (≤) (*mk-less* (≤))
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *order-trans*)

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] =
admissible-leI[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]
ccpo.admissible-not-below[*THEN admissible-subst*, *OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

end

setup ‹*Sign.map-naming* (*Name-Space.mandatory-path ccpo*)›

inductive *compact* :: ('*a set* ⇒ '*a*) ⇒ ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a* ⇒ *bool*
for *lub ord x*
where *compact*:
⟦ *ccpo.admissible lub ord* (λ*y*. ¬ *ord x y*);
ccpo.admissible lub ord (λ*y*. *x* ≠ *y*) ⟧
⟹ *compact lub ord x*

setup ‹*Sign.map-naming Name-Space.parent-path*›

context *ccpo* **begin**

lemma *compactI*:
assumes *ccpo.admissible Sup* (≤) (λ*y*. ¬ *x* ≤ *y*)
shows *ccpo.compact Sup* (≤) *x*
using *assms*
proof(*rule ccpo.compact.intros*)
have *neq*: (λ*y*. *x* ≠ *y*) = (λ*y*. ¬ *x* ≤ *y* ∨ ¬ *y* ≤ *x*) **by**(*auto*)
show *ccpo.admissible Sup* (≤) (λ*y*. *x* ≠ *y*)
by(*subst neq*)(*rule admissible-disj admissible-not-below assms*)+
qed

lemma *compact-bot*:
assumes *x* = *Sup* {}
shows *ccpo.compact Sup* (≤) *x*
proof(*rule compactI*)
show *ccpo.admissible Sup* (≤) (λ*y*. ¬ *x* ≤ *y*) **using** *assms*
by(*auto intro*!: *ccpo.admissibleI intro*: *ccpo-Sup-least chain-empty*)
qed

end

lemma *admissible-compact-neq'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-compact-neq*: *ccpo.compact lub ord k* ⟹ *ccpo.admissible lub ord* (λ*x*. *k* ≠ *x*)
by(*simp add*: *ccpo.compact.simps*)

lemma *admissible-neq-compact'* [*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-neq-compact*: *ccpo.compact lub ord k* ⟹ *ccpo.admissible lub ord* (λ*x*. *x* ≠ *k*)
by(*subst eq-commute*)(*rule admissible-compact-neq*)

context *partial-function-definitions* **begin**

lemmas [*cont-intro*, *simp*] = *ccpo.compact-bot*[*OF Partial-Function.ccpo*[*OF partial-function-definitions-axioms*]]

end

context *ccpo* **begin**

lemma *fixp-strong-induct*:
  assumes [*cont-intro*]: *ccpo.admissible Sup* (≤) *P*
  and *mono*: *monotone* (≤) (≤) *f*
  and *bot*: *P* (⨆{})
  and *step*: ⋀*x*. ⟦ *x* ≤ *ccpo-class.fixp f*; *P x* ⟧ ⟹ *P* (*f x*)
  shows *P* (*ccpo-class.fixp f*)
proof(*rule fixp-induct*[**where** *P*=λ*x*. *x* ≤ *ccpo-class.fixp f* ∧ *P x*, *THEN conjunct2*])
  note [*cont-intro*] = *admissible-leI*
  show *ccpo.admissible Sup* (≤) (λ*x*. *x* ≤ *ccpo-class.fixp f* ∧ *P x*) **by** *simp*
next
  show ⨆{} ≤ *ccpo-class.fixp f* ∧ *P* (⨆{})
    by(*auto simp add*: *bot intro*: *ccpo-Sup-least chain-empty*)
next
  fix *x*
  assume *x* ≤ *ccpo-class.fixp f* ∧ *P x*
  thus *f x* ≤ *ccpo-class.fixp f* ∧ *P* (*f x*)
    by(*subst fixp-unfold*[*OF mono*])(*auto dest*: *monotoneD*[*OF mono*] *intro*: *step*)
qed(*rule mono*)

end

context *partial-function-definitions* **begin**

lemma *fixp-strong-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'*); *le-fun* (*U f'*) (*U f*) ⟧ ⟹ *P* (*U* (*F f'*))
  shows *P* (*U f*)
  unfolding *eq inverse*
  apply (*rule ccpo.fixp-strong-induct*[*OF ccpo adm*])
    apply (*insert mono*, *auto simp*: *monotone-def fun-ord-def bot fun-lub-def*)[*2*]

apply (*rule-tac f'5=C x* **in** *step*)
apply (*simp-all add*: *inverse eq*)
done

end

15.3 (=) as order

definition *lub-singleton* :: ('*a set* ⇒ '*a*) ⇒ *bool*
where *lub-singleton lub* ⟷ (∀ *a*. *lub* {*a*} = *a*)

definition *the-Sup* :: '*a set* ⇒ '*a*
where *the-Sup A* = (*THE a*. *a* ∈ *A*)

lemma *lub-singleton-the-Sup* [*cont-intro*, *simp*]: *lub-singleton the-Sup*
by(*simp add*: *lub-singleton-def the-Sup-def*)

lemma (**in** *ccpo*) *lub-singleton*: *lub-singleton Sup*
by(*simp add*: *lub-singleton-def*)

lemma (**in** *partial-function-definitions*) *lub-singleton* [*cont-intro*, *simp*]: *lub-singleton lub*
by(*rule ccpo.lub-singleton*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemma *preorder-eq* [*cont-intro*, *simp*]:
class.preorder (=) (*mk-less* (=))
by(*unfold-locales*)(*simp-all add*: *mk-less-def*)

lemma *monotone-eqI* [*cont-intro*]:
assumes *class.preorder ord* (*mk-less ord*)
shows *monotone* (=) *ord f*
proof −
interpret *preorder ord mk-less ord* **by** *fact*
show *?thesis* **by**(*simp add*: *monotone-def*)
qed

lemma *cont-eqI* [*cont-intro*]:
fixes *f* :: '*a* ⇒ '*b*
assumes *lub-singleton lub*
shows *cont the-Sup* (=) *lub ord f*
proof(*rule contI*)
fix *Y* :: '*a set*
assume *Complete-Partial-Order.chain* (=) *Y Y* ≠ {}
then obtain *a* **where** *Y* = {*a*} **by**(*auto simp add*: *chain-def*)
thus *f* (*the-Sup Y*) = *lub* (*f* ' *Y*) **using** *assms*
by(*simp add*: *the-Sup-def lub-singleton-def*)
qed

lemma *mcont-eqI* [*cont-intro*, *simp*]:

⟦ *class.preorder ord* (*mk-less ord*); *lub-singleton lub* ⟧
⟹ *mcont the-Sup* (=) *lub ord f*
by(*simp add*: *mcont-def cont-eqI monotone-eqI*)

15.4 ccpo for products

definition *prod-lub* :: ('*a set* ⇒ '*a*) ⇒ ('*b set* ⇒ '*b*) ⇒ ('*a* × '*b*) *set* ⇒ '*a* × '*b*
where *prod-lub Sup-a Sup-b Y* = (*Sup-a* (*fst* ‘ *Y*), *Sup-b* (*snd* ‘ *Y*))

lemma *lub-singleton-prod-lub* [*cont-intro*, *simp*]:
⟦ *lub-singleton luba*; *lub-singleton lubb* ⟧ ⟹ *lub-singleton* (*prod-lub luba lubb*)
by(*simp add*: *lub-singleton-def prod-lub-def*)

lemma *prod-lub-empty* [*simp*]: *prod-lub luba lubb* {} = (*luba* {}, *lubb* {})
by(*simp add*: *prod-lub-def*)

lemma *preorder-rel-prodI* [*cont-intro*, *simp*]:
assumes *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *class.preorder* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
proof −
interpret *a*: *preorder orda mk-less orda* **by** *fact*
interpret *b*: *preorder ordb mk-less ordb* **by** *fact*
show *?thesis* **by**(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *a.order-trans b.order-trans*)
qed

lemma *order-rel-prodI*:
assumes *a*: *class.order orda* (*mk-less orda*)
and *b*: *class.order ordb* (*mk-less ordb*)
shows *class.order* (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.order ?ord ?ord'*)
proof(*intro class.order.intro class.order-axioms.intro*)
interpret *a*: *order orda mk-less orda* **by**(*fact a*)
interpret *b*: *order ordb mk-less ordb* **by**(*fact b*)
show *class.preorder ?ord ?ord'* **by**(*rule preorder-rel-prodI*) *unfold-locales*

fix *x y*
assume *?ord x y ?ord y x*
thus *x* = *y* **by**(*cases x y rule*: *prod.exhaust*[*case-product prod.exhaust*]) *auto*
qed

lemma *monotone-rel-prodI*:
assumes *mono2*: ⋀*a*. *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))
and *mono1*: ⋀*b*. *monotone orda ordc* (λ*a*. *f* (*a*, *b*))
and *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)
and *c*: *class.preorder ordc* (*mk-less ordc*)
shows *monotone* (*rel-prod orda ordb*) *ordc f*

proof −
interpret *a*: *preorder orda mk-less orda* **by**(*rule a*)
interpret *b*: *preorder ordb mk-less ordb* **by**(*rule b*)
interpret *c*: *preorder ordc mk-less ordc* **by**(*rule c*)
show *?thesis* **using** *mono2 mono1*
by(*auto 7 2 simp add*: *monotone-def intro*: *c.order-trans*)
qed

lemma *monotone-rel-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (λ*a*. *f* (*a*, *b*))
proof −
interpret *preorder ordb mk-less ordb* **by**(*rule preorder*)
show *?thesis* **using** *mono* **by**(*simp add*: *monotone-def*)
qed

lemma *monotone-rel-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc f*
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))
proof −
interpret *preorder orda mk-less orda* **by**(*rule preorder*)
show *?thesis* **using** *mono* **by**(*simp add*: *monotone-def*)
qed

lemma *monotone-case-prodI*:
⟦ ⋀*a*. *monotone ordb ordc* (*f a*); ⋀*b*. *monotone orda ordc* (λ*a*. *f a b*);
class.preorder orda (*mk-less orda*); *class.preorder ordb* (*mk-less ordb*);
class.preorder ordc (*mk-less ordc*) ⟧
⟹ *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
by(*rule monotone-rel-prodI*) *simp-all*

lemma *monotone-case-prodD1*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder ordb* (*mk-less ordb*)
shows *monotone orda ordc* (λ*a*. *f a b*)
using *monotone-rel-prodD1*[*OF assms*] **by** *simp*

lemma *monotone-case-prodD2*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) *ordc* (*case-prod f*)
and *preorder*: *class.preorder orda* (*mk-less orda*)
shows *monotone ordb ordc* (*f a*)
using *monotone-rel-prodD2*[*OF assms*] **by** *simp*

context
fixes *orda ordb ordc*
assumes *a*: *class.preorder orda* (*mk-less orda*)
and *b*: *class.preorder ordb* (*mk-less ordb*)

and *c*: *class.preorder ordc* (*mk-less ordc*)
begin

lemma *monotone-rel-prod-iff*:
monotone (*rel-prod orda ordb*) *ordc f* ⟷
(∀ *a*. *monotone ordb ordc* (λ*b*. *f* (*a*, *b*))) ∧
(∀ *b*. *monotone orda ordc* (λ*a*. *f* (*a*, *b*)))
using *a b c* **by**(*blast intro*: *monotone-rel-prodI dest*: *monotone-rel-prodD1 monotone-rel-prodD2*)

lemma *monotone-case-prod-iff* [*simp*]:
monotone (*rel-prod orda ordb*) *ordc* (*case-prod f*) ⟷
(∀ *a*. *monotone ordb ordc* (*f a*)) ∧ (∀ *b*. *monotone orda ordc* (λ*a*. *f a b*))
by(*simp add*: *monotone-rel-prod-iff*)

end

lemma *monotone-case-prod-apply-iff*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
by(*simp add*: *monotone-def*)

lemma *monotone-case-prod-applyD*:
monotone orda ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *monotone orda ordb* (*case-prod* (λ*a b*. *f a b y*))
by(*simp add*: *monotone-case-prod-apply-iff*)

lemma *monotone-case-prod-applyI*:
monotone orda ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *monotone orda ordb* (λ*x*. (*case-prod f x*) *y*)
by(*simp add*: *monotone-case-prod-apply-iff*)

lemma *cont-case-prod-apply-iff*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
by(*simp add*: *cont-def split-def*)

lemma *cont-case-prod-applyI*:
cont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
⟹ *cont luba orda lubb ordb* (λ*x*. (*case-prod f x*) *y*)
by(*simp add*: *cont-case-prod-apply-iff*)

lemma *cont-case-prod-applyD*:
cont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*)
⟹ *cont luba orda lubb ordb* (*case-prod* (λ*a b*. *f a b y*))
by(*simp add*: *cont-case-prod-apply-iff*)

lemma *mcont-case-prod-apply-iff* [*simp*]:

mcont luba orda lubb ordb (λ*x*. (*case-prod f x*) *y*) ⟷
mcont luba orda lubb ordb (*case-prod* (λ*a b*. *f a b y*))
by(*simp add*: *mcont-def monotone-case-prod-apply-iff cont-case-prod-apply-iff*)

lemma *cont-prodD1*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder orda* (*mk-less orda*)
and *luba*: *lub-singleton luba*
shows *cont lubb ordb lubc ordc* (λ*y*. *f* (*x*, *y*))
proof(*rule contI*)
interpret *preorder orda mk-less orda* **by** *fact*

fix *Y* :: *'b set*
let *?Y* = {*x*} × *Y*
assume *Complete-Partial-Order.chain ordb Y Y* ≠ {}
hence *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *?Y ?Y* ≠ {}
by(*simp-all add*: *chain-def*)
with *cont* **have** *f* (*prod-lub luba lubb ?Y*) = *lubc* (*f* ‘ *?Y*) **by**(*rule contD*)
moreover have *f* ‘ *?Y* = (λ*y*. *f* (*x*, *y*)) ‘ *Y* **by** *auto*
ultimately show *f* (*x*, *lubb Y*) = *lubc* ((λ*y*. *f* (*x*, *y*)) ‘ *Y*) **using** *luba*
by(*simp add*: *prod-lub-def* ‹*Y* ≠ {}› *lub-singleton-def*)
qed

lemma *cont-prodD2*:
assumes *cont*: *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc f*
and *class.preorder ordb* (*mk-less ordb*)
and *lubb*: *lub-singleton lubb*
shows *cont luba orda lubc ordc* (λ*x*. *f* (*x*, *y*))
proof(*rule contI*)
interpret *preorder ordb mk-less ordb* **by** *fact*

fix *Y*
assume *Y*: *Complete-Partial-Order.chain orda Y Y* ≠ {}
let *?Y* = *Y* × {*y*}
have *f* (*luba Y*, *y*) = *f* (*prod-lub luba lubb ?Y*)
using *lubb* **by**(*simp add*: *prod-lub-def Y lub-singleton-def*)
also from *Y* **have** *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *?Y ?Y* ≠
{}
by(*simp-all add*: *chain-def*)
with *cont* **have** *f* (*prod-lub luba lubb ?Y*) = *lubc* (*f* ‘ *?Y*) **by**(*rule contD*)
also have *f* ‘ *?Y* = (λ*x*. *f* (*x*, *y*)) ‘ *Y* **by** *auto*
finally show *f* (*luba Y*, *y*) = *lubc* ... **.**
qed

lemma *cont-case-prodD1*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder orda* (*mk-less orda*)
and *lub-singleton luba*
shows *cont lubb ordb lubc ordc* (*f x*)

using *cont-prodD1*[*OF assms*] **by** *simp*

lemma *cont-case-prodD2*:
assumes *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubc ordc* (*case-prod f*)
and *class.preorder ordb* (*mk-less ordb*)
and *lub-singleton lubb*
shows *cont luba orda lubc ordc* (λ*x*. *f x y*)
using *cont-prodD2*[*OF assms*] **by** *simp*

context *ccpo* **begin**

lemma *cont-prodI*:
assumes *mono*: *monotone* (*rel-prod orda ordb*) (≤) *f*
and *cont1*: ⋀*x*. *cont lubb ordb Sup* (≤) (λ*y*. *f* (*x*, *y*))
and *cont2*: ⋀*y*. *cont luba orda Sup* (≤) (λ*x*. *f* (*x*, *y*))
and *class.preorder orda* (*mk-less orda*)
and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* (≤) *f*
proof(*rule contI*)
interpret *a*: *preorder orda mk-less orda* **by** *fact*
interpret *b*: *preorder ordb mk-less ordb* **by** *fact*

fix *Y*
assume *chain*: *Complete-Partial-Order.chain* (*rel-prod orda ordb*) *Y*
and *Y* ≠ {}
have *f* (*prod-lub luba lubb Y*) = *f* (*luba* (*fst* ‘ *Y*), *lubb* (*snd* ‘ *Y*))
by(*simp add*: *prod-lub-def*)
also from *cont2* **have** *f* (*luba* (*fst* ‘ *Y*), *lubb* (*snd* ‘ *Y*)) = ⨆((λ*x*. *f* (*x*, *lubb* (*snd* ‘ *Y*))) ‘ *fst* ‘ *Y*)
by(*rule contD*)(*simp-all add*: *chain-rel-prodD1*[*OF chain*] ‹*Y* ≠ {}›)
also from *cont1* **have** ⋀*x*. *f* (*x*, *lubb* (*snd* ‘ *Y*)) = ⨆((λ*y*. *f* (*x*, *y*)) ‘ *snd* ‘ *Y*)
by(*rule contD*)(*simp-all add*: *chain-rel-prodD2*[*OF chain*] ‹*Y* ≠ {}›)
hence ⨆((λ*x*. *f* (*x*, *lubb* (*snd* ‘ *Y*))) ‘ *fst* ‘ *Y*) = ⨆((λ*x*. … *x*) ‘ *fst* ‘ *Y*) **by** *simp*
also have … = ⨆((λ*x*. *f* (*fst x*, *snd x*)) ‘ *Y*)
unfolding *image-image* **using** *chain*
proof (*rule diag-Sup*)
show ⋀*y*. *y* ∈ *Y* ⟹ *monotone* (*rel-prod orda ordb*) (≤) (λ*x*. *f* (*fst x*, *snd y*))
by (*smt* (*verit*, *best*) *b.order-refl mono monotoneD monotoneI rel-prod-inject rel-prod-sel*)
qed (*use mono monotoneD* **in** *fastforce*)
finally show *f* (*prod-lub luba lubb Y*) = ⨆(*f* ‘ *Y*) **by** *simp*
qed

lemma *cont-case-prodI*:
assumes *monotone* (*rel-prod orda ordb*) (≤) (*case-prod f*)
and ⋀*x*. *cont lubb ordb Sup* (≤) (λ*y*. *f x y*)
and ⋀*y*. *cont luba orda Sup* (≤) (λ*x*. *f x y*)
and *class.preorder orda* (*mk-less orda*)

and *class.preorder ordb* (*mk-less ordb*)
shows *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* (≤) (*case-prod f*)
by(*rule cont-prodI*)(*simp-all add*: *assms*)

lemma *cont-case-prod-iff*:
⟦ *monotone* (*rel-prod orda ordb*) (≤) (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *Sup* (≤) (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb Sup* (≤) (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda Sup* (≤) (λ*x*. *f x y*))
by(*blast dest*: *cont-case-prodD1 cont-case-prodD2 intro*: *cont-case-prodI*)

end

context *partial-function-definitions* **begin**

lemma *mono2mono2*:
assumes *f*: *monotone* (*rel-prod ordb ordc*) *leq* (λ(*x*, *y*). *f x y*)
and *t*: *monotone orda ordb* (λ*x*. *t x*)
and *t'*: *monotone orda ordc* (λ*x*. *t' x*)
shows *monotone orda leq* (λ*x*. *f* (*t x*) (*t' x*))
by (*metis* (*mono-tags*, *lifting*) *case-prod-conv monotoneD monotoneI rel-prod.intros* *assms*)

lemma *cont-case-prodI* [*cont-intro*]:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
⋀*x*. *cont lubb ordb lub leq* (λ*y*. *f x y*);
⋀*y*. *cont luba orda lub leq* (λ*x*. *f x y*);
class.preorder orda (*mk-less orda*);
class.preorder ordb (*mk-less ordb*) ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*)
by(*rule ccpo.cont-case-prodI*)(*rule Partial-Function.ccpo*[*OF partial-function-definitions-axioms*])

lemma *cont-case-prod-iff*:
⟦ *monotone* (*rel-prod orda ordb*) *leq* (*case-prod f*);
class.preorder orda (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *cont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *cont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *cont luba orda lub leq* (λ*x*. *f x y*))
by(*blast dest*: *cont-case-prodD1 cont-case-prodD2 intro*: *cont-case-prodI*)

lemma *mcont-case-prod-iff* [*simp*]:
⟦ *class.preorder orda* (*mk-less orda*); *lub-singleton luba*;
class.preorder ordb (*mk-less ordb*); *lub-singleton lubb* ⟧
⟹ *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lub leq* (*case-prod f*) ⟷
(∀ *x*. *mcont lubb ordb lub leq* (λ*y*. *f x y*)) ∧ (∀ *y*. *mcont luba orda lub leq* (λ*x*. *f x y*))
unfolding *mcont-def* **by**(*auto simp add*: *cont-case-prod-iff*)

end

lemma *mono2mono-case-prod* [*cont-intro*]:
assumes $\bigwedge x\ y.$ *monotone orda ordb* $(\lambda f.$ *pair f x y*$)$
shows *monotone orda ordb* $(\lambda f.$ *case-prod* $($*pair f*$)$ *x*$)$
by(*rule monotoneI*)(*auto split*: *prod.split dest*: *monotoneD*[*OF assms*])

15.5 Complete lattices as ccpo

context *complete-lattice* **begin**

lemma *complete-lattice-ccpo*: *class.ccpo Sup* $(\leq)$ $(<)$
by(*unfold-locales*)(*fast intro*: *Sup-upper Sup-least*)+

lemma *complete-lattice-ccpo'*: *class.ccpo Sup* $(\leq)$ $($*mk-less* $(\leq))$
by(*unfold-locales*)(*auto simp add*: *mk-less-def intro*: *Sup-upper Sup-least*)

lemma *complete-lattice-partial-function-definitions*:
partial-function-definitions $(\leq)$ *Sup*
by(*unfold-locales*)(*auto intro*: *Sup-least Sup-upper*)

lemma *complete-lattice-partial-function-definitions-dual*:
partial-function-definitions $(\geq)$ *Inf*
by(*unfold-locales*)(*auto intro*: *Inf-lower Inf-greatest*)

lemmas [*cont-intro*, *simp*] =
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions*]
Partial-Function.ccpo[*OF complete-lattice-partial-function-definitions-dual*]

lemma *mono2mono-inf*:
assumes *f*: *monotone ord* $(\leq)$ $(\lambda x.$ *f x*$)$
and *g*: *monotone ord* $(\leq)$ $(\lambda x.$ *g x*$)$
shows *monotone ord* $(\leq)$ $(\lambda x.$ *f x* $\sqcap$ *g x*$)$
by(*auto 4 3 dest*: *monotoneD*[*OF f*] *monotoneD*[*OF g*] *intro*: *le-infI1 le-infI2 intro*!: *monotoneI*)

lemma *mcont-const* [*simp*]: *mcont lub ord Sup* $(\leq)$ $(\lambda$-. *c*$)$
by(*rule ccpo.mcont-const*[*OF complete-lattice-ccpo*])

lemma *mono2mono-sup*:
assumes *f*: *monotone ord* $(\leq)$ $(\lambda x.$ *f x*$)$
and *g*: *monotone ord* $(\leq)$ $(\lambda x.$ *g x*$)$
shows *monotone ord* $(\leq)$ $(\lambda x.$ *f x* $\sqcup$ *g x*$)$
by(*auto 4 3 intro*!: *monotoneI intro*: *sup.coboundedI1 sup.coboundedI2 dest*: *monotoneD*[*OF f*] *monotoneD*[*OF g*])

lemma *Sup-image-sup*:
assumes $Y \neq \{\}$

shows $\bigsqcup((\sqcup)\ x\ `\ Y) = x \sqcup \bigsqcup Y$
proof(*rule Sup-eqI*)
fix *y*
assume $y \in (\sqcup)\ x\ `\ Y$
then obtain *z* **where** $y = x \sqcup z$ **and** $z \in Y$ **by** *blast*
from ‹$z \in Y$› **have** $z \leq \bigsqcup Y$ **by**(*rule Sup-upper*)
with - **show** $y \leq x \sqcup \bigsqcup Y$ **unfolding** ‹$y = x \sqcup z$› **by**(*rule sup-mono*) *simp*
next
fix *y*
assume *upper*: $\bigwedge z.\ z \in (\sqcup)\ x\ `\ Y \Longrightarrow z \leq y$
show $x \sqcup \bigsqcup Y \leq y$ **unfolding** *Sup-insert*[*symmetric*]
proof(*rule Sup-least*)
fix *z*
assume $z \in$ *insert x Y*
from *assms* **obtain** z' **where** $z' \in Y$ **by** *blast*
let *?z* = *if* $z \in Y$ *then* $x \sqcup z$ *else* $x \sqcup z'$
have $z \leq x \sqcup$ *?z* **using** ‹$z' \in Y$› ‹$z \in$ *insert x Y*› **by** *auto*
also have $\ldots \leq y$ **by**(*rule upper*)(*auto split*: *if-split-asm intro*: ‹$z' \in Y$›)
finally show $z \leq y$ **.**
qed
qed

lemma *mcont-sup1*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ x \sqcup y)$
by(*auto 4 3 simp add*: *mcont-def sup.coboundedI1 sup.coboundedI2 intro*!: *monotoneI contI intro*: *Sup-image-sup*[*symmetric*])

lemma *mcont-sup2*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda x.\ x \sqcup y)$
by(*subst sup-commute*)(*rule mcont-sup1*)

lemma *mcont2mcont-sup* [*cont-intro*, *simp*]:
⟦ *mcont lub ord Sup* $(\leq)$ $(\lambda x.\ f\ x)$;
mcont lub ord Sup $(\leq)$ $(\lambda x.\ g\ x)$ ⟧
$\Longrightarrow$ *mcont lub ord Sup* $(\leq)$ $(\lambda x.\ f\ x \sqcup g\ x)$
by(*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *mcont-sup1 mcont-sup2 ccpo.mcont-const*[*OF complete-lattice-ccpo*])

end

lemmas [*cont-intro*] = *admissible-leI*[*OF complete-lattice-ccpo'*]

context *complete-distrib-lattice* **begin**

lemma *mcont-inf1*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda y.\ x \sqcap y)$
by(*auto intro*: *monotoneI contI simp add*: *le-infI2 inf-Sup mcont-def*)

lemma *mcont-inf2*: *mcont Sup* $(\leq)$ *Sup* $(\leq)$ $(\lambda x.\ x \sqcap y)$
by(*auto intro*: *monotoneI contI simp add*: *le-infI1 Sup-inf mcont-def*)

lemma *mcont2mcont-inf* [*cont-intro*, *simp*]:

⟦ *mcont lub ord Sup* (≤) (λ*x*. *f x*);
mcont lub ord Sup (≤) (λ*x*. *g x*) ⟧
⟹ *mcont lub ord Sup* (≤) (λ*x*. *f x* ⊓ *g x*)
by(*best intro*: *ccpo.mcont2mcont'*[*OF complete-lattice-ccpo*] *mcont-inf1 mcont-inf2 ccpo.mcont-const*[*OF complete-lattice-ccpo*])

end

interpretation *lfp*: *partial-function-definitions* (≤) :: - :: *complete-lattice* ⇒ - *Sup*
by(*rule complete-lattice-partial-function-definitions*)

declaration ‹*Partial-Function.init lfp* ***term*** ‹*lfp.fixp-fun*› ***term*** ‹*lfp.mono-body*›
@{*thm lfp.fixp-rule-uc*} @{*thm lfp.fixp-induct-uc*} *NONE*›

interpretation *gfp*: *partial-function-definitions* (≥) :: - :: *complete-lattice* ⇒ - *Inf*
by(*rule complete-lattice-partial-function-definitions-dual*)

declaration ‹*Partial-Function.init gfp* ***term*** ‹*gfp.fixp-fun*› ***term*** ‹*gfp.mono-body*›
@{*thm gfp.fixp-rule-uc*} @{*thm gfp.fixp-induct-uc*} *NONE*›

lemma *insert-mono* [*partial-function-mono*]:
monotone (*fun-ord* (⊆)) (⊆) *A* ⟹ *monotone* (*fun-ord* (⊆)) (⊆) (λ*y*. *insert x* (*A y*))
by(*rule monotoneI*)(*auto simp add*: *fun-ord-def dest*: *monotoneD*)

lemma *mono2mono-insert* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-insert*: *monotone* (⊆) (⊆) (*insert x*)
by(*rule monotoneI*) *blast*

lemma *mcont2mcont-insert*[*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-insert*: *mcont Union* (⊆) *Union* (⊆) (*insert x*)
by(*blast intro*: *mcontI contI monotone-insert*)

lemma *mono2mono-image* [*THEN lfp.mono2mono*, *cont-intro*, *simp*]:
shows *monotone-image*: *monotone* (⊆) (⊆) ((`) *f*)
by (*simp add*: *image-mono monoI*)

lemma *cont-image*: *cont Union* (⊆) *Union* (⊆) ((`) *f*)
by (*meson contI image-Union*)

lemma *mcont2mcont-image* [*THEN lfp.mcont2mcont*, *cont-intro*, *simp*]:
shows *mcont-image*: *mcont Union* (⊆) *Union* (⊆) ((`) *f*)
by(*blast intro*: *mcontI monotone-image cont-image*)

context *complete-lattice* **begin**

lemma *monotone-Sup* [*cont-intro*, *simp*]:
monotone ord (⊆) *f* ⟹ *monotone ord* (≤) (λ*x*. ⨆*f x*)
by(*blast intro*: *monotoneI Sup-least Sup-upper dest*: *monotoneD*)

lemma *cont-Sup*:
assumes *cont lub ord Union* (⊆) *f*
shows *cont lub ord Sup* (≤) (λ*x*. ⨆*f x*)
proof −
have ⋀*Y*. ⟦*Complete-Partial-Order.chain ord Y*; *Y* ≠ {}⟧
⟹ ⨆ ⋃ (*f* ‘ *Y*) = (⨆*x*∈*Y*. ⨆ *f x*)
by (*blast intro*: *Sup-least Sup-upper order-trans order.antisym*)
with *assms* **show** *?thesis*
by (*force simp*: *cont-def*)
qed

lemma *mcont-Sup*: *mcont lub ord Union* (⊆) *f* ⟹ *mcont lub ord Sup* (≤) (λ*x*. ⨆*f x*)
unfolding *mcont-def* **by**(*blast intro*: *monotone-Sup cont-Sup*)

lemma *monotone-SUP*:
⟦ *monotone ord* (⊆) *f*; ⋀*y*. *monotone ord* (≤) (λ*x*. *g x y*) ⟧ ⟹ *monotone ord* (≤) (λ*x*. ⨆*y*∈*f x*. *g x y*)
by(*rule monotoneI*)(*blast dest*: *monotoneD intro*: *Sup-upper order-trans intro*!: *Sup-least*)

lemma *monotone-SUP2*:
(⋀*y*. *y* ∈ *A* ⟹ *monotone ord* (≤) (λ*x*. *g x y*)) ⟹ *monotone ord* (≤) (λ*x*. ⨆*y*∈*A*. *g x y*)
by(*rule monotoneI*)(*blast intro*: *Sup-upper order-trans dest*: *monotoneD intro*!: *Sup-least*)

lemma *cont-SUP*:
assumes *f*: *mcont lub ord Union* (⊆) *f*
and *g*: ⋀*y*. *mcont lub ord Sup* (≤) (λ*x*. *g x y*)
shows *cont lub ord Sup* (≤) (λ*x*. ⨆*y*∈*f x*. *g x y*)
proof(*rule contI*)
fix *Y*
assume *chain*: *Complete-Partial-Order.chain ord Y*
and *Y*: *Y* ≠ {}
show ⨆(*g* (*lub Y*) ‘ *f* (*lub Y*)) = ⨆((λ*x*. ⨆(*g x* ‘ *f x*)) ‘ *Y*) (**is** *?lhs* = *?rhs*)
proof(*rule order.antisym*)
show *?lhs* ≤ *?rhs*
proof(*rule Sup-least*)
fix *x*
assume *x* ∈ *g* (*lub Y*) ‘ *f* (*lub Y*)
with *mcont-contD*[*OF f chain Y*] *mcont-contD*[*OF g chain Y*]
obtain *y z* **where** *y* ∈ *Y z* ∈ *f y*
and *x*: *x* = ⨆((λ*x*. *g x z*) ‘ *Y*) **by** *auto*
show *x* ≤ *?rhs* **unfolding** *x*
proof(*rule Sup-least*)
fix *u*
assume *u* ∈ (λ*x*. *g x z*) ‘ *Y*

```
      then obtain y' where u = g y' z y' ∈ Y by auto
      from chain ‹y ∈ Y› ‹y' ∈ Y› have ord y y' ∨ ord y' y by(rule chainD)
      thus u ≤ ?rhs
      proof
        note ‹u = g y' z› also
        assume ord y y'
        with f have f y ⊆ f y' by(rule mcont-monoD)
        with ‹z ∈ f y›
        have g y' z ≤ ⨆(g y' ‘ f y') by(auto intro: Sup-upper)
        also have ... ≤ ?rhs using ‹y' ∈ Y› by(auto intro: Sup-upper)
        finally show ?thesis .
      next
        note ‹u = g y' z› also
        assume ord y' y
        with g have g y' z ≤ g y z by(rule mcont-monoD)
        also have ... ≤ ⨆(g y ‘ f y) using ‹z ∈ f y›
          by(auto intro: Sup-upper)
        also have ... ≤ ?rhs using ‹y ∈ Y› by(auto intro: Sup-upper)
        finally show ?thesis .
      qed
    qed
  qed
next
  show ?rhs ≤ ?lhs
  proof(rule Sup-least)
    fix x
    assume x ∈ (λx. ⨆(g x ‘ f x)) ‘ Y
    then obtain y where x: x = ⨆(g y ‘ f y) and y ∈ Y by auto
    show x ≤ ?lhs unfolding x
    proof(rule Sup-least)
      fix u
      assume u ∈ g y ‘ f y
      then obtain z where u = g y z z ∈ f y by auto
      note ‹u = g y z›
      also have g y z ≤ ⨆((λx. g x z) ‘ Y)
        using ‹y ∈ Y› by(auto intro: Sup-upper)
      also have ... = g (lub Y) z by(simp add: mcont-contD[OF g chain Y])
      also have ... ≤ ?lhs using ‹z ∈ f y› ‹y ∈ Y›
        by(auto intro: Sup-upper simp add: mcont-contD[OF f chain Y])
      finally show u ≤ ?lhs .
    qed
  qed
qed

lemma mcont-SUP [cont-intro, simp]:
  ⟦ mcont lub ord Union (⊆) f; ⋀y. mcont lub ord Sup (≤) (λx. g x y) ⟧
  ⟹ mcont lub ord Sup (≤) (λx. ⨆y∈f x. g x y)
by(blast intro: mcontI cont-SUP monotone-SUP mcont-mono)
```

end

lemma *admissible-Ball* [*cont-intro*, *simp*]:
⟦ ⋀*x*. *ccpo.admissible lub ord* (λ*A*. *P A x*);
mcont lub ord Union (⊆) *f*;
class.ccpo lub ord (*mk-less ord*) ⟧
⟹ *ccpo.admissible lub ord* (λ*A*. ∀ *x*∈*f A*. *P A x*)
unfolding *Ball-def* **by** *simp*

lemma *admissible-Bex'*[*THEN admissible-subst*, *cont-intro*, *simp*]:
shows *admissible-Bex*: *ccpo.admissible Union* (⊆) (λ*A*. ∃ *x*∈*A*. *P x*)
using *ccpo.admissible-def* **by** *fastforce*

15.6 Parallel fixpoint induction

context
fixes *luba* :: *'a set* ⇒ *'a*
and *orda* :: *'a* ⇒ *'a* ⇒ *bool*
and *lubb* :: *'b set* ⇒ *'b*
and *ordb* :: *'b* ⇒ *'b* ⇒ *bool*
assumes *a*: *class.ccpo luba orda* (*mk-less orda*)
and *b*: *class.ccpo lubb ordb* (*mk-less ordb*)
begin

interpretation *a*: *ccpo luba orda mk-less orda* **by**(*rule a*)
interpretation *b*: *ccpo lubb ordb mk-less ordb* **by**(*rule b*)

lemma *ccpo-rel-prodI*:
class.ccpo (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*mk-less* (*rel-prod orda ordb*))
(**is** *class.ccpo ?lub ?ord ?ord'*)
proof(*intro class.ccpo.intro class.ccpo-axioms.intro*)
show *class.order ?ord ?ord'*
by(*rule order-rel-prodI*) *intro-locales*
show ⋀*A x*. ⟦*Complete-Partial-Order.chain* (*rel-prod orda ordb*) *A*; *x* ∈ *A*⟧
⟹ *rel-prod orda ordb x* (*prod-lub luba lubb A*)
by (*simp add*: *a.ccpo-Sup-upper b.ccpo-Sup-upper chain-rel-prodD1 chain-rel-prodD2*
prod-lub-def rel-prod-sel)
show ⋀*A z*. ⟦*Complete-Partial-Order.chain* (*rel-prod orda ordb*) *A*;
⋀*x*. *x* ∈ *A* ⟹ *rel-prod orda ordb x z*⟧
⟹ *rel-prod orda ordb* (*prod-lub luba lubb A*) *z*
by (*metis* (*full-types*) *a.ccpo-Sup-below-iff b.ccpo-Sup-least chain-rel-prodD1*
chain-rel-prodD2 imageE prod.sel prod-lub-def rel-prod-sel)
qed

interpretation *ab*: *ccpo prod-lub luba lubb rel-prod orda ordb mk-less* (*rel-prod orda ordb*)
by(*rule ccpo-rel-prodI*)

lemma *monotone-map-prod* [*simp*]:
monotone (*rel-prod orda ordb*) (*rel-prod ordc ordd*) (*map-prod f g*) ⟷
monotone orda ordc f ∧ *monotone ordb ordd g*
by(*auto simp add*: *monotone-def*)

lemma *parallel-fixp-induct*:
assumes *adm*: *ccpo.admissible* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. *P* (*fst x*) (*snd x*))
and *f*: *monotone orda orda f*
and *g*: *monotone ordb ordb g*
and *bot*: *P* (*luba* {}) (*lubb* {})
and *step*: ⋀*x y*. *P x y* ⟹ *P* (*f x*) (*g y*)
shows *P* (*ccpo.fixp luba orda f*) (*ccpo.fixp lubb ordb g*)
proof −
let *?lub* = *prod-lub luba lubb*
and *?ord* = *rel-prod orda ordb*
and *?P* = λ(*x*, *y*). *P x y*
from *adm* **have** *adm'*: *ccpo.admissible ?lub ?ord ?P* **by**(*simp add*: *split-def*)
hence *?P* (*ccpo.fixp* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*map-prod f g*))
by(*rule ab.fixp-induct*)(*auto simp add*: *f g step bot*)
also have *ccpo.fixp* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (*map-prod f g*) =
(*ccpo.fixp luba orda f*, *ccpo.fixp lubb ordb g*) (**is** *?lhs* = (*?rhs1*, *?rhs2*))
proof(*rule ab.order.antisym*)
have *ccpo.admissible ?lub ?ord* (λ*xy*. *?ord xy* (*?rhs1*, *?rhs2*))
by(*rule admissible-leI*[*OF ccpo-rel-prodI*])(*auto simp add*: *prod-lub-def chain-empty intro*: *a.ccpo-Sup-least b.ccpo-Sup-least*)
thus *?ord ?lhs* (*?rhs1*, *?rhs2*)
by(*rule ab.fixp-induct*)(*auto 4 3 dest*: *monotoneD*[*OF f*] *monotoneD*[*OF g*] *simp add*: *b.fixp-unfold*[*OF g*, *symmetric*] *a.fixp-unfold*[*OF f*, *symmetric*] *f g intro*: *a.ccpo-Sup-least b.ccpo-Sup-least chain-empty*)
next
have *ccpo.admissible luba orda* (λ*x*. *orda x* (*fst ?lhs*))
by(*rule admissible-leI*[*OF a*])(*auto intro*: *a.ccpo-Sup-least simp add*: *chain-empty*)
hence *orda ?rhs1* (*fst ?lhs*) **using** *f*
proof(*rule a.fixp-induct*)
fix *x*
assume *orda x* (*fst ?lhs*)
thus *orda* (*f x*) (*fst ?lhs*)
by(*subst ab.fixp-unfold*)(*auto simp add*: *f g dest*: *monotoneD*[*OF f*])
qed(*auto intro*: *a.ccpo-Sup-least chain-empty*)
moreover
have *ccpo.admissible lubb ordb* (λ*y*. *ordb y* (*snd ?lhs*))
by(*rule admissible-leI*[*OF b*])(*auto intro*: *b.ccpo-Sup-least simp add*: *chain-empty*)
hence *ordb ?rhs2* (*snd ?lhs*) **using** *g*
proof(*rule b.fixp-induct*)
fix *y*
assume *ordb y* (*snd ?lhs*)
thus *ordb* (*g y*) (*snd ?lhs*)
by (*smt* (*verit*, *best*) *ab.fixp-unfold f g monotoneD monotone-map-prod*

snd-map-prod)
qed(*auto intro*: *b.ccpo-Sup-least chain-empty*)
ultimately show *?ord* (*?rhs1*, *?rhs2*) *?lhs*
by(*simp add*: *rel-prod-conv split-beta*)
qed
finally show *?thesis* **by** *simp*
qed

end

lemma *parallel-fixp-induct-uc*:
assumes *a*: *partial-function-definitions orda luba*
and *b*: *partial-function-definitions ordb lubb*
and *F*: $\bigwedge x.$ *monotone* (*fun-ord orda*) *orda* ($\lambda f.$ *U1* (*F* (*C1 f*)) *x*)
and *G*: $\bigwedge y.$ *monotone* (*fun-ord ordb*) *ordb* ($\lambda g.$ *U2* (*G* (*C2 g*)) *y*)
and *eq1*: *f* ≡ *C1* (*ccpo.fixp* (*fun-lub luba*) (*fun-ord orda*) ($\lambda f.$ *U1* (*F* (*C1 f*))))
and *eq2*: *g* ≡ *C2* (*ccpo.fixp* (*fun-lub lubb*) (*fun-ord ordb*) ($\lambda g.$ *U2* (*G* (*C2 g*))))
and *inverse*: $\bigwedge f.$ *U1* (*C1 f*) = *f*
and *inverse2*: $\bigwedge g.$ *U2* (*C2 g*) = *g*
and *adm*: *ccpo.admissible* (*prod-lub* (*fun-lub luba*) (*fun-lub lubb*)) (*rel-prod* (*fun-ord orda*) (*fun-ord ordb*)) ($\lambda x.$ *P* (*fst x*) (*snd x*))
and *bot*: *P* (λ-. *luba* {}) (λ-. *lubb* {})
and *step*: $\bigwedge f\ g.$ *P* (*U1 f*) (*U2 g*) ⟹ *P* (*U1* (*F f*)) (*U2* (*G g*))
shows *P* (*U1 f*) (*U2 g*)
unfolding *eq1 eq2 inverse inverse2*
proof (*rule parallel-fixp-induct*[*OF partial-function-definitions.ccpo*[*OF a*] *partial-function-definitions.ccpo*[*OF b*] *adm*])
show *monotone* (*fun-ord orda*) (*fun-ord orda*) ($\lambda f.$ *U1* (*F* (*C1 f*)))
monotone (*fun-ord ordb*) (*fun-ord ordb*) ($\lambda g.$ *U2* (*G* (*C2 g*)))
using *F G* **by**(*simp-all add*: *monotone-def fun-ord-def*)
show *P* (*fun-lub luba* {}) (*fun-lub lubb* {})
by (*simp add*: *fun-lub-def bot*)
show $\bigwedge x\ y.$ *P x y* ⟹ *P* (*U1* (*F* (*C1 x*))) (*U2* (*G* (*C2 y*)))
by (*simp add*: *inverse inverse2 local.step*)
qed

lemmas *parallel-fixp-induct-1-1* = *parallel-fixp-induct-uc*[
of - - - - $\lambda x.$ *x* - $\lambda x.$ *x* $\lambda x.$ *x* - $\lambda x.$ *x*,
OF - - - - - - *refl refl*]

lemmas *parallel-fixp-induct-2-2* = *parallel-fixp-induct-uc*[
of - - - - *case-prod* - *curry case-prod* - *curry*,
where *P*=$\lambda f\ g.$ *P* (*curry f*) (*curry g*),
unfolded case-prod-curry curry-case-prod curry-K,
OF - - - - - - *refl refl*]
for *P*

lemma *monotone-fst*: *monotone* (*rel-prod orda ordb*) *orda fst*
by(*auto intro*: *monotoneI*)

lemma *mcont-fst*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *luba orda fst*
by(*auto intro*!: *mcontI monotoneI contI simp add*: *prod-lub-def*)

lemma *mcont2mcont-fst* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord luba orda* (λ*x*. *fst* (*t x*))
by (*simp add*: *mcont-def monotone-on-def prod-lub-def cont-def image-image rel-prod-sel*)

lemma *monotone-snd*: *monotone* (*rel-prod orda ordb*) *ordb snd*
by(*auto intro*: *monotoneI*)

lemma *mcont-snd*: *mcont* (*prod-lub luba lubb*) (*rel-prod orda ordb*) *lubb ordb snd*
by(*auto intro*!: *mcontI monotoneI contI simp add*: *prod-lub-def*)

lemma *mcont2mcont-snd* [*cont-intro*, *simp*]:
mcont lub ord (*prod-lub luba lubb*) (*rel-prod orda ordb*) *t*
⟹ *mcont lub ord lubb ordb* (λ*x*. *snd* (*t x*))
by(*auto intro*!: *mcontI monotoneI contI dest*: *mcont-monoD mcont-contD simp add*: *rel-prod-sel split-beta prod-lub-def image-image*)

lemma *monotone-Pair*:
⟦ *monotone ord orda f*; *monotone ord ordb g* ⟧
⟹ *monotone ord* (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
by(*simp add*: *monotone-def*)

lemma *cont-Pair*:
⟦ *cont lub ord luba orda f*; *cont lub ord lubb ordb g* ⟧
⟹ *cont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
by(*rule contI*)(*auto simp add*: *prod-lub-def image-image dest*!: *contD*)

lemma *mcont-Pair*:
⟦ *mcont lub ord luba orda f*; *mcont lub ord lubb ordb g* ⟧
⟹ *mcont lub ord* (*prod-lub luba lubb*) (*rel-prod orda ordb*) (λ*x*. (*f x*, *g x*))
by(*rule mcontI*)(*simp-all add*: *monotone-Pair mcont-mono cont-Pair*)

context *partial-function-definitions*
begin

Specialised versions of *mcont-call* for admissibility proofs for parallel fixpoint inductions

lemmas *mcont-call-fst* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-fst*]
lemmas *mcont-call-snd* [*cont-intro*] = *mcont-call*[*THEN mcont2mcont*, *OF mcont-snd*]
end

lemma *map-option-mono* [*partial-function-mono*]:
mono-option B ⟹ *mono-option* (λ*f*. *map-option g* (*B f*))
unfolding *map-conv-bind-option* **by**(*rule bind-mono*) *simp-all*

lemma *compact-flat-lub* [*cont-intro*]: *ccpo.compact* (*flat-lub x*) (*flat-ord x*) *y*
using *flat-interpretation*[*THEN ccpo*]
proof(*rule ccpo.compactI*[*OF - ccpo.admissibleI*])
fix *A*
assume *chain*: *Complete-Partial-Order.chain* (*flat-ord x*) *A*
and *A*: $A \neq \{\}$
and $*$: $\forall z{\in}A.\ \neg$ *flat-ord x y z*
from *A* **obtain** *z* **where** $z \in A$ **by** *blast*
with $*$ **have** *z*: $\neg$ *flat-ord x y z* **..**
hence *y*: $x \neq y\ y \neq z$ **by**(*auto simp add*: *flat-ord-def*)
have $y \neq (THE\ z.\ z \in A - \{x\})$ **if** $\neg\ A \subseteq \{x\}$
proof −
from *that* **obtain** z' **where** $z' \in A\ z' \neq x$ **by** *auto*
then have $(THE\ z.\ z \in A - \{x\}) = z'$
by(*intro the-equality*)(*auto dest*: *chainD*[*OF chain*] *simp add*: *flat-ord-def*)
moreover have $z' \neq y$ **using** ‹$z' \in A$› $*$ **by**(*auto simp add*: *flat-ord-def*)
ultimately show *?thesis* **by** *simp*
qed
with *z* **show** $\neg$ *flat-ord x y* (*flat-lub x A*)
by(*simp add*: *flat-ord-def flat-lub-def*)
qed

end

theory *Conditional-Parametricity*
imports *Main*
keywords *parametric-constant* :: *thy-decl*
begin

context includes *lifting-syntax* **begin**

qualified definition *Rel-match* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a \Rightarrow 'b \Rightarrow bool$ **where**
Rel-match R x y = *R x y*

named-theorems *parametricity-preprocess*

lemma *bi-unique-Rel-match* [*parametricity-preprocess*]:
bi-unique A = *Rel-match* (*A* ===> *A* ===> (=)) (=) (=)
unfolding *bi-unique-alt-def2 Rel-match-def* **..**

lemma *bi-total-Rel-match* [*parametricity-preprocess*]:
bi-total A = *Rel-match* ((*A* ===> (=)) ===> (=)) *All All*
unfolding *bi-total-alt-def2 Rel-match-def* **..**

lemma *is-equality-Rel*: *is-equality A* $\Longrightarrow$ *Transfer.Rel A t t*
by (*fact transfer-raw*)

lemma *Rel-Rel-match*: *Transfer.Rel R x y* ⟹ *Rel-match R x y*
unfolding *Rel-match-def Rel-def* **.**

lemma *Rel-match-Rel*: *Rel-match R x y* ⟹ *Transfer.Rel R x y*
unfolding *Rel-match-def Rel-def* **.**

lemma *Rel-Rel-match-eq*: *Transfer.Rel R x y* = *Rel-match R x y*
using *Rel-Rel-match Rel-match-Rel* **by** *fast*

lemma *Rel-match-app*:
assumes *Rel-match* (*A* ===> *B*) *f g* **and** *Transfer.Rel A x y*
shows *Rel-match B* (*f x*) (*g y*)
using *assms Rel-match-Rel Rel-app Rel-Rel-match* **by** *fast*

end

ML-file ‹*conditional-parametricity.ML*›

end
theory *Confluence* **imports**
Main
begin

16 Confluence

definition *semiconfluentp* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
semiconfluentp r ⟷ r^{-1-1} *OO* r^{**} ≤ r^{**} *OO* r^{-1-1**}

definition *confluentp* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
confluentp r ⟷ r^{-1-1**} *OO* r^{**} ≤ r^{**} *OO* r^{-1-1**}

definition *strong-confluentp* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ *bool* **where**
strong-confluentp r ⟷ r^{-1-1} *OO r* ≤ r^{**} *OO* $(r^{-1-1})^{==}$

lemma *semiconfluentpI* [*intro?*]:
semiconfluentp r **if** ⋀*x y z*. ⟦ *r x y*; r^{**} *x z* ⟧ ⟹ ∃ *u*. r^{**} *y u* ∧ r^{**} *z u*
using *that* **unfolding** *semiconfluentp-def rtranclp-conversep* **by** *blast*

lemma *semiconfluentpD*: ∃ *u*. r^{**} *y u* ∧ r^{**} *z u* **if** *semiconfluentp r r x y* r^{**} *x z*
using *that* **unfolding** *semiconfluentp-def rtranclp-conversep* **by** *blast*

lemma *confluentpI*:
confluentp r **if** ⋀*x y z*. ⟦ r^{**} *x y*; r^{**} *x z* ⟧ ⟹ ∃ *u*. r^{**} *y u* ∧ r^{**} *z u*
using *that* **unfolding** *confluentp-def rtranclp-conversep* **by** *blast*

lemma *confluentpD*: ∃ *u*. r^{**} *y u* ∧ r^{**} *z u* **if** *confluentp r* r^{**} *x y* r^{**} *x z*
using *that* **unfolding** *confluentp-def rtranclp-conversep* **by** *blast*

lemma *strong-confluentpI* [*intro?*]:

strong-confluentp r **if** $\bigwedge x\ y\ z.\ [\![\ r\ x\ y;\ r\ x\ z\ ]\!] \Longrightarrow \exists\, u.\ r^{**}\ y\ u \wedge r^{==}\ z\ u$
using *that* **unfolding** *strong-confluentp-def* **by** *blast*

lemma *strong-confluentpD*: $\exists\, u.\ r^{**}\ y\ u \wedge r^{==}\ z\ u$ **if** *strong-confluentp r r x y r x z*
using *that* **unfolding** *strong-confluentp-def* **by** *blast*

lemma *semiconfluentp-imp-confluentp*: *confluentp r* **if** *r*: *semiconfluentp r*
proof(*rule confluentpI*)
show $\exists\, u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$ **if** $r^{**}\ x\ y\ r^{**}\ x\ z$ **for** *x y z*
using *that*(*2*,*1*)
by(*induction arbitrary*: *y rule*: *converse-rtranclp-induct*)
(*blast intro*: *rtranclp-trans dest*: *r*[*THEN semiconfluentpD*])+
qed

lemma *confluentp-imp-semiconfluentp*: *semiconfluentp r* **if** *confluentp r*
using *that* **by**(*auto intro*!: *semiconfluentpI dest*: *confluentpD*[*OF that*])

lemma *confluentp-eq-semiconfluentp*: *confluentp r* $\longleftrightarrow$ *semiconfluentp r*
by(*blast intro*: *semiconfluentp-imp-confluentp confluentp-imp-semiconfluentp*)

lemma *confluentp-conv-strong-confluentp-rtranclp*:
confluentp r $\longleftrightarrow$ *strong-confluentp* (r^{**})
by(*auto simp add*: *confluentp-def strong-confluentp-def rtranclp-conversep*)

lemma *strong-confluentp-into-semiconfluentp*:
semiconfluentp r **if** *r*: *strong-confluentp r*
proof
show $\exists\, u.\ r^{**}\ y\ u \wedge r^{**}\ z\ u$ **if** $r\ x\ y\ r^{**}\ x\ z$ **for** *x y z*
using *that*(*2*,*1*)
apply(*induction arbitrary*: *y rule*: *converse-rtranclp-induct*)
subgoal by *blast*
subgoal for *a b c*
by (*drule* (*1*) *strong-confluentpD*[*OF r*, *of a c*])(*auto 10 0 intro*: *rtranclp-trans*)
done
qed

lemma *strong-confluentp-imp-confluentp*: *confluentp r* **if** *strong-confluentp r*
unfolding *confluentp-eq-semiconfluentp* **using** *that* **by**(*rule strong-confluentp-into-semiconfluentp*)

lemma *semiconfluentp-equivclp*: *equivclp r* = $r^{**}\ OO\ r^{-1-1**}$ **if** *r*: *semiconfluentp r*
proof(*rule antisym*[*rotated*] *r-OO-conversep-into-equivclp predicate2I*)+
show ($r^{**}\ OO\ r^{-1-1**}$) *x y* **if** *equivclp r x y* **for** *x y* **using** *that* **unfolding** *equivclp-def rtranclp-conversep*
by(*induction rule*: *converse-rtranclp-induct*)
(*blast elim*!: *symclpE intro*: *converse-rtranclp-into-rtranclp rtranclp-trans dest*: *semiconfluentpD*[*OF r*])+
qed

end
theory *Confluent-Quotient* **imports**
Confluence
begin

Functors with finite setters preserve wide intersection for any equivalence relation that respects the mapper.

lemma *Inter-finite-subset*:
assumes $\forall A \in \mathcal{A}$. *finite* A
shows $\exists \mathcal{B} \subseteq \mathcal{A}$. *finite* $\mathcal{B} \wedge (\bigcap \mathcal{B}) = (\bigcap \mathcal{A})$
proof(*cases* $\mathcal{A} = \{\}$)
case *False*
then obtain A **where** A: $A \in \mathcal{A}$ **by** *auto*
then have *finA*: *finite* A **using** *assms* **by** *auto*
hence *fin*: *finite* $(A - \bigcap \mathcal{A})$ **by**(*rule finite-subset*[*rotated*]) *auto*
let *?P* $= \lambda x\ A.\ A \in \mathcal{A} \wedge x \notin A$
define f **where** $f\ x = Eps\ (?P\ x)$ **for** x
let *?B* $= insert\ A\ (f\ `\ (A - \bigcap \mathcal{A}))$
have *?P x* $(f\ x)$ **if** $x \in A - \bigcap \mathcal{A}$ **for** x **unfolding** *f-def* **by**(*rule someI-ex*)(*use that A* **in** *auto*)
hence $(\bigcap ?\mathcal{B}) = (\bigcap \mathcal{A})$ *?B* $\subseteq \mathcal{A}$ **using** A **by** *auto*
moreover have *finite ?B* **using** *fin* **by** *simp*
ultimately show *?thesis* **by** *blast*
qed *simp*

locale *wide-intersection-finite* =
fixes $E :: 'Fa \Rightarrow 'Fa \Rightarrow bool$
and $mapFa :: ('a \Rightarrow 'a) \Rightarrow 'Fa \Rightarrow 'Fa$
and $setFa :: 'Fa \Rightarrow 'a\ set$
assumes *equiv*: *equivp E*
and *map-E*: $E\ x\ y \Longrightarrow E\ (mapFa\ f\ x)\ (mapFa\ f\ y)$
and *map-id*: $mapFa\ id\ x = x$
and *map-cong*: $\forall a \in setFa\ x.\ f\ a = g\ a \Longrightarrow mapFa\ f\ x = mapFa\ g\ x$
and *set-map*: $setFa\ (mapFa\ f\ x) = f\ `\ setFa\ x$
and *finite*: *finite* $(setFa\ x)$
begin

lemma *binary-intersection*:
assumes $E\ y\ z$ **and** y: $setFa\ y \subseteq Y$ **and** z: $setFa\ z \subseteq Z$ **and** a: $a \in Y\ a \in Z$
shows $\exists x.\ E\ x\ y \wedge setFa\ x \subseteq Y \wedge setFa\ x \subseteq Z$
proof −
let *?f* $= \lambda b.$ *if* $b \in Z$ *then* b *else* a
let *?u* $= mapFa$ *?f y*
from ‹$E\ y\ z$› **have** E *?u* $(mapFa$ *?f z*$)$ **by**(*rule map-E*)
also have $mapFa$ *?f z* $= mapFa\ id\ z$ **by**(*rule map-cong*)(*use z* **in** *auto*)
also have ... $= z$ **by**(*rule map-id*)
finally have E *?u y* **using** ‹$E\ y\ z$› *equivp-symp*[*OF equiv*] *equivp-transp*[*OF equiv*]
by *blast*

moreover have *setFa ?u* ⊆ *Y* **using** *a y* **by**(*subst set-map*) *auto*
moreover have *setFa ?u* ⊆ *Z* **using** *a* **by**(*subst set-map*) *auto*
ultimately show *?thesis* **by** *blast*
qed

lemma *finite-intersection*:
assumes *E*: ∀ *y*∈*A*. *E y z*
and *fin*: *finite A*
and *sub*: ∀ *y*∈*A*. *setFa y* ⊆ *Y y* ∧ *a* ∈ *Y y*
shows ∃ *x*. *E x z* ∧ (∀ *y*∈*A*. *setFa x* ⊆ *Y y*)
using *fin E sub*
proof(*induction*)
case *empty*
then show *?case* **using** *equivp-reflp*[*OF equiv*, *of z*] **by**(*auto*)
next
case (*insert y A*)
then obtain *x* **where** *x*: *E x z* ∀ *y*∈*A*. *setFa x* ⊆ *Y y* ∧ *a* ∈ *Y y* **by** *auto*
hence *set-x*: *setFa x* ⊆ (⋂ *y*∈*A*. *Y y*) *a* ∈ (⋂ *y*∈*A*. *Y y*) **by** *auto*
from *insert.prems* **have** *E y z* **and** *set-y*: *setFa y* ⊆ *Y y a* ∈ *Y y* **by** *auto*
from ‹*E y z*› ‹*E x z*› **have** *E x y* **using** *equivp-symp*[*OF equiv*] *equivp-transp*[*OF equiv*] **by** *blast*
from *binary-intersection*[*OF this set-x*(*1*) *set-y*(*1*) *set-x*(*2*) *set-y*(*2*)]
obtain *x'* **where** *E x' x setFa x'* ⊆ ⋂ (*Y* ‘ *A*) *setFa x'* ⊆ *Y y* **by** *blast*
then show *?case* **using** ‹*E x z*› *equivp-transp*[*OF equiv*] **by** *blast*
qed

lemma *wide-intersection*:
assumes *inter-nonempty*: ⋂ *Ss* ≠ {}
shows (⋂ *As* ∈ *Ss*. {(*x*, *x'*). *E x x'*} `` {*x*. *setFa x* ⊆ *As*}) ⊆ {(*x*, *x'*). *E x x'*} `` {*x*. *setFa x* ⊆ ⋂ *Ss*} (**is** *?lhs* ⊆ *?rhs*)
proof
fix *x*
assume *lhs*: *x* ∈ *?lhs*
from *inter-nonempty* **obtain** *a* **where** *a*: ∀ *As* ∈ *Ss*. *a* ∈ *As* **by** *auto*
from *lhs* **obtain** *y* **where** *y*: ⋀*As*. *As* ∈ *Ss* ⟹ *E* (*y As*) *x* ∧ *setFa* (*y As*) ⊆ *As*
by *atomize-elim*(*rule choice*, *auto*)
define *Ts* **where** *Ts* = (λ*As*. *insert a* (*setFa* (*y As*))) ‘ *Ss*
have *Ts-subset*: (⋂ *Ts*) ⊆ (⋂ *Ss*) **using** *a* **unfolding** *Ts-def* **by**(*auto dest*: *y*)
have *Ts-finite*: ∀ *Bs* ∈ *Ts*. *finite Bs* **unfolding** *Ts-def* **by**(*auto dest*: *y intro*: *finite*)
from *Inter-finite-subset*[*OF this*] **obtain** *Us*
where *Us*: *Us* ⊆ *Ts* **and** *finite-Us*: *finite Us* **and** *Int-Us*: (⋂ *Us*) ⊆ (⋂ *Ts*) **by** *force*
let *?P* = λ*U As*. *As* ∈ *Ss* ∧ *U* = *insert a* (*setFa* (*y As*))
define *Y* **where** *Y U* = *Eps* (*?P U*) **for** *U*
have *Y*: *?P U* (*Y U*) **if** *U* ∈ *Us* **for** *U* **unfolding** *Y-def*
by(*rule someI-ex*)(*use that Us* **in** ‹*auto simp add*: *Ts-def*›)
let *?f* = λ*U*. *y* (*Y U*)
have ∗: ∀ *z*∈(*?f* ‘ *Us*). *E z x* **by**(*auto dest*!: *Y y*)

have $**$: $\forall z{\in}(?f\ `\ Us).\ setFa\ z \subseteq insert\ a\ (setFa\ z) \wedge a \in insert\ a\ (setFa\ z)$ **by** *auto*
from *finite-intersection*[*OF* $*$ - $**$] *finite-Us* **obtain** u
where u: $E\ u\ x$ **and** *set-u*: $\forall z{\in}(?f\ `\ Us).\ setFa\ u \subseteq insert\ a\ (setFa\ z)$ **by** *auto*
from *set-u* **have** $setFa\ u \subseteq (\bigcap\ Us)$ **by**(*auto dest*: Y)
with *Int-Us Ts-subset* **have** $setFa\ u \subseteq (\bigcap\ Ss)$ **by** *auto*
with u **show** $x \in ?rhs$ **by** *auto*
qed

end

Subdistributivity for quotients via confluence

lemma *rtranclp-transp-reflp*: $R^{**} = R$ **if** *transp R reflp R*
apply(*rule ext iffI*)+
subgoal premises *prems* **for** $x\ y$ **using** *prems* **by**(*induction*)(*use that* **in** ‹*auto intro*: *reflpD transpD*›)
subgoal by(*rule r-into-rtranclp*)
done

lemma *rtranclp-equivp*: $R^{**} = R$ **if** *equivp R*
using *that* **by**(*simp add*: *rtranclp-transp-reflp equivp-reflp-symp-transp*)

locale *confluent-quotient* =
fixes *Rb* :: *'Fb* ⇒ *'Fb* ⇒ *bool*
and *Ea* :: *'Fa* ⇒ *'Fa* ⇒ *bool*
and *Eb* :: *'Fb* ⇒ *'Fb* ⇒ *bool*
and *Ec* :: *'Fc* ⇒ *'Fc* ⇒ *bool*
and *Eab* :: *'Fab* ⇒ *'Fab* ⇒ *bool*
and *Ebc* :: *'Fbc* ⇒ *'Fbc* ⇒ *bool*
and *π-Faba* :: *'Fab* ⇒ *'Fa*
and *π-Fabb* :: *'Fab* ⇒ *'Fb*
and *π-Fbcb* :: *'Fbc* ⇒ *'Fb*
and *π-Fbcc* :: *'Fbc* ⇒ *'Fc*
and *rel-Fab* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ *'Fa* ⇒ *'Fb* ⇒ *bool*
and *rel-Fbc* :: (*'b* ⇒ *'c* ⇒ *bool*) ⇒ *'Fb* ⇒ *'Fc* ⇒ *bool*
and *rel-Fac* :: (*'a* ⇒ *'c* ⇒ *bool*) ⇒ *'Fa* ⇒ *'Fc* ⇒ *bool*
and *set-Fab* :: *'Fab* ⇒ (*'a* × *'b*) *set*
and *set-Fbc* :: *'Fbc* ⇒ (*'b* × *'c*) *set*
assumes *confluent*: *confluentp Rb*
and *retract1-ab*: $\bigwedge x\ y.\ Rb\ (\pi\text{-}Fabb\ x)\ y \Longrightarrow \exists z.\ Eab\ x\ z \wedge y = \pi\text{-}Fabb\ z \wedge set\text{-}Fab\ z \subseteq set\text{-}Fab\ x$
and *retract1-bc*: $\bigwedge x\ y.\ Rb\ (\pi\text{-}Fbcb\ x)\ y \Longrightarrow \exists z.\ Ebc\ x\ z \wedge y = \pi\text{-}Fbcb\ z \wedge set\text{-}Fbc\ z \subseteq set\text{-}Fbc\ x$
and *generated-b*: $Eb \leq equivclp\ Rb$
and *transp-a*: *transp Ea*
and *transp-c*: *transp Ec*
and *equivp-ab*: *equivp Eab*
and *equivp-bc*: *equivp Ebc*
and *in-rel-Fab*: $\bigwedge A\ x\ y.\ rel\text{-}Fab\ A\ x\ y \longleftrightarrow (\exists z.\ z \in \{x.\ set\text{-}Fab\ x \subseteq \{(x,\ y).\ A$

$x\ y\}\} \wedge \pi\text{-}Faba\ z = x \wedge \pi\text{-}Fabb\ z = y$)
and *in-rel-Fbc*: $\bigwedge B\ x\ y.\ rel\text{-}Fbc\ B\ x\ y \longleftrightarrow (\exists z.\ z \in \{x.\ set\text{-}Fbc\ x \subseteq \{(x,\ y).\ B\ x\ y\}\} \wedge \pi\text{-}Fbcb\ z = x \wedge \pi\text{-}Fbcc\ z = y)$
and *rel-compp*: $\bigwedge A\ B.\ rel\text{-}Fac\ (A\ OO\ B) = rel\text{-}Fab\ A\ OO\ rel\text{-}Fbc\ B$
and π*-Faba-respect*: *rel-fun Eab Ea* π*-Faba* π*-Faba*
and π*-Fbcc-respect*: *rel-fun Ebc Ec* π*-Fbcc* π*-Fbcc*
begin

lemma *retract-ab*: $Rb^{**}\ (\pi\text{-}Fabb\ x)\ y \Longrightarrow \exists z.\ Eab\ x\ z \wedge y = \pi\text{-}Fabb\ z \wedge set\text{-}Fab\ z \subseteq set\text{-}Fab\ x$
by(*induction rule*: *rtranclp-induct*)(*blast dest*: *retract1-ab intro*: *equivp-transp*[*OF equivp-ab*] *equivp-reflp*[*OF equivp-ab*])+

lemma *retract-bc*: $Rb^{**}\ (\pi\text{-}Fbcb\ x)\ y \Longrightarrow \exists z.\ Ebc\ x\ z \wedge y = \pi\text{-}Fbcb\ z \wedge set\text{-}Fbc\ z \subseteq set\text{-}Fbc\ x$
by(*induction rule*: *rtranclp-induct*)(*blast dest*: *retract1-bc intro*: *equivp-transp*[*OF equivp-bc*] *equivp-reflp*[*OF equivp-bc*])+

lemma *subdistributivity*: $rel\text{-}Fab\ A\ OO\ Eb\ OO\ rel\text{-}Fbc\ B \leq Ea\ OO\ rel\text{-}Fac\ (A\ OO\ B)\ OO\ Ec$
proof(*rule predicate2I*; *elim relcomppE*)
fix $x\ y\ y'\ z$
assume $rel\text{-}Fab\ A\ x\ y$ **and** $Eb\ y\ y'$ **and** $rel\text{-}Fbc\ B\ y'\ z$
then obtain $xy\ y'z$
where *xy*: $set\text{-}Fab\ xy \subseteq \{(a,\ b).\ A\ a\ b\}\ x = \pi\text{-}Faba\ xy\ y = \pi\text{-}Fabb\ xy$
and $y'z$: $set\text{-}Fbc\ y'z \subseteq \{(a,\ b).\ B\ a\ b\}\ y' = \pi\text{-}Fbcb\ y'z\ z = \pi\text{-}Fbcc\ y'z$
by(*auto simp add*: *in-rel-Fab in-rel-Fbc*)
from ‹$Eb\ y\ y'$› **have** $equivclp\ Rb\ y\ y'$ **using** *generated-b* **by** *blast*
then obtain u **where** u: $Rb^{**}\ y\ u\ Rb^{**}\ y'\ u$
unfolding *semiconfluentp-equivclp*[*OF confluent*[*THEN confluentp-imp-semiconfluentp*]]
by(*auto simp add*: *rtranclp-conversep*)
with $xy\ y'z$ **obtain** $xy'\ y'z'$
where *retract1*: $Eab\ xy\ xy'\ \pi\text{-}Fabb\ xy' = u\ set\text{-}Fab\ xy' \subseteq set\text{-}Fab\ xy$
and *retract2*: $Ebc\ y'z\ y'z'\ \pi\text{-}Fbcb\ y'z' = u\ set\text{-}Fbc\ y'z' \subseteq set\text{-}Fbc\ y'z$
by(*auto dest*!: *retract-ab retract-bc*)
from *retract1*(*1*) xy **have** $Ea\ x\ (\pi\text{-}Faba\ xy')$ **by**(*auto dest*: π*-Faba-respect*[*THEN rel-funD*])
moreover have $rel\text{-}Fab\ A\ (\pi\text{-}Faba\ xy')\ u$ **using** xy *retract1* **by**(*auto simp add*: *in-rel-Fab*)
moreover have $rel\text{-}Fbc\ B\ u\ (\pi\text{-}Fbcc\ y'z')$ **using** $y'z$ *retract2* **by**(*auto simp add*: *in-rel-Fbc*)
moreover have $Ec\ (\pi\text{-}Fbcc\ y'z')\ z$ **using** *retract2* $y'z$ *equivp-symp*[*OF equivp-bc*]
by(*auto intro*: π*-Fbcc-respect*[*THEN rel-funD*])
ultimately show $(Ea\ OO\ rel\text{-}Fac\ (A\ OO\ B)\ OO\ Ec)\ x\ z$ **unfolding** *rel-compp* **by** *blast*
qed

end

end

17 Old Datatype package: constructing datatypes from Cartesian Products and Disjoint Sums

theory *Old-Datatype*
imports *Main*
begin

17.1 The datatype universe

definition *Node* = $\{p.\ \exists f\ x\ k.\ p = (f :: nat \Rightarrow 'b + nat,\ x :: 'a + nat) \wedge f\ k = Inr\ 0\}$

typedef $('a, 'b)$ *node* = *Node* :: $((nat \Rightarrow 'b + nat) * ('a + nat))$ *set*
  morphisms *Rep-Node Abs-Node*
  unfolding *Node-def* **by** *auto*

Datatypes will be represented by sets of type *node*

type-synonym $'a$ *item* = $('a,$ *unit*$)$ *node set*
type-synonym $('a, 'b)$ *dtree* = $('a, 'b)$ *node set*

definition *Push* :: $[('b + nat), nat \Rightarrow ('b + nat)] \Rightarrow (nat \Rightarrow ('b + nat))$

  where *Push* == $(\%b\ h.$ *case-nat b h*$)$

definition *Push-Node* :: $[('b + nat), ('a, 'b)\ node] \Rightarrow ('a, 'b)\ node$
  where *Push-Node* == $(\%n\ x.$ *Abs-Node* $($*apfst* $($*Push n*$)$ $($*Rep-Node x*$)))$

definition *Atom* :: $('a + nat) \Rightarrow ('a, 'b)$ *dtree*
  where *Atom* == $(\%x.\ \{$*Abs-Node*$((\%k.$ *Inr 0*$,\ x))\})$
definition *Scons* :: $[('a, 'b)$ *dtree*$, ('a, 'b)$ *dtree*$] \Rightarrow ('a, 'b)$ *dtree*
  where *Scons M N* == $($*Push-Node* $($*Inr 1*$)$ ‘ *M*$)$ *Un* $($*Push-Node* $($*Inr* $($*Suc 1*$))$ ‘ *N*$)$

definition *Leaf* :: $'a \Rightarrow ('a, 'b)$ *dtree*
  where *Leaf* == *Atom* ∘ *Inl*
definition *Numb* :: $nat \Rightarrow ('a, 'b)$ *dtree*
  where *Numb* == *Atom* ∘ *Inr*

definition *In0* :: $('a, 'b)$ *dtree* $\Rightarrow ('a, 'b)$ *dtree*
  where *In0*(M) == *Scons* $($*Numb 0*$)$ *M*

definition *In1* :: ('a, 'b) *dtree* => ('a, 'b) *dtree*
  where *In1*(*M*) == *Scons* (*Numb* 1) *M*

definition *Lim* :: ('b => ('a, 'b) *dtree*) => ('a, 'b) *dtree*
  where *Lim f* == $\bigcup${*z*. ∃ *x*. *z* = *Push-Node* (*Inl x*) ' (*f x*)}

definition *ndepth* :: ('a, 'b) *node* => *nat*
  where *ndepth*(*n*) == (%(*f*,*x*). *LEAST k*. *f k* = *Inr* 0) (*Rep-Node n*)
definition *ntrunc* :: [*nat*, ('a, 'b) *dtree*] => ('a, 'b) *dtree*
  where *ntrunc k N* == {*n*. *n*∈*N* ∧ *ndepth*(*n*)<*k*}

definition *uprod* :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*
  where *uprod A B* == *UN x*:*A*. *UN y*:*B*. { *Scons x y* }
definition *usum* :: [('a, 'b) *dtree set*, ('a, 'b) *dtree set*]=> ('a, 'b) *dtree set*
  where *usum A B* == *In0*'*A Un In1*'*B*

definition *Split* :: [[('a, 'b) *dtree*, ('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c
  where *Split c M* == *THE u*. ∃ *x y*. *M* = *Scons x y* ∧ *u* = *c x y*

definition *Case* :: [[('a, 'b) *dtree*]=>'c, [('a, 'b) *dtree*]=>'c, ('a, 'b) *dtree*] => 'c
  where *Case c d M* == *THE u*. (∃ *x* . *M* = *In0*(*x*) ∧ *u* = *c*(*x*)) ∨ (∃ *y* . *M* = *In1*(*y*) ∧ *u* = *d*(*y*))

definition *dprod* :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
    => (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*
  where *dprod r s* == *UN* (*x*,*x*'):*r*. *UN* (*y*,*y*'):*s*. {(*Scons x y*, *Scons x*' *y*')}

definition *dsum* :: [(('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*, (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*]
    => (('a, 'b) *dtree* * ('a, 'b) *dtree*)*set*
 where *dsum r s* == (*UN* (*x*,*x*'):*r*. {(*In0*(*x*),*In0*(*x*'))}) *Un* (*UN* (*y*,*y*'):*s*. {(*In1*(*y*),*In1*(*y*'))})

lemma *apfst-convE*:
    [| *q* = *apfst f p*; !!*x y*. [| *p* = (*x*,*y*); *q* = (*f*(*x*),*y*) |] ==> *R*
     |] ==> *R*
by (*force simp add*: *apfst-def*)

lemma *Push-inject1*: *Push i f* = *Push j g* ==> *i*=*j*

apply (*simp add*: *Push-def fun-eq-iff*)
apply (*drule-tac x=0* **in** *spec*, *simp*)
done

lemma *Push-inject2*: *Push i f = Push j g ==> f=g*
apply (*auto simp add*: *Push-def fun-eq-iff*)
apply (*drule-tac x=Suc x* **in** *spec*, *simp*)
done

lemma *Push-inject*:
[| *Push i f =Push j g*; [| *i=j*; *f=g* |] ==> *P* |] ==> *P*
by (*blast dest*: *Push-inject1 Push-inject2*)

lemma *Push-neq-K0*: *Push (Inr (Suc k)) f = (%z. Inr 0) ==> P*
by (*auto simp add*: *Push-def fun-eq-iff split*: *nat.split-asm*)

lemmas *Abs-Node-inj = Abs-Node-inject* [*THEN* [*2*] *rev-iffD1*]

lemma *Node-K0-I*: (*λk. Inr 0*, *a*) ∈ *Node*
by (*simp add*: *Node-def*)

lemma *Node-Push-I*: *p* ∈ *Node* ⟹ *apfst* (*Push i*) *p* ∈ *Node*
apply (*simp add*: *Node-def Push-def*)
apply (*fast intro*!: *apfst-conv nat.case*(*2*)[*THEN trans*])
done

17.2 Freeness: Distinctness of Constructors

lemma *Scons-not-Atom* [*iff*]: *Scons M N* ≠ *Atom*(*a*)
unfolding *Atom-def Scons-def Push-Node-def One-nat-def*
by (*blast intro*: *Node-K0-I Rep-Node* [*THEN Node-Push-I*]
dest!: *Abs-Node-inj*
elim!: *apfst-convE sym* [*THEN Push-neq-K0*])

lemmas *Atom-not-Scons* [*iff*] = *Scons-not-Atom* [*THEN not-sym*]

lemma *inj-Atom*: *inj*(*Atom*)
apply (*simp add*: *Atom-def*)
apply (*blast intro*!: *inj-onI Node-K0-I dest*!: *Abs-Node-inj*)
done
lemmas *Atom-inject = inj-Atom* [*THEN injD*]

lemma *Atom-Atom-eq* [*iff*]: (*Atom*(*a*)=*Atom*(*b*)) = (*a*=*b*)
by (*blast dest*!: *Atom-inject*)

lemma *inj-Leaf*: *inj*(*Leaf*)
apply (*simp add*: *Leaf-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inl-inject*])
done

lemmas *Leaf-inject* [*dest*!] = *inj-Leaf* [*THEN injD*]

lemma *inj-Numb*: *inj*(*Numb*)
apply (*simp add*: *Numb-def o-def*)
apply (*rule inj-onI*)
apply (*erule Atom-inject* [*THEN Inr-inject*])
done

lemmas *Numb-inject* [*dest*!] = *inj-Numb* [*THEN injD*]

lemma *Push-Node-inject*:
[| *Push-Node i m* =*Push-Node j n*; [| *i*=*j*; *m*=*n* |] ==> *P*
|] ==> *P*
apply (*simp add*: *Push-Node-def*)
apply (*erule Abs-Node-inj* [*THEN apfst-convE*])
apply (*rule Rep-Node* [*THEN Node-Push-I*])+
apply (*erule sym* [*THEN apfst-convE*])
apply (*blast intro*: *Rep-Node-inject* [*THEN iffD1*] *trans sym elim*!: *Push-inject*)
done

lemma *Scons-inject-lemma1*: *Scons M N* <= *Scons M' N'* ==> *M*<=*M'*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

lemma *Scons-inject-lemma2*: *Scons M N* <= *Scons M' N'* ==> *N*<=*N'*
unfolding *Scons-def One-nat-def*
by (*blast dest*!: *Push-Node-inject*)

lemma *Scons-inject1*: *Scons M N* = *Scons M' N'* ==> *M*=*M'*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma1*)
done

lemma *Scons-inject2*: *Scons M N = Scons M′ N′ ==> N=N′*
apply (*erule equalityE*)
apply (*iprover intro*: *equalityI Scons-inject-lemma2*)
done

lemma *Scons-inject*:
[| *Scons M N = Scons M′ N′*; [| *M=M′*; *N=N′* |] ==> *P* |] ==> *P*
by (*iprover dest*: *Scons-inject1 Scons-inject2*)

lemma *Scons-Scons-eq* [*iff*]: (*Scons M N = Scons M′ N′*) = (*M=M′* ∧ *N=N′*)
by (*blast elim*!: *Scons-inject*)

lemma *Scons-not-Leaf* [*iff*]: *Scons M N* ≠ *Leaf*(*a*)
unfolding *Leaf-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Leaf-not-Scons* [*iff*] = *Scons-not-Leaf* [*THEN not-sym*]

lemma *Scons-not-Numb* [*iff*]: *Scons M N* ≠ *Numb*(*k*)
unfolding *Numb-def o-def* **by** (*rule Scons-not-Atom*)

lemmas *Numb-not-Scons* [*iff*] = *Scons-not-Numb* [*THEN not-sym*]

lemma *Leaf-not-Numb* [*iff*]: *Leaf*(*a*) ≠ *Numb*(*k*)
by (*simp add*: *Leaf-def Numb-def*)

lemmas *Numb-not-Leaf* [*iff*] = *Leaf-not-Numb* [*THEN not-sym*]

lemma *ndepth-K0*: *ndepth* (*Abs-Node*(%*k*. *Inr 0*, *x*)) = *0*
by (*simp add*: *ndepth-def Node-K0-I* [*THEN Abs-Node-inverse*] *Least-equality*)

lemma *ndepth-Push-Node-aux*:
case-nat (*Inr* (*Suc i*)) *f k = Inr 0* ⟶ *Suc*(*LEAST x*. *f x = Inr 0*) ≤ *k*
apply (*induct-tac k*, *auto*)
apply (*erule Least-le*)
done

lemma *ndepth-Push-Node*:

ndepth (*Push-Node* (*Inr* (*Suc i*)) *n*) = *Suc*(*ndepth*(*n*))
apply (*insert Rep-Node* [*of n*, *unfolded Node-def*])
apply (*auto simp add*: *ndepth-def Push-Node-def*
Rep-Node [*THEN Node-Push-I*, *THEN Abs-Node-inverse*])
apply (*rule Least-equality*)
apply (*auto simp add*: *Push-def ndepth-Push-Node-aux*)
apply (*erule LeastI*)
done

lemma *ntrunc-0* [*simp*]: *ntrunc 0 M* = {}
by (*simp add*: *ntrunc-def*)

lemma *ntrunc-Atom* [*simp*]: *ntrunc* (*Suc k*) (*Atom a*) = *Atom*(*a*)
by (*auto simp add*: *Atom-def ntrunc-def ndepth-K0*)

lemma *ntrunc-Leaf* [*simp*]: *ntrunc* (*Suc k*) (*Leaf a*) = *Leaf*(*a*)
unfolding *Leaf-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Numb* [*simp*]: *ntrunc* (*Suc k*) (*Numb i*) = *Numb*(*i*)
unfolding *Numb-def o-def* **by** (*rule ntrunc-Atom*)

lemma *ntrunc-Scons* [*simp*]:
ntrunc (*Suc k*) (*Scons M N*) = *Scons* (*ntrunc k M*) (*ntrunc k N*)
unfolding *Scons-def ntrunc-def One-nat-def*
by (*auto simp add*: *ndepth-Push-Node*)

lemma *ntrunc-one-In0* [*simp*]: *ntrunc* (*Suc 0*) (*In0 M*) = {}
apply (*simp add*: *In0-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In0* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In0 M*) = *In0* (*ntrunc* (*Suc k*) *M*)
by (*simp add*: *In0-def*)

lemma *ntrunc-one-In1* [*simp*]: *ntrunc* (*Suc 0*) (*In1 M*) = {}
apply (*simp add*: *In1-def*)
apply (*simp add*: *Scons-def*)
done

lemma *ntrunc-In1* [*simp*]: *ntrunc* (*Suc*(*Suc k*)) (*In1 M*) = *In1* (*ntrunc* (*Suc k*) *M*)

by (*simp add*: *In1-def*)

17.3 Set Constructions

lemma *uprodI* [*intro*!]: $[\![M{\in}A;\ N{\in}B]\!] \Longrightarrow Scons\ M\ N \in uprod\ A\ B$
by (*simp add*: *uprod-def*)

lemma *uprodE* [*elim*!]:
$[\![c \in uprod\ A\ B;$
$\bigwedge x\ y.\ [\![x \in A;\ y \in B;\ c = Scons\ x\ y]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*auto simp add*: *uprod-def*)

lemma *uprodE2*: $[\![Scons\ M\ N \in uprod\ A\ B;\ [\![M \in A;\ N \in B]\!] \Longrightarrow P]\!] \Longrightarrow P$
by (*auto simp add*: *uprod-def*)

lemma *usum-In0I* [*intro*]: $M \in A \Longrightarrow In0(M) \in usum\ A\ B$
by (*simp add*: *usum-def*)

lemma *usum-In1I* [*intro*]: $N \in B \Longrightarrow In1(N) \in usum\ A\ B$
by (*simp add*: *usum-def*)

lemma *usumE* [*elim*!]:
$[\![u \in usum\ A\ B;$
$\bigwedge x.\ [\![x \in A;\ u{=}In0(x)]\!] \Longrightarrow P;$
$\bigwedge y.\ [\![y \in B;\ u{=}In1(y)]\!] \Longrightarrow P$
$]\!] \Longrightarrow P$
by (*auto simp add*: *usum-def*)

lemma *In0-not-In1* [*iff*]: $In0(M) \neq In1(N)$
unfolding *In0-def In1-def One-nat-def* **by** *auto*

lemmas *In1-not-In0* [*iff*] = *In0-not-In1* [*THEN not-sym*]

lemma *In0-inject*: $In0(M) = In0(N) \Longrightarrow M{=}N$
by (*simp add*: *In0-def*)

lemma *In1-inject*: $In1(M) = In1(N) \Longrightarrow M{=}N$
by (*simp add*: *In1-def*)

lemma *In0-eq* [*iff*]: (*In0 M = In0 N*) = (*M=N*)
by (*blast dest*!: *In0-inject*)

lemma *In1-eq* [*iff*]: (*In1 M = In1 N*) = (*M=N*)
by (*blast dest*!: *In1-inject*)

lemma *inj-In0*: *inj In0*
by (*blast intro*!: *inj-onI*)

lemma *inj-In1*: *inj In1*
by (*blast intro*!: *inj-onI*)

lemma *Lim-inject*: *Lim f = Lim g ==> f = g*
apply (*simp add*: *Lim-def*)
apply (*rule ext*)
apply (*blast elim*!: *Push-Node-inject*)
done

lemma *ntrunc-subsetI*: *ntrunc k M <= M*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-subsetD*: (!!*k*. *ntrunc k M <= N*) *==> M<=N*
by (*auto simp add*: *ntrunc-def*)

lemma *ntrunc-equality*: (!!*k*. *ntrunc k M = ntrunc k N*) *==> M=N*
apply (*rule equalityI*)
apply (*rule-tac* [!] *ntrunc-subsetD*)
apply (*rule-tac* [!] *ntrunc-subsetI* [*THEN* [*2*] *subset-trans*], *auto*)
done

lemma *ntrunc-o-equality*:
[| !!*k*. (*ntrunc*(*k*) ∘ *h1*) = (*ntrunc*(*k*) ∘ *h2*) |] *==> h1=h2*
apply (*rule ntrunc-equality* [*THEN ext*])
apply (*simp add*: *fun-eq-iff*)
done

lemma *uprod-mono*: [| *A<=A'*; *B<=B'* |] *==> uprod A B <= uprod A' B'*
by (*simp add*: *uprod-def*, *blast*)

lemma *usum-mono*: [| *A*<=*A'*; *B*<=*B'* |] ==> *usum A B* <= *usum A' B'*
by (*simp add*: *usum-def*, *blast*)

lemma *Scons-mono*: [| *M*<=*M'*; *N*<=*N'* |] ==> *Scons M N* <= *Scons M' N'*
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-mono*: *M*<=*N* ==> *In0*(*M*) <= *In0*(*N*)
by (*simp add*: *In0-def Scons-mono*)

lemma *In1-mono*: *M*<=*N* ==> *In1*(*M*) <= *In1*(*N*)
by (*simp add*: *In1-def Scons-mono*)

lemma *Split* [*simp*]: *Split c* (*Scons M N*) = *c M N*
by (*simp add*: *Split-def*)

lemma *Case-In0* [*simp*]: *Case c d* (*In0 M*) = *c*(*M*)
by (*simp add*: *Case-def*)

lemma *Case-In1* [*simp*]: *Case c d* (*In1 N*) = *d*(*N*)
by (*simp add*: *Case-def*)

lemma *ntrunc-UN1*: *ntrunc k* (*UN x. f*(*x*)) = (*UN x. ntrunc k* (*f x*))
by (*simp add*: *ntrunc-def*, *blast*)

lemma *Scons-UN1-x*: *Scons* (*UN x. f x*) *M* = (*UN x. Scons* (*f x*) *M*)
by (*simp add*: *Scons-def*, *blast*)

lemma *Scons-UN1-y*: *Scons M* (*UN x. f x*) = (*UN x. Scons M* (*f x*))
by (*simp add*: *Scons-def*, *blast*)

lemma *In0-UN1*: *In0*(*UN x. f*(*x*)) = (*UN x. In0*(*f*(*x*)))
by (*simp add*: *In0-def Scons-UN1-y*)

lemma *In1-UN1*: *In1*(*UN x. f*(*x*)) = (*UN x. In1*(*f*(*x*)))
by (*simp add*: *In1-def Scons-UN1-y*)

lemma *dprodI* [*intro*!]:
⟦(*M*,*M'*) ∈ *r*; (*N*,*N'*) ∈ *s*⟧ ⟹ (*Scons M N*, *Scons M' N'*) ∈ *dprod r s*
by (*auto simp add*: *dprod-def*)

lemma *dprodE* [*elim*!]:
⟦*c* ∈ *dprod r s*;
⋀*x y x′ y′*. ⟦(*x*,*x′*) ∈ *r*; (*y*,*y′*) ∈ *s*;
c = (*Scons x y*, *Scons x′ y′*)⟧ ⟹ *P*
⟧ ⟹ *P*
by (*auto simp add*: *dprod-def*)

lemma *dsum-In0I* [*intro*]: (*M*,*M′*) ∈ *r* ⟹ (*In0*(*M*), *In0*(*M′*)) ∈ *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsum-In1I* [*intro*]: (*N*,*N′*) ∈ *s* ⟹ (*In1*(*N*), *In1*(*N′*)) ∈ *dsum r s*
by (*auto simp add*: *dsum-def*)

lemma *dsumE* [*elim*!]:
⟦*w* ∈ *dsum r s*;
⋀*x x′*. ⟦ (*x*,*x′*) ∈ *r*; *w* = (*In0*(*x*), *In0*(*x′*)) ⟧ ⟹ *P*;
⋀*y y′*. ⟦ (*y*,*y′*) ∈ *s*; *w* = (*In1*(*y*), *In1*(*y′*)) ⟧ ⟹ *P*
⟧ ⟹ *P*
by (*auto simp add*: *dsum-def*)

lemma *dprod-mono*: [| *r*<=*r′*; *s*<=*s′* |] ==> *dprod r s* <= *dprod r′ s′*
by *blast*

lemma *dsum-mono*: [| *r*<=*r′*; *s*<=*s′* |] ==> *dsum r s* <= *dsum r′ s′*
by *blast*

lemma *dprod-Sigma*: (*dprod* (*A* × *B*) (*C* × *D*)) <= (*uprod A C*) × (*uprod B D*)
by *blast*

lemmas *dprod-subset-Sigma* = *subset-trans* [*OF dprod-mono dprod-Sigma*]

lemma *dprod-subset-Sigma2*:
(*dprod* (*Sigma A B*) (*Sigma C D*)) <= *Sigma* (*uprod A C*) (*Split* (%*x y*. *uprod* (*B x*) (*D y*)))
by *auto*

lemma *dsum-Sigma*: (*dsum* (*A* × *B*) (*C* × *D*)) <= (*usum A C*) × (*usum B D*)

by *blast*

lemmas *dsum-subset-Sigma = subset-trans [OF dsum-mono dsum-Sigma]*

lemma *Domain-dprod [simp]*: *Domain (dprod r s) = uprod (Domain r) (Domain s)*
by *auto*

lemma *Domain-dsum [simp]*: *Domain (dsum r s) = usum (Domain r) (Domain s)*
by *auto*

hides popular names

hide-type (**open**) *node item*
hide-const (**open**) *Push Node Atom Leaf Numb Lim Split Case*

ML-file ‹~~/*src/HOL/Tools/Old-Datatype/old-datatype.ML*›

end

18 Bijections between natural numbers and other types

theory *Nat-Bijection*
imports *Main*
begin

18.1 Type *nat* × *nat*

Triangle numbers: 0, 1, 3, 6, 10, 15, ...

definition *triangle* :: *nat* ⇒ *nat*
where *triangle n = (n * Suc n) div 2*

lemma *triangle-0 [simp]*: *triangle 0 = 0*
by (*simp add*: *triangle-def*)

lemma *triangle-Suc [simp]*: *triangle (Suc n) = triangle n + Suc n*
by (*simp add*: *triangle-def*)

Cantor pairing of natural numbers

definition *prod-encode* :: *nat* × *nat* ⇒ *nat*
where *prod-encode* = (λ(*m*, *n*). *triangle* (*m* + *n*) + *m*)

In this auxiliary function, *triangle k* + *m* is an invariant.

fun *prod-decode-aux* :: *nat* ⇒ *nat* ⇒ *nat* × *nat*

where *prod-decode-aux k m* =
(*if m* ≤ *k then* (*m*, *k* − *m*) *else prod-decode-aux* (*Suc k*) (*m* − *Suc k*))

declare *prod-decode-aux.simps* [*simp del*]

definition *prod-decode* :: *nat* ⇒ *nat* × *nat*
where *prod-decode* = *prod-decode-aux 0*

lemma *prod-encode-prod-decode-aux*: *prod-encode* (*prod-decode-aux k m*) = *triangle k* + *m*
proof (*induction k m rule*: *prod-decode-aux.induct*)
case (*1 k m*)
then show *?case*
by (*simp add*: *prod-encode-def prod-decode-aux.simps*)
qed

lemma *prod-decode-inverse* [*simp*]: *prod-encode* (*prod-decode n*) = *n*
by (*simp add*: *prod-decode-def prod-encode-prod-decode-aux*)

lemma *prod-decode-triangle-add*: *prod-decode* (*triangle k* + *m*) = *prod-decode-aux k m*
proof (*induct k arbitrary*: *m*)
case *0*
then show *?case*
by (*simp add*: *prod-decode-def*)
next
case (*Suc k*)
then show *?case*
by (*metis add.assoc add-diff-cancel-left' le-add1 not-less-eq-eq prod-decode-aux.simps triangle-Suc*)
qed

lemma *prod-encode-inverse* [*simp*]: *prod-decode* (*prod-encode x*) = *x*
unfolding *prod-encode-def*
proof (*induct x*)
case (*Pair a b*)
then show *?case*
by (*simp add*: *prod-decode-triangle-add prod-decode-aux.simps*)
qed

lemma *inj-prod-encode*: *inj-on prod-encode A*
by (*rule inj-on-inverseI*) (*rule prod-encode-inverse*)

lemma *inj-prod-decode*: *inj-on prod-decode A*
by (*rule inj-on-inverseI*) (*rule prod-decode-inverse*)

lemma *surj-prod-encode*: *surj prod-encode*
by (*rule surjI*) (*rule prod-decode-inverse*)

lemma *surj-prod-decode*: *surj prod-decode*
by (*rule surjI*) (*rule prod-encode-inverse*)

lemma *bij-prod-encode*: *bij prod-encode*
by (*rule bijI* [*OF inj-prod-encode surj-prod-encode*])

lemma *bij-prod-decode*: *bij prod-decode*
by (*rule bijI* [*OF inj-prod-decode surj-prod-decode*])

lemma *prod-encode-eq* [*simp*]: *prod-encode* x = *prod-encode* $y \longleftrightarrow x = y$
by (*rule inj-prod-encode* [*THEN inj-eq*])

lemma *prod-decode-eq* [*simp*]: *prod-decode* x = *prod-decode* $y \longleftrightarrow x = y$
by (*rule inj-prod-decode* [*THEN inj-eq*])

Ordering properties

lemma *le-prod-encode-1*: $a \leq$ *prod-encode* (a, b)
by (*simp add*: *prod-encode-def*)

lemma *le-prod-encode-2*: $b \leq$ *prod-encode* (a, b)
by (*induct b*) (*simp-all add*: *prod-encode-def*)

lemma *prod-encode-mono1*:
assumes $a' \leq a$
shows *prod-encode* $(a', b) \leq$ *prod-encode* (a, b)
using *assms*
by (*induction a*; *fastforce simp*: *le-Suc-eq prod-encode-def*)

lemma *prod-encode-mono2*:
assumes $b' \leq b$
shows *prod-encode* $(a, b') \leq$ *prod-encode* (a, b)
using *assms*
by (*induction b*; *fastforce simp*: *le-Suc-eq prod-encode-def*)

lemma *prod-encode-mono*:
assumes $a' \leq a$ $b' \leq b$
shows *prod-encode* $(a', b') \leq$ *prod-encode* (a, b)
by (*meson assms le-trans prod-encode-mono1 prod-encode-mono2*)

18.2 Type *nat* + *nat*

definition *sum-encode* :: *nat* + *nat* ⇒ *nat*
where *sum-encode x* = (*case x of Inl a* ⇒ *2* ∗ *a* | *Inr b* ⇒ *Suc* (*2* ∗ *b*))

definition *sum-decode* :: *nat* ⇒ *nat* + *nat*
where *sum-decode n* = (*if even n then Inl* (*n div 2*) *else Inr* (*n div 2*))

lemma *sum-encode-inverse* [*simp*]: *sum-decode* (*sum-encode x*) = *x*
by (*induct x*) (*simp-all add*: *sum-decode-def sum-encode-def*)

lemma *sum-decode-inverse* [*simp*]: *sum-encode* (*sum-decode n*) = *n*
by (*simp add*: *even-two-times-div-two sum-decode-def sum-encode-def*)

lemma *inj-sum-encode*: *inj-on sum-encode A*
by (*rule inj-on-inverseI*) (*rule sum-encode-inverse*)

lemma *inj-sum-decode*: *inj-on sum-decode A*
by (*rule inj-on-inverseI*) (*rule sum-decode-inverse*)

lemma *surj-sum-encode*: *surj sum-encode*
by (*rule surjI*) (*rule sum-decode-inverse*)

lemma *surj-sum-decode*: *surj sum-decode*
by (*rule surjI*) (*rule sum-encode-inverse*)

lemma *bij-sum-encode*: *bij sum-encode*
by (*rule bijI* [*OF inj-sum-encode surj-sum-encode*])

lemma *bij-sum-decode*: *bij sum-decode*
by (*rule bijI* [*OF inj-sum-decode surj-sum-decode*])

lemma *sum-encode-eq*: *sum-encode x* = *sum-encode y* ⟷ *x* = *y*
by (*rule inj-sum-encode* [*THEN inj-eq*])

lemma *sum-decode-eq*: *sum-decode x* = *sum-decode y* ⟷ *x* = *y*
by (*rule inj-sum-decode* [*THEN inj-eq*])

18.3 Type *int*

definition *int-encode* :: *int* ⇒ *nat*
where *int-encode i* = *sum-encode* (*if 0* ≤ *i then Inl* (*nat i*) *else Inr* (*nat* (− *i* − *1*)))

definition *int-decode* :: *nat* ⇒ *int*
where *int-decode n* = (*case sum-decode n of Inl a* ⇒ *int a* | *Inr b* ⇒ − *int b* − *1*)

lemma *int-encode-inverse* [*simp*]: *int-decode* (*int-encode x*) = *x*
by (*simp add*: *int-decode-def int-encode-def*)

lemma *int-decode-inverse* [*simp*]: *int-encode* (*int-decode n*) = *n*
unfolding *int-decode-def int-encode-def*
using *sum-decode-inverse* [*of n*] **by** (*cases sum-decode n*) *simp-all*

lemma *inj-int-encode*: *inj-on int-encode A*
by (*rule inj-on-inverseI*) (*rule int-encode-inverse*)

lemma *inj-int-decode*: *inj-on int-decode A*
by (*rule inj-on-inverseI*) (*rule int-decode-inverse*)

lemma *surj-int-encode*: *surj int-encode*
by (*rule surjI*) (*rule int-decode-inverse*)

lemma *surj-int-decode*: *surj int-decode*
by (*rule surjI*) (*rule int-encode-inverse*)

lemma *bij-int-encode*: *bij int-encode*
by (*rule bijI* [*OF inj-int-encode surj-int-encode*])

lemma *bij-int-decode*: *bij int-decode*
by (*rule bijI* [*OF inj-int-decode surj-int-decode*])

lemma *int-encode-eq*: *int-encode x* = *int-encode y* ⟷ *x* = *y*
by (*rule inj-int-encode* [*THEN inj-eq*])

lemma *int-decode-eq*: *int-decode x* = *int-decode y* ⟷ *x* = *y*
by (*rule inj-int-decode* [*THEN inj-eq*])

18.4 Type *nat list*

fun *list-encode* :: *nat list* ⇒ *nat*
where
list-encode [] = *0*
| *list-encode* (*x* # *xs*) = *Suc* (*prod-encode* (*x*, *list-encode xs*))

function *list-decode* :: *nat* ⇒ *nat list*
where
list-decode 0 = []
| *list-decode* (*Suc n*) = (*case prod-decode n of* (*x*, *y*) ⇒ *x* # *list-decode y*)
by *pat-completeness auto*

termination *list-decode*
proof −
have ⋀*n x y*. (*x*, *y*) = *prod-decode n* ⟹ *y* < *Suc n*
by (*metis le-imp-less-Suc le-prod-encode-2 prod-decode-inverse*)
then show *?thesis*
using *termination* **by** *blast*
qed

lemma *list-encode-inverse* [*simp*]: *list-decode* (*list-encode x*) = *x*
by (*induct x rule*: *list-encode.induct*) *simp-all*

lemma *list-decode-inverse* [*simp*]: *list-encode* (*list-decode n*) = *n*
proof (*induct n rule*: *list-decode.induct*)
case (*2 n*)
then show *?case*
by (*metis list-encode.simps*(*2*) *list-encode-inverse prod-decode-inverse surj-pair*)
qed *auto*

lemma *inj-list-encode*: *inj-on list-encode A*
by (*rule inj-on-inverseI*) (*rule list-encode-inverse*)

lemma *inj-list-decode*: *inj-on list-decode A*
by (*rule inj-on-inverseI*) (*rule list-decode-inverse*)

lemma *surj-list-encode*: *surj list-encode*
by (*rule surjI*) (*rule list-decode-inverse*)

lemma *surj-list-decode*: *surj list-decode*
by (*rule surjI*) (*rule list-encode-inverse*)

lemma *bij-list-encode*: *bij list-encode*
by (*rule bijI* [*OF inj-list-encode surj-list-encode*])

lemma *bij-list-decode*: *bij list-decode*
by (*rule bijI* [*OF inj-list-decode surj-list-decode*])

lemma *list-encode-eq*: *list-encode x* = *list-encode y* ⟷ *x* = *y*
by (*rule inj-list-encode* [*THEN inj-eq*])

lemma *list-decode-eq*: *list-decode x* = *list-decode y* ⟷ *x* = *y*
by (*rule inj-list-decode* [*THEN inj-eq*])

18.5 Finite sets of naturals

18.5.1 Preliminaries

lemma *finite-vimage-Suc-iff*: *finite* (*Suc* −‘ *F*) ⟷ *finite F*
proof
have *F* ⊆ *insert 0* (*Suc* ‘ *Suc* −‘ *F*)
using *nat.nchotomy* **by** *force*
moreover
assume *finite* (*Suc* −‘ *F*)
then have *finite* (*insert 0* (*Suc* ‘ *Suc* −‘ *F*))
by *blast*
ultimately show *finite F*
using *finite-subset* **by** *blast*
qed (*force intro*: *finite-vimageI inj-Suc*)

lemma *vimage-Suc-insert-0*: *Suc* −‘ *insert 0 A* = *Suc* −‘ *A*
by *auto*

lemma *vimage-Suc-insert-Suc*: *Suc* −‘ *insert* (*Suc n*) *A* = *insert n* (*Suc* −‘ *A*)
by *auto*

lemma *div2-even-ext-nat*:
fixes *x y* :: *nat*
assumes *x div 2* = *y div 2*

```
    and even x ⟷ even y
  shows x = y
proof −
  from ‹even x ⟷ even y› have x mod 2 = y mod 2
    by (simp only: even-iff-mod-2-eq-zero) auto
  with assms have x div 2 * 2 + x mod 2 = y div 2 * 2 + y mod 2
    by simp
  then show ?thesis
    by simp
qed
```

18.5.2 From sets to naturals

```
definition set-encode :: nat set ⇒ nat
  where set-encode = sum ((^) 2)

lemma set-encode-empty [simp]: set-encode {} = 0
  by (simp add: set-encode-def)

lemma set-encode-inf: ¬ finite A ⟹ set-encode A = 0
  by (simp add: set-encode-def)

lemma set-encode-insert [simp]: finite A ⟹ n ∉ A ⟹ set-encode (insert n A)
= 2^n + set-encode A
  by (simp add: set-encode-def)

lemma even-set-encode-iff: finite A ⟹ even (set-encode A) ⟷ 0 ∉ A
  by (induct set: finite) (auto simp: set-encode-def)

lemma set-encode-vimage-Suc: set-encode (Suc −‘ A) = set-encode A div 2
proof (induction A rule: infinite-finite-induct)
  case (infinite A)
  then show ?case
    by (simp add: finite-vimage-Suc-iff set-encode-inf)
next
  case (insert x A)
  show ?case
  proof (cases x)
    case 0
    with insert show ?thesis
      by (simp add: even-set-encode-iff vimage-Suc-insert-0)
  next
    case (Suc y)
    with insert show ?thesis
      by (simp add: finite-vimageI add.commute vimage-Suc-insert-Suc)
  qed
qed auto

lemmas set-encode-div-2 = set-encode-vimage-Suc [symmetric]
```

18.5.3 From naturals to sets

definition *set-decode* :: *nat* ⇒ *nat set*
where *set-decode x* = {*n*. *odd* (*x div 2* ^ *n*)}

lemma *set-decode-0* [*simp*]: *0* ∈ *set-decode x* ⟷ *odd x*
by (*simp add*: *set-decode-def*)

lemma *set-decode-Suc* [*simp*]: *Suc n* ∈ *set-decode x* ⟷ *n* ∈ *set-decode* (*x div 2*)
by (*simp add*: *set-decode-def div-mult2-eq*)

lemma *set-decode-zero* [*simp*]: *set-decode 0* = {}
by (*simp add*: *set-decode-def*)

lemma *set-decode-div-2*: *set-decode* (*x div 2*) = *Suc* −‘ *set-decode x*
by *auto*

lemma *set-decode-plus-power-2*:
n ∉ *set-decode z* ⟹ *set-decode* (*2* ^ *n* + *z*) = *insert n* (*set-decode z*)
proof (*induct n arbitrary*: *z*)
case *0*
show *?case*
proof (*rule set-eqI*)
show *q* ∈ *set-decode* (*2* ^ *0* + *z*) ⟷ *q* ∈ *insert 0* (*set-decode z*) **for** *q*
by (*induct q*) (*use 0* **in** *simp-all*)
qed
next
case (*Suc n*)
show *?case*
proof (*rule set-eqI*)
show *q* ∈ *set-decode* (*2* ^ *Suc n* + *z*) ⟷ *q* ∈ *insert* (*Suc n*) (*set-decode z*) **for** *q*
by (*induct q*) (*use Suc* **in** *simp-all*)
qed
qed

lemma *finite-set-decode* [*simp*]: *finite* (*set-decode n*)
proof (*induction n rule*: *less-induct*)
case (*less n*)
show *?case*
proof (*cases n* = *0*)
case *False*
then show *?thesis*
using *less.IH* [*of n div 2*] *finite-vimage-Suc-iff set-decode-div-2* **by** *auto*
qed *auto*
qed

18.5.4 Proof of isomorphism

lemma *set-decode-inverse* [*simp*]: *set-encode* (*set-decode n*) = *n*

proof (*induction n rule*: *less-induct*)
case (*less n*)
show *?case*
proof (*cases n = 0*)
case *False*
then have *set-encode* (*set-decode* (*n div 2*)) = *n div 2*
using *less.IH* **by** *auto*
then show *?thesis*
by (*metis div2-even-ext-nat even-set-encode-iff finite-set-decode set-decode-0 set-decode-div-2 set-encode-div-2*)
qed *auto*
qed

lemma *set-encode-inverse* [*simp*]: *finite A* ⟹ *set-decode* (*set-encode A*) = *A*
proof (*induction rule*: *finite-induct*)
case (*insert x A*)
then show *?case*
by (*simp add*: *set-decode-plus-power-2*)
qed *auto*

lemma *inj-on-set-encode*: *inj-on set-encode* (*Collect finite*)
by (*rule inj-on-inverseI* [**where** *g* = *set-decode*]) *simp*

lemma *set-encode-eq*: *finite A* ⟹ *finite B* ⟹ *set-encode A* = *set-encode B* ⟷ *A* = *B*
by (*rule iffI*) (*simp-all add*: *inj-onD* [*OF inj-on-set-encode*])

lemma *subset-decode-imp-le*:
assumes *set-decode m* ⊆ *set-decode n*
shows *m* ≤ *n*
proof −
have *n* = *m* + *set-encode* (*set-decode n* − *set-decode m*)
proof −
obtain *A B* **where**
m = *set-encode A finite A*
n = *set-encode B finite B*
by (*metis finite-set-decode set-decode-inverse*)
with *assms* **show** *?thesis*
by (*metis add.commute set-encode-def set-encode-inverse sum.subset-diff*)
qed
then show *?thesis*
by (*metis le-add1*)
qed

end

19 Encoding (almost) everything into natural numbers

theory *Countable*
imports *Old-Datatype HOL.Rat Nat-Bijection*
begin

19.1 The class of countable types

class *countable* =
  assumes *ex-inj*: $\exists$ *to-nat* :: *'a* $\Rightarrow$ *nat*. *inj to-nat*

lemma *countable-classI*:
  fixes *f* :: *'a* $\Rightarrow$ *nat*
  assumes $\bigwedge x\ y.\ f\ x = f\ y \Longrightarrow x = y$
  shows *OFCLASS*(*'a*, *countable-class*)
proof (*intro-classes*, *rule exI*)
  show *inj f*
    by (*rule injI* [*OF assms*]) *assumption*
qed

19.2 Conversion functions

definition *to-nat* :: *'a*::*countable* $\Rightarrow$ *nat* **where**
  to-nat = (*SOME f*. *inj f*)

definition *from-nat* :: *nat* $\Rightarrow$ *'a*::*countable* **where**
  from-nat = *inv* (*to-nat* :: *'a* $\Rightarrow$ *nat*)

lemma *inj-to-nat* [*simp*]: *inj to-nat*
  by (*metis ex-inj someI-ex to-nat-def*)

lemma *inj-on-to-nat*[*simp*, *intro*]: *inj-on to-nat S*
  using *inj-to-nat* **by** (*auto simp*: *inj-on-def*)

lemma *surj-from-nat* [*simp*]: *surj from-nat*
  unfolding *from-nat-def* **by** (*simp add*: *inj-imp-surj-inv*)

lemma *to-nat-split* [*simp*]: *to-nat x* = *to-nat y* $\longleftrightarrow$ *x* = *y*
  using *injD* [*OF inj-to-nat*] **by** *auto*

lemma *from-nat-to-nat* [*simp*]:
  from-nat (*to-nat x*) = *x*
  by (*simp add*: *from-nat-def*)

19.3 Finite types are countable

subclass (**in** *finite*) *countable*
proof

obtain n **and** $f :: nat \Rightarrow 'a$
where $UNIV = f \text{ ` } \{i.\ i < n\}$
by (*metis finite-conv-nat-seg-image local.finite*)
then show $\exists$ *to-nat* $:: 'a \Rightarrow nat.$ *inj to-nat*
by (*meson finite-imp-inj-to-nat-seg local.finite*)
qed

19.4 Automatically proving countability of old-style datatypes

context
begin

qualified inductive *finite-item* $:: 'a$ *Old-Datatype.item* $\Rightarrow$ *bool* **where**
undefined: *finite-item undefined*
| *In0*: *finite-item* $x \Longrightarrow$ *finite-item* (*Old-Datatype.In0* x)
| *In1*: *finite-item* $x \Longrightarrow$ *finite-item* (*Old-Datatype.In1* x)
| *Leaf*: *finite-item* (*Old-Datatype.Leaf* a)
| *Scons*: $[\![$*finite-item* x; *finite-item* $y]\!] \Longrightarrow$ *finite-item* (*Old-Datatype.Scons* x y)

qualified function *nth-item* $::$ *nat* $\Rightarrow$ ($'a$::*countable*) *Old-Datatype.item*
where
nth-item 0 $=$ *undefined*
| *nth-item* (*Suc* n) $=$
(*case sum-decode* n *of*
Inl $i \Rightarrow$
(*case sum-decode* i *of*
Inl $j \Rightarrow$ *Old-Datatype.In0* (*nth-item* j)
| *Inr* $j \Rightarrow$ *Old-Datatype.In1* (*nth-item* j))
| *Inr* $i \Rightarrow$
(*case sum-decode* i *of*
Inl $j \Rightarrow$ *Old-Datatype.Leaf* (*from-nat* j)
| *Inr* $j \Rightarrow$
(*case prod-decode* j *of*
(a, b) $\Rightarrow$ *Old-Datatype.Scons* (*nth-item* a) (*nth-item* b))))
by *pat-completeness auto*

lemma *le-sum-encode-Inl*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inl* y)
unfolding *sum-encode-def* **by** *simp*

lemma *le-sum-encode-Inr*: $x \leq y \Longrightarrow x \leq$ *sum-encode* (*Inr* y)
unfolding *sum-encode-def* **by** *simp*

qualified termination
by (*relation measure id*)
(*auto simp flip*: *sum-encode-eq prod-encode-eq*
simp: *le-imp-less-Suc le-sum-encode-Inl le-sum-encode-Inr*
le-prod-encode-1 le-prod-encode-2)

lemma *nth-item-covers*: *finite-item* $x \Longrightarrow \exists n.$ *nth-item* $n = x$

proof (*induct set*: *finite-item*)
case *undefined*
have *nth-item 0 = undefined* **by** *simp*
thus *?case* **..**
next
case (*In0 x*)
then obtain *n* **where** *nth-item n = x* **by** *fast*
hence *nth-item* (*Suc* (*sum-encode* (*Inl* (*sum-encode* (*Inl n*))))) = *Old-Datatype.In0 x* **by** *simp*
thus *?case* **..**
next
case (*In1 x*)
then obtain *n* **where** *nth-item n = x* **by** *fast*
hence *nth-item* (*Suc* (*sum-encode* (*Inl* (*sum-encode* (*Inr n*))))) = *Old-Datatype.In1 x* **by** *simp*
thus *?case* **..**
next
case (*Leaf a*)
have *nth-item* (*Suc* (*sum-encode* (*Inr* (*sum-encode* (*Inl* (*to-nat a*)))))) = *Old-Datatype.Leaf a*
by *simp*
thus *?case* **..**
next
case (*Scons x y*)
then obtain *i j* **where** *nth-item i = x* **and** *nth-item j = y* **by** *fast*
hence *nth-item*
(*Suc* (*sum-encode* (*Inr* (*sum-encode* (*Inr* (*prod-encode* (*i*, *j*))))))) = *Old-Datatype.Scons x y*
by *simp*
thus *?case* **..**
qed

theorem *countable-datatype*:
fixes *Rep* :: *'b* ⇒ (*'a*::*countable*) *Old-Datatype.item*
fixes *Abs* :: (*'a*::*countable*) *Old-Datatype.item* ⇒ *'b*
fixes *rep-set* :: (*'a*::*countable*) *Old-Datatype.item* ⇒ *bool*
assumes *type*: *type-definition Rep Abs* (*Collect rep-set*)
assumes *finite-item*: ⋀*x*. *rep-set x* ⟹ *finite-item x*
shows *OFCLASS*(*'b*, *countable-class*)
proof
define *f* **where** *f y* = (*LEAST n*. *nth-item n* = *Rep y*) **for** *y*
{
fix *y* :: *'b*
have *rep-set* (*Rep y*)
using *type-definition.Rep* [*OF type*] **by** *simp*
hence *Abs* (*nth-item* (*f y*)) = *y*
unfolding *f-def*
by (*metis* (*mono-tags*, *lifting*) *LeastI finite-item nth-item-covers type type-definition.Rep-inverse*)

```
  }
  thus ∃f::'b ⇒ nat. inj f
    by (metis injI)
qed

ML ‹
  fun old-countable-datatype-tac ctxt =
    SUBGOAL (fn (goal, -) =>
      let
        val ty-name =
          (case goal of
             (- $ Const (const-name ‹Pure.type›, Type (type-name ‹itself›, [Type
(n, -)]))) => n
           | - => raise Match)
        val typedef-info = hd (Typedef.get-info ctxt ty-name)
        val typedef-thm = #type-definition (snd typedef-info)
        val pred-name =
          (case HOLogic.dest-Trueprop (Thm.concl-of typedef-thm) of
            (- $ - $ - $ (- $ Const (n, -))) => n
          | - => raise Match)
        val induct-info = Inductive.the-inductive-global ctxt pred-name
        val pred-names = #names (fst induct-info)
        val induct-thms = #inducts (snd induct-info)
        val alist = pred-names ~~ induct-thms
        val induct-thm = the (AList.lookup (op =) alist pred-name)
        val vars = rev (Term.add-vars (Thm.prop-of induct-thm) [])
        val insts = vars |> map (fn (-, T) => try (Thm.cterm-of ctxt)
          (Const (const-name ‹Countable.finite-item›, T)))
        val induct-thm' = Thm.instantiate' [] insts induct-thm
        val rules = @{thms finite-item.intros}
      in
        SOLVED' (fn i => EVERY
          [resolve-tac ctxt @{thms countable-datatype} i,
           resolve-tac ctxt [typedef-thm] i,
           eresolve-tac ctxt [induct-thm'] i,
           REPEAT (resolve-tac ctxt rules i ORELSE assume-tac ctxt i)]) 1
      end)
›

end
```

19.5 Automatically proving countability of datatypes

ML-file ‹../Tools/BNF/bnf-lfp-countable.ML›

```
ML ‹
fun countable-datatype-tac ctxt st =
  (case try ‹HEADGOAL (old-countable-datatype-tac ctxt) st› of
    SOME res => res
```

```
  | NONE => BNF-LFP-Countable.countable-datatype-tac ctxt st);

(* compatibility *)
fun countable-tac ctxt =
  SELECT-GOAL (countable-datatype-tac ctxt);
›

method-setup countable-datatype = ‹
  Scan.succeed (SIMPLE-METHOD o countable-datatype-tac)
› prove countable class instances for datatypes
```

19.6 More Countable types

Naturals

```
instance nat :: countable
  by (rule countable-classI [of id]) simp
```

Pairs

```
instance prod :: (countable, countable) countable
  by (rule countable-classI [of λ(x, y). prod-encode (to-nat x, to-nat y)])
    (auto simp add: prod-encode-eq)
```

Sums

```
instance sum :: (countable, countable) countable
  by (rule countable-classI [of (λx. case x of Inl a ⇒ to-nat (False, to-nat a)
                                  | Inr b ⇒ to-nat (True, to-nat b))])
    (simp split: sum.split-asm)
```

Integers

```
instance int :: countable
  by (rule countable-classI [of int-encode]) (simp add: int-encode-eq)
```

Options

```
instance option :: (countable) countable
  by countable-datatype
```

Lists

```
instance list :: (countable) countable
  by countable-datatype
```

String literals

```
instance String.literal :: countable
  by (rule countable-classI [of to-nat ∘ String.explode]) (simp add: String.explode-inject)
```

Functions

```
instance fun :: (finite, countable) countable
proof
  obtain xs :: 'a list where xs: set xs = UNIV
```

using *finite-list* [*OF finite-UNIV*] **..**
show ∃ *to-nat*::('*a* ⇒ '*b*) ⇒ *nat*. *inj to-nat*
proof
show *inj* (λ*f*. *to-nat* (*map f xs*))
by (*metis* (*no-types*, *lifting*) *ext UNIV-I injI map-eq-conv to-nat-split xs*)
qed
qed

Typereps

instance *typerep* :: *countable*
by *countable-datatype*

19.7 The rationals are countably infinite

definition *nat-to-rat-surj* :: *nat* ⇒ *rat* **where**
nat-to-rat-surj n = (*let* (*a*, *b*) = *prod-decode n in Fract* (*int-decode a*) (*int-decode b*))

lemma *surj-nat-to-rat-surj*: *surj nat-to-rat-surj*
unfolding *surj-def*
proof
fix *r*::*rat*
show ∃ *n*. *r* = *nat-to-rat-surj n*
proof (*cases r*)
fix *i j* **assume** [*simp*]: *r* = *Fract i j* **and** *j* > *0*
have *r* = (*let m* = *int-encode i*; *n* = *int-encode j in nat-to-rat-surj* (*prod-encode* (*m*, *n*)))
by (*simp add*: *Let-def nat-to-rat-surj-def*)
thus ∃ *n*. *r* = *nat-to-rat-surj n* **by**(*auto simp*: *Let-def*)
qed
qed

lemma *Rats-eq-range-nat-to-rat-surj*: ℚ = *range nat-to-rat-surj*
by (*simp add*: *Rats-def surj-nat-to-rat-surj*)

context *field-char-0*
begin

lemma *Rats-eq-range-of-rat-o-nat-to-rat-surj*:
ℚ = *range* (*of-rat* ∘ *nat-to-rat-surj*)
using *surj-nat-to-rat-surj*
by (*auto simp*: *Rats-def image-def surj-def*) (*blast intro*: *arg-cong*[**where** *f* = *of-rat*])

lemma *surj-of-rat-nat-to-rat-surj*:
r ∈ ℚ ⟹ ∃ *n*. *r* = *of-rat* (*nat-to-rat-surj n*)
by (*simp add*: *Rats-eq-range-of-rat-o-nat-to-rat-surj image-def*)

end

instance *rat* :: *countable*
proof
show ∃ *to-nat*::*rat* ⇒ *nat*. *inj to-nat*
using *surj-imp-inj-inv surj-nat-to-rat-surj* **by** *blast*
qed

theorem *rat-denum*: ∃ *f* :: *nat* ⇒ *rat*. *surj f*
using *surj-nat-to-rat-surj* **by** *metis*

end

20 Infinite Sets and Related Concepts

theory *Infinite-Set*
imports *Main*
begin

20.1 The set of natural numbers is infinite

lemma *infinite-nat-iff-unbounded-le*: *infinite S* ⟷ (∀ *m*. ∃ *n*≥*m*. *n* ∈ *S*)
for *S* :: *nat set*
using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-sequentially*)

lemma *infinite-nat-iff-unbounded*: *infinite S* ⟷ (∀ *m*. ∃ *n*>*m*. *n* ∈ *S*)
for *S* :: *nat set*
using *frequently-cofinite*[*of* λ*x*. *x* ∈ *S*]
by (*simp add*: *cofinite-eq-sequentially frequently-def eventually-at-top-dense*)

lemma *finite-nat-iff-bounded*: *finite S* ⟷ (∃ *k*. *S* ⊆ {..<*k*})
for *S* :: *nat set*
using *infinite-nat-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-iff-bounded-le*: *finite S* ⟷ (∃ *k*. *S* ⊆ {.. *k*})
for *S* :: *nat set*
using *infinite-nat-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *finite-nat-bounded*: *finite S* ⟹ ∃ *k*. *S* ⊆ {..<*k*}
for *S* :: *nat set*
by (*simp add*: *finite-nat-iff-bounded*)

For a set of natural numbers to be infinite, it is enough to know that for any number larger than some *k*, there is some larger number that is an element of the set.

lemma *unbounded-k-infinite*: ∀ *m*>*k*. ∃ *n*>*m*. *n* ∈ *S* ⟹ *infinite* (*S*::*nat set*)
by (*metis finite-nat-set-iff-bounded gt-ex order-less-not-sym order-less-trans*)

lemma *nat-not-finite*: *finite* (*UNIV*::*nat set*) ⟹ *R*
by *simp*

lemma *range-inj-infinite*:
fixes $f :: nat \Rightarrow 'a$
assumes *inj f*
shows *infinite* (*range f*)
proof
assume *finite* (*range f*)
from *this assms* **have** *finite* (*UNIV*::*nat set*)
by (*rule finite-imageD*)
then show *False* **by** *simp*
qed

20.2 The set of integers is also infinite

lemma *infinite-int-iff-infinite-nat-abs*: *infinite* $S \longleftrightarrow$ *infinite* ((*nat* $\circ$ *abs*) ' *S*)
for *S* :: *int set*
proof (*unfold Not-eq-iff*, *rule iffI*)
assume *finite* ((*nat* $\circ$ *abs*) ' *S*)
then have *finite* (*nat* ' (*abs* ' *S*))
by (*simp add*: *image-image cong*: *image-cong*)
moreover have *inj-on nat* (*abs* ' *S*)
by (*rule inj-onI*) *auto*
ultimately have *finite* (*abs* ' *S*)
by (*rule finite-imageD*)
then show *finite S*
by (*rule finite-image-absD*)
qed *simp*

proposition *infinite-int-iff-unbounded-le*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| \geq m \wedge n \in S)$
for *S* :: *int set*
by (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded-le o-def image-def*)
(*metis abs-ge-zero nat-le-eq-zle le-nat-iff*)

proposition *infinite-int-iff-unbounded*: *infinite* $S \longleftrightarrow (\forall\, m.\ \exists\, n.\ |n| > m \wedge n \in S)$
for *S* :: *int set*
by (*simp add*: *infinite-int-iff-infinite-nat-abs infinite-nat-iff-unbounded o-def image-def*)
(*metis* (*full-types*) *nat-le-iff nat-mono not-le*)

proposition *finite-int-iff-bounded*: *finite* $S \longleftrightarrow (\exists\, k.\ abs\ `\ S \subseteq \{..<k\})$
for *S* :: *int set*
using *infinite-int-iff-unbounded-le*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

proposition *finite-int-iff-bounded-le*: *finite* $S \longleftrightarrow (\exists\, k.\ abs\ `\ S \subseteq \{..\ k\})$
for *S* :: *int set*
using *infinite-int-iff-unbounded*[*of S*] **by** (*simp add*: *subset-eq*) (*metis not-le*)

lemma *infinite-split*: — courtesy of Michael Schmidt
fixes $S :: 'a\ set$
assumes *infinite S*
obtains *A B*
where $A \subseteq S\ B \subseteq S$ *infinite A infinite B* $A \cap B = \{\}$
proof−
obtain $f :: nat \Rightarrow 'a$
where *f-inj*: *inj f* **and** *f-img*: *range* $f \subseteq S$
using *assms infinite-countable-subset* **by** *blast*
let *?A* = *range* $(\lambda n.\ f\ (2{*}n))$
let *?B* = *range* $(\lambda n.\ f\ (2{*}n{+}1))$
have *a-inf*: *infinite ?A*
using *finite-imageD*[*of* $\lambda n.\ f\ (2{*}n)$ *UNIV*] *f-inj infinite-UNIV-nat* **unfolding** *inj-def*
by *fastforce*
have *b-inf*: *infinite ?B*
using *finite-imageD*[*of* $\lambda n.\ f\ (2{*}n{+}1)$ *UNIV*] *f-inj infinite-UNIV-nat* **unfolding** *inj-def*
by *fastforce*
from *f-inj* **have** $?A \cap ?B = \{\}$ **unfolding** *inj-def disjoint-iff* **using** *double-not-eq-Suc-double*
by *auto*
from *that*[*OF - - a-inf b-inf this*] *f-img* **show** *?thesis*
by *blast*
qed

20.3 Infinitely Many and Almost All

We often need to reason about the existence of infinitely many (resp., all but finitely many) objects satisfying some predicate, so we introduce corresponding binders and their proof rules.

lemma *not-INFM* [*simp*]: $\neg\ (INFM\ x.\ P\ x) \longleftrightarrow (MOST\ x.\ \neg\ P\ x)$
by (*rule not-frequently*)

lemma *not-MOST* [*simp*]: $\neg\ (MOST\ x.\ P\ x) \longleftrightarrow (INFM\ x.\ \neg\ P\ x)$
by (*rule not-eventually*)

lemma *INFM-const* [*simp*]: $(INFM\ x{::}'a.\ P) \longleftrightarrow P \wedge$ *infinite* $(UNIV{::}'a\ set)$
by (*simp add*: *frequently-const-iff*)

lemma *MOST-const* [*simp*]: $(MOST\ x{::}'a.\ P) \longleftrightarrow P \vee$ *finite* $(UNIV{::}'a\ set)$
by (*simp add*: *eventually-const-iff*)

lemma *INFM-imp-distrib*: $(INFM\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow ((MOST\ x.\ P\ x) \longrightarrow (INFM\ x.\ Q\ x))$
by (*rule frequently-imp-iff*)

lemma *MOST-imp-iff*: $MOST\ x.\ P\ x \Longrightarrow (MOST\ x.\ P\ x \longrightarrow Q\ x) \longleftrightarrow (MOST\ x.\ Q\ x)$

by (*auto intro*: *eventually-rev-mp eventually-mono*)

lemma *INFM-conjI*: *INFM x. P x* $\Longrightarrow$ *MOST x. Q x* $\Longrightarrow$ *INFM x. P x* $\wedge$ *Q x*
by (*rule frequently-rev-mp*[*of P*]) (*auto elim*: *eventually-mono*)

Properties of quantifiers with injective functions.

lemma *INFM-inj*: *INFM x. P* (*f x*) $\Longrightarrow$ *inj f* $\Longrightarrow$ *INFM x. P x*
using *finite-vimageI*[*of* {*x. P x*} *f*] **by** (*auto simp*: *frequently-cofinite*)

lemma *MOST-inj*: *MOST x. P x* $\Longrightarrow$ *inj f* $\Longrightarrow$ *MOST x. P* (*f x*)
using *finite-vimageI*[*of* {*x.* $\neg$ *P x*} *f*] **by** (*auto simp*: *eventually-cofinite*)

Properties of quantifiers with singletons.

lemma *not-INFM-eq* [*simp*]:
$\neg$ (*INFM x. x* = *a*)
$\neg$ (*INFM x. a* = *x*)
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-neq* [*simp*]:
MOST x. x $\neq$ *a*
MOST x. a $\neq$ *x*
unfolding *eventually-cofinite* **by** *simp-all*

lemma *INFM-neq* [*simp*]:
(*INFM x*::'*a. x* $\neq$ *a*) $\longleftrightarrow$ *infinite* (*UNIV*::'*a set*)
(*INFM x*::'*a. a* $\neq$ *x*) $\longleftrightarrow$ *infinite* (*UNIV*::'*a set*)
unfolding *frequently-cofinite* **by** *simp-all*

lemma *MOST-eq* [*simp*]:
(*MOST x*::'*a. x* = *a*) $\longleftrightarrow$ *finite* (*UNIV*::'*a set*)
(*MOST x*::'*a. a* = *x*) $\longleftrightarrow$ *finite* (*UNIV*::'*a set*)
unfolding *eventually-cofinite* **by** *simp-all*

lemma *MOST-eq-imp*:
MOST x. x = *a* $\longrightarrow$ *P x*
MOST x. a = *x* $\longrightarrow$ *P x*
unfolding *eventually-cofinite* **by** *simp-all*

Properties of quantifiers over the naturals.

lemma *MOST-nat*: ($\forall_\infty n.$ *P n*) $\longleftrightarrow$ ($\exists\, m.\ \forall\, n{>}m.$ *P n*)
for *P* :: *nat* $\Rightarrow$ *bool*
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded-le subset-eq simp flip*: *not-le*)

lemma *MOST-nat-le*: ($\forall_\infty n.$ *P n*) $\longleftrightarrow$ ($\exists\, m.\ \forall\, n{\geq}m.$ *P n*)
for *P* :: *nat* $\Rightarrow$ *bool*
by (*auto simp add*: *eventually-cofinite finite-nat-iff-bounded subset-eq simp flip*: *not-le*)

lemma *INFM-nat*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall m.\ \exists n{>}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded*)

lemma *INFM-nat-le*: $(\exists_\infty n.\ P\ n) \longleftrightarrow (\forall m.\ \exists n{\geq}m.\ P\ n)$
for $P :: nat \Rightarrow bool$
by (*simp add*: *frequently-cofinite infinite-nat-iff-unbounded-le*)

lemma *MOST-INFM*: *infinite* $(UNIV::'a\ set) \Longrightarrow MOST\ x::'a.\ P\ x \Longrightarrow INFM\ x::'a.\ P\ x$
by (*simp add*: *eventually-frequently*)

lemma *MOST-Suc-iff*: $(MOST\ n.\ P\ (Suc\ n)) \longleftrightarrow (MOST\ n.\ P\ n)$
by (*simp add*: *cofinite-eq-sequentially*)

lemma *MOST-SucI*: $MOST\ n.\ P\ n \Longrightarrow MOST\ n.\ P\ (Suc\ n)$
and *MOST-SucD*: $MOST\ n.\ P\ (Suc\ n) \Longrightarrow MOST\ n.\ P\ n$
by (*simp-all add*: *MOST-Suc-iff*)

lemma *MOST-ge-nat*: $MOST\ n::nat.\ m \leq n$
by (*simp add*: *cofinite-eq-sequentially*)

— legacy names
lemma *Inf-many-def*: *Inf-many* $P \longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *Alm-all-def*: *Alm-all* $P \longleftrightarrow \neg\ (INFM\ x.\ \neg\ P\ x)$ **by** *simp*
lemma *INFM-iff-infinite*: $(INFM\ x.\ P\ x) \longleftrightarrow$ *infinite* $\{x.\ P\ x\}$ **by** (*fact frequently-cofinite*)
lemma *MOST-iff-cofinite*: $(MOST\ x.\ P\ x) \longleftrightarrow$ *finite* $\{x.\ \neg\ P\ x\}$ **by** (*fact eventually-cofinite*)
lemma *INFM-EX*: $(\exists_\infty x.\ P\ x) \Longrightarrow (\exists x.\ P\ x)$ **by** (*fact frequently-ex*)
lemma *ALL-MOST*: $\forall x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x$ **by** (*fact always-eventually*)
lemma *INFM-mono*: $\exists_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \exists_\infty x.\ Q\ x$ **by** (*fact frequently-elim1*)
lemma *MOST-mono*: $\forall_\infty x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow Q\ x) \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-mono*)
lemma *INFM-disj-distrib*: $(\exists_\infty x.\ P\ x \vee Q\ x) \longleftrightarrow (\exists_\infty x.\ P\ x) \vee (\exists_\infty x.\ Q\ x)$ **by** (*fact frequently-disj-iff*)
lemma *MOST-rev-mp*: $\forall_\infty x.\ P\ x \Longrightarrow \forall_\infty x.\ P\ x \longrightarrow Q\ x \Longrightarrow \forall_\infty x.\ Q\ x$ **by** (*fact eventually-rev-mp*)
lemma *MOST-conj-distrib*: $(\forall_\infty x.\ P\ x \wedge Q\ x) \longleftrightarrow (\forall_\infty x.\ P\ x) \wedge (\forall_\infty x.\ Q\ x)$ **by** (*fact eventually-conj-iff*)
lemma *MOST-conjI*: $MOST\ x.\ P\ x \Longrightarrow MOST\ x.\ Q\ x \Longrightarrow MOST\ x.\ P\ x \wedge Q\ x$ **by** (*fact eventually-conj*)
lemma *INFM-finite-Bex-distrib*: *finite* $A \Longrightarrow (INFM\ y.\ \exists x{\in}A.\ P\ x\ y) \longleftrightarrow (\exists x{\in}A.\ INFM\ y.\ P\ x\ y)$ **by** (*fact frequently-bex-finite-distrib*)
lemma *MOST-finite-Ball-distrib*: *finite* $A \Longrightarrow (MOST\ y.\ \forall x{\in}A.\ P\ x\ y) \longleftrightarrow (\forall x{\in}A.\ MOST\ y.\ P\ x\ y)$ **by** (*fact eventually-ball-finite-distrib*)
lemma *INFM-E*: $INFM\ x.\ P\ x \Longrightarrow (\bigwedge x.\ P\ x \Longrightarrow thesis) \Longrightarrow thesis$ **by** (*fact frequentlyE*)

lemma *MOST-I*: $(\bigwedge x.\ P\ x) \Longrightarrow MOST\ x.\ P\ x$ **by** (*rule eventuallyI*)
lemmas *MOST-iff-finiteNeg* = *MOST-iff-cofinite*

20.4 Enumeration of an Infinite Set

The set's element type must be wellordered (e.g. the natural numbers).

Could be generalized to $enumerate'\ S\ n = (SOME\ t.\ t \in s \wedge finite\ \{s \in S.\ s < t\} \wedge card\ \{s \in S.\ s < t\} = n)$.

primrec (**in** *wellorder*) *enumerate* :: *'a set* ⇒ *nat* ⇒ *'a*
  where
    enumerate-0: *enumerate S 0* = (*LEAST n. n* ∈ *S*)
  | *enumerate-Suc*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*LEAST n. n* ∈ *S*}) *n*

lemma *enumerate-Suc'*: *enumerate S* (*Suc n*) = *enumerate* (*S* − {*enumerate S 0*}) *n*
  by *simp*

lemma *enumerate-in-set*: *infinite S* ⟹ *enumerate S n* ∈ *S*
proof (*induct n arbitrary*: *S*)
  case *0*
  then show *?case*
    by (*fastforce intro*: *LeastI dest*!: *infinite-imp-nonempty*)
next
  case (*Suc n*)
  then show *?case*
    by *simp* (*metis DiffE infinite-remove*)
qed

declare *enumerate-0* [*simp del*] *enumerate-Suc* [*simp del*]

lemma *enumerate-step*: *infinite S* ⟹ *enumerate S n* < *enumerate S* (*Suc n*)
proof (*induction n arbitrary*: *S*)
  case *0*
  then have *enumerate S 0* ≤ *enumerate S* (*Suc 0*)
    by (*simp add*: *enumerate-0 Least-le enumerate-in-set*)
  moreover have *enumerate* (*S* − {*enumerate S 0*}) *0* ∈ *S* − {*enumerate S 0*}
    by (*meson 0.prems enumerate-in-set infinite-remove*)
  then have *enumerate S 0* ≠ *enumerate* (*S* − {*enumerate S 0*}) *0*
    by *auto*
  ultimately show *?case*
    by (*simp add*: *enumerate-Suc'*)
next
  case (*Suc n*)
  then show *?case*
    by (*simp add*: *enumerate-Suc'*)
qed

lemma *enumerate-mono*: *m* < *n* ⟹ *infinite S* ⟹ *enumerate S m* < *enumerate*

S n
by (*induct m n rule*: *less-Suc-induct*) (*auto intro*: *enumerate-step*)

lemma *enumerate-mono-iff* [*simp*]:
infinite S $\Longrightarrow$ *enumerate S m* $<$ *enumerate S n* $\longleftrightarrow$ $m < n$
by (*metis enumerate-mono less-asym less-linear*)

lemma *enumerate-mono-le-iff* [*simp*]:
infinite S $\Longrightarrow$ *enumerate S m* $\leq$ *enumerate S n* $\longleftrightarrow$ $m \leq n$
by (*meson enumerate-mono-iff not-le*)

lemma *le-enumerate*:
assumes *S*: *infinite S*
shows $n \leq$ *enumerate S n*
using *S*
proof (*induct n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have $n \leq$ *enumerate S n* **by** *simp*
also note *enumerate-mono*[*of n Suc n*, *OF* - ‹*infinite S*›]
finally show *?case* **by** *simp*
qed

lemma *infinite-enumerate*:
assumes *fS*: *infinite S*
shows $\exists\, r{::}nat \Rightarrow nat.$ *strict-mono r* $\wedge$ $(\forall\, n.$ *r n* $\in$ *S*$)$
unfolding *strict-mono-def*
using *enumerate-in-set*[*OF fS*] *enumerate-mono*[*of* - - *S*] *fS* **by** *blast*

lemma *enumerate-Suc''*:
fixes *S* :: $'a{::}wellorder$ *set*
assumes *infinite S*
shows *enumerate S* (*Suc n*) = (*LEAST s*. $s \in S \wedge$ *enumerate S n* $< s$)
using *assms*
proof (*induct n arbitrary*: *S*)
case *0*
then have $\forall\, s \in S.$ *enumerate S 0* $\leq s$
by (*auto simp*: *enumerate.simps intro*: *Least-le*)
then show *?case*
unfolding *enumerate-Suc' enumerate-0*[*of S* $-$ {*enumerate S 0*}]
by (*intro arg-cong*[**where** *f* = *Least*] *ext*) *auto*
next
case (*Suc n S*)
show *?case*
proof (*rule order.antisym*)
have *S*: *infinite* (*S* $-$ {*wellorder-class.enumerate S 0*})
using *Suc* **by** *auto*

show *wellorder-class.enumerate S (Suc (Suc n)) ≤ (LEAST s. s ∈ S ∧ wellorder-class.enumerate S (Suc n) < s)*
using *enumerate-mono[OF zero-less-Suc] ‹infinite S› S*
by (*smt* (*verit*, *best*) *LeastI-ex Suc.hyps enumerate-0 enumerate-Suc enumerate-in-set*
enumerate-step insertE insert-Diff linorder-not-less not-less-Least)
qed (*simp add*: *Least-le Suc.prems enumerate-in-set*)
qed

lemma *enumerate-Ex*:
fixes *S* :: *nat set*
assumes *S*: *infinite S*
and *s*: *s* ∈ *S*
shows ∃ *n*. *enumerate S n* = *s*
using *s*
proof (*induct s rule*: *less-induct*)
case (*less s*)
show *?case*
proof (*cases* ∃ *y*∈*S*. *y* < *s*)
case *True*
let *?y* = *Max* {*s'*∈*S*. *s'* < *s*}
from *True* **have** *y*: ⋀*x*. *?y* < *x* ⟷ (∀ *s'*∈*S*. *s'* < *s* ⟶ *s'* < *x*)
by (*subst Max-less-iff*) *auto*
then have *y-in*: *?y* ∈ {*s'*∈*S*. *s'* < *s*}
by (*intro Max-in*) *auto*
with *less.hyps*[*of ?y*] **obtain** *n* **where** *enumerate S n* = *?y*
by *auto*
with *S* **have** *enumerate S* (*Suc n*) = *s*
by (*auto simp*: *y less enumerate-Suc'' intro*!: *Least-equality*)
then show *?thesis* **by** *auto*
next
case *False*
then have ∀ *t*∈*S*. *s* ≤ *t* **by** *auto*
with ‹*s* ∈ *S*› **show** *?thesis*
by (*auto intro*!: *exI*[*of - 0*] *Least-equality simp*: *enumerate-0*)
qed
qed

lemma *inj-enumerate*:
fixes *S* :: *'a*::*wellorder set*
assumes *S*: *infinite S*
shows *inj* (*enumerate S*)
unfolding *inj-on-def*
proof *clarsimp*
show ⋀*x y*. *enumerate S x* = *enumerate S y* ⟹ *x* = *y*
by (*metis neq-iff enumerate-mono*[*OF - ‹infinite S›*])
qed

To generalise this, we'd need a condition that all initial segments were finite

lemma *bij-enumerate*:
fixes S :: *nat set*
assumes S: *infinite S*
shows *bij-betw* (*enumerate S*) *UNIV S*
proof −
have $\forall s \in S.\ \exists i.$ *enumerate S i* $= s$
using *enumerate-Ex*[*OF S*] **by** *auto*
moreover note ‹*infinite S*› *inj-enumerate*
ultimately show *?thesis*
unfolding *bij-betw-def* **by** (*auto intro*: *enumerate-in-set*)
qed

lemma
fixes S :: *nat set*
assumes S: *infinite S*
shows *range-enumerate*: *range* (*enumerate S*) $= S$
and *strict-mono-enumerate*: *strict-mono* (*enumerate S*)
by (*auto simp add*: *bij-betw-imp-surj-on bij-enumerate assms strict-mono-def*)

A pair of weird and wonderful lemmas from HOL Light.

lemma *finite-transitivity-chain*:
assumes *finite A*
and R: $\bigwedge x.\ \neg\ R\ x\ x\ \bigwedge x\ y\ z.\ [\![R\ x\ y;\ R\ y\ z]\!] \Longrightarrow R\ x\ z$
and A: $\bigwedge x.\ x \in A \Longrightarrow \exists y.\ y \in A \wedge R\ x\ y$
shows $A = \{\}$
using ‹*finite A*› A
proof (*induct A*)
case *empty*
then show *?case* **by** *simp*
next
case (*insert a A*)
have *False*
using *R*(*1*)[*of a*] *R*(*2*)[*of - a*] *insert*(*3*,*4*) **by** *blast*
thus *?case* **..**
qed

corollary *Union-maximal-sets*:
assumes *finite* $\mathcal{F}$
shows $\bigcup \{T \in \mathcal{F}.\ \forall U{\in}\mathcal{F}.\ \neg\ T \subset U\} = \bigcup \mathcal{F}$
(**is** *?lhs* = *?rhs*)
proof
show *?lhs* $\subseteq$ *?rhs* **by** *force*
show *?rhs* $\subseteq$ *?lhs*
proof (*rule Union-subsetI*)
fix S
assume $S \in \mathcal{F}$
have $\{T \in \mathcal{F}.\ S \subseteq T\} = \{\}$
if $\neg\ (\exists y.\ y \in \{T \in \mathcal{F}.\ \forall U{\in}\mathcal{F}.\ \neg\ T \subset U\} \wedge S \subseteq y)$
proof −

have §: $\bigwedge x.\ x \in \mathcal{F} \wedge S \subseteq x \Longrightarrow \exists\, y.\ y \in \mathcal{F} \wedge S \subseteq y \wedge x \subset y$
using *that* **by** (*blast intro*: *dual-order.trans psubset-imp-subset*)
show *?thesis*
proof (*rule finite-transitivity-chain* [*of* - $\lambda T\ U.\ S \subseteq T \wedge T \subset U$])
qed (*use assms* **in** ‹*auto intro*: §›)
qed
with ‹$S \in \mathcal{F}$› **show** $\exists\, y.\ y \in \{T \in \mathcal{F}.\ \forall\, U{\in}\mathcal{F}.\ \neg\ T \subset U\} \wedge S \subseteq y$
by *blast*
qed
qed

20.5 Properties of *enumerate* on finite sets

lemma *finite-enumerate-in-set*: $[\![$*finite* S; $n <$ *card* $S]\!] \Longrightarrow$ *enumerate* $S\ n \in S$
proof (*induction n arbitrary*: S)
case *0*
then show *?case*
by (*metis all-not-in-conv card.empty enumerate.simps*(*1*) *not-less0 wellorder-Least-lemma*(*1*))
next
case (*Suc n*)
then have *wellorder-class.enumerate* $(S - \{LEAST\ n.\ n \in S\})\ n \in S$
by (*metis Diff-empty Diff-insert0 Suc-lessD Suc-less-eq card.insert-remove*
finite-Diff insert-Diff insert-Diff-single insert-iff)
then
show *?case*
using *Suc.prems Suc.IH* [*of* $S - \{LEAST\ n.\ n \in S\}$]
by (*simp add*: *enumerate.simps*)
qed

lemma *finite-enumerate-step*: $[\![$*finite* S; *Suc* $n <$ *card* $S]\!] \Longrightarrow$ *enumerate* $S\ n <$ *enumerate* S (*Suc* n)
proof (*induction n arbitrary*: S)
case *0*
then have *enumerate* $S\ 0 \leq$ *enumerate* S (*Suc 0*)
by (*simp add*: *Least-le enumerate.simps*(*1*) *finite-enumerate-in-set*)
moreover have *enumerate* $(S - \{$*enumerate* $S\ 0\})\ 0 \in S - \{$*enumerate* $S\ 0\}$
by (*metis 0 Suc-lessD Suc-less-eq card-Suc-Diff1 enumerate-in-set finite-enumerate-in-set*)
then have *enumerate* $S\ 0 \neq$ *enumerate* $(S - \{$*enumerate* $S\ 0\})\ 0$
by *auto*
ultimately show *?case*
by (*simp add*: *enumerate-Suc′*)
next
case (*Suc n*)
then show *?case*
by (*simp add*: *enumerate-Suc′ finite-enumerate-in-set*)
qed

lemma *finite-enumerate-mono*: $[\![m < n$; *finite* S; $n <$ *card* $S]\!] \Longrightarrow$ *enumerate* $S\ m$ $<$ *enumerate* $S\ n$

by (*induct m n rule*: *less-Suc-induct*) (*auto intro*: *finite-enumerate-step*)

lemma *finite-enumerate-mono-iff* [*simp*]:
⟦*finite S*; *m* < *card S*; *n* < *card S*⟧ ⟹ *enumerate S m* < *enumerate S n* ⟷ *m* < *n*
by (*metis finite-enumerate-mono less-asym less-linear*)

lemma *finite-le-enumerate*:
assumes *finite S n* < *card S*
shows *n* ≤ *enumerate S n*
using *assms*
proof (*induction n*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
then have *n* ≤ *enumerate S n* **by** *simp*
also note *finite-enumerate-mono*[*of n Suc n*, *OF* - ‹*finite S*›]
finally show *?case*
using *Suc.prems*(*2*) *Suc-leI* **by** *blast*
qed

lemma *finite-enumerate*:
assumes *fS*: *finite S*
shows ∃ *r*::*nat*⇒*nat*. *strict-mono-on* {..<*card S*} *r* ∧ (∀ *n*<*card S*. *r n* ∈ *S*)
unfolding *strict-mono-def*
using *finite-enumerate-in-set*[*OF fS*] *finite-enumerate-mono*[*of* - - *S*] *fS*
by (*metis lessThan-iff strict-mono-on-def*)

lemma *finite-enumerate-Suc''*:
fixes *S* :: '*a*::*wellorder set*
assumes *finite S Suc n* < *card S*
shows *enumerate S* (*Suc n*) = (*LEAST s*. *s* ∈ *S* ∧ *enumerate S n* < *s*)
using *assms*
proof (*induction n arbitrary*: *S*)
case *0*
then have ∀ *s* ∈ *S*. *enumerate S 0* ≤ *s*
by (*auto simp*: *enumerate.simps intro*: *Least-le*)
then show *?case*
unfolding *enumerate-Suc' enumerate-0*[*of S* − {*enumerate S 0*}]
by (*metis Diff-iff dual-order.strict-iff-order singletonD singletonI*)
next
case (*Suc n S*)
then have *Suc n* < *card* (*S* − {*enumerate S 0*})
using *Suc.prems*(*2*) *finite-enumerate-in-set* **by** *force*
then show *?case*
apply (*subst* (*1 2*) *enumerate-Suc'*)
apply (*simp add*: *Suc*)
apply (*intro arg-cong*[**where** *f* = *Least*] *HOL.ext*)

```
    using finite-enumerate-mono[OF zero-less-Suc ‹finite S›, of n] Suc.prems
    by (auto simp flip: enumerate-Suc')
qed

lemma finite-enumerate-initial-segment:
  fixes S :: 'a::wellorder set
  assumes finite S and n: n < card (S ∩ {..<s})
  shows enumerate (S ∩ {..<s}) n = enumerate S n
  using n
proof (induction n)
  case 0
  have (LEAST n. n ∈ S ∧ n < s) = (LEAST n. n ∈ S)
  proof (rule Least-equality)
    have ∃ t. t ∈ S ∧ t < s
      by (metis 0 card-gt-0-iff disjoint-iff-not-equal lessThan-iff)
    then show (LEAST n. n ∈ S) ∈ S ∧ (LEAST n. n ∈ S) < s
      by (meson LeastI Least-le le-less-trans)
  qed (simp add: Least-le)
  then show ?case
    by (auto simp: enumerate-0)
next
  case (Suc n)
  then have less-card: Suc n < card S
    by (meson assms(1) card-mono inf-sup-ord(1) leD le-less-linear order.trans)
  obtain T where T: T ∈ {s ∈ S. enumerate S n < s}
    by (metis Infinite-Set.enumerate-step enumerate-in-set finite-enumerate-in-set
finite-enumerate-step less-card mem-Collect-eq)
  have (LEAST x. x ∈ S ∧ x < s ∧ enumerate S n < x) = (LEAST x. x ∈ S ∧
enumerate S n < x)
        (is - = ?r)
  proof (intro Least-equality conjI)
    show ?r ∈ S
      by (metis (mono-tags, lifting) LeastI mem-Collect-eq T)
    have ¬ s ≤ ?r
      using not-less-Least [of - λx. x ∈ S ∧ enumerate S n < x] Suc assms
    by (metis (mono-tags, lifting) Int-Collect Suc-lessD finite-Int finite-enumerate-in-set
finite-enumerate-step lessThan-def less-le-trans)
    then show ?r < s
      by auto
    show enumerate S n < ?r
      by (metis (no-types, lifting) LeastI mem-Collect-eq T)
  qed (auto simp: Least-le)
  then show ?case
    using Suc assms by (simp add: finite-enumerate-Suc'' less-card)
qed

lemma finite-enumerate-Ex:
  fixes S :: 'a::wellorder set
  assumes S: finite S
```

and *s*: $s \in S$
shows $\exists\, n{<}\mathit{card}\ S.\ \mathit{enumerate}\ S\ n = s$
using *s S*
proof (*induction s arbitrary*: *S rule*: *less-induct*)
case (*less s*)
show *?case*
proof (*cases* $\exists\, y{\in}S.\ y < s$)
case *True*
let *?T* = $S \cap \{..{<}s\}$
have *finite ?T*
using *less.prems*(*2*) **by** *blast*
have *TS*: *card ?T* < *card S*
using *less.prems* **by** (*blast intro*: *psubset-card-mono* [*OF* ‹*finite S*›])
from *True* **have** *y*: $\bigwedge x.\ \mathit{Max}\ ?T < x \longleftrightarrow (\forall\, s'{\in}S.\ s' < s \longrightarrow s' < x)$
by (*subst Max-less-iff*) (*auto simp*: ‹*finite ?T*›)
then have *y-in*: *Max ?T* $\in \{s'{\in}S.\ s' < s\}$
using *Max-in* ‹*finite ?T*› **by** *fastforce*
with *less.IH*[*of Max ?T ?T*] **obtain** *n* **where** *n*: *enumerate ?T n* = *Max ?T n* < *card ?T*
using ‹*finite ?T*› **by** *blast*
then have *Suc n* < *card S*
using *TS less-trans-Suc* **by** *blast*
with *S n* **have** *enumerate S* (*Suc n*) = *s*
by (*subst finite-enumerate-Suc''*) (*auto simp*: *y finite-enumerate-initial-segment less finite-enumerate-Suc'' intro*!: *Least-equality*)
then show *?thesis*
using ‹*Suc n* < *card S*› **by** *blast*
next
case *False*
then have $\forall\, t{\in}S.\ s \le t$ **by** *auto*
moreover have *0* < *card S*
using *card-0-eq less.prems* **by** *blast*
ultimately show *?thesis*
using ‹$s \in S$›
by (*auto intro*!: *exI*[*of - 0*] *Least-equality simp*: *enumerate-0*)
qed
qed

lemma *finite-enum-subset*:
assumes $\bigwedge i.\ i < \mathit{card}\ X \Longrightarrow \mathit{enumerate}\ X\ i = \mathit{enumerate}\ Y\ i$ **and** *finite X finite Y card X* ≤ *card Y*
shows $X \subseteq Y$
by (*metis assms finite-enumerate-Ex finite-enumerate-in-set less-le-trans subsetI*)

lemma *finite-enum-ext*:
assumes $\bigwedge i.\ i < \mathit{card}\ X \Longrightarrow \mathit{enumerate}\ X\ i = \mathit{enumerate}\ Y\ i$ **and** *finite X finite Y card X* = *card Y*
shows $X = Y$
by (*intro antisym finite-enum-subset*) (*auto simp*: *assms*)

lemma *finite-bij-enumerate*:
fixes S :: *'a*::*wellorder set*
assumes S: *finite S*
shows *bij-betw* (*enumerate S*) {..<*card S*} *S*
proof −
have $\bigwedge n\ m.\ [\![n \neq m;\ n < card\ S;\ m < card\ S]\!] \Longrightarrow enumerate\ S\ n \neq enumerate\ S\ m$
using *finite-enumerate-mono*[*OF* - ‹*finite S*›] **by** (*auto simp*: *neq-iff*)
then have *inj-on* (*enumerate S*) {..<*card S*}
by (*auto simp*: *inj-on-def*)
moreover have $\forall\, s \in S.\ \exists\, i{<}card\ S.\ enumerate\ S\ i = s$
using *finite-enumerate-Ex*[*OF S*] **by** *auto*
moreover note ‹*finite S*›
ultimately show *?thesis*
unfolding *bij-betw-def* **by** (*auto intro*: *finite-enumerate-in-set*)
qed

lemma *ex-bij-betw-strict-mono-card*:
fixes M :: *'a*::*wellorder set*
assumes *finite M*
obtains *h* **where** *bij-betw h* {..<*card M*} *M* **and** *strict-mono-on* {..<*card M*} *h*
proof
show *bij-betw* (*enumerate M*) {..<*card M*} *M*
by (*simp add*: *assms finite-bij-enumerate*)
show *strict-mono-on* {..<*card M*} (*enumerate M*)
by (*simp add*: *assms finite-enumerate-mono strict-mono-on-def*)
qed

end

21 Countable sets

theory *Countable-Set*
imports *Countable Infinite-Set*
begin

21.1 Predicate for countable sets

definition *countable* :: *'a set* ⇒ *bool* **where**
countable S ⟷ (∃*f*::*'a* ⇒ *nat*. *inj-on f S*)

lemma *countable-as-injective-image-subset*: *countable S* ⟷ (∃*f*. ∃*K*::*nat set*. *S* = *f* ‘ *K* ∧ *inj-on f K*)
by (*metis countable-def inj-on-the-inv-into the-inv-into-onto*)

lemma *countableE*:
assumes *S*: *countable S* **obtains** *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
using *S* **by** (*auto simp*: *countable-def*)

lemma *countableI*: *inj-on* (*f*::*'a* ⇒ *nat*) *S* ⟹ *countable S*
by (*auto simp*: *countable-def*)

lemma *countableI'*: *inj-on* (*f*::*'a* ⇒ *'b*::*countable*) *S* ⟹ *countable S*
using *comp-inj-on*[*of f S to-nat*] **by** (*auto intro*: *countableI*)

lemma *countableE-bij*:
assumes *S*: *countable S* **obtains** *f* :: *nat* ⇒ *'a* **and** *C* :: *nat set* **where** *bij-betw f C S*
using *S* **by** (*blast elim*: *countableE dest*: *inj-on-imp-bij-betw bij-betw-inv*)

lemma *countableI-bij*: *bij-betw f* (*C*::*nat set*) *S* ⟹ *countable S*
by (*blast intro*: *countableI bij-betw-inv-into bij-betw-imp-inj-on*)

lemma *countable-finite*: *finite S* ⟹ *countable S*
by (*blast dest*: *finite-imp-inj-to-nat-seg countableI*)

lemma *countableI-bij1*: *bij-betw f A B* ⟹ *countable A* ⟹ *countable B*
by (*blast elim*: *countableE-bij intro*: *bij-betw-trans countableI-bij*)

lemma *countableI-bij2*: *bij-betw f B A* ⟹ *countable A* ⟹ *countable B*
by (*blast elim*: *countableE-bij intro*: *bij-betw-trans bij-betw-inv-into countableI-bij*)

lemma *countable-iff-bij*[*simp*]: *bij-betw f A B* ⟹ *countable A* ⟷ *countable B*
by (*blast intro*: *countableI-bij1 countableI-bij2*)

lemma *countable-subset*: *A* ⊆ *B* ⟹ *countable B* ⟹ *countable A*
by (*auto simp*: *countable-def intro*: *inj-on-subset*)

lemma *countableI-type*[*intro*, *simp*]: *countable* (*A*:: *'a* :: *countable set*)
using *countableI*[*of to-nat A*] **by** *auto*

21.2 Enumerate a countable set

lemma *countableE-infinite*:
assumes *countable S infinite S*
obtains *e* :: *'a* ⇒ *nat* **where** *bij-betw e S UNIV*
proof −
obtain *f* :: *'a* ⇒ *nat* **where** *inj-on f S*
using ‹*countable S*› **by** (*rule countableE*)
then have *bij-betw f S* (*f*‘*S*)
unfolding *bij-betw-def* **by** *simp*
moreover
from ‹*inj-on f S*› ‹*infinite S*› **have** *inf-fS*: *infinite* (*f*‘*S*)
by (*auto dest*: *finite-imageD*)
ultimately show *thesis*
by (*meson bij-betw-inv bij-betw-trans bij-enumerate that*)
qed

lemma *countable-infiniteE'*:
assumes *countable A infinite A*
obtains *g* **where** *bij-betw g* (*UNIV* :: *nat set*) *A*
by (*meson assms bij-betw-inv countableE-infinite*)

lemma *countable-enum-cases*:
assumes *countable S*
obtains (*finite*) *f* :: *'a* ⇒ *nat* **where** *finite S bij-betw f S* {..<*card S*}
| (*infinite*) *f* :: *'a* ⇒ *nat* **where** *infinite S bij-betw f S UNIV*
using *ex-bij-betw-finite-nat*[*of S*] *countableE-infinite* ‹*countable S*›
by (*cases finite S*) (*auto simp add*: *atLeast0LessThan*)

definition *to-nat-on* :: *'a set* ⇒ *'a* ⇒ *nat* **where**
to-nat-on S = (*SOME f*. *if finite S then bij-betw f S* {..< *card S*} *else bij-betw f S UNIV*)

definition *from-nat-into* :: *'a set* ⇒ *nat* ⇒ *'a* **where**
from-nat-into S n = (*if n* ∈ *to-nat-on S* ‘ *S then inv-into S* (*to-nat-on S*) *n else SOME s*. *s*∈*S*)

lemma *to-nat-on-finite*: *finite S* ⟹ *bij-betw* (*to-nat-on S*) *S* {..< *card S*}
using *ex-bij-betw-finite-nat* **unfolding** *to-nat-on-def*
by (*intro someI2-ex*[**where** *Q*=λ*f*. *bij-betw f S* {..<*card S*}]) (*auto simp add*: *atLeast0LessThan*)

lemma *to-nat-on-infinite*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*to-nat-on S*) *S UNIV*
using *countableE-infinite* **unfolding** *to-nat-on-def*
by (*intro someI2-ex*[**where** *Q*=λ*f*. *bij-betw f S UNIV*]) *auto*

lemma *bij-betw-from-nat-into-finite*: *finite S* ⟹ *bij-betw* (*from-nat-into S*) {..< *card S*} *S*
unfolding *from-nat-into-def*[*abs-def*]
using *to-nat-on-finite*[*of S*]
by (*smt* (*verit*, *ccfv-SIG*) *bij-betw-cong bij-betw-imp-surj-on bij-betw-inv-into*)

lemma *bij-betw-from-nat-into*: *countable S* ⟹ *infinite S* ⟹ *bij-betw* (*from-nat-into S*) *UNIV S*
unfolding *from-nat-into-def*[*abs-def*]
using *to-nat-on-infinite*[*of S*, *unfolded bij-betw-def*]
by (*auto cong*: *bij-betw-cong intro*: *bij-betw-inv-into to-nat-on-infinite*)

The sum/product over the enumeration of a finite set equals simply the sum/product over the set

context *comm-monoid-set*
begin

lemma *card-from-nat-into*:

F (λ*i*. *h* (*from-nat-into A i*)) {..<*card A*} = *F h A*
proof (*cases finite A*)
case *True*
have *F* (λ*i*. *h* (*from-nat-into A i*)) {..<*card A*} = *F h* (*from-nat-into A* ' {..<*card A*})
by (*metis True bij-betw-def bij-betw-from-nat-into-finite reindex-cong*)
also have ... = *F h A*
by (*metis True bij-betw-def bij-betw-from-nat-into-finite*)
finally show *?thesis* **.**
qed *auto*

end

lemma *countable-as-injective-image*:
assumes *countable A infinite A*
obtains *f* :: *nat* ⇒ *'a* **where** *A* = *range f inj f*
by (*metis bij-betw-def bij-betw-from-nat-into* [*OF assms*])

lemma *inj-on-to-nat-on*[*intro*]: *countable A* ⟹ *inj-on* (*to-nat-on A*) *A*
using *to-nat-on-infinite*[*of A*] *to-nat-on-finite*[*of A*]
by (*cases finite A*) (*auto simp*: *bij-betw-def*)

lemma *to-nat-on-inj*[*simp*]:
countable A ⟹ *a* ∈ *A* ⟹ *b* ∈ *A* ⟹ *to-nat-on A a* = *to-nat-on A b* ⟷ *a* = *b*
using *inj-on-to-nat-on*[*of A*] **by** (*auto dest*: *inj-onD*)

lemma *from-nat-into-to-nat-on*[*simp*]: *countable A* ⟹ *a* ∈ *A* ⟹ *from-nat-into A* (*to-nat-on A a*) = *a*
by (*auto simp*: *from-nat-into-def intro*!: *inv-into-f-f*)

lemma *subset-range-from-nat-into*: *countable A* ⟹ *A* ⊆ *range* (*from-nat-into A*)
by (*auto intro*: *from-nat-into-to-nat-on*[*symmetric*])

lemma *from-nat-into*: *A* ≠ {} ⟹ *from-nat-into A n* ∈ *A*
unfolding *from-nat-into-def* **by** (*metis equals0I inv-into-into someI-ex*)

lemma *range-from-nat-into-subset*: *A* ≠ {} ⟹ *range* (*from-nat-into A*) ⊆ *A*
using *from-nat-into*[*of A*] **by** *auto*

lemma *range-from-nat-into*[*simp*]: *A* ≠ {} ⟹ *countable A* ⟹ *range* (*from-nat-into A*) = *A*
by (*metis equalityI range-from-nat-into-subset subset-range-from-nat-into*)

lemma *image-to-nat-on*: *countable A* ⟹ *infinite A* ⟹ *to-nat-on A* ' *A* = *UNIV*
using *to-nat-on-infinite*[*of A*] **by** (*simp add*: *bij-betw-def*)

lemma *to-nat-on-surj*: *countable A* ⟹ *infinite A* ⟹ ∃ *a*∈*A*. *to-nat-on A a* = *n*
by (*metis* (*no-types*) *image-iff iso-tuple-UNIV-I image-to-nat-on*)

lemma *to-nat-on-from-nat-into*[*simp*]: $n \in$ *to-nat-on* A ' $A \Longrightarrow$ *to-nat-on* A (*from-nat-into* A n) $= n$
by (*simp add*: *f-inv-into-f from-nat-into-def*)

lemma *to-nat-on-from-nat-into-infinite*[*simp*]:
countable $A \Longrightarrow$ *infinite* $A \Longrightarrow$ *to-nat-on* A (*from-nat-into* A n) $= n$
by (*metis image-iff to-nat-on-surj to-nat-on-from-nat-into*)

lemma *from-nat-into-inj*:
countable $A \Longrightarrow m \in$ *to-nat-on* A ' $A \Longrightarrow n \in$ *to-nat-on* A ' $A \Longrightarrow$
from-nat-into A m = *from-nat-into* A $n \longleftrightarrow m = n$
by (*subst to-nat-on-inj*[*symmetric*, *of* A]) *auto*

lemma *from-nat-into-inj-infinite*[*simp*]:
countable $A \Longrightarrow$ *infinite* $A \Longrightarrow$ *from-nat-into* A m = *from-nat-into* A $n \longleftrightarrow m = n$
using *image-to-nat-on*[*of* A] *from-nat-into-inj*[*of* A m n] **by** *simp*

lemma *eq-from-nat-into-iff*:
countable $A \Longrightarrow x \in A \Longrightarrow i \in$ *to-nat-on* A ' $A \Longrightarrow x =$ *from-nat-into* A $i \longleftrightarrow i =$ *to-nat-on* A x
by *auto*

lemma *from-nat-into-surj*: *countable* $A \Longrightarrow a \in A \Longrightarrow \exists n.$ *from-nat-into* A $n = a$
by (*rule exI*[*of* - *to-nat-on* A a]) *simp*

lemma *from-nat-into-inject*[*simp*]:
$A \neq \{\} \Longrightarrow$ *countable* $A \Longrightarrow B \neq \{\} \Longrightarrow$ *countable* $B \Longrightarrow$ *from-nat-into* A = *from-nat-into* $B \longleftrightarrow A = B$
by (*metis range-from-nat-into*)

lemma *inj-on-from-nat-into*: *inj-on from-nat-into* ($\{A.\ A \neq \{\} \wedge$ *countable* $A\}$)
unfolding *inj-on-def* **by** *auto*

21.3 Closure properties of countability

lemma *countable-SIGMA*[*intro*, *simp*]:
countable $I \Longrightarrow (\bigwedge i.\ i \in I \Longrightarrow$ *countable* $(A\ i)) \Longrightarrow$ *countable* (*SIGMA* $i : I.\ A\ i$)
by (*intro countableI'*[*of* $\lambda(i, a).$ (*to-nat-on* I i, *to-nat-on* $(A\ i)$ a)]) (*auto simp*: *inj-on-def*)

lemma *countable-image*[*intro*, *simp*]:
assumes *countable* A
shows *countable* (f'A)
by (*metis assms countable-iff-bij countable-subset inj-on-imp-bij-betw subsetI subset-image-inj*)

lemma *countable-image-inj-on*: *countable* (*f* ‘ *A*) ⟹ *inj-on f A* ⟹ *countable A*
by (*metis countable-image the-inv-into-onto*)

lemma *countable-image-inj-Int-vimage*:
⟦*inj-on f S*; *countable A*⟧ ⟹ *countable* (*S* ∩ *f* −‘ *A*)
by (*meson countable-image-inj-on countable-subset image-subset-iff-subset-vimage inf-le2 inj-on-Int*)

lemma *countable-image-inj-gen*:
⟦*inj-on f S*; *countable A*⟧ ⟹ *countable* {*x* ∈ *S*. *f x* ∈ *A*}
using *countable-image-inj-Int-vimage*
by (*auto simp*: *vimage-def Collect-conj-eq*)

lemma *countable-image-inj-eq*:
inj-on f S ⟹ *countable*(*f* ‘ *S*) ⟷ *countable S*
using *countable-image-inj-on* **by** *blast*

lemma *countable-image-inj*:
⟦*countable A*; *inj f*⟧ ⟹ *countable* {*x*. *f x* ∈ *A*}
using *countable-image-inj-gen* **by** *fastforce*

lemma *countable-UN*[*intro*, *simp*]:
fixes *I* :: *'i set* **and** *A* :: *'i* => *'a set*
assumes *I*: *countable I*
assumes *A*: ⋀*i*. *i* ∈ *I* ⟹ *countable* (*A i*)
shows *countable* (⋃ *i*∈*I*. *A i*)
proof −
have (⋃ *i*∈*I*. *A i*) = *snd* ‘ (*SIGMA i* : *I*. *A i*) **by** (*auto simp*: *image-iff*)
then show *?thesis* **by** (*simp add*: *assms*)
qed

lemma *countable-Un*[*intro*]: *countable A* ⟹ *countable B* ⟹ *countable* (*A* ∪ *B*)
by (*rule countable-UN*[*of* {*True*, *False*} λ*True* ⇒ *A* | *False* ⇒ *B*, *simplified*])
(*simp split*: *bool.split*)

lemma *countable-Un-iff*[*simp*]: *countable* (*A* ∪ *B*) ⟷ *countable A* ∧ *countable B*
by (*metis countable-Un countable-subset inf-sup-ord*(*3*,*4*))

lemma *countable-Plus*[*intro*, *simp*]:
countable A ⟹ *countable B* ⟹ *countable* (*A* <+> *B*)
by (*simp add*: *Plus-def*)

lemma *countable-empty*[*intro*, *simp*]: *countable* {}
by (*blast intro*: *countable-finite*)

lemma *countable-insert*[*intro*, *simp*]: *countable A* ⟹ *countable* (*insert a A*)
using *countable-Un*[*of* {*a*} *A*] **by** (*auto simp*: *countable-finite*)

lemma *countable-Int1*[*intro*, *simp*]: *countable A* ⟹ *countable* (*A* ∩ *B*)

by (*force intro*: *countable-subset*)

lemma *countable-Int2*[*intro*, *simp*]: *countable B* ⟹ *countable* (*A* ∩ *B*)
by (*blast intro*: *countable-subset*)

lemma *countable-INT*[*intro*, *simp*]: *i* ∈ *I* ⟹ *countable* (*A i*) ⟹ *countable* (⋂ *i*∈*I*. *A i*)
by (*blast intro*: *countable-subset*)

lemma *countable-Diff*[*intro*, *simp*]: *countable A* ⟹ *countable* (*A* − *B*)
by (*blast intro*: *countable-subset*)

lemma *countable-insert-eq* [*simp*]: *countable* (*insert x A*) = *countable A*
by *auto* (*metis Diff-insert-absorb countable-Diff insert-absorb*)

lemma *countable-vimage*: *B* ⊆ *range f* ⟹ *countable* (*f* −‘ *B*) ⟹ *countable B*
by (*metis Int-absorb2 countable-image image-vimage-eq*)

lemma *surj-countable-vimage*: *surj f* ⟹ *countable* (*f* −‘ *B*) ⟹ *countable B*
by (*metis countable-vimage top-greatest*)

lemma *countable-Collect*[*simp*]: *countable A* ⟹ *countable* {*a* ∈ *A*. *φ a*}
by (*metis Collect-conj-eq Int-absorb Int-commute Int-def countable-Int1*)

lemma *countable-Image*:
assumes ⋀*y*. *y* ∈ *Y* ⟹ *countable* (*X* `` {*y*})
assumes *countable Y*
shows *countable* (*X* `` *Y*)
by (*metis Image-eq-UN assms countable-UN*)

lemma *countable-relpow*:
fixes *X* :: *'a rel*
assumes *Image-X*: ⋀*Y*. *countable Y* ⟹ *countable* (*X* `` *Y*)
assumes *Y*: *countable Y*
shows *countable* ((*X* ^^ *i*) `` *Y*)
using *Y* **by** (*induct i arbitrary*: *Y*) (*auto simp*: *relcomp-Image Image-X*)

lemma *countable-funpow*:
fixes *f* :: *'a set* ⇒ *'a set*
assumes ⋀*A*. *countable A* ⟹ *countable* (*f A*)
and *countable A*
shows *countable* ((*f* ^^ *n*) *A*)
by(*induction n*)(*simp-all add*: *assms*)

lemma *countable-rtrancl*:
(⋀*Y*. *countable Y* ⟹ *countable* (*X* `` *Y*)) ⟹ *countable Y* ⟹ *countable* (*X*\* `` *Y*)
unfolding *rtrancl-is-UN-relpow UN-Image* **by** (*intro countable-UN countableI-type countable-relpow*)

lemma *countable-lists*[*intro*, *simp*]:
assumes *A*: *countable A* **shows** *countable* (*lists A*)
by (*meson assms countableI' countable-def inj-on-map-lists*)

lemma *Collect-finite-eq-lists*: *Collect finite* = *set* ' *lists UNIV*
using *finite-list* **by** *auto*

lemma *countable-Collect-finite*: *countable* (*Collect* (*finite*::'*a*::*countable set*⇒*bool*))
by (*simp add*: *Collect-finite-eq-lists*)

lemma *countable-int*: *countable* ℤ
unfolding *Ints-def* **by** *auto*

lemma *countable-rat*: *countable* ℚ
unfolding *Rats-def* **by** *auto*

lemma *Collect-finite-subset-eq-lists*: {*A*. *finite A* ∧ *A* ⊆ *T*} = *set* ' *lists T*
using *finite-list* **by** (*auto simp*: *lists-eq-set*)

lemma *countable-Collect-finite-subset*:
countable T ⟹ *countable* {*A*. *finite A* ∧ *A* ⊆ *T*}
unfolding *Collect-finite-subset-eq-lists* **by** *auto*

lemma *countable-Fpow*: *countable S* ⟹ *countable* (*Fpow S*)
using *countable-Collect-finite-subset*
by (*force simp add*: *Fpow-def conj-commute*)

lemma *countable-set-option* [*simp*]: *countable* (*set-option x*)
by (*cases x*) *auto*

21.4 Misc lemmas

lemma *countable-subset-image*:
countable B ∧ *B* ⊆ (*f* ' *A*) ⟷ (∃ *A'*. *countable A'* ∧ *A'* ⊆ *A* ∧ (*B* = *f* ' *A'*))
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
show *?rhs*
by (*rule exI* [**where** *x*=*inv-into A f* ' *B*])
(*use* ‹*?lhs*› **in** ‹*auto simp*: *f-inv-into-f subset-iff image-inv-into-cancel inv-into-into*›)
next
assume *?rhs*
then show *?lhs* **by** *force*
qed

lemma *ex-subset-image-inj*:
(∃ *T*. *T* ⊆ *f* ' *S* ∧ *P T*) ⟷ (∃ *T*. *T* ⊆ *S* ∧ *inj-on f T* ∧ *P* (*f* ' *T*))
by (*auto simp*: *subset-image-inj*)

lemma *all-subset-image-inj*:
(∀ *T*. *T* ⊆ *f* ‘ *S* ⟶ *P T*) ⟷ (∀ *T*. *T* ⊆ *S* ∧ *inj-on f T* ⟶ *P*(*f* ‘ *T*))
by (*metis subset-image-inj*)

lemma *ex-countable-subset-image-inj*:
(∃ *T*. *countable T* ∧ *T* ⊆ *f* ‘ *S* ∧ *P T*) ⟷
(∃ *T*. *countable T* ∧ *T* ⊆ *S* ∧ *inj-on f T* ∧ *P* (*f* ‘ *T*))
by (*metis countable-image-inj-eq subset-image-inj*)

lemma *all-countable-subset-image-inj*:
(∀ *T*. *countable T* ∧ *T* ⊆ *f* ‘ *S* ⟶ *P T*) ⟷ (∀ *T*. *countable T* ∧ *T* ⊆ *S* ∧ *inj-on f T* ⟶*P*(*f* ‘ *T*))
by (*metis countable-image-inj-eq subset-image-inj*)

lemma *ex-countable-subset-image*:
(∃ *T*. *countable T* ∧ *T* ⊆ *f* ‘ *S* ∧ *P T*) ⟷ (∃ *T*. *countable T* ∧ *T* ⊆ *S* ∧ *P* (*f* ‘ *T*))
by (*metis countable-subset-image*)

lemma *all-countable-subset-image*:
(∀ *T*. *countable T* ∧ *T* ⊆ *f* ‘ *S* ⟶ *P T*) ⟷ (∀ *T*. *countable T* ∧ *T* ⊆ *S* ⟶ *P*(*f* ‘ *T*))
by (*metis countable-subset-image*)

lemma *countable-image-eq*:
countable(*f* ‘ *S*) ⟷ (∃ *T*. *countable T* ∧ *T* ⊆ *S* ∧ *f* ‘ *S* = *f* ‘ *T*)
by (*metis countable-image countable-image-inj-eq order-refl subset-image-inj*)

lemma *countable-image-eq-inj*:
countable(*f* ‘ *S*) ⟷ (∃ *T*. *countable T* ∧ *T* ⊆ *S* ∧ *f* ‘ *S* = *f* ‘ *T* ∧ *inj-on f T*)
by (*metis countable-image-inj-eq order-refl subset-image-inj*)

lemma *infinite-countable-subset'*:
assumes *X*: *infinite X* **shows** ∃ *C*⊆*X*. *countable C* ∧ *infinite C*
proof −
obtain *f* :: *nat* ⇒ *'a* **where** *inj f range f* ⊆ *X*
using *infinite-countable-subset* [*OF X*] **by** *blast*
then show *?thesis*
by (*intro exI*[*of - range f*]) (*auto simp*: *range-inj-infinite*)
qed

lemma *countable-all*:
assumes *S*: *countable S*
shows (∀ *s*∈*S*. *P s*) ⟷ (∀ *n*::*nat*. *from-nat-into S n* ∈ *S* ⟶ *P* (*from-nat-into S n*))
using *S*[*THEN subset-range-from-nat-into*] **by** *auto*

lemma *finite-sequence-to-countable-set*:

assumes *countable X*
obtains *F* **where** $\bigwedge i.\ F\ i \subseteq X\ \bigwedge i.\ F\ i \subseteq F\ (Suc\ i)\ \bigwedge i.\ finite\ (F\ i)\ (\bigcup i.\ F\ i) = X$
proof −
show *thesis*
apply (*rule that*[*of* $\lambda i.\ if\ X = \{\}\ then\ \{\}\ else\ from\text{-}nat\text{-}into\ X\ `\ \{..i\}$])
apply (*auto simp add*: *image-iff intro*: *from-nat-into split*: *if-splits*)
using *assms from-nat-into-surj* **by** (*fastforce cong*: *image-cong*)
qed

lemma *transfer-countable*[*transfer-rule*]:
bi-unique R $\Longrightarrow$ *rel-fun* (*rel-set R*) (=) *countable countable*
by (*rule rel-funI*, *erule* (*1*) *bi-unique-rel-set-lemma*)
(*auto dest*: *countable-image-inj-on*)

21.5 Uncountable

abbreviation *uncountable* **where**
uncountable A $\equiv \neg$ *countable A*

lemma *uncountable-def*: *uncountable A* $\longleftrightarrow$ $A \neq \{\} \wedge \neg\ (\exists f{::}(nat \Rightarrow 'a).\ range\ f = A)$
by (*auto intro*: *inj-on-inv-into simp*: *countable-def*)
(*metis all-not-in-conv inj-on-iff-surj subset-UNIV*)

lemma *uncountable-bij-betw*: *bij-betw f A B* $\Longrightarrow$ *uncountable B* $\Longrightarrow$ *uncountable A*
unfolding *bij-betw-def* **by** (*metis countable-image*)

lemma *uncountable-infinite*: *uncountable A* $\Longrightarrow$ *infinite A*
by (*metis countable-finite*)

lemma *uncountable-minus-countable*:
uncountable A $\Longrightarrow$ *countable B* $\Longrightarrow$ *uncountable* $(A - B)$
using *countable-Un*[*of B A − B*] **by** *auto*

lemma *countable-Diff-eq* [*simp*]: *countable* $(A - \{x\})$ = *countable A*
by (*meson countable-Diff countable-empty countable-insert uncountable-minus-countable*)

Every infinite set can be covered by a pairwise disjoint family of infinite sets. This version doesn't achieve equality, as it only covers a countable subset

lemma *infinite-infinite-partition*:
assumes *infinite A*
obtains *C* :: $nat \Rightarrow 'a\ set$
where *pairwise* $(\lambda i\ j.\ disjnt\ (C\ i)\ (C\ j))\ UNIV\ (\bigcup i.\ C\ i) \subseteq A\ \bigwedge i.\ infinite\ (C\ i)$
proof −
obtain *f* :: $nat \Rightarrow 'a$ **where** *range f* $\subseteq$ *A inj f*
using *assms infinite-countable-subset* **by** *blast*

let *?C* = λi. *range* (λj. *f* (*prod-encode* (*i*,*j*)))
show *thesis*
proof
show *pairwise* ($\lambda i\ j$. *disjnt* (*?C i*) (*?C j*)) *UNIV*
by (*auto simp*: *pairwise-def disjnt-def inj-on-eq-iff* [*OF* ‹*inj f*›] *inj-on-eq-iff* [*OF inj-prod-encode*, *of* - *UNIV*])
show ($\bigcup i$. *?C i*) ⊆ *A*
using ‹*range f* ⊆ *A*› **by** *blast*
have *infinite* (*range* (λj. *f* (*prod-encode* (*i*, *j*)))) **for** *i*
by (*rule range-inj-infinite*) (*meson Pair-inject* ‹*inj f*› *inj-def prod-encode-eq*)
then show $\bigwedge i$. *infinite* (*?C i*)
using *that* **by** *auto*
qed
qed

end

22 Countable Complete Lattices

theory *Countable-Complete-Lattices*
imports *Main Countable-Set*
begin

lemma *UNIV-nat-eq*: *UNIV* = *insert 0* (*range Suc*)
by (*metis UNIV-eq-I nat.nchotomy insertCI rangeI*)

class *countable-complete-lattice* = *lattice* + *Inf* + *Sup* + *bot* + *top* +
assumes *ccInf-lower*: *countable A* ⟹ *x* ∈ *A* ⟹ *Inf A* ≤ *x*
assumes *ccInf-greatest*: *countable A* ⟹ ($\bigwedge x$. *x* ∈ *A* ⟹ *z* ≤ *x*) ⟹ *z* ≤ *Inf A*
assumes *ccSup-upper*: *countable A* ⟹ *x* ∈ *A* ⟹ *x* ≤ *Sup A*
assumes *ccSup-least*: *countable A* ⟹ ($\bigwedge x$. *x* ∈ *A* ⟹ *x* ≤ *z*) ⟹ *Sup A* ≤ *z*
assumes *ccInf-empty* [*simp*]: *Inf* {} = *top*
assumes *ccSup-empty* [*simp*]: *Sup* {} = *bot*
begin

subclass *bounded-lattice*
proof
fix *a*
show *bot* ≤ *a* **by** (*auto intro*: *ccSup-least simp only*: *ccSup-empty* [*symmetric*])
show *a* ≤ *top* **by** (*auto intro*: *ccInf-greatest simp only*: *ccInf-empty* [*symmetric*])
qed

lemma *ccINF-lower*: *countable A* ⟹ *i* ∈ *A* ⟹ (*INF i* ∈ *A*. *f i*) ≤ *f i*
using *ccInf-lower* [*of f* ‘ *A*] **by** *simp*

lemma *ccINF-greatest*: *countable A* ⟹ ($\bigwedge i$. *i* ∈ *A* ⟹ *u* ≤ *f i*) ⟹ *u* ≤ (*INF i* ∈ *A*. *f i*)
using *ccInf-greatest* [*of f* ‘ *A*] **by** *auto*

lemma *ccSUP-upper*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow f\ i \leq (SUP\ i \in A.\ f\ i)$
using *ccSup-upper* [*of f ' A*] **by** *simp*

lemma *ccSUP-least*: *countable A* $\Longrightarrow$ $(\bigwedge i.\ i \in A \Longrightarrow f\ i \leq u) \Longrightarrow (SUP\ i \in A.\ f\ i) \leq u$
using *ccSup-least* [*of f ' A*] **by** *auto*

lemma *ccInf-lower2*: *countable A* $\Longrightarrow$ $u \in A \Longrightarrow u \leq v \Longrightarrow Inf\ A \leq v$
using *ccInf-lower* [*of A u*] **by** *auto*

lemma *ccINF-lower2*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow f\ i \leq u \Longrightarrow (INF\ i \in A.\ f\ i) \leq u$
using *ccINF-lower* [*of A i f*] **by** *auto*

lemma *ccSup-upper2*: *countable A* $\Longrightarrow$ $u \in A \Longrightarrow v \leq u \Longrightarrow v \leq Sup\ A$
using *ccSup-upper* [*of A u*] **by** *auto*

lemma *ccSUP-upper2*: *countable A* $\Longrightarrow$ $i \in A \Longrightarrow u \leq f\ i \Longrightarrow u \leq (SUP\ i \in A.\ f\ i)$
using *ccSUP-upper* [*of A i f*] **by** *auto*

lemma *le-ccInf-iff*: *countable A* $\Longrightarrow$ $b \leq Inf\ A \longleftrightarrow (\forall\, a{\in}A.\ b \leq a)$
by (*auto intro*: *ccInf-greatest dest*: *ccInf-lower*)

lemma *le-ccINF-iff*: *countable A* $\Longrightarrow$ $u \leq (INF\ i \in A.\ f\ i) \longleftrightarrow (\forall\, i{\in}A.\ u \leq f\ i)$
using *le-ccInf-iff* [*of f ' A*] **by** *simp*

lemma *ccSup-le-iff*: *countable A* $\Longrightarrow$ $Sup\ A \leq b \longleftrightarrow (\forall\, a{\in}A.\ a \leq b)$
by (*auto intro*: *ccSup-least dest*: *ccSup-upper*)

lemma *ccSUP-le-iff*: *countable A* $\Longrightarrow$ $(SUP\ i \in A.\ f\ i) \leq u \longleftrightarrow (\forall\, i{\in}A.\ f\ i \leq u)$
using *ccSup-le-iff* [*of f ' A*] **by** *simp*

lemma *ccInf-insert* [*simp*]: *countable A* $\Longrightarrow$ $Inf\ (insert\ a\ A) = inf\ a\ (Inf\ A)$
by (*force intro*: *le-infI le-infI1 le-infI2 order.antisym ccInf-greatest ccInf-lower*)

lemma *ccINF-insert* [*simp*]: *countable A* $\Longrightarrow$ $(INF\ x{\in}insert\ a\ A.\ f\ x) = inf\ (f\ a)\ (Inf\ (f\ `\ A))$
unfolding *image-insert* **by** *simp*

lemma *ccSup-insert* [*simp*]: *countable A* $\Longrightarrow$ $Sup\ (insert\ a\ A) = sup\ a\ (Sup\ A)$
by (*force intro*: *le-supI le-supI1 le-supI2 order.antisym ccSup-least ccSup-upper*)

lemma *ccSUP-insert* [*simp*]: *countable A* $\Longrightarrow$ $(SUP\ x{\in}insert\ a\ A.\ f\ x) = sup\ (f\ a)\ (Sup\ (f\ `\ A))$
unfolding *image-insert* **by** *simp*

lemma *ccINF-empty* [*simp*]: $(INF\ x{\in}\{\}.\ f\ x) = top$
unfolding *image-empty* **by** *simp*

lemma *ccSUP-empty* [*simp*]: $(SUP\ x{\in}\{\}.\ f\ x) = bot$
unfolding *image-empty* **by** *simp*

lemma *ccInf-superset-mono*: *countable* $A \Longrightarrow B \subseteq A \Longrightarrow Inf\ A \leq Inf\ B$
by (*auto intro*: *ccInf-greatest ccInf-lower countable-subset*)

lemma *ccSup-subset-mono*: *countable* $B \Longrightarrow A \subseteq B \Longrightarrow Sup\ A \leq Sup\ B$
by (*auto intro*: *ccSup-least ccSup-upper countable-subset*)

lemma *ccInf-mono*:
assumes [*intro*]: *countable* B *countable* A
assumes $\bigwedge b.\ b \in B \Longrightarrow \exists\, a{\in}A.\ a \leq b$
shows $Inf\ A \leq Inf\ B$
proof (*rule ccInf-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** $Inf\ A \leq a$ **by** (*rule ccInf-lower*[*rotated*]) *auto*
with ‹$a \leq b$› **show** $Inf\ A \leq b$ **by** *auto*
qed *auto*

lemma *ccINF-mono*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (\bigwedge m.\ m \in B \Longrightarrow \exists\, n{\in}A.\ f\ n \leq g\ m) \Longrightarrow (INF\ n{\in}A.\ f\ n) \leq (INF\ n{\in}B.\ g\ n)$
using *ccInf-mono* [*of g ' B f ' A*] **by** *auto*

lemma *ccSup-mono*:
assumes [*intro*]: *countable* B *countable* A
assumes $\bigwedge a.\ a \in A \Longrightarrow \exists\, b{\in}B.\ a \leq b$
shows $Sup\ A \leq Sup\ B$
proof (*rule ccSup-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** $b \leq Sup\ B$ **by** (*rule ccSup-upper*[*rotated*]) *auto*
with ‹$a \leq b$› **show** $a \leq Sup\ B$ **by** *auto*
qed *auto*

lemma *ccSUP-mono*:
countable $A \Longrightarrow$ *countable* $B \Longrightarrow (\bigwedge n.\ n \in A \Longrightarrow \exists\, m{\in}B.\ f\ n \leq g\ m) \Longrightarrow (SUP\ n{\in}A.\ f\ n) \leq (SUP\ n{\in}B.\ g\ n)$
using *ccSup-mono* [*of g ' B f ' A*] **by** *auto*

lemma *ccINF-superset-mono*:
countable $A \Longrightarrow B \subseteq A \Longrightarrow (\bigwedge x.\ x \in B \Longrightarrow f\ x \leq g\ x) \Longrightarrow (INF\ x{\in}A.\ f\ x) \leq (INF\ x{\in}B.\ g\ x)$
by (*blast intro*: *ccINF-mono countable-subset dest*: *subsetD*)

lemma *ccSUP-subset-mono*:
countable $B \Longrightarrow A \subseteq B \Longrightarrow (\bigwedge x.\ x \in A \Longrightarrow f\ x \leq g\ x) \Longrightarrow (SUP\ x{\in}A.\ f\ x) \leq$

(*SUP x∈B. g x*)
by (*blast intro*: *ccSUP-mono countable-subset dest*: *subsetD*)

lemma *less-eq-ccInf-inter*: *countable A* ⟹ *countable B* ⟹ *sup* (*Inf A*) (*Inf B*) ≤ *Inf* (*A* ∩ *B*)
by (*auto intro*: *ccInf-greatest ccInf-lower*)

lemma *ccSup-inter-less-eq*: *countable A* ⟹ *countable B* ⟹ *Sup* (*A* ∩ *B*) ≤ *inf* (*Sup A*) (*Sup B*)
by (*auto intro*: *ccSup-least ccSup-upper*)

lemma *ccInf-union-distrib*: *countable A* ⟹ *countable B* ⟹ *Inf* (*A* ∪ *B*) = *inf* (*Inf A*) (*Inf B*)
by (*rule order.antisym*) (*auto intro*: *ccInf-greatest ccInf-lower le-infI1 le-infI2*)

lemma *ccINF-union*:
countable A ⟹ *countable B* ⟹ (*INF i∈A* ∪ *B. M i*) = *inf* (*INF i∈A. M i*) (*INF i∈B. M i*)
by (*auto intro*!: *order.antisym ccINF-mono intro*: *le-infI1 le-infI2 ccINF-greatest ccINF-lower*)

lemma *ccSup-union-distrib*: *countable A* ⟹ *countable B* ⟹ *Sup* (*A* ∪ *B*) = *sup* (*Sup A*) (*Sup B*)
by (*rule order.antisym*) (*auto intro*: *ccSup-least ccSup-upper le-supI1 le-supI2*)

lemma *ccSUP-union*:
countable A ⟹ *countable B* ⟹ (*SUP i∈A* ∪ *B. M i*) = *sup* (*SUP i∈A. M i*) (*SUP i∈B. M i*)
by (*auto intro*!: *order.antisym ccSUP-mono intro*: *le-supI1 le-supI2 ccSUP-least ccSUP-upper*)

lemma *ccINF-inf-distrib*: *countable A* ⟹ *inf* (*INF a∈A. f a*) (*INF a∈A. g a*) = (*INF a∈A. inf* (*f a*) (*g a*))
by (*rule order.antisym*) (*rule ccINF-greatest*, *auto intro*: *le-infI1 le-infI2 ccINF-lower ccINF-mono*)

lemma *ccSUP-sup-distrib*: *countable A* ⟹ *sup* (*SUP a∈A. f a*) (*SUP a∈A. g a*) = (*SUP a∈A. sup* (*f a*) (*g a*))
by (*rule order.antisym*[*rotated*]) (*rule ccSUP-least*, *auto intro*: *le-supI1 le-supI2 ccSUP-upper ccSUP-mono*)

lemma *ccINF-const* [*simp*]: *A* ≠ {} ⟹ (*INF i* ∈ *A. f*) = *f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccSUP-const* [*simp*]: *A* ≠ {} ⟹ (*SUP i* ∈ *A. f*) = *f*
unfolding *image-constant-conv* **by** *auto*

lemma *ccINF-top* [*simp*]: (*INF x∈A. top*) = *top*

by (*cases* $A = \{\}$) *simp-all*

lemma *ccSUP-bot* [*simp*]: $(SUP\ x{\in}A.\ bot) = bot$
by (*cases* $A = \{\}$) *simp-all*

lemma *ccINF-commute*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow (INF\ i{\in}A.\ INF\ j{\in}B.\ f\ i\ j) = (INF\ j{\in}B.\ INF\ i{\in}A.\ f\ i\ j)$
by (*iprover intro*: *ccINF-lower ccINF-greatest order-trans order.antisym*)

lemma *ccSUP-commute*: *countable* $A \Longrightarrow$ *countable* $B \Longrightarrow (SUP\ i{\in}A.\ SUP\ j{\in}B.\ f\ i\ j) = (SUP\ j{\in}B.\ SUP\ i{\in}A.\ f\ i\ j)$
by (*iprover intro*: *ccSUP-upper ccSUP-least order-trans order.antisym*)

end

context
fixes $a :: 'a{::}\{countable\text{-}complete\text{-}lattice, linorder\}$
begin

lemma *less-ccSup-iff*: *countable* $S \Longrightarrow a < Sup\ S \longleftrightarrow (\exists\, x{\in}S.\ a < x)$
unfolding *not-le* [*symmetric*] **by** (*subst ccSup-le-iff*) *auto*

lemma *less-ccSUP-iff*: *countable* $A \Longrightarrow a < (SUP\ i{\in}A.\ f\ i) \longleftrightarrow (\exists\, x{\in}A.\ a < f\ x)$
using *less-ccSup-iff* [*of f ' A*] **by** *simp*

lemma *ccInf-less-iff*: *countable* $S \Longrightarrow Inf\ S < a \longleftrightarrow (\exists\, x{\in}S.\ x < a)$
unfolding *not-le* [*symmetric*] **by** (*subst le-ccInf-iff*) *auto*

lemma *ccINF-less-iff*: *countable* $A \Longrightarrow (INF\ i{\in}A.\ f\ i) < a \longleftrightarrow (\exists\, x{\in}A.\ f\ x < a)$
using *ccInf-less-iff* [*of f ' A*] **by** *simp*

end

class *countable-complete-distrib-lattice* = *countable-complete-lattice* +
assumes *sup-ccInf*: *countable* $B \Longrightarrow sup\ a\ (Inf\ B) = (INF\ b{\in}B.\ sup\ a\ b)$
assumes *inf-ccSup*: *countable* $B \Longrightarrow inf\ a\ (Sup\ B) = (SUP\ b{\in}B.\ inf\ a\ b)$
begin

lemma *sup-ccINF*:
countable $B \Longrightarrow sup\ a\ (INF\ b{\in}B.\ f\ b) = (INF\ b{\in}B.\ sup\ a\ (f\ b))$
by (*simp only*: *sup-ccInf image-image countable-image*)

lemma *inf-ccSUP*:
countable $B \Longrightarrow inf\ a\ (SUP\ b{\in}B.\ f\ b) = (SUP\ b{\in}B.\ inf\ a\ (f\ b))$
by (*simp only*: *inf-ccSup image-image countable-image*)

subclass *distrib-lattice*
proof
fix *a b c*

from *sup-ccInf*[*of* {*b*, *c*} *a*] **have** *sup a* (*Inf* {*b*, *c*}) = (*INF d*∈{*b*, *c*}. *sup a d*)
by *simp*
then show *sup a* (*inf b c*) = *inf* (*sup a b*) (*sup a c*)
by *simp*
qed

lemma *ccInf-sup*:
countable B ⟹ *sup* (*Inf B*) *a* = (*INF b*∈*B*. *sup b a*)
by (*simp add*: *sup-ccInf sup-commute*)

lemma *ccSup-inf*:
countable B ⟹ *inf* (*Sup B*) *a* = (*SUP b*∈*B*. *inf b a*)
by (*simp add*: *inf-ccSup inf-commute*)

lemma *ccINF-sup*:
countable B ⟹ *sup* (*INF b*∈*B*. *f b*) *a* = (*INF b*∈*B*. *sup* (*f b*) *a*)
by (*simp add*: *sup-ccINF sup-commute*)

lemma *ccSUP-inf*:
countable B ⟹ *inf* (*SUP b*∈*B*. *f b*) *a* = (*SUP b*∈*B*. *inf* (*f b*) *a*)
by (*simp add*: *inf-ccSUP inf-commute*)

lemma *ccINF-sup-distrib2*:
countable A ⟹ *countable B* ⟹ *sup* (*INF a*∈*A*. *f a*) (*INF b*∈*B*. *g b*) = (*INF a*∈*A*. *INF b*∈*B*. *sup* (*f a*) (*g b*))
by (*subst ccINF-commute*) (*simp-all add*: *sup-ccINF ccINF-sup*)

lemma *ccSUP-inf-distrib2*:
countable A ⟹ *countable B* ⟹ *inf* (*SUP a*∈*A*. *f a*) (*SUP b*∈*B*. *g b*) = (*SUP a*∈*A*. *SUP b*∈*B*. *inf* (*f a*) (*g b*))
by (*subst ccSUP-commute*) (*simp-all add*: *inf-ccSUP ccSUP-inf*)

context
fixes *f* :: *'a* ⇒ *'b*::*countable-complete-lattice*
assumes *mono f*
begin

lemma *mono-ccInf*:
countable A ⟹ *f* (*Inf A*) ≤ (*INF x*∈*A*. *f x*)
using ‹*mono f*›
by (*auto intro*!: *countable-complete-lattice-class.ccINF-greatest intro*: *ccInf-lower dest*: *monoD*)

lemma *mono-ccSup*:
countable A ⟹ (*SUP x*∈*A*. *f x*) ≤ *f* (*Sup A*)
using ‹*mono f*› **by** (*auto intro*: *countable-complete-lattice-class.ccSUP-least ccSup-upper dest*: *monoD*)

lemma *mono-ccINF*:

countable I $\Longrightarrow$ *f* (*INF i* $\in$ *I*. *A i*) $\leq$ (*INF x* $\in$ *I*. *f* (*A x*))
by (*intro countable-complete-lattice-class.ccINF-greatest monoD*[*OF* ‹*mono f*›] *ccINF-lower*)

lemma *mono-ccSUP*:
countable I $\Longrightarrow$ (*SUP x* $\in$ *I*. *f* (*A x*)) $\leq$ *f* (*SUP i* $\in$ *I*. *A i*)
by (*intro countable-complete-lattice-class.ccSUP-least monoD*[*OF* ‹*mono f*›] *ccSUP-upper*)

end

end

22.0.1 Instances of countable complete lattices

instance *fun* :: (*type*, *countable-complete-lattice*) *countable-complete-lattice*
by *standard*
(*auto simp*: *le-fun-def intro*!: *ccSUP-upper ccSUP-least ccINF-lower ccINF-greatest*)

subclass (**in** *complete-lattice*) *countable-complete-lattice*
by *standard* (*auto intro*: *Sup-upper Sup-least Inf-lower Inf-greatest*)

subclass (**in** *complete-distrib-lattice*) *countable-complete-distrib-lattice*
by *standard* (*auto intro*: *sup-Inf inf-Sup*)

end

23 Type of (at Most) Countable Sets

theory *Countable-Set-Type*
imports *Countable-Set*
begin

23.1 Cardinal stuff

context
includes *cardinal-syntax*
begin

lemma *countable-card-of-nat*: *countable A* $\longleftrightarrow$ |*A*| $\leq o$ |*UNIV*::*nat set*|
unfolding *countable-def card-of-ordLeq*[*symmetric*] **by** *auto*

lemma *countable-card-le-natLeq*: *countable A* $\longleftrightarrow$ |*A*| $\leq o$ *natLeq*
unfolding *countable-card-of-nat* **using** *card-of-nat ordLeq-ordIso-trans ordIso-symmetric*
by *blast*

lemma *countable-or-card-of*:
assumes *countable A*
shows (*finite A* $\wedge$ |*A*| $<o$ |*UNIV*::*nat set*|) $\vee$

(*infinite A* $\wedge$ $|A| =o |UNIV::nat\ set|$)
by (*metis assms countable-card-of-nat infinite-iff-card-of-nat ordIso-iff-ordLeq ordLeq-iff-ordLess-or-ordIso*)

lemma *countable-cases-card-of*[*elim*]:
assumes *countable A*
obtains (*Fin*) *finite A* $|A| <o |UNIV::nat\ set|$
| (*Inf*) *infinite A* $|A| =o |UNIV::nat\ set|$
using *assms countable-or-card-of* **by** *blast*

lemma *countable-or*:
countable A $\Longrightarrow$ ($\exists$ *f*::'*a*$\Rightarrow$*nat. finite A* $\wedge$ *inj-on f A*) $\vee$ ($\exists$ *f*::'*a*$\Rightarrow$*nat. infinite A* $\wedge$ *bij-betw f A UNIV*)
by (*elim countable-enum-cases*) *fastforce*+

lemma *countable-cases*[*elim*]:
assumes *countable A*
obtains (*Fin*) *f* :: '*a*$\Rightarrow$*nat* **where** *finite A inj-on f A*
| (*Inf*) *f* :: '*a*$\Rightarrow$*nat* **where** *infinite A bij-betw f A UNIV*
using *assms countable-or* **by** *metis*

lemma *countable-ordLeq*:
assumes $|A| \leq o |B|$ **and** *countable B*
shows *countable A*
using *assms* **unfolding** *countable-card-of-nat* **by**(*rule ordLeq-transitive*)

lemma *countable-ordLess*:
assumes *AB*: $|A| <o |B|$ **and** *B*: *countable B*
shows *countable A*
using *countable-ordLeq*[*OF ordLess-imp-ordLeq*[*OF AB*] *B*] **.**

end

23.2 The type of countable sets

typedef '*a cset* = {*A* :: '*a set. countable A*} **morphisms** *rcset acset*
by (*rule exI*[*of* - {}]) *simp*

setup-lifting *type-definition-cset*

declare
rcset-inverse[*simp*]
acset-inverse[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
acset-inject[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]
rcset[*Transfer.transferred*, *unfolded mem-Collect-eq*, *simp*]

instantiation *cset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

lift-definition *bot-cset* :: *'a cset* **is** {} **parametric** *empty-transfer* **by** *simp*

lift-definition *less-eq-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*
is *subset-eq* **parametric** *subset-transfer* **.**

definition *less-cset* :: *'a cset* ⇒ *'a cset* ⇒ *bool*
where *xs* < *ys* ≡ *xs* ≤ *ys* ∧ *xs* ≠ (*ys*::*'a cset*)

lemma *less-cset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-cset A*) ===> (*pcr-cset A*) ===> (=)) (⊂) (<)
unfolding *less-cset-def*[*abs-def*] *psubset-eq*[*abs-def*] **by** *transfer-prover*

lift-definition *sup-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *union* **parametric** *union-transfer* **by** *simp*

lift-definition *inf-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *inter* **parametric** *inter-transfer* **by** *simp*

lift-definition *minus-cset* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset*
is *minus* **parametric** *Diff-transfer* **by** *simp*

instance by *standard* (*transfer*; *auto*)+

end

abbreviation *cempty* :: *'a cset* **where** *cempty* ≡ *bot*
abbreviation *csubset-eq* :: *'a cset* ⇒ *'a cset* ⇒ *bool* **where** *csubset-eq xs ys* ≡ *xs* ≤ *ys*
abbreviation *csubset* :: *'a cset* ⇒ *'a cset* ⇒ *bool* **where** *csubset xs ys* ≡ *xs* < *ys*
abbreviation *cUn* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset* **where** *cUn xs ys* ≡ *sup xs ys*
abbreviation *cInt* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset* **where** *cInt xs ys* ≡ *inf xs ys*
abbreviation *cDiff* :: *'a cset* ⇒ *'a cset* ⇒ *'a cset* **where** *cDiff xs ys* ≡ *minus xs ys*

lift-definition *cin* :: *'a* ⇒ *'a cset* ⇒ *bool* **is** (∈) **parametric** *member-transfer*
.
lift-definition *cinsert* :: *'a* ⇒ *'a cset* ⇒ *'a cset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
by (*rule countable-insert*)
abbreviation *csingle* :: *'a* ⇒ *'a cset* **where** *csingle x* ≡ *cinsert x cempty*
lift-definition *cimage* :: (*'a* ⇒ *'b*) ⇒ *'a cset* ⇒ *'b cset* **is** (‘) **parametric** *image-transfer*
by (*rule countable-image*)
lift-definition *cBall* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Ball* **parametric** *Ball-transfer*
.
lift-definition *cBex* :: *'a cset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **is** *Bex* **parametric** *Bex-transfer*
.
lift-definition *cUnion* :: *'a cset cset* ⇒ *'a cset* **is** *Union* **parametric** *Union-transfer*

using *countable-UN* [*of* - *id*] **by** *auto*
abbreviation (*input*) *cUNION* :: *'a cset* ⇒ (*'a* ⇒ *'b cset*) ⇒ *'b cset*
where *cUNION A f* ≡ *cUnion* (*cimage f A*)

lemma *Union-conv-UNION*: ⋃ *A* = ⋃ (*id* ‘ *A*)
by *simp*

lemmas *cset-eqI* = *set-eqI*[*Transfer.transferred*]
lemmas *cset-eq-iff*[*no-atp*] = *set-eq-iff*[*Transfer.transferred*]
lemmas *cBallI*[*intro*!] = *ballI*[*Transfer.transferred*]
lemmas *cbspec*[*dest?*] = *bspec*[*Transfer.transferred*]
lemmas *cBallE*[*elim*] = *ballE*[*Transfer.transferred*]
lemmas *cBexI*[*intro*] = *bexI*[*Transfer.transferred*]
lemmas *rev-cBexI*[*intro?*] = *rev-bexI*[*Transfer.transferred*]
lemmas *cBexCI* = *bexCI*[*Transfer.transferred*]
lemmas *cBexE*[*elim*!] = *bexE*[*Transfer.transferred*]
lemmas *cBall-triv*[*simp*] = *ball-triv*[*Transfer.transferred*]
lemmas *cBex-triv*[*simp*] = *bex-triv*[*Transfer.transferred*]
lemmas *cBex-triv-one-point1*[*simp*] = *bex-triv-one-point1*[*Transfer.transferred*]
lemmas *cBex-triv-one-point2*[*simp*] = *bex-triv-one-point2*[*Transfer.transferred*]
lemmas *cBex-one-point1*[*simp*] = *bex-one-point1*[*Transfer.transferred*]
lemmas *cBex-one-point2*[*simp*] = *bex-one-point2*[*Transfer.transferred*]
lemmas *cBall-one-point1*[*simp*] = *ball-one-point1*[*Transfer.transferred*]
lemmas *cBall-one-point2*[*simp*] = *ball-one-point2*[*Transfer.transferred*]
lemmas *cBall-conj-distrib* = *ball-conj-distrib*[*Transfer.transferred*]
lemmas *cBex-disj-distrib* = *bex-disj-distrib*[*Transfer.transferred*]
lemmas *cBall-cong* = *ball-cong*[*Transfer.transferred*]
lemmas *cBex-cong* = *bex-cong*[*Transfer.transferred*]
lemmas *csubsetI*[*intro*!] = *subsetI*[*Transfer.transferred*]
lemmas *csubsetD*[*elim*, *intro?*] = *subsetD*[*Transfer.transferred*]
lemmas *rev-csubsetD*[*no-atp*,*intro?*] = *rev-subsetD*[*Transfer.transferred*]
lemmas *csubsetCE*[*no-atp*,*elim*] = *subsetCE*[*Transfer.transferred*]
lemmas *csubset-eq*[*no-atp*] = *subset-eq*[*Transfer.transferred*]
lemmas *contra-csubsetD*[*no-atp*] = *contra-subsetD*[*Transfer.transferred*]
lemmas *csubset-refl* = *subset-refl*[*Transfer.transferred*]
lemmas *csubset-trans* = *subset-trans*[*Transfer.transferred*]
lemmas *cset-rev-mp* = *rev-subsetD*[*Transfer.transferred*]
lemmas *cset-mp* = *subsetD*[*Transfer.transferred*]
lemmas *csubset-not-fsubset-eq*[*code*] = *subset-not-subset-eq*[*Transfer.transferred*]
lemmas *eq-cmem-trans* = *eq-mem-trans*[*Transfer.transferred*]
lemmas *csubset-antisym*[*intro*!] = *subset-antisym*[*Transfer.transferred*]
lemmas *cequalityD1* = *equalityD1*[*Transfer.transferred*]
lemmas *cequalityD2* = *equalityD2*[*Transfer.transferred*]
lemmas *cequalityE* = *equalityE*[*Transfer.transferred*]
lemmas *cequalityCE*[*elim*] = *equalityCE*[*Transfer.transferred*]
lemmas *eqcset-imp-iff* = *eqset-imp-iff*[*Transfer.transferred*]
lemmas *eqcelem-imp-iff* = *eqelem-imp-iff*[*Transfer.transferred*]
lemmas *cempty-iff*[*simp*] = *empty-iff*[*Transfer.transferred*]
lemmas *cempty-fsubsetI*[*iff*] = *empty-subsetI*[*Transfer.transferred*]

lemmas *equals-cemptyI* = *equals0I*[*Transfer.transferred*]
lemmas *equals-cemptyD* = *equals0D*[*Transfer.transferred*]
lemmas *cBall-cempty*[*simp*] = *ball-empty*[*Transfer.transferred*]
lemmas *cBex-cempty*[*simp*] = *bex-empty*[*Transfer.transferred*]
lemmas *cInt-iff*[*simp*] = *Int-iff*[*Transfer.transferred*]
lemmas *cIntI*[*intro*!] = *IntI*[*Transfer.transferred*]
lemmas *cIntD1* = *IntD1*[*Transfer.transferred*]
lemmas *cIntD2* = *IntD2*[*Transfer.transferred*]
lemmas *cIntE*[*elim*!] = *IntE*[*Transfer.transferred*]
lemmas *cUn-iff*[*simp*] = *Un-iff*[*Transfer.transferred*]
lemmas *cUnI1*[*elim?*] = *UnI1*[*Transfer.transferred*]
lemmas *cUnI2*[*elim?*] = *UnI2*[*Transfer.transferred*]
lemmas *cUnCI*[*intro*!] = *UnCI*[*Transfer.transferred*]
lemmas *cuUnE*[*elim*!] = *UnE*[*Transfer.transferred*]
lemmas *cDiff-iff*[*simp*] = *Diff-iff*[*Transfer.transferred*]
lemmas *cDiffI*[*intro*!] = *DiffI*[*Transfer.transferred*]
lemmas *cDiffD1* = *DiffD1*[*Transfer.transferred*]
lemmas *cDiffD2* = *DiffD2*[*Transfer.transferred*]
lemmas *cDiffE*[*elim*!] = *DiffE*[*Transfer.transferred*]
lemmas *cinsert-iff*[*simp*] = *insert-iff*[*Transfer.transferred*]
lemmas *cinsertI1* = *insertI1*[*Transfer.transferred*]
lemmas *cinsertI2* = *insertI2*[*Transfer.transferred*]
lemmas *cinsertE*[*elim*!] = *insertE*[*Transfer.transferred*]
lemmas *cinsertCI*[*intro*!] = *insertCI*[*Transfer.transferred*]
lemmas *csubset-cinsert-iff* = *subset-insert-iff*[*Transfer.transferred*]
lemmas *cinsert-ident* = *insert-ident*[*Transfer.transferred*]
lemmas *csingletonI*[*intro*!,*no-atp*] = *singletonI*[*Transfer.transferred*]
lemmas *csingletonD*[*dest*!,*no-atp*] = *singletonD*[*Transfer.transferred*]
lemmas *fsingletonE* = *csingletonD* [*elim-format*]
lemmas *csingleton-iff* = *singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-inject*[*dest*!] = *singleton-inject*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq*[*iff*,*no-atp*] = *singleton-insert-inj-eq*[*Transfer.transferred*]
lemmas *csingleton-finsert-inj-eq'*[*iff*,*no-atp*] = *singleton-insert-inj-eq'*[*Transfer.transferred*]
lemmas *csubset-csingletonD* = *subset-singletonD*[*Transfer.transferred*]
lemmas *cDiff-single-cinsert* = *Diff-single-insert*[*Transfer.transferred*]
lemmas *cdoubleton-eq-iff* = *doubleton-eq-iff*[*Transfer.transferred*]
lemmas *cUn-csingleton-iff* = *Un-singleton-iff*[*Transfer.transferred*]
lemmas *csingleton-cUn-iff* = *singleton-Un-iff*[*Transfer.transferred*]
lemmas *cimage-eqI*[*simp*, *intro*] = *image-eqI*[*Transfer.transferred*]
lemmas *cimageI* = *imageI*[*Transfer.transferred*]
lemmas *rev-cimage-eqI* = *rev-image-eqI*[*Transfer.transferred*]
lemmas *cimageE*[*elim*!] = *imageE*[*Transfer.transferred*]
lemmas *Compr-cimage-eq* = *Compr-image-eq*[*Transfer.transferred*]
lemmas *cimage-cUn* = *image-Un*[*Transfer.transferred*]
lemmas *cimage-iff* = *image-iff*[*Transfer.transferred*]
lemmas *cimage-csubset-iff*[*no-atp*] = *image-subset-iff*[*Transfer.transferred*]
lemmas *cimage-csubsetI* = *image-subsetI*[*Transfer.transferred*]
lemmas *cimage-ident*[*simp*] = *image-ident*[*Transfer.transferred*]
lemmas *if-split-cin1* = *if-split-mem1*[*Transfer.transferred*]

lemmas *if-split-cin2* = *if-split-mem2*[*Transfer.transferred*]
lemmas *cpsubsetI*[*intro*!,*no-atp*] = *psubsetI*[*Transfer.transferred*]
lemmas *cpsubsetE*[*elim*!,*no-atp*] = *psubsetE*[*Transfer.transferred*]
lemmas *cpsubset-finsert-iff* = *psubset-insert-iff*[*Transfer.transferred*]
lemmas *cpsubset-eq* = *psubset-eq*[*Transfer.transferred*]
lemmas *cpsubset-imp-fsubset* = *psubset-imp-subset*[*Transfer.transferred*]
lemmas *cpsubset-trans* = *psubset-trans*[*Transfer.transferred*]
lemmas *cpsubsetD* = *psubsetD*[*Transfer.transferred*]
lemmas *cpsubset-csubset-trans* = *psubset-subset-trans*[*Transfer.transferred*]
lemmas *csubset-cpsubset-trans* = *subset-psubset-trans*[*Transfer.transferred*]
lemmas *cpsubset-imp-ex-fmem* = *psubset-imp-ex-mem*[*Transfer.transferred*]
lemmas *csubset-cinsertI* = *subset-insertI*[*Transfer.transferred*]
lemmas *csubset-cinsertI2* = *subset-insertI2*[*Transfer.transferred*]
lemmas *csubset-cinsert* = *subset-insert*[*Transfer.transferred*]
lemmas *cUn-upper1* = *Un-upper1*[*Transfer.transferred*]
lemmas *cUn-upper2* = *Un-upper2*[*Transfer.transferred*]
lemmas *cUn-least* = *Un-least*[*Transfer.transferred*]
lemmas *cInt-lower1* = *Int-lower1*[*Transfer.transferred*]
lemmas *cInt-lower2* = *Int-lower2*[*Transfer.transferred*]
lemmas *cInt-greatest* = *Int-greatest*[*Transfer.transferred*]
lemmas *cDiff-csubset* = *Diff-subset*[*Transfer.transferred*]
lemmas *cDiff-csubset-conv* = *Diff-subset-conv*[*Transfer.transferred*]
lemmas *csubset-cempty*[*simp*] = *subset-empty*[*Transfer.transferred*]
lemmas *not-cpsubset-cempty*[*iff*] = *not-psubset-empty*[*Transfer.transferred*]
lemmas *cinsert-is-cUn* = *insert-is-Un*[*Transfer.transferred*]
lemmas *cinsert-not-cempty*[*simp*] = *insert-not-empty*[*Transfer.transferred*]
lemmas *cempty-not-cinsert* = *empty-not-insert*[*Transfer.transferred*]
lemmas *cinsert-absorb* = *insert-absorb*[*Transfer.transferred*]
lemmas *cinsert-absorb2*[*simp*] = *insert-absorb2*[*Transfer.transferred*]
lemmas *cinsert-commute* = *insert-commute*[*Transfer.transferred*]
lemmas *cinsert-csubset*[*simp*] = *insert-subset*[*Transfer.transferred*]
lemmas *cinsert-cinter-cinsert*[*simp*] = *insert-inter-insert*[*Transfer.transferred*]
lemmas *cinsert-disjoint*[*simp*,*no-atp*] = *insert-disjoint*[*Transfer.transferred*]
lemmas *disjoint-cinsert*[*simp*,*no-atp*] = *disjoint-insert*[*Transfer.transferred*]
lemmas *cimage-cempty*[*simp*] = *image-empty*[*Transfer.transferred*]
lemmas *cimage-cinsert*[*simp*] = *image-insert*[*Transfer.transferred*]
lemmas *cimage-constant* = *image-constant*[*Transfer.transferred*]
lemmas *cimage-constant-conv* = *image-constant-conv*[*Transfer.transferred*]
lemmas *cimage-cimage* = *image-image*[*Transfer.transferred*]
lemmas *cinsert-cimage*[*simp*] = *insert-image*[*Transfer.transferred*]
lemmas *cimage-is-cempty*[*iff*] = *image-is-empty*[*Transfer.transferred*]
lemmas *cempty-is-cimage*[*iff*] = *empty-is-image*[*Transfer.transferred*]
lemmas *cimage-cong* = *image-cong*[*Transfer.transferred*]
lemmas *cimage-cInt-csubset* = *image-Int-subset*[*Transfer.transferred*]
lemmas *cimage-cDiff-csubset* = *image-diff-subset*[*Transfer.transferred*]
lemmas *cInt-absorb* = *Int-absorb*[*Transfer.transferred*]
lemmas *cInt-left-absorb* = *Int-left-absorb*[*Transfer.transferred*]
lemmas *cInt-commute* = *Int-commute*[*Transfer.transferred*]
lemmas *cInt-left-commute* = *Int-left-commute*[*Transfer.transferred*]

lemmas *cInt-assoc* = *Int-assoc*[*Transfer.transferred*]
lemmas *cInt-ac* = *Int-ac*[*Transfer.transferred*]
lemmas *cInt-absorb1* = *Int-absorb1*[*Transfer.transferred*]
lemmas *cInt-absorb2* = *Int-absorb2*[*Transfer.transferred*]
lemmas *cInt-cempty-left* = *Int-empty-left*[*Transfer.transferred*]
lemmas *cInt-cempty-right* = *Int-empty-right*[*Transfer.transferred*]
lemmas *disjoint-iff-cnot-equal* = *disjoint-iff-not-equal*[*Transfer.transferred*]
lemmas *cInt-cUn-distrib* = *Int-Un-distrib*[*Transfer.transferred*]
lemmas *cInt-cUn-distrib2* = *Int-Un-distrib2*[*Transfer.transferred*]
lemmas *cInt-csubset-iff*[*no-atp*, *simp*] = *Int-subset-iff*[*Transfer.transferred*]
lemmas *cUn-absorb* = *Un-absorb*[*Transfer.transferred*]
lemmas *cUn-left-absorb* = *Un-left-absorb*[*Transfer.transferred*]
lemmas *cUn-commute* = *Un-commute*[*Transfer.transferred*]
lemmas *cUn-left-commute* = *Un-left-commute*[*Transfer.transferred*]
lemmas *cUn-assoc* = *Un-assoc*[*Transfer.transferred*]
lemmas *cUn-ac* = *Un-ac*[*Transfer.transferred*]
lemmas *cUn-absorb1* = *Un-absorb1*[*Transfer.transferred*]
lemmas *cUn-absorb2* = *Un-absorb2*[*Transfer.transferred*]
lemmas *cUn-cempty-left* = *Un-empty-left*[*Transfer.transferred*]
lemmas *cUn-cempty-right* = *Un-empty-right*[*Transfer.transferred*]
lemmas *cUn-cinsert-left*[*simp*] = *Un-insert-left*[*Transfer.transferred*]
lemmas *cUn-cinsert-right*[*simp*] = *Un-insert-right*[*Transfer.transferred*]
lemmas *cInt-cinsert-left* = *Int-insert-left*[*Transfer.transferred*]
lemmas *cInt-cinsert-left-if0*[*simp*] = *Int-insert-left-if0*[*Transfer.transferred*]
lemmas *cInt-cinsert-left-if1*[*simp*] = *Int-insert-left-if1*[*Transfer.transferred*]
lemmas *cInt-cinsert-right* = *Int-insert-right*[*Transfer.transferred*]
lemmas *cInt-cinsert-right-if0*[*simp*] = *Int-insert-right-if0*[*Transfer.transferred*]
lemmas *cInt-cinsert-right-if1*[*simp*] = *Int-insert-right-if1*[*Transfer.transferred*]
lemmas *cUn-cInt-distrib* = *Un-Int-distrib*[*Transfer.transferred*]
lemmas *cUn-cInt-distrib2* = *Un-Int-distrib2*[*Transfer.transferred*]
lemmas *cUn-cInt-crazy* = *Un-Int-crazy*[*Transfer.transferred*]
lemmas *csubset-cUn-eq* = *subset-Un-eq*[*Transfer.transferred*]
lemmas *cUn-cempty*[*iff*] = *Un-empty*[*Transfer.transferred*]
lemmas *cUn-csubset-iff*[*no-atp*, *simp*] = *Un-subset-iff*[*Transfer.transferred*]
lemmas *cUn-cDiff-cInt* = *Un-Diff-Int*[*Transfer.transferred*]
lemmas *cDiff-cInt2* = *Diff-Int2*[*Transfer.transferred*]
lemmas *cUn-cInt-assoc-eq* = *Un-Int-assoc-eq*[*Transfer.transferred*]
lemmas *cBall-cUn* = *ball-Un*[*Transfer.transferred*]
lemmas *cBex-cUn* = *bex-Un*[*Transfer.transferred*]
lemmas *cDiff-eq-cempty-iff*[*simp*,*no-atp*] = *Diff-eq-empty-iff*[*Transfer.transferred*]
lemmas *cDiff-cancel*[*simp*] = *Diff-cancel*[*Transfer.transferred*]
lemmas *cDiff-idemp*[*simp*] = *Diff-idemp*[*Transfer.transferred*]
lemmas *cDiff-triv* = *Diff-triv*[*Transfer.transferred*]
lemmas *cempty-cDiff*[*simp*] = *empty-Diff*[*Transfer.transferred*]
lemmas *cDiff-cempty*[*simp*] = *Diff-empty*[*Transfer.transferred*]
lemmas *cDiff-cinsert0*[*simp*,*no-atp*] = *Diff-insert0*[*Transfer.transferred*]
lemmas *cDiff-cinsert* = *Diff-insert*[*Transfer.transferred*]
lemmas *cDiff-cinsert2* = *Diff-insert2*[*Transfer.transferred*]
lemmas *cinsert-cDiff-if* = *insert-Diff-if*[*Transfer.transferred*]

lemmas *cinsert-cDiff1*[*simp*] = *insert-Diff1*[*Transfer.transferred*]
lemmas *cinsert-cDiff-single*[*simp*] = *insert-Diff-single*[*Transfer.transferred*]
lemmas *cinsert-cDiff* = *insert-Diff*[*Transfer.transferred*]
lemmas *cDiff-cinsert-absorb* = *Diff-insert-absorb*[*Transfer.transferred*]
lemmas *cDiff-disjoint*[*simp*] = *Diff-disjoint*[*Transfer.transferred*]
lemmas *cDiff-partition* = *Diff-partition*[*Transfer.transferred*]
lemmas *double-cDiff* = *double-diff*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel*[*simp*] = *Un-Diff-cancel*[*Transfer.transferred*]
lemmas *cUn-cDiff-cancel2*[*simp*] = *Un-Diff-cancel2*[*Transfer.transferred*]
lemmas *cDiff-cUn* = *Diff-Un*[*Transfer.transferred*]
lemmas *cDiff-cInt* = *Diff-Int*[*Transfer.transferred*]
lemmas *cUn-cDiff* = *Un-Diff*[*Transfer.transferred*]
lemmas *cInt-cDiff* = *Int-Diff*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib* = *Diff-Int-distrib*[*Transfer.transferred*]
lemmas *cDiff-cInt-distrib2* = *Diff-Int-distrib2*[*Transfer.transferred*]
lemmas *cset-eq-csubset* = *set-eq-subset*[*Transfer.transferred*]
lemmas *csubset-iff*[*no-atp*] = *subset-iff*[*Transfer.transferred*]
lemmas *csubset-iff-pfsubset-eq* = *subset-iff-psubset-eq*[*Transfer.transferred*]
lemmas *all-not-cin-conv*[*simp*] = *all-not-in-conv*[*Transfer.transferred*]
lemmas *ex-cin-conv* = *ex-in-conv*[*Transfer.transferred*]
lemmas *cimage-mono* = *image-mono*[*Transfer.transferred*]
lemmas *cinsert-mono* = *insert-mono*[*Transfer.transferred*]
lemmas *cunion-mono* = *Un-mono*[*Transfer.transferred*]
lemmas *cinter-mono* = *Int-mono*[*Transfer.transferred*]
lemmas *cminus-mono* = *Diff-mono*[*Transfer.transferred*]
lemmas *cin-mono* = *in-mono*[*Transfer.transferred*]
lemmas *cLeast-mono* = *Least-mono*[*Transfer.transferred*]
lemmas *cequalityI* = *equalityI*[*Transfer.transferred*]
lemmas *cUN-iff* [*simp*] = *UN-iff*[*Transfer.transferred*]
lemmas *cUN-I* [*intro*] = *UN-I*[*Transfer.transferred*]
lemmas *cUN-E* [*elim*!] = *UN-E*[*Transfer.transferred*]
lemmas *cUN-upper* = *UN-upper*[*Transfer.transferred*]
lemmas *cUN-least* = *UN-least*[*Transfer.transferred*]
lemmas *cUN-cinsert-distrib* = *UN-insert-distrib*[*Transfer.transferred*]
lemmas *cUN-empty* [*simp*] = *UN-empty*[*Transfer.transferred*]
lemmas *cUN-empty2* [*simp*] = *UN-empty2*[*Transfer.transferred*]
lemmas *cUN-absorb* = *UN-absorb*[*Transfer.transferred*]
lemmas *cUN-cinsert* [*simp*] = *UN-insert*[*Transfer.transferred*]
lemmas *cUN-cUn* [*simp*] = *UN-Un*[*Transfer.transferred*]
lemmas *cUN-cUN-flatten* = *UN-UN-flatten*[*Transfer.transferred*]
lemmas *cUN-csubset-iff* = *UN-subset-iff*[*Transfer.transferred*]
lemmas *cUN-constant* [*simp*] = *UN-constant*[*Transfer.transferred*]
lemmas *cimage-cUnion* = *image-Union*[*Transfer.transferred*]
lemmas *cUNION-cempty-conv* [*simp*] = *UNION-empty-conv*[*Transfer.transferred*]
lemmas *cBall-cUN* = *ball-UN*[*Transfer.transferred*]
lemmas *cBex-cUN* = *bex-UN*[*Transfer.transferred*]
lemmas *cUn-eq-cUN* = *Un-eq-UN*[*Transfer.transferred*]
lemmas *cUN-mono* = *UN-mono*[*Transfer.transferred*]
lemmas *cimage-cUN* = *image-UN*[*Transfer.transferred*]

lemmas *cUN-csingleton* [*simp*] = *UN-singleton*[*Transfer.transferred*]

23.3 Additional lemmas

23.3.1 *cempty*

lemma *cemptyE* [*elim*!]: *cin a cempty* $\Longrightarrow$ *P* **by** *simp*

23.3.2 *cinsert*

lemma *countable-insert-iff*: *countable* (*insert x A*) $\longleftrightarrow$ *countable A*
by (*metis Diff-eq-empty-iff countable-empty countable-insert subset-insertI uncountable-minus-countable*)

lemma *set-cinsert*:
assumes *cin x A*
obtains *B* **where** *A* = *cinsert x B* **and** ¬ *cin x B*
using *assms* **by** *transfer*(*erule Set.set-insert*, *simp add*: *countable-insert-iff*)

lemma *mk-disjoint-cinsert*: *cin a A* $\Longrightarrow$ ∃ *B*. *A* = *cinsert a B* ∧ ¬ *cin a B*
by (*rule exI*[**where** *x* = *cDiff A* (*csingle a*)]) *blast*

23.3.3 *cimage*

lemma *subset-cimage-iff*: *csubset-eq B* (*cimage f A*) $\longleftrightarrow$ (∃ *AA*. *csubset-eq AA A* ∧ *B* = *cimage f AA*)
by *transfer* (*metis countable-subset image-mono mem-Collect-eq subset-imageE*)

23.3.4 bounded quantification

lemma *cBex-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBex A* (λ*x*. *P x* ∧ *Q*) = (*cBex A P* ∧ *Q*)
⋀*A P Q*. *cBex A* (λ*x*. *P* ∧ *Q x*) = (*P* ∧ *cBex A Q*)
⋀*P*. *cBex cempty P* = *False*
⋀*a B P*. *cBex* (*cinsert a B*) *P* = (*P a* ∨ *cBex B P*)
⋀*A P f*. *cBex* (*cimage f A*) *P* = *cBex A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *cBex A P*) = *cBall A* (λ*x*. ¬ *P x*)
by *auto*

lemma *cBall-simps* [*simp*, *no-atp*]:
⋀*A P Q*. *cBall A* (λ*x*. *P x* ∨ *Q*) = (*cBall A P* ∨ *Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* ∨ *Q x*) = (*P* ∨ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P* $\longrightarrow$ *Q x*) = (*P* $\longrightarrow$ *cBall A Q*)
⋀*A P Q*. *cBall A* (λ*x*. *P x* $\longrightarrow$ *Q*) = (*cBex A P* $\longrightarrow$ *Q*)
⋀*P*. *cBall cempty P* = *True*
⋀*a B P*. *cBall* (*cinsert a B*) *P* = (*P a* ∧ *cBall B P*)
⋀*A P f*. *cBall* (*cimage f A*) *P* = *cBall A* (λ*x*. *P* (*f x*))
⋀*A P*. (¬ *cBall A P*) = *cBex A* (λ*x*. ¬ *P x*)
by *auto*

lemma *atomize-cBall*:
  $(\bigwedge x.\ cin\ x\ A \Longrightarrow P\ x) == Trueprop\ (cBall\ A\ (\lambda x.\ P\ x))$
 unfolding *atomize-all atomize-imp*
 by (*rule equal-intr-rule*; *blast*)

23.3.5 *cUnion*

lemma *cUNION-cimage*: $cUNION\ (cimage\ f\ A)\ g = cUNION\ A\ (g \circ f)$
 by *transfer simp*

23.4 Setup for Lifting/Transfer

23.4.1 Relator and predicator properties

lift-definition *rel-cset* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow 'a\ cset \Rightarrow 'b\ cset \Rightarrow bool$
 is *rel-set* **parametric** *rel-set-transfer* **.**

lemma *rel-cset-alt-def*:
  rel-cset R a b $\longleftrightarrow$
   $(\forall\, t \in rcset\ a.\ \exists\, u \in rcset\ b.\ R\ t\ u) \wedge$
   $(\forall\, t \in rcset\ b.\ \exists\, u \in rcset\ a.\ R\ u\ t)$
by(*simp add*: *rel-cset-def rel-set-def*)

lemma *rel-cset-iff*:
  rel-cset R a b $\longleftrightarrow$
   $(\forall\, t.\ cin\ t\ a \longrightarrow (\exists\, u.\ cin\ u\ b \wedge R\ t\ u)) \wedge$
   $(\forall\, t.\ cin\ t\ b \longrightarrow (\exists\, u.\ cin\ u\ a \wedge R\ u\ t))$
by *transfer*(*auto simp add*: *rel-set-def*)

lemma *rel-cset-cUNION*:
  $[\![$ *rel-cset Q A B*; *rel-fun Q* (*rel-cset R*) *f g* $]\!]$
  $\Longrightarrow$ *rel-cset R* (*cUnion* (*cimage f A*)) (*cUnion* (*cimage g B*))
unfolding *rel-fun-def* **by** *transfer*(*erule rel-set-UNION*, *simp add*: *rel-fun-def*)

lemma *rel-cset-csingle-iff* [*simp*]: *rel-cset R* (*csingle x*) (*csingle y*) $\longleftrightarrow$ *R x y*
by *transfer*(*auto simp add*: *rel-set-def*)

23.4.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemmas *cempty-parametric* [*transfer-rule*] = *empty-transfer*[*Transfer.transferred*]

lemma *cinsert-parametric* [*transfer-rule*]:
  (*A* ===> *rel-cset A* ===> *rel-cset A*) *cinsert cinsert*
  unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUn-parametric* [*transfer-rule*]:

(*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cUn cUn*
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cUnion-parametric* [*transfer-rule*]:
(*rel-cset* (*rel-cset A*) ===> *rel-cset A*) *cUnion cUnion*
unfolding *rel-fun-def*
by *transfer* (*auto simp*: *rel-set-def*, *metis*+)

lemma *cimage-parametric* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-cset A* ===> *rel-cset B*) *cimage cimage*
unfolding *rel-fun-def rel-cset-iff* **by** *blast*

lemma *cBall-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> (=)) ===> (=)) *cBall cBall*
unfolding *rel-cset-iff rel-fun-def* **by** *blast*

lemma *cBex-parametric* [*transfer-rule*]:
(*rel-cset A* ===> (*A* ===> (=)) ===> (=)) *cBex cBex*
unfolding *rel-cset-iff rel-fun-def* **by** *blast*

lemma *rel-cset-parametric* [*transfer-rule*]:
((*A* ===> *B* ===> (=)) ===> *rel-cset A* ===> *rel-cset B* ===> (=))
rel-cset rel-cset
unfolding *rel-fun-def*
using *rel-set-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*, **where**
A = *A* **and** *B* = *B*]
by *simp*

Rules requiring bi-unique, bi-total or right-total relations

lemma *cin-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*A* ===> *rel-cset A* ===> (=)) *cin cin*
unfolding *rel-fun-def rel-cset-iff bi-unique-def* **by** *metis*

lemma *cInt-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cInt cInt*
unfolding *rel-fun-def*
using *inter-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *cDiff-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> *rel-cset A*) *cDiff cDiff*
unfolding *rel-fun-def*
using *Diff-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *csubset-parametric* [*transfer-rule*]:
bi-unique A ⟹ (*rel-cset A* ===> *rel-cset A* ===> (=)) *csubset-eq csubset-eq*
unfolding *rel-fun-def*
using *subset-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by**
blast

end

lifting-update *cset.lifting*
lifting-forget *cset.lifting*

23.5 Registration as BNF

context
includes *cardinal-syntax*
begin

lemma *card-of-countable-sets-range*:
fixes $A :: 'a\ set$
shows $|\{X.\ X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq o\ |\{f::nat \Rightarrow 'a.\ range\ f \subseteq A\}|$
proof (*intro card-of-ordLeqI*[*of from-nat-into*])
qed (*use inj-on-from-nat-into* **in** ‹*auto simp*: *inj-on-def*›)

lemma *card-of-countable-sets-Func*:
$|\{X.\ X \subseteq A \wedge countable\ X \wedge X \neq \{\}\}| \leq o\ |A|\ ^c\ natLeq$
using *card-of-countable-sets-range card-of-Func-UNIV*[*THEN ordIso-symmetric*]
unfolding *cexp-def Field-natLeq Field-card-of*
by (*rule ordLeq-ordIso-trans*)

lemma *ordLeq-countable-subsets*:
$|A| \leq o\ |\{X.\ X \subseteq A \wedge countable\ X\}|$
proof −
have $\bigwedge a.\ a \in A \Longrightarrow \{a\} \in \{X.\ X \subseteq A \wedge countable\ X\}$
by *auto*
with *card-of-ordLeqI*[*of* $\lambda\ a.\ \{a\}$] **show** *?thesis*
using *inj-singleton* **by** *blast*
qed

end

lemma *finite-countable-subset*:
$finite\ \{X.\ X \subseteq A \wedge countable\ X\} \longleftrightarrow finite\ A$
using *card-of-ordLeq-infinite ordLeq-countable-subsets* **by** *force*

lemma *rcset-to-rcset*: $countable\ A \Longrightarrow rcset\ (the\text{-}inv\ rcset\ A) = A$
including *cset.lifting*
by (*meson CollectI f-the-inv-into-f inj-on-inverseI rangeI rcset-induct rcset-inverse*)

lemma *Collect-Int-Times*: $\{(x,\ y).\ R\ x\ y\} \cap A \times B = \{(x,\ y).\ R\ x\ y \wedge x \in A \wedge y \in B\}$
by *auto*

lemma *rel-cset-aux*:
(∀ *t* ∈ *rcset a*. ∃ *u* ∈ *rcset b*. *R t u*) ∧ (∀ *t* ∈ *rcset b*. ∃ *u* ∈ *rcset a*. *R u t*) ⟷
((*BNF-Def.Grp* {*x*. *rcset x* ⊆ {(*a*, *b*). *R a b*}} (*cimage fst*))$^{-1-1}$ *OO*
BNF-Def.Grp {*x*. *rcset x* ⊆ {(*a*, *b*). *R a b*}} (*cimage snd*)) *a b* (**is** *?L* = *?R*)
proof
assume *?L*
define *R′* **where** *R′* = *the-inv rcset* (*Collect* (*case-prod R*) ∩ (*rcset a* × *rcset b*))
(**is** - = *the-inv rcset ?L′*)
have *L*: *countable ?L′* **by** *auto*
hence ∗: *rcset R′* = *?L′* **unfolding** *R′-def* **by** (*intro rcset-to-rcset*)
thus *?R* **unfolding** *Grp-def relcompp.simps conversep.simps* **including** *cset.lifting*
proof (*intro CollectI case-prodI exI*[*of* - *a*] *exI*[*of* - *b*] *exI*[*of* - *R′*] *conjI refl*)
from ∗ ‹*?L*› **show** *a* = *cimage fst R′* **by** *transfer* (*auto simp*: *image-def Collect-Int-Times*)
from ∗ ‹*?L*› **show** *b* = *cimage snd R′* **by** *transfer* (*auto simp*: *image-def Collect-Int-Times*)
qed *simp-all*
next
assume *?R* **thus** *?L* **unfolding** *Grp-def relcompp.simps conversep.simps*
by (*simp add*: *subset-eq Ball-def*)(*transfer*, *auto simp add*: *cimage.rep-eq*, *metis snd-conv*, *metis fst-conv*)
qed

context
includes *cardinal-syntax*
begin

bnf *′a cset*
map: *cimage*
sets: *rcset*
bd: *card-suc natLeq*
wits: *cempty*
rel: *rel-cset*
proof −
show *cimage id* = *id* **by** *auto*
next
fix *f g* **show** *cimage* (*g* ∘ *f*) = *cimage g* ∘ *cimage f* **by** *fastforce*
next
fix *C f g* **assume** *eq*: ⋀*a*. *a* ∈ *rcset C* ⟹ *f a* = *g a*
thus *cimage f C* = *cimage g C* **including** *cset.lifting* **by** *transfer force*
next
fix *f* **show** *rcset* ∘ *cimage f* = (‘) *f* ∘ *rcset* **including** *cset.lifting* **by** *transfer′ fastforce*
next
show *card-order* (*card-suc natLeq*) **by** (*rule card-order-card-suc*[*OF natLeq-card-order*])
next
show *cinfinite* (*card-suc natLeq*) **using** *Cinfinite-card-suc*[*OF natLeq-Cinfinite natLeq-card-order*]
by *simp*

next
show *regularCard* (*card-suc natLeq*) **using** *natLeq-card-order natLeq-Cinfinite*
by (*rule regularCard-card-suc*)
next
fix *C*
have |*rcset C*| $\leq o$ *natLeq* **including** *cset.lifting* **by** *transfer* (*unfold countable-card-le-natLeq*)
then show |*rcset C*| $<o$ *card-suc natLeq*
using *card-suc-greater natLeq-card-order ordLeq-ordLess-trans* **by** *blast*
next
fix *R S*
show *rel-cset R OO rel-cset S* $\leq$ *rel-cset* (*R OO S*)
unfolding *rel-cset-alt-def*[*abs-def*] **by** *fast*
next
fix *R*
show *rel-cset R* = ($\lambda x\ y.\ \exists z.$ *rcset z* $\subseteq$ {(*x*, *y*). *R x y*} $\wedge$
cimage fst z = *x* $\wedge$ *cimage snd z* = *y*)
unfolding *rel-cset-alt-def*[*abs-def*] *rel-cset-aux*[*unfolded OO-Grp-alt*] **by** *simp*
qed(*simp add*: *bot-cset.rep-eq*)

end

end

24 Debugging facilities for code generated towards Isabelle/ML

theory *Debug*
imports *Main*
begin

context
begin

qualified definition *trace* :: *String.literal* $\Rightarrow$ *unit* **where**
[*simp*]: *trace s* = ()

qualified definition *tracing* :: *String.literal* $\Rightarrow$ *'a* $\Rightarrow$ *'a* **where**
[*simp*]: *tracing s* = *id*

lemma [*code*]:
tracing s = (*let u* = *trace s in id*)
by *simp*

qualified definition *flush* :: *'a* $\Rightarrow$ *unit* **where**
[*simp*]: *flush x* = ()

qualified definition *flushing* :: *'a* $\Rightarrow$ *'b* $\Rightarrow$ *'b* **where**

[*simp*]: *flushing x = id*

lemma [*code*, *code-unfold*]:
flushing x = (let u = flush x in id)
by *simp*

qualified definition *timing* :: *String.literal* ⇒ (*'a* ⇒ *'b*) ⇒ *'a* ⇒ *'b* **where**
[*simp*]: *timing s f x = f x*

end

code-printing
constant *Debug.trace* ⇀ (*Eval*) *Output.tracing*
| **constant** *Debug.flush* ⇀ (*Eval*) *Output.tracing*/ (@{*make'-string*} -) — note indirection via antiquotation
| **constant** *Debug.timing* ⇀ (*Eval*) *Timing.timeap'-msg*

code-reserved (*Eval*) *Output Timing*

end

25 Sequence of Properties on Subsequences

theory *Diagonal-Subsequence*
imports *Complex-Main*
begin

locale *subseqs* =
fixes *P*::*nat*⇒(*nat*⇒*nat*)⇒*bool*
assumes *ex-subseq*: ⋀*n s*. *strict-mono* (*s*::*nat*⇒*nat*) ⟹ ∃ *r'*. *strict-mono r'* ∧ *P n* (*s* ∘ *r'*)
begin

definition *reduce* **where** *reduce s n* = (*SOME r'*::*nat*⇒*nat*. *strict-mono r'* ∧ *P n* (*s* ∘ *r'*))

lemma *subseq-reduce*[*intro*, *simp*]:
strict-mono s ⟹ *strict-mono* (*reduce s n*)
unfolding *reduce-def* **by** (*rule someI2-ex*[*OF ex-subseq*]) *auto*

lemma *reduce-holds*:
strict-mono s ⟹ *P n* (*s* ∘ *reduce s n*)
unfolding *reduce-def* **by** (*rule someI2-ex*[*OF ex-subseq*]) (*auto simp*: *o-def*)

primrec *seqseq* :: *nat* ⇒ *nat* ⇒ *nat* **where**
seqseq 0 = id
| *seqseq* (*Suc n*) = *seqseq n* ∘ *reduce* (*seqseq n*) *n*

lemma *subseq-seqseq*[*intro*, *simp*]: *strict-mono* (*seqseq n*)

proof (*induct n*)
case *0* **thus** *?case* **by** (*simp add*: *strict-mono-def*)
next
case (*Suc n*) **thus** *?case* **by** (*subst seqseq.simps*) (*auto intro*!: *strict-mono-o*)
qed

lemma *seqseq-holds*:
P n (*seqseq* (*Suc n*))
proof −
have *P n* (*seqseq n* ∘ *reduce* (*seqseq n*) *n*)
by (*intro reduce-holds subseq-seqseq*)
thus *?thesis* **by** *simp*
qed

definition *diagseq* :: *nat* ⇒ *nat* **where** *diagseq i* = *seqseq i i*

lemma *diagseq-mono*: *diagseq n* < *diagseq* (*Suc n*)
proof −
have *diagseq n* < *seqseq n* (*Suc n*)
using *subseq-seqseq*[*of n*] **by** (*simp add*: *diagseq-def strict-mono-def*)
also have ... ≤ *seqseq n* (*reduce* (*seqseq n*) *n* (*Suc n*))
using *strict-mono-less-eq seq-suble* **by** *blast*
also have ... = *diagseq* (*Suc n*) **by** (*simp add*: *diagseq-def*)
finally show *?thesis* **.**
qed

lemma *subseq-diagseq*: *strict-mono diagseq*
using *diagseq-mono* **by** (*simp add*: *strict-mono-Suc-iff diagseq-def*)

primrec *fold-reduce* **where**
fold-reduce n 0 = *id*
| *fold-reduce n* (*Suc k*) = *fold-reduce n k* ∘ *reduce* (*seqseq* (*n* + *k*)) (*n* + *k*)

lemma *subseq-fold-reduce*[*intro*, *simp*]: *strict-mono* (*fold-reduce n k*)
proof (*induct k*)
case (*Suc k*) **from** *strict-mono-o*[*OF this subseq-reduce*] **show** *?case* **by** (*simp add*: *o-def*)
qed (*simp add*: *strict-mono-def*)

lemma *ex-subseq-reduce-index*: *seqseq* (*n* + *k*) = *seqseq n* ∘ *fold-reduce n k*
by (*induct k*) *simp-all*

lemma *seqseq-fold-reduce*: *seqseq n* = *fold-reduce 0 n*
by (*induct n*) (*simp-all*)

lemma *diagseq-fold-reduce*: *diagseq n* = *fold-reduce 0 n n*
using *seqseq-fold-reduce* **by** (*simp add*: *diagseq-def*)

lemma *fold-reduce-add*: *fold-reduce 0* (*m* + *n*) = *fold-reduce 0 m* ∘ *fold-reduce m*

n
by (*induct n*) *simp-all*

lemma *diagseq-add*: *diagseq* (*k* + *n*) = (*seqseq k* ∘ (*fold-reduce k n*)) (*k* + *n*)
proof −
have *diagseq* (*k* + *n*) = *fold-reduce 0* (*k* + *n*) (*k* + *n*)
by (*simp add*: *diagseq-fold-reduce*)
also have ... = (*seqseq k* ∘ *fold-reduce k n*) (*k* + *n*)
unfolding *fold-reduce-add seqseq-fold-reduce* **..**
finally show *?thesis* **.**
qed

lemma *diagseq-sub*:
assumes *m* ≤ *n* **shows** *diagseq n* = (*seqseq m* ∘ (*fold-reduce m* (*n* − *m*))) *n*
using *diagseq-add*[*of m n* − *m*] *assms* **by** *simp*

lemma *subseq-diagonal-rest*: *strict-mono* (λ*x*. *fold-reduce k x* (*k* + *x*))
unfolding *strict-mono-Suc-iff fold-reduce.simps o-def*
proof
fix *n*
have *fold-reduce k n* (*k* + *n*) < *fold-reduce k n* (*k* + *Suc n*) (**is** *?lhs* < -)
by (*auto intro*: *strict-monoD*)
also have ... ≤ *fold-reduce k n* (*reduce* (*seqseq* (*k* + *n*)) (*k* + *n*) (*k* + *Suc n*))
by (*auto intro*: *less-mono-imp-le-mono seq-suble strict-monoD*)
finally show *?lhs* < ... **.**
qed

lemma *diagseq-seqseq*: *diagseq* ∘ ((+) *k*) = (*seqseq k* ∘ (λ*x*. *fold-reduce k x* (*k* + *x*)))
by (*auto simp*: *o-def diagseq-add*)

lemma *diagseq-holds*:
assumes *subseq-stable*: ⋀*r s n*. *strict-mono r* ⟹ *P n s* ⟹ *P n* (*s* ∘ *r*)
shows *P k* (*diagseq* ∘ ((+) (*Suc k*)))
unfolding *diagseq-seqseq* **by** (*intro subseq-stable subseq-diagonal-rest seqseq-holds*)

end

end

26 Common discrete functions

theory *Discrete-Functions*
imports *Complex-Main*
begin

26.1 Discrete logarithm

fun *floor-log* :: *nat* ⇒ *nat*

where [*simp del*]: *floor-log n* = (*if n* < *2 then 0 else Suc* (*floor-log* (*n div 2*)))

lemma *floor-log-induct* [*consumes 1*, *case-names one double*]:
fixes *n* :: *nat*
assumes *n* > *0*
assumes *one*: *P 1*
assumes *double*: ⋀*n*. *n* ≥ *2* ⟹ *P* (*n div 2*) ⟹ *P n*
shows *P n*
using ‹*n* > *0*› **proof** (*induct n rule*: *floor-log.induct*)
fix *n*
assume ¬ *n* < *2* ⟹
0 < *n div 2* ⟹ *P* (*n div 2*)
then have ∗: *n* ≥ *2* ⟹ *P* (*n div 2*) **by** *simp*
assume *n* > *0*
show *P n*
proof (*cases n* = *1*)
case *True*
with *one* **show** *?thesis* **by** *simp*
next
case *False*
with ‹*n* > *0*› **have** *n* ≥ *2* **by** *auto*
with ∗ **have** *P* (*n div 2*) **.**
with ‹*n* ≥ *2*› **show** *?thesis* **by** (*rule double*)
qed
qed

lemma *floor-log-zero* [*simp*]: *floor-log 0* = *0*
by (*simp add*: *floor-log.simps*)

lemma *floor-log-one* [*simp*]: *floor-log 1* = *0*
by (*simp add*: *floor-log.simps*)

lemma *floor-log-Suc-zero* [*simp*]: *floor-log* (*Suc 0*) = *0*
using *floor-log-one* **by** *simp*

lemma *floor-log-rec*: *n* ≥ *2* ⟹ *floor-log n* = *Suc* (*floor-log* (*n div 2*))
by (*simp add*: *floor-log.simps*)

lemma *floor-log-twice* [*simp*]: *n* ≠ *0* ⟹ *floor-log* (*2* ∗ *n*) = *Suc* (*floor-log n*)
by (*simp add*: *floor-log-rec*)

lemma *floor-log-half* [*simp*]: *floor-log* (*n div 2*) = *floor-log n* − *1*
proof (*cases n* < *2*)
case *True*
then have *n* = *0* ∨ *n* = *1* **by** *arith*
then show *?thesis* **by** (*auto simp del*: *One-nat-def*)
next
case *False*
then show *?thesis* **by** (*simp add*: *floor-log-rec*)

qed

lemma *floor-log-power* [*simp*]: *floor-log* (*2* ^ *n*) = *n*
by (*induct n*) *simp-all*

lemma *floor-log-mono*: *mono floor-log*
proof
fix *m n* :: *nat*
assume *m* ≤ *n*
then show *floor-log m* ≤ *floor-log n*
proof (*induct m arbitrary*: *n rule*: *floor-log.induct*)
case (*1 m*)
then have *mn2*: *m div 2* ≤ *n div 2* **by** *arith*
show *floor-log m* ≤ *floor-log n*
proof (*cases m* ≥ *2*)
case *False*
then have *m* = *0* ∨ *m* = *1* **by** *arith*
then show *?thesis* **by** (*auto simp del*: *One-nat-def*)
next
case *True* **then have** ¬ *m* < *2* **by** *simp*
with *mn2* **have** *n* ≥ *2* **by** *arith*
from *True* **have** *m2-0*: *m div 2* ≠ *0* **by** *arith*
with *mn2* **have** *n2-0*: *n div 2* ≠ *0* **by** *arith*
from ‹¬ *m* < *2*› *1.hyps mn2* **have** *floor-log* (*m div 2*) ≤ *floor-log* (*n div 2*)
by *blast*
with *m2-0 n2-0* **have** *floor-log* (*2* ∗ (*m div 2*)) ≤ *floor-log* (*2* ∗ (*n div 2*))
by *simp*
with *m2-0 n2-0* ‹*m* ≥ *2*› ‹*n* ≥ *2*› **show** *?thesis* **by** (*simp only*: *floor-log-rec* [*of m*] *floor-log-rec* [*of n*]) *simp*
qed
qed
qed

lemma *floor-log-exp2-le*:
assumes *n* > *0*
shows *2* ^ *floor-log n* ≤ *n*
using *assms*
proof (*induct n rule*: *floor-log-induct*)
case *one*
then show *?case* **by** *simp*
next
case (*double n*)
with *floor-log-mono* **have** *floor-log n* ≥ *Suc 0*
by (*simp add*: *floor-log.simps*)
assume *2* ^ *floor-log* (*n div 2*) ≤ *n div 2*
with ‹*n* ≥ *2*› **have** *2* ^ (*floor-log n* − *Suc 0*) ≤ *n div 2* **by** *simp*
then have *2* ^ (*floor-log n* − *Suc 0*) ∗ *2* ^ *1* ≤ *n div 2* ∗ *2* **by** *simp*
with ‹*floor-log n* ≥ *Suc 0*› **have** *2* ^ *floor-log n* ≤ *n div 2* ∗ *2*
unfolding *power-add* [*symmetric*] **by** *simp*

also have *n div 2 * 2 ≤ n* **by** (*cases even n*) *simp-all*
finally show *?case* **.**
qed

lemma *floor-log-exp2-gt*: *2 * 2 ^ floor-log n > n*
proof (*cases n > 0*)
case *True*
thus *?thesis*
proof (*induct n rule*: *floor-log-induct*)
case (*double n*)
thus *?case*
by (*cases even n*) (*auto elim*!: *evenE oddE simp*: *field-simps floor-log.simps*)
qed *simp-all*
qed *simp-all*

lemma *floor-log-exp2-ge*: *2 * 2 ^ floor-log n ≥ n*
using *floor-log-exp2-gt*[*of n*] **by** *simp*

lemma *floor-log-le-iff*: *m ≤ n ⟹ floor-log m ≤ floor-log n*
by (*rule monoD* [*OF floor-log-mono*])

lemma *floor-log-eqI*:
assumes *n > 0 2^k ≤ n n < 2 * 2^k*
shows *floor-log n = k*
proof (*rule antisym*)
from ‹*n > 0*› **have** *2 ^ floor-log n ≤ n* **by** (*rule floor-log-exp2-le*)
also have *... < 2 ^ Suc k* **using** *assms* **by** *simp*
finally have *floor-log n < Suc k* **by** (*subst* (*asm*) *power-strict-increasing-iff*) *simp-all*
thus *floor-log n ≤ k* **by** *simp*
next
have *2^k ≤ n* **by** *fact*
also have *... < 2^(Suc* (*floor-log n*)) **by** (*simp add*: *floor-log-exp2-gt*)
finally have *k < Suc* (*floor-log n*) **by** (*subst* (*asm*) *power-strict-increasing-iff*) *simp-all*
thus *k ≤ floor-log n* **by** *simp*
qed

lemma *floor-log-altdef*: *floor-log n = (if n = 0 then 0 else nat ⌊log 2* (*real-of-nat n*)⌋)
proof (*cases n = 0*)
case *False*
have ⌊*log 2* (*real-of-nat n*)⌋ = *int* (*floor-log n*)
proof (*rule floor-unique*)
from *False* **have** *2 powr* (*real* (*floor-log n*)) ≤ *real n*
by (*simp add*: *powr-realpow floor-log-exp2-le*)
hence *log 2* (*2 powr* (*real* (*floor-log n*))) ≤ *log 2* (*real n*)
using *False* **by** (*subst log-le-cancel-iff*) *simp-all*
also have *log 2* (*2 powr* (*real* (*floor-log n*))) = *real* (*floor-log n*) **by** *simp*

```
    finally show real-of-int (int (floor-log n)) ≤ log 2 (real n) by simp
  next
    have real n < real (2 * 2 ^ floor-log n)
      by (subst of-nat-less-iff) (rule floor-log-exp2-gt)
    also have ... = 2 powr (real (floor-log n) + 1)
      by (simp add: powr-add powr-realpow)
    finally have log 2 (real n) < log 2 ...
      using False by (subst log-less-cancel-iff) simp-all
    also have ... = real (floor-log n) + 1 by simp
    finally show log 2 (real n) < real-of-int (int (floor-log n)) + 1 by simp
  qed
  thus ?thesis by simp
qed simp-all
```

26.2 Discrete square root

```
definition floor-sqrt :: nat ⇒ nat
  where floor-sqrt n = Max {m. m^2 ≤ n}

lemma floor-sqrt-aux:
  fixes n :: nat
  shows finite {m. m^2 ≤ n} and {m. m^2 ≤ n} ≠ {}
proof −
  have **: m ≤ n if m^2 ≤ n for m
    using that by (cases m) (simp-all add: power2-eq-square)
  then have {m. m^2 ≤ n} ⊆ {m. m ≤ n} by auto
  then show finite {m. m^2 ≤ n} by (rule finite-subset) rule
  have 0^2 ≤ n by simp
  then show *: {m. m^2 ≤ n} ≠ {} by blast
qed

lemma floor-sqrt-unique:
  assumes m^2 ≤ n n < (Suc m)^2
  shows   floor-sqrt n = m
proof −
  have m' ≤ m if m'^2 ≤ n for m'
  proof −
    note that
    also note assms(2)
    finally have m' < Suc m by (rule power-less-imp-less-base) simp-all
    thus m' ≤ m by simp
  qed
  with ‹m^2 ≤ n› floor-sqrt-aux[of n] show ?thesis unfolding floor-sqrt-def
    by (intro antisym Max.boundedI Max.coboundedI) simp-all
qed

lemma floor-sqrt-inverse-power2 [simp]: floor-sqrt (n^2) = n
proof −
  have {m. m ≤ n} ≠ {} by auto
```

then have *Max* $\{m.\ m \leq n\} \leq n$ **by** *auto*
then show *?thesis*
by (*auto simp add*: *floor-sqrt-def power2-nat-le-eq-le intro*: *antisym*)
qed

lemma *floor-sqrt-zero* [*simp*]: *floor-sqrt 0 = 0*
using *floor-sqrt-inverse-power2* [*of 0*] **by** *simp*

lemma *floor-sqrt-one* [*simp*]: *floor-sqrt 1 = 1*
using *floor-sqrt-inverse-power2* [*of 1*] **by** *simp*

lemma *floor-sqrt-Suc-0* [*simp*]:
‹*floor-sqrt* (*Suc 0*) = *1*›
using *floor-sqrt-inverse-power2* [*of 1*] **by** *simp*

lemma *mono-floor-sqrt*: *mono floor-sqrt*
proof
fix *m n* :: *nat*
have ∗: *0* ∗ *0* ≤ *m* **by** *simp*
assume *m* ≤ *n*
then show *floor-sqrt m* ≤ *floor-sqrt n*
by (*auto intro*!: *Max-mono* ‹*0* ∗ *0* ≤ *m*› *finite-less-ub simp add*: *power2-eq-square floor-sqrt-def*)
qed

lemma *mono-floor-sqrt'*: *m* ≤ *n* ⟹ *floor-sqrt m* ≤ *floor-sqrt n*
using *mono-floor-sqrt* **unfolding** *mono-def* **by** *auto*

lemma *floor-sqrt-greater-zero-iff* [*simp*]: *floor-sqrt n > 0* ⟷ *n > 0*
proof −
have ∗: *0 < Max* $\{m.\ m^2 \leq n\}$ ⟷ (∃ *a*∈$\{m.\ m^2 \leq n\}$. *0 < a*)
by (*rule Max-gr-iff*) (*fact floor-sqrt-aux*)+
show *?thesis*
proof
assume *0 < floor-sqrt n*
then have *0 < Max* $\{m.\ m^2 \leq n\}$ **by** (*simp add*: *floor-sqrt-def*)
with ∗ **show** *0 < n* **by** (*auto dest*: *power2-nat-le-imp-le*)
next
assume *0 < n*
then have $1^2 \leq n \wedge 0 < (1::nat)$ **by** *simp*
then have ∃ *q*. $q^2 \leq n \wedge 0 < q$ **..**
with ∗ **have** *0 < Max* $\{m.\ m^2 \leq n\}$ **by** *blast*
then show *0 < floor-sqrt n* **by** (*simp add*: *floor-sqrt-def*)
qed
qed

lemma *floor-sqrt-power2-le* [*simp*]: $(floor\text{-}sqrt\ n)^2 \leq n$
proof (*cases n > 0*)
case *False* **then show** *?thesis* **by** *simp*

next
case *True* **then have** *floor-sqrt n > 0* **by** *simp*
then have *mono* (*times* (*Max* {*m*. $m^2 \leq n$})) **by** (*auto intro*: *mono-times-nat simp add*: *floor-sqrt-def*)
then have ∗: *Max* {*m*. $m^2 \leq n$} ∗ *Max* {*m*. $m^2 \leq n$} = *Max* (*times* (*Max* {*m*. $m^2 \leq n$}) ‘ {*m*. $m^2 \leq n$})
using *floor-sqrt-aux* [*of n*] **by** (*rule mono-Max-commute*)
have $\bigwedge a.\ a * a \leq n \Longrightarrow$ *Max* {*m*. $m * m \leq n$} ∗ $a \leq n$
proof −
fix *q*
assume $q * q \leq n$
show *Max* {*m*. $m * m \leq n$} ∗ $q \leq n$
proof (*cases q > 0*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **then have** *mono* (*times q*) **by** (*rule mono-times-nat*)
then have q ∗ *Max* {*m*. $m * m \leq n$} = *Max* (*times q* ‘ {*m*. $m * m \leq n$})
using *floor-sqrt-aux* [*of n*] **by** (*auto simp add*: *power2-eq-square intro*: *mono-Max-commute*)
then have *Max* {*m*. $m * m \leq n$} ∗ q = *Max* (*times q* ‘ {*m*. $m * m \leq n$}) **by** (*simp add*: *ac-simps*)
moreover have *finite* ((∗) q ‘ {*m*. $m * m \leq n$})
by (*metis* (*mono-tags*) *finite-imageI finite-less-ub le-square*)
moreover have $\exists x.\ x * x \leq n$
by (*metis* ‹$q * q \leq n$›)
ultimately show *?thesis*
by *simp* (*metis* ‹$q * q \leq n$› *le-cases mult-le-mono1 mult-le-mono2 order-trans*)
qed
qed
then have *Max* ((∗) (*Max* {*m*. $m * m \leq n$}) ‘ {*m*. $m * m \leq n$}) ≤ n
apply (*subst Max-le-iff*)
apply (*metis* (*mono-tags*) *finite-imageI finite-less-ub le-square*)
apply *auto*
apply (*metis le0 mult-0-right*)
done
with ∗ **show** *?thesis* **by** (*simp add*: *floor-sqrt-def power2-eq-square*)
qed

lemma *floor-sqrt-le*: *floor-sqrt n* ≤ n
using *floor-sqrt-aux* [*of n*] **by** (*auto simp add*: *floor-sqrt-def intro*: *power2-nat-le-imp-le*)

Additional facts about the discrete square root, thanks to Julian Biendarra, Manuel Eberl

lemma *Suc-floor-sqrt-power2-gt*: n < (*Suc* (*floor-sqrt n*))^2
using *Max-ge*[*OF floor-sqrt-aux*(*1*), *of floor-sqrt n* + *1 n*]
by (*cases n* < (*Suc* (*floor-sqrt n*))^2) (*simp-all add*: *floor-sqrt-def*)

lemma *le-floor-sqrt-iff*: $x \leq$ *floor-sqrt y* ⟷ x^2 ≤ y
proof −

```
  have x ≤ floor-sqrt y ⟷ (∃ z. z^2 ≤ y ∧ x ≤ z)
    using Max-ge-iff[OF floor-sqrt-aux, of x y] by (simp add: floor-sqrt-def)
  also have ... ⟷ x^2 ≤ y
  proof safe
    fix z assume x ≤ z z ^ 2 ≤ y
   thus x^2 ≤ y by (intro le-trans[of x^2 z^2 y]) (simp-all add: power2-nat-le-eq-le)
  qed auto
  finally show ?thesis .
qed

lemma le-floor-sqrtI: x^2 ≤ y ⟹ x ≤ floor-sqrt y
  by (simp add: le-floor-sqrt-iff)

lemma floor-sqrt-le-iff:
  ‹floor-sqrt y ≤ x ⟷ (∀ z. z^2 ≤ y ⟶ z ≤ x)›
  using Max.bounded-iff [OF floor-sqrt-aux]
  by (simp add: floor-sqrt-def)

lemma floor-sqrt-leI:
  (⋀z. z^2 ≤ y ⟹ z ≤ x) ⟹ floor-sqrt y ≤ x
  by (simp add: floor-sqrt-le-iff)

lemma floor-sqrt-less-eq-half:
  ‹floor-sqrt n ≤ Suc n div 2›
proof (rule floor-sqrt-leI)
  fix m
  assume ‹m^2 ≤ n›
  have ‹m < Suc (Suc n div 2)›
  proof (rule ccontr, unfold not-less)
    assume ‹Suc (Suc n div 2) ≤ m›
    then have ‹(Suc (Suc n div 2))^2 ≤ m^2›
      by simp
    then have ‹(Suc (Suc n div 2))^2 ≤ n›
      using ‹m^2 ≤ n› by (rule order-trans)
    then show False
      by (simp only: Suc-eq-plus1 power2-sum algebra-simps) auto
  qed
  then show ‹m ≤ Suc n div 2›
    by simp
qed

lemma floor-sqrt-Suc:
  floor-sqrt (Suc n) = (if ∃ m. Suc n = m^2 then Suc (floor-sqrt n) else floor-sqrt
n)
proof cases
  assume ∃ m. Suc n = m^2
  then obtain m where m-def: Suc n = m^2 by blast
  then have lhs: floor-sqrt (Suc n) = m by simp
  from m-def floor-sqrt-power2-le[of n]
```

```
    have (floor-sqrt n)^2 < m^2 by linarith
  with power2-less-imp-less have lt-m: floor-sqrt n < m by blast
  from m-def Suc-floor-sqrt-power2-gt[of n]
    have m^2 ≤ (Suc(floor-sqrt n))^2
      by linarith
  with power2-nat-le-eq-le have m ≤ Suc (floor-sqrt n) by blast
  with lt-m have m = Suc (floor-sqrt n) by simp
  with lhs m-def show ?thesis by fastforce
next
  assume asm: ¬ (∃ m. Suc n = m^2)
  hence Suc n ≠ (floor-sqrt (Suc n))^2 by simp
  with floor-sqrt-power2-le[of Suc n]
    have floor-sqrt (Suc n) ≤ floor-sqrt n by (intro le-floor-sqrtI) linarith
  moreover have floor-sqrt (Suc n) ≥ floor-sqrt n
    by (intro monoD[OF mono-floor-sqrt]) simp-all
  ultimately show ?thesis using asm by simp
qed
```

Computation by divide and conquer

```
definition floor-sqrt-between :: ‹nat ⇒ nat ⇒ nat ⇒ nat›
  where floor-sqrt-between-eq:
    ‹floor-sqrt-between m q n =
      (if floor-sqrt n ∈ {m..<m + q} then floor-sqrt n else 0)›
    — The 0 is not for relevant regular computation and can be chosen arbitrarily.

lemma floor-sqrt-between-out-of-bounds:
  ‹floor-sqrt-between m 0 n = 0›
  by (simp add: floor-sqrt-between-eq)

lemma floor-sqrt-between-singleton:
  ‹floor-sqrt-between m (Suc 0) n =
    (if m^2 ≤ n ∧ n < (Suc m)^2 then m else 0)›
 by (auto simp add: floor-sqrt-between-eq Suc-floor-sqrt-power2-gt floor-sqrt-unique)

lemma floor-sqrt-between-rec:
  ‹floor-sqrt-between m q n = (
    let
      r = q div 2;
      p = m + r;
      s = p^2
    in
      if s = n
      then p
      else if s < n
        then floor-sqrt-between (m + r) (q - r) n
        else floor-sqrt-between m r n
  )› if ‹q > 0›
  using that le-floor-sqrt-iff [of ‹m + q div 2› n]
  by (auto simp add: floor-sqrt-between-eq Let-def not-less)
```

lemma *floor-sqrt-between-code* [*code*]:
  ‹*floor-sqrt-between m q n* = (
    if q = *0 then 0*
    else if q = *1*
    then if $m^2 \leq n \wedge n < (Suc\ m)^2$
      then m
      else 0
    else
      let
        $r = q\ div\ 2$;
        $p = m + r$;
        $s = p^2$
      in
        if s = *n*
        then p
        else if s < *n*
          then floor-sqrt-between (*m* + *r*) (*q* − *r*) *n*
          else floor-sqrt-between m r n
)›
proof −
  consider ‹*q* = *0*› | ‹*q* = *1*› | ‹*q* ≥ *2*›
    by (*cases* ‹*q* ≥ *2*›; *cases q*) *simp-all*
  then show *?thesis*
    by *cases*
      (*simp-all add*: *floor-sqrt-between-out-of-bounds floor-sqrt-between-singleton floor-sqrt-between-rec*)
qed

lemma [*code*]:
  ‹*floor-sqrt n* = *floor-sqrt-between 0* (*Suc* (*Suc n div 2*)) *n*›
  using *floor-sqrt-less-eq-half* [*of n*] **by** (*simp add*: *floor-sqrt-between-eq*)

end

27 Pi and Function Sets

theory *FuncSet*
  imports *Main*
  abbrevs *PiE* = Pi_E
    and *PIE* = Π_E
begin

definition *Pi* :: *'a set* ⇒ (*'a* ⇒ *'b set*) ⇒ (*'a* ⇒ *'b*) *set*
  where *Pi A B* = {*f*. ∀ *x*. *x* ∈ *A* ⟶ *f x* ∈ *B x*}

definition *extensional* :: *'a set* ⇒ (*'a* ⇒ *'b*) *set*
  where *extensional A* = {*f*. ∀ *x*. *x* ∉ *A* ⟶ *f x* = *undefined*}

definition *restrict* :: $('a \Rightarrow 'b) \Rightarrow 'a\ set \Rightarrow 'a \Rightarrow 'b$
where $restrict\ f\ A = (\lambda x.\ if\ x \in A\ then\ f\ x\ else\ undefined)$

abbreviation *funcset* :: $'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$
where $funcset\ A\ B \equiv Pi\ A\ (\lambda\text{-}.\ B)$

open-bundle *funcset-syntax*
begin
notation *funcset* (**infixr** ‹→› *60*)
end

syntax
$\text{-}Pi :: pttrn \Rightarrow 'a\ set \Rightarrow 'b\ set \Rightarrow ('a \Rightarrow 'b)\ set$
(‹(‹*indent=3 notation=‹binder* Π∈››Π -∈-./ -)› *10*)
$\text{-}lam :: pttrn \Rightarrow 'a\ set \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'b)$
(‹(‹*indent=3 notation=‹binder* λ∈››λ-∈-./ -)› [*0*, *0*, *3*] *3*)
syntax-consts
$\text{-}Pi \rightleftharpoons Pi$ **and**
$\text{-}lam \rightleftharpoons restrict$
translations
$\Pi\ x{\in}A.\ B \rightleftharpoons CONST\ Pi\ A\ (\lambda x.\ B)$
$\lambda x{\in}A.\ f \rightleftharpoons CONST\ restrict\ (\lambda x.\ f)\ A$

definition *compose* :: $'a\ set \Rightarrow ('b \Rightarrow 'c) \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a \Rightarrow 'c)$
where $compose\ A\ g\ f = (\lambda x{\in}A.\ g\ (f\ x))$

27.1 Basic Properties of *Pi*

lemma *Pi-I*[*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
by (*simp add*: *Pi-def*)

lemma *Pi-I'*[*simp*]: $(\bigwedge x.\ x \in A \longrightarrow f\ x \in B\ x) \Longrightarrow f \in Pi\ A\ B$
by (*simp add*:*Pi-def*)

lemma *funcsetI*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B) \Longrightarrow f \in A \to B$
by (*simp add*: *Pi-def*)

lemma *Pi-mem*: $f \in Pi\ A\ B \Longrightarrow x \in A \Longrightarrow f\ x \in B\ x$
by (*simp add*: *Pi-def*)

lemma *Pi-iff*: $f \in Pi\ I\ X \longleftrightarrow (\forall i{\in}I.\ f\ i \in X\ i)$
unfolding *Pi-def* **by** *auto*

lemma *PiE* [*elim*]: $f \in Pi\ A\ B \Longrightarrow (f\ x \in B\ x \Longrightarrow Q) \Longrightarrow (x \notin A \Longrightarrow Q) \Longrightarrow Q$
by (*auto simp*: *Pi-def*)

lemma *Pi-cong*: $(\bigwedge w.\ w \in A \Longrightarrow f\ w = g\ w) \Longrightarrow f \in Pi\ A\ B \longleftrightarrow g \in Pi\ A\ B$
by (*auto simp*: *Pi-def*)

lemma *funcset-id* [*simp*]: $(\lambda x.\ x) \in A \to A$
by *auto*

lemma *funcset-mem*: $f \in A \to B \Longrightarrow x \in A \Longrightarrow f\ x \in B$
by (*simp add*: *Pi-def*)

lemma *funcset-image*: $f \in A \to B \Longrightarrow f\ `\ A \subseteq B$
by *auto*

lemma *image-subset-iff-funcset*: $F\ `\ A \subseteq B \longleftrightarrow F \in A \to B$
by *auto*

lemma *funcset-to-empty-iff*: $A \to \{\} = (\textit{if } A{=}\{\} \textit{ then } \textit{UNIV} \textit{ else } \{\})$
by *auto*

lemma *Pi-eq-empty*[*simp*]: $(\Pi\ x \in A.\ B\ x) = \{\} \longleftrightarrow (\exists\, x{\in}A.\ B\ x = \{\})$
proof −
have $\exists\, x{\in}A.\ B\ x = \{\}$ **if** $\bigwedge f.\ \exists\, y.\ y \in A \wedge f\ y \notin B\ y$
using *that* [*of* $\lambda u.$ *SOME* $y.\ y \in B\ u$] *some-in-eq* **by** *blast*
then show *?thesis*
by *force*
qed

lemma *Pi-empty* [*simp*]: *Pi* $\{\}$ B = *UNIV*
by (*simp add*: *Pi-def*)

lemma *Pi-Int*: *Pi* $I\ E \cap$ *Pi* $I\ F = (\Pi\ i{\in}I.\ E\ i \cap F\ i)$
by *auto*

lemma *Pi-UN*:
fixes A :: *nat* ⇒ $'i$ ⇒ $'a$ *set*
assumes *finite* I
and *mono*: $\bigwedge i\ n\ m.\ i \in I \Longrightarrow n \le m \Longrightarrow A\ n\ i \subseteq A\ m\ i$
shows $(\bigcup n.\ \textit{Pi}\ I\ (A\ n)) = (\Pi\ i{\in}I.\ \bigcup n.\ A\ n\ i)$
proof (*intro set-eqI iffI*)
fix f
assume $f \in (\Pi\ i{\in}I.\ \bigcup n.\ A\ n\ i)$
then have $\forall\, i{\in}I.\ \exists\, n.\ f\ i \in A\ n\ i$
by *auto*
from *bchoice*[*OF this*] **obtain** n **where** n: $f\ i \in A\ (n\ i)\ i$ **if** $i \in I$ **for** i
by *auto*
obtain k **where** k: $n\ i \le k$ **if** $i \in I$ **for** i
using ‹*finite* I› *finite-nat-set-iff-bounded-le*[*of* n‘I] **by** *auto*
have $f \in \textit{Pi}\ I\ (A\ k)$
proof (*intro Pi-I*)
fix i
assume $i \in I$
from *mono*[*OF this*, *of* $n\ i\ k$] k[*OF this*] n[*OF this*]
show $f\ i \in A\ k\ i$ **by** *auto*

qed
then show $f \in (\bigcup n.\ Pi\ I\ (A\ n))$
by *auto*
qed *auto*

lemma *Pi-UNIV* [*simp*]: $A \to UNIV = UNIV$
by (*simp add*: *Pi-def*)

Covariance of Pi-sets in their second argument

lemma *Pi-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi\ A\ B \subseteq Pi\ A\ C$
by *auto*

Contravariance of Pi-sets in their first argument

lemma *Pi-anti-mono*: $A' \subseteq A \Longrightarrow Pi\ A\ B \subseteq Pi\ A'\ B$
by *auto*

lemma *prod-final*:
assumes *1*: $fst \circ f \in Pi\ A\ B$
and *2*: $snd \circ f \in Pi\ A\ C$
shows $f \in (\Pi\ z \in A.\ B\ z \times C\ z)$
proof (*rule Pi-I*)
fix z
assume z: $z \in A$
have $f\ z = (fst\ (f\ z),\ snd\ (f\ z))$
by *simp*
also have $\ldots \in B\ z \times C\ z$
by (*metis SigmaI PiE o-apply 1 2 z*)
finally show $f\ z \in B\ z \times C\ z$ **.**
qed

lemma *Pi-split-domain*[*simp*]: $x \in Pi\ (I \cup J)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x \in Pi\ J\ X$
by (*auto simp*: *Pi-def*)

lemma *Pi-split-insert-domain*[*simp*]: $x \in Pi\ (insert\ i\ I)\ X \longleftrightarrow x \in Pi\ I\ X \wedge x\ i \in X\ i$
by (*auto simp*: *Pi-def*)

lemma *Pi-cancel-fupd-range*[*simp*]: $i \notin I \Longrightarrow x \in Pi\ I\ (B(i := b)) \longleftrightarrow x \in Pi\ I\ B$
by (*auto simp*: *Pi-def*)

lemma *Pi-cancel-fupd*[*simp*]: $i \notin I \Longrightarrow x(i := a) \in Pi\ I\ B \longleftrightarrow x \in Pi\ I\ B$
by (*auto simp*: *Pi-def*)

lemma *Pi-fupd-iff*: $i \in I \Longrightarrow f \in Pi\ I\ (B(i := A)) \longleftrightarrow f \in Pi\ (I - \{i\})\ B \wedge f\ i \in A$
using *mk-disjoint-insert* **by** *fastforce*

lemma *fst-Pi*: $fst \in A \times B \to A$ **and** *snd-Pi*: $snd \in A \times B \to B$
by *auto*

27.2 Composition With a Restricted Domain: *compose*

lemma *funcset-compose*: $f \in A \to B \Longrightarrow g \in B \to C \Longrightarrow \mathit{compose}\ A\ g\ f \in A \to C$
by (*simp add*: *Pi-def compose-def restrict-def*)

lemma *compose-assoc*:
assumes $f \in A \to B$
shows $\mathit{compose}\ A\ h\ (\mathit{compose}\ A\ g\ f) = \mathit{compose}\ A\ (\mathit{compose}\ B\ h\ g)\ f$
using *assms* **by** (*simp add*: *fun-eq-iff Pi-def compose-def restrict-def*)

lemma *compose-eq*: $x \in A \Longrightarrow \mathit{compose}\ A\ g\ f\ x = g\ (f\ x)$
by (*simp add*: *compose-def restrict-def*)

lemma *surj-compose*: $f\ `\ A = B \Longrightarrow g\ `\ B = C \Longrightarrow \mathit{compose}\ A\ g\ f\ `\ A = C$
by (*auto simp add*: *image-def compose-eq*)

27.3 Bounded Abstraction: *restrict*

lemma *restrict-cong*: $I = J \Longrightarrow (\bigwedge i.\ i \in J =simp=> f\ i = g\ i) \Longrightarrow \mathit{restrict}\ f\ I = \mathit{restrict}\ g\ J$
by (*auto simp*: *restrict-def fun-eq-iff simp-implies-def*)

lemma *restrictI*[*intro*!]: $(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) \in \mathit{Pi}\ A\ B$
by (*simp add*: *Pi-def restrict-def*)

lemma *restrict-apply*[*simp*]: $(\lambda y{\in}A.\ f\ y)\ x = (\mathit{if}\ x \in A\ \mathit{then}\ f\ x\ \mathit{else}\ \mathit{undefined})$
by (*simp add*: *restrict-def*)

lemma *restrict-apply'*: $x \in A \Longrightarrow (\lambda y{\in}A.\ f\ y)\ x = f\ x$
by *simp*

lemma *restrict-ext*: $(\bigwedge x.\ x \in A \Longrightarrow f\ x = g\ x) \Longrightarrow (\lambda x{\in}A.\ f\ x) = (\lambda x{\in}A.\ g\ x)$
by (*simp add*: *fun-eq-iff Pi-def restrict-def*)

lemma *restrict-UNIV*: $\mathit{restrict}\ f\ \mathit{UNIV} = f$
by (*simp add*: *restrict-def*)

lemma *inj-on-restrict-eq* [*simp*]: $\mathit{inj\text{-}on}\ (\mathit{restrict}\ f\ A)\ A \longleftrightarrow \mathit{inj\text{-}on}\ f\ A$
by (*simp add*: *inj-on-def restrict-def*)

lemma *inj-on-restrict-iff*: $A \subseteq B \Longrightarrow \mathit{inj\text{-}on}\ (\mathit{restrict}\ f\ B)\ A \longleftrightarrow \mathit{inj\text{-}on}\ f\ A$
by (*metis inj-on-cong restrict-def subset-iff*)

lemma *Id-compose*: $f \in A \to B \Longrightarrow f \in \mathit{extensional}\ A \Longrightarrow \mathit{compose}\ A\ (\lambda y{\in}B.\ y)\ f = f$
by (*auto simp add*: *fun-eq-iff compose-def extensional-def Pi-def*)

lemma *compose-Id*: $g \in A \to B \Longrightarrow g \in \mathit{extensional}\ A \Longrightarrow \mathit{compose}\ A\ g\ (\lambda x{\in}A.\ x) = g$

by (*auto simp add*: *fun-eq-iff compose-def extensional-def Pi-def*)

lemma *image-restrict-eq* [*simp*]: (*restrict f A*) ‘ *A* = *f* ‘ *A*
by (*auto simp add*: *restrict-def*)

lemma *restrict-restrict*[*simp*]: *restrict* (*restrict f A*) *B* = *restrict f* (*A* ∩ *B*)
unfolding *restrict-def* **by** (*simp add*: *fun-eq-iff*)

lemma *restrict-fupd*[*simp*]: *i* ∉ *I* ⟹ *restrict* (*f* (*i* := *x*)) *I* = *restrict f I*
by (*auto simp*: *restrict-def*)

lemma *restrict-upd*[*simp*]: *i* ∉ *I* ⟹ (*restrict f I*)(*i* := *y*) = *restrict* (*f*(*i* := *y*)) (*insert i I*)
by (*auto simp*: *fun-eq-iff*)

lemma *restrict-Pi-cancel*: *restrict x I* ∈ *Pi I A* ⟷ *x* ∈ *Pi I A*
by (*auto simp*: *restrict-def Pi-def*)

lemma *sum-restrict′* [*simp*]: *sum′* (λ*i*∈*I*. *g i*) *I* = *sum′* (λ*i*. *g i*) *I*
by (*simp add*: *sum.G-def conj-commute cong*: *conj-cong*)

lemma *prod-restrict′* [*simp*]: *prod′* (λ*i*∈*I*. *g i*) *I* = *prod′* (λ*i*. *g i*) *I*
by (*simp add*: *prod.G-def conj-commute cong*: *conj-cong*)

27.4 Bijections Between Sets

The definition of *bij-betw* is in *Fun.thy*, but most of the theorems belong here, or need at least *Hilbert-Choice*.

lemma *bij-betwI*:
assumes *f* ∈ *A* → *B*
and *g* ∈ *B* → *A*
and *g-f*: ⋀*x*. *x*∈*A* ⟹ *g* (*f x*) = *x*
and *f-g*: ⋀*y*. *y*∈*B* ⟹ *f* (*g y*) = *y*
shows *bij-betw f A B*
unfolding *bij-betw-def*
proof
show *inj-on f A*
by (*metis g-f inj-on-def*)
have *f* ‘ *A* ⊆ *B*
using ‹*f* ∈ *A* → *B*› **by** *auto*
moreover
have *B* ⊆ *f* ‘ *A*
by *auto* (*metis Pi-mem* ‹*g* ∈ *B* → *A*› *f-g image-iff*)
ultimately show *f* ‘ *A* = *B*
by *blast*
qed

lemma *bij-betw-imp-funcset*: *bij-betw f A B* ⟹ *f* ∈ *A* → *B*
by (*auto simp add*: *bij-betw-def*)

lemma *inj-on-compose*: *bij-betw f A B* $\Longrightarrow$ *inj-on g B* $\Longrightarrow$ *inj-on* (*compose A g f*) *A*
by (*auto simp add*: *bij-betw-def inj-on-def compose-eq*)

lemma *bij-betw-compose*: *bij-betw f A B* $\Longrightarrow$ *bij-betw g B C* $\Longrightarrow$ *bij-betw* (*compose A g f*) *A C*
by (*simp add*: *bij-betw-def inj-on-compose surj-compose*)

lemma *bij-betw-restrict-eq* [*simp*]: *bij-betw* (*restrict f A*) *A B* = *bij-betw f A B*
by (*simp add*: *bij-betw-def*)

27.5 Extensionality

lemma *extensional-empty*[*simp*]: *extensional* {} = {λx. *undefined*}
unfolding *extensional-def* **by** *auto*

lemma *extensional-arb*: *f* $\in$ *extensional A* $\Longrightarrow$ *x* $\notin$ *A* $\Longrightarrow$ *f x* = *undefined*
by (*simp add*: *extensional-def*)

lemma *restrict-extensional* [*simp*]: *restrict f A* $\in$ *extensional A*
by (*simp add*: *restrict-def extensional-def*)

lemma *compose-extensional* [*simp*]: *compose A f g* $\in$ *extensional A*
by (*simp add*: *compose-def*)

lemma *extensionalityI*:
assumes *f* $\in$ *extensional A*
and *g* $\in$ *extensional A*
and $\bigwedge x$. *x* $\in$ *A* $\Longrightarrow$ *f x* = *g x*
shows *f* = *g*
using *assms* **by** (*force simp add*: *fun-eq-iff extensional-def*)

lemma *extensional-restrict*: *f* $\in$ *extensional A* $\Longrightarrow$ *restrict f A* = *f*
by (*rule extensionalityI*[*OF restrict-extensional*]) *auto*

lemma *extensional-subset*: *f* $\in$ *extensional A* $\Longrightarrow$ *A* $\subseteq$ *B* $\Longrightarrow$ *f* $\in$ *extensional B*
unfolding *extensional-def* **by** *auto*

lemma *inv-into-funcset*: *f* ' *A* = *B* $\Longrightarrow$ ($\lambda x \in B$. *inv-into A f x*) $\in$ *B* $\to$ *A*
by (*unfold inv-into-def*) (*fast intro*: *someI2*)

lemma *compose-inv-into-id*: *bij-betw f A B* $\Longrightarrow$ *compose A* ($\lambda y \in B$. *inv-into A f y*) *f* = ($\lambda x \in A$. *x*)
by (*smt* (*verit*, *best*) *bij-betwE bij-betw-inv-into-left compose-def restrict-apply' restrict-ext*)

lemma *compose-id-inv-into*: *f* ' *A* = *B* $\Longrightarrow$ *compose B f* ($\lambda y \in B$. *inv-into A f y*) = ($\lambda x \in B$. *x*)

by (*smt* (*verit*, *best*) *compose-def f-inv-into-f restrict-apply' restrict-ext*)

lemma *extensional-insert*[*intro*, *simp*]:
assumes $a \in$ *extensional* (*insert i I*)
shows $a(i := b) \in$ *extensional* (*insert i I*)
using *assms* **unfolding** *extensional-def* **by** *auto*

lemma *extensional-Int*[*simp*]: *extensional I* $\cap$ *extensional I'* = *extensional* (*I* $\cap$ *I'*)
unfolding *extensional-def* **by** *auto*

lemma *extensional-UNIV*[*simp*]: *extensional UNIV* = *UNIV*
by (*auto simp*: *extensional-def*)

lemma *restrict-extensional-sub*[*intro*]: *A* $\subseteq$ *B* $\Longrightarrow$ *restrict f A* $\in$ *extensional B*
unfolding *restrict-def extensional-def* **by** *auto*

lemma *extensional-insert-undefined*[*intro*, *simp*]:
a $\in$ *extensional* (*insert i I*) $\Longrightarrow$ *a*(*i* := *undefined*) $\in$ *extensional I*
unfolding *extensional-def* **by** *auto*

lemma *extensional-insert-cancel*[*intro*, *simp*]:
a $\in$ *extensional I* $\Longrightarrow$ *a* $\in$ *extensional* (*insert i I*)
unfolding *extensional-def* **by** *auto*

27.6 Cardinality

lemma *card-inj*: *f* $\in$ *A* $\to$ *B* $\Longrightarrow$ *inj-on f A* $\Longrightarrow$ *finite B* $\Longrightarrow$ *card A* $\leq$ *card B*
by (*rule card-inj-on-le*) *auto*

lemma *card-bij*:
assumes *f* $\in$ *A* $\to$ *B inj-on f A*
and *g* $\in$ *B* $\to$ *A inj-on g B*
and *finite A finite B*
shows *card A* = *card B*
using *assms* **by** (*blast intro*: *card-inj order-antisym*)

27.7 Extensional Function Spaces

definition *PiE* :: *'a set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b set*) $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) *set*
where *PiE S T* = *Pi S T* $\cap$ *extensional S*

abbreviation $Pi_E\ A\ B \equiv PiE\ A\ B$

syntax
-PiE :: *pttrn* $\Rightarrow$ *'a set* $\Rightarrow$ *'b set* $\Rightarrow$ (*'a* $\Rightarrow$ *'b*) *set*
(‹(‹*indent=3 notation=‹binder* Π_E∈››Π_E *-*∈*-./ -*)› *10*)
syntax-consts
-PiE $\rightleftharpoons$ Pi_E
translations

$\Pi_E\ x{\in}A.\ B \rightleftharpoons CONST\ Pi_E\ A\ (\lambda x.\ B)$

abbreviation *extensional-funcset* :: *'a set* ⇒ *'b set* ⇒ (*'a* ⇒ *'b*) *set* (**infixr** ‹$\rightarrow_E$› *60*)
where $A \rightarrow_E B \equiv (\Pi_E\ i{\in}A.\ B)$

lemma *extensional-funcset-def*: *extensional-funcset S T* = (*S* → *T*) ∩ *extensional S*
by (*simp add*: *PiE-def*)

lemma *PiE-empty-domain*[*simp*]: $Pi_E\ \{\}\ T = \{\lambda x.\ undefined\}$
unfolding *PiE-def* **by** *simp*

lemma *PiE-UNIV-domain*: $Pi_E\ UNIV\ T = Pi\ UNIV\ T$
unfolding *PiE-def* **by** *simp*

lemma *PiE-empty-range*[*simp*]: *i* ∈ *I* ⟹ *F i* = {} ⟹ (Π$_E$ *i*∈*I*. *F i*) = {}
unfolding *PiE-def* **by** *auto*

lemma *PiE-eq-empty-iff*: $Pi_E\ I\ F = \{\} \longleftrightarrow (\exists\, i{\in}I.\ F\ i = \{\})$
proof
assume $Pi_E\ I\ F = \{\}$
show ∃ *i*∈*I*. *F i* = {}
proof (*rule ccontr*)
assume ¬ *?thesis*
then have ∀ *i*. ∃ *y*. (*i* ∈ *I* ⟶ *y* ∈ *F i*) ∧ (*i* ∉ *I* ⟶ *y* = *undefined*)
by *auto*
from *choice*[*OF this*]
obtain *f* **where** ∀ *x*. (*x* ∈ *I* ⟶ *f x* ∈ *F x*) ∧ (*x* ∉ *I* ⟶ *f x* = *undefined*) **..**
then have *f* ∈ $Pi_E\ I\ F$
by (*auto simp*: *extensional-def PiE-def*)
with ‹$Pi_E\ I\ F = \{\}$› **show** *False*
by *auto*
qed
qed (*auto simp*: *PiE-def*)

lemma *PiE-arb*: *f* ∈ $Pi_E\ S\ T$ ⟹ *x* ∉ *S* ⟹ *f x* = *undefined*
unfolding *PiE-def* **by** *auto* (*auto dest*!: *extensional-arb*)

lemma *PiE-mem*: *f* ∈ $Pi_E\ S\ T$ ⟹ *x* ∈ *S* ⟹ *f x* ∈ *T x*
unfolding *PiE-def* **by** *auto*

lemma *PiE-fun-upd*: *y* ∈ *T x* ⟹ *f* ∈ $Pi_E\ S\ T$ ⟹ *f*(*x* := *y*) ∈ Pi_E (*insert x S*) *T*
unfolding *PiE-def extensional-def* **by** *auto*

lemma *fun-upd-in-PiE*: *x* ∉ *S* ⟹ *f* ∈ Pi_E (*insert x S*) *T* ⟹ *f*(*x* := *undefined*) ∈ $Pi_E\ S\ T$
unfolding *PiE-def extensional-def* **by** *auto*

lemma *PiE-insert-eq*: $Pi_E\ (insert\ x\ S)\ T = (\lambda(y,\ g).\ g(x := y))\ `\ (T\ x \times Pi_E\ S\ T)$
proof −
have $f \in (\lambda(y,\ g).\ g(x := y))\ `\ (T\ x \times Pi_E\ S\ T)$ **if** $f \in Pi_E\ (insert\ x\ S)\ T\ x \notin S$ **for** f
using *that*
by (*auto intro*!: *image-eqI*[**where** $x=(f\ x, f(x := undefined))$] *intro*: *fun-upd-in-PiE PiE-mem*)
moreover
have $f \in (\lambda(y,\ g).\ g(x := y))\ `\ (T\ x \times Pi_E\ S\ T)$ **if** $f \in Pi_E\ (insert\ x\ S)\ T\ x \in S$ **for** f
using *that*
by (*auto intro*!: *image-eqI*[**where** $x=(f\ x,\ f)$] *intro*: *fun-upd-in-PiE PiE-mem simp*: *insert-absorb*)
ultimately show *?thesis*
by (*auto intro*: *PiE-fun-upd*)
qed

lemma *PiE-Int*: $Pi_E\ I\ A \cap Pi_E\ I\ B = Pi_E\ I\ (\lambda x.\ A\ x \cap B\ x)$
by (*auto simp*: *PiE-def*)

lemma *PiE-cong*: $(\bigwedge i.\ i{\in}I \Longrightarrow A\ i = B\ i) \Longrightarrow Pi_E\ I\ A = Pi_E\ I\ B$
unfolding *PiE-def* **by** (*auto simp*: *Pi-cong*)

lemma *PiE-E* [*elim*]:
assumes $f \in Pi_E\ A\ B$
obtains $x \in A$ **and** $f\ x \in B\ x$
| $x \notin A$ **and** $f\ x = undefined$
using *assms* **by** (*auto simp*: *Pi-def PiE-def extensional-def*)

lemma *PiE-I*[*intro*!]:
$(\bigwedge x.\ x \in A \Longrightarrow f\ x \in B\ x) \Longrightarrow (\bigwedge x.\ x \notin A \Longrightarrow f\ x = undefined) \Longrightarrow f \in Pi_E\ A\ B$
by (*simp add*: *PiE-def extensional-def*)

lemma *PiE-mono*: $(\bigwedge x.\ x \in A \Longrightarrow B\ x \subseteq C\ x) \Longrightarrow Pi_E\ A\ B \subseteq Pi_E\ A\ C$
by *auto*

lemma *PiE-iff*: $f \in Pi_E\ I\ X \longleftrightarrow (\forall\, i{\in}I.\ f\ i \in X\ i) \wedge f \in extensional\ I$
by (*simp add*: *PiE-def Pi-iff*)

lemma *restrict-PiE-iff*: $restrict\ f\ I \in Pi_E\ I\ X \longleftrightarrow (\forall\, i \in I.\ f\ i \in X\ i)$
by (*simp add*: *PiE-iff*)

lemma *ext-funcset-to-sing-iff* [*simp*]: $A \to_E \{a\} = \{\lambda x{\in}A.\ a\}$
by (*auto simp*: *PiE-def Pi-iff extensionalityI*)

lemma *PiE-restrict*[*simp*]: $f \in Pi_E\ A\ B \Longrightarrow restrict\ f\ A = f$

by (*simp add*: *extensional-restrict PiE-def*)

lemma *restrict-PiE*[*simp*]: *restrict f I* $\in$ Pi_E *I S* $\longleftrightarrow$ *f* $\in$ *Pi I S*
by (*auto simp*: *PiE-iff*)

lemma *PiE-eq-subset*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
and *eq*: $Pi_E\ I\ F = Pi_E\ I\ F'$
and $i \in I$
shows $F\ i \subseteq F'\ i$
proof
fix *x*
assume $x \in F\ i$
with *ne* **have** $\forall j.\ \exists y.\ (j \in I \longrightarrow y \in F\ j \wedge (i = j \longrightarrow x = y)) \wedge (j \notin I \longrightarrow y = \mathit{undefined})$
by *auto*
from *choice*[*OF this*] **obtain** *f*
where *f*: $\forall j.\ (j \in I \longrightarrow f\ j \in F\ j \wedge (i = j \longrightarrow x = f\ j)) \wedge (j \notin I \longrightarrow f\ j = \mathit{undefined})$ **..**
then have $f \in Pi_E\ I\ F$
by (*auto simp*: *extensional-def PiE-def*)
then have $f \in Pi_E\ I\ F'$
using *assms* **by** *simp*
then show $x \in F'\ i$
using *f* ‹$i \in I$› **by** (*auto simp*: *PiE-def*)
qed

lemma *PiE-eq-iff-not-empty*:
assumes *ne*: $\bigwedge i.\ i \in I \Longrightarrow F\ i \neq \{\}$ $\bigwedge i.\ i \in I \Longrightarrow F'\ i \neq \{\}$
shows $Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i)$
proof (*intro iffI ballI*)
fix *i*
assume *eq*: $Pi_E\ I\ F = Pi_E\ I\ F'$
assume *i*: $i \in I$
show $F\ i = F'\ i$
using *PiE-eq-subset*[*of I F F'*, *OF ne eq i*]
using *PiE-eq-subset*[*of I F' F*, *OF ne*(*2*,*1*) *eq*[*symmetric*] *i*]
by *auto*
qed (*auto simp*: *PiE-def*)

lemma *PiE-eq-iff*: $Pi_E\ I\ F = Pi_E\ I\ F' \longleftrightarrow (\forall i{\in}I.\ F\ i = F'\ i) \vee ((\exists i{\in}I.\ F\ i = \{\}) \wedge (\exists i{\in}I.\ F'\ i = \{\}))$
proof (*intro iffI disjCI*)
assume *eq*[*simp*]: $Pi_E\ I\ F = Pi_E\ I\ F'$
assume $\neg\ ((\exists i{\in}I.\ F\ i = \{\}) \wedge (\exists i{\in}I.\ F'\ i = \{\}))$
then have $(\forall i{\in}I.\ F\ i \neq \{\}) \wedge (\forall i{\in}I.\ F'\ i \neq \{\})$
using *PiE-eq-empty-iff*[*of I F*] *PiE-eq-empty-iff*[*of I F'*] **by** *auto*
with *PiE-eq-iff-not-empty*[*of I F F'*] **show** $\forall i{\in}I.\ F\ i = F'\ i$
by *auto*

next
assume $(\forall i{\in}I.\ F\ i = F'\ i) \vee (\exists i{\in}I.\ F\ i = \{\}) \wedge (\exists i{\in}I.\ F'\ i = \{\})$
then show $Pi_E\ I\ F = Pi_E\ I\ F'$
using *PiE-eq-empty-iff*[*of I F*] *PiE-eq-empty-iff*[*of I F'*] **by** (*auto simp*: *PiE-def*)
qed

lemma *extensional-funcset-fun-upd-restricts-rangeI*:
$\forall y \in S.\ f\ x \neq f\ y \Longrightarrow f \in (insert\ x\ S) \rightarrow_E T \Longrightarrow f(x := undefined) \in S \rightarrow_E (T - \{f\ x\})$
unfolding *extensional-funcset-def extensional-def*
by (*auto split*: *if-split-asm*)

lemma *extensional-funcset-fun-upd-extends-rangeI*:
assumes $a \in T\ f \in S \rightarrow_E (T - \{a\})$
shows $f(x := a) \in insert\ x\ S \rightarrow_E T$
using *assms* **unfolding** *extensional-funcset-def extensional-def* **by** *auto*

lemma *subset-PiE*:
$PiE\ I\ S \subseteq PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \vee (\forall i \in I.\ S\ i \subseteq T\ i)$ (**is** *?lhs* $\longleftrightarrow$ - $\vee$ *?rhs*)
proof (*cases PiE I S* $= \{\}$)
case *False*
moreover have *?lhs* $=$ *?rhs*
proof
assume *L*: *?lhs*
have $\bigwedge i.\ i{\in}I \Longrightarrow S\ i \neq \{\}$
using *False PiE-eq-empty-iff* **by** *blast*
with *L* **show** *?rhs*
by (*simp add*: *PiE-Int PiE-eq-iff inf.absorb-iff2*)
qed *auto*
ultimately show *?thesis*
by *simp*
qed *simp*

lemma *PiE-eq*: $PiE\ I\ S = PiE\ I\ T \longleftrightarrow PiE\ I\ S = \{\} \wedge PiE\ I\ T = \{\} \vee (\forall i \in I.\ S\ i = T\ i)$
by (*auto simp*: *PiE-eq-iff PiE-eq-empty-iff*)

lemma *PiE-UNIV* [*simp*]: $PiE\ UNIV\ (\lambda i.\ UNIV) = UNIV$
by *blast*

lemma *image-projection-PiE*:
$(\lambda f.\ f\ i)\ `\ (PiE\ I\ S) = (if\ PiE\ I\ S = \{\}\ then\ \{\}\ else\ if\ i \in I\ then\ S\ i\ else\ \{undefined\})$
proof −
have $(\lambda f.\ f\ i)\ `\ Pi_E\ I\ S = S\ i$ **if** $i \in I\ f \in PiE\ I\ S$ **for** f
proof −
have $x \in S\ i \Longrightarrow \exists f{\in}Pi_E\ I\ S.\ x = f\ i$ **for** x
using *that*

```
    by (force intro: bexI [where x=λk. if k=i then x else f k])
  then show ?thesis
    using that by force
 qed
 then show ?thesis
  by (smt (verit) PiE-arb equals0I image-cong image-constant image-empty)
qed

lemma PiE-singleton:
 assumes f ∈ extensional A
 shows PiE A (λx. {f x}) = {f}
proof −
 have g = f if g ∈ PiE A (λx. {f x}) for g
 proof −
  from that have g x = f x for x
   using assms by (cases x ∈ A) (auto simp: extensional-def)
  then show ?thesis by (simp add: fun-eq-iff)
 qed
 with assms show ?thesis
  by (auto simp: extensional-def)
qed

lemma PiE-eq-singleton: (Π_E i∈I. S i) = {λi∈I. f i} ⟷ (∀ i∈I. S i = {f i})
 by (metis (mono-tags, lifting) PiE-eq PiE-singleton insert-not-empty restrict-apply'
restrict-extensional)

lemma PiE-over-singleton-iff: (Π_E x∈{a}. B x) = (⋃ b ∈ B a. {λx ∈ {a}. b})
proof −
 have ∃ xa∈B a. x = (λx∈{a}. xa) if x a ∈ B a and x ∈ extensional {a} for x
  using that PiE-singleton by fastforce
 then show ?thesis
  by (auto simp: PiE-iff split: if-split-asm)
qed

lemma all-PiE-elements:
 (∀ z ∈ PiE I S. ∀ i ∈ I. P i (z i)) ⟷ PiE I S = {} ∨ (∀ i ∈ I. ∀ x ∈ S i. P i x)
 (is ?lhs = ?rhs)
proof (cases PiE I S = {})
 case False
 then obtain f where f: ⋀i. i ∈ I ⟹ f i ∈ S i
  by fastforce
 show ?thesis
 proof
  assume L: ?lhs
  have P i x
   if i ∈ I x ∈ S i for i x
  proof −
   have (λj ∈ I. if j=i then x else f j) ∈ PiE I S
    by (simp add: f that(2))
```

then have $P\ i\ ((\lambda j \in I.\ \mathit{if}\ j{=}i\ \mathit{then}\ x\ \mathit{else}\ f\ j)\ i)$
using *L that* **by** *blast*
with *that* **show** *?thesis*
by *simp*
qed
then show *?rhs*
by (*simp add*: *False*)
qed *fastforce*
qed *simp*

lemma *PiE-ext*: $[\![x \in \mathit{PiE}\ k\ s;\ y \in \mathit{PiE}\ k\ s;\ \bigwedge i.\ i \in k \Longrightarrow x\ i = y\ i]\!] \Longrightarrow x = y$
by (*metis ext PiE-E*)

27.7.1 Injective Extensional Function Spaces

lemma *extensional-funcset-fun-upd-inj-onI*:
assumes $f \in S \to_E (T - \{a\})$
and *inj-on f S*
shows *inj-on* $(f(x := a))\ S$
using *assms*
unfolding *extensional-funcset-def* **by** (*auto intro*!: *inj-on-fun-updI*)

lemma *extensional-funcset-extend-domain-inj-on-eq*:
assumes $x \notin S$
shows $\{f.\ f \in (\mathit{insert}\ x\ S) \to_E T \wedge \mathit{inj\text{-}on}\ f\ (\mathit{insert}\ x\ S)\} =$
$(\lambda(y,\ g).\ g(x{:=}y))\ `\ \{(y,\ g).\ y \in T \wedge g \in S \to_E (T - \{y\}) \wedge \mathit{inj\text{-}on}\ g\ S\}$
proof −
have *False* **if** $f \in S \to_E T - \{a\}$ **and** $a = (\mathit{if}\ y = x\ \mathit{then}\ a\ \mathit{else}\ f\ y)$ **and** $y \in S$ **for** *a f y*
using *assms that* **by** (*auto dest*!: *PiE-mem split*: *if-split-asm*)
moreover
have $\exists b.\ b \in S \to_E T - \{f\ x\} \wedge \mathit{inj\text{-}on}\ b\ S \wedge f = b(x := f\ x)$
if $f \in \mathit{insert}\ x\ S \to_E T$ **and** *inj-on f S* **and** $\forall xb{\in}S.\ f\ x \neq f\ xb$ **for** *f*
using *that*
unfolding *inj-on-def*
by (*smt* (*verit*, *ccfv-threshold*) *PiE-restrict fun-upd-apply fun-upd-triv insert-Diff insert-iff*
restrict-PiE-iff restrict-upd)
ultimately show *?thesis*
using *assms*
apply (*auto simp*: *image-iff intro*: *extensional-funcset-fun-upd-inj-onI*
extensional-funcset-fun-upd-extends-rangeI del: *PiE-I PiE-E*)
apply (*smt* (*verit*, *best*) *PiE-cong PiE-mem inj-on-def insertCI*)
apply *blast*
done
qed

lemma *extensional-funcset-extend-domain-inj-onI*:
assumes $x \notin S$

shows *inj-on* $(\lambda(y,\ g).\ g(x := y))$ $\{(y,\ g).\ y \in T \wedge g \in S \to_E (T - \{y\}) \wedge$ *inj-on g S*$\}$
using *assms*
by (*simp add*: *inj-on-def*) (*metis PiE-restrict fun-upd-apply restrict-fupd*)

27.7.2 Misc properties of functions, composition and restriction from HOL Light

lemma *function-factors-left-gen*:
$(\forall x\ y.\ P\ x \wedge P\ y \wedge g\ x = g\ y \longrightarrow f\ x = f\ y) \longleftrightarrow (\exists h.\ \forall x.\ P\ x \longrightarrow f\ x = h(g\ x))$
(**is** *?lhs* = *?rhs*)
proof
assume *L*: *?lhs*
then show *?rhs*
apply (*rule-tac x*=*f* ∘ *inv-into* (*Collect P*) *g* **in** *exI*)
unfolding *o-def*
by (*metis* (*mono-tags*, *opaque-lifting*) *f-inv-into-f imageI inv-into-into mem-Collect-eq*)
qed *auto*

lemma *function-factors-left*: $(\forall x\ y.\ (g\ x = g\ y) \longrightarrow (f\ x = f\ y)) \longleftrightarrow (\exists h.\ f = h \circ g)$
using *function-factors-left-gen* [*of* $\lambda x.$ *True g f*] **unfolding** *o-def* **by** *blast*

lemma *function-factors-right-gen*: $(\forall x.\ P\ x \longrightarrow (\exists y.\ g\ y = f\ x)) \longleftrightarrow (\exists h.\ \forall x.\ P\ x \longrightarrow f\ x = g(h\ x))$
by *metis*

lemma *function-factors-right*: $(\forall x.\ \exists y.\ g\ y = f\ x) \longleftrightarrow (\exists h.\ f = g \circ h)$
unfolding *o-def* **by** *metis*

lemma *restrict-compose-right*: *restrict* $(g \circ$ *restrict f S*$)\ S =$ *restrict* $(g \circ f)\ S$
by *auto*

lemma *restrict-compose-left*: f ‘ $S \subseteq T \Longrightarrow$ *restrict* $($*restrict g T* $\circ f)\ S =$ *restrict* $(g \circ f)\ S$
by *fastforce*

27.7.3 Cardinality

lemma *finite-PiE*: *finite S* $\Longrightarrow (\bigwedge i.\ i \in S \Longrightarrow$ *finite* $(T\ i)) \Longrightarrow$ *finite* $(\Pi_E\ i \in S.\ T\ i)$
by (*induct S arbitrary*: *T rule*: *finite-induct*) (*simp-all add*: *PiE-insert-eq*)

lemma *inj-combinator*: $x \notin S \Longrightarrow$ *inj-on* $(\lambda(y,\ g).\ g(x := y))$ $(T\ x \times Pi_E\ S\ T)$
proof (*safe intro*!: *inj-onI ext*)
fix *f y g z*
assume $x \notin S$
assume *fg*: $f \in Pi_E\ S\ T\ g \in Pi_E\ S\ T$
assume $f(x := y) = g(x := z)$
then have $*$: $\bigwedge i.\ (f(x := y))\ i = (g(x := z))\ i$

```
    unfolding fun-eq-iff by auto
  from this[of x] show y = z by simp
  fix i from *[of i] ‹x ∉ S› fg show f i = g i
    by (auto split: if-split-asm simp: PiE-def extensional-def)
qed

lemma card-PiE: finite S ⟹ card (Π_E i ∈ S. T i) = (∏ i∈S. card (T i))
proof (induct rule: finite-induct)
  case empty
  then show ?case
    by auto
next
  case (insert x S)
  then show ?case
    by (simp add: PiE-insert-eq inj-combinator card-image card-cartesian-product)
qed

lemma card-funcsetE: finite A ⟹ card (A →_E B) = card B ^ card A
  by (subst card-PiE) auto

lemma card-inj-on-subset-funcset:
  assumes finB: finite B
    and finC: finite C
    and AB: A ⊆ B
  shows card {f ∈ B →_E C. inj-on f A} =
    card C^(card B − card A) * prod ((−) (card C)) {0 ..< card A}
proof −
  define D where D = B − A
  from AB have B: B = A ∪ D and disj: A ∩ D = {}
    unfolding D-def by auto
  have sub: card B − card A = card D
    unfolding D-def using finB AB
    by (metis card-Diff-subset finite-subset)
  from finB B have finite A finite D by auto
  then show ?thesis
    unfolding sub unfolding B using disj
  proof (induct A rule: finite-induct)
    case empty
    from card-funcsetE[OF this(1), of C] show ?case
      by auto
  next
    case (insert a A)
    have {f. f ∈ insert a A ∪ D →_E C ∧ inj-on f (insert a A)} =
      {f(a := c) | f c. f ∈ A ∪ D →_E C ∧ inj-on f A ∧ c ∈ C − f ‘ A}
      (is ?l = ?r)
    proof
      show ?r ⊆ ?l
        by (auto intro: inj-on-fun-updI split: if-splits)
      have f ∈ ?r if f: f ∈ ?l for f
```

proof −
let *?g* = *f*(*a* := *undefined*)
let *?h* = *?g*(*a* := *f a*)
have *mem*: *f a* ∈ *C* − *?g* ‘ *A* **using** *insert*(*1*,*2*,*4*,*5*) *f* **by** *auto*
from *f* **have** *f*: *f* ∈ *insert a A* ∪ *D* →$_E$ *C inj-on f* (*insert a A*) **by** *auto*
hence *?g* ∈ *A* ∪ *D* →$_E$ *C inj-on ?g A* **using** ‹*a* ∉ *A*› ‹*insert a A* ∩ *D* = {}›
by (*auto split*: *if-splits simp*: *inj-on-def*)
with *mem* **have** *?h* ∈ *?r* **by** *blast*
also have *?h* = *f* **by** *auto*
finally show *?thesis* **.**
qed
then show *?l* ⊆ *?r* **by** *auto*
qed
also have ... = (λ (*f*, *c*). *f* (*a* := *c*)) ‘
(*Sigma* {*f* . *f* ∈ *A* ∪ *D* →$_E$ *C* ∧ *inj-on f A*} (λ *f*. *C* − *f* ‘ *A*))
by *auto*
also have *card* (...) = *card* (*Sigma* {*f* . *f* ∈ *A* ∪ *D* →$_E$ *C* ∧ *inj-on f A*} (λ *f*. *C* − *f* ‘ *A*))
proof (*rule card-image*, *intro inj-onI*, *clarsimp*, *goal-cases*)
case (*1 f c g d*)
let *?f* = *f*(*a* := *c*, *a* := *undefined*)
let *?g* = *g*(*a* := *d*, *a* := *undefined*)
from *1* **have** *id*: *f*(*a* := *c*) = *g*(*a* := *d*)
by *auto*
from *fun-upd-eqD*[*OF id*]
have *cd*: *c* = *d*
by *auto*
from *id* **have** *?f* = *?g*
by *auto*
also have *?f* = *f* **using** ‹*f* ∈ *A* ∪ *D* →$_E$ *C*› *insert*(*1*,*2*,*4*,*5*)
by (*intro ext*, *auto*)
also have *?g* = *g* **using** ‹*g* ∈ *A* ∪ *D* →$_E$ *C*› *insert*(*1*,*2*,*4*,*5*)
by (*intro ext*, *auto*)
finally show *f* = *g* ∧ *c* = *d*
using *cd* **by** *auto*
qed
also have ... = (∑*f*∈{*f* ∈ *A* ∪ *D* →$_E$ *C*. *inj-on f A*}. *card* (*C* − *f* ‘ *A*))
by (*rule card-SigmaI*, *rule finite-subset*[*of* - *A* ∪ *D* →$_E$ *C*],
insert ‹*finite C*› ‹*finite D*› ‹*finite A*›, *auto intro*!: *finite-PiE*)
also have ... = (∑*f*∈{*f* ∈ *A* ∪ *D* →$_E$ *C*. *inj-on f A*}. *card C* − *card A*)
by (*rule sum.cong*[*OF refl*], *subst card-Diff-subset*, *insert* ‹*finite A*›, *auto simp*: *card-image*)
also have ... = (*card C* − *card A*) ∗ *card* {*f* ∈ *A* ∪ *D* →$_E$ *C*. *inj-on f A*}
by *simp*
also have ... = *card C* ^ *card D* ∗ ((*card C* − *card A*) ∗ *prod* ((−) (*card C*)) {*0*..<*card A*})
using *insert* **by** (*auto simp*: *ac-simps*)
also have (*card C* − *card A*) ∗ *prod* ((−) (*card C*)) {*0*..<*card A*} =
prod ((−) (*card C*)) {*0*..<*Suc* (*card A*)} **by** *simp*

also have *Suc* (*card A*) = *card* (*insert a A*) **using** *insert* **by** *auto*
finally show *?case* **.**
qed
qed

27.8 The pigeonhole principle

An alternative formulation of this is that for a function mapping a finite set A of cardinality m to a finite set B of cardinality n, there exists an element $y \in B$ that is hit at least $\lceil \frac{m}{n} \rceil$ times. However, since we do not have real numbers or rounding yet, we state it in the following equivalent form:

lemma *pigeonhole-card*:
assumes $f \in A \to B$ *finite A finite B* $B \neq \{\}$
shows $\exists\, y{\in}B.$ *card* $(f -` \{y\} \cap A) * card\ B \geq card\ A$
proof −
from *assms* **have** *card* $B > 0$
by *auto*
define M **where** $M = Max\ ((\lambda y.\ card\ (f -` \{y\} \cap A)) `` B)$
have $A = (\bigcup y{\in}B.\ f -` \{y\} \cap A)$
using *assms* **by** *auto*
also have *card* ... $= (\sum i{\in}B.\ card\ (f -` \{i\} \cap A))$
using *assms* **by** (*subst card-UN-disjoint*) *auto*
also have ... $\leq (\sum i{\in}B.\ M)$
unfolding *M-def* **using** *assms* **by** (*intro sum-mono Max.coboundedI*) *auto*
also have ... = *card B* ∗ *M*
by *simp*
finally have ∗: $M * card\ B \geq card\ A$
by (*simp add*: *mult-ac*)
from *assms* **have** $M \in (\lambda y.\ card\ (f -` \{y\} \cap A)) `` B$
unfolding *M-def* **by** (*intro Max-in*) *auto*
with ∗ **show** *?thesis*
by *blast*
qed

27.9 Products of sums

lemma *prod-sum-PiE*:
fixes $f :: 'a \Rightarrow 'b \Rightarrow 'c ::$ *comm-semiring-1*
assumes *finite*: *finite A* **and** *finite*: $\bigwedge x.\ x \in A \Longrightarrow$ *finite* $(B\ x)$
shows $(\prod x{\in}A.\ \sum y{\in}B\ x.\ f\ x\ y) = (\sum g{\in}PiE\ A\ B.\ \prod x{\in}A.\ f\ x\ (g\ x))$
using *assms*
proof (*induction A rule*: *finite-induct*)
case *empty*
thus *?case* **by** *auto*
next
case (*insert x A*)
have $(\sum g{\in}Pi_E\ (insert\ x\ A)\ B.\ \prod x{\in}insert\ x\ A.\ f\ x\ (g\ x)) =$
$(\sum g{\in}Pi_E\ (insert\ x\ A)\ B.\ f\ x\ (g\ x) * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
using *insert* **by** *simp*

also have $(\lambda g.\ \prod x'{\in}A.\ f\ x'\ (g\ x')) = (\lambda g.\ \prod x'{\in}A.\ f\ x'\ (\mathit{if}\ x' = x\ \mathit{then}\ \mathit{undefined}\ \mathit{else}\ g\ x'))$
using *insert* **by** (*intro ext prod.cong*) *auto*
also have $(\sum g{\in}Pi_E\ (\mathit{insert}\ x\ A)\ B.\ f\ x\ (g\ x) * \ldots\ g) =$
$(\sum (y,g){\in}B\ x \times Pi_E\ A\ B.\ f\ x\ y * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
using *insert.prems insert.hyps*
by (*intro sum.reindex-bij-witness*[*of* - $\lambda(y,g).\ g(x := y)$ $\lambda g.\ (g\ x,\ g(x := \mathit{undefined}))$])
(*auto simp*: *PiE-def extensional-def*)
also have $\ldots = (\sum y{\in}B\ x.\ \sum g{\in}Pi_E\ A\ B.\ f\ x\ y * (\prod x'{\in}A.\ f\ x'\ (g\ x')))$
by (*subst sum.cartesian-product*) *auto*
also have $\ldots = (\sum y{\in}B\ x.\ f\ x\ y) * (\sum g{\in}Pi_E\ A\ B.\ \prod x'{\in}A.\ f\ x'\ (g\ x'))$
using *insert* **by** (*subst sum.swap*) (*simp add*: *sum-distrib-left sum-distrib-right*)
also have $(\sum g{\in}Pi_E\ A\ B.\ \prod x'{\in}A.\ f\ x'\ (g\ x')) = (\prod x{\in}A.\ \sum y{\in}B\ x.\ f\ x\ y)$
using *insert.prems* **by** (*intro insert.IH* [*symmetric*]) *auto*
also have $(\sum y{\in}B\ x.\ f\ x\ y) * \ldots = (\prod x{\in}\mathit{insert}\ x\ A.\ \sum y{\in}B\ x.\ f\ x\ y)$
using *insert.hyps* **by** *simp*
finally show *?case* **..**
qed

end

28 Partitions and Disjoint Sets

theory *Disjoint-Sets*
imports *FuncSet*
begin

lemma *mono-imp-UN-eq-last*: *mono A* $\Longrightarrow (\bigcup i{\leq}n.\ A\ i) = A\ n$
unfolding *mono-def* **by** *auto*

28.1 Set of Disjoint Sets

abbreviation *disjoint* :: *'a set set* $\Rightarrow$ *bool* **where** *disjoint* $\equiv$ *pairwise disjnt*

lemma *disjoint-def*: *disjoint A* $\longleftrightarrow (\forall\, a{\in}A.\ \forall\, b{\in}A.\ a \neq b \longrightarrow a \cap b = \{\})$
unfolding *pairwise-def disjnt-def* **by** *auto*

lemma *disjointI*:
$(\bigwedge a\ b.\ a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}) \Longrightarrow$ *disjoint A*
unfolding *disjoint-def* **by** *auto*

lemma *disjointD*:
disjoint A $\Longrightarrow a \in A \Longrightarrow b \in A \Longrightarrow a \neq b \Longrightarrow a \cap b = \{\}$
unfolding *disjoint-def* **by** *auto*

lemma *disjoint-image*: *inj-on f* $(\bigcup A) \Longrightarrow$ *disjoint A* $\Longrightarrow$ *disjoint* $((`)\ f\ `\ A)$
unfolding *inj-on-def disjoint-def* **by** *blast*

lemma assumes *disjoint* $(A \cup B)$
shows *disjoint-unionD1*: *disjoint A* **and** *disjoint-unionD2*: *disjoint B*
using *assms* **by** (*simp-all add*: *disjoint-def*)

lemma *disjoint-INT*:
assumes $*$: $\bigwedge i.\ i \in I \Longrightarrow$ *disjoint* $(F\ i)$
shows *disjoint* $\{\bigcap i{\in}I.\ X\ i \mid X.\ \forall\, i{\in}I.\ X\ i \in F\ i\}$
proof (*safe intro*!: *disjointI del*: *equalityI*)
fix $A\ B :: 'a \Rightarrow 'b\ set$ **assume** $(\bigcap i{\in}I.\ A\ i) \neq (\bigcap i{\in}I.\ B\ i)$
then obtain i **where** $A\ i \neq B\ i\ i \in I$
by *auto*
moreover assume $\forall\, i{\in}I.\ A\ i \in F\ i\ \forall\, i{\in}I.\ B\ i \in F\ i$
ultimately show $(\bigcap i{\in}I.\ A\ i) \cap (\bigcap i{\in}I.\ B\ i) = \{\}$
using $*[OF\ \langle i{\in}I\rangle,\ THEN\ disjointD,\ of\ A\ i\ B\ i]$
by (*auto simp flip*: *INT-Int-distrib*)
qed

lemma *diff-Union-pairwise-disjoint*:
assumes *pairwise disjnt* $\mathcal{A}\ \mathcal{B} \subseteq \mathcal{A}$
shows $\bigcup\mathcal{A} - \bigcup\mathcal{B} = \bigcup(\mathcal{A} - \mathcal{B})$
proof −
have *False*
if x: $x \in A\ x \in B$ **and** AB: $A \in \mathcal{A}\ A \notin \mathcal{B}\ B \in \mathcal{B}$ **for** $x\ A\ B$
proof −
have $A \cap B = \{\}$
using *assms disjointD AB* **by** *blast*
with x **show** *?thesis*
by *blast*
qed
then show *?thesis* **by** *auto*
qed

lemma *Int-Union-pairwise-disjoint*:
assumes *pairwise disjnt* $(\mathcal{A} \cup \mathcal{B})$
shows $\bigcup\mathcal{A} \cap \bigcup\mathcal{B} = \bigcup(\mathcal{A} \cap \mathcal{B})$
proof −
have *False*
if x: $x \in A\ x \in B$ **and** AB: $A \in \mathcal{A}\ A \notin \mathcal{B}\ B \in \mathcal{B}$ **for** $x\ A\ B$
proof −
have $A \cap B = \{\}$
using *assms disjointD AB* **by** *blast*
with x **show** *?thesis*
by *blast*
qed
then show *?thesis* **by** *auto*
qed

lemma *psubset-Union-pairwise-disjoint*:
assumes $\mathcal{B}$: *pairwise disjnt* $\mathcal{B}$ **and** $\mathcal{A} \subset \mathcal{B} - \{\{\}\}$

shows $\bigcup \mathcal{A} \subset \bigcup \mathcal{B}$
unfolding *psubset-eq*
proof
show $\bigcup \mathcal{A} \subseteq \bigcup \mathcal{B}$
using *assms* **by** *blast*
have $\mathcal{A} \subseteq \mathcal{B} \bigcup (\mathcal{B} - \mathcal{A} \cap (\mathcal{B} - \{\{\}\})) \neq \{\}$
using *assms* **by** *blast+*
then show $\bigcup \mathcal{A} \neq \bigcup \mathcal{B}$
using *diff-Union-pairwise-disjoint* [*OF* $\mathcal{B}$] **by** *blast*
qed

28.1.1 Family of Disjoint Sets

definition *disjoint-family-on* :: ('*i* ⇒ '*a set*) ⇒ '*i set* ⇒ *bool* **where**
disjoint-family-on A S ⟷ (∀ *m*∈*S*. ∀ *n*∈*S*. *m* ≠ *n* ⟶ *A m* ∩ *A n* = {})

abbreviation *disjoint-family A* ≡ *disjoint-family-on A UNIV*

lemma *disjoint-family-elem-disjnt*:
assumes *infinite A finite C*
and *df*: *disjoint-family-on B A*
obtains *x* **where** *x* ∈ *A disjnt C* (*B x*)
proof −
have *False* **if** ∗: ∀ *x* ∈ *A*. ∃ *y*. *y* ∈ *C* ∧ *y* ∈ *B x*
proof −
obtain *g* **where** *g*: ∀ *x* ∈ *A*. *g x* ∈ *C* ∧ *g x* ∈ *B x*
using ∗ **by** *metis*
with *df* **have** *inj-on g A*
by (*fastforce simp add*: *inj-on-def disjoint-family-on-def*)
then have *infinite* (*g* ' *A*)
using ‹*infinite A*› *finite-image-iff* **by** *blast*
then show *False*
by (*meson* ‹*finite C*› *finite-subset g image-subset-iff*)
qed
then show *?thesis*
by (*force simp*: *disjnt-iff intro*: *that*)
qed

lemma *disjoint-family-onD*:
disjoint-family-on A I ⟹ *i* ∈ *I* ⟹ *j* ∈ *I* ⟹ *i* ≠ *j* ⟹ *A i* ∩ *A j* = {}
by (*auto simp*: *disjoint-family-on-def*)

lemma *disjoint-family-subset*: *disjoint-family A* ⟹ (⋀*x*. *B x* ⊆ *A x*) ⟹ *disjoint-family B*
by (*force simp add*: *disjoint-family-on-def*)

lemma *disjoint-family-on-insert*:
i ∉ *I* ⟹ *disjoint-family-on A* (*insert i I*) ⟷ *A i* ∩ (⋃*i*∈*I*. *A i*) = {} ∧ *disjoint-family-on A I*

by (*fastforce simp*: *disjoint-family-on-def*)

lemma *disjoint-family-on-bisimulation*:
assumes *disjoint-family-on f S*
and $\bigwedge n\ m.\ n \in S \Longrightarrow m \in S \Longrightarrow n \neq m \Longrightarrow f\ n \cap f\ m = \{\} \Longrightarrow g\ n \cap g\ m = \{\}$
shows *disjoint-family-on g S*
using *assms* **unfolding** *disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-on-mono*:
$A \subseteq B \Longrightarrow$ *disjoint-family-on f B* $\Longrightarrow$ *disjoint-family-on f A*
unfolding *disjoint-family-on-def* **by** *auto*

lemma *disjoint-family-Suc*:
$(\bigwedge n.\ A\ n \subseteq A\ (Suc\ n)) \Longrightarrow$ *disjoint-family* $(\lambda i.\ A\ (Suc\ i) - A\ i)$
using *lift-Suc-mono-le*[*of A*]
by (*auto simp add*: *disjoint-family-on-def*)
(*metis insert-absorb insert-subset le-SucE le-antisym not-le-imp-less less-imp-le*)

lemma *disjoint-family-on-disjoint-image*:
disjoint-family-on A I $\Longrightarrow$ *disjoint* $(A\ `\ I)$
unfolding *disjoint-family-on-def disjoint-def* **by** *force*

lemma *disjoint-family-on-vimageI*: *disjoint-family-on F I* $\Longrightarrow$ *disjoint-family-on* $(\lambda i.\ f\ -`\ F\ i)\ I$
by (*auto simp*: *disjoint-family-on-def*)

lemma *disjoint-image-disjoint-family-on*:
assumes *d*: *disjoint* $(A\ `\ I)$ **and** *i*: *inj-on A I*
shows *disjoint-family-on A I*
unfolding *disjoint-family-on-def*
proof (*intro ballI impI*)
fix *n m* **assume** *nm*: $m \in I\ n \in I$ **and** $n \neq m$
with *i*[*THEN inj-onD*, *of n m*] **show** $A\ n \cap A\ m = \{\}$
by (*intro disjointD*[*OF d*]) *auto*
qed

lemma *disjoint-family-on-iff-disjoint-image*:
assumes $\bigwedge i.\ i \in I \Longrightarrow A\ i \neq \{\}$
shows *disjoint-family-on A I* $\longleftrightarrow$ *disjoint* $(A\ `\ I) \wedge$ *inj-on A I*
proof
assume *disjoint-family-on A I*
then show *disjoint* $(A\ `\ I) \wedge$ *inj-on A I*
by (*metis* (*mono-tags*, *lifting*) *assms disjoint-family-onD disjoint-family-on-disjoint-image inf.idem inj-onI*)
qed (*use disjoint-image-disjoint-family-on* **in** *metis*)

lemma *card-UN-disjoint'*:
assumes *disjoint-family-on A I* $\bigwedge i.\ i \in I \Longrightarrow$ *finite* $(A\ i)$ *finite I*

shows *card* $(\bigcup i{\in}I.\ A\ i) = (\sum i{\in}I.\ card\ (A\ i))$
using *assms* **by** (*simp add*: *card-UN-disjoint disjoint-family-on-def*)

lemma *disjoint-UN*:
assumes *F*: $\bigwedge i.\ i \in I \Longrightarrow$ *disjoint* $(F\ i)$ **and** $*$: *disjoint-family-on* $(\lambda i.\ \bigcup(F\ i))$ *I*
shows *disjoint* $(\bigcup i{\in}I.\ F\ i)$
proof (*safe intro*!: *disjointI del*: *equalityI*)
fix *A B i j* **assume** $A \neq B\ A \in F\ i\ i \in I\ B \in F\ j\ j \in I$
show $A \cap B = \{\}$
proof *cases*
assume $i = j$ **with** *F*[*of i*] ‹$i \in I$› ‹$A \in F\ i$› ‹$B \in F\ j$› ‹$A \neq B$› **show** $A \cap B = \{\}$
by (*auto dest*: *disjointD*)
next
assume $i \neq j$
with $*$ ‹$i{\in}I$› ‹$j{\in}I$› **have** $(\bigcup(F\ i)) \cap (\bigcup(F\ j)) = \{\}$
by (*rule disjoint-family-onD*)
with ‹$A{\in}F\ i$› ‹$i{\in}I$› ‹$B{\in}F\ j$› ‹$j{\in}I$›
show $A \cap B = \{\}$
by *auto*
qed
qed

lemma *distinct-list-bind*:
assumes *distinct xs* $\bigwedge x.\ x \in$ *set xs* $\Longrightarrow$ *distinct* $(f\ x)$
disjoint-family-on $(set \circ f)$ $(set\ xs)$
shows *distinct* $(List.bind\ xs\ f)$
using *assms*
by (*induction xs*)
(*auto simp*: *disjoint-family-on-def distinct-map inj-on-def set-list-bind*)

lemma *bij-betw-UNION-disjoint*:
assumes *disj*: *disjoint-family-on* $A'\ I$
assumes *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
from *bij* **show** *eq*: $f\ `\bigcup(A\ `\ I) = \bigcup(A'\ `\ I)$
by (*auto simp*: *bij-betw-def image-UN*)
show *inj-on f* $(\bigcup(A\ `\ I))$
proof (*rule inj-onI*, *clarify*)
fix *i j x y* **assume** *A*: $i \in I\ j \in I\ x \in A\ i\ y \in A\ j$ **and** *B*: $f\ x = f\ y$
from *A bij*[*of i*] *bij*[*of j*] **have** $f\ x \in A'\ i\ f\ y \in A'\ j$
by (*auto simp*: *bij-betw-def*)
with *B* **have** $A'\ i \cap A'\ j \neq \{\}$ **by** *auto*
with *disj A* **have** $i = j$ **unfolding** *disjoint-family-on-def* **by** *blast*
with *A B bij*[*of i*] **show** $x = y$ **by** (*auto simp*: *bij-betw-def dest*: *inj-onD*)
qed

qed

lemma *disjoint-union*: *disjoint* $C \Longrightarrow$ *disjoint* $B \Longrightarrow \bigcup C \cap \bigcup B = \{\} \Longrightarrow$ *disjoint* $(C \cup B)$
using *disjoint-UN*[*of* $\{C, B\}$ $\lambda x.\ x$] **by** (*auto simp add*: *disjoint-family-on-def*)

Sum/product of the union of a finite disjoint family

context *comm-monoid-set*
begin

lemma *UNION-disjoint-family*:
assumes *finite* I **and** $\forall i{\in}I.$ *finite* $(A\ i)$
and *disjoint-family-on* $A\ I$
shows $F\ g\ (\bigcup (A\ `\ I)) = F\ (\lambda x.\ F\ g\ (A\ x))\ I$
using *assms* **unfolding** *disjoint-family-on-def* **by** (*rule UNION-disjoint*)

lemma *Union-disjoint-sets*:
assumes $\forall A{\in}C.$ *finite* A **and** *disjoint* C
shows $F\ g\ (\bigcup C) = (F \circ F)\ g\ C$
using *assms* **unfolding** *disjoint-def* **by** (*rule Union-disjoint*)

end

The union of an infinite disjoint family of non-empty sets is infinite.

lemma *infinite-disjoint-family-imp-infinite-UNION*:
assumes ¬*finite* A $\bigwedge x.\ x \in A \Longrightarrow f\ x \neq \{\}$ *disjoint-family-on* $f\ A$
shows ¬*finite* $(\bigcup (f\ `\ A))$
proof −
define g **where** $g\ x = (SOME\ y.\ y \in f\ x)$ **for** x
have g: $g\ x \in f\ x$ **if** $x \in A$ **for** x
unfolding *g-def* **by** (*rule someI-ex*, *insert assms*(*2*) *that*) *blast*
have *inj-on-g*: *inj-on* $g\ A$
proof (*rule inj-onI*, *rule ccontr*)
fix $x\ y$ **assume** A: $x \in A\ y \in A\ g\ x = g\ y\ x \neq y$
with *g*[*of* x] *g*[*of* y] **have** $g\ x \in f\ x\ g\ x \in f\ y$ **by** *auto*
with A ‹$x \neq y$› *assms* **show** *False*
by (*auto simp*: *disjoint-family-on-def inj-on-def*)
qed
from g **have** $g\ `\ A \subseteq \bigcup (f\ `\ A)$ **by** *blast*
moreover from *inj-on-g* ‹¬*finite* A› **have** ¬*finite* $(g\ `\ A)$
using *finite-imageD* **by** *blast*
ultimately show *?thesis* **using** *finite-subset* **by** *blast*
qed

28.2 Construct Disjoint Sequences

definition *disjointed* :: $(nat \Rightarrow 'a\ set) \Rightarrow nat \Rightarrow 'a\ set$ **where**
disjointed $A\ n = A\ n - (\bigcup i{\in}\{0..<n\}.\ A\ i)$

lemma *finite-UN-disjointed-eq*: $(\bigcup i{\in}\{0..{<}n\}.\ disjointed\ A\ i) = (\bigcup i{\in}\{0..{<}n\}.\ A\ i)$
proof (*induct n*)
  case *0* **show** *?case* **by** *simp*
next
  case (*Suc n*)
  thus *?case* **by** (*simp add*: *atLeastLessThanSuc disjointed-def*)
qed

lemma *UN-disjointed-eq*: $(\bigcup i.\ disjointed\ A\ i) = (\bigcup i.\ A\ i)$
  by (*rule UN-finite2-eq* [**where** *k=0*])
    (*simp add*: *finite-UN-disjointed-eq*)

lemma *less-disjoint-disjointed*: $m < n \Longrightarrow disjointed\ A\ m \cap disjointed\ A\ n = \{\}$
  by (*auto simp add*: *disjointed-def*)

lemma *disjoint-family-disjointed*: *disjoint-family* (*disjointed A*)
  by (*simp add*: *disjoint-family-on-def*)
    (*metis neq-iff Int-commute less-disjoint-disjointed*)

lemma *disjointed-subset*: $disjointed\ A\ n \subseteq A\ n$
  by (*auto simp add*: *disjointed-def*)

lemma *disjointed-0*[*simp*]: $disjointed\ A\ 0 = A\ 0$
  by (*simp add*: *disjointed-def*)

lemma *disjointed-mono*: $mono\ A \Longrightarrow disjointed\ A\ (Suc\ n) = A\ (Suc\ n) - A\ n$
  using *mono-imp-UN-eq-last*[*of A*] **by** (*simp add*: *disjointed-def atLeastLessThanSuc-atLeastAtMost atLeast0AtMost*)

28.3 Partitions

Partitions P of a set A. We explicitly disallow empty sets.

definition *partition-on* :: *'a set* ⇒ *'a set set* ⇒ *bool*
where
  $partition\text{-}on\ A\ P \longleftrightarrow \bigcup P = A \wedge disjoint\ P \wedge \{\} \notin P$

lemma *partition-onI*:
  $\bigcup P = A \Longrightarrow (\bigwedge p\ q.\ p \in P \Longrightarrow q \in P \Longrightarrow p \neq q \Longrightarrow disjnt\ p\ q) \Longrightarrow \{\} \notin P \Longrightarrow partition\text{-}on\ A\ P$
  by (*auto simp*: *partition-on-def pairwise-def*)

lemma *partition-onD1*: $partition\text{-}on\ A\ P \Longrightarrow A = \bigcup P$
  by (*auto simp*: *partition-on-def*)

lemma *partition-onD2*: $partition\text{-}on\ A\ P \Longrightarrow disjoint\ P$
  by (*auto simp*: *partition-on-def*)

lemma *partition-onD3*: $partition\text{-}on\ A\ P \Longrightarrow \{\} \notin P$

by (*auto simp*: *partition-on-def*)

28.4 Constructions of partitions

lemma *partition-on-empty*: *partition-on* {} *P* ⟷ *P* = {}
unfolding *partition-on-def* **by** *fastforce*

lemma *partition-on-space*: *A* ≠ {} ⟹ *partition-on A* {*A*}
by (*auto simp*: *partition-on-def disjoint-def*)

lemma *partition-on-singletons*: *partition-on A* ((λ*x*. {*x*}) ‘ *A*)
by (*auto simp*: *partition-on-def disjoint-def*)

lemma *partition-on-transform*:
assumes *P*: *partition-on A P*
assumes *F-UN*: ⋃(*F* ‘ *P*) = *F* (⋃*P*) **and** *F-disjnt*: ⋀*p q*. *p* ∈ *P* ⟹ *q* ∈ *P* ⟹ *disjnt p q* ⟹ *disjnt* (*F p*) (*F q*)
shows *partition-on* (*F A*) (*F* ‘ *P* − {{}})
proof −
have ⋃(*F* ‘ *P* − {{}}) = *F A*
unfolding *P*[*THEN partition-onD1*] *F-UN*[*symmetric*] **by** *auto*
with *P* **show** *?thesis*
by (*auto simp add*: *partition-on-def pairwise-def intro*!: *F-disjnt*)
qed

lemma *partition-on-restrict*: *partition-on A P* ⟹ *partition-on* (*B* ∩ *A*) ((∩) *B* ‘ *P* − {{}})
by (*intro partition-on-transform*) (*auto simp*: *disjnt-def*)

lemma *partition-on-vimage*: *partition-on A P* ⟹ *partition-on* (*f* −‘ *A*) ((−‘) *f* ‘ *P* − {{}})
by (*intro partition-on-transform*) (*auto simp*: *disjnt-def*)

lemma *partition-on-inj-image*:
assumes *P*: *partition-on A P* **and** *f*: *inj-on f A*
shows *partition-on* (*f* ‘ *A*) ((‘) *f* ‘ *P* − {{}})
proof (*rule partition-on-transform*[*OF P*])
show *p* ∈ *P* ⟹ *q* ∈ *P* ⟹ *disjnt p q* ⟹ *disjnt* (*f* ‘ *p*) (*f* ‘ *q*) **for** *p q*
using *f*[*THEN inj-onD*] *P*[*THEN partition-onD1*] **by** (*auto simp*: *disjnt-def*)
qed *auto*

lemma *partition-on-insert*:
assumes *disjnt p* (⋃*P*)
shows *partition-on A* (*insert p P*) ⟷ *partition-on* (*A*−*p*) *P* ∧ *p* ⊆ *A* ∧ *p* ≠ {}
using *assms*
by (*auto simp*: *partition-on-def disjnt-iff pairwise-insert*)

28.5 Finiteness of partitions

lemma *finitely-many-partition-on*:

assumes *finite A*
shows *finite* {*P*. *partition-on A P*}
proof (*rule finite-subset*)
show {*P*. *partition-on A P*} ⊆ *Pow* (*Pow A*)
unfolding *partition-on-def* **by** *auto*
show *finite* (*Pow* (*Pow A*))
using *assms* **by** *simp*
qed

lemma *finite-elements*: *finite A* ⟹ *partition-on A P* ⟹ *finite P*
using *partition-onD1*[*of A P*] **by** (*simp add*: *finite-UnionD*)

lemma *product-partition*:
assumes *partition-on A P* **and** ⋀*p*. *p* ∈ *P* ⟹ *finite p*
shows *card A* = (∑ *p*∈*P*. *card p*)
using *assms* **unfolding** *partition-on-def* **by** (*meson card-Union-disjoint*)

28.6 Equivalence of partitions and equivalence classes

lemma *partition-on-quotient*:
assumes *r*: *equiv A r*
shows *partition-on A* (*A* // *r*)
proof (*rule partition-onI*)
from *r* **have** *r* ⊆ *A* × *A* **and** *refl-on A r*
by (*auto elim*: *equivE*)
then show ⋃(*A* // *r*) = *A* {} ∉ *A* // *r*
by (*auto simp*: *refl-on-def quotient-def*)

fix *p q* **assume** *p* ∈ *A* // *r q* ∈ *A* // *r p* ≠ *q*
then obtain *x y* **where** *x* ∈ *A y* ∈ *A p* = *r* `` {*x*} *q* = *r* `` {*y*}
by (*auto simp*: *quotient-def*)
with *r equiv-class-eq-iff*[*OF r*, *of x y*] ‹*p* ≠ *q*› **show** *disjnt p q*
by (*auto simp*: *disjnt-equiv-class*)
qed

lemma *equiv-partition-on*:
assumes *P*: *partition-on A P*
shows *equiv A* {(*x*, *y*). ∃ *p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*}
proof (*rule equivI*)
have *A* = ⋃ *P*
using *P* **by** (*auto simp*: *partition-on-def*)

show {(*x*, *y*). ∃ *p* ∈ *P*. *x* ∈ *p* ∧ *y* ∈ *p*} ⊆ *A* × *A*
unfolding ‹*A* = ⋃ *P*› **by** *blast*

show *refl-on A* {(*x*, *y*). ∃ *p*∈*P*. *x* ∈ *p* ∧ *y* ∈ *p*}
unfolding *refl-on-def* ‹*A* = ⋃ *P*› **by** *auto*
next
show *trans* {(*x*, *y*). ∃ *p*∈*P*. *x* ∈ *p* ∧ *y* ∈ *p*}

using P **by** (*auto simp only*: *trans-def disjoint-def partition-on-def*)
next
show *sym* $\{(x, y).\ \exists p{\in}P.\ x \in p \wedge y \in p\}$
by (*auto simp only*: *sym-def*)
qed

lemma *partition-on-eq-quotient*:
assumes P: *partition-on A P*
shows $A\ //\ \{(x, y).\ \exists p \in P.\ x \in p \wedge y \in p\} = P$
unfolding *quotient-def*
proof *safe*
fix x **assume** $x \in A$
then obtain p **where** $p \in P\ x \in p\ \bigwedge q.\ q \in P \Longrightarrow x \in q \Longrightarrow p = q$
using P **by** (*auto simp*: *partition-on-def disjoint-def*)
then have $\{y.\ \exists p{\in}P.\ x \in p \wedge y \in p\} = p$
by (*safe intro*!: *bexI*[*of - p*]) *simp*
then show $\{(x, y).\ \exists p{\in}P.\ x \in p \wedge y \in p\}\ ``\ \{x\} \in P$
by (*simp add*: ‹$p \in P$›)
next
fix p **assume** $p \in P$
then have $p \neq \{\}$
using P **by** (*auto simp*: *partition-on-def*)
then obtain x **where** $x \in p$
by *auto*
then have $x \in A\ \bigwedge q.\ q \in P \Longrightarrow x \in q \Longrightarrow p = q$
using P ‹$p \in P$› **by** (*auto simp*: *partition-on-def disjoint-def*)
with ‹$p{\in}P$› ‹$x \in p$› **have** $\{y.\ \exists p{\in}P.\ x \in p \wedge y \in p\} = p$
by (*safe intro*!: *bexI*[*of - p*]) *simp*
then show $p \in (\bigcup x{\in}A.\ \{\{(x, y).\ \exists p{\in}P.\ x \in p \wedge y \in p\}\ ``\ \{x\}\})$
by (*auto intro*: ‹$x \in A$›)
qed

lemma *partition-on-alt*: *partition-on A P* $\longleftrightarrow$ $(\exists r.\ equiv\ A\ r \wedge P = A\ //\ r)$
by (*auto simp*: *partition-on-eq-quotient intro*!: *partition-on-quotient intro*: *equiv-partition-on*)

lemma (**in** *comm-monoid-set*) *partition*:
assumes *finite X partition-on X A*
shows $F\ g\ X = F\ (\lambda B.\ F\ g\ B)\ A$
proof −
have *finite A*
using *assms finite-UnionD partition-onD1* **by** *auto*
have [*intro*]: *finite B* **if** $B \in A$ **for** B
by (*rule finite-subset*[*OF - assms*(*1*)])
(*use assms that* **in** ‹*auto simp*: *partition-on-def*›)
have $F\ g\ X = F\ g\ (\bigcup A)$
using *assms* **by** (*simp add*: *partition-on-def*)
also have $\ldots = (F \circ F)\ g\ A$
by (*rule Union-disjoint*) (*use assms* ‹*finite A*› **in** ‹*auto simp*: *partition-on-def disjoint-def*›)

finally show *?thesis*
by *simp*
qed

If h is an involution on X with no fixed points in X and $f(h(x)) = -f(x)$ then $\sum_{x \in X} f(x) = 0$.

This is easy to show in a ring with characteristic not equal to 2, since then we can do

$$\sum_{x \in X} f(x) = \sum_{x \in X} f(h(x)) = -\sum_{x \in X} f(x)$$

and therefore $2 \sum_{x \in X} f(x) = 0$.

However, the following proof also works in rings of characteristic 2. The idea is to simply partition X into a disjoint union of doubleton sets of the form $\{x, h(x)\}$.

lemma *sum-involution-eq-0*:
assumes *f-h*: $\bigwedge x.\ x \in X \Longrightarrow f\ (h\ x) + f\ x = 0$
assumes *h*: $\bigwedge x.\ x \in X \Longrightarrow h\ x \in X\ \bigwedge x.\ x \in X \Longrightarrow h\ (h\ x) = x\ \bigwedge x.\ x \in X \Longrightarrow h\ x \neq x$
shows $(\sum x{\in}X.\ f\ x) = 0$
proof (*cases finite X*)
assume *fin*: *finite X*
define *R* **where** $R = \{(x,y).\ x \in X \wedge y \in X \wedge (x = y \vee h\ x = y)\}$
have *R*: *equiv X R*
unfolding *equiv-def R-def* **using** *h*(*2*,*3*)
by (*auto simp*: *refl-on-def sym-def trans-def*)
define *A* **where** $A = X\ //\ R$
have *partition*: *partition-on X A*
unfolding *A-def* **using** *R* **by** (*rule partition-on-quotient*)

have $(\sum x{\in}X.\ f\ x) = (\sum B{\in}A.\ \sum x{\in}B.\ f\ x)$
by (*subst sum.partition*[*OF fin partition*]) *auto*
also have $\ldots = (\sum B{\in}A.\ 0)$
proof (*rule sum.cong*)
fix *B* **assume** *B*: $B \in A$
then obtain *x* **where** *x*: $x \in X$ **and** [*simp*]: $B = R\ ``\ \{x\}$
using *R* **by** (*metis A-def quotientE*)
have $R\ ``\ \{x\} = \{x,\ h\ x\}\ h\ x \neq x$
using *h*(*1*) **by** (*auto simp*: *R-def*)
thus $(\sum x{\in}B.\ f\ x) = 0$
using *x f-h*[*of x*] **by** (*simp add*: *add.commute*)
qed *auto*
finally show *?thesis*
by *simp*
qed *auto*

28.7 Refinement of partitions

definition *refines* :: *'a set* ⇒ *'a set set* ⇒ *'a set set* ⇒ *bool*

where *refines A P Q* ≡
partition-on A P ∧ *partition-on A Q* ∧ (∀ *X*∈*P*. ∃ *Y*∈*Q*. *X* ⊆ *Y*)

lemma *refines-refl*: *partition-on A P* ⟹ *refines A P P*
using *refines-def* **by** *blast*

lemma *refines-asym1*:
assumes *refines A P Q refines A Q P*
shows *P* ⊆ *Q*
proof
fix *X*
assume *X* ∈ *P*
then obtain *Y X'* **where** *Y* ∈ *Q X* ⊆ *Y X'* ∈ *P Y* ⊆ *X'*
by (*meson assms refines-def*)
then have *X'* = *X*
using *assms*(*2*) **unfolding** *partition-on-def refines-def*
by (*metis* ‹*X* ∈ *P*› ‹*X* ⊆ *Y*› *disjnt-self-iff-empty disjnt-subset1 pairwiseD*)
then show *X* ∈ *Q*
using ‹*X* ⊆ *Y*› ‹*Y* ∈ *Q*› ‹*Y* ⊆ *X'*› **by** *force*
qed

lemma *refines-asym*: ⟦*refines A P Q*; *refines A Q P*⟧ ⟹ *P*=*Q*
by (*meson antisym-conv refines-asym1*)

lemma *refines-trans*: ⟦*refines A P Q*; *refines A Q R*⟧ ⟹ *refines A P R*
by (*meson order.trans refines-def*)

lemma *refines-obtains-subset*:
assumes *refines A P Q q* ∈ *Q*
shows *partition-on q* {*p* ∈ *P*. *p* ⊆ *q*}
proof −
have *p* ⊆ *q* ∨ *disjnt p q* **if** *p* ∈ *P* **for** *p*
using *that assms* **unfolding** *refines-def partition-on-def disjoint-def*
by (*metis disjnt-def disjnt-subset1*)
with *assms* **have** *q* ⊆ *Union* {*p* ∈ *P*. *p* ⊆ *q*}
using *assms*
by (*clarsimp simp*: *refines-def disjnt-iff partition-on-def*) (*metis Union-iff*)
with *assms* **have** *q* = *Union* {*p* ∈ *P*. *p* ⊆ *q*}
by *auto*
then show *?thesis*
using *assms* **by** (*auto simp*: *refines-def disjoint-def partition-on-def*)
qed

28.8 The coarsest common refinement of a set of partitions

definition *common-refinement* :: *'a set set set* ⇒ *'a set set*
where *common-refinement* $\mathcal{P}$ ≡ (⋃*f* ∈ (Π$_E$ *P*∈$\mathcal{P}$. *P*). {⋂ (*f* ‘ $\mathcal{P}$)}) − {{}}

With non-extensional function space

lemma *common-refinement*: *common-refinement* $\mathcal{P}$ = (⋃*f* ∈ (Π *P*∈$\mathcal{P}$. *P*). {⋂ (*f*

‘ $\mathcal{P}$)}) − {{}}
(**is** *?lhs* = *?rhs*)
proof
show *?rhs* ⊆ *?lhs*
apply (*clarsimp simp add*: *common-refinement-def PiE-def Ball-def*)
by (*metis restrict-Pi-cancel image-restrict-eq restrict-extensional*)
qed (*auto simp add*: *common-refinement-def PiE-def*)

lemma *common-refinement-exists*: ⟦$X \in$ *common-refinement* $\mathcal{P}$; $P \in \mathcal{P}$⟧ ⟹ ∃ R∈P. $X \subseteq R$
by (*auto simp add*: *common-refinement*)

lemma *Union-common-refinement*: ⋃ (*common-refinement* $\mathcal{P}$) = (⋂ P∈$\mathcal{P}$. ⋃ P)
proof
show (⋂ P∈$\mathcal{P}$. ⋃ P) ⊆ ⋃ (*common-refinement* $\mathcal{P}$)
proof (*clarsimp simp*: *common-refinement*)
fix x
assume ∀ P∈$\mathcal{P}$. ∃ X∈P. $x \in X$
then obtain F **where** F: ⋀P. P∈$\mathcal{P}$ ⟹ $F\ P \in P \wedge x \in F\ P$
by *metis*
then have $x \in \bigcap$ (F ‘ $\mathcal{P}$)
by *force*
with F **show** ∃ X∈(⋃x∈Π P∈$\mathcal{P}$. P. {⋂ (x ‘ $\mathcal{P}$)}) − {{}}. $x \in X$
by (*auto simp add*: *Pi-iff Bex-def*)
qed
qed (*auto simp*: *common-refinement-def*)

lemma *partition-on-common-refinement*:
assumes A: ⋀P. $P \in \mathcal{P}$ ⟹ *partition-on* A P **and** $\mathcal{P} \neq$ {}
shows *partition-on* A (*common-refinement* $\mathcal{P}$)
proof (*rule partition-onI*)
show ⋃ (*common-refinement* $\mathcal{P}$) = A
using *assms* **by** (*simp add*: *partition-on-def Union-common-refinement*)
fix P Q
assume $P \in$ *common-refinement* $\mathcal{P}$ **and** $Q \in$ *common-refinement* $\mathcal{P}$ **and** $P \neq Q$
then obtain f g **where** f: $f \in (\Pi_E\ P$∈$\mathcal{P}$. $P)$ **and** P: $P = \bigcap$ (f ‘ $\mathcal{P}$) **and** $P \neq$ {}
and g: $g \in (\Pi_E\ P$∈$\mathcal{P}$. $P)$ **and** Q: $Q = \bigcap$ (g ‘ $\mathcal{P}$) **and** $Q \neq$ {}
by (*auto simp add*: *common-refinement-def*)
have f=g **if** $x \in P$ $x \in Q$ **for** x
proof (*rule extensionalityI* [*of* - $\mathcal{P}$])
fix R
assume $R \in \mathcal{P}$
with *that* P Q f g A [*unfolded partition-on-def*, *OF* ‹$R \in \mathcal{P}$›]
show $f\ R = g\ R$
by (*metis INT-E Int-iff PiE-iff disjointD emptyE*)
qed (*use PiE-iff f g* **in** *auto*)
then show *disjnt* P Q

by (*metis P Q ‹P ≠ Q› disjnt-iff*)
qed (*simp add*: *common-refinement-def*)

lemma *refines-common-refinement*:
assumes $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *partition-on A P P* $\in \mathcal{P}$
shows *refines A* (*common-refinement* $\mathcal{P}$) *P*
unfolding *refines-def*
proof (*intro conjI strip*)
fix *X*
assume $X \in$ *common-refinement* $\mathcal{P}$
with *assms* **show** $\exists\, Y{\in}P.\ X \subseteq Y$
by (*auto simp*: *common-refinement-def*)
qed (*use assms partition-on-common-refinement* **in** *auto*)

The common refinement is itself refined by any other

lemma *common-refinement-coarsest*:
assumes $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *partition-on A P partition-on A R* $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *refines A R P* $\mathcal{P} \neq \{\}$
shows *refines A R* (*common-refinement* $\mathcal{P}$)
unfolding *refines-def*
proof (*intro conjI ballI partition-on-common-refinement*)
fix *X*
assume $X \in R$
have $\exists\, p \in P.\ X \subseteq p$ **if** $P \in \mathcal{P}$ **for** *P*
by (*meson ‹X ∈ R› assms*(*3*) *refines-def that*)
then obtain *F* **where** *f*: $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow F\ P \in P \wedge X \subseteq F\ P$
by *metis*
with *‹partition-on A R› ‹X ∈ R› ‹*$\mathcal{P} \neq \{\}$*›*
have $\bigcap\ (F\ `\ \mathcal{P}) \in$ *common-refinement* $\mathcal{P}$
apply (*simp add*: *partition-on-def common-refinement Pi-iff Bex-def*)
by (*metis* (*no-types*, *lifting*) *cINF-greatest subset-empty*)
with *f* **show** $\exists\, Y{\in}$*common-refinement* $\mathcal{P}.\ X \subseteq Y$
by (*metis ‹*$\mathcal{P} \neq \{\}$*› cINF-greatest*)
qed (*use assms* **in** *auto*)

lemma *finite-common-refinement*:
assumes *finite* $\mathcal{P}$ $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *finite P*
shows *finite* (*common-refinement* $\mathcal{P}$)
proof −
have *finite* ($\Pi_E\ P{\in}\mathcal{P}.\ P$)
by (*simp add*: *assms finite-PiE*)
then show *?thesis*
by (*auto simp*: *common-refinement-def*)
qed

lemma *card-common-refinement*:
assumes *finite* $\mathcal{P}$ $\bigwedge P.\ P \in \mathcal{P} \Longrightarrow$ *finite P*
shows *card* (*common-refinement* $\mathcal{P}$) $\leq (\prod P \in \mathcal{P}.$ *card P*)
proof −

have *card* (*common-refinement* $\mathcal{P}$) $\leq$ *card* ($\bigcup f \in (\Pi_E\ P{\in}\mathcal{P}.\ P).\ \{\bigcap\ (f\ `\ \mathcal{P})\}$)
unfolding *common-refinement-def* **by** (*meson card-Diff1-le*)
also have ... $\leq$ ($\sum f{\in}(\Pi_E\ P{\in}\mathcal{P}.\ P).$ *card*$\{\bigcap\ (f\ `\ \mathcal{P})\}$)
by (*metis assms finite-PiE card-UN-le*)
also have ... = *card*($\Pi_E\ P{\in}\mathcal{P}.\ P$)
by *simp*
also have ... = ($\prod P \in \mathcal{P}.$ *card P*)
by (*simp add*: *assms*(*1*) *card-PiE dual-order.eq-iff*)
finally show *?thesis* **.**
qed

end

29 Type of finite sets defined as a subtype of sets

theory *FSet*
imports *Main Countable*
begin

29.1 Definition of the type

typedef *'a fset* = {*A* :: *'a set*. *finite A*} **morphisms** *fset Abs-fset*
by *auto*

setup-lifting *type-definition-fset*

29.2 Basic operations and type class instantiations

instantiation *fset* :: (*finite*) *finite*
begin
instance by (*standard*; *transfer*; *simp*)
end

instantiation *fset* :: (*type*) {*bounded-lattice-bot*, *distrib-lattice*, *minus*}
begin

lift-definition *bot-fset* :: *'a fset* **is** {} **parametric** *empty-transfer* **by** *simp*

lift-definition *less-eq-fset* :: *'a fset* $\Rightarrow$ *'a fset* $\Rightarrow$ *bool* **is** *subset-eq* **parametric** *subset-transfer*
.

definition *less-fset* :: *'a fset* $\Rightarrow$ *'a fset* $\Rightarrow$ *bool* **where** $xs < ys \equiv xs \leq ys \wedge xs \neq$ (*ys*::*'a fset*)

lemma *less-fset-transfer*[*transfer-rule*]:
includes *lifting-syntax*
assumes [*transfer-rule*]: *bi-unique A*
shows ((*pcr-fset A*) ===> (*pcr-fset A*) ===> (=)) ($\subset$) (<)

unfolding *less-fset-def*[*abs-def*] *psubset-eq*[*abs-def*] **by** *transfer-prover*

lift-definition *sup-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *union* **parametric** *union-transfer*
by *simp*

lift-definition *inf-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *inter* **parametric** *inter-transfer*
by *simp*

lift-definition *minus-fset* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* **is** *minus* **parametric** *Diff-transfer*
by *simp*

instance
by (*standard*; *transfer*; *auto*)+

end

abbreviation *fempty* :: *'a fset* (‹{||}›) **where** {||} ≡ *bot*
abbreviation *fsubset-eq* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** ‹|⊆|› *50*) **where** *xs* |⊆| *ys* ≡ *xs* ≤ *ys*
abbreviation *fsubset* :: *'a fset* ⇒ *'a fset* ⇒ *bool* (**infix** ‹|⊂|› *50*) **where** *xs* |⊂| *ys* ≡ *xs* < *ys*
abbreviation *funion* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** ‹|∪|› *65*) **where** *xs* |∪| *ys* ≡ *sup xs ys*
abbreviation *finter* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** ‹|∩|› *65*) **where** *xs* |∩| *ys* ≡ *inf xs ys*
abbreviation *fminus* :: *'a fset* ⇒ *'a fset* ⇒ *'a fset* (**infixl** ‹|−|› *65*) **where** *xs* |−| *ys* ≡ *minus xs ys*

instantiation *fset* :: (*equal*) *equal*
begin
definition *HOL.equal A B* ⟷ *A* |⊆| *B* ∧ *B* |⊆| *A*
instance by *intro-classes* (*auto simp add*: *equal-fset-def*)
end

instantiation *fset* :: (*type*) *conditionally-complete-lattice*
begin

context includes *lifting-syntax*
begin

lemma *right-total-Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *right-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*)
(λ*S*. *if finite* (⋂ *S* ∩ *Collect* (*Domainp A*)) *then* ⋂ *S* ∩ *Collect* (*Domainp A*) *else* {})
(λ*S*. *if finite* (*Inf S*) *then Inf S else* {})

by *transfer-prover*

lemma *Inf-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A* **and** [*transfer-rule*]: *bi-total A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
(λ*A*. *if finite* (*Inf A*) *then Inf A else* {})
by *transfer-prover*

lift-definition *Inf-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Inf A*) *then Inf A else* {}
parametric *right-total-Inf-fset-transfer Inf-fset-transfer* **by** *simp*

lemma *Sup-fset-transfer*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*rel-set* (*rel-set A*) ===> *rel-set A*) (λ*A*. *if finite* (*Sup A*) *then Sup A else* {})
(λ*A*. *if finite* (*Sup A*) *then Sup A else* {}) **by** *transfer-prover*

lift-definition *Sup-fset* :: *'a fset set* ⇒ *'a fset* **is** λ*A*. *if finite* (*Sup A*) *then Sup A else* {}
parametric *Sup-fset-transfer* **by** *simp*

lemma *finite-Sup*: ∃ *z*. *finite z* ∧ (∀ *a*. *a* ∈ *X* ⟶ *a* ≤ *z*) ⟹ *finite* (*Sup X*)
by (*auto intro*: *finite-subset*)

lemma *transfer-bdd-below*[*transfer-rule*]: (*rel-set* (*pcr-fset* (=)) ===> (=)) *bdd-below bdd-below*
by *auto*

end

instance
proof
fix *x z* :: *'a fset*
fix *X* :: *'a fset set*
show *x* ∈ *X* ⟹ *bdd-below X* ⟹ *Inf X* |⊆| *x* **by** *transfer auto*
show *X* ≠ {} ⟹ (⋀*x*. *x* ∈ *X* ⟹ *z* |⊆| *x*) ⟹ *z* |⊆| *Inf X* **by** *transfer* (*clarsimp*, *blast*)
show *x* |⊆| *Sup X* **if** *x* ∈ *X bdd-above X*
proof −
from *that* **obtain** *z* **where** *x* ∈ *X* (⋀*x*. *x* ∈ *X* ⟹ *x* |⊆| *z*)
by (*auto simp*: *bdd-above-def*)
then show *?thesis*
by *transfer* (*auto intro*!: *finite-Sup*)
qed
show *X* ≠ {} ⟹ (⋀*x*. *x* ∈ *X* ⟹ *x* |⊆| *z*) ⟹ *Sup X* |⊆| *z*
by *transfer* (*clarsimp*, *blast*)
qed

end

instantiation *fset* :: (*finite*) *complete-lattice*
begin

lift-definition *top-fset* :: *'a fset* **is** *UNIV* **parametric** *right-total-UNIV-transfer UNIV-transfer*
by *simp*

instance
by (*standard*; *transfer*; *auto*)

end

instantiation *fset* :: (*finite*) *complete-boolean-algebra*
begin

lift-definition *uminus-fset* :: *'a fset* ⇒ *'a fset* **is** *uminus*
parametric *right-total-Compl-transfer Compl-transfer* **by** *simp*

instance
by (*standard*; *transfer*) (*simp-all add*: *Inf-Sup Diff-eq*)
end

abbreviation *fUNIV* :: *'a*::*finite fset* **where** *fUNIV* ≡ *top*
abbreviation *fuminus* :: *'a*::*finite fset* ⇒ *'a fset* (‹|−| -› [*81*] *80*) **where** |−| *x* ≡ *uminus x*

declare *top-fset.rep-eq*[*simp*]

29.3 Other operations

lift-definition *finsert* :: *'a* ⇒ *'a fset* ⇒ *'a fset* **is** *insert* **parametric** *Lifting-Set.insert-transfer*
by *simp*

syntax
-fset :: *args* => *'a fset* (‹(‹*indent=2 notation=‹mixfix finite set enumeration›*›{|-|})›)
syntax-consts
-fset ⇌ *finsert*
translations
{|*x*, *xs*|} == *CONST finsert x* {|*xs*|}
{|*x*|} == *CONST finsert x* {||}

abbreviation *fmember* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** ‹|∈|› *50*) **where**
x |∈| *X* ≡ *x* ∈ *fset X*

abbreviation *not-fmember* :: *'a* ⇒ *'a fset* ⇒ *bool* (**infix** ‹|∉|› *50*) **where**
x |∉| *X* ≡ *x* ∉ *fset X*

context
begin

qualified abbreviation *Ball* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Ball X ≡ *Set.Ball* (*fset X*)

alias *fBall* = *FSet.Ball*

qualified abbreviation *Bex* :: *'a fset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool* **where**
Bex X ≡ *Set.Bex* (*fset X*)

alias *fBex* = *FSet.Bex*

end

syntax (*input*)
-fBall :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* !››! (-/|:|-)./ -)› [*0*, *0*, *10*] *10*)
-fBex :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ?›› *?* (-/|:|-)./ -)› [*0*, *0*, *10*] *10*)
-fBex1 :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ?!›› *?*! (-/:-)./ -)› [*0*, *0*, *10*] *10*)

syntax
-fBall :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∀›› ∀ (-/|∈|-)./ -)› [*0*, *0*, *10*] *10*)
-fBex :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∃›› ∃ (-/|∈|-)./ -)› [*0*, *0*, *10*] *10*)
-fBnex :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∄›› ∄ (-/|∈|-)./ -)› [*0*, *0*, *10*] *10*)
-fBex1 :: *pttrn* ⇒ *'a fset* ⇒ *bool* ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∃!›› ∃! (-/|∈|-)./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-fBall -fBnex ⇌ *fBall* **and**
-fBex ⇌ *fBex* **and**
-fBex1 ⇌ *Ex1*

translations
∀ *x*|∈|*A*. *P* ⇌ *CONST FSet.Ball A* (λ*x*. *P*)
∃ *x*|∈|*A*. *P* ⇌ *CONST FSet.Bex A* (λ*x*. *P*)
∄ *x*|∈|*A*. *P* ⇌ *CONST fBall A* (λ*x*. ¬ *P*)
∃! *x*|∈|*A*. *P* ⇀ ∃!*x*. *x* |∈| *A* ∧ *P*

typed-print-translation ‹
[(***const-syntax*** ‹*fBall*›, *Syntax-Trans.preserve-binder-abs2-tr'* ***syntax-const*** ‹*-fBall*›),
(***const-syntax*** ‹*fBex*›, *Syntax-Trans.preserve-binder-abs2-tr'* ***syntax-const*** ‹*-fBex*›)]
› — to avoid eta-contraction of body

syntax
-setlessfAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∀ ››∀ -|⊂|-./ -)› [*0*, *0*, *10*] *10*)
-setlessfEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∃ ››∃ -|⊂|-./ -)› [*0*, *0*, *10*] *10*)
-setlefAll :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∀ ››∀ -|⊆|-./ -)› [*0*, *0*, *10*] *10*)
-setlefEx :: [*idt*, *'a*, *bool*] ⇒ *bool* (‹(‹*indent=3 notation=‹binder finite* ∃ ››∃ -|⊆|-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-setlessfAll -setlefAll ⇌ *All* **and**
-setlessfEx -setlefEx ⇌ *Ex*

translations
∀ *A*|⊂|*B*. *P* ⇀ ∀ *A*. *A* |⊂| *B* ⟶ *P*
∃ *A*|⊂|*B*. *P* ⇀ ∃ *A*. *A* |⊂| *B* ∧ *P*
∀ *A*|⊆|*B*. *P* ⇀ ∀ *A*. *A* |⊆| *B* ⟶ *P*
∃ *A*|⊆|*B*. *P* ⇀ ∃ *A*. *A* |⊆| *B* ∧ *P*

context includes *lifting-syntax*
begin

lemma *fmember-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *pcr-fset A* ===> (=)) (∈) (|∈|)
by *transfer-prover*

lemma *fBall-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*pcr-fset A* ===> (*A* ===> (=)) ===> (=)) (*Ball*) (*fBall*)
by *transfer-prover*

lemma *fBex-transfer0*[*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*pcr-fset A* ===> (*A* ===> (=)) ===> (=)) (*Bex*) (*fBex*)
by *transfer-prover*

lift-definition *ffilter* :: (*'a* ⇒ *bool*) ⇒ *'a fset* ⇒ *'a fset* **is** *Set.filter*
parametric *Lifting-Set.filter-transfer* **by** *simp*

lift-definition *fPow* :: *'a fset* ⇒ *'a fset fset* **is** *Pow* **parametric** *Pow-transfer*
by (*simp add*: *finite-subset*)

lift-definition *fcard* :: *'a fset* ⇒ *nat* **is** *card* **parametric** *card-transfer* **.**

lift-definition *fimage* :: (*'a* ⇒ *'b*) ⇒ *'a fset* ⇒ *'b fset* (**infixr** ‹|'|› *90*) **is** *image*
parametric *image-transfer* **by** *simp*

lift-definition *fthe-elem* :: *'a fset* ⇒ *'a* **is** *the-elem* **.**

lift-definition *fbind* :: *'a fset* ⇒ (*'a* ⇒ *'b fset*) ⇒ *'b fset* **is** *Set.bind* **parametric** *bind-transfer*
by (*simp add*: *Set.bind-def*)

lift-definition *ffUnion* :: *'a fset fset* ⇒ *'a fset* **is** *Union* **parametric** *Union-transfer*
by *simp*

lift-definition *ffold* :: (*'a* ⇒ *'b* ⇒ *'b*) ⇒ *'b* ⇒ *'a fset* ⇒ *'b* **is** *Finite-Set.fold* **.**

lift-definition *fset-of-list* :: *'a list* ⇒ *'a fset* **is** *set* **by** (*rule finite-set*)

lift-definition *sorted-list-of-fset* :: *'a*::*linorder fset* ⇒ *'a list* **is** *sorted-list-of-set* **.**

29.4 Transferred lemmas from Set.thy

lemma *fset-eqI*: (⋀*x*. (*x* |∈| *A*) = (*x* |∈| *B*)) ⟹ *A* = *B*
by (*rule set-eqI*[*Transfer.transferred*])

lemma *fset-eq-iff*[*no-atp*]: (*A* = *B*) = (∀*x*. (*x* |∈| *A*) = (*x* |∈| *B*))
by (*rule set-eq-iff*[*Transfer.transferred*])

lemma *fBallI*[*no-atp*]: (⋀*x*. *x* |∈| *A* ⟹ *P x*) ⟹ *fBall A P*
using *ballI* **.**

lemma *fbspec*[*no-atp*]: *fBall A P* ⟹ *x* |∈| *A* ⟹ *P x*
using *bspec* **.**

lemma *fBallE*[*no-atp*]: *fBall A P* ⟹ (*P x* ⟹ *Q*) ⟹ (*x* |∉| *A* ⟹ *Q*) ⟹ *Q*
using *ballE* **.**

lemma *fBexI*[*no-atp*]: *P x* ⟹ *x* |∈| *A* ⟹ *fBex A P*
using *bexI* **.**

lemma *rev-fBexI*[*no-atp*]: *x* |∈| *A* ⟹ *P x* ⟹ *fBex A P*
using *rev-bexI* **.**

lemma *fBexCI*[*no-atp*]: (*fBall A* (λ*x*. ¬ *P x*) ⟹ *P a*) ⟹ *a* |∈| *A* ⟹ *fBex A P*
using *bexCI* **.**

lemma *fBexE*[*no-atp*]: *fBex A P* ⟹ (⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q*) ⟹ *Q*
using *bexE* **.**

lemma *fBall-triv*[*no-atp*]: *fBall A* (λ*x*. *P*) = ((∃*x*. *x* |∈| *A*) ⟶ *P*)
using *ball-triv* **.**

lemma *fBex-triv*[*no-atp*]: *fBex A* (λ*x*. *P*) = ((∃*x*. *x* |∈| *A*) ∧ *P*)

using *bex-triv* **.**

lemma *fBex-triv-one-point1*[*no-atp*]: *fBex A* (λ*x*. *x* = *a*) = (*a* |∈| *A*)
using *bex-triv-one-point1* **.**

lemma *fBex-triv-one-point2*[*no-atp*]: *fBex A* ((=) *a*) = (*a* |∈| *A*)
using *bex-triv-one-point2* **.**

lemma *fBex-one-point1*[*no-atp*]: *fBex A* (λ*x*. *x* = *a* ∧ *P x*) = (*a* |∈| *A* ∧ *P a*)
using *bex-one-point1* **.**

lemma *fBex-one-point2*[*no-atp*]: *fBex A* (λ*x*. *a* = *x* ∧ *P x*) = (*a* |∈| *A* ∧ *P a*)
using *bex-one-point2* **.**

lemma *fBall-one-point1*[*no-atp*]: *fBall A* (λ*x*. *x* = *a* ⟶ *P x*) = (*a* |∈| *A* ⟶ *P a*)
using *ball-one-point1* **.**

lemma *fBall-one-point2*[*no-atp*]: *fBall A* (λ*x*. *a* = *x* ⟶ *P x*) = (*a* |∈| *A* ⟶ *P a*)
using *ball-one-point2* **.**

lemma *fBall-conj-distrib*: *fBall A* (λ*x*. *P x* ∧ *Q x*) = (*fBall A P* ∧ *fBall A Q*)
using *ball-conj-distrib* **.**

lemma *fBex-disj-distrib*: *fBex A* (λ*x*. *P x* ∨ *Q x*) = (*fBex A P* ∨ *fBex A Q*)
using *bex-disj-distrib* **.**

lemma *fBall-cong*[*fundef-cong*]: *A* = *B* ⟹ (⋀*x*. *x* |∈| *B* ⟹ *P x* = *Q x*) ⟹ *fBall A P* = *fBall B Q*
by (*rule ball-cong*[*Transfer.transferred*])

lemma *fBex-cong*[*fundef-cong*]: *A* = *B* ⟹ (⋀*x*. *x* |∈| *B* ⟹ *P x* = *Q x*) ⟹ *fBex A P* = *fBex B Q*
by (*rule bex-cong*[*Transfer.transferred*])

lemma *fsubsetI*[*intro*!]: (⋀*x*. *x* |∈| *A* ⟹ *x* |∈| *B*) ⟹ *A* |⊆| *B*
by (*rule subsetI*[*Transfer.transferred*])

lemma *fsubsetD*[*elim*, *intro?*]: *A* |⊆| *B* ⟹ *c* |∈| *A* ⟹ *c* |∈| *B*
by (*rule subsetD*[*Transfer.transferred*])

lemma *rev-fsubsetD*[*no-atp*,*intro?*]: *c* |∈| *A* ⟹ *A* |⊆| *B* ⟹ *c* |∈| *B*
by (*rule rev-subsetD*[*Transfer.transferred*])

lemma *fsubsetCE*[*no-atp*,*elim*]: *A* |⊆| *B* ⟹ (*c* |∉| *A* ⟹ *P*) ⟹ (*c* |∈| *B* ⟹ *P*) ⟹ *P*
by (*rule subsetCE*[*Transfer.transferred*])

lemma *fsubset-eq*[*no-atp*]: $(A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ x \mid\in\mid B)$
by (*rule subset-eq*[*Transfer.transferred*])

lemma *contra-fsubsetD*[*no-atp*]: $A \mid\subseteq\mid B \Longrightarrow c \mid\notin\mid B \Longrightarrow c \mid\notin\mid A$
by (*rule contra-subsetD*[*Transfer.transferred*])

lemma *fsubset-refl*: $A \mid\subseteq\mid A$
by (*rule subset-refl*[*Transfer.transferred*])

lemma *fsubset-trans*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid C \Longrightarrow A \mid\subseteq\mid C$
by (*rule subset-trans*[*Transfer.transferred*])

lemma *fset-rev-mp*: $c \mid\in\mid A \Longrightarrow A \mid\subseteq\mid B \Longrightarrow c \mid\in\mid B$
by (*rule rev-subsetD*[*Transfer.transferred*])

lemma *fset-mp*: $A \mid\subseteq\mid B \Longrightarrow c \mid\in\mid A \Longrightarrow c \mid\in\mid B$
by (*rule subsetD*[*Transfer.transferred*])

lemma *fsubset-not-fsubset-eq*[*code*]: $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge \neg\ B \mid\subseteq\mid A)$
by (*rule subset-not-subset-eq*[*Transfer.transferred*])

lemma *eq-fmem-trans*: $a = b \Longrightarrow b \mid\in\mid A \Longrightarrow a \mid\in\mid A$
by (*rule eq-mem-trans*[*Transfer.transferred*])

lemma *fsubset-antisym*[*intro*!]: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow A = B$
by (*rule subset-antisym*[*Transfer.transferred*])

lemma *fequalityD1*: $A = B \Longrightarrow A \mid\subseteq\mid B$
by (*rule equalityD1*[*Transfer.transferred*])

lemma *fequalityD2*: $A = B \Longrightarrow B \mid\subseteq\mid A$
by (*rule equalityD2*[*Transfer.transferred*])

lemma *fequalityE*: $A = B \Longrightarrow (A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow P) \Longrightarrow P$
by (*rule equalityE*[*Transfer.transferred*])

lemma *fequalityCE*[*elim*]:
$A = B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow P) \Longrightarrow (c \mid\notin\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow P) \Longrightarrow P$
by (*rule equalityCE*[*Transfer.transferred*])

lemma *eqfset-imp-iff*: $A = B \Longrightarrow (x \mid\in\mid A) = (x \mid\in\mid B)$
by (*rule eqset-imp-iff*[*Transfer.transferred*])

lemma *eqfelem-imp-iff*: $x = y \Longrightarrow (x \mid\in\mid A) = (y \mid\in\mid A)$
by (*rule eqelem-imp-iff*[*Transfer.transferred*])

lemma *fempty-iff*[*simp*]: $(c \mid\in\mid \{||\}) = \mathit{False}$
by (*rule empty-iff*[*Transfer.transferred*])

lemma *fempty-fsubsetI*[*iff*]: $\{||\} \mid\subseteq\mid x$
by (*rule empty-subsetI*[*Transfer.transferred*])

lemma *equalsffemptyI*: $(\bigwedge y.\ y \mid\in\mid A \Longrightarrow False) \Longrightarrow A = \{||\}$
by (*rule equals0I*[*Transfer.transferred*])

lemma *equalsffemptyD*: $A = \{||\} \Longrightarrow a \mid\notin\mid A$
by (*rule equals0D*[*Transfer.transferred*])

lemma *fBall-fempty*[*simp*]: *fBall* $\{||\}\ P = True$
by (*rule ball-empty*[*Transfer.transferred*])

lemma *fBex-fempty*[*simp*]: *fBex* $\{||\}\ P = False$
by (*rule bex-empty*[*Transfer.transferred*])

lemma *fPow-iff*[*iff*]: $(A \mid\in\mid fPow\ B) = (A \mid\subseteq\mid B)$
by (*rule Pow-iff*[*Transfer.transferred*])

lemma *fPowI*: $A \mid\subseteq\mid B \Longrightarrow A \mid\in\mid fPow\ B$
by (*rule PowI*[*Transfer.transferred*])

lemma *fPowD*: $A \mid\in\mid fPow\ B \Longrightarrow A \mid\subseteq\mid B$
by (*rule PowD*[*Transfer.transferred*])

lemma *fPow-bottom*: $\{||\} \mid\in\mid fPow\ B$
by (*rule Pow-bottom*[*Transfer.transferred*])

lemma *fPow-top*: $A \mid\in\mid fPow\ A$
by (*rule Pow-top*[*Transfer.transferred*])

lemma *fPow-not-fempty*: $fPow\ A \neq \{||\}$
by (*rule Pow-not-empty*[*Transfer.transferred*])

lemma *finter-iff*[*simp*]: $(c \mid\in\mid A \mid\cap\mid B) = (c \mid\in\mid A \wedge c \mid\in\mid B)$
by (*rule Int-iff*[*Transfer.transferred*])

lemma *finterI*[*intro*!]: $c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow c \mid\in\mid A \mid\cap\mid B$
by (*rule IntI*[*Transfer.transferred*])

lemma *finterD1*: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow c \mid\in\mid A$
by (*rule IntD1*[*Transfer.transferred*])

lemma *finterD2*: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow c \mid\in\mid B$
by (*rule IntD2*[*Transfer.transferred*])

lemma *finterE*[*elim*!]: $c \mid\in\mid A \mid\cap\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\in\mid B \Longrightarrow P) \Longrightarrow P$
by (*rule IntE*[*Transfer.transferred*])

lemma *funion-iff*[*simp*]: $(c \mid\in\mid A \mid\cup\mid B) = (c \mid\in\mid A \vee c \mid\in\mid B)$

by (*rule Un-iff*[*Transfer.transferred*])

lemma *funionI1*[*elim?*]: $c \mid\in\mid A \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
by (*rule UnI1*[*Transfer.transferred*])

lemma *funionI2*[*elim?*]: $c \mid\in\mid B \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
by (*rule UnI2*[*Transfer.transferred*])

lemma *funionCI*[*intro!*]: $(c \mid\notin\mid B \Longrightarrow c \mid\in\mid A) \Longrightarrow c \mid\in\mid A \mid\cup\mid B$
by (*rule UnCI*[*Transfer.transferred*])

lemma *funionE*[*elim!*]: $c \mid\in\mid A \mid\cup\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow P) \Longrightarrow (c \mid\in\mid B \Longrightarrow P) \Longrightarrow P$
by (*rule UnE*[*Transfer.transferred*])

lemma *fminus-iff*[*simp*]: $(c \mid\in\mid A \mid-\mid B) = (c \mid\in\mid A \wedge c \mid\notin\mid B)$
by (*rule Diff-iff*[*Transfer.transferred*])

lemma *fminusI*[*intro!*]: $c \mid\in\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow c \mid\in\mid A \mid-\mid B$
by (*rule DiffI*[*Transfer.transferred*])

lemma *fminusD1*: $c \mid\in\mid A \mid-\mid B \Longrightarrow c \mid\in\mid A$
by (*rule DiffD1*[*Transfer.transferred*])

lemma *fminusD2*: $c \mid\in\mid A \mid-\mid B \Longrightarrow c \mid\in\mid B \Longrightarrow P$
by (*rule DiffD2*[*Transfer.transferred*])

lemma *fminusE*[*elim!*]: $c \mid\in\mid A \mid-\mid B \Longrightarrow (c \mid\in\mid A \Longrightarrow c \mid\notin\mid B \Longrightarrow P) \Longrightarrow P$
by (*rule DiffE*[*Transfer.transferred*])

lemma *finsert-iff*[*simp*]: $(a \mid\in\mid \mathit{finsert}\ b\ A) = (a = b \vee a \mid\in\mid A)$
by (*rule insert-iff*[*Transfer.transferred*])

lemma *finsertI1*: $a \mid\in\mid \mathit{finsert}\ a\ B$
by (*rule insertI1*[*Transfer.transferred*])

lemma *finsertI2*: $a \mid\in\mid B \Longrightarrow a \mid\in\mid \mathit{finsert}\ b\ B$
by (*rule insertI2*[*Transfer.transferred*])

lemma *finsertE*[*elim!*]: $a \mid\in\mid \mathit{finsert}\ b\ A \Longrightarrow (a = b \Longrightarrow P) \Longrightarrow (a \mid\in\mid A \Longrightarrow P) \Longrightarrow P$
by (*rule insertE*[*Transfer.transferred*])

lemma *finsertCI*[*intro!*]: $(a \mid\notin\mid B \Longrightarrow a = b) \Longrightarrow a \mid\in\mid \mathit{finsert}\ b\ B$
by (*rule insertCI*[*Transfer.transferred*])

lemma *fsubset-finsert-iff*:
$(A \mid\subseteq\mid \mathit{finsert}\ x\ B) = (\mathit{if}\ x \mid\in\mid A\ \mathit{then}\ A \mid-\mid \{\mid x \mid\}\} \mid\subseteq\mid B\ \mathit{else}\ A \mid\subseteq\mid B)$
by (*rule subset-insert-iff*[*Transfer.transferred*])

lemma *finsert-ident*: $x \mid\notin\mid A \Longrightarrow x \mid\notin\mid B \Longrightarrow (\mathit{finsert}\ x\ A = \mathit{finsert}\ x\ B) = (A = B)$
by (*rule insert-ident*[*Transfer.transferred*])

lemma *fsingletonI*[*intro*!,*no-atp*]: $a \mid\in\mid \{\mid a \mid\}$
by (*rule singletonI*[*Transfer.transferred*])

lemma *fsingletonD*[*dest*!,*no-atp*]: $b \mid\in\mid \{\mid a \mid\} \Longrightarrow b = a$
by (*rule singletonD*[*Transfer.transferred*])

lemma *fsingleton-iff*: $(b \mid\in\mid \{\mid a \mid\}) = (b = a)$
by (*rule singleton-iff*[*Transfer.transferred*])

lemma *fsingleton-inject*[*dest*!]: $\{\mid a \mid\} = \{\mid b \mid\} \Longrightarrow a = b$
by (*rule singleton-inject*[*Transfer.transferred*])

lemma *fsingleton-finsert-inj-eq*[*iff*,*no-atp*]: $(\{\mid b \mid\} = \mathit{finsert}\ a\ A) = (a = b \wedge A \mid\subseteq\mid \{\mid b \mid\})$
by (*rule singleton-insert-inj-eq*[*Transfer.transferred*])

lemma *fsingleton-finsert-inj-eq'*[*iff*,*no-atp*]: $(\mathit{finsert}\ a\ A = \{\mid b \mid\}) = (a = b \wedge A \mid\subseteq\mid \{\mid b \mid\})$
by (*rule singleton-insert-inj-eq'*[*Transfer.transferred*])

lemma *fsubset-fsingletonD*: $A \mid\subseteq\mid \{\mid x \mid\} \Longrightarrow A = \{\mid\mid\} \vee A = \{\mid x \mid\}$
by (*rule subset-singletonD*[*Transfer.transferred*])

lemma *fminus-single-finsert*: $A \mid-\mid \{\mid x \mid\} \mid\subseteq\mid B \Longrightarrow A \mid\subseteq\mid \mathit{finsert}\ x\ B$
by (*rule Diff-single-insert*[*Transfer.transferred*])

lemma *fdoubleton-eq-iff*: $(\{\mid a,\ b \mid\} = \{\mid c,\ d \mid\}) = (a = c \wedge b = d \vee a = d \wedge b = c)$
by (*rule doubleton-eq-iff*[*Transfer.transferred*])

lemma *funion-fsingleton-iff*:
$(A \mid\cup\mid B = \{\mid x \mid\}) = (A = \{\mid\mid\} \wedge B = \{\mid x \mid\} \vee A = \{\mid x \mid\} \wedge B = \{\mid\mid\} \vee A = \{\mid x \mid\} \wedge B = \{\mid x \mid\})$
by (*rule Un-singleton-iff*[*Transfer.transferred*])

lemma *fsingleton-funion-iff*:
$(\{\mid x \mid\} = A \mid\cup\mid B) = (A = \{\mid\mid\} \wedge B = \{\mid x \mid\} \vee A = \{\mid x \mid\} \wedge B = \{\mid\mid\} \vee A = \{\mid x \mid\} \wedge B = \{\mid x \mid\})$
by (*rule singleton-Un-iff*[*Transfer.transferred*])

lemma *fimage-eqI*[*simp*, *intro*]: $b = f\ x \Longrightarrow x \mid\in\mid A \Longrightarrow b \mid\in\mid f \mid`\mid A$
by (*rule image-eqI*[*Transfer.transferred*])

lemma *fimageI*: $x \mid\in\mid A \Longrightarrow f\ x \mid\in\mid f \mid`\mid A$

by (*rule imageI*[*Transfer.transferred*])

lemma *rev-fimage-eqI*: $x \mid\in\mid A \Longrightarrow b = f\ x \Longrightarrow b \mid\in\mid f \mid`\mid A$
by (*rule rev-image-eqI*[*Transfer.transferred*])

lemma *fimageE*[*elim*!]: $b \mid\in\mid f \mid`\mid A \Longrightarrow (\bigwedge x.\ b = f\ x \Longrightarrow x \mid\in\mid A \Longrightarrow \mathit{thesis}) \Longrightarrow \mathit{thesis}$
by (*rule imageE*[*Transfer.transferred*])

lemma *Compr-fimage-eq*: $\{x.\ x \mid\in\mid f \mid`\mid A \wedge P\ x\} = f\ `\ \{x.\ x \mid\in\mid A \wedge P\ (f\ x)\}$
by (*rule Compr-image-eq*[*Transfer.transferred*])

lemma *fimage-funion*: $f \mid`\mid (A \mid\cup\mid B) = f \mid`\mid A \mid\cup\mid f \mid`\mid B$
by (*rule image-Un*[*Transfer.transferred*])

lemma *fimage-iff*: $(z \mid\in\mid f \mid`\mid A) = \mathit{fBex}\ A\ (\lambda x.\ z = f\ x)$
by (*rule image-iff*[*Transfer.transferred*])

lemma *fimage-fsubset-iff*[*no-atp*]: $(f \mid`\mid A \mid\subseteq\mid B) = \mathit{fBall}\ A\ (\lambda x.\ f\ x \mid\in\mid B)$
by (*rule image-subset-iff*[*Transfer.transferred*])

lemma *fimage-fsubsetI*: $(\bigwedge x.\ x \mid\in\mid A \Longrightarrow f\ x \mid\in\mid B) \Longrightarrow f \mid`\mid A \mid\subseteq\mid B$
by (*rule image-subsetI*[*Transfer.transferred*])

lemma *fimage-ident*[*simp*]: $(\lambda x.\ x) \mid`\mid Y = Y$
by (*rule image-ident*[*Transfer.transferred*])

lemma *if-split-fmem1*: $((\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y) \mid\in\mid b) = ((Q \longrightarrow x \mid\in\mid b) \wedge (\neg\ Q \longrightarrow y \mid\in\mid b))$
by (*rule if-split-mem1*[*Transfer.transferred*])

lemma *if-split-fmem2*: $(a \mid\in\mid (\mathit{if}\ Q\ \mathit{then}\ x\ \mathit{else}\ y)) = ((Q \longrightarrow a \mid\in\mid x) \wedge (\neg\ Q \longrightarrow a \mid\in\mid y))$
by (*rule if-split-mem2*[*Transfer.transferred*])

lemma *pfsubsetI*[*intro*!,*no-atp*]: $A \mid\subseteq\mid B \Longrightarrow A \neq B \Longrightarrow A \mid\subset\mid B$
by (*rule psubsetI*[*Transfer.transferred*])

lemma *pfsubsetE*[*elim*!,*no-atp*]: $A \mid\subset\mid B \Longrightarrow (A \mid\subseteq\mid B \Longrightarrow \neg\ B \mid\subseteq\mid A \Longrightarrow R) \Longrightarrow R$
by (*rule psubsetE*[*Transfer.transferred*])

lemma *pfsubset-finsert-iff*:
$(A \mid\subset\mid \mathit{finsert}\ x\ B) =$
$(\mathit{if}\ x \mid\in\mid B\ \mathit{then}\ A \mid\subset\mid B\ \mathit{else}\ \mathit{if}\ x \mid\in\mid A\ \mathit{then}\ A \mid-\mid \{\mid x\mid\} \mid\subset\mid B\ \mathit{else}\ A \mid\subseteq\mid B)$
by (*rule psubset-insert-iff*[*Transfer.transferred*])

lemma *pfsubset-eq*: $(A \mid\subset\mid B) = (A \mid\subseteq\mid B \wedge A \neq B)$
by (*rule psubset-eq*[*Transfer.transferred*])

lemma *pfsubset-imp-fsubset*: $A \mid\subset\mid B \Longrightarrow A \mid\subseteq\mid B$
by (*rule psubset-imp-subset*[*Transfer.transferred*])

lemma *pfsubset-trans*: $A \mid\subset\mid B \Longrightarrow B \mid\subset\mid C \Longrightarrow A \mid\subset\mid C$
by (*rule psubset-trans*[*Transfer.transferred*])

lemma *pfsubsetD*: $A \mid\subset\mid B \Longrightarrow c \mid\in\mid A \Longrightarrow c \mid\in\mid B$
by (*rule psubsetD*[*Transfer.transferred*])

lemma *pfsubset-fsubset-trans*: $A \mid\subset\mid B \Longrightarrow B \mid\subseteq\mid C \Longrightarrow A \mid\subset\mid C$
by (*rule psubset-subset-trans*[*Transfer.transferred*])

lemma *fsubset-pfsubset-trans*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subset\mid C \Longrightarrow A \mid\subset\mid C$
by (*rule subset-psubset-trans*[*Transfer.transferred*])

lemma *pfsubset-imp-ex-fmem*: $A \mid\subset\mid B \Longrightarrow \exists\, b.\ b \mid\in\mid B \mid-\mid A$
by (*rule psubset-imp-ex-mem*[*Transfer.transferred*])

lemma *fimage-fPow-mono*: $f \mid`\mid A \mid\subseteq\mid B \Longrightarrow (\mid`\mid)\ f \mid`\mid fPow\ A \mid\subseteq\mid fPow\ B$
by (*rule image-Pow-mono*[*Transfer.transferred*])

lemma *fimage-fPow-surj*: $f \mid`\mid A = B \Longrightarrow (\mid`\mid)\ f \mid`\mid fPow\ A = fPow\ B$
by (*rule image-Pow-surj*[*Transfer.transferred*])

lemma *fsubset-finsertI*: $B \mid\subseteq\mid finsert\ a\ B$
by (*rule subset-insertI*[*Transfer.transferred*])

lemma *fsubset-finsertI2*: $A \mid\subseteq\mid B \Longrightarrow A \mid\subseteq\mid finsert\ b\ B$
by (*rule subset-insertI2*[*Transfer.transferred*])

lemma *fsubset-finsert*: $x \mid\notin\mid A \Longrightarrow (A \mid\subseteq\mid finsert\ x\ B) = (A \mid\subseteq\mid B)$
by (*rule subset-insert*[*Transfer.transferred*])

lemma *funion-upper1*: $A \mid\subseteq\mid A \mid\cup\mid B$
by (*rule Un-upper1*[*Transfer.transferred*])

lemma *funion-upper2*: $B \mid\subseteq\mid A \mid\cup\mid B$
by (*rule Un-upper2*[*Transfer.transferred*])

lemma *funion-least*: $A \mid\subseteq\mid C \Longrightarrow B \mid\subseteq\mid C \Longrightarrow A \mid\cup\mid B \mid\subseteq\mid C$
by (*rule Un-least*[*Transfer.transferred*])

lemma *finter-lower1*: $A \mid\cap\mid B \mid\subseteq\mid A$
by (*rule Int-lower1*[*Transfer.transferred*])

lemma *finter-lower2*: $A \mid\cap\mid B \mid\subseteq\mid B$
by (*rule Int-lower2*[*Transfer.transferred*])

lemma *finter-greatest*: $C \mid\subseteq\mid A \Longrightarrow C \mid\subseteq\mid B \Longrightarrow C \mid\subseteq\mid A \mid\cap\mid B$
by (*rule Int-greatest*[*Transfer.transferred*])

lemma *fminus-fsubset*: $A \mid-\mid B \mid\subseteq\mid A$
by (*rule Diff-subset*[*Transfer.transferred*])

lemma *fminus-fsubset-conv*: $(A \mid-\mid B \mid\subseteq\mid C) = (A \mid\subseteq\mid B \mid\cup\mid C)$
by (*rule Diff-subset-conv*[*Transfer.transferred*])

lemma *fsubset-fempty*[*simp*]: $(A \mid\subseteq\mid \{||\}) = (A = \{||\})$
by (*rule subset-empty*[*Transfer.transferred*])

lemma *not-pfsubset-fempty*[*iff*]: $\neg\ A \mid\subset\mid \{||\}$
by (*rule not-psubset-empty*[*Transfer.transferred*])

lemma *finsert-is-funion*: *finsert* $a\ A = \{|a|\} \mid\cup\mid A$
by (*rule insert-is-Un*[*Transfer.transferred*])

lemma *finsert-not-fempty*[*simp*]: *finsert* $a\ A \neq \{||\}$
by (*rule insert-not-empty*[*Transfer.transferred*])

lemma *fempty-not-finsert*: $\{||\} \neq$ *finsert* $a\ A$
by (*rule empty-not-insert*[*Transfer.transferred*])

lemma *finsert-absorb*: $a \mid\in\mid A \Longrightarrow$ *finsert* $a\ A = A$
by (*rule insert-absorb*[*Transfer.transferred*])

lemma *finsert-absorb2*[*simp*]: *finsert* x (*finsert* $x\ A$) = *finsert* $x\ A$
by (*rule insert-absorb2*[*Transfer.transferred*])

lemma *finsert-commute*: *finsert* x (*finsert* $y\ A$) = *finsert* y (*finsert* $x\ A$)
by (*rule insert-commute*[*Transfer.transferred*])

lemma *finsert-fsubset*[*simp*]: (*finsert* $x\ A \mid\subseteq\mid B) = (x \mid\in\mid B \wedge A \mid\subseteq\mid B)$
by (*rule insert-subset*[*Transfer.transferred*])

lemma *finsert-inter-finsert*[*simp*]: *finsert* $a\ A \mid\cap\mid$ *finsert* $a\ B =$ *finsert* $a\ (A \mid\cap\mid B)$
by (*rule insert-inter-insert*[*Transfer.transferred*])

lemma *finsert-disjoint*[*simp*,*no-atp*]:
$(\textit{finsert}\ a\ A \mid\cap\mid B = \{||\}) = (a \mid\notin\mid B \wedge A \mid\cap\mid B = \{||\})$
$(\{||\} = \textit{finsert}\ a\ A \mid\cap\mid B) = (a \mid\notin\mid B \wedge \{||\} = A \mid\cap\mid B)$
by (*rule insert-disjoint*[*Transfer.transferred*])+

lemma *disjoint-finsert*[*simp*,*no-atp*]:
$(B \mid\cap\mid \textit{finsert}\ a\ A = \{||\}) = (a \mid\notin\mid B \wedge B \mid\cap\mid A = \{||\})$
$(\{||\} = A \mid\cap\mid \textit{finsert}\ b\ B) = (b \mid\notin\mid A \wedge \{||\} = A \mid\cap\mid B)$
by (*rule disjoint-insert*[*Transfer.transferred*])+

lemma *fimage-fempty*[*simp*]: $f\ |`|\ \{||\} = \{||\}$
by (*rule image-empty*[*Transfer.transferred*])

lemma *fimage-finsert*[*simp*]: $f\ |`|\ \mathit{finsert}\ a\ B = \mathit{finsert}\ (f\ a)\ (f\ |`|\ B)$
by (*rule image-insert*[*Transfer.transferred*])

lemma *fimage-constant*: $x\ |\in|\ A \Longrightarrow (\lambda x.\ c)\ |`|\ A = \{|c|\}$
by (*rule image-constant*[*Transfer.transferred*])

lemma *fimage-constant-conv*: $(\lambda x.\ c)\ |`|\ A = (\mathit{if}\ A = \{||\}\ \mathit{then}\ \{||\}\ \mathit{else}\ \{|c|\})$
by (*rule image-constant-conv*[*Transfer.transferred*])

lemma *fimage-fimage*: $f\ |`|\ g\ |`|\ A = (\lambda x.\ f\ (g\ x))\ |`|\ A$
by (*rule image-image*[*Transfer.transferred*])

lemma *finsert-fimage*[*simp*]: $x\ |\in|\ A \Longrightarrow \mathit{finsert}\ (f\ x)\ (f\ |`|\ A) = f\ |`|\ A$
by (*rule insert-image*[*Transfer.transferred*])

lemma *fimage-is-fempty*[*iff*]: $(f\ |`|\ A = \{||\}) = (A = \{||\})$
by (*rule image-is-empty*[*Transfer.transferred*])

lemma *fempty-is-fimage*[*iff*]: $(\{||\} = f\ |`|\ A) = (A = \{||\})$
by (*rule empty-is-image*[*Transfer.transferred*])

lemma *fimage-cong*: $M = N \Longrightarrow (\bigwedge x.\ x\ |\in|\ N \Longrightarrow f\,x = g\,x) \Longrightarrow f\ |`|\ M = g\ |`|\ N$
by (*rule image-cong*[*Transfer.transferred*])

lemma *fimage-finter-fsubset*: $f\ |`|\ (A\ |\cap|\ B)\ |\subseteq|\ f\ |`|\ A\ |\cap|\ f\ |`|\ B$
by (*rule image-Int-subset*[*Transfer.transferred*])

lemma *fimage-fminus-fsubset*: $f\ |`|\ A\ |-|\ f\ |`|\ B\ |\subseteq|\ f\ |`|\ (A\ |-|\ B)$
by (*rule image-diff-subset*[*Transfer.transferred*])

lemma *finter-absorb*: $A\ |\cap|\ A = A$
by (*rule Int-absorb*[*Transfer.transferred*])

lemma *finter-left-absorb*: $A\ |\cap|\ (A\ |\cap|\ B) = A\ |\cap|\ B$
by (*rule Int-left-absorb*[*Transfer.transferred*])

lemma *finter-commute*: $A\ |\cap|\ B = B\ |\cap|\ A$
by (*rule Int-commute*[*Transfer.transferred*])

lemma *finter-left-commute*: $A\ |\cap|\ (B\ |\cap|\ C) = B\ |\cap|\ (A\ |\cap|\ C)$
by (*rule Int-left-commute*[*Transfer.transferred*])

lemma *finter-assoc*: $A\ |\cap|\ B\ |\cap|\ C = A\ |\cap|\ (B\ |\cap|\ C)$
by (*rule Int-assoc*[*Transfer.transferred*])

lemma *finter-ac*:
$A \mid\cap\mid B \mid\cap\mid C = A \mid\cap\mid (B \mid\cap\mid C)$
$A \mid\cap\mid (A \mid\cap\mid B) = A \mid\cap\mid B$
$A \mid\cap\mid B = B \mid\cap\mid A$
$A \mid\cap\mid (B \mid\cap\mid C) = B \mid\cap\mid (A \mid\cap\mid C)$
by (*rule Int-ac*[*Transfer.transferred*])+

lemma *finter-absorb1*: $B \mid\subseteq\mid A \Longrightarrow A \mid\cap\mid B = B$
by (*rule Int-absorb1*[*Transfer.transferred*])

lemma *finter-absorb2*: $A \mid\subseteq\mid B \Longrightarrow A \mid\cap\mid B = A$
by (*rule Int-absorb2*[*Transfer.transferred*])

lemma *finter-fempty-left*: $\{||\} \mid\cap\mid B = \{||\}$
by (*rule Int-empty-left*[*Transfer.transferred*])

lemma *finter-fempty-right*: $A \mid\cap\mid \{||\} = \{||\}$
by (*rule Int-empty-right*[*Transfer.transferred*])

lemma *disjoint-iff-fnot-equal*: $(A \mid\cap\mid B = \{||\}) = fBall\ A\ (\lambda x.\ fBall\ B\ ((\neq)\ x))$
by (*rule disjoint-iff-not-equal*[*Transfer.transferred*])

lemma *finter-funion-distrib*: $A \mid\cap\mid (B \mid\cup\mid C) = A \mid\cap\mid B \mid\cup\mid (A \mid\cap\mid C)$
by (*rule Int-Un-distrib*[*Transfer.transferred*])

lemma *finter-funion-distrib2*: $B \mid\cup\mid C \mid\cap\mid A = B \mid\cap\mid A \mid\cup\mid (C \mid\cap\mid A)$
by (*rule Int-Un-distrib2*[*Transfer.transferred*])

lemma *finter-fsubset-iff*[*no-atp*, *simp*]: $(C \mid\subseteq\mid A \mid\cap\mid B) = (C \mid\subseteq\mid A \wedge C \mid\subseteq\mid B)$
by (*rule Int-subset-iff*[*Transfer.transferred*])

lemma *funion-absorb*: $A \mid\cup\mid A = A$
by (*rule Un-absorb*[*Transfer.transferred*])

lemma *funion-left-absorb*: $A \mid\cup\mid (A \mid\cup\mid B) = A \mid\cup\mid B$
by (*rule Un-left-absorb*[*Transfer.transferred*])

lemma *funion-commute*: $A \mid\cup\mid B = B \mid\cup\mid A$
by (*rule Un-commute*[*Transfer.transferred*])

lemma *funion-left-commute*: $A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$
by (*rule Un-left-commute*[*Transfer.transferred*])

lemma *funion-assoc*: $A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$
by (*rule Un-assoc*[*Transfer.transferred*])

lemma *funion-ac*:
$A \mid\cup\mid B \mid\cup\mid C = A \mid\cup\mid (B \mid\cup\mid C)$
$A \mid\cup\mid (A \mid\cup\mid B) = A \mid\cup\mid B$

$A \mid\cup\mid B = B \mid\cup\mid A$
$A \mid\cup\mid (B \mid\cup\mid C) = B \mid\cup\mid (A \mid\cup\mid C)$
by (*rule Un-ac*[*Transfer.transferred*])+

lemma *funion-absorb1*: $A \mid\subseteq\mid B \Longrightarrow A \mid\cup\mid B = B$
by (*rule Un-absorb1*[*Transfer.transferred*])

lemma *funion-absorb2*: $B \mid\subseteq\mid A \Longrightarrow A \mid\cup\mid B = A$
by (*rule Un-absorb2*[*Transfer.transferred*])

lemma *funion-fempty-left*: $\{||\} \mid\cup\mid B = B$
by (*rule Un-empty-left*[*Transfer.transferred*])

lemma *funion-fempty-right*: $A \mid\cup\mid \{||\} = A$
by (*rule Un-empty-right*[*Transfer.transferred*])

lemma *funion-finsert-left*[*simp*]: *finsert a B* $\mid\cup\mid$ *C* = *finsert a* (*B* $\mid\cup\mid$ *C*)
by (*rule Un-insert-left*[*Transfer.transferred*])

lemma *funion-finsert-right*[*simp*]: *A* $\mid\cup\mid$ *finsert a B* = *finsert a* (*A* $\mid\cup\mid$ *B*)
by (*rule Un-insert-right*[*Transfer.transferred*])

lemma *finter-finsert-left*: *finsert a B* $\mid\cap\mid$ *C* = (*if a* $\mid\in\mid$ *C then finsert a* (*B* $\mid\cap\mid$ *C*) *else B* $\mid\cap\mid$ *C*)
by (*rule Int-insert-left*[*Transfer.transferred*])

lemma *finter-finsert-left-ifffempty*[*simp*]: *a* $\mid\notin\mid$ *C* $\Longrightarrow$ *finsert a B* $\mid\cap\mid$ *C* = *B* $\mid\cap\mid$ *C*
by (*rule Int-insert-left-if0*[*Transfer.transferred*])

lemma *finter-finsert-left-if1*[*simp*]: *a* $\mid\in\mid$ *C* $\Longrightarrow$ *finsert a B* $\mid\cap\mid$ *C* = *finsert a* (*B* $\mid\cap\mid$ *C*)
by (*rule Int-insert-left-if1*[*Transfer.transferred*])

lemma *finter-finsert-right*:
A $\mid\cap\mid$ *finsert a B* = (*if a* $\mid\in\mid$ *A then finsert a* (*A* $\mid\cap\mid$ *B*) *else A* $\mid\cap\mid$ *B*)
by (*rule Int-insert-right*[*Transfer.transferred*])

lemma *finter-finsert-right-ifffempty*[*simp*]: *a* $\mid\notin\mid$ *A* $\Longrightarrow$ *A* $\mid\cap\mid$ *finsert a B* = *A* $\mid\cap\mid$ *B*
by (*rule Int-insert-right-if0*[*Transfer.transferred*])

lemma *finter-finsert-right-if1*[*simp*]: *a* $\mid\in\mid$ *A* $\Longrightarrow$ *A* $\mid\cap\mid$ *finsert a B* = *finsert a* (*A* $\mid\cap\mid$ *B*)
by (*rule Int-insert-right-if1*[*Transfer.transferred*])

lemma *funion-finter-distrib*: *A* $\mid\cup\mid$ (*B* $\mid\cap\mid$ *C*) = *A* $\mid\cup\mid$ *B* $\mid\cap\mid$ (*A* $\mid\cup\mid$ *C*)
by (*rule Un-Int-distrib*[*Transfer.transferred*])

lemma *funion-finter-distrib2*: *B* $\mid\cap\mid$ *C* $\mid\cup\mid$ *A* = *B* $\mid\cup\mid$ *A* $\mid\cap\mid$ (*C* $\mid\cup\mid$ *A*)

by (*rule Un-Int-distrib2*[*Transfer.transferred*])

lemma *funion-finter-crazy*:
$A \mid\cap\mid B \mid\cup\mid (B \mid\cap\mid C) \mid\cup\mid (C \mid\cap\mid A) = A \mid\cup\mid B \mid\cap\mid (B \mid\cup\mid C) \mid\cap\mid (C \mid\cup\mid A)$
by (*rule Un-Int-crazy*[*Transfer.transferred*])

lemma *fsubset-funion-eq*: $(A \mid\subseteq\mid B) = (A \mid\cup\mid B = B)$
by (*rule subset-Un-eq*[*Transfer.transferred*])

lemma *funion-fempty*[*iff*]: $(A \mid\cup\mid B = \{||\}) = (A = \{||\} \wedge B = \{||\})$
by (*rule Un-empty*[*Transfer.transferred*])

lemma *funion-fsubset-iff*[*no-atp, simp*]: $(A \mid\cup\mid B \mid\subseteq\mid C) = (A \mid\subseteq\mid C \wedge B \mid\subseteq\mid C)$
by (*rule Un-subset-iff*[*Transfer.transferred*])

lemma *funion-fminus-finter*: $A \mid-\mid B \mid\cup\mid (A \mid\cap\mid B) = A$
by (*rule Un-Diff-Int*[*Transfer.transferred*])

lemma *ffunion-empty*[*simp*]: *ffUnion* $\{||\} = \{||\}$
by (*rule Union-empty*[*Transfer.transferred*])

lemma *ffunion-mono*: $A \mid\subseteq\mid B \Longrightarrow$ *ffUnion* $A \mid\subseteq\mid$ *ffUnion* B
by (*rule Union-mono*[*Transfer.transferred*])

lemma *ffunion-insert*[*simp*]: *ffUnion* (*finsert* a B) $= a \mid\cup\mid$ *ffUnion* B
by (*rule Union-insert*[*Transfer.transferred*])

lemma *fminus-finter2*: $A \mid\cap\mid C \mid-\mid (B \mid\cap\mid C) = A \mid\cap\mid C \mid-\mid B$
by (*rule Diff-Int2*[*Transfer.transferred*])

lemma *funion-finter-assoc-eq*: $(A \mid\cap\mid B \mid\cup\mid C = A \mid\cap\mid (B \mid\cup\mid C)) = (C \mid\subseteq\mid A)$
by (*rule Un-Int-assoc-eq*[*Transfer.transferred*])

lemma *fBall-funion*: *fBall* $(A \mid\cup\mid B)$ $P =$ (*fBall* A $P \wedge$ *fBall* B P)
by (*rule ball-Un*[*Transfer.transferred*])

lemma *fBex-funion*: *fBex* $(A \mid\cup\mid B)$ $P =$ (*fBex* A $P \vee$ *fBex* B P)
by (*rule bex-Un*[*Transfer.transferred*])

lemma *fminus-eq-fempty-iff*[*simp,no-atp*]: $(A \mid-\mid B = \{||\}) = (A \mid\subseteq\mid B)$
by (*rule Diff-eq-empty-iff*[*Transfer.transferred*])

lemma *fminus-cancel*[*simp*]: $A \mid-\mid A = \{||\}$
by (*rule Diff-cancel*[*Transfer.transferred*])

lemma *fminus-idemp*[*simp*]: $A \mid-\mid B \mid-\mid B = A \mid-\mid B$
by (*rule Diff-idemp*[*Transfer.transferred*])

lemma *fminus-triv*: $A \mid\cap\mid B = \{||\} \Longrightarrow A \mid-\mid B = A$

by (*rule Diff-triv*[*Transfer.transferred*])

lemma *fempty-fminus*[*simp*]: $\{||\} \mid-\mid A = \{||\}$
by (*rule empty-Diff*[*Transfer.transferred*])

lemma *fminus-fempty*[*simp*]: $A \mid-\mid \{||\} = A$
by (*rule Diff-empty*[*Transfer.transferred*])

lemma *fminus-finsertffempty*[*simp*,*no-atp*]: $x \mid\notin\mid A \Longrightarrow A \mid-\mid \mathit{finsert}\ x\ B = A \mid-\mid B$
by (*rule Diff-insert0*[*Transfer.transferred*])

lemma *fminus-finsert*: $A \mid-\mid \mathit{finsert}\ a\ B = A \mid-\mid B \mid-\mid \{|a|\}$
by (*rule Diff-insert*[*Transfer.transferred*])

lemma *fminus-finsert2*: $A \mid-\mid \mathit{finsert}\ a\ B = A \mid-\mid \{|a|\} \mid-\mid B$
by (*rule Diff-insert2*[*Transfer.transferred*])

lemma *finsert-fminus-if*: $\mathit{finsert}\ x\ A \mid-\mid B = (\mathit{if}\ x \mid\in\mid B\ \mathit{then}\ A \mid-\mid B\ \mathit{else}\ \mathit{finsert}\ x\ (A \mid-\mid B))$
by (*rule insert-Diff-if*[*Transfer.transferred*])

lemma *finsert-fminus1*[*simp*]: $x \mid\in\mid B \Longrightarrow \mathit{finsert}\ x\ A \mid-\mid B = A \mid-\mid B$
by (*rule insert-Diff1*[*Transfer.transferred*])

lemma *finsert-fminus-single*[*simp*]: $\mathit{finsert}\ a\ (A \mid-\mid \{|a|\}) = \mathit{finsert}\ a\ A$
by (*rule insert-Diff-single*[*Transfer.transferred*])

lemma *finsert-fminus*: $a \mid\in\mid A \Longrightarrow \mathit{finsert}\ a\ (A \mid-\mid \{|a|\}) = A$
by (*rule insert-Diff*[*Transfer.transferred*])

lemma *fminus-finsert-absorb*: $x \mid\notin\mid A \Longrightarrow \mathit{finsert}\ x\ A \mid-\mid \{|x|\} = A$
by (*rule Diff-insert-absorb*[*Transfer.transferred*])

lemma *fminus-disjoint*[*simp*]: $A \mid\cap\mid (B \mid-\mid A) = \{||\}$
by (*rule Diff-disjoint*[*Transfer.transferred*])

lemma *fminus-partition*: $A \mid\subseteq\mid B \Longrightarrow A \mid\cup\mid (B \mid-\mid A) = B$
by (*rule Diff-partition*[*Transfer.transferred*])

lemma *double-fminus*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid C \Longrightarrow B \mid-\mid (C \mid-\mid A) = A$
by (*rule double-diff*[*Transfer.transferred*])

lemma *funion-fminus-cancel*[*simp*]: $A \mid\cup\mid (B \mid-\mid A) = A \mid\cup\mid B$
by (*rule Un-Diff-cancel*[*Transfer.transferred*])

lemma *funion-fminus-cancel2*[*simp*]: $B \mid-\mid A \mid\cup\mid A = B \mid\cup\mid A$
by (*rule Un-Diff-cancel2*[*Transfer.transferred*])

lemma *fminus-funion*: $A \mid-\mid (B \mid\cup\mid C) = A \mid-\mid B \mid\cap\mid (A \mid-\mid C)$
by (*rule Diff-Un*[*Transfer.transferred*])

lemma *fminus-finter*: $A \mid-\mid (B \mid\cap\mid C) = A \mid-\mid B \mid\cup\mid (A \mid-\mid C)$
by (*rule Diff-Int*[*Transfer.transferred*])

lemma *funion-fminus*: $A \mid\cup\mid B \mid-\mid C = A \mid-\mid C \mid\cup\mid (B \mid-\mid C)$
by (*rule Un-Diff*[*Transfer.transferred*])

lemma *finter-fminus*: $A \mid\cap\mid B \mid-\mid C = A \mid\cap\mid (B \mid-\mid C)$
by (*rule Int-Diff*[*Transfer.transferred*])

lemma *fminus-finter-distrib*: $C \mid\cap\mid (A \mid-\mid B) = C \mid\cap\mid A \mid-\mid (C \mid\cap\mid B)$
by (*rule Diff-Int-distrib*[*Transfer.transferred*])

lemma *fminus-finter-distrib2*: $A \mid-\mid B \mid\cap\mid C = A \mid\cap\mid C \mid-\mid (B \mid\cap\mid C)$
by (*rule Diff-Int-distrib2*[*Transfer.transferred*])

lemma *fUNIV-bool*[*no-atp*]: $fUNIV = \{\mid False,\ True \mid\}$
by (*rule UNIV-bool*[*Transfer.transferred*])

lemma *fPow-fempty*[*simp*]: $fPow\ \{||\} = \{\mid\{||\}\mid\}$
by (*rule Pow-empty*[*Transfer.transferred*])

lemma *fPow-finsert*: $fPow\ (finsert\ a\ A) = fPow\ A \mid\cup\mid finsert\ a \mid`\mid fPow\ A$
by (*rule Pow-insert*[*Transfer.transferred*])

lemma *funion-fPow-fsubset*: $fPow\ A \mid\cup\mid fPow\ B \mid\subseteq\mid fPow\ (A \mid\cup\mid B)$
by (*rule Un-Pow-subset*[*Transfer.transferred*])

lemma *fPow-finter-eq*[*simp*]: $fPow\ (A \mid\cap\mid B) = fPow\ A \mid\cap\mid fPow\ B$
by (*rule Pow-Int-eq*[*Transfer.transferred*])

lemma *fset-eq-fsubset*: $(A = B) = (A \mid\subseteq\mid B \wedge B \mid\subseteq\mid A)$
by (*rule set-eq-subset*[*Transfer.transferred*])

lemma *fsubset-iff*[*no-atp*]: $(A \mid\subseteq\mid B) = (\forall t.\ t \mid\in\mid A \longrightarrow t \mid\in\mid B)$
by (*rule subset-iff*[*Transfer.transferred*])

lemma *fsubset-iff-pfsubset-eq*: $(A \mid\subseteq\mid B) = (A \mid\subset\mid B \vee A = B)$
by (*rule subset-iff-psubset-eq*[*Transfer.transferred*])

lemma *all-not-fin-conv*[*simp*]: $(\forall x.\ x \mid\notin\mid A) = (A = \{||\})$
by (*rule all-not-in-conv*[*Transfer.transferred*])

lemma *ex-fin-conv*: $(\exists x.\ x \mid\in\mid A) = (A \neq \{||\})$
by (*rule ex-in-conv*[*Transfer.transferred*])

lemma *fimage-mono*: $A \mid\subseteq\mid B \Longrightarrow f \mid`\mid A \mid\subseteq\mid f \mid`\mid B$

by (*rule image-mono*[*Transfer.transferred*])

lemma *fPow-mono*: $A \mid\subseteq\mid B \Longrightarrow fPow\ A \mid\subseteq\mid fPow\ B$
by (*rule Pow-mono*[*Transfer.transferred*])

lemma *finsert-mono*: $C \mid\subseteq\mid D \Longrightarrow finsert\ a\ C \mid\subseteq\mid finsert\ a\ D$
by (*rule insert-mono*[*Transfer.transferred*])

lemma *funion-mono*: $A \mid\subseteq\mid C \Longrightarrow B \mid\subseteq\mid D \Longrightarrow A \mid\cup\mid B \mid\subseteq\mid C \mid\cup\mid D$
by (*rule Un-mono*[*Transfer.transferred*])

lemma *finter-mono*: $A \mid\subseteq\mid C \Longrightarrow B \mid\subseteq\mid D \Longrightarrow A \mid\cap\mid B \mid\subseteq\mid C \mid\cap\mid D$
by (*rule Int-mono*[*Transfer.transferred*])

lemma *fminus-mono*: $A \mid\subseteq\mid C \Longrightarrow D \mid\subseteq\mid B \Longrightarrow A \mid-\mid B \mid\subseteq\mid C \mid-\mid D$
by (*rule Diff-mono*[*Transfer.transferred*])

lemma *fin-mono*: $A \mid\subseteq\mid B \Longrightarrow x \mid\in\mid A \longrightarrow x \mid\in\mid B$
by (*rule in-mono*[*Transfer.transferred*])

lemma *fthe-felem-eq*[*simp*]: $fthe\text{-}elem\ \{|x|\} = x$
by (*rule the-elem-eq*[*Transfer.transferred*])

lemma *fLeast-mono*:
$mono\ f \Longrightarrow fBex\ S\ (\lambda x.\ fBall\ S\ ((\leq)\ x)) \Longrightarrow (LEAST\ y.\ y \mid\in\mid f \mid`\mid S) = f\ (LEAST\ x.\ x \mid\in\mid S)$
by (*rule Least-mono*[*Transfer.transferred*])

lemma *fbind-fbind*: $fbind\ (fbind\ A\ B)\ C = fbind\ A\ (\lambda x.\ fbind\ (B\ x)\ C)$
by (*rule Set.bind-bind*[*Transfer.transferred*])

lemma *fempty-fbind*[*simp*]: $fbind\ \{||\}\ f = \{||\}$
by (*rule empty-bind*[*Transfer.transferred*])

lemma *nonfempty-fbind-const*: $A \neq \{||\} \Longrightarrow fbind\ A\ (\lambda\text{-}.\ B) = B$
by (*rule nonempty-bind-const*[*Transfer.transferred*])

lemma *fbind-const*: $fbind\ A\ (\lambda\text{-}.\ B) = (if\ A = \{||\}\ then\ \{||\}\ else\ B)$
by (*rule bind-const*[*Transfer.transferred*])

lemma *ffmember-filter*[*simp*]: $(x \mid\in\mid ffilter\ P\ A) = (x \mid\in\mid A \wedge P\ x)$
by *transfer simp*

lemma *fequalityI*: $A \mid\subseteq\mid B \Longrightarrow B \mid\subseteq\mid A \Longrightarrow A = B$
by (*rule equalityI*[*Transfer.transferred*])

lemma *fset-of-list-simps*[*simp*]:
$fset\text{-}of\text{-}list\ [] = \{||\}$
$fset\text{-}of\text{-}list\ (x21\ \#\ x22) = finsert\ x21\ (fset\text{-}of\text{-}list\ x22)$

by (*rule set-simps*[*Transfer.transferred*])+

lemma *fset-of-list-append*[*simp*]: *fset-of-list* (*xs* @ *ys*) = *fset-of-list xs* |∪| *fset-of-list ys*
by (*rule set-append*[*Transfer.transferred*])

lemma *fset-of-list-rev*[*simp*]: *fset-of-list* (*rev xs*) = *fset-of-list xs*
by (*rule set-rev*[*Transfer.transferred*])

lemma *fset-of-list-map*[*simp*]: *fset-of-list* (*map f xs*) = *f* |‘| *fset-of-list xs*
by (*rule set-map*[*Transfer.transferred*])

29.5 Additional lemmas

29.5.1 *ffUnion*

lemma *ffUnion-funion-distrib*[*simp*]: *ffUnion* (*A* |∪| *B*) = *ffUnion A* |∪| *ffUnion B*
by (*rule Union-Un-distrib*[*Transfer.transferred*])

29.5.2 *fbind*

lemma *fbind-cong*[*fundef-cong*]: *A* = *B* ⟹ (⋀*x*. *x* |∈| *B* ⟹ *f x* = *g x*) ⟹ *fbind A f* = *fbind B g*
by *transfer force*

29.5.3 *fsingleton*

lemma *fsingletonE*: *b* |∈| {|*a*|} ⟹ (*b* = *a* ⟹ *thesis*) ⟹ *thesis*
by (*rule fsingletonD* [*elim-format*])

29.5.4 *femepty*

lemma *fempty-ffilter*[*simp*]: *ffilter* (λ-. *False*) *A* = {||}
by *transfer auto*

lemma *femptyE* [*elim*!]: *a* |∈| {||} ⟹ *P*
by *simp*

29.5.5 *fset*

lemma *fset-simps*[*simp*]:
fset {||} = {}
fset (*finsert x X*) = *insert x* (*fset X*)
by (*rule bot-fset.rep-eq finsert.rep-eq*)+

lemma *finite-fset* [*simp*]:
shows *finite* (*fset S*)
by *transfer simp*

lemmas *fset-cong = fset-inject*

lemma *filter-fset* [*simp*]:
shows *fset* (*ffilter P xs*) = *Collect P* ∩ *fset xs*
by *transfer auto*

lemma *inter-fset*[*simp*]: *fset* (*A* |∩| *B*) = *fset A* ∩ *fset B*
by (*rule inf-fset.rep-eq*)

lemma *union-fset*[*simp*]: *fset* (*A* |∪| *B*) = *fset A* ∪ *fset B*
by (*rule sup-fset.rep-eq*)

lemma *minus-fset*[*simp*]: *fset* (*A* |−| *B*) = *fset A* − *fset B*
by (*rule minus-fset.rep-eq*)

29.5.6 *ffilter*

lemma *subset-ffilter*:
ffilter P A |⊆| *ffilter Q A* = (∀ *x*. *x* |∈| *A* ⟶ *P x* ⟶ *Q x*)
by *transfer auto*

lemma *eq-ffilter*:
(*ffilter P A* = *ffilter Q A*) = (∀ *x*. *x* |∈| *A* ⟶ *P x* = *Q x*)
by *transfer auto*

lemma *pfsubset-ffilter*:
(⋀*x*. *x* |∈| *A* ⟹ *P x* ⟹ *Q x*) ⟹ (*x* |∈| *A* ∧ ¬ *P x* ∧ *Q x*) ⟹
ffilter P A |⊂| *ffilter Q A*
unfolding *less-fset-def* **by** (*auto simp add*: *subset-ffilter eq-ffilter*)

29.5.7 *fset-of-list*

lemma *fset-of-list-filter*[*simp*]:
fset-of-list (*filter P xs*) = *ffilter P* (*fset-of-list xs*)
by *transfer auto*

lemma *fset-of-list-subset*[*intro*]:
set xs ⊆ *set ys* ⟹ *fset-of-list xs* |⊆| *fset-of-list ys*
by *transfer simp*

lemma *fset-of-list-elem*: (*x* |∈| *fset-of-list xs*) ⟷ (*x* ∈ *set xs*)
by *transfer simp*

29.5.8 *finsert*

lemma *set-finsert*:
assumes *x* |∈| *A*
obtains *B* **where** *A* = *finsert x B* **and** *x* |∉| *B*
using *assms* **by** *transfer* (*metis Set.set-insert finite-insert*)

lemma *mk-disjoint-finsert*: $a \mid\in\mid A \Longrightarrow \exists B.\ A = \mathit{finsert}\ a\ B \wedge a \mid\notin\mid B$
by (*rule exI* [**where** $x = A \mid-\mid \{\mid a \mid\}$]) *blast*

lemma *finsert-eq-iff*:
assumes $a \mid\notin\mid A$ **and** $b \mid\notin\mid B$
shows (*finsert a A = finsert b B*) =
(*if a = b then A = B else* $\exists C.\ A = \mathit{finsert}\ b\ C \wedge b \mid\notin\mid C \wedge B = \mathit{finsert}\ a\ C \wedge a \mid\notin\mid C$)
using *assms* **by** *transfer* (*force simp*: *insert-eq-iff*)

29.5.9 *fimage*

lemma *subset-fimage-iff*: $(B \mid\subseteq\mid f \mid`\mid A) = (\exists\ AA.\ AA \mid\subseteq\mid A \wedge B = f \mid`\mid AA)$
by *transfer* (*metis mem-Collect-eq rev-finite-subset subset-image-iff*)

lemma *fimage-strict-mono*:
assumes *inj-on f* (*fset B*) **and** $A \mid\subset\mid B$
shows $f \mid`\mid A \mid\subset\mid f \mid`\mid B$
— TODO: Configure transfer framework to lift $[\![\mathit{inj\text{-}on}\ ?f\ ?B;\ ?A \subset ?B]\!] \Longrightarrow ?f\ `\ ?A \subset ?f\ `\ ?B$.
proof (*rule pfsubsetI*)
from ‹$A \mid\subset\mid B$› **have** $A \mid\subseteq\mid B$
by (*rule pfsubset-imp-fsubset*)
thus $f \mid`\mid A \mid\subseteq\mid f \mid`\mid B$
by (*rule fimage-mono*)
next
from ‹$A \mid\subset\mid B$› **have** $A \mid\subseteq\mid B$ **and** $A \neq B$
by (*simp-all add*: *pfsubset-eq*)

have *fset A* $\neq$ *fset B*
using ‹$A \neq B$›
by (*simp add*: *fset-cong*)
hence *f ` fset A* $\neq$ *f ` fset B*
using ‹$A \mid\subseteq\mid B$›
by (*simp add*: *inj-on-image-eq-iff*[*OF* ‹*inj-on f* (*fset B*)›] *less-eq-fset.rep-eq*)
hence *fset* $(f \mid`\mid A) \neq$ *fset* $(f \mid`\mid B)$
by (*simp add*: *fimage.rep-eq*)
thus $f \mid`\mid A \neq f \mid`\mid B$
by (*simp add*: *fset-cong*)
qed

29.5.10 bounded quantification

lemma *bex-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.$ *fBex A* $(\lambda x.\ P\ x \wedge Q) = (\mathit{fBex}\ A\ P \wedge Q)$
$\bigwedge A\ P\ Q.$ *fBex A* $(\lambda x.\ P \wedge Q\ x) = (P \wedge \mathit{fBex}\ A\ Q)$
$\bigwedge P.$ *fBex* $\{\mid\mid\}$ $P =$ *False*
$\bigwedge a\ B\ P.$ *fBex* (*finsert a B*) $P = (P\ a \vee \mathit{fBex}\ B\ P)$
$\bigwedge A\ P\ f.$ *fBex* $(f \mid`\mid A)$ $P =$ *fBex A* $(\lambda x.\ P\ (f\ x))$

$\bigwedge A\ P.\ (\neg\ fBex\ A\ P) = fBall\ A\ (\lambda x.\ \neg\ P\ x)$
by *auto*

lemma *ball-simps* [*simp*, *no-atp*]:
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P\ x \vee Q) = (fBall\ A\ P \vee Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P \vee Q\ x) = (P \vee fBall\ A\ Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P \longrightarrow Q\ x) = (P \longrightarrow fBall\ A\ Q)$
$\bigwedge A\ P\ Q.\ fBall\ A\ (\lambda x.\ P\ x \longrightarrow Q) = (fBex\ A\ P \longrightarrow Q)$
$\bigwedge P.\ fBall\ \{||\}\ P = True$
$\bigwedge a\ B\ P.\ fBall\ (finsert\ a\ B)\ P = (P\ a \wedge fBall\ B\ P)$
$\bigwedge A\ P\ f.\ fBall\ (f\ |`|\ A)\ P = fBall\ A\ (\lambda x.\ P\ (f\ x))$
$\bigwedge A\ P.\ (\neg\ fBall\ A\ P) = fBex\ A\ (\lambda x.\ \neg\ P\ x)$
by *auto*

lemma *atomize-fBall*:
$(\bigwedge x.\ x\ |\in|\ A \Longrightarrow P\ x) \equiv Trueprop\ (fBall\ A\ (\lambda x.\ P\ x))$
by (*simp add*: *Set.atomize-ball*)

lemma *fBall-mono*[*mono*]: $P \leq Q \Longrightarrow fBall\ S\ P \leq fBall\ S\ Q$
by *auto*

lemma *fBex-mono*[*mono*]: $P \leq Q \Longrightarrow fBex\ S\ P \leq fBex\ S\ Q$
by *auto*

end

29.5.11 *fcard*

lemma *fcard-fempty*:
$fcard\ \{||\} = 0$
by *transfer* (*rule card.empty*)

lemma *fcard-finsert-disjoint*:
$x\ |\notin|\ A \Longrightarrow fcard\ (finsert\ x\ A) = Suc\ (fcard\ A)$
by *transfer* (*rule card-insert-disjoint*)

lemma *fcard-finsert-if*:
$fcard\ (finsert\ x\ A) = (if\ x\ |\in|\ A\ then\ fcard\ A\ else\ Suc\ (fcard\ A))$
by *transfer* (*rule card-insert-if*)

lemma *fcard-0-eq* [*simp*, *no-atp*]:
$fcard\ A = 0 \longleftrightarrow A = \{||\}$
by *transfer* (*rule card-0-eq*)

lemma *fcard-Suc-fminus1*:
$x\ |\in|\ A \Longrightarrow Suc\ (fcard\ (A\ |-|\ \{|x|\})) = fcard\ A$
by *transfer* (*rule card-Suc-Diff1*)

lemma *fcard-fminus-fsingleton*:

$x \mid\in\mid A \Longrightarrow fcard\ (A \mid-\mid \{\mid x \mid\}) = fcard\ A - 1$
by *transfer* (*rule card-Diff-singleton*)

lemma *fcard-fminus-fsingleton-if*:
$fcard\ (A \mid-\mid \{\mid x \mid\}) = (if\ x \mid\in\mid A\ then\ fcard\ A - 1\ else\ fcard\ A)$
by *transfer* (*rule card-Diff-singleton-if*)

lemma *fcard-fminus-finsert*[*simp*]:
assumes $a \mid\in\mid A$ **and** $a \mid\notin\mid B$
shows $fcard\ (A \mid-\mid finsert\ a\ B) = fcard\ (A \mid-\mid B) - 1$
using *assms* **by** *transfer* (*rule card-Diff-insert*)

lemma *fcard-finsert*: $fcard\ (finsert\ x\ A) = Suc\ (fcard\ (A \mid-\mid \{\mid x \mid\}))$
by *transfer* (*rule card.insert-remove*)

lemma *fcard-finsert-le*: $fcard\ A \leq fcard\ (finsert\ x\ A)$
by *transfer* (*rule card-insert-le*)

lemma *fcard-mono*:
$A \mid\subseteq\mid B \Longrightarrow fcard\ A \leq fcard\ B$
by *transfer* (*rule card-mono*)

lemma *fcard-seteq*: $A \mid\subseteq\mid B \Longrightarrow fcard\ B \leq fcard\ A \Longrightarrow A = B$
by *transfer* (*rule card-seteq*)

lemma *pfsubset-fcard-mono*: $A \mid\subset\mid B \Longrightarrow fcard\ A < fcard\ B$
by *transfer* (*rule psubset-card-mono*)

lemma *fcard-funion-finter*:
$fcard\ A + fcard\ B = fcard\ (A \mid\cup\mid B) + fcard\ (A \mid\cap\mid B)$
by *transfer* (*rule card-Un-Int*)

lemma *fcard-funion-disjoint*:
$A \mid\cap\mid B = \{\mid\mid\} \Longrightarrow fcard\ (A \mid\cup\mid B) = fcard\ A + fcard\ B$
by *transfer* (*rule card-Un-disjoint*)

lemma *fcard-funion-fsubset*:
$B \mid\subseteq\mid A \Longrightarrow fcard\ (A \mid-\mid B) = fcard\ A - fcard\ B$
by *transfer* (*rule card-Diff-subset*)

lemma *diff-fcard-le-fcard-fminus*:
$fcard\ A - fcard\ B \leq fcard(A \mid-\mid B)$
by *transfer* (*rule diff-card-le-card-Diff*)

lemma *fcard-fminus1-less*: $x \mid\in\mid A \Longrightarrow fcard\ (A \mid-\mid \{\mid x \mid\}) < fcard\ A$
by *transfer* (*rule card-Diff1-less*)

lemma *fcard-fminus2-less*:
$x \mid\in\mid A \Longrightarrow y \mid\in\mid A \Longrightarrow fcard\ (A \mid-\mid \{\mid x \mid\} \mid-\mid \{\mid y \mid\}) < fcard\ A$

by *transfer* (*rule card-Diff2-less*)

lemma *fcard-fminus1-le*: *fcard* (*A* |−| {|*x*|}) ≤ *fcard A*
by *transfer* (*rule card-Diff1-le*)

lemma *fcard-pfsubset*: *A* |⊆| *B* ⟹ *fcard A* < *fcard B* ⟹ *A* < *B*
by *transfer* (*rule card-psubset*)

29.5.12 *sorted-list-of-fset*

lemma *sorted-list-of-fset-simps*[*simp*]:
set (*sorted-list-of-fset S*) = *fset S*
fset-of-list (*sorted-list-of-fset S*) = *S*
by (*transfer*, *simp*)+

29.5.13 *ffold*

context *comp-fun-commute*
begin
lemma *ffold-empty*[*simp*]: *ffold f z* {||} = *z*
by (*rule fold-empty*[*Transfer.transferred*])

lemma *ffold-finsert* [*simp*]:
assumes *x* |∉| *A*
shows *ffold f z* (*finsert x A*) = *f x* (*ffold f z A*)
using *assms* **by** (*transfer fixing*: *f*) (*rule fold-insert*)

lemma *ffold-fun-left-comm*:
f x (*ffold f z A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-fun-left-comm*)

lemma *ffold-finsert2*:
x |∉| *A* ⟹ *ffold f z* (*finsert x A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-insert2*)

lemma *ffold-rec*:
assumes *x* |∈| *A*
shows *ffold f z A* = *f x* (*ffold f z* (*A* |−| {|*x*|}))
using *assms* **by** (*transfer fixing*: *f*) (*rule fold-rec*)

lemma *ffold-finsert-fremove*:
ffold f z (*finsert x A*) = *f x* (*ffold f z* (*A* |−| {|*x*|}))
by (*transfer fixing*: *f*) (*rule fold-insert-remove*)
end

lemma *ffold-fimage*:
assumes *inj-on g* (*fset A*)
shows *ffold f z* (*g* |‘| *A*) = *ffold* (*f* ∘ *g*) *z A*
using *assms* **by** *transfer′* (*rule fold-image*)

lemma *ffold-cong*:
assumes *comp-fun-commute f comp-fun-commute g*
$\bigwedge x.\ x \mid\in\mid A \Longrightarrow f\ x = g\ x$
and $s = t$ **and** $A = B$
shows *ffold f s A = ffold g t B*
using *assms*[*unfolded comp-fun-commute-def'*]
by *transfer* (*meson Finite-Set.fold-cong subset-UNIV*)

context *comp-fun-idem*
begin

lemma *ffold-finsert-idem*:
ffold f z (*finsert x A*) = *f x* (*ffold f z A*)
by (*transfer fixing*: *f*) (*rule fold-insert-idem*)

declare *ffold-finsert* [*simp del*] *ffold-finsert-idem* [*simp*]

lemma *ffold-finsert-idem2*:
ffold f z (*finsert x A*) = *ffold f* (*f x z*) *A*
by (*transfer fixing*: *f*) (*rule fold-insert-idem2*)

end

29.5.14 $(\mid\subset\mid)$

lemma *wfP-pfsubset*: *wfP* $(\mid\subset\mid)$
proof (*rule wfp-if-convertible-to-nat*)
show $\bigwedge x\ y.\ x \mid\subset\mid y \Longrightarrow$ *fcard x* < *fcard y*
by (*rule pfsubset-fcard-mono*)
qed

29.5.15 Group operations

locale *comm-monoid-fset* = *comm-monoid*
begin

sublocale *set*: *comm-monoid-set* **..**

lift-definition *F* :: ('*b* ⇒ '*a*) ⇒ '*b fset* ⇒ '*a* **is** *set.F* **.**

lemma *cong*[*fundef-cong*]: $A = B \Longrightarrow (\bigwedge x.\ x \mid\in\mid B \Longrightarrow g\ x = h\ x) \Longrightarrow F\ g\ A = F\ h\ B$
by (*rule set.cong*[*Transfer.transferred*])

lemma *cong-simp*[*cong*]:
$[\![\ A = B;\ \bigwedge x.\ x \mid\in\mid B =simp=> g\ x = h\ x\ ]\!] \Longrightarrow F\ g\ A = F\ h\ B$
unfolding *simp-implies-def* **by** (*auto cong*: *cong*)

end

context *comm-monoid-add* **begin**

sublocale *fsum*: *comm-monoid-fset plus 0*
  rewrites *comm-monoid-set.F plus 0 = sum*
  defines *fsum = fsum.F*
proof −
  show *comm-monoid-fset* (+) *0* **by** *standard*

  show *comm-monoid-set.F* (+) *0 = sum* **unfolding** *sum-def* **..**
qed

end

29.5.16 Semilattice operations

locale *semilattice-fset = semilattice*
begin

sublocale *set*: *semilattice-set* **..**

lift-definition *F* :: *'a fset* ⇒ *'a* **is** *set.F* **.**

lemma *eq-fold*: *F* (*finsert x A*) = *ffold f x A*
  by *transfer* (*rule set.eq-fold*)

lemma *singleton* [*simp*]: *F* {|*x*|} = *x*
  by *transfer* (*rule set.singleton*)

lemma *insert-not-elem*: *x* |∉| *A* ⟹ *A* ≠ {||} ⟹ *F* (*finsert x A*) = *x* ∗ *F A*
  by *transfer* (*rule set.insert-not-elem*)

lemma *in-idem*: *x* |∈| *A* ⟹ *x* ∗ *F A* = *F A*
  by *transfer* (*rule set.in-idem*)

lemma *insert* [*simp*]: *A* ≠ {||} ⟹ *F* (*finsert x A*) = *x* ∗ *F A*
  by *transfer* (*rule set.insert*)

end

locale *semilattice-order-fset = binary?*: *semilattice-order* + *semilattice-fset*
begin

end

context *linorder* **begin**

sublocale *fMin*: *semilattice-order-fset min less-eq less*
  rewrites *semilattice-set.F min = Min*

defines *fMin* = *fMin.F*
proof −
show *semilattice-order-fset min* (≤) (<) **by** *standard*

show *semilattice-set.F min* = *Min* **unfolding** *Min-def* **..**
qed

sublocale *fMax*: *semilattice-order-fset max greater-eq greater*
rewrites *semilattice-set.F max* = *Max*
defines *fMax* = *fMax.F*
proof −
show *semilattice-order-fset max* (≥) (>)
by *standard*

show *semilattice-set.F max* = *Max*
unfolding *Max-def* **..**
qed

end

lemma *mono-fMax-commute*: *mono f* ⟹ *A* ≠ {||} ⟹ *f* (*fMax A*) = *fMax* (*f* |‘| *A*)
by *transfer* (*rule mono-Max-commute*)

lemma *mono-fMin-commute*: *mono f* ⟹ *A* ≠ {||} ⟹ *f* (*fMin A*) = *fMin* (*f* |‘| *A*)
by *transfer* (*rule mono-Min-commute*)

lemma *fMax-in*[*simp*]: *A* ≠ {||} ⟹ *fMax A* |∈| *A*
by *transfer* (*rule Max-in*)

lemma *fMin-in*[*simp*]: *A* ≠ {||} ⟹ *fMin A* |∈| *A*
by *transfer* (*rule Min-in*)

lemma *fMax-ge*[*simp*]: *x* |∈| *A* ⟹ *x* ≤ *fMax A*
by *transfer* (*rule Max-ge*)

lemma *fMin-le*[*simp*]: *x* |∈| *A* ⟹ *fMin A* ≤ *x*
by *transfer* (*rule Min-le*)

lemma *fMax-eqI*: (⋀*y*. *y* |∈| *A* ⟹ *y* ≤ *x*) ⟹ *x* |∈| *A* ⟹ *fMax A* = *x*
by *transfer* (*rule Max-eqI*)

lemma *fMin-eqI*: (⋀*y*. *y* |∈| *A* ⟹ *x* ≤ *y*) ⟹ *x* |∈| *A* ⟹ *fMin A* = *x*
by *transfer* (*rule Min-eqI*)

lemma *fMax-finsert*[*simp*]: *fMax* (*finsert x A*) = (*if A* = {||} *then x else max x* (*fMax A*))
by *transfer simp*

lemma *fMin-finsert*[*simp*]: *fMin* (*finsert x A*) = (*if A* = {||} *then x else min x* (*fMin A*))
by *transfer simp*

context *linorder* **begin**

lemma *fset-linorder-max-induct*[*case-names fempty finsert*]:
assumes *P* {||}
and ⋀*x S*. ⟦∀ *y*. *y* |∈| *S* ⟶ *y* < *x*; *P S*⟧ ⟹ *P* (*finsert x S*)
shows *P S*
proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** (*transfer fixing*: *less*) (*auto intro*: *finite-linorder-max-induct*)
qed

lemma *fset-linorder-min-induct*[*case-names fempty finsert*]:
assumes *P* {||}
and ⋀*x S*. ⟦∀ *y*. *y* |∈| *S* ⟶ *y* > *x*; *P S*⟧ ⟹ *P* (*finsert x S*)
shows *P S*
proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** (*transfer fixing*: *less*) (*auto intro*: *finite-linorder-min-induct*)
qed

end

29.6 Choice in fsets

lemma *fset-choice*:
assumes ∀ *x*. *x* |∈| *A* ⟶ (∃ *y*. *P x y*)
shows ∃ *f*. ∀ *x*. *x* |∈| *A* ⟶ *P x* (*f x*)
using *assms* **by** *transfer metis*

29.7 Induction and Cases rules for fsets

lemma *fset-exhaust* [*case-names empty insert*, *cases type*: *fset*]:
assumes *fempty-case*: *S* = {||} ⟹ *P*
and *finsert-case*: ⋀*x S′*. *S* = *finsert x S′* ⟹ *P*
shows *P*
using *assms* **by** *transfer blast*

lemma *fset-induct* [*case-names empty insert*]:
assumes *fempty-case*: *P* {||}
and *finsert-case*: ⋀*x S*. *P S* ⟹ *P* (*finsert x S*)
shows *P S*

proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** *transfer* (*auto intro*: *finite-induct*)
qed

lemma *fset-induct-stronger* [*case-names empty insert*, *induct type*: *fset*]:
assumes *empty-fset-case*: $P\ \{||\}$
and *insert-fset-case*: $\bigwedge x\ S.\ [\![x\ |\notin|\ S;\ P\ S]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ S)$
shows $P\ S$
proof −

note *Domainp-forall-transfer*[*transfer-rule*]
show *?thesis*
using *assms* **by** *transfer* (*auto intro*: *finite-induct*)
qed

lemma *fset-card-induct*:
assumes *empty-fset-case*: $P\ \{||\}$
and *card-fset-Suc-case*: $\bigwedge S\ T.\ \mathit{Suc}\ (\mathit{fcard}\ S) = (\mathit{fcard}\ T) \Longrightarrow P\ S \Longrightarrow P\ T$
shows $P\ S$
proof (*induct S*)
case *empty*
show $P\ \{||\}$ **by** (*rule empty-fset-case*)
next
case (*insert x S*)
have *h*: $P\ S$ **by** *fact*
have $x\ |\notin|\ S$ **by** *fact*
then have $\mathit{Suc}\ (\mathit{fcard}\ S) = \mathit{fcard}\ (\mathit{finsert}\ x\ S)$
by *transfer auto*
then show $P\ (\mathit{finsert}\ x\ S)$
using *h card-fset-Suc-case* **by** *simp*
qed

lemma *fset-strong-cases*:
obtains $xs = \{||\}$
| *ys x* **where** $x\ |\notin|\ ys$ **and** $xs = \mathit{finsert}\ x\ ys$
by *auto*

lemma *fset-induct2*:
$P\ \{||\}\ \{||\} \Longrightarrow$
$(\bigwedge x\ xs.\ x\ |\notin|\ xs \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ \{||\}) \Longrightarrow$
$(\bigwedge y\ ys.\ y\ |\notin|\ ys \Longrightarrow P\ \{||\}\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$(\bigwedge x\ xs\ y\ ys.\ [\![P\ xs\ ys;\ x\ |\notin|\ xs;\ y\ |\notin|\ ys]\!] \Longrightarrow P\ (\mathit{finsert}\ x\ xs)\ (\mathit{finsert}\ y\ ys)) \Longrightarrow$
$P\ xsa\ ysa$
by (*induct xsa arbitrary*: *ysa*; *metis fset-induct-stronger*)

29.8 Lemmas depending on induction

lemma *ffUnion-fsubset-iff*: *ffUnion A* $|\subseteq|$ *B* $\longleftrightarrow$ *fBall A* ($\lambda x.$ *x* $|\subseteq|$ *B*)
by (*induction A*) *simp-all*

29.9 Setup for Lifting/Transfer

29.9.1 Relator and predicator properties

lift-definition *rel-fset* :: ('*a* $\Rightarrow$ '*b* $\Rightarrow$ *bool*) $\Rightarrow$ '*a fset* $\Rightarrow$ '*b fset* $\Rightarrow$ *bool* **is** *rel-set*
parametric *rel-set-transfer* **.**

lemma *rel-fset-alt-def*: *rel-fset R* = (λA *B*. ($\forall$ *x*.$\exists$ *y*. *x*$|\in|$*A* $\longrightarrow$ *y*$|\in|$*B* $\wedge$ *R x y*)
$\wedge$ ($\forall$ *y*. $\exists$ *x*. *y*$|\in|$*B* $\longrightarrow$ *x*$|\in|$*A* $\wedge$ *R x y*))
by *transfer'* (*metis* (*no-types*, *opaque-lifting*) *rel-set-def*)

lemma *finite-rel-set*:
assumes *fin*: *finite X finite Z*
assumes *R-S*: *rel-set* (*R OO S*) *X Z*
shows $\exists$ *Y*. *finite Y* $\wedge$ *rel-set R X Y* $\wedge$ *rel-set S Y Z*
proof −
obtain *f g* **where** *f*: $\forall$ *x*$\in$*X*. *R x* (*f x*) $\wedge$ ($\exists$ *z*$\in$*Z*. *S* (*f x*) *z*)
and *g*: $\forall$ *z*$\in$*Z*. *S* (*g z*) *z* $\wedge$ ($\exists$ *x*$\in$*X*. *R x* (*g z*))
using *R-S*[*unfolded rel-set-def OO-def*] **by** *metis*

let *?Y* = *f* ‘ *X* $\cup$ *g* ‘ *Z*
have *finite ?Y* **by** (*simp add*: *fin*)
moreover have *rel-set R X ?Y*
unfolding *rel-set-def*
using *f g* **by** *clarsimp blast*
moreover have *rel-set S ?Y Z*
unfolding *rel-set-def*
using *f g* **by** *clarsimp blast*
ultimately show *?thesis* **by** *metis*
qed

29.9.2 Transfer rules for the Transfer package

Unconditional transfer rules

context includes *lifting-syntax*
begin

lemma *fempty-transfer* [*transfer-rule*]:
rel-fset A {||} {||}
by (*rule empty-transfer*[*Transfer.transferred*])

lemma *finsert-transfer* [*transfer-rule*]:
(*A* ===> *rel-fset A* ===> *rel-fset A*) *finsert finsert*
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *funion-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *funion funion*
unfolding *rel-fun-def rel-fset-alt-def* **by** *blast*

lemma *ffUnion-transfer* [*transfer-rule*]:
(*rel-fset* (*rel-fset A*) ===> *rel-fset A*) *ffUnion ffUnion*
unfolding *rel-fun-def rel-fset-alt-def* **by** *transfer* (*simp*, *fast*)

lemma *fimage-transfer* [*transfer-rule*]:
((*A* ===> *B*) ===> *rel-fset A* ===> *rel-fset B*) *fimage fimage*
unfolding *rel-fun-def rel-fset-alt-def* **by** *simp blast*

lemma *fBall-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> (=)) ===> (=)) *fBall fBall*
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fBex-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> (=)) ===> (=)) *fBex fBex*
unfolding *rel-fset-alt-def rel-fun-def* **by** *blast*

lemma *fPow-transfer* [*transfer-rule*]:
(*rel-fset A* ===> *rel-fset* (*rel-fset A*)) *fPow fPow*
unfolding *rel-fun-def*
using *Pow-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*]
by *blast*

lemma *rel-fset-transfer* [*transfer-rule*]:
((*A* ===> *B* ===> (=)) ===> *rel-fset A* ===> *rel-fset B* ===> (=))
rel-fset rel-fset
unfolding *rel-fun-def*
using *rel-set-transfer*[*unfolded rel-fun-def*,*rule-format*, *Transfer.transferred*, **where** *A* = *A* **and** *B* = *B*]
by *simp*

lemma *bind-transfer* [*transfer-rule*]:
(*rel-fset A* ===> (*A* ===> *rel-fset B*) ===> *rel-fset B*) *fbind fbind*
unfolding *rel-fun-def*
using *bind-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

Rules requiring bi-unique, bi-total or right-total relations

lemma *fmember-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*A* ===> *rel-fset A* ===> (=)) (|∈|) (|∈|)
using *assms* **unfolding** *rel-fun-def rel-fset-alt-def bi-unique-def* **by** *metis*

lemma *finter-transfer* [*transfer-rule*]:
assumes *bi-unique A*

shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) *finter finter*
using *assms* **unfolding** *rel-fun-def*
using *inter-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fminus-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> *rel-fset A*) (|−|) (|−|)
using *assms* **unfolding** *rel-fun-def*
using *Diff-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fsubset-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows (*rel-fset A* ===> *rel-fset A* ===> (=)) (|⊆|) (|⊆|)
using *assms* **unfolding** *rel-fun-def*
using *subset-transfer*[*unfolded rel-fun-def*, *rule-format*, *Transfer.transferred*] **by** *blast*

lemma *fSup-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Sup Sup*
unfolding *rel-fun-def*
apply *clarify*
apply *transfer′*
using *Sup-fset-transfer*[*unfolded rel-fun-def*] **by** *blast*

lemma *fInf-transfer* [*transfer-rule*]:
assumes *bi-unique A* **and** *bi-total A*
shows (*rel-set* (*rel-fset A*) ===> *rel-fset A*) *Inf Inf*
using *assms* **unfolding** *rel-fun-def*
apply *clarify*
apply *transfer′*
using *Inf-fset-transfer*[*unfolded rel-fun-def*] **by** *blast*

lemma *ffilter-transfer* [*transfer-rule*]:
assumes *bi-unique A*
shows ((*A* ===> (=)) ===> *rel-fset A* ===> *rel-fset A*) *ffilter ffilter*
using *assms Lifting-Set.filter-transfer*
unfolding *rel-fun-def* **by** (*metis ffilter.rep-eq rel-fset.rep-eq*)

lemma *card-transfer* [*transfer-rule*]:
bi-unique A ⟹ (*rel-fset A* ===> (=)) *fcard fcard*
using *card-transfer* **unfolding** *rel-fun-def*
by (*metis fcard.rep-eq rel-fset.rep-eq*)

end

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

29.10 BNF setup

context
includes *fset.lifting*
begin

lemma *rel-fset-alt*:
rel-fset R a b $\longleftrightarrow$ $(\forall t \in$ *fset a*. $\exists u \in$ *fset b*. *R t u*$) \wedge (\forall t \in$ *fset b*. $\exists u \in$ *fset a*. *R u t*$)$
by *transfer* (*simp add*: *rel-set-def*)

lemma *fset-to-fset*: *finite A* $\Longrightarrow$ *fset* (*the-inv fset A*) = *A*
by (*metis CollectI f-the-inv-into-f fset-cases fset-cong inj-onI rangeI*)

lemma *rel-fset-aux*:
$(\forall t \in$ *fset a*. $\exists u \in$ *fset b*. *R t u*$) \wedge (\forall u \in$ *fset b*. $\exists t \in$ *fset a*. *R t u*$) \longleftrightarrow$
(*BNF-Def.Grp* {*a*. *fset a* $\subseteq$ {(*a*, *b*). *R a b*}} (*fimage fst*))$^{-1-1}$ *OO*
BNF-Def.Grp {*a*. *fset a* $\subseteq$ {(*a*, *b*). *R a b*}} (*fimage snd*)) *a b* (**is** *?L* = *?R*)
proof
assume *?L*
define *R′* **where** *R′* =
the-inv fset (*Collect* (*case-prod R*) $\cap$ (*fset a* $\times$ *fset b*)) (**is** - = *the-inv fset ?L′*)
have *finite ?L′* **by** (*intro finite-Int*[*OF disjI2*] *finite-cartesian-product*) (*transfer*, *simp*)+
hence $*$: *fset R′* = *?L′* **unfolding** *R′-def* **by** (*intro fset-to-fset*)
show *?R* **unfolding** *Grp-def relcompp.simps conversep.simps*
proof (*intro CollectI case-prodI exI*[*of - a*] *exI*[*of - b*] *exI*[*of - R′*] *conjI refl*)
from $*$ **show** *a* = *fimage fst R′* **using** *conjunct1*[*OF* ‹*?L*›]
by (*transfer*, *auto simp add*: *image-def Int-def split*: *prod.splits*)
from $*$ **show** *b* = *fimage snd R′* **using** *conjunct2*[*OF* ‹*?L*›]
by (*transfer*, *auto simp add*: *image-def Int-def split*: *prod.splits*)
qed (*auto simp add*: $*$)
next
assume *?R* **thus** *?L* **unfolding** *Grp-def relcompp.simps conversep.simps*
using *Product-Type.Collect-case-prodD* **by** *blast*
qed

bnf *'a fset*
map: *fimage*
sets: *fset*
bd: *natLeq*
wits: {||}
rel: *rel-fset*
apply −
apply *transfer′* **apply** *simp*
apply *transfer′* **apply** *force*

apply *transfer* **apply** *force*
apply *transfer'* **apply** *force*
apply (*rule natLeq-card-order*)
apply (*rule natLeq-cinfinite*)
apply (*rule regularCard-natLeq*)
apply *transfer* **apply** (*metis finite-iff-ordLess-natLeq*)
apply (*fastforce simp*: *rel-fset-alt*)
apply (*simp add*: *Grp-def relcompp.simps conversep.simps fun-eq-iff rel-fset-alt rel-fset-aux*[*unfolded OO-Grp-alt*])
apply *transfer* **apply** *simp*
done

lemma *rel-fset-fset*: *rel-set* χ (*fset A1*) (*fset A2*) = *rel-fset* χ *A1 A2*
by (*simp add*: *rel-fset.rep-eq*)

end

declare
fset.map-comp[*simp*]
fset.map-id[*simp*]
fset.set-map[*simp*]

29.11 Size setup

context includes *fset.lifting*
begin
lift-definition *size-fset* :: ('*a* ⇒ *nat*) ⇒ '*a fset* ⇒ *nat* **is** $\lambda f.$ *sum* (*Suc* ∘ *f*) **.**
end

instantiation *fset* :: (*type*) *size*
begin
definition *size-fset* **where**
size-fset-overloaded-def: *size-fset* = *FSet.size-fset* (λ-. *0*)
instance ..
end

lemma *size-fset-simps*[*simp*]: *size-fset f X* = ($\sum x \in$ *fset X*. *Suc* (*f x*))
by (*rule size-fset-def*[*THEN meta-eq-to-obj-eq*, *THEN fun-cong*, *THEN fun-cong*, *unfolded map-fun-def comp-def id-apply*])

lemma *size-fset-overloaded-simps*[*simp*]: *size X* = ($\sum x \in$ *fset X*. *Suc 0*)
by (*rule size-fset-simps*[*of* λ-. *0*, *unfolded add-0-left add-0-right*, *folded size-fset-overloaded-def*])

lemma *fset-size-o-map*: *inj f* ⟹ *size-fset g* ∘ *fimage f* = *size-fset* (*g* ∘ *f*)
unfolding *fun-eq-iff*
by (*simp add*: *inj-def inj-onI sum.reindex*)

setup ‹

BNF-LFP-Size.register-size-global ***type-name*** ‹*fset*› ***const-name*** ‹*size-fset*›
@{*thm size-fset-overloaded-def*} @{*thms size-fset-simps size-fset-overloaded-simps*}
@{*thms fset-size-o-map*}
›

lifting-update *fset.lifting*
lifting-forget *fset.lifting*

29.12 Advanced relator customization

Set vs. sum relators:

lemma *rel-set-rel-sum*[*simp*]:
rel-set (*rel-sum* χ φ) *A1 A2* $\longleftrightarrow$
rel-set χ (*Inl* −‘ *A1*) (*Inl* −‘ *A2*) ∧ *rel-set* φ (*Inr* −‘ *A1*) (*Inr* −‘ *A2*)
(**is** *?L* $\longleftrightarrow$ *?Rl* ∧ *?Rr*)
proof *safe*
assume *L*: *?L*
show *?Rl* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
fix *l1* **assume** *Inl l1* ∈ *A1*
then obtain *a2* **where** *a2*: *a2* ∈ *A2* **and** *rel-sum* χ φ (*Inl l1*) *a2*
using *L* **unfolding** *rel-set-def* **by** *auto*
then obtain *l2* **where** *a2* = *Inl l2* ∧ χ *l1 l2* **by** (*cases a2*, *auto*)
thus ∃ *l2*. *Inl l2* ∈ *A2* ∧ χ *l1 l2* **using** *a2* **by** *auto*
next
fix *l2* **assume** *Inl l2* ∈ *A2*
then obtain *a1* **where** *a1*: *a1* ∈ *A1* **and** *rel-sum* χ φ *a1* (*Inl l2*)
using *L* **unfolding** *rel-set-def* **by** *auto*
then obtain *l1* **where** *a1* = *Inl l1* ∧ χ *l1 l2* **by** (*cases a1*, *auto*)
thus ∃ *l1*. *Inl l1* ∈ *A1* ∧ χ *l1 l2* **using** *a1* **by** *auto*
qed
show *?Rr* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
fix *r1* **assume** *Inr r1* ∈ *A1*
then obtain *a2* **where** *a2*: *a2* ∈ *A2* **and** *rel-sum* χ φ (*Inr r1*) *a2*
using *L* **unfolding** *rel-set-def* **by** *auto*
then obtain *r2* **where** *a2* = *Inr r2* ∧ φ *r1 r2* **by** (*cases a2*, *auto*)
thus ∃ *r2*. *Inr r2* ∈ *A2* ∧ φ *r1 r2* **using** *a2* **by** *auto*
next
fix *r2* **assume** *Inr r2* ∈ *A2*
then obtain *a1* **where** *a1*: *a1* ∈ *A1* **and** *rel-sum* χ φ *a1* (*Inr r2*)
using *L* **unfolding** *rel-set-def* **by** *auto*
then obtain *r1* **where** *a1* = *Inr r1* ∧ φ *r1 r2* **by** (*cases a1*, *auto*)
thus ∃ *r1*. *Inr r1* ∈ *A1* ∧ φ *r1 r2* **using** *a1* **by** *auto*
qed
next
assume *Rl*: *?Rl* **and** *Rr*: *?Rr*
show *?L* **unfolding** *rel-set-def Bex-def vimage-eq* **proof** *safe*
fix *a1* **assume** *a1*: *a1* ∈ *A1*
show ∃ *a2*. *a2* ∈ *A2* ∧ *rel-sum* χ φ *a1 a2*
proof(*cases a1*)

case (*Inl l1*) **then obtain** *l2* **where** *Inl l2* ∈ *A2* ∧ χ *l1 l2*
using *Rl a1* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inl* **by** *auto*
next
case (*Inr r1*) **then obtain** *r2* **where** *Inr r2* ∈ *A2* ∧ φ *r1 r2*
using *Rr a1* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inr* **by** *auto*
qed
next
fix *a2* **assume** *a2*: *a2* ∈ *A2*
show ∃ *a1*. *a1* ∈ *A1* ∧ *rel-sum* χ φ *a1 a2*
proof(*cases a2*)
case (*Inl l2*) **then obtain** *l1* **where** *Inl l1* ∈ *A1* ∧ χ *l1 l2*
using *Rl a2* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inl* **by** *auto*
next
case (*Inr r2*) **then obtain** *r1* **where** *Inr r1* ∈ *A1* ∧ φ *r1 r2*
using *Rr a2* **unfolding** *rel-set-def* **by** *blast*
thus *?thesis* **unfolding** *Inr* **by** *auto*
qed
qed
qed

29.12.1 Countability

lemma *exists-fset-of-list*: ∃ *xs*. *fset-of-list xs* = *S*
including *fset.lifting*
by *transfer* (*rule finite-list*)

lemma *fset-of-list-surj*[*simp*, *intro*]: *surj fset-of-list*
by (*metis exists-fset-of-list surj-def*)

instance *fset* :: (*countable*) *countable*
proof
obtain *to-nat* :: *'a list* ⇒ *nat* **where** *inj to-nat*
by (*metis ex-inj*)
moreover have *inj* (*inv fset-of-list*)
using *fset-of-list-surj* **by** (*rule surj-imp-inj-inv*)
ultimately have *inj* (*to-nat* ∘ *inv fset-of-list*)
by (*rule inj-compose*)
thus ∃ *to-nat*::*'a fset* ⇒ *nat*. *inj to-nat*
by *auto*
qed

29.13 Quickcheck setup

Setup adapted from sets.

notation *Quickcheck-Exhaustive.orelse* (**infixr** ‹*orelse*› *55*)

context
includes *term-syntax*
begin

definition [*code-unfold*]:
valterm-femptyset = *Code-Evaluation.valtermify* ({||} :: ('*a* :: *typerep*) *fset*)

definition [*code-unfold*]:
valtermify-finsert x s = *Code-Evaluation.valtermify finsert* {·} (*x* :: ('*a* :: *typerep* ∗ -)) {·} *s*

end

instantiation *fset* :: (*exhaustive*) *exhaustive*
begin

fun *exhaustive-fset* **where**
exhaustive-fset f i = (*if i* = *0 then None else* (*f* {||} *orelse exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.exhaustive* (λ*x*. *if x* |∈| *A then None else f* (*finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ..

end

instantiation *fset* :: (*full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-fset* **where**
full-exhaustive-fset f i = (*if i* = *0 then None else* (*f valterm-femptyset orelse full-exhaustive-fset* (λ*A*. *f A orelse Quickcheck-Exhaustive.full-exhaustive* (λ*x*. *if fst x* |∈| *fst A then None else f* (*valtermify-finsert x A*)) (*i* − *1*)) (*i* − *1*)))

instance ..

end

no-notation *Quickcheck-Exhaustive.orelse* (**infixr** ‹*orelse*› *55*)

instantiation *fset* :: (*random*) *random*
begin

context
includes *state-combinator-syntax*
begin

fun *random-aux-fset* :: *natural* ⇒ *natural* ⇒ *natural* × *natural* ⇒ ('*a fset* × (*unit* ⇒ *term*)) × *natural* × *natural* **where**
random-aux-fset 0 j = *Quickcheck-Random.collapse* (*Random.select-weight* [(*1*, *Pair*

```
valterm-femptyset)]) |
random-aux-fset (Code-Numeral.Suc i) j =
  Quickcheck-Random.collapse (Random.select-weight
    [(1, Pair valterm-femptyset),
     (Code-Numeral.Suc i,
        Quickcheck-Random.random j ○→ (λx. random-aux-fset i j ○→ (λs. Pair
(valtermify-finsert x s))))])

lemma [code]:
  random-aux-fset i j =
   Quickcheck-Random.collapse (Random.select-weight [(1, Pair valterm-femptyset),
      (i, Quickcheck-Random.random j ○→ (λx. random-aux-fset (i − 1) j ○→ (λs.
Pair (valtermify-finsert x s))))])
proof (induct i rule: natural.induct)
  case zero
 show ?case by (subst select-weight-drop-zero[symmetric]) (simp add: less-natural-def)
next
  case (Suc i)
  show ?case by (simp only: random-aux-fset.simps Suc-natural-minus-one)
qed

definition random-fset i = random-aux-fset i i

instance ..

end

end
```

29.14 Code Generation Setup

The following *code-unfold* lemmas are so the pre-processor of the code generator will perform conversions like, e.g., $(x \mid\in\mid f \mid`\mid fset\text{-}of\text{-}list\ xs) = (x \in f\ `\ set\ xs)$.

```
declare
  ffilter.rep-eq[code-unfold]
  fimage.rep-eq[code-unfold]
  finsert.rep-eq[code-unfold]
  fset-of-list.rep-eq[code-unfold]
  inf-fset.rep-eq[code-unfold]
  minus-fset.rep-eq[code-unfold]
  sup-fset.rep-eq[code-unfold]
  uminus-fset.rep-eq[code-unfold]

end
```

30 Type of finite maps defined as a subtype of maps

theory *Finite-Map*
imports *FSet AList Conditional-Parametricity*
abbrevs $(= = \subseteq_f$
begin

30.1 Auxiliary constants and lemmas over *map*

parametric-constant *map-add-transfer*[*transfer-rule*]: *map-add-def*
parametric-constant *map-of-transfer*[*transfer-rule*]: *map-of-def*

context includes *lifting-syntax* **begin**

abbreviation *rel-map* :: $('b \Rightarrow 'c \Rightarrow bool) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'c) \Rightarrow bool$ **where**
rel-map f $\equiv (=)$ ===> *rel-option f*

lemma *ran-transfer*[*transfer-rule*]: (*rel-map A* ===> *rel-set A*) *ran ran*
proof
fix *m n*
assume *rel-map A m n*
show *rel-set A* (*ran m*) (*ran n*)
proof (*rule rel-setI*)
fix *x*
assume *x* ∈ *ran m*
then obtain *a* **where** *m a* = *Some x*
unfolding *ran-def* **by** *auto*

have *rel-option A* (*m a*) (*n a*)
using ‹*rel-map A m n*›
by (*auto dest*: *rel-funD*)
then obtain *y* **where** *n a* = *Some y A x y*
unfolding ‹*m a* = -›
by *cases auto*
then show ∃ *y* ∈ *ran n*. *A x y*
unfolding *ran-def* **by** *blast*
next
fix *y*
assume *y* ∈ *ran n*
then obtain *a* **where** *n a* = *Some y*
unfolding *ran-def* **by** *auto*

have *rel-option A* (*m a*) (*n a*)
using ‹*rel-map A m n*›
by (*auto dest*: *rel-funD*)
then obtain *x* **where** *m a* = *Some x A x y*
unfolding ‹*n a* = -›
by *cases auto*

then show $\exists\, x \in \mathit{ran}\ m.\ A\ x\ y$
unfolding *ran-def* **by** *blast*
qed
qed

lemma *ran-alt-def*: $\mathit{ran}\ m = (\mathit{the} \circ m)\ `\ \mathit{dom}\ m$
unfolding *ran-def dom-def* **by** *force*

parametric-constant *dom-transfer*[*transfer-rule*]: *dom-def*

definition *map-upd* :: $'a \Rightarrow 'b \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
$\mathit{map\text{-}upd}\ k\ v\ m = m(k \mapsto v)$

parametric-constant *map-upd-transfer*[*transfer-rule*]: *map-upd-def*

definition *map-filter* :: $('a \Rightarrow \mathit{bool}) \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
$\mathit{map\text{-}filter}\ P\ m = (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ m\ x\ \mathit{else}\ \mathit{None})$

parametric-constant *map-filter-transfer*[*transfer-rule*]: *map-filter-def*

lemma *map-filter-map-of*[*simp*]: $\mathit{map\text{-}filter}\ P\ (\mathit{map\text{-}of}\ m) = \mathit{map\text{-}of}\ [(k,\ \text{-}) \leftarrow m.\ P\ k]$
proof
fix *x*
show $\mathit{map\text{-}filter}\ P\ (\mathit{map\text{-}of}\ m)\ x = \mathit{map\text{-}of}\ [(k,\ \text{-}) \leftarrow m.\ P\ k]\ x$
by (*induct m*) (*auto simp*: *map-filter-def*)
qed

lemma *map-filter-finite*[*intro*]:
assumes $\mathit{finite}\ (\mathit{dom}\ m)$
shows $\mathit{finite}\ (\mathit{dom}\ (\mathit{map\text{-}filter}\ P\ m))$
proof −
from *assms* **have** ‹$\mathit{finite}\ (\mathit{dom}\ (\lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ m\ x\ \mathit{else}\ \mathit{None}))$›
by (*rule rev-finite-subset*) (*auto split*: *if-splits*)
then show *?thesis*
by (*simp add*: *map-filter-def*)
qed

definition *map-drop* :: $'a \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
$\mathit{map\text{-}drop}\ a = \mathit{map\text{-}filter}\ (\lambda a'.\ a' \neq a)$

parametric-constant *map-drop-transfer*[*transfer-rule*]: *map-drop-def*

definition *map-drop-set* :: $'a\ \mathit{set} \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**
$\mathit{map\text{-}drop\text{-}set}\ A = \mathit{map\text{-}filter}\ (\lambda a.\ a \notin A)$

parametric-constant *map-drop-set-transfer*[*transfer-rule*]: *map-drop-set-def*

definition *map-restrict-set* :: $'a\ \mathit{set} \Rightarrow ('a \rightharpoonup 'b) \Rightarrow ('a \rightharpoonup 'b)$ **where**

map-restrict-set A = map-filter (λ*a*. *a* ∈ *A*)

parametric-constant *map-restrict-set-transfer*[*transfer-rule*]: *map-restrict-set-def*

lemma *map-restrict-set-conv-restrict-map*: *map-restrict-set A m = Map.restrict-map m A*
unfolding *map-restrict-set-def map-filter-def Map.restrict-map-def* **..**

definition *map-pred* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a* ⇀ *'b*) ⇒ *bool* **where**
map-pred P m ⟷ (∀ *x*. *case m x of None* ⇒ *True* | *Some y* ⇒ *P x y*)

parametric-constant *map-pred-transfer*[*transfer-rule*]: *map-pred-def*

definition *rel-map-on-set* :: *'a set* ⇒ (*'b* ⇒ *'c* ⇒ *bool*) ⇒ (*'a* ⇀ *'b*) ⇒ (*'a* ⇀ *'c*) ⇒ *bool* **where**
rel-map-on-set S P = eq-onp (λ*x*. *x* ∈ *S*) ===> *rel-option P*

lemma *dom-comp*: *dom* (*m* ∘$_m$ *n*) ⊆ *dom n*
unfolding *map-comp-def dom-def*
by (*auto split*: *option.splits*)

lemma *dom-comp-finite*: *finite* (*dom n*) ⟹ *finite* (*dom* (*map-comp m n*))
by (*metis finite-subset dom-comp*)

parametric-constant *map-comp-transfer*[*transfer-rule*]: *map-comp-def*

end

30.2 Abstract characterisation

typedef (*'a*, *'b*) *fmap* = {*m*. *finite* (*dom m*)} :: (*'a* ⇀ *'b*) *set*
morphisms *fmlookup Abs-fmap*
proof
show *Map.empty* ∈ {*m*. *finite* (*dom m*)}
by *auto*
qed

setup-lifting *type-definition-fmap*

lemma *dom-fmlookup-finite*[*intro*, *simp*]: *finite* (*dom* (*fmlookup m*))
using *fmap.fmlookup* **by** *auto*

lemma *fmap-ext*:
assumes ⋀*x*. *fmlookup m x = fmlookup n x*
shows *m = n*
using *assms*
by *transfer' auto*

30.3 Operations

context
includes *fset.lifting*
begin

lift-definition *fmran* :: ('*a*, '*b*) *fmap* ⇒ '*b fset*
is *ran*
parametric *ran-transfer*
by (*rule finite-ran*)

lemma *fmlookup-ran-iff*: *y* |∈| *fmran m* ⟷ (∃ *x*. *fmlookup m x* = *Some y*)
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmranI*: *fmlookup m x* = *Some y* ⟹ *y* |∈| *fmran m* **by** (*auto simp*: *fmlookup-ran-iff*)

lemma *fmranE*[*elim*]:
assumes *y* |∈| *fmran m*
obtains *x* **where** *fmlookup m x* = *Some y*
using *assms* **by** (*auto simp*: *fmlookup-ran-iff*)

lift-definition *fmdom* :: ('*a*, '*b*) *fmap* ⇒ '*a fset*
is *dom*
parametric *dom-transfer*
.

lemma *fmlookup-dom-iff*: *x* |∈| *fmdom m* ⟷ (∃ *a*. *fmlookup m x* = *Some a*)
by *transfer' auto*

lemma *fmdom-notI*: *fmlookup m x* = *None* ⟹ *x* |∉| *fmdom m* **by** (*simp add*: *fmlookup-dom-iff*)
lemma *fmdomI*: *fmlookup m x* = *Some y* ⟹ *x* |∈| *fmdom m* **by** (*simp add*: *fmlookup-dom-iff*)
lemma *fmdom-notD*[*dest*]: *x* |∉| *fmdom m* ⟹ *fmlookup m x* = *None* **by** (*simp add*: *fmlookup-dom-iff*)

lemma *fmdomE*[*elim*]:
assumes *x* |∈| *fmdom m*
obtains *y* **where** *fmlookup m x* = *Some y*
using *assms* **by** (*auto simp*: *fmlookup-dom-iff*)

lift-definition *fmdom'* :: ('*a*, '*b*) *fmap* ⇒ '*a set*
is *dom*
parametric *dom-transfer*
.

lemma *fmlookup-dom'-iff*: *x* ∈ *fmdom' m* ⟷ (∃ *a*. *fmlookup m x* = *Some a*)
by *transfer' auto*

lemma *fmdom'-notI*: *fmlookup m x = None ⟹ x ∉ fmdom' m* **by** (*simp add*: *fmlookup-dom'-iff*)
lemma *fmdom'I*: *fmlookup m x = Some y ⟹ x ∈ fmdom' m* **by** (*simp add*: *fmlookup-dom'-iff*)
lemma *fmdom'-notD*[*dest*]: *x ∉ fmdom' m ⟹ fmlookup m x = None* **by** (*simp add*: *fmlookup-dom'-iff*)

lemma *fmdom'E*[*elim*]:
  assumes *x ∈ fmdom' m*
  obtains *x y* **where** *fmlookup m x = Some y*
using *assms* **by** (*auto simp*: *fmlookup-dom'-iff*)

lemma *fmdom'-alt-def*: *fmdom' m = fset* (*fmdom m*)
by *transfer' force*

lemma *finite-fmdom'*[*simp*]: *finite* (*fmdom' m*)
unfolding *fmdom'-alt-def* **by** *simp*

lemma *dom-fmlookup*[*simp*]: *dom* (*fmlookup m*) = *fmdom' m*
by *transfer' simp*

lift-definition *fmempty* :: ('*a*, '*b*) *fmap*
  is *Map.empty*
by *simp*

lemma *fmempty-lookup*[*simp*]: *fmlookup fmempty x = None*
by *transfer' simp*

lemma *fmdom-empty*[*simp*]: *fmdom fmempty* = {||} **by** *transfer' simp*
lemma *fmdom'-empty*[*simp*]: *fmdom' fmempty* = {} **by** *transfer' simp*
lemma *fmran-empty*[*simp*]: *fmran fmempty = fempty* **by** *transfer'* (*auto simp*: *ran-def map-filter-def*)

lift-definition *fmupd* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap*
  is *map-upd*
  parametric *map-upd-transfer*
unfolding *map-upd-def*[*abs-def*]
by *simp*

lemma *fmupd-lookup*[*simp*]: *fmlookup* (*fmupd a b m*) *a'* = (*if a = a' then Some b else fmlookup m a'*)
by *transfer'* (*auto simp*: *map-upd-def*)

lemma *fmdom-fmupd*[*simp*]: *fmdom* (*fmupd a b m*) = *finsert a* (*fmdom m*) **by** *transfer* (*simp add*: *map-upd-def*)
lemma *fmdom'-fmupd*[*simp*]: *fmdom'* (*fmupd a b m*) = *insert a* (*fmdom' m*) **by** *transfer* (*simp add*: *map-upd-def*)

lemma *fmupd-reorder-neq*:

assumes $a \neq b$
shows *fmupd a x* (*fmupd b y m*) = *fmupd b y* (*fmupd a x m*)
using *assms*
by *transfer'* (*auto simp*: *map-upd-def*)

lemma *fmupd-idem*[*simp*]: *fmupd a x* (*fmupd a y m*) = *fmupd a x m*
by *transfer'* (*auto simp*: *map-upd-def*)

lift-definition *fmfilter* :: ('*a* ⇒ *bool*) ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap*
is *map-filter*
parametric *map-filter-transfer*
by *auto*

lemma *fmdom-filter*[*simp*]: *fmdom* (*fmfilter P m*) = *ffilter P* (*fmdom m*)
by *transfer'* (*auto simp*: *map-filter-def split*: *if-splits*)

lemma *fmdom'-filter*[*simp*]: *fmdom'* (*fmfilter P m*) = *Set.filter P* (*fmdom' m*)
by *transfer'* (*auto simp*: *map-filter-def split*: *if-splits*)

lemma *fmlookup-filter*[*simp*]: *fmlookup* (*fmfilter P m*) *x* = (*if P x then fmlookup m x else None*)
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmfilter-empty*[*simp*]: *fmfilter P fmempty* = *fmempty*
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmfilter-true*[*simp*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x*
shows *fmfilter P m* = *m*
proof (*rule fmap-ext*)
fix *x*
have *fmlookup m x* = *None* **if** ¬ *P x*
using *that assms* **by** *fastforce*
then show *fmlookup* (*fmfilter P m*) *x* = *fmlookup m x*
by *simp*
qed

lemma *fmfilter-false*[*simp*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ ¬ *P x*
shows *fmfilter P m* = *fmempty*
using *assms* **by** *transfer'* (*fastforce simp*: *map-filter-def*)

lemma *fmfilter-comp*[*simp*]: *fmfilter P* (*fmfilter Q m*) = *fmfilter* (λ*x*. *P x* ∧ *Q x*) *m*
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmfilter-comm*: *fmfilter P* (*fmfilter Q m*) = *fmfilter Q* (*fmfilter P m*)
unfolding *fmfilter-comp* **by** *meson*

lemma *fmfilter-cong*[*cong*]:
  assumes $\bigwedge x\ y.$ *fmlookup* $m\ x =$ *Some* $y \Longrightarrow P\ x = Q\ x$
  shows *fmfilter* $P\ m =$ *fmfilter* $Q\ m$
proof (*rule fmap-ext*)
  fix x
  have *fmlookup* $m\ x =$ *None* **if** $P\ x \neq Q\ x$
    using *that assms* **by** *fastforce*
  then show *fmlookup* (*fmfilter* $P\ m$) $x =$ *fmlookup* (*fmfilter* $Q\ m$) x
    by *auto*
qed

lemma *fmfilter-cong'*[*fundef-cong*]:
  assumes $m = n\ \bigwedge x.\ x \in$ *fmdom'* $m \Longrightarrow P\ x = Q\ x$
  shows *fmfilter* $P\ m =$ *fmfilter* $Q\ n$
using *assms*(*2*) **unfolding** *assms*(*1*)
by (*rule fmfilter-cong*) (*metis fmdom'I*)

lemma *fmfilter-upd*[*simp*]:
  fmfilter P (*fmupd* $x\ y\ m$) = (*if* $P\ x$ *then fmupd* $x\ y$ (*fmfilter* $P\ m$) *else fmfilter* P m)
by *transfer'* (*auto simp*: *map-upd-def map-filter-def*)

lift-definition *fmdrop* :: $'a \Rightarrow$ ($'a$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap*
  is *map-drop*
  parametric *map-drop-transfer*
unfolding *map-drop-def* **by** *auto*

lemma *fmdrop-lookup*[*simp*]: *fmlookup* (*fmdrop* $a\ m$) $a =$ *None*
by *transfer'* (*auto simp*: *map-drop-def map-filter-def*)

lift-definition *fmdrop-set* :: $'a$ *set* $\Rightarrow$ ($'a$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap*
  is *map-drop-set*
  parametric *map-drop-set-transfer*
unfolding *map-drop-set-def* **by** *auto*

lift-definition *fmdrop-fset* :: $'a$ *fset* $\Rightarrow$ ($'a$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap*
  is *map-drop-set*
  parametric *map-drop-set-transfer*
unfolding *map-drop-set-def* **by** *auto*

lift-definition *fmrestrict-set* :: $'a$ *set* $\Rightarrow$ ($'a$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap*
  is *map-restrict-set*
  parametric *map-restrict-set-transfer*
unfolding *map-restrict-set-def* **by** *auto*

lift-definition *fmrestrict-fset* :: $'a$ *fset* $\Rightarrow$ ($'a$, $'b$) *fmap* $\Rightarrow$ ($'a$, $'b$) *fmap*
  is *map-restrict-set*
  parametric *map-restrict-set-transfer*
unfolding *map-restrict-set-def* **by** *auto*

lemma *fmfilter-alt-defs*:
fmdrop a = fmfilter (λ*a'*. *a'* ≠ *a*)
fmdrop-set A = fmfilter (λ*a*. *a* ∉ *A*)
fmdrop-fset B = fmfilter (λ*a*. *a* |∉| *B*)
fmrestrict-set A = fmfilter (λ*a*. *a* ∈ *A*)
fmrestrict-fset B = fmfilter (λ*a*. *a* |∈| *B*)
by (*transfer'*; *simp add*: *map-drop-def map-drop-set-def map-restrict-set-def*)+

lemma *fmdom-drop*[*simp*]: *fmdom* (*fmdrop a m*) = *fmdom m* − {|*a*|} **unfolding** *fmfilter-alt-defs* **by** *auto*
lemma *fmdom'-drop*[*simp*]: *fmdom'* (*fmdrop a m*) = *fmdom' m* − {*a*} **unfolding** *fmfilter-alt-defs* **by** *auto*
lemma *fmdom'-drop-set*[*simp*]: *fmdom'* (*fmdrop-set A m*) = *fmdom' m* − *A* **unfolding** *fmfilter-alt-defs* **by** *auto*
lemma *fmdom-drop-fset*[*simp*]: *fmdom* (*fmdrop-fset A m*) = *fmdom m* − *A* **unfolding** *fmfilter-alt-defs* **by** *auto*
lemma *fmdom'-restrict-set*: *fmdom'* (*fmrestrict-set A m*) ⊆ *A* **unfolding** *fmfilter-alt-defs* **by** *auto*
lemma *fmdom-restrict-fset*: *fmdom* (*fmrestrict-fset A m*) |⊆| *A* **unfolding** *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-fmupd*: *fmdrop x* (*fmupd y z m*) = (*if x* = *y then fmdrop x m else fmupd y z* (*fmdrop x m*))
by *transfer'* (*auto simp*: *map-drop-def map-filter-def map-upd-def*)

lemma *fmdrop-idle*: *x* |∉| *fmdom B* ⟹ *fmdrop x B* = *B*
by *transfer'* (*auto simp*: *map-drop-def map-filter-def*)

lemma *fmdrop-idle'*: *x* ∉ *fmdom' B* ⟹ *fmdrop x B* = *B*
by *transfer'* (*auto simp*: *map-drop-def map-filter-def*)

lemma *fmdrop-fmupd-same*: *fmdrop x* (*fmupd x y m*) = *fmdrop x m*
by *transfer'* (*auto simp*: *map-drop-def map-filter-def map-upd-def*)

lemma *fmdom'-restrict-set-precise*: *fmdom'* (*fmrestrict-set A m*) = *fmdom' m* ∩ *A*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdom'-restrict-fset-precise*: *fmdom* (*fmrestrict-fset A m*) = *fmdom m* |∩| *A*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdom'-drop-fset*[*simp*]: *fmdom'* (*fmdrop-fset A m*) = *fmdom' m* − *fset A*
unfolding *fmfilter-alt-defs* **by** *transfer'* (*auto simp*: *map-filter-def split*: *if-splits*)

lemma *fmdom'-restrict-fset*: *fmdom'* (*fmrestrict-fset A m*) ⊆ *fset A*
unfolding *fmfilter-alt-defs* **by** *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmlookup-drop*[*simp*]:
fmlookup (*fmdrop a m*) *x* = (*if x* ≠ *a then fmlookup m x else None*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-drop-set*[*simp*]:
fmlookup (*fmdrop-set A m*) *x* = (*if x* ∉ *A then fmlookup m x else None*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-drop-fset*[*simp*]:
fmlookup (*fmdrop-fset A m*) *x* = (*if x* |∉| *A then fmlookup m x else None*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-restrict-set*[*simp*]:
fmlookup (*fmrestrict-set A m*) *x* = (*if x* ∈ *A then fmlookup m x else None*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmlookup-restrict-fset*[*simp*]:
fmlookup (*fmrestrict-fset A m*) *x* = (*if x* |∈| *A then fmlookup m x else None*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-dom*[*simp*]: *fmrestrict-set* (*fmdom′ m*) *m* = *m*
by (*rule fmap-ext*) *auto*

lemma *fmrestrict-fset-dom*[*simp*]: *fmrestrict-fset* (*fmdom m*) *m* = *m*
by (*rule fmap-ext*) *auto*

lemma *fmdrop-empty*[*simp*]: *fmdrop a fmempty* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-empty*[*simp*]: *fmdrop-set A fmempty* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-empty*[*simp*]: *fmdrop-fset A fmempty* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmdom*[*simp*]: *fmdrop-fset* (*fmdom A*) *A* = *fmempty*
by *transfer′* (*auto simp*: *map-drop-set-def map-filter-def*)

lemma *fmdrop-set-fmdom*[*simp*]: *fmdrop-set* (*fmdom′ A*) *A* = *fmempty*
by *transfer′* (*auto simp*: *map-drop-set-def map-filter-def*)

lemma *fmrestrict-set-empty*[*simp*]: *fmrestrict-set A fmempty* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-empty*[*simp*]: *fmrestrict-fset A fmempty* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-null*[*simp*]: *fmdrop-set* {} *m* = *m*
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fset-null*[*simp*]: *fmdrop-fset* {||} *m* = *m*
by (*rule fmap-ext*) *auto*

lemma *fmdrop-set-single*[*simp*]: *fmdrop-set* {*a*} *m* = *fmdrop a m*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-single*[*simp*]: *fmdrop-fset* {|*a*|} *m* = *fmdrop a m*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-null*[*simp*]: *fmrestrict-set* {} *m* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-null*[*simp*]: *fmrestrict-fset* {||} *m* = *fmempty*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-comm*: *fmdrop a* (*fmdrop b m*) = *fmdrop b* (*fmdrop a m*)
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-comm*)

lemma *fmdrop-set-insert*[*simp*]: *fmdrop-set* (*insert x S*) *m* = *fmdrop x* (*fmdrop-set S m*)
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fset-insert*[*simp*]: *fmdrop-fset* (*finsert x S*) *m* = *fmdrop x* (*fmdrop-fset S m*)
by (*rule fmap-ext*) *auto*

lemma *fmrestrict-set-twice*[*simp*]: *fmrestrict-set S* (*fmrestrict-set T m*) = *fmrestrict-set* (*S* ∩ *T*) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-fset-twice*[*simp*]: *fmrestrict-fset S* (*fmrestrict-fset T m*) = *fmrestrict-fset* (*S* |∩| *T*) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-set-drop*[*simp*]: *fmrestrict-set S* (*fmdrop b m*) = *fmrestrict-set* (*S* − {*b*}) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmrestrict-fset-drop*[*simp*]: *fmrestrict-fset S* (*fmdrop b m*) = *fmrestrict-fset* (*S* − {| *b* |}) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-fmrestrict-set*[*simp*]: *fmdrop b* (*fmrestrict-set S m*) = *fmrestrict-set* (*S* − {*b*}) *m*
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fmrestrict-fset*[*simp*]: *fmdrop b* (*fmrestrict-fset S m*) = *fmrestrict-fset* (*S* − {| *b* |}) *m*

by (*rule fmap-ext*) *auto*

lemma *fmdrop-idem*[*simp*]: *fmdrop a* (*fmdrop a m*) = *fmdrop a m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-set-twice*[*simp*]: *fmdrop-set S* (*fmdrop-set T m*) = *fmdrop-set* (*S* ∪ *T*) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-fset-twice*[*simp*]: *fmdrop-fset S* (*fmdrop-fset T m*) = *fmdrop-fset* (*S* |∪| *T*) *m*
unfolding *fmfilter-alt-defs* **by** *auto*

lemma *fmdrop-set-fmdrop*[*simp*]: *fmdrop-set S* (*fmdrop b m*) = *fmdrop-set* (*insert b S*) *m*
by (*rule fmap-ext*) *auto*

lemma *fmdrop-fset-fmdrop*[*simp*]: *fmdrop-fset S* (*fmdrop b m*) = *fmdrop-fset* (*finsert b S*) *m*
by (*rule fmap-ext*) *auto*

lift-definition *fmadd* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* (**infixl** ‹+$+_f$› *100*)
  is *map-add*
  parametric *map-add-transfer*
  by *simp*

lemma *fmlookup-add*[*simp*]:
  fmlookup (*m* +$+_f$ *n*) *x* = (*if x* |∈| *fmdom n then fmlookup n x else fmlookup m x*)
  by *transfer'* (*auto simp*: *map-add-def split*: *option.splits*)

lemma *fmdom-add*[*simp*]: *fmdom* (*m* +$+_f$ *n*) = *fmdom m* |∪| *fmdom n* **by** *transfer' auto*
lemma *fmdom'-add*[*simp*]: *fmdom'* (*m* +$+_f$ *n*) = *fmdom' m* ∪ *fmdom' n* **by** *transfer' auto*

lemma *fmadd-drop-left-dom*: *fmdrop-fset* (*fmdom n*) *m* +$+_f$ *n* = *m* +$+_f$ *n*
  by (*rule fmap-ext*) *auto*

lemma *fmadd-restrict-right-dom*: *fmrestrict-fset* (*fmdom n*) (*m* +$+_f$ *n*) = *n*
  by (*rule fmap-ext*) *auto*

lemma *fmfilter-add-distrib*[*simp*]: *fmfilter P* (*m* +$+_f$ *n*) = *fmfilter P m* +$+_f$ *fmfilter P n*
  by *transfer'* (*auto simp*: *map-filter-def map-add-def*)

lemma *fmdrop-add-distrib*[*simp*]: *fmdrop a* (*m* +$+_f$ *n*) = *fmdrop a m* +$+_f$ *fmdrop a n*

unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-add-distrib*[*simp*]: *fmdrop-set A* (*m* ++$_f$ *n*) = *fmdrop-set A m* ++$_f$ *fmdrop-set A n*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-add-distrib*[*simp*]: *fmdrop-fset A* (*m* ++$_f$ *n*) = *fmdrop-fset A m* ++$_f$ *fmdrop-fset A n*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-add-distrib*[*simp*]:
fmrestrict-set A (*m* ++$_f$ *n*) = *fmrestrict-set A m* ++$_f$ *fmrestrict-set A n*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-add-distrib*[*simp*]:
fmrestrict-fset A (*m* ++$_f$ *n*) = *fmrestrict-fset A m* ++$_f$ *fmrestrict-fset A n*
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmadd-empty*[*simp*]: *fmempty* ++$_f$ *m* = *m m* ++$_f$ *fmempty* = *m*
by (*transfer'*; *auto*)+

lemma *fmadd-idempotent*[*simp*]: *m* ++$_f$ *m* = *m*
by *transfer'* (*auto simp*: *map-add-def split*: *option.splits*)

lemma *fmadd-assoc*[*simp*]: *m* ++$_f$ (*n* ++$_f$ *p*) = *m* ++$_f$ *n* ++$_f$ *p*
by *transfer' simp*

lemma *fmadd-fmupd*[*simp*]: *m* ++$_f$ *fmupd a b n* = *fmupd a b* (*m* ++$_f$ *n*)
by (*rule fmap-ext*) *simp*

lift-definition *fmpred* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*a*, '*b*) *fmap* ⇒ *bool*
is *map-pred*
parametric *map-pred-transfer*
.

lemma *fmpredI*[*intro*]:
assumes ⋀*x y*. *fmlookup m x* = *Some y* ⟹ *P x y*
shows *fmpred P m*
using *assms*
by *transfer'* (*auto simp*: *map-pred-def split*: *option.splits*)

lemma *fmpredD*[*dest*]: *fmpred P m* ⟹ *fmlookup m x* = *Some y* ⟹ *P x y*
by *transfer'* (*auto simp*: *map-pred-def split*: *option.split-asm*)

lemma *fmpred-iff*: *fmpred P m* ⟷ (∀ *x y*. *fmlookup m x* = *Some y* ⟶ *P x y*)
by *auto*

lemma *fmpred-alt-def*: *fmpred P m* ⟷ *fBall* (*fmdom m*) (λ*x*. *P x* (*the* (*fmlookup m x*)))

unfolding *fmpred-iff*
using *fmdomI* **by** *fastforce*

lemma *fmpred-mono-strong*:
assumes $\bigwedge x\ y.$ *fmlookup* $m\ x =$ *Some* $y \Longrightarrow P\ x\ y \Longrightarrow Q\ x\ y$
shows *fmpred* $P\ m \Longrightarrow$ *fmpred* $Q\ m$
using *assms* **unfolding** *fmpred-iff* **by** *auto*

lemma *fmpred-mono*[*mono*]: $P \leq Q \Longrightarrow$ *fmpred* $P \leq$ *fmpred* Q
by *auto*

lemma *fmpred-empty*[*intro*!, *simp*]: *fmpred* P *fmempty*
by *auto*

lemma *fmpred-upd*[*intro*]: *fmpred* $P\ m \Longrightarrow P\ x\ y \Longrightarrow$ *fmpred* P (*fmupd* $x\ y\ m$)
by *transfer'* (*auto simp*: *map-pred-def map-upd-def*)

lemma *fmpred-updD*[*dest*]: *fmpred* P (*fmupd* $x\ y\ m$) $\Longrightarrow P\ x\ y$
by *auto*

lemma *fmpred-add*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* $P\ n \Longrightarrow$ *fmpred* P ($m +\!\!+_f n$)
by *transfer'* (*auto simp*: *map-pred-def map-add-def split*: *option.splits*)

lemma *fmpred-filter*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmfilter* $Q\ m$)
by *transfer'* (*auto simp*: *map-pred-def map-filter-def*)

lemma *fmpred-drop*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmdrop* $a\ m$)
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fmpred-drop-set*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmdrop-set* $A\ m$)
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fmpred-drop-fset*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmdrop-fset* $A\ m$)
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fmpred-restrict-set*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmrestrict-set* $A\ m$)
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fmpred-restrict-fset*[*intro*]: *fmpred* $P\ m \Longrightarrow$ *fmpred* P (*fmrestrict-fset* $A\ m$)
by (*auto simp*: *fmfilter-alt-defs*)

lemma *fmpred-cases*[*consumes 1*]:
assumes *fmpred* $P\ m$
obtains (*none*) *fmlookup* $m\ x =$ *None* | (*some*) y **where** *fmlookup* $m\ x =$ *Some* y $P\ x\ y$
using *assms* **by** *auto*

lift-definition *fmsubset* :: ('*a*, '*b*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ *bool* (**infix** ‹$\subseteq_f$› *50*)
is *map-le*

.

lemma *fmsubset-refl*[*simp*]: $m \subseteq_f m$
by *transfer* (*rule map-le-refl*)

lemma *fmsubset-trans*[*trans*]: $m_1 \subseteq_f m_2 \Longrightarrow m_2 \subseteq_f m_3 \Longrightarrow m_1 \subseteq_f m_3$
by *transfer* (*rule map-le-trans*)

lemma *fmsubset-antisym*: $m_1 \subseteq_f m_2 \Longrightarrow m_2 \subseteq_f m_1 \Longrightarrow m_1 = m_2$
by *transfer* (*rule map-le-antisym*)

lemma *fmsubset-alt-def*: $m \subseteq_f n \longleftrightarrow$ *fmpred* ($\lambda k\ v$. *fmlookup* $n\ k$ = *Some* v) m
by *transfer'* (*auto simp*: *map-pred-def map-le-def dom-def split*: *option.splits*)

lemma *fmsubset-pred*: *fmpred* $P\ m \Longrightarrow n \subseteq_f m \Longrightarrow$ *fmpred* $P\ n$
unfolding *fmsubset-alt-def fmpred-iff*
by *auto*

lemma *fmsubset-filter-mono*: $m \subseteq_f n \Longrightarrow$ *fmfilter* $P\ m \subseteq_f$ *fmfilter* $P\ n$
unfolding *fmsubset-alt-def fmpred-iff*
by *auto*

lemma *fmsubset-drop-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop* $a\ m \subseteq_f$ *fmdrop* $a\ n$
unfolding *fmfilter-alt-defs* **by** (*rule fmsubset-filter-mono*)

lemma *fmsubset-drop-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-set* $A\ m \subseteq_f$ *fmdrop-set* $A\ n$
unfolding *fmfilter-alt-defs* **by** (*rule fmsubset-filter-mono*)

lemma *fmsubset-drop-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmdrop-fset* $A\ m \subseteq_f$ *fmdrop-fset* A n
unfolding *fmfilter-alt-defs* **by** (*rule fmsubset-filter-mono*)

lemma *fmsubset-restrict-set-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-set* $A\ m \subseteq_f$ *fmrestrict-set* $A\ n$
unfolding *fmfilter-alt-defs* **by** (*rule fmsubset-filter-mono*)

lemma *fmsubset-restrict-fset-mono*: $m \subseteq_f n \Longrightarrow$ *fmrestrict-fset* $A\ m \subseteq_f$ *fmrestrict-fset* $A\ n$
unfolding *fmfilter-alt-defs* **by** (*rule fmsubset-filter-mono*)

lemma *fmfilter-subset*[*simp*]: *fmfilter* $P\ m \subseteq_f m$
unfolding *fmsubset-alt-def fmpred-iff* **by** *auto*

lemma *fmsubset-drop*[*simp*]: *fmdrop* $a\ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-subset*)

lemma *fmsubset-drop-set*[*simp*]: *fmdrop-set* $S\ m \subseteq_f m$
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-subset*)

lemma *fmsubset-drop-fset*[*simp*]: *fmdrop-fset S m* $\subseteq_f$ *m*
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-subset*)

lemma *fmsubset-restrict-set*[*simp*]: *fmrestrict-set S m* $\subseteq_f$ *m*
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-subset*)

lemma *fmsubset-restrict-fset*[*simp*]: *fmrestrict-fset S m* $\subseteq_f$ *m*
unfolding *fmfilter-alt-defs* **by** (*rule fmfilter-subset*)

lift-definition *fset-of-fmap* :: ('*a*, '*b*) *fmap* ⇒ ('*a* × '*b*) *fset* **is** *Map.graph*
by *simp*

lemma *fset-of-fmap-inj*[*intro*, *simp*]: *inj fset-of-fmap*
apply *rule*
apply *transfer'*
using *inj-on-graph* **unfolding** *inj-def* **by** *auto*

lemma *fset-of-fmap-iff*[*simp*]: (*a*, *b*) |∈| *fset-of-fmap m* ⟷ *fmlookup m a* = *Some b*
by *transfer'* (*auto simp*: *Map.graph-def*)

lemma *fset-of-fmap-iff'*: (*a*, *b*) ∈ *fset* (*fset-of-fmap m*) ⟷ *fmlookup m a* = *Some b*
by *simp*

lift-definition *fmap-of-list* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *fmap*
is *map-of*
parametric *map-of-transfer*
by (*rule finite-dom-map-of*)

lemma *fmap-of-list-simps*[*simp*]:
fmap-of-list [] = *fmempty*
fmap-of-list ((*k*, *v*) # *kvs*) = *fmupd k v* (*fmap-of-list kvs*)
by (*transfer*, *simp add*: *map-upd-def*)+

lemma *fmap-of-list-app*[*simp*]: *fmap-of-list* (*xs* @ *ys*) = *fmap-of-list ys* ++$_f$ *fmap-of-list xs*
by *transfer' simp*

lemma *fmupd-alt-def*: *fmupd k v m* = *m* ++$_f$ *fmap-of-list* [(*k*, *v*)]
by *simp*

lemma *fmpred-of-list*[*intro*]:
assumes ⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *P k v*
shows *fmpred P* (*fmap-of-list xs*)
using *assms*
by (*induction xs*) (*transfer'*; *auto simp*: *map-pred-def*)+

lemma *fmap-of-list-SomeD*: *fmlookup* (*fmap-of-list xs*) *k* = *Some v* ⟹ (*k*, *v*) ∈

set xs
by *transfer'* (*auto dest*: *map-of-SomeD*)

lemma *fmdom-fmap-of-list*[*simp*]: *fmdom* (*fmap-of-list xs*) = *fset-of-list* (*map fst xs*)
by *transfer'* (*simp add*: *dom-map-of-conv-image-fst*)

lift-definition *fmrel-on-fset* :: *'a fset* ⇒ (*'b* ⇒ *'c* ⇒ *bool*) ⇒ (*'a*, *'b*) *fmap* ⇒ (*'a*, *'c*) *fmap* ⇒ *bool*
is *rel-map-on-set*
.

lemma *fmrel-on-fset-alt-def*: *fmrel-on-fset S P m n* ⟷ *fBall S* (λ*x*. *rel-option P* (*fmlookup m x*) (*fmlookup n x*))
by *transfer'* (*auto simp*: *rel-map-on-set-def eq-onp-def rel-fun-def*)

lemma *fmrel-on-fsetI*[*intro*]:
assumes ⋀*x*. *x* |∈| *S* ⟹ *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
shows *fmrel-on-fset S P m n*
by (*simp add*: *assms fmrel-on-fset-alt-def*)

lemma *fmrel-on-fset-mono*[*mono*]: *R* ≤ *Q* ⟹ *fmrel-on-fset S R* ≤ *fmrel-on-fset S Q*
unfolding *fmrel-on-fset-alt-def*[*abs-def*]
using *option.rel-mono* **by** *blast*

lemma *fmrel-on-fsetD*: *x* |∈| *S* ⟹ *fmrel-on-fset S P m n* ⟹ *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
unfolding *fmrel-on-fset-alt-def*
by *auto*

lemma *fmrel-on-fsubset*: *fmrel-on-fset S R m n* ⟹ *T* |⊆| *S* ⟹ *fmrel-on-fset T R m n*
unfolding *fmrel-on-fset-alt-def*
by *auto*

lemma *fmrel-on-fset-unionI*:
fmrel-on-fset A R m n ⟹ *fmrel-on-fset B R m n* ⟹ *fmrel-on-fset* (*A* |∪| *B*) *R m n*
unfolding *fmrel-on-fset-alt-def*
by *auto*

lemma *fmrel-on-fset-updateI*:
assumes *fmrel-on-fset S P m n P* v_1 v_2
shows *fmrel-on-fset* (*finsert k S*) *P* (*fmupd k* v_1 *m*) (*fmupd k* v_2 *n*)
using *assms*
unfolding *fmrel-on-fset-alt-def*
by *auto*

lift-definition *fmimage* :: (*'a*, *'b*) *fmap* ⇒ *'a fset* ⇒ *'b fset* **is** λ*m S*. {*b*|*a b*. *m a* = *Some b* ∧ *a* ∈ *S*}
by (*smt* (*verit*, *del-insts*) *Collect-mono-iff finite-surj ran-alt-def ran-def*)

lemma *fmimage-alt-def*: *fmimage m S* = *fmran* (*fmrestrict-fset S m*)
by *transfer'* (*auto simp*: *ran-def map-restrict-set-def map-filter-def*)

lemma *fmimage-empty*[*simp*]: *fmimage m fempty* = *fempty*
by *transfer' auto*

lemma *fmimage-subset-ran*[*simp*]: *fmimage m S* |⊆| *fmran m*
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmimage-dom*[*simp*]: *fmimage m* (*fmdom m*) = *fmran m*
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmimage-inter*: *fmimage m* (*A* |∩| *B*) |⊆| *fmimage m A* |∩| *fmimage m B*
by *transfer' auto*

lemma *fimage-inter-dom*[*simp*]:
fmimage m (*fmdom m* |∩| *A*) = *fmimage m A*
fmimage m (*A* |∩| *fmdom m*) = *fmimage m A*
by (*transfer'*; *auto*)+

lemma *fmimage-union*[*simp*]: *fmimage m* (*A* |∪| *B*) = *fmimage m A* |∪| *fmimage m B*
by *transfer' auto*

lemma *fmimage-Union*[*simp*]: *fmimage m* (*ffUnion A*) = *ffUnion* (*fmimage m* |‘| *A*)
by *transfer' auto*

lemma *fmimage-filter*[*simp*]: *fmimage* (*fmfilter P m*) *A* = *fmimage m* (*ffilter P A*)
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmimage-drop*[*simp*]: *fmimage* (*fmdrop a m*) *A* = *fmimage m* (*A* − {|*a*|})
by (*simp add*: *fmimage-alt-def*)

lemma *fmimage-drop-fset*[*simp*]: *fmimage* (*fmdrop-fset B m*) *A* = *fmimage m* (*A* − *B*)
by *transfer'* (*auto simp*: *map-filter-def map-drop-set-def*)

lemma *fmimage-restrict-fset*[*simp*]: *fmimage* (*fmrestrict-fset B m*) *A* = *fmimage m* (*A* |∩| *B*)
by *transfer'* (*auto simp*: *map-filter-def map-restrict-set-def*)

lemma *fmfilter-ran*[*simp*]: *fmran* (*fmfilter P m*) = *fmimage m* (*ffilter P* (*fmdom m*))
by *transfer'* (*auto simp*: *ran-def map-filter-def*)

lemma *fmran-drop*[*simp*]: *fmran* (*fmdrop a m*) = *fmimage m* (*fmdom m* − {|*a*|})
by *transfer'* (*auto simp*: *ran-def map-drop-def map-filter-def*)

lemma *fmran-drop-fset*[*simp*]: *fmran* (*fmdrop-fset A m*) = *fmimage m* (*fmdom m* − *A*)
by *transfer'* (*auto simp*: *ran-def map-drop-set-def map-filter-def*)

lemma *fmran-restrict-fset*: *fmran* (*fmrestrict-fset A m*) = *fmimage m* (*fmdom m* |∩| *A*)
by *transfer'* (*auto simp*: *ran-def map-restrict-set-def map-filter-def*)

lemma *fmlookup-image-iff*: *y* |∈| *fmimage m A* ⟷ (∃ *x*. *fmlookup m x* = *Some y* ∧ *x* |∈| *A*)
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmimageI*: *fmlookup m x* = *Some y* ⟹ *x* |∈| *A* ⟹ *y* |∈| *fmimage m A*
by (*auto simp*: *fmlookup-image-iff*)

lemma *fmimageE*[*elim*]:
assumes *y* |∈| *fmimage m A*
obtains *x* **where** *fmlookup m x* = *Some y x* |∈| *A*
using *assms* **by** (*auto simp*: *fmlookup-image-iff*)

lift-definition *fmcomp* :: ('*b*, '*c*) *fmap* ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*c*) *fmap* (**infixl** ‹∘$_f$› *55*)
is *map-comp*
parametric *map-comp-transfer*
by (*rule dom-comp-finite*)

lemma *fmlookup-comp*[*simp*]: *fmlookup* (*m* ∘$_f$ *n*) *x* = *Option.bind* (*fmlookup n x*) (*fmlookup m*)
by *transfer'* (*auto simp*: *map-comp-def split*: *option.splits*)

end

30.4 BNF setup

lift-bnf ('*a*, *fmran'*: '*b*) *fmap* [*wits*: *Map.empty*]
for *map*: *fmmap*
rel: *fmrel*
by *auto*

declare *fmap.pred-mono*[*mono*]

lemma *fmran'-alt-def*: *fmran'* *m* = *fset* (*fmran m*)
including *fset.lifting*
by *transfer'* (*auto simp*: *ran-def fun-eq-iff*)

lemma *fmlookup-ran'-iff*: $y \in fmran'\ m \longleftrightarrow (\exists\, x.\ fmlookup\ m\ x = Some\ y)$
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmran'I*: $fmlookup\ m\ x = Some\ y \Longrightarrow y \in fmran'\ m$
by (*auto simp*: *fmlookup-ran'-iff*)

lemma *fmran'E*[*elim*]:
assumes $y \in fmran'\ m$
obtains x **where** $fmlookup\ m\ x = Some\ y$
using *assms* **by** (*auto simp*: *fmlookup-ran'-iff*)

lemma *fmrel-iff*: $fmrel\ R\ m\ n \longleftrightarrow (\forall\, x.\ rel\text{-}option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x))$
by *transfer'* (*auto simp*: *rel-fun-def*)

lemma *fmrelI*[*intro*]:
assumes $\bigwedge x.\ rel\text{-}option\ R\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
shows $fmrel\ R\ m\ n$
using *assms*
by *transfer' auto*

lemma *fmrel-upd*[*intro*]: $fmrel\ P\ m\ n \Longrightarrow P\ x\ y \Longrightarrow fmrel\ P\ (fmupd\ k\ x\ m)\ (fmupd\ k\ y\ n)$
by *transfer'* (*auto simp*: *map-upd-def rel-fun-def*)

lemma *fmrelD*[*dest*]: $fmrel\ P\ m\ n \Longrightarrow rel\text{-}option\ P\ (fmlookup\ m\ x)\ (fmlookup\ n\ x)$
by *transfer'* (*auto simp*: *rel-fun-def*)

lemma *fmrel-addI*[*intro*]:
assumes *fmrel P m n fmrel P a b*
shows $fmrel\ P\ (m +\!\!+_f a)\ (n +\!\!+_f b)$
by (*smt* (*verit*, *del-insts*) *assms domIff fmdom.rep-eq fmlookup-add fmrel-iff option.rel-sel*)

lemma *fmrel-cases*[*consumes 1*]:
assumes *fmrel P m n*
obtains (*none*) *fmlookup m x = None fmlookup n x = None*
| (*some*) *a b* **where** *fmlookup m x = Some a fmlookup n x = Some b P a b*
proof −
from *assms* **have** *rel-option P* (*fmlookup m x*) (*fmlookup n x*)
by *auto*
then show *thesis*
using *none some*
by (*cases rule*: *option.rel-cases*) *auto*
qed

lemma *fmrel-filter*[*intro*]: $fmrel\ P\ m\ n \Longrightarrow fmrel\ P\ (fmfilter\ Q\ m)\ (fmfilter\ Q\ n)$
unfolding *fmrel-iff* **by** *auto*

lemma *fmrel-drop*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop a m*) (*fmdrop a n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-drop-set*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop-set A m*) (*fmdrop-set A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-drop-fset*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmdrop-fset A m*) (*fmdrop-fset A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-restrict-set*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmrestrict-set A m*) (*fmrestrict-set A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-restrict-fset*[*intro*]: *fmrel P m n* ⟹ *fmrel P* (*fmrestrict-fset A m*) (*fmrestrict-fset A n*)
unfolding *fmfilter-alt-defs* **by** *blast*

lemma *fmrel-on-fset-fmrel-restrict*:
fmrel-on-fset S P m n ⟷ *fmrel P* (*fmrestrict-fset S m*) (*fmrestrict-fset S n*)
unfolding *fmrel-on-fset-alt-def fmrel-iff*
by *auto*

lemma *fmrel-on-fset-refl-strong*:
assumes ⋀*x y*. *x* |∈| *S* ⟹ *fmlookup m x* = *Some y* ⟹ *P y y*
shows *fmrel-on-fset S P m m*
unfolding *fmrel-on-fset-fmrel-restrict fmrel-iff*
using *assms*
by (*simp add*: *option.rel-sel*)

lemma *fmrel-on-fset-addI*:
assumes *fmrel-on-fset S P m n fmrel-on-fset S P a b*
shows *fmrel-on-fset S P* (*m* ++$_f$ *a*) (*n* ++$_f$ *b*)
using *assms*
unfolding *fmrel-on-fset-fmrel-restrict*
by *auto*

lemma *fmrel-fmdom-eq*:
assumes *fmrel P x y*
shows *fmdom x* = *fmdom y*
proof −
have *a* |∈| *fmdom x* ⟷ *a* |∈| *fmdom y* **for** *a*
proof −
have *rel-option P* (*fmlookup x a*) (*fmlookup y a*)
using *assms* **by** (*simp add*: *fmrel-iff*)
thus *?thesis*
by *cases* (*auto intro*: *fmdomI*)

```
    qed
  thus ?thesis
    by auto
qed

lemma fmrel-fmdom'-eq: fmrel P x y ⟹ fmdom' x = fmdom' y
unfolding fmdom'-alt-def
by (metis fmrel-fmdom-eq)

lemma fmrel-rel-fmran:
  assumes fmrel P x y
  shows rel-fset P (fmran x) (fmran y)
proof −
  {
    fix b
    assume b |∈| fmran x
    then obtain a where fmlookup x a = Some b
      by auto
    moreover have rel-option P (fmlookup x a) (fmlookup y a)
      using assms by auto
    ultimately have ∃ b'. b' |∈| fmran y ∧ P b b'
      by (metis option-rel-Some1 fmranI)
  }
  moreover
  {
    fix b
    assume b |∈| fmran y
    then obtain a where fmlookup y a = Some b
      by auto
    moreover have rel-option P (fmlookup x a) (fmlookup y a)
      using assms by auto
    ultimately have ∃ b'. b' |∈| fmran x ∧ P b' b
      by (metis option-rel-Some2 fmranI)
  }
  ultimately show ?thesis
    unfolding rel-fset-alt-def
    by auto
qed

lemma fmrel-rel-fmran': fmrel P x y ⟹ rel-set P (fmran' x) (fmran' y)
  unfolding fmran'-alt-def
  by (metis fmrel-rel-fmran rel-fset-fset)

lemma pred-fmap-fmpred[simp]: pred-fmap P = fmpred (λ-. P)
  unfolding fmap.pred-set fmran'-alt-def
  using fmranI by fastforce

lemma pred-fmap-id[simp]: pred-fmap id (fmmap f m) ⟷ pred-fmap f m
  unfolding fmap.pred-set fmap.set-map
```

by *simp*

lemma *pred-fmapD*: *pred-fmap P m* $\Longrightarrow$ *x* $|\in|$ *fmran m* $\Longrightarrow$ *P x*
by *auto*

lemma *fmlookup-map*[*simp*]: *fmlookup* (*fmmap f m*) *x* = *map-option f* (*fmlookup m x*)
by *transfer'* *auto*

lemma *fmpred-map*[*simp*]: *fmpred P* (*fmmap f m*) $\longleftrightarrow$ *fmpred* (λ*k v*. *P k* (*f v*)) *m*
unfolding *fmpred-iff pred-fmap-def fmap.set-map*
by *auto*

lemma *fmpred-id*[*simp*]: *fmpred* (λ-. *id*) (*fmmap f m*) $\longleftrightarrow$ *fmpred* (λ-. *f*) *m*
by *simp*

lemma *fmmap-add*[*simp*]: *fmmap f* (*m* $+\!\!+_f$ *n*) = *fmmap f m* $+\!\!+_f$ *fmmap f n*
by *transfer'* (*auto simp*: *map-add-def fun-eq-iff split*: *option.splits*)

lemma *fmmap-empty*[*simp*]: *fmmap f fmempty* = *fmempty*
by *transfer auto*

lemma *fmdom-map*[*simp*]: *fmdom* (*fmmap f m*) = *fmdom m*
including *fset.lifting*
by *transfer'* *simp*

lemma *fmdom'-map*[*simp*]: *fmdom'* (*fmmap f m*) = *fmdom' m*
by *transfer'* *simp*

lemma *fmran-fmmap*[*simp*]: *fmran* (*fmmap f m*) = *f* $|`|$ *fmran m*
including *fset.lifting*
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmran'-fmmap*[*simp*]: *fmran'* (*fmmap f m*) = *f* ‘ *fmran' m*
by *transfer'* (*auto simp*: *ran-def*)

lemma *fmfilter-fmmap*[*simp*]: *fmfilter P* (*fmmap f m*) = *fmmap f* (*fmfilter P m*)
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmdrop-fmmap*[*simp*]: *fmdrop a* (*fmmap f m*) = *fmmap f* (*fmdrop a m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-fmmap*[*simp*]: *fmdrop-set A* (*fmmap f m*) = *fmmap f* (*fmdrop-set A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmmap*[*simp*]: *fmdrop-fset A* (*fmmap f m*) = *fmmap f* (*fmdrop-fset A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-fmmap*[*simp*]: *fmrestrict-set A* (*fmmap f m*) = *fmmap f* (*fmrestrict-set A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-fmmap*[*simp*]: *fmrestrict-fset A* (*fmmap f m*) = *fmmap f* (*fmrestrict-fset A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmmap-subset*[*intro*]: $m \subseteq_f n \Longrightarrow$ *fmmap f m* $\subseteq_f$ *fmmap f n*
by *transfer'* (*auto simp*: *map-le-def*)

lemma *fmmap-fset-of-fmap*: *fset-of-fmap* (*fmmap f m*) = (λ(*k*, *v*). (*k*, *f v*)) |‘| *fset-of-fmap m*
including *fset.lifting*
by *transfer'* (*auto simp*: *Map.graph-def*)

lemma *fmmap-fmupd*: *fmmap f* (*fmupd x y m*) = *fmupd x* (*f y*) (*fmmap f m*)
by *transfer'* (*auto simp*: *fun-eq-iff map-upd-def*)

30.5 *size* setup

definition *size-fmap* :: ('*a* ⇒ *nat*) ⇒ ('*b* ⇒ *nat*) ⇒ ('*a*, '*b*) *fmap* ⇒ *nat* **where**
[*simp*]: *size-fmap f g m* = *size-fset* (λ(*a*, *b*). *f a* + *g b*) (*fset-of-fmap m*)

instantiation *fmap* :: (*type*, *type*) *size* **begin**

definition *size-fmap* **where**
size-fmap-overloaded-def: *size-fmap* = *Finite-Map.size-fmap* (λ-. *0*) (λ-. *0*)

instance ..

end

lemma *size-fmap-overloaded-simps*[*simp*]: *size x* = *size* (*fset-of-fmap x*)
unfolding *size-fmap-overloaded-def*
by *simp*

lemma *fmap-size-o-map*: *size-fmap f g* ∘ *fmmap h* = *size-fmap f* (*g* ∘ *h*)
proof −
have *inj*: *inj-on* (λ(*k*, *v*). (*k*, *h v*)) (*fset* (*fset-of-fmap m*)) **for** *m*
using *inj-on-def* **by** *force*
show *?thesis*
unfolding *size-fmap-def*
apply (*clarsimp simp*: *fun-eq-iff fmmap-fset-of-fmap sum.reindex* [*OF inj*])
by (*rule sum.cong*) (*auto split*: *prod.splits*)
qed

setup ‹

BNF-LFP-Size.register-size-global ***type-name*** ‹*fmap*› ***const-name*** ‹*size-fmap*›
@{*thm size-fmap-overloaded-def*} @{*thms size-fmap-def size-fmap-overloaded-simps*}
@{*thms fmap-size-o-map*}
›

30.6 Additional operations

lift-definition *fmmap-keys* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *fmap* ⇒ ('*a*, '*c*) *fmap* **is**
λ*f m a*. *map-option* (*f a*) (*m a*)
unfolding *dom-def*
by *simp*

lemma *fmpred-fmmap-keys*[*simp*]: *fmpred P* (*fmmap-keys f m*) = *fmpred* (λ*a b*. *P a* (*f a b*)) *m*
by *transfer'* (*auto simp*: *map-pred-def split*: *option.splits*)

lemma *fmdom-fmmap-keys*[*simp*]: *fmdom* (*fmmap-keys f m*) = *fmdom m*
including *fset.lifting*
by *transfer' auto*

lemma *fmlookup-fmmap-keys*[*simp*]: *fmlookup* (*fmmap-keys f m*) *x* = *map-option* (*f x*) (*fmlookup m x*)
by *transfer' simp*

lemma *fmfilter-fmmap-keys*[*simp*]: *fmfilter P* (*fmmap-keys f m*) = *fmmap-keys f* (*fmfilter P m*)
by *transfer'* (*auto simp*: *map-filter-def*)

lemma *fmdrop-fmmap-keys*[*simp*]: *fmdrop a* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop a m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-set-fmmap-keys*[*simp*]: *fmdrop-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-set A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmdrop-fset-fmmap-keys*[*simp*]: *fmdrop-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmdrop-fset A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-set-fmmap-keys*[*simp*]: *fmrestrict-set A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-set A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmrestrict-fset-fmmap-keys*[*simp*]: *fmrestrict-fset A* (*fmmap-keys f m*) = *fmmap-keys f* (*fmrestrict-fset A m*)
unfolding *fmfilter-alt-defs* **by** *simp*

lemma *fmmap-keys-subset*[*intro*]: *m* ⊆$_f$ *n* ⟹ *fmmap-keys f m* ⊆$_f$ *fmmap-keys f*

n
by *transfer'* (*auto simp*: *map-le-def dom-def*)

definition *sorted-list-of-fmap* :: ('*a*::*linorder*, '*b*) *fmap* ⇒ ('*a* × '*b*) *list* **where**
sorted-list-of-fmap m = *map* (λ*k*. (*k*, *the* (*fmlookup m k*))) (*sorted-list-of-fset* (*fmdom m*))

lemma *list-all-sorted-list*[*simp*]: *list-all P* (*sorted-list-of-fmap m*) = *fmpred* (*curry P*) *m*
unfolding *sorted-list-of-fmap-def curry-def list.pred-map*
by (*smt* (*verit*, *best*) *Ball-set comp-def fmpred-alt-def sorted-list-of-fset-simps*(*1*))

lemma *map-of-sorted-list*[*simp*]: *map-of* (*sorted-list-of-fmap m*) = *fmlookup m*
unfolding *sorted-list-of-fmap-def*
including *fset.lifting*
by *transfer* (*simp add*: *map-of-map-keys*)

30.7 Additional properties

lemma *fmchoice'*:
assumes *finite S* ∀ *x* ∈ *S*. ∃ *y*. *Q x y*
shows ∃ *m*. *fmdom' m* = *S* ∧ *fmpred Q m*
proof −
obtain *f* **where** *f*: *Q x* (*f x*) **if** *x* ∈ *S* **for** *x*
using *assms* **by** *metis*
define *f'* **where** *f' x* = (*if x* ∈ *S then Some* (*f x*) *else None*) **for** *x*

have *eq-onp* (λ*m*. *finite* (*dom m*)) *f' f'*
unfolding *eq-onp-def f'-def dom-def* **using** *assms* **by** *auto*

show *?thesis*
apply (*rule exI*[**where** *x* = *Abs-fmap f'*])
apply (*subst fmpred.abs-eq*, *fact*)
apply (*subst fmdom'.abs-eq*, *fact*)
unfolding *f'-def dom-def map-pred-def* **using** *f*
by *auto*
qed

30.8 Lifting/transfer setup

context includes *lifting-syntax* **begin**

lemma *fmempty-transfer*[*simp*, *intro*, *transfer-rule*]: *fmrel P fmempty fmempty*
by *transfer auto*

lemma *fmadd-transfer*[*transfer-rule*]:
(*fmrel P* ===> *fmrel P* ===> *fmrel P*) *fmadd fmadd*
by (*intro fmrel-addI rel-funI*)

lemma *fmupd-transfer*[*transfer-rule*]:

```
  ((=) ===> P ===> fmrel P ===> fmrel P) fmupd fmupd
  by auto

end

lemma Quotient-fmap-bnf[quot-map]:
  assumes Quotient R Abs Rep T
  shows Quotient (fmrel R) (fmmap Abs) (fmmap Rep) (fmrel T)
unfolding Quotient-alt-def4 proof safe
  fix m n
  assume fmrel T m n
  then have fmlookup (fmmap Abs m) x = fmlookup n x for x
    using assms unfolding Quotient-alt-def
      by (cases rule: fmrel-cases[where x = x]) auto
  then show fmmap Abs m = n
    by (rule fmap-ext)
next
  fix m
  show fmrel T (fmmap Rep m) m
    unfolding fmap.rel-map
    by (metis (mono-tags) Quotient-alt-def assms fmap.rel-refl)
next
  from assms have R = T OO T^-1-1
    unfolding Quotient-alt-def4 by simp
  then show fmrel R = fmrel T OO (fmrel T)^-1-1
    by (simp add: fmap.rel-compp fmap.rel-conversep)
qed
```

30.9 View as datatype

```
lemma fmap-distinct[simp]:
  fmempty ≠ fmupd k v m
  fmupd k v m ≠ fmempty
  by (transfer'; auto simp: map-upd-def fun-eq-iff)+

lifting-update fmap.lifting

lemma fmap-exhaust[cases type: fmap]:
  obtains (fmempty) m = fmempty
        | (fmupd) x y m' where m = fmupd x y m' x |∉| fmdom m'
using that including fmap.lifting and fset.lifting
proof transfer
  fix m P
  assume finite (dom m)
  assume empty: P if m = Map.empty
  assume map-upd: P if finite (dom m') m = map-upd x y m' x ∉ dom m' for x
y m'

  show P
```

```
    proof (cases m = Map.empty)
      case True thus ?thesis using empty by simp
    next
      case False
      hence dom m ≠ {} by simp
      then obtain x where x ∈ dom m by blast

      let ?m' = map-drop x m

      show ?thesis
        proof (rule map-upd)
          show finite (dom ?m')
            using ‹finite (dom m)›
            unfolding map-drop-def
            by auto
        next
          show m = map-upd x (the (m x)) ?m'
            using ‹x ∈ dom m› unfolding map-drop-def map-filter-def map-upd-def
            by auto
        next
          show x ∉ dom ?m'
            unfolding map-drop-def map-filter-def
            by auto
        qed
    qed
qed

lemma fmap-induct[case-names fmempty fmupd, induct type: fmap]:
  assumes P fmempty
  assumes (⋀x y m. P m ⟹ fmlookup m x = None ⟹ P (fmupd x y m))
  shows P m
proof (induction fmdom m arbitrary: m rule: fset-induct-stronger)
  case empty
  hence m = fmempty
    by (metis fmrestrict-fset-dom fmrestrict-fset-null)
  with assms show ?case
    by simp
next
  case (insert x S)
  hence S = fmdom (fmdrop x m)
    by auto
  with insert have P (fmdrop x m)
    by auto
  moreover
  obtain y where fmlookup m x = Some y
    using insert.hyps by force
  hence m = fmupd x y (fmdrop x m)
    by (auto intro: fmap-ext)
  ultimately show ?case
```

by (*metis assms*(*2*) *fmdrop-lookup*)
qed

30.10 Code setup

instantiation *fmap* :: (*type*, *equal*) *equal* **begin**

definition *equal-fmap* ≡ *fmrel HOL.equal*

instance proof
fix *m n* :: ('*a*, '*b*) *fmap*
have *fmrel* (=) *m n* ⟷ (*m* = *n*)
by *transfer'* (*simp add*: *option.rel-eq rel-fun-eq*)
then show *equal-class.equal m n* ⟷ (*m* = *n*)
unfolding *equal-fmap-def*
by (*simp add*: *equal-eq*[*abs-def*])
qed

end

lemma *fBall-alt-def*: *fBall S P* ⟷ (∀ *x*. *x* |∈| *S* ⟶ *P x*)
by *force*

lemma *fmrel-code*:
fmrel R m n ⟷
fBall (*fmdom m*) (λ*x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*)) ∧
fBall (*fmdom n*) (λ*x*. *rel-option R* (*fmlookup m x*) (*fmlookup n x*))
unfolding *fmrel-iff fmlookup-dom-iff fBall-alt-def*
by (*metis option.collapse option.rel-sel*)

lemmas [*code*] =
fmrel-code
fmran'-alt-def
fmdom'-alt-def
fmfilter-alt-defs
pred-fmap-fmpred
fmsubset-alt-def
fmupd-alt-def
fmrel-on-fset-alt-def
fmpred-alt-def

code-datatype *fmap-of-list*
quickcheck-generator *fmap constructors*: *fmap-of-list*

context includes *fset.lifting* **begin**

lemma *fmlookup-of-list*[*code*]: *fmlookup* (*fmap-of-list m*) = *map-of m*
by *transfer simp*

lemma *fmempty-of-list*[*code*]: *fmempty* = *fmap-of-list* []
by *transfer simp*

lemma *fmran-of-list*[*code*]: *fmran* (*fmap-of-list m*) = *snd* |‘| *fset-of-list* (*AList.clearjunk m*)
by *transfer* (*auto simp*: *ran-map-of*)

lemma *fmdom-of-list*[*code*]: *fmdom* (*fmap-of-list m*) = *fst* |‘| *fset-of-list m*
by *transfer* (*auto simp*: *dom-map-of-conv-image-fst*)

lemma *fmfilter-of-list*[*code*]: *fmfilter P* (*fmap-of-list m*) = *fmap-of-list* (*filter* (λ(*k*, -). *P k*) *m*)
by *transfer'* *auto*

lemma *fmadd-of-list*[*code*]: *fmap-of-list m* ++$_f$ *fmap-of-list n* = *fmap-of-list* (*AList.merge m n*)
by *transfer* (*simp add*: *merge-conv'*)

lemma *fmmap-of-list*[*code*]: *fmmap f* (*fmap-of-list m*) = *fmap-of-list* (*map* (*apsnd f*) *m*)
apply *transfer*
by (*metis* (*no-types*, *lifting*) *apsnd-conv map-eq-conv map-of-map old.prod.case old.prod.exhaust*)

lemma *fmmap-keys-of-list*[*code*]:
fmmap-keys f (*fmap-of-list m*) = *fmap-of-list* (*map* (λ(*a*, *b*). (*a*, *f a b*)) *m*)
apply *transfer*
subgoal for *f m* **by** (*induction m*) (*auto simp*: *apsnd-def map-prod-def fun-eq-iff*)
done

lemma *fmimage-of-list*[*code*]:
fmimage (*fmap-of-list m*) *A* = *fset-of-list* (*map snd* (*filter* (λ(*k*, -). *k* |∈| *A*) (*AList.clearjunk m*)))
apply (*subst fmimage-alt-def*)
apply (*subst fmfilter-alt-defs*)
apply (*subst fmfilter-of-list*)
apply (*subst fmran-of-list*)
apply *transfer'*
by (*metis AList.restrict-eq clearjunk-restrict list.set-map*)

lemma *fmcomp-list*[*code*]:
fmap-of-list m ∘$_f$ *fmap-of-list n* = *fmap-of-list* (*AList.compose n m*)
by (*rule fmap-ext*) (*simp add*: *fmlookup-of-list compose-conv map-comp-def split*: *option.splits*)

end

30.11 Instances

lemma *exists-map-of*:
  assumes *finite* (*dom m*) **shows** ∃ *xs*. *map-of xs* = *m*
  using *assms*
proof (*induction dom m arbitrary*: *m*)
  case *empty*
  hence *m* = *Map.empty*
    by *auto*
  moreover have *map-of* [] = *Map.empty*
    by *simp*
  ultimately show *?case*
    by *blast*
next
  case (*insert x F*)
  hence *F* = *dom* (*map-drop x m*)
    unfolding *map-drop-def map-filter-def dom-def* **by** *auto*
  with *insert* **have** ∃ *xs'*. *map-of xs'* = *map-drop x m*
    by *auto*
  then obtain *xs'* **where** *map-of xs'* = *map-drop x m*
    ..
  moreover obtain *y* **where** *m x* = *Some y*
    using *insert* **unfolding** *dom-def* **by** *blast*
  ultimately have *map-of* ((*x*, *y*) # *xs'*) = *m*
    using ‹*insert x F* = *dom m*›
    unfolding *map-drop-def map-filter-def*
    by *auto*
  thus *?case*
    ..
qed

lemma *exists-fmap-of-list*: ∃ *xs*. *fmap-of-list xs* = *m*
by *transfer* (*rule exists-map-of*)

lemma *fmap-of-list-surj*[*simp*, *intro*]: *surj fmap-of-list*
proof −
  have *x* ∈ *range fmap-of-list* **for** *x* :: ('*a*, '*b*) *fmap*
    unfolding *image-iff*
    using *exists-fmap-of-list* **by** (*metis UNIV-I*)
  thus *?thesis* **by** *auto*
qed

instance *fmap* :: (*countable*, *countable*) *countable*
proof
  obtain *to-nat* :: ('*a* × '*b*) *list* ⇒ *nat* **where** *inj to-nat*
    by (*metis ex-inj*)
  moreover have *inj* (*inv fmap-of-list*)
    using *fmap-of-list-surj* **by** (*rule surj-imp-inj-inv*)
  ultimately have *inj* (*to-nat* ∘ *inv fmap-of-list*)
    by (*rule inj-compose*)

thus ∃ *to-nat*::(*'a*, *'b*) *fmap* ⇒ *nat*. *inj to-nat*
by *auto*
qed

instance *fmap* :: (*finite*, *finite*) *finite*
proof
show *finite* (*UNIV* :: (*'a*, *'b*) *fmap set*)
by (*rule finite-imageD*) *auto*
qed

lifting-update *fmap.lifting*
lifting-forget *fmap.lifting*

30.12 Tests

export-code
Ball fset fmrel fmran fmran' fmdom fmdom' fmpred pred-fmap fmsubset fmupd fmrel-on-fset
fmdrop fmdrop-set fmdrop-fset fmrestrict-set fmrestrict-fset fmimage fmlookup fmempty
fmfilter fmadd fmmap fmmap-keys fmcomp
checking *SML Scala Haskell? OCaml?*

— *lifting* through *fmap*

experiment begin

context includes *fset.lifting* **begin**

lift-definition *test1* :: (*'a*, *'b fset*) *fmap* **is** *fmempty* :: (*'a*, *'b set*) *fmap*
by *auto*

lift-definition *test2* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b fset*) *fmap* **is** λ*a b*. *fmupd a* {*b*} *fmempty*
by *auto*

end

end

end

31 Disjoint FSets

theory *Disjoint-FSets*
imports
HOL−Library.Finite-Map
Disjoint-Sets
begin

context
includes *fset.lifting*
begin

lift-definition *fdisjnt* :: *'a fset* ⇒ *'a fset* ⇒ *bool* **is** *disjnt* **.**

lemma *fdisjnt-alt-def*: *fdisjnt M N* ⟷ (*M* |∩| *N* = {||})
by *transfer* (*simp add*: *disjnt-def*)

lemma *fdisjnt-insert*: *x* |∉| *N* ⟹ *fdisjnt M N* ⟹ *fdisjnt* (*finsert x M*) *N*
by *transfer'* (*rule disjnt-insert*)

lemma *fdisjnt-subset-right*: *N'* |⊆| *N* ⟹ *fdisjnt M N* ⟹ *fdisjnt M N'*
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-subset-left*: *N'* |⊆| *N* ⟹ *fdisjnt N M* ⟹ *fdisjnt N' M*
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-union-right*: *fdisjnt M A* ⟹ *fdisjnt M B* ⟹ *fdisjnt M* (*A* |∪| *B*)
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-union-left*: *fdisjnt A M* ⟹ *fdisjnt B M* ⟹ *fdisjnt* (*A* |∪| *B*) *M*
unfolding *fdisjnt-alt-def* **by** *auto*

lemma *fdisjnt-swap*: *fdisjnt M N* ⟹ *fdisjnt N M*
including *fset.lifting* **by** *transfer'* (*auto simp*: *disjnt-def*)

lemma *distinct-append-fset*:
assumes *distinct xs distinct ys fdisjnt* (*fset-of-list xs*) (*fset-of-list ys*)
shows *distinct* (*xs* @ *ys*)
using *assms*
by *transfer'* (*simp add*: *disjnt-def*)

lemma *fdisjnt-contrI*:
assumes ⋀*x*. *x* |∈| *M* ⟹ *x* |∈| *N* ⟹ *False*
shows *fdisjnt M N*
using *assms*
by *transfer'* (*auto simp*: *disjnt-def*)

lemma *fdisjnt-Union-left*: *fdisjnt* (*ffUnion S*) *T* ⟷ *fBall S* (λ*S*. *fdisjnt S T*)
by *transfer'* (*auto simp*: *disjnt-def*)

lemma *fdisjnt-Union-right*: *fdisjnt T* (*ffUnion S*) ⟷ *fBall S* (λ*S*. *fdisjnt T S*)
by *transfer'* (*auto simp*: *disjnt-def*)

lemma *fdisjnt-ge-max*: *fBall X* (λ*x*. *x* > *fMax Y*) ⟹ *fdisjnt X Y*
by *transfer* (*auto intro*: *disjnt-ge-max*)

end

lemma *fmadd-disjnt*: *fdisjnt* (*fmdom m*) (*fmdom n*) $\Longrightarrow$ $m ++_f n = n ++_f m$
unfolding *fdisjnt-alt-def*
including *fset.lifting* **and** *fmap.lifting*
apply *transfer*
apply (*rule ext*)
apply (*auto simp*: *map-add-def split*: *option.splits*)
done

end

32 Lists with elements distinct as canonical example for datatype invariants

theory *Dlist*
imports *Confluent-Quotient*
begin

32.1 The type of distinct lists

typedef *'a dlist* = {*xs*::*'a list*. *distinct xs*}
  morphisms *list-of-dlist Abs-dlist*
proof
  show [] ∈ {*xs*. *distinct xs*} **by** *simp*
qed

context begin

qualified definition *dlist-eq* **where** *dlist-eq* = *BNF-Def.vimage2p remdups remdups* (=)

qualified lemma *equivp-dlist-eq*: *equivp dlist-eq*
  unfolding *dlist-eq-def* **by**(*rule equivp-vimage2p*)(*rule identity-equivp*)

qualified definition *abs-dlist* :: *'a list* ⇒ *'a dlist* **where** *abs-dlist* = *Abs-dlist o remdups*

definition *qcr-dlist* :: *'a list* ⇒ *'a dlist* ⇒ *bool* **where** *qcr-dlist x y* ⟷ *y* = *abs-dlist x*

qualified lemma *Quotient-dlist-remdups*: *Quotient dlist-eq abs-dlist list-of-dlist qcr-dlist*
  unfolding *Quotient-def dlist-eq-def qcr-dlist-def vimage2p-def abs-dlist-def*
  by (*auto simp add*: *fun-eq-iff Abs-dlist-inject*
    list-of-dlist[*simplified*] *list-of-dlist-inverse distinct-remdups-id*)

end

locale *Quotient-dlist* **begin**
setup-lifting *Dlist.Quotient-dlist-remdups Dlist.equivp-dlist-eq*[*THEN equivp-reflp2*]
end

setup-lifting *type-definition-dlist*

lemma *dlist-eq-iff*:
dxs = *dys* ⟷ *list-of-dlist dxs* = *list-of-dlist dys*
by (*simp add*: *list-of-dlist-inject*)

lemma *dlist-eqI*:
list-of-dlist dxs = *list-of-dlist dys* ⟹ *dxs* = *dys*
by (*simp add*: *dlist-eq-iff*)

Formal, totalized constructor for *'a dlist*:

definition *Dlist* :: *'a list* ⇒ *'a dlist* **where**
Dlist xs = *Abs-dlist* (*remdups xs*)

lemma *distinct-list-of-dlist* [*simp*, *intro*]:
distinct (*list-of-dlist dxs*)
using *list-of-dlist* [*of dxs*] **by** *simp*

lemma *list-of-dlist-Dlist* [*simp*]:
list-of-dlist (*Dlist xs*) = *remdups xs*
by (*simp add*: *Dlist-def Abs-dlist-inverse*)

lemma *remdups-list-of-dlist* [*simp*]:
remdups (*list-of-dlist dxs*) = *list-of-dlist dxs*
by *simp*

lemma *Dlist-list-of-dlist* [*simp*, *code abstype*]:
Dlist (*list-of-dlist dxs*) = *dxs*
by (*simp add*: *Dlist-def list-of-dlist-inverse distinct-remdups-id*)

Fundamental operations:

context
begin

qualified definition *empty* :: *'a dlist* **where**
empty = *Dlist* []

qualified definition *insert* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
insert x dxs = *Dlist* (*List.insert x* (*list-of-dlist dxs*))

qualified definition *remove* :: *'a* ⇒ *'a dlist* ⇒ *'a dlist* **where**
remove x dxs = *Dlist* (*remove1 x* (*list-of-dlist dxs*))

qualified definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a dlist* ⇒ *'b dlist* **where**

map f dxs = Dlist (remdups (List.map f (list-of-dlist dxs)))

qualified definition *filter* :: *('a ⇒ bool) ⇒ 'a dlist ⇒ 'a dlist* **where**
filter P dxs = Dlist (List.filter P (list-of-dlist dxs))

qualified definition *rotate* :: *nat ⇒ 'a dlist ⇒ 'a dlist* **where**
rotate n dxs = Dlist (List.rotate n (list-of-dlist dxs))

end

Derived operations:

context
begin

qualified definition *null* :: *'a dlist ⇒ bool* **where**
null dxs = List.null (list-of-dlist dxs)

qualified definition *member* :: *'a dlist ⇒ 'a ⇒ bool* **where**
member dxs = List.member (list-of-dlist dxs)

qualified definition *length* :: *'a dlist ⇒ nat* **where**
length dxs = List.length (list-of-dlist dxs)

qualified definition *fold* :: *('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b* **where**
fold f dxs = List.fold f (list-of-dlist dxs)

qualified definition *foldr* :: *('a ⇒ 'b ⇒ 'b) ⇒ 'a dlist ⇒ 'b ⇒ 'b* **where**
foldr f dxs = List.foldr f (list-of-dlist dxs)

end

32.2 Executable version obeying invariant

lemma *list-of-dlist-empty* [*simp*, *code abstract*]:
list-of-dlist Dlist.empty = []
by (*simp add*: *Dlist.empty-def*)

lemma *list-of-dlist-insert* [*simp*, *code abstract*]:
list-of-dlist (Dlist.insert x dxs) = List.insert x (list-of-dlist dxs)
by (*simp add*: *Dlist.insert-def*)

lemma *list-of-dlist-remove* [*simp*, *code abstract*]:
list-of-dlist (Dlist.remove x dxs) = remove1 x (list-of-dlist dxs)
by (*simp add*: *Dlist.remove-def*)

lemma *list-of-dlist-map* [*simp*, *code abstract*]:
list-of-dlist (Dlist.map f dxs) = remdups (List.map f (list-of-dlist dxs))
by (*simp add*: *Dlist.map-def*)

lemma *list-of-dlist-filter* [*simp*, *code abstract*]:

list-of-dlist (*Dlist.filter P dxs*) = *List.filter P* (*list-of-dlist dxs*)
by (*simp add*: *Dlist.filter-def*)

lemma *list-of-dlist-rotate* [*simp*, *code abstract*]:
list-of-dlist (*Dlist.rotate n dxs*) = *List.rotate n* (*list-of-dlist dxs*)
by (*simp add*: *Dlist.rotate-def*)

Explicit executable conversion

definition *dlist-of-list* [*simp*]:
dlist-of-list = *Dlist*

lemma [*code abstract*]:
list-of-dlist (*dlist-of-list xs*) = *remdups xs*
by *simp*

Equality

instantiation *dlist* :: (*equal*) *equal*
begin

definition *HOL.equal dxs dys* ⟷ *HOL.equal* (*list-of-dlist dxs*) (*list-of-dlist dys*)

instance
by *standard* (*simp add*: *equal-dlist-def equal list-of-dlist-inject*)

end

declare *equal-dlist-def* [*code*]

lemma [*code nbe*]: *HOL.equal* (*dxs* :: *'a*::*equal dlist*) *dxs* ⟷ *True*
by (*fact equal-refl*)

32.3 Induction principle and case distinction

lemma *dlist-induct* [*case-names empty insert*, *induct type*: *dlist*]:
assumes *empty*: *P Dlist.empty*
assumes *insrt*: ⋀*x dxs*. ¬ *Dlist.member dxs x* ⟹ *P dxs* ⟹ *P* (*Dlist.insert x dxs*)
shows *P dxs*
proof (*cases dxs*)
case (*Abs-dlist xs*)
then have *distinct xs* **and** *dxs*: *dxs* = *Dlist xs*
by (*simp-all add*: *Dlist-def distinct-remdups-id*)
from ‹*distinct xs*› **have** *P* (*Dlist xs*)
proof (*induct xs*)
case *Nil* **from** *empty* **show** *?case* **by** (*simp add*: *Dlist.empty-def*)
next
case (*Cons x xs*)
then have ¬ *Dlist.member* (*Dlist xs*) *x* **and** *P* (*Dlist xs*)
by (*simp-all add*: *Dlist.member-def*)
with *insrt* **have** *P* (*Dlist.insert x* (*Dlist xs*)) **.**

```
    with Cons show ?case by (simp add: Dlist.insert-def distinct-remdups-id)
  qed
  with dxs show P dxs by simp
qed

lemma dlist-case [cases type: dlist]:
  obtains (empty) dxs = Dlist.empty
    | (insert) x dys where ¬ Dlist.member dys x and dxs = Dlist.insert x dys
proof (cases dxs)
  case (Abs-dlist xs)
  then have dxs: dxs = Dlist xs and distinct: distinct xs
    by (simp-all add: Dlist-def distinct-remdups-id)
  show thesis
  proof (cases xs)
    case Nil with dxs
    have dxs = Dlist.empty by (simp add: Dlist.empty-def)
    with empty show ?thesis .
  next
    case (Cons x xs)
    with dxs distinct have ¬ Dlist.member (Dlist xs) x
      and dxs = Dlist.insert x (Dlist xs)
      by (simp-all add: Dlist.member-def Dlist.insert-def distinct-remdups-id)
    with insert show ?thesis .
  qed
qed
```

32.4 Functorial structure

```
functor map: map
  by (simp-all add: remdups-map-remdups fun-eq-iff dlist-eq-iff)
```

32.5 Quickcheck generators

```
quickcheck-generator dlist predicate: distinct constructors: Dlist.empty, Dlist.insert
```

32.6 BNF instance

```
context begin

qualified inductive double :: 'a list ⇒ 'a list ⇒ bool where
  double (xs @ ys) (xs @ x # ys) if x ∈ set ys

qualified lemma strong-confluentp-double: strong-confluentp double
proof
  fix xs ys zs :: 'a list
  assume ys: double xs ys and zs: double xs zs
  consider (left) as y bs z cs where xs = as @ bs @ cs ys = as @ y # bs @ cs zs
= as @ bs @ z # cs y ∈ set (bs @ cs) z ∈ set cs
    | (right) as y bs z cs where xs = as @ bs @ cs ys = as @ bs @ y # cs zs = as
@ z # bs @ cs y ∈ set cs z ∈ set (bs @ cs)
```

```
  proof −
    show thesis using ys zs
     by(clarsimp simp add: double.simps append-eq-append-conv2)(auto intro: that)
  qed
  then show ∃ us. double** ys us ∧ double== zs us
  proof cases
    case left
    let ?us = as @ y # bs @ z # cs
    have double ys ?us double zs ?us using left
      by(auto 4 4 simp add: double.simps)(metis append-Cons append-assoc)+
    then show ?thesis by blast
  next
    case right
    let ?us = as @ z # bs @ y # cs
    have double ys ?us double zs ?us using right
      by(auto 4 4 simp add: double.simps)(metis append-Cons append-assoc)+
    then show ?thesis by blast
  qed
qed

qualified lemma double-Cons1 [simp]: double xs (x # xs) if x ∈ set xs
  using double.intros[of x xs []] that by simp

qualified lemma double-Cons-same [simp]: double xs ys ⟹ double (x # xs) (x
# ys)
  by(auto simp add: double.simps Cons-eq-append-conv)

qualified lemma doubles-Cons-same: double** xs ys ⟹ double** (x # xs) (x #
ys)
  by(induction rule: rtranclp-induct)(auto intro: rtranclp.rtrancl-into-rtrancl)

qualified lemma remdups-into-doubles: double** (remdups xs) xs
  by(induction xs)(auto intro: doubles-Cons-same rtranclp.rtrancl-into-rtrancl)

qualified lemma dlist-eq-into-doubles: Dlist.dlist-eq ≤ equivclp double
  by(auto 4 4 simp add: Dlist.dlist-eq-def vimage2p-def
    intro: equivclp-trans converse-rtranclp-into-equivclp rtranclp-into-equivclp remdups-into-doubles)

qualified lemma factor-double-map: double (map f xs) ys ⟹ ∃ zs. Dlist.dlist-eq
xs zs ∧ ys = map f zs ∧ set zs ⊆ set xs
 by(auto simp add: double.simps Dlist.dlist-eq-def vimage2p-def map-eq-append-conv)
    (metis (no-types, opaque-lifting) list.simps(9) map-append remdups.simps(2)
remdups-append2 set-append set-eq-subset set-remdups)

qualified lemma dlist-eq-set-eq: Dlist.dlist-eq xs ys ⟹ set xs = set ys
  by(simp add: Dlist.dlist-eq-def vimage2p-def)(metis set-remdups)

qualified lemma dlist-eq-map-respect: Dlist.dlist-eq xs ys ⟹ Dlist.dlist-eq (map
f xs) (map f ys)
```

by(*clarsimp simp add*: *Dlist.dlist-eq-def vimage2p-def*)(*metis remdups-map-remdups*)

qualified lemma *confluent-quotient-dlist*:
confluent-quotient double Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq Dlist.dlist-eq
(*map fst*) (*map snd*) (*map fst*) (*map snd*) *list-all2 list-all2 list-all2 set set*
by(*unfold-locales*)(*auto intro*: *strong-confluentp-imp-confluentp strong-confluentp-double dest*: *factor-double-map dlist-eq-into-doubles*[*THEN predicate2D*] *dlist-eq-set-eq simp add*: *list.in-rel list.rel-compp dlist-eq-map-respect Dlist.equivp-dlist-eq equivp-imp-transp*)

lifting-update *dlist.lifting*
lifting-forget *dlist.lifting*

end

context begin
interpretation *Quotient-dlist*: *Quotient-dlist* **.**

lift-bnf (*plugins del*: *code*) *'a dlist*
subgoal for *A B* **by**(*rule confluent-quotient.subdistributivity*[*OF Dlist.confluent-quotient-dlist*])
subgoal by(*force dest*: *Dlist.dlist-eq-set-eq intro*: *equivp-reflp*[*OF Dlist.equivp-dlist-eq*])
done

qualified lemma *list-of-dlist-transfer*[*transfer-rule*]:
bi-unique R ⟹ (*rel-fun* (*Quotient-dlist.pcr-dlist R*) (*list-all2 R*)) *remdups list-of-dlist*
unfolding *rel-fun-def Quotient-dlist.pcr-dlist-def qcr-dlist-def Dlist.abs-dlist-def*
by (*auto simp*: *Abs-dlist-inverse intro*!: *remdups-transfer*[*THEN rel-funD*])

lemma *list-of-dlist-map-dlist*[*simp*]:
list-of-dlist (*map-dlist f xs*) = *remdups* (*map f* (*list-of-dlist xs*))
by *transfer* (*auto simp*: *remdups-map-remdups*)

end

end

33 Type of dual ordered lattices

theory *Dual-Ordered-Lattice*
imports *Main*
begin

The *dual* of an ordered structure is an isomorphic copy of the underlying type, with the $\leq$ relation defined as the inverse of the original one.

The class of lattices is closed under formation of dual structures. This means that for any theorem of lattice theory, the dualized statement holds as well; this important fact simplifies many proofs of lattice theory.

typedef *'a dual = UNIV :: 'a set*
morphisms *undual dual* **..**

setup-lifting *type-definition-dual*

code-datatype *dual*

lemma *dual-eqI*:
x = y **if** *undual x = undual y*
using *that* **by** *transfer assumption*

lemma *dual-eq-iff*:
x = y ⟷ undual x = undual y
by *transfer simp*

lemma *eq-dual-iff* [*iff*]:
dual x = dual y ⟷ x = y
by *transfer simp*

lemma *undual-dual* [*simp*, *code*]:
undual (*dual x*) = *x*
by *transfer rule*

lemma *dual-undual* [*simp*]:
dual (*undual x*) = *x*
by *transfer rule*

lemma *undual-comp-dual* [*simp*]:
undual ∘ dual = id
by (*simp add*: *fun-eq-iff*)

lemma *dual-comp-undual* [*simp*]:
dual ∘ undual = id
by (*simp add*: *fun-eq-iff*)

lemma *inj-dual*:
inj dual
by (*rule injI*) *simp*

lemma *inj-undual*:
inj undual
by (*rule injI*) (*rule dual-eqI*)

lemma *surj-dual*:
surj dual
by (*rule surjI* [*of - undual*]) *simp*

lemma *surj-undual*:
surj undual

by (*rule surjI* [*of - dual*]) *simp*

lemma *bij-dual*:
bij dual
using *inj-dual surj-dual* **by** (*rule bijI*)

lemma *bij-undual*:
bij undual
using *inj-undual surj-undual* **by** (*rule bijI*)

instance *dual* :: (*finite*) *finite*
proof
from *finite* **have** *finite* (*range dual* :: *'a dual set*)
by (*rule finite-imageI*)
then show *finite* (*UNIV* :: *'a dual set*)
by (*simp add*: *surj-dual*)
qed

instantiation *dual* :: (*equal*) *equal*
begin

lift-definition *equal-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is *HOL.equal* **.**

instance
by (*standard*; *transfer*) (*simp add*: *equal*)

end

33.1 Pointwise ordering

instantiation *dual* :: (*ord*) *ord*
begin

lift-definition *less-eq-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is (≥) **.**

lift-definition *less-dual* :: *'a dual* ⇒ *'a dual* ⇒ *bool*
is (>) **.**

instance ..

end

lemma *dual-less-eqI*:
$x \leq y$ **if** *undual* $y \leq$ *undual* x
using *that* **by** *transfer assumption*

lemma *dual-less-eq-iff*:

$x \leq y \longleftrightarrow \mathit{undual}\ y \leq \mathit{undual}\ x$
by *transfer simp*

lemma *less-eq-dual-iff* [*iff*]:
$\mathit{dual}\ x \leq \mathit{dual}\ y \longleftrightarrow y \leq x$
by *transfer simp*

lemma *dual-lessI*:
$x < y$ **if** $\mathit{undual}\ y < \mathit{undual}\ x$
using *that* **by** *transfer assumption*

lemma *dual-less-iff*:
$x < y \longleftrightarrow \mathit{undual}\ y < \mathit{undual}\ x$
by *transfer simp*

lemma *less-dual-iff* [*iff*]:
$\mathit{dual}\ x < \mathit{dual}\ y \longleftrightarrow y < x$
by *transfer simp*

instance *dual* :: (*preorder*) *preorder*
by (*standard*; *transfer*) (*auto simp add*: *less-le-not-le intro*: *order-trans*)

instance *dual* :: (*order*) *order*
by (*standard*; *transfer*) *simp*

33.2 Binary infimum and supremum

instantiation *dual* :: (*sup*) *inf*
begin

lift-definition *inf-dual* :: *'a dual* ⇒ *'a dual* ⇒ *'a dual*
is *sup* **.**

instance ..

end

lemma *undual-inf-eq* [*simp*]:
$\mathit{undual}\ (\mathit{inf}\ x\ y) = \mathit{sup}\ (\mathit{undual}\ x)\ (\mathit{undual}\ y)$
by (*fact inf-dual.rep-eq*)

lemma *dual-sup-eq* [*simp*]:
$\mathit{dual}\ (\mathit{sup}\ x\ y) = \mathit{inf}\ (\mathit{dual}\ x)\ (\mathit{dual}\ y)$
by *transfer rule*

instantiation *dual* :: (*inf*) *sup*
begin

lift-definition *sup-dual* :: *'a dual* ⇒ *'a dual* ⇒ *'a dual*

is *inf* **.**

instance ..

end

lemma *undual-sup-eq* [*simp*]:
undual (*sup x y*) = *inf* (*undual x*) (*undual y*)
by (*fact sup-dual.rep-eq*)

lemma *dual-inf-eq* [*simp*]:
dual (*inf x y*) = *sup* (*dual x*) (*dual y*)
by *transfer simp*

instance *dual* :: (*semilattice-sup*) *semilattice-inf*
by (*standard*; *transfer*) *simp-all*

instance *dual* :: (*semilattice-inf*) *semilattice-sup*
by (*standard*; *transfer*) *simp-all*

instance *dual* :: (*lattice*) *lattice* **..**

instance *dual* :: (*distrib-lattice*) *distrib-lattice*
by (*standard*; *transfer*) (*fact inf-sup-distrib1*)

33.3 Top and bottom elements

instantiation *dual* :: (*top*) *bot*
begin

lift-definition *bot-dual* :: *'a dual*
is *top* **.**

instance ..

end

lemma *undual-bot-eq* [*simp*]:
undual bot = *top*
by (*fact bot-dual.rep-eq*)

lemma *dual-top-eq* [*simp*]:
dual top = *bot*
by *transfer rule*

instantiation *dual* :: (*bot*) *top*
begin

lift-definition *top-dual* :: *'a dual*

is *bot* **.**

instance ..

end

lemma *undual-top-eq* [*simp*]:
undual top = *bot*
by (*fact top-dual.rep-eq*)

lemma *dual-bot-eq* [*simp*]:
dual bot = *top*
by *transfer rule*

instance *dual* :: (*order-top*) *order-bot*
by (*standard*; *transfer*) *simp*

instance *dual* :: (*order-bot*) *order-top*
by (*standard*; *transfer*) *simp*

instance *dual* :: (*bounded-lattice-top*) *bounded-lattice-bot* **..**

instance *dual* :: (*bounded-lattice-bot*) *bounded-lattice-top* **..**

instance *dual* :: (*bounded-lattice*) *bounded-lattice* **..**

33.4 Complement

instantiation *dual* :: (*uminus*) *uminus*
begin

lift-definition *uminus-dual* :: *'a dual* ⇒ *'a dual*
is *uminus* **.**

instance ..

end

lemma *undual-uminus-eq* [*simp*]:
undual (− *x*) = − *undual x*
by (*fact uminus-dual.rep-eq*)

lemma *dual-uminus-eq* [*simp*]:
dual (− *x*) = − *dual x*
by *transfer rule*

instantiation *dual* :: (*boolean-algebra*) *boolean-algebra*
begin

lift-definition *minus-dual* :: *'a dual ⇒ 'a dual ⇒ 'a dual*
is *λx y. − (y − x)* **.**

instance
by (*standard*; *transfer*) (*simp-all add*: *diff-eq ac-simps*)

end

lemma *undual-minus-eq* [*simp*]:
undual (x − y) = − (undual y − undual x)
by (*fact minus-dual.rep-eq*)

lemma *dual-minus-eq* [*simp*]:
dual (x − y) = − (dual y − dual x)
by *transfer simp*

33.5 Complete lattice operations

The class of complete lattices is closed under formation of dual structures.

instantiation *dual* :: (*Sup*) *Inf*
begin

lift-definition *Inf-dual* :: *'a dual set ⇒ 'a dual*
is *Sup* **.**

instance ..

end

lemma *undual-Inf-eq* [*simp*]:
undual (Inf A) = Sup (undual ‘ A)
by (*fact Inf-dual.rep-eq*)

lemma *dual-Sup-eq* [*simp*]:
dual (Sup A) = Inf (dual ‘ A)
by *transfer simp*

instantiation *dual* :: (*Inf*) *Sup*
begin

lift-definition *Sup-dual* :: *'a dual set ⇒ 'a dual*
is *Inf* **.**

instance ..

end

lemma *undual-Sup-eq* [*simp*]:
undual (Sup A) = Inf (undual ‘ A)

by (*fact Sup-dual.rep-eq*)

lemma *dual-Inf-eq* [*simp*]:
dual (*Inf A*) = *Sup* (*dual* ' *A*)
by *transfer simp*

instance *dual* :: (*complete-lattice*) *complete-lattice*
by (*standard*; *transfer*) (*auto intro*: *Inf-lower Sup-upper Inf-greatest Sup-least*)

context
fixes *f* :: *'a*::*complete-lattice* ⇒ *'a*
and *g* :: *'a dual* ⇒ *'a dual*
assumes *mono f*
defines *g* ≡ *dual* ∘ *f* ∘ *undual*
begin

private lemma *mono-dual*:
mono g
proof
fix *x y* :: *'a dual*
assume *x* ≤ *y*
then have *undual y* ≤ *undual x*
by (*simp add*: *dual-less-eq-iff*)
with ‹*mono f*› **have** *f* (*undual y*) ≤ *f* (*undual x*)
by (*rule monoD*)
then have (*dual* ∘ *f* ∘ *undual*) *x* ≤ (*dual* ∘ *f* ∘ *undual*) *y*
by *simp*
then show *g x* ≤ *g y*
by (*simp add*: *g-def*)
qed

lemma *lfp-dual-gfp*:
lfp f = *undual* (*gfp g*) (**is** *?lhs* = *?rhs*)
proof (*rule antisym*)
have *dual* (*undual* (*g* (*gfp g*))) ≤ *dual* (*f* (*undual* (*gfp g*)))
by (*simp add*: *g-def*)
with *mono-dual* **have** *f* (*undual* (*gfp g*)) ≤ *undual* (*gfp g*)
by (*simp add*: *gfp-unfold* [**where** *f* = *g*, *symmetric*] *dual-less-eq-iff*)
then show *?lhs* ≤ *?rhs*
by (*rule lfp-lowerbound*)
from ‹*mono f*› **have** *dual* (*lfp f*) ≤ *dual* (*undual* (*gfp g*))
by (*simp add*: *lfp-fixpoint gfp-upperbound g-def*)
then show *?rhs* ≤ *?lhs*
by (*simp only*: *less-eq-dual-iff*)
qed

lemma *gfp-dual-lfp*:
gfp f = *undual* (*lfp g*)
proof −

have *mono* (λ*x*. *undual* (*undual x*))
by (*rule monoI*) (*simp add*: *dual-less-eq-iff*)
moreover have *mono* (λ*a*. *dual* (*dual* (*f a*)))
using ‹*mono f*› **by** (*auto intro*: *monoI dest*: *monoD*)
moreover have *gfp f* = *gfp* (λ*x*. *undual* (*undual* (*dual* (*dual* (*f x*)))))
by *simp*
ultimately have *undual* (*undual* (*gfp* (λ*x*. *dual*
(*dual* (*f* (*undual* (*undual x*)))))))) =
gfp (λ*x*. *undual* (*undual* (*dual* (*dual* (*f x*)))))
by (*subst gfp-rolling* [**where** *g* = λ*x*. *undual* (*undual x*)]) *simp-all*
then have *gfp f* =
undual
(*undual*
(*gfp* (λ*x*. *dual* (*dual* (*f* (*undual* (*undual x*))))))))
by *simp*
also have ... = *undual* (*undual* (*gfp* (*dual* ∘ *g* ∘ *undual*)))
by (*simp add*: *comp-def g-def*)
also have ... = *undual* (*lfp g*)
using *mono-dual* **by** (*simp only*: *Dual-Ordered-Lattice.lfp-dual-gfp*)
finally show *?thesis* **.**
qed

end

Finally

lifting-update *dual.lifting*
lifting-forget *dual.lifting*

end

34 Equipollence and Other Relations Connected with Cardinality

theory *Equipollence*
imports *FuncSet Countable-Set*
begin

34.1 Eqpoll

definition *eqpoll* :: *'a set* ⇒ *'b set* ⇒ *bool* (**infixl** ‹≈› *50*)
where *eqpoll A B* ≡ ∃*f*. *bij-betw f A B*

definition *lepoll* :: *'a set* ⇒ *'b set* ⇒ *bool* (**infixl** ‹≲› *50*)
where *lepoll A B* ≡ ∃*f*. *inj-on f A* ∧ *f* ‘ *A* ⊆ *B*

definition *lesspoll* :: *'a set* ⇒ *'b set* ⇒ *bool* (**infixl** ‹≺› *50*)
where *A* ≺ *B* == *A* ≲ *B* ∧ ~(*A* ≈ *B*)

lemma *lepoll-def'*: *lepoll A B* ≡ ∃*f*. *inj-on f A* ∧ *f* ∈ *A* → *B*
by (*simp add*: *Pi-iff image-subset-iff lepoll-def*)

lemma *eqpoll-empty-iff-empty* [*simp*]: *A* ≈ {} ⟷ *A*={}
by (*simp add*: *bij-betw-iff-bijections eqpoll-def*)

lemma *lepoll-empty-iff-empty* [*simp*]: *A* ≲ {} ⟷ *A* = {}
by (*auto simp*: *lepoll-def*)

lemma *not-lesspoll-empty*: ¬ *A* ≺ {}
by (*simp add*: *lesspoll-def*)

lemma *lepoll-relational-full*:
assumes ⋀*y*. *y* ∈ *B* ⟹ ∃*x*. *x* ∈ *A* ∧ *R x y*
and ⋀*x y y'*. ⟦*x* ∈ *A*; *y* ∈ *B*; *y'* ∈ *B*; *R x y*; *R x y'*⟧ ⟹ *y* = *y'*
shows *B* ≲ *A*
proof −
obtain *f* **where** *f*: ⋀*y*. *y* ∈ *B* ⟹ *f y* ∈ *A* ∧ *R* (*f y*) *y*
using *assms* **by** *metis*
with *assms* **have** *inj-on f B*
by (*metis inj-onI*)
with *f* **show** *?thesis*
unfolding *lepoll-def* **by** *blast*
qed

lemma *eqpoll-iff-card-of-ordIso*: *A* ≈ *B* ⟷ *ordIso2* (*card-of A*) (*card-of B*)
by (*simp add*: *card-of-ordIso eqpoll-def*)

lemma *eqpoll-refl* [*iff*]: *A* ≈ *A*
by (*simp add*: *card-of-refl eqpoll-iff-card-of-ordIso*)

lemma *eqpoll-finite-iff*: *A* ≈ *B* ⟹ *finite A* ⟷ *finite B*
by (*meson bij-betw-finite eqpoll-def*)

lemma *eqpoll-iff-card*:
assumes *finite A finite B*
shows *A* ≈ *B* ⟷ *card A* = *card B*
using *assms* **by** (*auto simp*: *bij-betw-iff-card eqpoll-def*)

lemma *eqpoll-singleton-iff*: *A* ≈ {*x*} ⟷ (∃*u*. *A* = {*u*})
by (*metis card.infinite card-1-singleton-iff eqpoll-finite-iff eqpoll-iff-card not-less-eq-eq*)

lemma *eqpoll-doubleton-iff*: *A* ≈ {*x*,*y*} ⟷ (∃*u v*. *A* = {*u*,*v*} ∧ (*u*=*v* ⟷ *x*=*y*))
proof (*cases x*=*y*)
case *True*
then show *?thesis*
by (*simp add*: *eqpoll-singleton-iff*)
next

case *False*
then show *?thesis*
by (*smt* (*verit*, *ccfv-threshold*) *card-1-singleton-iff card-Suc-eq-finite eqpoll-finite-iff eqpoll-iff-card finite.insertI singleton-iff*)
qed

lemma *lepoll-antisym*:
assumes $A \lesssim B$ $B \lesssim A$ **shows** $A \approx B$
using *assms* **unfolding** *eqpoll-def lepoll-def* **by** (*metis Schroeder-Bernstein*)

lemma *lepoll-trans* [*trans*]:
assumes $A \lesssim B$ $B \lesssim C$ **shows** $A \lesssim C$
proof −
obtain *f g* **where** *fg*: *inj-on f A inj-on g B* **and** $f : A \to B$ $g \in B \to C$
by (*metis assms lepoll-def'*)
then have $g \circ f \in A \to C$
by *auto*
with *fg* **show** *?thesis*
unfolding *lepoll-def*
by (*metis ‹f ∈ A → B› comp-inj-on image-subset-iff-funcset inj-on-subset*)
qed

lemma *lepoll-trans1* [*trans*]: $[\![A \approx B;\ B \lesssim C]\!] \Longrightarrow A \lesssim C$
by (*meson card-of-ordLeq eqpoll-iff-card-of-ordIso lepoll-def lepoll-trans ordIso-iff-ordLeq*)

lemma *lepoll-trans2* [*trans*]: $[\![A \lesssim B;\ B \approx C]\!] \Longrightarrow A \lesssim C$
by (*metis bij-betw-def eqpoll-def lepoll-def lepoll-trans order-refl*)

lemma *eqpoll-sym*: $A \approx B \Longrightarrow B \approx A$
unfolding *eqpoll-def*
using *bij-betw-the-inv-into* **by** *auto*

lemma *eqpoll-trans* [*trans*]: $[\![A \approx B;\ B \approx C]\!] \Longrightarrow A \approx C$
unfolding *eqpoll-def* **using** *bij-betw-trans* **by** *blast*

lemma *eqpoll-imp-lepoll*: $A \approx B \Longrightarrow A \lesssim B$
unfolding *eqpoll-def lepoll-def* **by** (*metis bij-betw-def order-refl*)

lemma *subset-imp-lepoll*: $A \subseteq B \Longrightarrow A \lesssim B$
by (*force simp*: *lepoll-def*)

lemma *lepoll-refl* [*iff*]: $A \lesssim A$
by (*simp add*: *subset-imp-lepoll*)

lemma *lepoll-iff*: $A \lesssim B \longleftrightarrow (\exists\, g.\ A \subseteq g\ `\ B)$
unfolding *lepoll-def*
proof *safe*
fix *g* **assume** $A \subseteq g\ `\ B$
then show $\exists\, f.$ *inj-on f A* $\wedge$ $f\ `\ A \subseteq B$

by (*rule-tac x=inv-into B g* **in** *exI*) (*auto simp*: *inv-into-into inj-on-inv-into*)
qed (*metis image-mono the-inv-into-onto*)

lemma *empty-lepoll* [*iff*]: $\{\} \lesssim A$
by (*simp add*: *lepoll-iff*)

lemma *subset-image-lepoll*: $B \subseteq f \mathbin{`} A \Longrightarrow B \lesssim A$
by (*auto simp*: *lepoll-iff*)

lemma *image-lepoll*: $f \mathbin{`} A \lesssim A$
by (*auto simp*: *lepoll-iff*)

lemma *infinite-le-lepoll*: *infinite* $A \longleftrightarrow (UNIV{::}nat\ set) \lesssim A$
by (*simp add*: *infinite-iff-countable-subset lepoll-def*)

lemma *lepoll-Pow-self*: $A \lesssim Pow\ A$
unfolding *lepoll-def inj-def*
proof (*intro exI conjI*)
show *inj-on* ($\lambda x.\ \{x\}$) A
by (*auto simp*: *inj-on-def*)
qed *auto*

lemma *eqpoll-iff-bijections*:
$A \approx B \longleftrightarrow (\exists f\ g.\ (\forall\, x \in A.\ f\ x \in B \wedge g(f\ x) = x) \wedge (\forall\, y \in B.\ g\ y \in A \wedge f(g\ y) = y))$
by (*auto simp*: *eqpoll-def bij-betw-iff-bijections*)

lemma *lepoll-restricted-funspace*:
$\{f.\ f \mathbin{`} A \subseteq B \wedge \{x.\ f\ x \neq k\ x\} \subseteq A \wedge finite\ \{x.\ f\ x \neq k\ x\}\} \lesssim Fpow\ (A \times B)$
proof −
have ∗: $\exists\, U \in Fpow\ (A \times B).\ f = (\lambda x.\ if\ \exists\, y.\ (x,\ y) \in U\ then\ SOME\ y.\ (x,y) \in U\ else\ k\ x)$
if $f \mathbin{`} A \subseteq B\ \{x.\ f\ x \neq k\ x\} \subseteq A\ finite\ \{x.\ f\ x \neq k\ x\}$ **for** f
apply (*rule-tac x*=($\lambda x.\ (x,\ f\ x)$) ‘ $\{x.\ f\ x \neq k\ x\}$ **in** *bexI*)
using *that* **by** (*auto simp*: *image-def Fpow-def*)
show *?thesis*
apply (*rule subset-image-lepoll* [**where** $f = \lambda U\ x.\ if\ \exists\, y.\ (x,y) \in U\ then\ @y.\ (x,y) \in U\ else\ k\ x$])
using ∗ **by** (*auto simp*: *image-def*)
qed

lemma *singleton-lepoll*: $\{x\} \lesssim insert\ y\ A$
by (*force simp*: *lepoll-def*)

lemma *singleton-eqpoll*: $\{x\} \approx \{y\}$
by (*blast intro*: *lepoll-antisym singleton-lepoll*)

lemma *subset-singleton-iff-lepoll*: $(\exists\, x.\ S \subseteq \{x\}) \longleftrightarrow S \lesssim \{()\}$
using *lepoll-iff* **by** *fastforce*

lemma *infinite-insert-lepoll*:
assumes *infinite A* **shows** *insert a A* $\lesssim$ *A*
proof −
obtain *f* :: *nat* ⇒ *'a* **where** *inj f* **and** *f*: *range f* ⊆ *A*
using *assms infinite-countable-subset* **by** *blast*
let *?g* = (λ*z*. *if z=a then f 0 else if z* ∈ *range f then f* (*Suc* (*inv f z*)) *else z*)
show *?thesis*
unfolding *lepoll-def*
proof (*intro exI conjI*)
show *inj-on ?g* (*insert a A*)
using *inj-on-eq-iff* [*OF* ‹*inj f*›]
by (*auto simp*: *inj-on-def*)
show *?g* ‘ *insert a A* ⊆ *A*
using *f* **by** *auto*
qed
qed

lemma *infinite-insert-eqpoll*: *infinite A* ⟹ *insert a A* ≈ *A*
by (*simp add*: *lepoll-antisym infinite-insert-lepoll subset-imp-lepoll subset-insertI*)

lemma *finite-lepoll-infinite*:
assumes *infinite A finite B* **shows** *B* $\lesssim$ *A*
proof −
have *B* $\lesssim$ (*UNIV*::*nat set*)
unfolding *lepoll-def*
using *finite-imp-inj-to-nat-seg* [*OF* ‹*finite B*›] **by** *blast*
then show *?thesis*
using ‹*infinite A*› *infinite-le-lepoll lepoll-trans* **by** *auto*
qed

lemma *countable-lepoll*: ⟦*countable A*; *B* $\lesssim$ *A*⟧ ⟹ *countable B*
by (*meson countable-image countable-subset lepoll-iff*)

lemma *countable-eqpoll*: ⟦*countable A*; *B* ≈ *A*⟧ ⟹ *countable B*
using *countable-lepoll eqpoll-imp-lepoll* **by** *blast*

34.2 The strict relation

lemma *lesspoll-not-refl* [*iff*]: ~ (*i* ≺ *i*)
by (*simp add*: *lepoll-antisym lesspoll-def*)

lemma *lesspoll-imp-lepoll*: *A* ≺ *B* ==> *A* $\lesssim$ *B*
by (*unfold lesspoll-def*, *blast*)

lemma *lepoll-iff-leqpoll*: *A* $\lesssim$ *B* ⟷ *A* ≺ *B* | *A* ≈ *B*
using *eqpoll-imp-lepoll lesspoll-def* **by** *blast*

lemma *lesspoll-trans* [*trans*]: ⟦*X* ≺ *Y*; *Y* ≺ *Z*⟧ ⟹ *X* ≺ *Z*

by (*meson eqpoll-sym lepoll-antisym lepoll-trans lepoll-trans1 lesspoll-def*)

lemma *lesspoll-trans1* [*trans*]: $[\![X \lesssim Y;\ Y \prec Z]\!] \Longrightarrow X \prec Z$
by (*meson eqpoll-sym lepoll-antisym lepoll-trans lepoll-trans1 lesspoll-def*)

lemma *lesspoll-trans2* [*trans*]: $[\![X \prec Y;\ Y \lesssim Z]\!] \Longrightarrow X \prec Z$
by (*meson eqpoll-imp-lepoll eqpoll-sym lepoll-antisym lepoll-trans lesspoll-def*)

lemma *eq-lesspoll-trans* [*trans*]: $[\![X \approx Y;\ Y \prec Z]\!] \Longrightarrow X \prec Z$
using *eqpoll-imp-lepoll lesspoll-trans1* **by** *blast*

lemma *lesspoll-eq-trans* [*trans*]: $[\![X \prec Y;\ Y \approx Z]\!] \Longrightarrow X \prec Z$
using *eqpoll-imp-lepoll lesspoll-trans2* **by** *blast*

lemma *lesspoll-Pow-self*: $A \prec Pow\ A$
unfolding *lesspoll-def bij-betw-def eqpoll-def*
by (*meson lepoll-Pow-self Cantors-theorem*)

lemma *finite-lesspoll-infinite*:
assumes *infinite A finite B* **shows** $B \prec A$
by (*meson assms eqpoll-finite-iff finite-lepoll-infinite lesspoll-def*)

lemma *countable-lesspoll*: $[\![countable\ A;\ B \prec A]\!] \Longrightarrow countable\ B$
using *countable-lepoll lesspoll-def* **by** *blast*

lemma *lepoll-iff-card-le*: $[\![finite\ A;\ finite\ B]\!] \Longrightarrow A \lesssim B \longleftrightarrow card\ A \leq card\ B$
by (*simp add*: *inj-on-iff-card-le lepoll-def*)

lemma *lepoll-iff-finite-card*: $A \lesssim \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A \leq n$
by (*metis card-lessThan finite-lessThan finite-surj lepoll-iff lepoll-iff-card-le*)

lemma *eqpoll-iff-finite-card*: $A \approx \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A = n$
by (*metis card-lessThan eqpoll-finite-iff eqpoll-iff-card finite-lessThan*)

lemma *lesspoll-iff-finite-card*: $A \prec \{..<n::nat\} \longleftrightarrow finite\ A \wedge card\ A < n$
by (*metis eqpoll-iff-finite-card lepoll-iff-finite-card lesspoll-def order-less-le*)

34.3 Mapping by an injection

lemma *inj-on-image-eqpoll-self*: $inj\text{-}on\ f\ A \Longrightarrow f\ `\ A \approx A$
by (*meson bij-betw-def eqpoll-def eqpoll-sym*)

lemma *inj-on-image-lepoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f\ `\ A \lesssim B \longleftrightarrow A \lesssim B$
by (*meson assms image-lepoll lepoll-def lepoll-trans order-refl*)

lemma *inj-on-image-lepoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \lesssim f\ `\ B \longleftrightarrow A \lesssim B$
by (*meson assms eq-iff image-lepoll lepoll-def lepoll-trans*)

lemma *inj-on-image-lesspoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f \mathbin{`} A \prec B \longleftrightarrow A \prec B$
by (*meson assms image-lepoll le-less lepoll-def lesspoll-trans1*)

lemma *inj-on-image-lesspoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \prec f \mathbin{`} B \longleftrightarrow A \prec B$
by (*meson assms eqpoll-sym inj-on-image-eqpoll-self lesspoll-eq-trans*)

lemma *inj-on-image-eqpoll-1* [*simp*]:
assumes *inj-on f A* **shows** $f \mathbin{`} A \approx B \longleftrightarrow A \approx B$
by (*metis assms eqpoll-trans inj-on-image-eqpoll-self eqpoll-sym*)

lemma *inj-on-image-eqpoll-2* [*simp*]:
assumes *inj-on f B* **shows** $A \approx f \mathbin{`} B \longleftrightarrow A \approx B$
by (*metis assms inj-on-image-eqpoll-1 eqpoll-sym*)

34.4 Inserting elements into sets

lemma *insert-lepoll-insertD*:
assumes *insert u A* $\lesssim$ *insert v B u* $\notin$ *A v* $\notin$ *B* **shows** $A \lesssim B$
proof −
obtain *f* **where** *inj*: *inj-on f* (*insert u A*) **and** *fim*: $f \mathbin{`} (\mathit{insert}\ u\ A) \subseteq \mathit{insert}\ v\ B$
by (*meson assms lepoll-def*)
show *?thesis*
unfolding *lepoll-def*
proof (*intro exI conjI*)
let $?g = \lambda x{\in}A.\ \mathit{if}\ f\ x = v\ \mathit{then}\ f\ u\ \mathit{else}\ f\ x$
show *inj-on ?g A*
using *inj* ‹$u \notin A$› **by** (*auto simp*: *inj-on-def*)
show $?g \mathbin{`} A \subseteq B$
using *fim* ‹$u \notin A$› *image-subset-iff inj inj-on-image-mem-iff* **by** *fastforce*
qed
qed

lemma *insert-eqpoll-insertD*: ⟦*insert u A* $\approx$ *insert v B*; $u \notin A$; $v \notin B$⟧ $\Longrightarrow$ $A \approx B$
by (*meson insert-lepoll-insertD eqpoll-imp-lepoll eqpoll-sym lepoll-antisym*)

lemma *insert-lepoll-cong*:
assumes $A \lesssim B$ $b \notin B$ **shows** *insert a A* $\lesssim$ *insert b B*
proof −
obtain *f* **where** *f*: *inj-on f A* $f \mathbin{`} A \subseteq B$
by (*meson assms lepoll-def*)
let $?f = \lambda u \in \mathit{insert}\ a\ A.\ \mathit{if}\ u{=}a\ \mathit{then}\ b\ \mathit{else}\ f\ u$
show *?thesis*
unfolding *lepoll-def*
proof (*intro exI conjI*)
show *inj-on ?f* (*insert a A*)
using *f* ‹$b \notin B$› **by** (*auto simp*: *inj-on-def*)

show *?f ‘ insert a A ⊆ insert b B*
using *f ‹b ∉ B›* **by** *auto*
qed
qed

lemma *insert-eqpoll-cong*:
⟦A ≈ B; a ∉ A; b ∉ B⟧ ⟹ insert a A ≈ insert b B
by (*meson eqpoll-imp-lepoll eqpoll-sym insert-lepoll-cong lepoll-antisym*)

lemma *insert-eqpoll-insert-iff*:
⟦a ∉ A; b ∉ B⟧ ⟹ insert a A ≈ insert b B ⟷ A ≈ B
by (*meson insert-eqpoll-insertD insert-eqpoll-cong*)

lemma *insert-lepoll-insert-iff*:
⟦a ∉ A; b ∉ B⟧ ⟹ (insert a A ≲ insert b B) ⟷ (A ≲ B)
by (*meson insert-lepoll-insertD insert-lepoll-cong*)

lemma *less-imp-insert-lepoll*:
assumes *A ≺ B* **shows** *insert a A ≲ B*
proof −
obtain *f* **where** *inj-on f A f ‘ A ⊂ B*
using *assms* **by** (*metis bij-betw-def eqpoll-def lepoll-def lesspoll-def psubset-eq*)
then obtain *b* **where** *b*: *b ∈ B b ∉ f ‘ A*
by *auto*
show *?thesis*
unfolding *lepoll-def*
proof (*intro exI conjI*)
show *inj-on* (*f*(*a*:=*b*)) (*insert a A*)
using *b ‹inj-on f A›* **by** (*auto simp*: *inj-on-def*)
show (*f*(*a*:=*b*)) ‘ *insert a A ⊆ B*
using *‹f ‘ A ⊂ B›* **by** (*auto simp*: *b*)
qed
qed

lemma *finite-insert-lepoll*: *finite A ⟹ (insert a A ≲ A) ⟷ (a ∈ A)*
proof (*induction A rule*: *finite-induct*)
case (*insert x A*)
then show *?case*
by (*metis insertI2 insert-lepoll-insert-iff insert-subsetI lepoll-trans subsetI subset-imp-lepoll*)
qed *auto*

34.5 Binary sums and unions

lemma *Un-lepoll-mono*:
assumes *A ≲ C B ≲ D disjnt C D* **shows** *A ∪ B ≲ C ∪ D*
proof −
obtain *f g* **where** *inj*: *inj-on f A inj-on g B* **and** *fg*: *f ‘ A ⊆ C g ‘ B ⊆ D*
by (*meson assms lepoll-def*)

have *inj-on* ($\lambda x.$ *if* $x \in A$ *then* $f\ x$ *else* $g\ x$) ($A \cup B$)
using *inj* ‹*disjnt C D*› *fg* **unfolding** *disjnt-iff*
by (*fastforce intro*: *inj-onI dest*: *inj-on-contraD split*: *if-split-asm*)
with *fg* **show** *?thesis*
unfolding *lepoll-def*
by (*rule-tac x*=$\lambda x.$ *if* $x \in A$ *then* $f\ x$ *else* $g\ x$ **in** *exI*) *auto*
qed

lemma *Un-eqpoll-cong*: $[\![A \approx C; B \approx D; \mathit{disjnt}\ A\ B; \mathit{disjnt}\ C\ D]\!] \Longrightarrow A \cup B \approx C \cup D$
by (*meson Un-lepoll-mono eqpoll-imp-lepoll eqpoll-sym lepoll-antisym*)

lemma *sum-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ **shows** A <+> $B \lesssim C$ <+> D
proof −
obtain *f g* **where** *inj-on f A* f ‘ $A \subseteq C$ *inj-on g B* g ‘ $B \subseteq D$
by (*meson assms lepoll-def*)
then show *?thesis*
unfolding *lepoll-def*
by (*rule-tac x*=*case-sum* (*Inl* ∘ *f*) (*Inr* ∘ *g*) **in** *exI*) (*force simp*: *inj-on-def*)
qed

lemma *sum-eqpoll-cong*: $[\![A \approx C; B \approx D]\!] \Longrightarrow A$ <+> $B \approx C$ <+> D
by (*meson eqpoll-imp-lepoll eqpoll-sym lepoll-antisym sum-lepoll-mono*)

34.6 Binary Cartesian products

lemma *times-square-lepoll*: $A \lesssim A \times A$
unfolding *lepoll-def inj-def*
proof (*intro exI conjI*)
show *inj-on* ($\lambda x.$ (x,x)) A
by (*auto simp*: *inj-on-def*)
qed *auto*

lemma *times-commute-eqpoll*: $A \times B \approx B \times A$
unfolding *eqpoll-def*
by (*force intro*: *bij-betw-byWitness* [**where** $f = \lambda(x,y).\ (y,x)$ **and** $f' = \lambda(x,y).\ (y,x)$])

lemma *times-assoc-eqpoll*: $(A \times B) \times C \approx A \times (B \times C)$
unfolding *eqpoll-def*
by (*force intro*: *bij-betw-byWitness* [**where** $f = \lambda((x,y),z).\ (x,(y,z))$ **and** $f' = \lambda(x,(y,z)).\ ((x,y),z)$])

lemma *times-singleton-eqpoll*: $\{a\} \times A \approx A$
proof −
have $\{a\} \times A = (\lambda x.\ (a,x))$ ‘ A
by *auto*
also have ... $\approx A$

proof (*rule inj-on-image-eqpoll-self*)
show *inj-on* (*Pair a*) *A*
by (*auto simp*: *inj-on-def*)
qed
finally show *?thesis* **.**
qed

lemma *times-lepoll-mono*:
assumes $A \lesssim C$ $B \lesssim D$ **shows** $A \times B \lesssim C \times D$
proof −
obtain *f g* **where** *inj-on f A f ' A* $\subseteq$ *C inj-on g B g ' B* $\subseteq$ *D*
by (*meson assms lepoll-def*)
then show *?thesis*
unfolding *lepoll-def*
by (*rule-tac x*=$\lambda(x,y)$. (*f x*, *g y*) **in** *exI*) (*auto simp*: *inj-on-def*)
qed

lemma *times-eqpoll-cong*: $[\![A \approx C; B \approx D]\!] \Longrightarrow A \times B \approx C \times D$
by (*metis eqpoll-imp-lepoll eqpoll-sym lepoll-antisym times-lepoll-mono*)

lemma
assumes $B \neq \{\}$ **shows** *lepoll-times1*: $A \lesssim A \times B$ **and** *lepoll-times2*: $A \lesssim B \times A$
using *assms lepoll-iff* **by** *fastforce+*

lemma *times-0-eqpoll*: $\{\} \times A \approx \{\}$
by (*simp add*: *eqpoll-iff-bijections*)

lemma *Sigma-inj-lepoll-mono*:
assumes *h*: *inj-on h A h ' A* $\subseteq$ *C* **and** $\bigwedge x.\ x \in A \Longrightarrow B\ x \lesssim D\ (h\ x)$
shows *Sigma A B* $\lesssim$ *Sigma C D*
proof −
have $\bigwedge x.\ x \in A \Longrightarrow \exists f.$ *inj-on f* (*B x*) $\wedge$ *f ' B x* $\subseteq$ *D* (*h x*)
by (*meson assms lepoll-def*)
then obtain *f* **where** $\bigwedge x.\ x \in A \Longrightarrow$ *inj-on* (*f x*) (*B x*) $\wedge$ *f x ' B x* $\subseteq$ *D* (*h x*)
by *metis*
with *h* **show** *?thesis*
unfolding *lepoll-def inj-on-def*
by (*rule-tac x*=$\lambda(x,y)$. (*h x*, *f x y*) **in** *exI*) *force*
qed

lemma *Sigma-lepoll-mono*:
assumes $A \subseteq C$ $\bigwedge x.\ x \in A \Longrightarrow B\ x \lesssim D\ x$ **shows** *Sigma A B* $\lesssim$ *Sigma C D*
using *Sigma-inj-lepoll-mono* [*of id*] *assms* **by** *auto*

lemma *sum-times-distrib-eqpoll*: $(A <+> B) \times C \approx (A \times C) <+> (B \times C)$
unfolding *eqpoll-def*
proof
show *bij-betw* ($\lambda(x,z)$. *case-sum*($\lambda y.$ *Inl*(y,z)) ($\lambda y.$ *Inr*(y,z)) *x*) (($A <+> B$) $\times$

C) (*A* × *C* <+> *B* × *C*)
by (*rule bij-betw-byWitness* [**where** *f′* = *case-sum* (λ(*x*,*z*). (*Inl x*, *z*)) (λ(*y*,*z*). (*Inr y*, *z*))]) *auto*
qed

lemma *Sigma-eqpoll-cong*:
assumes *h*: *bij-betw h A C* **and** *BD*: ⋀*x*. *x* ∈ *A* ⟹ *B x* ≈ *D* (*h x*)
shows *Sigma A B* ≈ *Sigma C D*
proof (*intro lepoll-antisym*)
show *Sigma A B* ≲ *Sigma C D*
by (*metis Sigma-inj-lepoll-mono bij-betw-def eqpoll-imp-lepoll subset-refl assms*)
have *inj-on* (*inv-into A h*) *C* ∧ *inv-into A h* ‘ *C* ⊆ *A*
by (*metis bij-betw-def bij-betw-inv-into h set-eq-subset*)
then show *Sigma C D* ≲ *Sigma A B*
by (*smt* (*verit*, *best*) *BD Sigma-inj-lepoll-mono bij-betw-inv-into-right eqpoll-sym h image-subset-iff lepoll-refl lepoll-trans2*)
qed

lemma *prod-insert-eqpoll*:
assumes *a* ∉ *A* **shows** *insert a A* × *B* ≈ *B* <+> *A* × *B*
unfolding *eqpoll-def*
proof
show *bij-betw* (λ(*x*,*y*). *if x*=*a then Inl y else Inr* (*x*,*y*)) (*insert a A* × *B*) (*B* <+> *A* × *B*)
by (*rule bij-betw-byWitness* [**where** *f′* = *case-sum* (λ*y*. (*a*,*y*)) *id*]) (*auto simp*: *assms*)
qed

34.7 General Unions

lemma *Union-eqpoll-Times*:
assumes *B*: ⋀*x*. *x* ∈ *A* ⟹ *F x* ≈ *B* **and** *disj*: *pairwise* (λ*x y*. *disjnt* (*F x*) (*F y*)) *A*
shows (⋃*x*∈*A*. *F x*) ≈ *A* × *B*
proof (*rule lepoll-antisym*)
obtain *b* **where** *b*: ⋀*x*. *x* ∈ *A* ⟹ *bij-betw* (*b x*) (*F x*) *B*
using *B* **unfolding** *eqpoll-def* **by** *metis*
show ⋃(*F* ‘ *A*) ≲ *A* × *B*
unfolding *lepoll-def*
proof (*intro exI conjI*)
define χ **where** χ ≡ λ*z*. *THE x*. *x* ∈ *A* ∧ *z* ∈ *F x*
have χ: χ *z* = *x* **if** *x* ∈ *A z* ∈ *F x* **for** *x z*
unfolding χ-*def*
by (*smt* (*verit*, *best*) *disj disjnt-iff pairwiseD that*(*1*,*2*) *theI-unique*)
let *?f* = λ*z*. (χ *z*, *b* (χ *z*) *z*)
show *inj-on ?f* (⋃(*F* ‘ *A*))
unfolding *inj-on-def*
by *clarify* (*metis* χ *b bij-betw-inv-into-left*)
show *?f* ‘ ⋃(*F* ‘ *A*) ⊆ *A* × *B*

```
    using χ b bij-betwE by blast
  qed
  show A × B ≲ ⋃(F ‘ A)
    unfolding lepoll-def
  proof (intro exI conjI)
    let ?f = λ(x,y). inv-into (F x) (b x) y
    have ∗: inv-into (F x) (b x) y ∈ F x if x ∈ A y ∈ B for x y
      by (metis b bij-betw-imp-surj-on inv-into-into that)
    then show inj-on ?f (A × B)
      unfolding inj-on-def
      by clarsimp (metis (mono-tags, lifting) b bij-betw-inv-into-right disj disjnt-iff
pairwiseD)
    show ?f ‘ (A × B) ⊆ ⋃ (F ‘ A)
      by clarsimp (metis b bij-betw-imp-surj-on inv-into-into)
  qed
qed

lemma UN-lepoll-UN:
  assumes A: ⋀x. x ∈ A ⟹ B x ≲ C x
    and disj: pairwise (λx y. disjnt (C x) (C y)) A
  shows ⋃ (B‘A) ≲ ⋃ (C‘A)
proof −
  obtain f where f: ⋀x. x ∈ A ⟹ inj-on (f x) (B x) ∧ f x ‘ (B x) ⊆ (C x)
    using A unfolding lepoll-def by metis
  show ?thesis
    unfolding lepoll-def
  proof (intro exI conjI)
    define χ where χ ≡ λz. @x. x ∈ A ∧ z ∈ B x
    have χ: χ z ∈ A ∧ z ∈ B (χ z) if x ∈ A z ∈ B x for x z
      unfolding χ-def by (metis (mono-tags, lifting) someI-ex that)
    let ?f = λz. (f (χ z) z)
    show inj-on ?f (⋃(B ‘ A))
      using disj f unfolding inj-on-def disjnt-iff pairwise-def image-subset-iff
      by (metis UN-iff χ)
    show ?f ‘ ⋃ (B ‘ A) ⊆ ⋃ (C ‘ A)
      using χ f unfolding image-subset-iff by blast
  qed
qed

lemma UN-eqpoll-UN:
  assumes A: ⋀x. x ∈ A ⟹ B x ≈ C x
    and B: pairwise (λx y. disjnt (B x) (B y)) A
    and C: pairwise (λx y. disjnt (C x) (C y)) A
  shows (⋃x∈A. B x) ≈ (⋃x∈A. C x)
proof (rule lepoll-antisym)
  show ⋃ (B ‘ A) ≲ ⋃ (C ‘ A)
    by (meson A C UN-lepoll-UN eqpoll-imp-lepoll)
  show ⋃ (C ‘ A) ≲ ⋃ (B ‘ A)
    by (simp add: A B UN-lepoll-UN eqpoll-imp-lepoll eqpoll-sym)
```

qed

34.8 General Cartesian products (Pi)

lemma *PiE-sing-eqpoll-self*: $(\{a\} \rightarrow_E B) \approx B$
proof −
have *1*: $x = y$
if $x \in \{a\} \rightarrow_E B\ y \in \{a\} \rightarrow_E B\ x\ a = y\ a$ **for** $x\ y$
by (*metis IntD2 PiE-def extensionalityI singletonD that*)
have *2*: $x \in (\lambda h.\ h\ a)\ `\ (\{a\} \rightarrow_E B)$ **if** $x \in B$ **for** x
using *that* **by** (*rule-tac* $x{=}\lambda z{\in}\{a\}.\ x$ **in** *image-eqI*) *auto*
show *?thesis*
unfolding *eqpoll-def bij-betw-def inj-on-def*
by (*force intro*: *1 2*)
qed

lemma *lepoll-funcset-right*:
assumes $B \lesssim B'$ **shows** $A \rightarrow_E B \lesssim A \rightarrow_E B'$
proof −
obtain f **where** f: *inj-on f B* $f\ `\ B \subseteq B'$
by (*meson assms lepoll-def*)
let $?G = \lambda g.\ \lambda z \in A.\ f(g\ z)$
have *inj-on ?G* $(A \rightarrow_E B)$
using f **by** (*smt* (*verit*, *best*) *PiE-ext PiE-mem inj-on-def restrict-apply'*)
moreover have $?G\ `\ (A \rightarrow_E B) \subseteq (A \rightarrow_E B')$
using f **by** *fastforce*
ultimately show *?thesis*
by (*meson lepoll-def*)
qed

lemma *lepoll-funcset-left*:
assumes $B \neq \{\}$ $A \lesssim A'$
shows $A \rightarrow_E B \lesssim A' \rightarrow_E B$
proof −
obtain b **where** $b \in B$
using *assms* **by** *blast*
obtain f **where** *inj-on f A* **and** *fim*: $f\ `\ A \subseteq A'$
using *assms* **by** (*auto simp*: *lepoll-def*)
then obtain h **where** h: $\bigwedge x.\ x \in A \Longrightarrow h\ (f\ x) = x$
using *the-inv-into-f-f* **by** *fastforce*
let $?F = \lambda g.\ \lambda u \in A'.\ \textit{if}\ h\ u \in A\ \textit{then}\ g(h\ u)\ \textit{else}\ b$
show *?thesis*
unfolding *lepoll-def inj-on-def*
proof (*intro exI conjI ballI impI ext*)
fix $k\ l\ x$
assume k: $k \in A \rightarrow_E B$ **and** l: $l \in A \rightarrow_E B$ **and** $?F\ k = ?F\ l$
then have $?F\ k\ (f\ x) = ?F\ l\ (f\ x)$
by *simp*
then show $k\ x = l\ x$

by (*smt* (*verit*, *best*) *PiE-arb fim h image-subset-iff k l restrict-apply'*)
next
show *?F* ‘ $(A \rightarrow_E B) \subseteq A' \rightarrow_E B$
using ‹$b \in B$› **by** *force*
qed
qed

lemma *lepoll-funcset*:
$[\![B \neq \{\}; A \lesssim A'; B \lesssim B']\!] \Longrightarrow A \rightarrow_E B \lesssim A' \rightarrow_E B'$
by (*rule lepoll-trans* [*OF lepoll-funcset-right lepoll-funcset-left*]) *auto*

lemma *lepoll-PiE*:
assumes $\bigwedge i.\ i \in A \Longrightarrow B\ i \lesssim C\ i$
shows *PiE A B* $\lesssim$ *PiE A C*
proof −
obtain *f* **where** *f*: $\bigwedge i.\ i \in A \Longrightarrow$ *inj-on* (*f i*) (*B i*) $\wedge$ (*f i*) ‘ *B i* $\subseteq$ *C i*
using *assms* **unfolding** *lepoll-def* **by** *metis*
let *?G* = $\lambda g.\ \lambda i \in A.\ f\ i\ (g\ i)$
have *inj-on ?G* (*PiE A B*)
by (*smt* (*verit*, *ccfv-SIG*) *PiE-ext PiE-iff f inj-on-def restrict-apply'*)
moreover have *?G* ‘ (*PiE A B*) $\subseteq$ (*PiE A C*)
using *f* **by** *fastforce*
ultimately show *?thesis*
by (*meson lepoll-def*)
qed

lemma *card-le-PiE-subindex*:
assumes $A \subseteq A'$ $Pi_E\ A'\ B \neq \{\}$
shows *PiE A B* $\lesssim$ *PiE A' B*
proof −
have $\bigwedge x.\ x \in A' \Longrightarrow \exists y.\ y \in B\ x$
using *assms* **by** *blast*
then obtain *g* **where** *g*: $\bigwedge x.\ x \in A' \Longrightarrow g\ x \in B\ x$
by *metis*
let *?F* = $\lambda f\ x.$ *if* $x \in A$ *then f x else if* $x \in A'$ *then g x else undefined*
have $Pi_E\ A\ B \subseteq (\lambda f.$ *restrict f A*) ‘ $Pi_E\ A'\ B$
proof
show $f \in Pi_E\ A\ B \Longrightarrow f \in (\lambda f.$ *restrict f A*) ‘ $Pi_E\ A'\ B$ **for** *f*
using ‹$A \subseteq A'$›
by (*rule-tac x*=*?F f* **in** *image-eqI*) (*auto simp*: *g fun-eq-iff*)
qed
then have $Pi_E\ A\ B \lesssim (\lambda f.\ \lambda i \in A.\ f\ i)$ ‘ $Pi_E\ A'\ B$
by (*simp add*: *subset-imp-lepoll*)
also have ... $\lesssim$ *PiE A' B*
by (*rule image-lepoll*)
finally show *?thesis* **.**
qed

lemma *finite-restricted-funspace*:
assumes *finite A finite B*
shows *finite* $\{f.\ f\ `\ A \subseteq B \wedge \{x.\ f\,x \neq k\,x\} \subseteq A\}$ (**is** *finite ?F*)
proof (*rule finite-subset*)
show *finite* $((\lambda U\ x.\ \mathit{if}\ \exists\, y.\ (x,y) \in U\ \mathit{then}\ @y.\ (x,y) \in U\ \mathit{else}\ k\ x)\ `\ \mathit{Pow}(A \times B))$ (**is** *finite ?G*)
using *assms* **by** *auto*
show *?F* $\subseteq$ *?G*
proof
fix *f*
assume $f \in$ *?F*
then show $f \in$ *?G*
by (*rule-tac* $x{=}(\lambda x.\ (x,f\,x))$ $`\{x.\ f\,x \neq k\,x\}$ **in** *image-eqI*) (*auto simp*: *fun-eq-iff image-def*)
qed
qed

proposition *finite-PiE-iff*:
$\mathit{finite}(\mathit{PiE}\ I\ S) \longleftrightarrow \mathit{PiE}\ I\ S = \{\} \vee \mathit{finite}\ \{i \in I.\ {}^\sim(\exists\, a.\ S\ i \subseteq \{a\})\} \wedge (\forall\, i \in I.\ \mathit{finite}(S\ i))$
(**is** *?lhs* = *?rhs*)
proof (*cases* $\mathit{PiE}\ I\ S = \{\}$)
case *False*
define *J* **where** $J \equiv \{i \in I.\ \nexists\, a.\ S\ i \subseteq \{a\}\}$
show *?thesis*
proof
assume *L*: *?lhs*
have *infinite* $(\mathit{Pi}_E\ I\ S)$ **if** *infinite J*
proof −
have $(\mathit{UNIV}{::}\mathit{nat}\ \mathit{set}) \lesssim (\mathit{UNIV}{::}(\mathit{nat}{\Rightarrow}\mathit{bool})\ \mathit{set})$
proof −
have $\forall\, N{::}\mathit{nat}\ \mathit{set}.\ \mathit{inj\text{-}on}\ (=)\ N$
by (*simp add*: *inj-on-def*)
then show *?thesis*
by (*meson infinite-iff-countable-subset infinite-le-lepoll top.extremum*)
qed
also have $\ldots = (\mathit{UNIV}{::}\mathit{nat}\ \mathit{set}) \to_E (\mathit{UNIV}{::}\mathit{bool}\ \mathit{set})$
by *auto*
also have $\ldots \lesssim J \to_E (\mathit{UNIV}{::}\mathit{bool}\ \mathit{set})$
by (*metis empty-not-UNIV infinite-le-lepoll lepoll-funcset-left that*)
also have $\ldots \lesssim \mathit{Pi}_E\ J\ S$
proof −
have ∗: $(\mathit{UNIV}{::}\mathit{bool}\ \mathit{set}) \lesssim S\ i$ **if** $i \in I$ **and** §: $\forall\, a.\ \neg\ S\ i \subseteq \{a\}$ **for** *i*
proof −
obtain *a b* **where** $\{a,b\} \subseteq S\ i\ a \neq b$
by (*metis* § *empty-subsetI insert-subset subsetI*)
then show *?thesis*

by (*metis Set.set-insert UNIV-bool insert-iff insert-lepoll-cong insert-subset singleton-lepoll*)
qed
show *?thesis*
by (*auto simp*: ∗ *J-def intro*: *lepoll-PiE*)
qed
also have ... $\lesssim Pi_E\ I\ S$
using *False* **by** (*auto simp*: *J-def intro*: *card-le-PiE-subindex*)
finally have (*UNIV*::*nat set*) $\lesssim Pi_E\ I\ S$ **.**
then show *?thesis*
by (*simp add*: *infinite-le-lepoll*)
qed
moreover have *finite* (*S i*) **if** $i \in I$ **for** *i*
proof (*rule finite-subset*)
obtain *f* **where** *f*: $f \in PiE\ I\ S$
using *False* **by** *blast*
show $S\ i \subseteq (\lambda f.\ f\ i)\ `\ Pi_E\ I\ S$
proof
show $s \in (\lambda f.\ f\ i)\ `\ Pi_E\ I\ S$ **if** $s \in S\ i$ **for** *s*
using *that f* ‹$i \in I$›
by (*rule-tac x*=$\lambda j.$ *if j* = *i then s else f j* **in** *image-eqI*) *auto*
qed
next
show *finite* ((λ*x*. *x i*) ‘ $Pi_E\ I\ S$)
using *L* **by** *blast*
qed
ultimately show *?rhs*
using *L*
by (*auto simp*: *J-def False*)
next
assume *R*: *?rhs*
have $\forall\, i \in I - J.\ \exists\, a.\ S\ i = \{a\}$
using *False J-def* **by** *blast*
then obtain *a* **where** *a*: $\forall\, i \in I - J.\ S\ i = \{a\ i\}$
by *metis*
let *?F* = {*f*. *f* ‘ *J* ⊆ (⋃ *i* ∈ *J*. *S i*) ∧ {*i*. *f i* ≠ (*if i* ∈ *I then a i else undefined*)} ⊆ *J*}
have ∗: *finite* ($Pi_E\ I\ S$)
if *finite J* **and** ∀ *i*∈*I*. *finite* (*S i*)
proof (*rule finite-subset*)
have ⋀*f j*. ⟦*f* ∈ $Pi_E\ I\ S$; *j* ∈ *J*⟧ ⟹ *f j* ∈ ⋃ (*S* ‘ *J*)
using *J-def* **by** *blast*
moreover
have ⋀*f j*. ⟦*f* ∈ $Pi_E\ I\ S$; *f j* ≠ (*if j* ∈ *I then a j else undefined*)⟧ ⟹ *j* ∈ *J*
by (*metis DiffI PiE-E a singletonD*)
ultimately show $Pi_E\ I\ S \subseteq$ *?F* **by** *force*
show *finite ?F*
proof (*rule finite-restricted-funspace* [*OF* ‹*finite J*›])
show *finite* (⋃ (*S* ‘ *J*))

```
      using that J-def by blast
    qed
  qed
  show ?lhs
    using R by (auto simp: ∗ J-def)
  qed
qed auto


corollary finite-funcset-iff:
  finite(I →_E S) ⟷ (∃ a. S ⊆ {a}) ∨ I = {} ∨ finite I ∧ finite S
  by (fastforce simp: finite-PiE-iff PiE-eq-empty-iff dest: subset-singletonD)
```

34.9 Misc other resultd

```
lemma lists-lepoll-mono:
  assumes A ≲ B shows lists A ≲ lists B
proof −
  obtain f where f: inj-on f A f ' A ⊆ B
    by (meson assms lepoll-def)
  moreover have inj-on (map f) (lists A)
    using f unfolding inj-on-def
    by clarsimp (metis list.inj-map-strong)
  ultimately show ?thesis
    unfolding lepoll-def by force
qed

lemma lepoll-lists: A ≲ lists A
  unfolding lepoll-def inj-on-def by(rule-tac x=λx. [x] in exI) auto
```

Dedekind's definition of infinite set

```
lemma infinite-iff-psubset: infinite A ⟷ (∃ B. B ⊂ A ∧ A≈B)
proof
  assume infinite A
  then obtain f :: nat ⇒ 'a where inj f and f: range f ⊆ A
    by (meson infinite-countable-subset)
  define C where C ≡ A − range f
  have C: A = range f ∪ C range f ∩ C = {}
    using f by (auto simp: C-def)
  have ∗: range (f ∘ Suc) ⊂ range f
    using inj-eq [OF ‹inj f›] by (fastforce simp: set-eq-iff)
  have range f ∪ C ≈ range (f ∘ Suc) ∪ C
  proof (intro Un-eqpoll-cong)
    show range f ≈ range (f ∘ Suc)
      by (meson ‹inj f› eqpoll-refl inj-Suc inj-compose inj-on-image-eqpoll-2)
    show disjnt (range f) C
      by (simp add: C disjnt-def)
    then show disjnt (range (f ∘ Suc)) C
      using ∗ disjnt-subset1 by blast
  qed auto
```

moreover have *range* $(f \circ Suc) \cup C \subset A$
using $*$ *f C-def* **by** *blast*
ultimately show $\exists B{\subset}A.\ A \approx B$
by (*metis C*(*1*))
next
assume $\exists B{\subset}A.\ A \approx B$ **then show** *infinite A*
by (*metis card-subset-eq eqpoll-finite-iff eqpoll-iff-card psubsetE*)
qed

lemma *infinite-iff-psubset-le*: *infinite* $A \longleftrightarrow (\exists B.\ B \subset A \wedge A \lesssim B)$
by (*meson eqpoll-imp-lepoll infinite-iff-psubset lepoll-antisym psubsetE subset-imp-lepoll*)

end

theory *Simps-Case-Conv*
imports *Case-Converter*
keywords *simps-of-case case-of-simps* :: *thy-decl*
abbrevs *simps-of-case case-of-simps* =
begin

ML-file ‹*simps-case-conv.ML*›

end

theory *Extended*
imports *Simps-Case-Conv*
begin

datatype *'a extended* = *Fin 'a* | *Pinf* (‹∞›) | *Minf* (‹$-\infty$›)

instantiation *extended* :: (*order*)*order*
begin

fun *less-eq-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
Fin x ≤ *Fin y* = (*x* ≤ *y*) |
- ≤ *Pinf* = *True* |
Minf ≤ - = *True* |
(-::*'a extended*) ≤ - = *False*

case-of-simps *less-eq-extended-case*: *less-eq-extended.simps*

definition *less-extended* :: *'a extended* ⇒ *'a extended* ⇒ *bool* **where**
((*x*::*'a extended*) < *y*) = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance
by *intro-classes* (*auto simp*: *less-extended-def less-eq-extended-case split*: *extended.splits*)

end

instance *extended* :: (*linorder*)*linorder*
by *intro-classes* (*auto simp*: *less-eq-extended-case split*:*extended.splits*)

lemma *Minf-le*[*simp*]: $Minf \leq y$
by(*cases y*) *auto*
lemma *le-Pinf*[*simp*]: $x \leq Pinf$
by(*cases x*) *auto*
lemma *le-Minf*[*simp*]: $x \leq Minf \longleftrightarrow x = Minf$
by(*cases x*) *auto*
lemma *Pinf-le*[*simp*]: $Pinf \leq x \longleftrightarrow x = Pinf$
by(*cases x*) *auto*

lemma *less-extended-simps*[*simp*]:
$Fin\ x < Fin\ y = (x < y)$
$Fin\ x < Pinf\ = True$
$Fin\ x < Minf\ = False$
$Pinf < h\ \ = False$
$Minf < Fin\ x\ = True$
$Minf < Pinf\ = True$
$l\ < Minf\ = False$
by (*auto simp add*: *less-extended-def*)

lemma *min-extended-simps*[*simp*]:
$min\ (Fin\ x)\ (Fin\ y) = Fin(min\ x\ y)$
$min\ xx\ \ Pinf\ = xx$
$min\ xx\ \ Minf\ = Minf$
$min\ Pinf\ \ yy\ \ = yy$
$min\ Minf\ \ yy\ \ = Minf$
by (*auto simp add*: *min-def*)

lemma *max-extended-simps*[*simp*]:
$max\ (Fin\ x)\ (Fin\ y) = Fin(max\ x\ y)$
$max\ xx\ \ Pinf\ = Pinf$
$max\ xx\ \ Minf\ = xx$
$max\ Pinf\ \ yy\ \ = Pinf$
$max\ Minf\ \ yy\ \ = yy$
by (*auto simp add*: *max-def*)

instantiation *extended* :: (*zero*)*zero*
begin
definition $0 = Fin(0::'a)$
instance ..
end

declare *zero-extended-def*[*symmetric*, *code-post*]

lemma *Minf-eq-zero*[*simp*]: $-\infty = 0 \longleftrightarrow False$ **and** *Pinf-eq-zero*[*simp*]: $\infty = 0 \longleftrightarrow False$
unfolding *zero-extended-def* **by** *auto*

instantiation *extended* :: (*one*)*one*
begin
definition $1 = Fin(1::'a)$
instance ..
end

declare *one-extended-def*[*symmetric*, *code-post*]

lemma *Minf-eq-one*[*simp*]: $-\infty = 1 \longleftrightarrow False$ **and** *Pinf-eq-one*[*simp*]: $\infty = 1 \longleftrightarrow False$
unfolding *one-extended-def* **by** *auto*

instantiation *extended* :: (*plus*)*plus*
begin

The following definition of of addition is totalized to make it asociative and commutative. Normally the sum of plus and minus infinity is undefined.

fun *plus-extended* **where**
$Fin\ x + Fin\ y = Fin(x+y)$ |
$Fin\ x + Pinf = Pinf$ |
$Pinf + Fin\ x = Pinf$ |
$Pinf + Pinf = Pinf$ |
$Minf + Fin\ y = Minf$ |
$Fin\ x + Minf = Minf$ |
$Minf + Minf = Minf$ |
$Minf + Pinf = Pinf$ |
$Pinf + Minf = Pinf$

case-of-simps *plus-case*: *plus-extended.simps*

instance ..

end

instance *extended* :: (*ab-semigroup-add*)*ab-semigroup-add*
by *intro-classes* (*simp-all add*: *ac-simps plus-case split*: *extended.splits*)

instance *extended* :: (*ordered-ab-semigroup-add*)*ordered-ab-semigroup-add*
by *intro-classes* (*auto simp*: *add-left-mono plus-case split*: *extended.splits*)

instance *extended* :: (*comm-monoid-add*)*comm-monoid-add*
proof
fix $x :: 'a\ extended$ **show** $0 + x = x$ **unfolding** *zero-extended-def* **by**(*cases*

x)*auto*
qed

instantiation *extended* :: (*uminus*)*uminus*
begin

fun *uminus-extended* **where**
− (*Fin x*) = *Fin* (− *x*) |
− *Pinf* = *Minf* |
− *Minf* = *Pinf*

instance ..

end

instantiation *extended* :: (*ab-group-add*)*minus*
begin
definition *x* − *y* = *x* + −(*y*::'*a extended*)
instance ..
end

lemma *minus-extended-simps*[*simp*]:
Fin x − *Fin y* = *Fin*(*x* − *y*)
Fin x − *Pinf* = *Minf*
Fin x − *Minf* = *Pinf*
Pinf − *Fin y* = *Pinf*
Pinf − *Minf* = *Pinf*
Minf − *Fin y* = *Minf*
Minf − *Pinf* = *Minf*
Minf − *Minf* = *Pinf*
Pinf − *Pinf* = *Pinf*
by (*simp-all add*: *minus-extended-def*)

Numerals:

instance *extended* :: ({*ab-semigroup-add*,*one*})*numeral* **..**

lemma *Fin-numeral*[*code-post*]: *Fin*(*numeral w*) = *numeral w*
apply (*induct w rule*: *num-induct*)
apply (*simp only*: *numeral-One one-extended-def*)
apply (*simp only*: *numeral-inc one-extended-def plus-extended.simps*(*1*)[*symmetric*])
done

lemma *Fin-neg-numeral*[*code-post*]: *Fin* (− *numeral w*) = − *numeral w*
by (*simp only*: *Fin-numeral uminus-extended.simps*[*symmetric*])

instantiation *extended* :: (*lattice*)*bounded-lattice*
begin

definition *bot* = *Minf*
definition *top* = *Pinf*

fun *inf-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
inf-extended (*Fin i*) (*Fin j*) = *Fin* (*inf i j*) |
inf-extended a Minf = *Minf* |
inf-extended Minf a = *Minf* |
inf-extended Pinf a = *a* |
inf-extended a Pinf = *a*

fun *sup-extended* :: *'a extended* ⇒ *'a extended* ⇒ *'a extended* **where**
sup-extended (*Fin i*) (*Fin j*) = *Fin* (*sup i j*) |
sup-extended a Pinf = *Pinf* |
sup-extended Pinf a = *Pinf* |
sup-extended Minf a = *a* |
sup-extended a Minf = *a*

case-of-simps *inf-extended-case*: *inf-extended.simps*
case-of-simps *sup-extended-case*: *sup-extended.simps*

instance
by (*intro-classes*) (*auto simp*: *inf-extended-case sup-extended-case less-eq-extended-case*
bot-extended-def top-extended-def split: *extended.splits*)
end

instantiation *extended* :: (*conditionally-complete-lattice*) *complete-lattice*
begin

definition
Inf A = (
if A = {} ∨ *A* = {∞} *then* ∞
else if −∞ ∈ *A* ∨ ¬ *bdd-below* (*Fin* −‘ *A*) *then* −∞
else Fin (*Inf* (*Fin* −‘ *A*)))

definition
Sup A = (
if A = {} ∨ *A* = {−∞} *then* −∞
else if ∞ ∈ *A* ∨ ¬ *bdd-above* (*Fin* −‘ *A*) *then* ∞
else Fin (*Sup* (*Fin* −‘ *A*)))

instance
proof *standard*
have [*dest*]: *Inf* (*Fin* −‘ *A*) ≤ *x* **if** *Fin x* ∈ *A bdd-below* (*Fin* −‘ *A*) **for** *A* **and** *x* :: *'a*
using *that* **by** (*intro cInf-lower*) *auto*
have ∗: *False* **if** ¬ *z* ≤ *Inf* (*Fin* −‘ *A*) ⋀*x*. *x* ∈ *A* ⟹ *Fin z* ≤ *x Fin x* ∈ *A* **for** *A* **and** *x z* :: *'a*
using *cInf-greatest*[*of Fin* −‘ *A z*] *that vimage-eq* **by** *force*

show *Inf A* ≤ *x* **if** *x* ∈ *A* **for** *x* :: *'a extended* **and** *A*
using *that* **unfolding** *Inf-extended-def* **by** (*cases x*) *auto*
show *z* ≤ *Inf A* **if** ⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x* **for** *z* :: *'a extended* **and** *A*
using *that*
unfolding *Inf-extended-def*
apply (*clarsimp*; *safe*)
apply *force*
apply *force*
subgoal
by (*cases z*; *force simp*: *bdd-below-def*)
subgoal
by (*cases z*; *force simp*: *bdd-below-def*)
subgoal for *x y*
by (*cases z*; *cases y*) (*auto elim*: ∗)
subgoal for *x y*
by (*cases z*; *cases y*; *simp*; *metis* ∗ *less-eq-extended.elims*(*2*))
done
have [*dest*]: *x* ≤ *Sup* (*Fin* −‘ *A*) **if** *Fin x* ∈ *A bdd-above* (*Fin* −‘ *A*) **for** *A* **and** *x* :: *'a*
using *that* **by** (*intro cSup-upper*) *auto*
have ∗: *False* **if** ¬ *Sup* (*Fin* −‘ *A*) ≤ *z* ⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *Fin z Fin x* ∈ *A* **for** *A* **and** *x z* :: *'a*
using *cSup-least*[*of Fin* −‘ *A z*] *that vimage-eq* **by** *force*
show *x* ≤ *Sup A* **if** *x* ∈ *A* **for** *x* :: *'a extended* **and** *A*
using *that* **unfolding** *Sup-extended-def* **by** (*cases x*) *auto*
show *Sup A* ≤ *z* **if** ⋀*x*. *x* ∈ *A* ⟹ *x* ≤ *z* **for** *z* :: *'a extended* **and** *A*
using *that*
unfolding *Sup-extended-def*
apply (*clarsimp*; *safe*)
apply *force*
apply *force*
subgoal
by (*cases z*; *force*)
subgoal
by (*cases z*; *force*)
subgoal for *x y*
by (*cases z*; *cases y*) (*auto elim*: ∗)
subgoal for *x y*
by (*cases z*; *cases y*; *simp*; *metis* ∗ *extended.exhaust*)
done
show *Inf* {} = (*top*::*'a extended*)
unfolding *Inf-extended-def top-extended-def* **by** *simp*
show *Sup* {} = (*bot*::*'a extended*)
unfolding *Sup-extended-def bot-extended-def* **by** *simp*
qed

end

instance *extended* :: ({*conditionally-complete-lattice*,*linorder*}) *complete-linorder*

..

end

35 Continuity and iterations

theory *Order-Continuity*
imports *Complex-Main Countable-Complete-Lattices*
begin

lemma *SUP-nat-binary*:
(*sup A* (*SUP x∈Collect* ((<) (*0*::*nat*)). *B*)) = (*sup A B*::'*a*::*countable-complete-lattice*)
apply (*subst image-constant*)
apply *auto*
done

lemma *INF-nat-binary*:
inf A (*INF x∈Collect* ((<) (*0*::*nat*)). *B*) = (*inf A B*::'*a*::*countable-complete-lattice*)
apply (*subst image-constant*)
apply *auto*
done

The name *continuous* is already taken in *Complex-Main*, so we use *sup-continuous* and *inf-continuous*. These names appear sometimes in literature and have the advantage that these names are duals.

named-theorems *order-continuous-intros*

35.1 Continuity for complete lattices

definition
sup-continuous :: ('*a*::*countable-complete-lattice* ⇒ '*b*::*countable-complete-lattice*) ⇒ *bool*
where
sup-continuous F ⟷ (∀ *M*::*nat* ⇒ '*a*. *mono M* ⟶ *F* (*SUP i*. *M i*) = (*SUP i*. *F* (*M i*)))

lemma *sup-continuousD*: *sup-continuous F* ⟹ *mono M* ⟹ *F* (*SUP i*::*nat*. *M i*) = (*SUP i*. *F* (*M i*))
by (*auto simp*: *sup-continuous-def*)

lemma *sup-continuous-mono*:
mono F **if** *sup-continuous F*
proof
fix *A B* :: '*a*
assume *A* ≤ *B*
let *?f* = λ*n*::*nat*. *if n* = *0 then A else B*
from ‹*A* ≤ *B*› **have** *incseq ?f*

by (*auto intro*: *monoI*)
with ‹*sup-continuous F*› **have** ∗: *F* (*SUP i*. *?f i*) = (*SUP i*. *F* (*?f i*))
by (*auto dest*: *sup-continuousD*)
from ‹*A* ≤ *B*› **have** *B* = *sup A B*
by (*simp add*: *le-iff-sup*)
then have *F B* = *F* (*sup A B*)
by *simp*
also have ... = *sup* (*F A*) (*F B*)
using ∗ **by** (*simp add*: *if-distrib SUP-nat-binary cong del*: *SUP-cong*)
finally show *F A* ≤ *F B*
by (*simp add*: *le-iff-sup*)
qed

lemma [*order-continuous-intros*]:
shows *sup-continuous-const*: *sup-continuous* (λ*x*. *c*)
and *sup-continuous-id*: *sup-continuous* (λ*x*. *x*)
and *sup-continuous-apply*: *sup-continuous* (λ*f*. *f x*)
and *sup-continuous-fun*: (⋀*s*. *sup-continuous* (λ*x*. *P x s*)) ⟹ *sup-continuous P*
and *sup-continuous-If*: *sup-continuous F* ⟹ *sup-continuous G* ⟹ *sup-continuous* (λ*f*. *if C then F f else G f*)
by (*auto simp*: *sup-continuous-def image-comp*)

lemma *sup-continuous-compose*:
assumes *f*: *sup-continuous f* **and** *g*: *sup-continuous g*
shows *sup-continuous* (λ*x*. *f* (*g x*))
unfolding *sup-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'c*
assume *M*: *mono M*
then have *mono* (λ*i*. *g* (*M i*))
using *sup-continuous-mono*[*OF g*] **by** (*auto simp*: *mono-def*)
with *M* **show** *f* (*g* (*Sup* (*M* ‘ *UNIV*))) = (*SUP i*. *f* (*g* (*M i*)))
by (*auto simp*: *sup-continuous-def g*[*THEN sup-continuousD*] *f*[*THEN sup-continuousD*])
qed

lemma *sup-continuous-sup*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous g* ⟹ *sup-continuous* (λ*x*. *sup* (*f x*) (*g x*))
by (*simp add*: *sup-continuous-def ccSUP-sup-distrib*)

lemma *sup-continuous-inf*[*order-continuous-intros*]:
fixes *P Q* :: *'a* :: *countable-complete-lattice* ⇒ *'b* :: *countable-complete-distrib-lattice*
assumes *P*: *sup-continuous P* **and** *Q*: *sup-continuous Q*
shows *sup-continuous* (λ*x*. *inf* (*P x*) (*Q x*))
unfolding *sup-continuous-def*
proof (*safe intro*!: *antisym*)
fix *M* :: *nat* ⇒ *'a* **assume** *M*: *incseq M*
have *inf* (*P* (*SUP i*. *M i*)) (*Q* (*SUP i*. *M i*)) ≤ (*SUP j i*. *inf* (*P* (*M i*)) (*Q* (*M j*)))

by (*simp add*: *sup-continuousD*[*OF P M*] *sup-continuousD*[*OF Q M*] *inf-ccSUP ccSUP-inf*)
also have ... ≤ (*SUP i*. *inf* (*P* (*M i*)) (*Q* (*M i*)))
proof (*intro ccSUP-least*)
fix *i j* **from** *M assms*[*THEN sup-continuous-mono*] **show** *inf* (*P* (*M i*)) (*Q* (*M j*)) ≤ (*SUP i*. *inf* (*P* (*M i*)) (*Q* (*M i*)))
by (*intro ccSUP-upper2*[*of - sup i j*] *inf-mono*) (*auto simp*: *mono-def*)
qed *auto*
finally show *inf* (*P* (*SUP i*. *M i*)) (*Q* (*SUP i*. *M i*)) ≤ (*SUP i*. *inf* (*P* (*M i*)) (*Q* (*M i*))) **.**

show (*SUP i*. *inf* (*P* (*M i*)) (*Q* (*M i*))) ≤ *inf* (*P* (*SUP i*. *M i*)) (*Q* (*SUP i*. *M i*))
unfolding *sup-continuousD*[*OF P M*] *sup-continuousD*[*OF Q M*] **by** (*intro ccSUP-least inf-mono ccSUP-upper*) *auto*
qed

lemma *sup-continuous-and*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∧ *Q x*)
using *sup-continuous-inf*[*of P Q*] **by** *simp*

lemma *sup-continuous-or*[*order-continuous-intros*]:
sup-continuous P ⟹ *sup-continuous Q* ⟹ *sup-continuous* (λ*x*. *P x* ∨ *Q x*)
by (*auto simp*: *sup-continuous-def*)

lemma *sup-continuous-lfp*:
assumes *sup-continuous F* **shows** *lfp F* = (*SUP i*. (*F* ^^ *i*) *bot*) (**is** *lfp F* = *?U*)
proof (*rule antisym*)
note *mono* = *sup-continuous-mono*[*OF* ‹*sup-continuous F*›]
show *?U* ≤ *lfp F*
proof (*rule SUP-least*)
fix *i* **show** (*F* ^^ *i*) *bot* ≤ *lfp F*
proof (*induct i*)
case (*Suc i*)
have (*F* ^^ *Suc i*) *bot* = *F* ((*F* ^^ *i*) *bot*) **by** *simp*
also have ... ≤ *F* (*lfp F*) **by** (*rule monoD*[*OF mono Suc*])
also have ... = *lfp F* **by** (*simp add*: *lfp-fixpoint*[*OF mono*])
finally show *?case* **.**
qed *simp*
qed
show *lfp F* ≤ *?U*
proof (*rule lfp-lowerbound*)
have *mono* (λ*i*::*nat*. (*F* ^^ *i*) *bot*)
proof −
have (*F* ^^ *i*) *bot* ≤ (*F* ^^ (*Suc i*)) *bot* **for** *i*::*nat*
proof (*induct i*)
case *0* **show** *?case* **by** *simp*
next
case *Suc* **thus** *?case* **using** *monoD*[*OF mono Suc*] **by** *auto*

```
    qed
    thus ?thesis by (auto simp add: mono-iff-le-Suc)
   qed
   hence F ?U = (SUP i. (F ^^ Suc i) bot)
    using ‹sup-continuous F› by (simp add: sup-continuous-def)
   also have ... ≤ ?U
    by (fast intro: SUP-least SUP-upper)
   finally show F ?U ≤ ?U .
  qed
qed

lemma lfp-transfer-bounded:
  assumes P: P bot ⋀x. P x ⟹ P (f x) ⋀M. (⋀i. P (M i)) ⟹ P (SUP i::nat.
M i)
  assumes α: ⋀M. mono M ⟹ (⋀i::nat. P (M i)) ⟹ α (SUP i. M i) = (SUP
i. α (M i))
  assumes f: sup-continuous f and g: sup-continuous g
  assumes [simp]: ⋀x. P x ⟹ x ≤ lfp f ⟹ α (f x) = g (α x)
  assumes g-bound: ⋀x. α bot ≤ g x
  shows α (lfp f) = lfp g
proof (rule antisym)
  note mono-g = sup-continuous-mono[OF g]
  note mono-f = sup-continuous-mono[OF f]
  have lfp-bound: α bot ≤ lfp g
   by (subst lfp-unfold[OF mono-g]) (rule g-bound)

  have P-pow: P ((f ^^ i) bot) for i
   by (induction i) (auto intro!: P)
  have incseq-pow: mono (λi. (f ^^ i) bot)
   unfolding mono-iff-le-Suc
  proof
   fix i show (f ^^ i) bot ≤ (f ^^ (Suc i)) bot
   proof (induct i)
    case Suc thus ?case using monoD[OF sup-continuous-mono[OF f] Suc] by
auto
   qed (simp add: le-fun-def)
  qed
  have P-lfp: P (lfp f)
   using P-pow unfolding sup-continuous-lfp[OF f] by (auto intro!: P)

  have iter-le-lfp: (f ^^ n) bot ≤ lfp f for n
   apply (induction n)
   apply simp
   apply (subst lfp-unfold[OF mono-f])
   apply (auto intro!: monoD[OF mono-f])
   done

  have α (lfp f) = (SUP i. α ((f^^i) bot))
   unfolding sup-continuous-lfp[OF f] using incseq-pow P-pow by (rule α)
```

```
  also have ... ≤ lfp g
  proof (rule SUP-least)
    fix i show α ((f^^i) bot) ≤ lfp g
    proof (induction i)
      case (Suc n) then show ?case
        by (subst lfp-unfold[OF mono-g]) (simp add: monoD[OF mono-g] P-pow
iter-le-lfp)
    qed (simp add: lfp-bound)
  qed
  finally show α (lfp f) ≤ lfp g .

  show lfp g ≤ α (lfp f)
  proof (induction rule: lfp-ordinal-induct[OF mono-g])
    case (1 S) then show ?case
      by (subst lfp-unfold[OF sup-continuous-mono[OF f]])
        (simp add: monoD[OF mono-g] P-lfp)
  qed (auto intro: Sup-least)
qed

lemma lfp-transfer:
  sup-continuous α ⟹ sup-continuous f ⟹ sup-continuous g ⟹
    (⋀x. α bot ≤ g x) ⟹ (⋀x. x ≤ lfp f ⟹ α (f x) = g (α x)) ⟹ α (lfp f) =
lfp g
  by (rule lfp-transfer-bounded[where P=top]) (auto dest: sup-continuousD)

definition
  inf-continuous :: ('a::countable-complete-lattice ⇒ 'b::countable-complete-lattice)
⇒ bool
where
  inf-continuous F ⟷ (∀ M::nat ⇒ 'a. antimono M ⟶ F (INF i. M i) = (INF
i. F (M i)))

lemma inf-continuousD: inf-continuous F ⟹ antimono M ⟹ F (INF i::nat. M
i) = (INF i. F (M i))
  by (auto simp: inf-continuous-def)

lemma inf-continuous-mono:
  mono F if inf-continuous F
proof
  fix A B :: 'a
  assume A ≤ B
  let ?f = λn::nat. if n = 0 then B else A
  from ‹A ≤ B› have decseq ?f
    by (auto intro: antimonoI)
  with ‹inf-continuous F› have *: F (INF i. ?f i) = (INF i. F (?f i))
    by (auto dest: inf-continuousD)
  from ‹A ≤ B› have A = inf B A
    by (simp add: inf.absorb-iff2)
  then have F A = F (inf B A)
```

by *simp*
also have ... = *inf* (*F B*) (*F A*)
using ∗ **by** (*simp add*: *if-distrib INF-nat-binary cong del*: *INF-cong*)
finally show *F A* ≤ *F B*
by (*simp add*: *inf.absorb-iff2*)
qed

lemma [*order-continuous-intros*]:
shows *inf-continuous-const*: *inf-continuous* (λ*x*. *c*)
and *inf-continuous-id*: *inf-continuous* (λ*x*. *x*)
and *inf-continuous-apply*: *inf-continuous* (λ*f*. *f x*)
and *inf-continuous-fun*: (⋀*s*. *inf-continuous* (λ*x*. *P x s*)) ⟹ *inf-continuous P*
and *inf-continuous-If*: *inf-continuous F* ⟹ *inf-continuous G* ⟹ *inf-continuous* (λ*f*. *if C then F f else G f*)
by (*auto simp*: *inf-continuous-def image-comp*)

lemma *inf-continuous-inf*[*order-continuous-intros*]:
inf-continuous f ⟹ *inf-continuous g* ⟹ *inf-continuous* (λ*x*. *inf* (*f x*) (*g x*))
by (*simp add*: *inf-continuous-def ccINF-inf-distrib*)

lemma *inf-continuous-sup*[*order-continuous-intros*]:
fixes *P Q* :: *'a* :: *countable-complete-lattice* ⇒ *'b* :: *countable-complete-distrib-lattice*
assumes *P*: *inf-continuous P* **and** *Q*: *inf-continuous Q*
shows *inf-continuous* (λ*x*. *sup* (*P x*) (*Q x*))
unfolding *inf-continuous-def*
proof (*safe intro*!: *antisym*)
fix *M* :: *nat* ⇒ *'a* **assume** *M*: *decseq M*
show *sup* (*P* (*INF i*. *M i*)) (*Q* (*INF i*. *M i*)) ≤ (*INF i*. *sup* (*P* (*M i*)) (*Q* (*M i*)))
unfolding *inf-continuousD*[*OF P M*] *inf-continuousD*[*OF Q M*] **by** (*intro ccINF-greatest sup-mono ccINF-lower*) *auto*

have (*INF i*. *sup* (*P* (*M i*)) (*Q* (*M i*))) ≤ (*INF j i*. *sup* (*P* (*M i*)) (*Q* (*M j*)))
proof (*intro ccINF-greatest*)
fix *i j* **from** *M assms*[*THEN inf-continuous-mono*] **show** *sup* (*P* (*M i*)) (*Q* (*M j*)) ≥ (*INF i*. *sup* (*P* (*M i*)) (*Q* (*M i*)))
by (*intro ccINF-lower2*[*of - sup i j*] *sup-mono*) (*auto simp*: *mono-def anti-mono-def*)
qed *auto*
also have ... ≤ *sup* (*P* (*INF i*. *M i*)) (*Q* (*INF i*. *M i*))
by (*simp add*: *inf-continuousD*[*OF P M*] *inf-continuousD*[*OF Q M*] *ccINF-sup sup-ccINF*)
finally show *sup* (*P* (*INF i*. *M i*)) (*Q* (*INF i*. *M i*)) ≥ (*INF i*. *sup* (*P* (*M i*)) (*Q* (*M i*))) **.**
qed

lemma *inf-continuous-and*[*order-continuous-intros*]:
inf-continuous P ⟹ *inf-continuous Q* ⟹ *inf-continuous* (λ*x*. *P x* ∧ *Q x*)
using *inf-continuous-inf*[*of P Q*] **by** *simp*

lemma *inf-continuous-or*[*order-continuous-intros*]:
  inf-continuous P ⟹ *inf-continuous Q* ⟹ *inf-continuous* (λ*x*. *P x* ∨ *Q x*)
  using *inf-continuous-sup*[*of P Q*] **by** *simp*

lemma *inf-continuous-compose*:
  assumes *f*: *inf-continuous f* **and** *g*: *inf-continuous g*
  shows *inf-continuous* (λ*x*. *f* (*g x*))
  unfolding *inf-continuous-def*
proof *safe*
  fix *M* :: *nat* ⇒ *'c*
  assume *M*: *antimono M*
  then have *antimono* (λ*i*. *g* (*M i*))
    using *inf-continuous-mono*[*OF g*] **by** (*auto simp*: *mono-def antimono-def*)
  with *M* **show** *f* (*g* (*Inf* (*M* ‘ *UNIV*))) = (*INF i*. *f* (*g* (*M i*)))
   by (*auto simp*: *inf-continuous-def g*[*THEN inf-continuousD*] *f*[*THEN inf-continuousD*])
qed

lemma *inf-continuous-gfp*:
  assumes *inf-continuous F* **shows** *gfp F* = (*INF i*. (*F* ^^ *i*) *top*) (**is** *gfp F* = *?U*)
proof (*rule antisym*)
  note *mono* = *inf-continuous-mono*[*OF* ‹*inf-continuous F*›]
  show *gfp F* ≤ *?U*
  proof (*rule INF-greatest*)
    fix *i* **show** *gfp F* ≤ (*F* ^^ *i*) *top*
    proof (*induct i*)
      case (*Suc i*)
      have *gfp F* = *F* (*gfp F*) **by** (*simp add*: *gfp-fixpoint*[*OF mono*])
      also have ... ≤ *F* ((*F* ^^ *i*) *top*) **by** (*rule monoD*[*OF mono Suc*])
      also have ... = (*F* ^^ *Suc i*) *top* **by** *simp*
      finally show *?case* **.**
    qed *simp*
  qed
  show *?U* ≤ *gfp F*
  proof (*rule gfp-upperbound*)
    have ∗: *antimono* (λ*i*::*nat*. (*F* ^^ *i*) *top*)
    proof −
      have (*F* ^^ *Suc i*) *top* ≤ (*F* ^^ *i*) *top* **for** *i*::*nat*
      proof (*induct i*)
        case *0*
        show *?case* **by** *simp*
      next
        case *Suc*
        thus *?case* **using** *monoD*[*OF mono Suc*] **by** *auto*
      qed
      thus *?thesis* **by** (*auto simp add*: *antimono-iff-le-Suc*)
    qed
    have *?U* ≤ (*INF i*. (*F* ^^ *Suc i*) *top*)
      by (*fast intro*: *INF-greatest INF-lower*)

```
    also have ... ≤ F ?U
      by (simp add: inf-continuousD ‹inf-continuous F› *)
    finally show ?U ≤ F ?U .
  qed
qed

lemma gfp-transfer:
  assumes α: inf-continuous α and f: inf-continuous f and g: inf-continuous g
  assumes [simp]: α top = top ⋀x. α (f x) = g (α x)
  shows α (gfp f) = gfp g
proof −
  have α (gfp f) = (INF i. α ((f^^i) top))
   unfolding inf-continuous-gfp[OF f] by (intro f α inf-continuousD antimono-funpow
inf-continuous-mono)
  moreover have α ((f^^i) top) = (g^^i) top for i
    by (induction i; simp)
  ultimately show ?thesis
    unfolding inf-continuous-gfp[OF g] by simp
qed

lemma gfp-transfer-bounded:
  assumes P: P (f top) ⋀x. P x ⟹ P (f x) ⋀M. antimono M ⟹ (⋀i. P (M
i)) ⟹ P (INF i::nat. M i)
  assumes α: ⋀M. antimono M ⟹ (⋀i::nat. P (M i)) ⟹ α (INF i. M i) =
(INF i. α (M i))
  assumes f: inf-continuous f and g: inf-continuous g
  assumes [simp]: ⋀x. P x ⟹ α (f x) = g (α x)
  assumes g-bound: ⋀x. g x ≤ α (f top)
  shows α (gfp f) = gfp g
proof (rule antisym)
  note mono-g = inf-continuous-mono[OF g]

  have P-pow: P ((f ^^ i) (f top)) for i
    by (induction i) (auto intro!: P)

  have antimono-pow: antimono (λi. (f ^^ i) top)
    unfolding antimono-iff-le-Suc
  proof
    fix i show (f ^^ Suc i) top ≤ (f ^^ i) top
    proof (induct i)
      case Suc thus ?case using monoD[OF inf-continuous-mono[OF f] Suc] by
auto
    qed (simp add: le-fun-def)
  qed
  have antimono-pow2: antimono (λi. (f ^^ i) (f top))
  proof
    show x ≤ y ⟹ (f ^^ y) (f top) ≤ (f ^^ x) (f top) for x y
      using antimono-pow[THEN antimonoD, of Suc x Suc y]
      unfolding funpow-Suc-right by simp
```

```
  qed

  have gfp-f: gfp f = (INF i. (f ^^ i) (f top))
    unfolding inf-continuous-gfp[OF f]
  proof (rule INF-eq)
    show ∃j∈UNIV. (f ^^ j) (f top) ≤ (f ^^ i) top for i
      by (intro bexI[of - i − 1]) (auto simp: diff-Suc funpow-Suc-right simp del:
funpow.simps(2) split: nat.split)
    show ∃j∈UNIV. (f ^^ j) top ≤ (f ^^ i) (f top) for i
      by (intro bexI[of - Suc i]) (auto simp: funpow-Suc-right simp del: fun-
pow.simps(2))
  qed

  have P-lfp: P (gfp f)
    unfolding gfp-f by (auto intro!: P P-pow antimono-pow2)

  have α (gfp f) = (INF i. α ((f^^i) (f top)))
    unfolding gfp-f by (rule α) (auto intro!: P-pow antimono-pow2)
  also have ... ≥ gfp g
  proof (rule INF-greatest)
    fix i show gfp g ≤ α ((f^^i) (f top))
    proof (induction i)
      case (Suc n) then show ?case
        by (subst gfp-unfold[OF mono-g]) (simp add: monoD[OF mono-g] P-pow)
    next
      case 0
      have gfp g ≤ α (f top)
        by (subst gfp-unfold[OF mono-g]) (rule g-bound)
      then show ?case
        by simp
    qed
  qed
  finally show gfp g ≤ α (gfp f) .

  show α (gfp f) ≤ gfp g
  proof (induction rule: gfp-ordinal-induct[OF mono-g])
    case (1 S) then show ?case
      by (subst gfp-unfold[OF inf-continuous-mono[OF f]])
        (simp add: monoD[OF mono-g] P-lfp)
  qed (auto intro: Inf-greatest)
qed
```

35.1.1 Least fixed points in countable complete lattices

```
definition (in countable-complete-lattice) cclfp :: ('a ⇒ 'a) ⇒ 'a
  where cclfp f = (SUP i. (f ^^ i) bot)

lemma cclfp-unfold:
  assumes sup-continuous F shows cclfp F = F (cclfp F)
```

proof −
have *cclfp F* = (*SUP i*. *F* ((*F* ^^ *i*) *bot*))
unfolding *cclfp-def*
by (*subst UNIV-nat-eq*) (*simp add*: *image-comp*)
also have ... = *F* (*cclfp F*)
unfolding *cclfp-def*
by (*intro sup-continuousD*[*symmetric*] *assms mono-funpow sup-continuous-mono*)
finally show *?thesis* **.**
qed

lemma *cclfp-lowerbound*: **assumes** *f*: *mono f* **and** *A*: *f A* ≤ *A* **shows** *cclfp f* ≤ *A*
unfolding *cclfp-def*
proof (*intro ccSUP-least*)
fix *i* **show** (*f* ^^ *i*) *bot* ≤ *A*
proof (*induction i*)
case (*Suc i*) **from** *monoD*[*OF f this*] *A* **show** *?case*
by *auto*
qed *simp*
qed *simp*

lemma *cclfp-transfer*:
assumes *sup-continuous* α *mono f*
assumes α *bot* = *bot* ⋀*x*. α (*f x*) = *g* (α *x*)
shows α (*cclfp f*) = *cclfp g*
proof −
have α (*cclfp f*) = (*SUP i*. α ((*f* ^^ *i*) *bot*))
unfolding *cclfp-def* **by** (*intro sup-continuousD assms mono-funpow sup-continuous-mono*)
moreover have α ((*f* ^^ *i*) *bot*) = (*g* ^^ *i*) *bot* **for** *i*
by (*induction i*) (*simp-all add*: *assms*)
ultimately show *?thesis*
by (*simp add*: *cclfp-def*)
qed

end

36 Extended natural numbers (i.e. with infinity)

theory *Extended-Nat*
imports *Main Countable Order-Continuity*
begin

class *infinity* =
fixes *infinity* :: *'a* (‹∞›)

context
fixes *f* :: *nat* ⇒ *'a*::{*canonically-ordered-monoid-add*, *linorder-topology*, *complete-linorder*}
begin

lemma *sums-SUP*[*simp*, *intro*]: *f sums* (*SUP n.* $\sum i{<}n.$ *f i*)
unfolding *sums-def* **by** (*intro LIMSEQ-SUP monoI sum-mono2 zero-le*) *auto*

lemma *suminf-eq-SUP*: *suminf f* = (*SUP n.* $\sum i{<}n.$ *f i*)
using *sums-SUP* **by** (*rule sums-unique*[*symmetric*])

end

36.1 Type definition

We extend the standard natural numbers by a special value indicating infinity.

typedef *enat* = *UNIV* :: *nat option set* **..**

TODO: introduce enat as coinductive datatype, enat is just *of-nat*

definition *enat* :: *nat* ⇒ *enat* **where**
enat n = *Abs-enat* (*Some n*)

instantiation *enat* :: *infinity*
begin

definition ∞ = *Abs-enat None*
instance ..

end

instance *enat* :: *countable*
proof
show ∃ *to-nat*::*enat* ⇒ *nat*. *inj to-nat*
by (*rule exI*[*of - to-nat* ∘ *Rep-enat*]) (*simp add*: *inj-on-def Rep-enat-inject*)
qed

old-rep-datatype *enat* ∞ :: *enat*
proof −
fix *P i* **assume** ⋀*j*. *P* (*enat j*) *P* ∞
then show *P i*
proof *induct*
case (*Abs-enat y*) **then show** *?case*
by (*cases y rule*: *option.exhaust*)
(*auto simp*: *enat-def infinity-enat-def*)
qed
qed (*auto simp add*: *enat-def infinity-enat-def Abs-enat-inject*)

declare [[*coercion enat*::*nat*⇒*enat*]]

lemmas *enat2-cases* = *enat.exhaust*[*case-product enat.exhaust*]
lemmas *enat3-cases* = *enat.exhaust*[*case-product enat.exhaust enat.exhaust*]

lemma *not-infinity-eq* [*iff*]: (*x* ≠ ∞) = (∃ *i*. *x* = *enat i*)

by (*cases x*) *auto*

lemma *not-enat-eq* [*iff*]: $(\forall y.\ x \neq enat\ y) = (x = \infty)$
by (*cases x*) *auto*

lemma *enat-ex-split*: $(\exists c{::}enat.\ P\ c) \longleftrightarrow P\ \infty \vee (\exists c{::}nat.\ P\ c)$
by (*metis enat.exhaust*)

primrec *the-enat* :: *enat* ⇒ *nat*
where *the-enat* (*enat n*) = *n*

36.2 Constructors and numbers

instantiation *enat* :: *zero-neq-one*
begin

definition
0 = *enat 0*

definition
1 = *enat 1*

instance
proof qed (*simp add*: *zero-enat-def one-enat-def*)

end

definition *eSuc* :: *enat* ⇒ *enat* **where**
eSuc i = (*case i of enat n* ⇒ *enat* (*Suc n*) | ∞ ⇒ ∞)

lemma *enat-0* [*code-post*]: *enat 0* = *0*
by (*simp add*: *zero-enat-def*)

lemma *enat-1* [*code-post*]: *enat 1* = *1*
by (*simp add*: *one-enat-def*)

lemma *enat-0-iff*: *enat x* = *0* ⟷ *x* = *0* *0* = *enat x* ⟷ *x* = *0*
by (*auto simp add*: *zero-enat-def*)

lemma *enat-1-iff*: *enat x* = *1* ⟷ *x* = *1* *1* = *enat x* ⟷ *x* = *1*
by (*auto simp add*: *one-enat-def*)

lemma *one-eSuc*: *1* = *eSuc 0*
by (*simp add*: *zero-enat-def one-enat-def eSuc-def*)

lemma *infinity-ne-i0* [*simp*]: (∞::*enat*) ≠ *0*
by (*simp add*: *zero-enat-def*)

lemma *i0-ne-infinity* [*simp*]: *0* ≠ (∞::*enat*)

by (*simp add*: *zero-enat-def*)

lemma *zero-one-enat-neq*:
¬ *0* = (*1*::*enat*)
¬ *1* = (*0*::*enat*)
unfolding *zero-enat-def one-enat-def* **by** *simp-all*

lemma *infinity-ne-i1* [*simp*]: (∞::*enat*) ≠ *1*
by (*simp add*: *one-enat-def*)

lemma *i1-ne-infinity* [*simp*]: *1* ≠ (∞::*enat*)
by (*simp add*: *one-enat-def*)

lemma *eSuc-enat*: *eSuc* (*enat n*) = *enat* (*Suc n*)
by (*simp add*: *eSuc-def*)

lemma *eSuc-infinity* [*simp*]: *eSuc* ∞ = ∞
by (*simp add*: *eSuc-def*)

lemma *eSuc-ne-0* [*simp*]: *eSuc n* ≠ *0*
by (*simp add*: *eSuc-def zero-enat-def split*: *enat.splits*)

lemma *zero-ne-eSuc* [*simp*]: *0* ≠ *eSuc n*
by (*rule eSuc-ne-0* [*symmetric*])

lemma *eSuc-inject* [*simp*]: *eSuc m* = *eSuc n* ⟷ *m* = *n*
by (*simp add*: *eSuc-def split*: *enat.splits*)

lemma *eSuc-enat-iff*: *eSuc x* = *enat y* ⟷ (∃ *n*. *y* = *Suc n* ∧ *x* = *enat n*)
by (*cases y*) (*auto simp*: *enat-0 eSuc-enat*[*symmetric*])

lemma *enat-eSuc-iff*: *enat y* = *eSuc x* ⟷ (∃ *n*. *y* = *Suc n* ∧ *enat n* = *x*)
by (*cases y*) (*auto simp*: *enat-0 eSuc-enat*[*symmetric*])

36.3 Addition

instantiation *enat* :: *comm-monoid-add*
begin

definition [*nitpick-simp*]:
m + *n* = (*case m of* ∞ ⇒ ∞ | *enat m* ⇒ (*case n of* ∞ ⇒ ∞ | *enat n* ⇒ *enat* (*m* + *n*)))

lemma *plus-enat-simps* [*simp*, *code*]:
fixes *q* :: *enat*
shows *enat m* + *enat n* = *enat* (*m* + *n*)
and ∞ + *q* = ∞
and *q* + ∞ = ∞
by (*simp-all add*: *plus-enat-def split*: *enat.splits*)

instance
proof
fix *n m q* :: *enat*
show *n + m + q = n + (m + q)*
by (*cases n m q rule*: *enat3-cases*) *auto*
show *n + m = m + n*
by (*cases n m rule*: *enat2-cases*) *auto*
show *0 + n = n*
by (*cases n*) (*simp-all add*: *zero-enat-def*)
qed

end

lemma *eSuc-plus-1*:
eSuc n = n + 1
by (*cases n*) (*simp-all add*: *eSuc-enat one-enat-def*)

lemma *plus-1-eSuc*:
1 + q = eSuc q
q + 1 = eSuc q
by (*simp-all add*: *eSuc-plus-1 ac-simps*)

lemma *iadd-Suc*: *eSuc m + n = eSuc (m + n)*
by (*simp add*: *eSuc-plus-1 ac-simps*)

lemma *iadd-Suc-right*: *m + eSuc n = eSuc (m + n)*
by (*metis add.commute iadd-Suc*)

36.4 Multiplication

instantiation *enat* :: {*comm-semiring-1*, *semiring-no-zero-divisors*}
begin

definition *times-enat-def* [*nitpick-simp*]:
*m * n = (case m of ∞ ⇒ if n = 0 then 0 else ∞ | enat m ⇒*
*(case n of ∞ ⇒ if m = 0 then 0 else ∞ | enat n ⇒ enat (m * n)))*

lemma *times-enat-simps* [*simp*, *code*]:
*enat m * enat n = enat (m * n)*
*∞ * ∞ = (∞::enat)*
*∞ * enat n = (if n = 0 then 0 else ∞)*
*enat m * ∞ = (if m = 0 then 0 else ∞)*
unfolding *times-enat-def zero-enat-def*
by (*simp-all split*: *enat.split*)

instance
proof
fix *a b c* :: *enat*

show *distr*: $(a + b) * c = a * c + b * c$
unfolding *times-enat-def zero-enat-def*
by (*simp split*: *enat.split add*: *distrib-right*)
show $a * (b + c) = a * b + a * c$
by (*cases a b c rule*: *enat3-cases*) (*auto simp*: *times-enat-def zero-enat-def distrib-left*)
qed (*auto simp*: *times-enat-def zero-enat-def one-enat-def split*: *enat.split*)

end

lemma *mult-eSuc*: $eSuc\ m * n = n + m * n$
unfolding *eSuc-plus-1* **by** (*simp add*: *algebra-simps*)

lemma *mult-eSuc-right*: $m * eSuc\ n = m + m * n$
by (*metis mult.commute mult-eSuc*)

lemma *of-nat-eq-enat*: *of-nat n* = *enat n*
by (*induct n*) (*auto simp*: *enat-0 plus-1-eSuc eSuc-enat*)

instance *enat* :: *semiring-char-0*
proof
have *inj enat* **by** (*rule injI*) *simp*
then show *inj* (λ*n*. *of-nat n* :: *enat*) **by** (*simp add*: *of-nat-eq-enat*)
qed

lemma *imult-is-infinity*: $((a::enat) * b = \infty) = (a = \infty \wedge b \neq 0 \vee b = \infty \wedge a \neq 0)$
by (*auto simp add*: *times-enat-def zero-enat-def split*: *enat.split*)

36.5 Numerals

lemma *numeral-eq-enat*:
numeral k = *enat* (*numeral k*)
by (*metis of-nat-eq-enat of-nat-numeral*)

lemma *enat-numeral* [*code-abbrev*]:
enat (*numeral k*) = *numeral k*
using *numeral-eq-enat* **..**

lemma *infinity-ne-numeral* [*simp*]: $(\infty::enat) \neq numeral\ k$
by (*simp add*: *numeral-eq-enat*)

lemma *numeral-ne-infinity* [*simp*]: *numeral k* ≠ (∞::*enat*)
by (*simp add*: *numeral-eq-enat*)

lemma *eSuc-numeral* [*simp*]: *eSuc* (*numeral k*) = *numeral* (*k* + *Num.One*)
by (*simp only*: *eSuc-plus-1 numeral-plus-one*)

36.6 Subtraction

instantiation *enat* :: *minus*
begin

definition *diff-enat-def*:
$a - b = (case\ a\ of\ (enat\ x) \Rightarrow (case\ b\ of\ (enat\ y) \Rightarrow enat\ (x - y) \mid \infty \Rightarrow 0) \mid \infty \Rightarrow \infty)$

instance ..

end

lemma *idiff-enat-enat* [*simp*, *code*]: $enat\ a - enat\ b = enat\ (a - b)$
by (*simp add*: *diff-enat-def*)

lemma *idiff-infinity* [*simp*, *code*]: $\infty - n = (\infty::enat)$
by (*simp add*: *diff-enat-def*)

lemma *idiff-infinity-right* [*simp*, *code*]: $enat\ a - \infty = 0$
by (*simp add*: *diff-enat-def*)

lemma *idiff-0* [*simp*]: $(0::enat) - n = 0$
by (*cases n*, *simp-all add*: *zero-enat-def*)

lemmas *idiff-enat-0* [*simp*] = *idiff-0* [*unfolded zero-enat-def*]

lemma *idiff-0-right* [*simp*]: $(n::enat) - 0 = n$
by (*cases n*) (*simp-all add*: *zero-enat-def*)

lemmas *idiff-enat-0-right* [*simp*] = *idiff-0-right* [*unfolded zero-enat-def*]

lemma *idiff-self* [*simp*]: $n \neq \infty \Longrightarrow (n::enat) - n = 0$
by (*auto simp*: *zero-enat-def*)

lemma *eSuc-minus-eSuc* [*simp*]: $eSuc\ n - eSuc\ m = n - m$
by (*simp add*: *eSuc-def split*: *enat.split*)

lemma *eSuc-minus-1* [*simp*]: $eSuc\ n - 1 = n$
by (*simp add*: *one-enat-def flip*: *eSuc-enat zero-enat-def*)

36.7 Ordering

instantiation *enat* :: *linordered-ab-semigroup-add*
begin

definition [*nitpick-simp*]:
$m \leq n = (case\ n\ of\ enat\ n1 \Rightarrow (case\ m\ of\ enat\ m1 \Rightarrow m1 \leq n1 \mid \infty \Rightarrow False) \mid \infty \Rightarrow True)$

definition [*nitpick-simp*]:
m < *n* = (*case m of enat m1* ⇒ (*case n of enat n1* ⇒ *m1* < *n1* | ∞ ⇒ *True*)
| ∞ ⇒ *False*)

lemma *enat-ord-simps* [*simp*]:
enat m ≤ *enat n* ⟷ *m* ≤ *n*
enat m < *enat n* ⟷ *m* < *n*
q ≤ (∞::*enat*)
q < (∞::*enat*) ⟷ *q* ≠ ∞
(∞::*enat*) ≤ *q* ⟷ *q* = ∞
(∞::*enat*) < *q* ⟷ *False*
by (*simp-all add*: *less-eq-enat-def less-enat-def split*: *enat.splits*)

lemma *numeral-le-enat-iff* [*simp*]:
shows *numeral m* ≤ *enat n* ⟷ *numeral m* ≤ *n*
by (*auto simp*: *numeral-eq-enat*)

lemma *numeral-less-enat-iff* [*simp*]:
shows *numeral m* < *enat n* ⟷ *numeral m* < *n*
by (*auto simp*: *numeral-eq-enat*)

lemma *enat-ord-code* [*code*]:
enat m ≤ *enat n* ⟷ *m* ≤ *n*
enat m < *enat n* ⟷ *m* < *n*
q ≤ (∞::*enat*) ⟷ *True*
enat m < ∞ ⟷ *True*
∞ ≤ *enat n* ⟷ *False*
(∞::*enat*) < *q* ⟷ *False*
by *simp-all*

instance
by *standard* (*auto simp add*: *less-eq-enat-def less-enat-def plus-enat-def split*: *enat.splits*)

end

instance *enat* :: *dioid*
proof
fix *a b* :: *enat* **show** (*a* ≤ *b*) = (∃ *c*. *b* = *a* + *c*)
by (*cases a b rule*: *enat2-cases*) (*auto simp*: *le-iff-add enat-ex-split*)
qed

instance *enat* :: {*linordered-nonzero-semiring*, *strict-ordered-comm-monoid-add*}
proof
fix *a b c* :: *enat*
show *a* ≤ *b* ⟹ *0* ≤ *c* ⟹*c* ∗ *a* ≤ *c* ∗ *b*
unfolding *times-enat-def less-eq-enat-def zero-enat-def*
by (*simp split*: *enat.splits*)
show *a* < *b* ⟹ *c* < *d* ⟹ *a* + *c* < *b* + *d* **for** *a b c d* :: *enat*

by (*cases a b c d rule*: *enat2-cases*[*case-product enat2-cases*]) *auto*
show $a < b \Longrightarrow a + 1 < b + 1$
by (*metis add-right-mono eSuc-minus-1 eSuc-plus-1 less-le*)
qed (*simp add*: *zero-enat-def one-enat-def*)

lemma *add-diff-assoc-enat*: $z \leq y \Longrightarrow x + (y - z) = x + y - (z::enat)$
by(*cases x*)(*auto simp add*: *diff-enat-def split*: *enat.split*)

lemma *enat-ord-number* [*simp*]:
$(numeral\ m :: enat) \leq numeral\ n \longleftrightarrow (numeral\ m :: nat) \leq numeral\ n$
$(numeral\ m :: enat) < numeral\ n \longleftrightarrow (numeral\ m :: nat) < numeral\ n$
by (*simp-all add*: *numeral-eq-enat*)

lemma *infinity-ileE* [*elim*!]: $\infty \leq enat\ m \Longrightarrow R$
by (*simp add*: *zero-enat-def less-eq-enat-def split*: *enat.splits*)

lemma *infinity-ilessE* [*elim*!]: $\infty < enat\ m \Longrightarrow R$
by *simp*

lemma *eSuc-ile-mono* [*simp*]: $eSuc\ n \leq eSuc\ m \longleftrightarrow n \leq m$
by (*simp add*: *eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *eSuc-mono* [*simp*]: $eSuc\ n < eSuc\ m \longleftrightarrow n < m$
by (*simp add*: *eSuc-def less-enat-def split*: *enat.splits*)

lemma *ile-eSuc* [*simp*]: $n \leq eSuc\ n$
by (*simp add*: *eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *not-eSuc-ilei0* [*simp*]: $\neg\ eSuc\ n \leq 0$
by (*simp add*: *zero-enat-def eSuc-def less-eq-enat-def split*: *enat.splits*)

lemma *i0-iless-eSuc* [*simp*]: $0 < eSuc\ n$
by (*simp add*: *zero-enat-def eSuc-def less-enat-def split*: *enat.splits*)

lemma *iless-eSuc0*[*simp*]: $(n < eSuc\ 0) = (n = 0)$
by (*simp add*: *zero-enat-def eSuc-def less-enat-def split*: *enat.split*)

lemma *ileI1*: $m < n \Longrightarrow eSuc\ m \leq n$
by (*simp add*: *eSuc-def less-eq-enat-def less-enat-def split*: *enat.splits*)

lemma *Suc-ile-eq*: $enat\ (Suc\ m) \leq n \longleftrightarrow enat\ m < n$
by (*cases n*) *auto*

lemma *iless-Suc-eq* [*simp*]: $enat\ m < eSuc\ n \longleftrightarrow enat\ m \leq n$
by (*auto simp add*: *eSuc-def less-enat-def split*: *enat.splits*)

lemma *imult-infinity*: $(0::enat) < n \Longrightarrow \infty * n = \infty$

by (*simp add*: *zero-enat-def less-enat-def split*: *enat.splits*)

lemma *imult-infinity-right*: $(0::enat) < n \Longrightarrow n * \infty = \infty$
by (*simp add*: *zero-enat-def less-enat-def split*: *enat.splits*)

lemma *enat-0-less-mult-iff*: $(0 < (m::enat) * n) = (0 < m \wedge 0 < n)$
by (*simp only*: *zero-less-iff-neq-zero mult-eq-0-iff*, *simp*)

lemma *mono-eSuc*: *mono eSuc*
by (*simp add*: *mono-def*)

lemma *min-enat-simps* [*simp*]:
min (*enat m*) (*enat n*) = *enat* (*min m n*)
min q 0 = *0*
min 0 q = *0*
min q $(\infty::enat)$ = *q*
min $(\infty::enat)$ *q* = *q*
by (*auto simp add*: *min-def*)

lemma *max-enat-simps* [*simp*]:
max (*enat m*) (*enat n*) = *enat* (*max m n*)
max q 0 = *q*
max 0 q = *q*
max q ∞ = $(\infty::enat)$
max ∞ *q* = $(\infty::enat)$
by (*simp-all add*: *max-def*)

lemma *enat-ile*: $n \leq enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
by (*cases n*) *simp-all*

lemma *enat-iless*: $n < enat\ m \Longrightarrow \exists\, k.\ n = enat\ k$
by (*cases n*) *simp-all*

lemma *iadd-le-enat-iff*:
$x + y \leq enat\ n \longleftrightarrow (\exists\, y'\ x'.\ x = enat\ x' \wedge y = enat\ y' \wedge x' + y' \leq n)$
by(*cases x y rule*: *enat.exhaust*[*case-product enat.exhaust*]) *simp-all*

lemma *chain-incr*: $\forall\, i.\ \exists\, j.\ Y\ i < Y\ j \Longrightarrow \exists\, j.\ enat\ k < Y\ j$
proof (*induction k*)
case *0*
then show *?case*
using *enat-0 zero-less-iff-neq-zero* **by** *fastforce*
next
case (*Suc k*)
then show *?case*
by (*meson Suc-ile-eq order-le-less-trans*)
qed

lemma *eSuc-max*: *eSuc* (*max x y*) = *max* (*eSuc x*) (*eSuc y*)

by (*simp add*: *eSuc-def split*: *enat.split*)

lemma *eSuc-Max*:
assumes *finite A* $A \neq \{\}$
shows *eSuc* (*Max A*) = *Max* (*eSuc* ' *A*)
by (*simp add*: *assms mono-Max-commute mono-eSuc*)

instantiation *enat* :: {*order-bot*, *order-top*}
begin

definition *bot-enat* :: *enat* **where** *bot-enat* = *0*
definition *top-enat* :: *enat* **where** *top-enat* = ∞

instance
by *standard* (*simp-all add*: *bot-enat-def top-enat-def*)

end

lemma *finite-enat-bounded*:
assumes *le-fin*: $\bigwedge y.\ y \in A \Longrightarrow y \leq enat\ n$
shows *finite A*
proof (*rule finite-subset*)
show *finite* (*enat* ' {..*n*}) **by** *blast*
have $A \subseteq enat\ `\ \{..n\}$
using *enat-ile le-fin* **by** *fastforce*
then show $A \subseteq enat\ `\ \{..n\}$ **.**
qed

36.8 Cancellation simprocs

lemma *add-diff-cancel-enat*[*simp*]: $x \neq \infty \Longrightarrow x + y - x = (y::enat)$
by (*metis add.commute add.right-neutral add-diff-assoc-enat idiff-self order-refl*)

lemma *enat-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::enat) \vee b = c$
unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

lemma *enat-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::enat) \vee b \leq c$
unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

lemma *enat-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::enat) \wedge b < c$
unfolding *plus-enat-def* **by** (*simp split*: *enat.split*)

lemma *plus-eq-infty-iff-enat*: $(m::enat) + n = \infty \longleftrightarrow m{=}\infty \vee n{=}\infty$
using *enat-add-left-cancel* **by** *fastforce*

ML ‹
structure Cancel-Enat-Common =
struct
(∗ *copied from src/HOL/Tools/nat-numeral-simprocs.ML* ∗)

```
  fun find-first-t -    - []          = raise TERM(find-first-t, [])
    | find-first-t past u (t::terms) =
        if u aconv t then (rev past @ terms)
        else find-first-t (t::past) u terms

  fun dest-summing (Const (const-name‹Groups.plus›, -) $ t $ u, ts) =
        dest-summing (t, dest-summing (u, ts))
    | dest-summing (t, ts) = t :: ts

  val mk-sum = Arith-Data.long-mk-sum
  fun dest-sum t = dest-summing (t, [])
  val find-first = find-first-t []
  val trans-tac = Numeral-Simprocs.trans-tac
  val norm-ss =
    HOL-basic-ss
    |> Simplifier.simpset-map context
      (Simplifier.add-simps @{thms ac-simps add-0-left add-0-right})
  fun norm-tac ctxt = ALLGOALS (simp-tac (put-simpset norm-ss ctxt))
  fun simplify-meta-eq ctxt cancel-th th =
    Arith-Data.simplify-meta-eq [] ctxt
      ([th, cancel-th] MRS trans)
  fun mk-eq (a, b) = HOLogic.mk-Trueprop (HOLogic.mk-eq (a, b))
end

structure Eq-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-eq
  val dest-bal = HOLogic.dest-bin const-name‹HOL.eq› typ‹enat›
  fun simp-conv - - = SOME @{thm enat-add-left-cancel}
)

structure Le-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name‹Orderings.less-eq›
  val dest-bal = HOLogic.dest-bin const-name‹Orderings.less-eq› typ‹enat›
  fun simp-conv - - = SOME @{thm enat-add-left-cancel-le}
)

structure Less-Enat-Cancel = ExtractCommonTermFun
(open Cancel-Enat-Common
  val mk-bal = HOLogic.mk-binrel const-name‹Orderings.less›
  val dest-bal = HOLogic.dest-bin const-name‹Orderings.less› typ‹enat›
  fun simp-conv - - = SOME @{thm enat-add-left-cancel-less}
)
›

simproc-setup enat-eq-cancel
  ((l::enat) + m = n | (l::enat) = m + n) =
  ‹K (fn ctxt => fn ct => Eq-Enat-Cancel.proc ctxt (Thm.term-of ct))›
```

simproc-setup *enat-le-cancel*
((*l*::*enat*) + *m* ≤ *n* | (*l*::*enat*) ≤ *m* + *n*) =
‹*K* (*fn ctxt* => *fn ct* => *Le-Enat-Cancel.proc ctxt* (*Thm.term-of ct*))›

simproc-setup *enat-less-cancel*
((*l*::*enat*) + *m* < *n* | (*l*::*enat*) < *m* + *n*) =
‹*K* (*fn ctxt* => *fn ct* => *Less-Enat-Cancel.proc ctxt* (*Thm.term-of ct*))›

TODO: add regression tests for these simprocs

TODO: add simprocs for combining and cancelling numerals

36.9 Well-ordering

lemma *less-enatE*:
⟦*n* < *enat m*; ⋀*k*. ⟦*n* = *enat k*; *k* < *m*⟧ ⟹ *P*⟧ ⟹ *P*
using *enat-iless enat-ord-simps*(*2*) **by** *blast*

lemma *less-infinityE*:
⟦*n* < ∞; ⋀*k*. *n* = *enat k* ⟹ *P*⟧ ⟹ *P*
by *auto*

lemma *enat-less-induct*:
assumes ⋀*n*. ∀ *m*::*enat*. *m* < *n* ⟶ *P m* ⟹ *P n*
shows *P n*
proof −
have *P* (*enat k*) **for** *k*
by (*induction k rule*: *less-induct*) (*metis less-enatE assms*)
then show *?thesis*
by (*metis enat.exhaust less-infinityE assms*)
qed

instance *enat* :: *wellorder*
proof
fix *P* **and** *n*
assume *hyp*: (⋀*n*::*enat*. (⋀*m*::*enat*. *m* < *n* ⟹ *P m*) ⟹ *P n*)
show *P n* **by** (*blast intro*: *enat-less-induct hyp*)
qed

36.10 Complete Lattice

instantiation *enat* :: *complete-lattice*
begin

definition *inf-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
inf-enat = *min*

definition *sup-enat* :: *enat* ⇒ *enat* ⇒ *enat* **where**
sup-enat = *max*

definition *Inf-enat* :: *enat set* ⇒ *enat* **where**
Inf-enat A = (*if A* = {} *then* ∞ *else* (*LEAST x*. *x* ∈ *A*))

definition *Sup-enat* :: *enat set* ⇒ *enat* **where**
Sup-enat A = (*if A* = {} *then 0 else if finite A then Max A else* ∞)

instance
proof
fix *x* :: *enat* **and** *A* :: *enat set*
show *x* ∈ *A* ⟹ *Inf A* ≤ *x*
unfolding *Inf-enat-def* **by** (*auto intro*: *Least-le*)
show (⋀*y*. *y* ∈ *A* ⟹ *x* ≤ *y*) ⟹ *x* ≤ *Inf A*
unfolding *Inf-enat-def*
by (*cases A* = {}) (*auto intro*: *LeastI2-ex*)
show *x* ∈ *A* ⟹ *x* ≤ *Sup A*
unfolding *Sup-enat-def* **by** (*cases finite A*) *auto*
show (⋀*y*. *y* ∈ *A* ⟹ *y* ≤ *x*) ⟹ *Sup A* ≤ *x*
unfolding *Sup-enat-def* **using** *finite-enat-bounded* **by** *auto*
qed (*simp-all add*: *inf-enat-def sup-enat-def bot-enat-def top-enat-def Inf-enat-def Sup-enat-def*)

end

instance *enat* :: *complete-linorder* **..**

lemma *eSuc-Sup*: *A* ≠ {} ⟹ *eSuc* (*Sup A*) = *Sup* (*eSuc* ‘ *A*)
by(*auto simp add*: *Sup-enat-def eSuc-Max inj-on-def dest*: *finite-imageD*)

lemma *sup-continuous-eSuc*: *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *eSuc* (*f x*))
using *eSuc-Sup* [*of* - ‘ *UNIV*] **by** (*auto simp*: *sup-continuous-def image-comp*)

36.11 Traditional theorem names

lemmas *enat-defs* = *zero-enat-def one-enat-def eSuc-def*
plus-enat-def less-eq-enat-def less-enat-def

lemma *iadd-is-0*: (*m* + *n* = (*0*::*enat*)) = (*m* = *0* ∧ *n* = *0*)
by (*rule add-eq-0-iff-both-eq-0*)

lemma *i0-lb* : (*0*::*enat*) ≤ *n*
by (*rule zero-le*)

lemma *ile0-eq*: *n* ≤ (*0*::*enat*) ⟷ *n* = *0*
by (*rule le-zero-eq*)

lemma *not-iless0*: ¬ *n* < (*0*::*enat*)
by (*rule not-less-zero*)

lemma *i0-less*[*simp*]: $(0::enat) < n \longleftrightarrow n \neq 0$
by (*rule zero-less-iff-neq-zero*)

lemma *imult-is-0*: $((m::enat) * n = 0) = (m = 0 \vee n = 0)$
by (*rule mult-eq-0-iff*)

end

37 Liminf and Limsup on conditionally complete lattices

theory *Liminf-Limsup*
imports *Complex-Main*
begin

lemma (**in** *conditionally-complete-linorder*) *le-cSup-iff*:
assumes $A \neq \{\}$ *bdd-above A*
shows $x \leq Sup\ A \longleftrightarrow (\forall y{<}x.\ \exists a{\in}A.\ y < a)$
proof *safe*
fix *y* **assume** $x \leq Sup\ A\ y < x$
then have $y < Sup\ A$ **by** *auto*
then show $\exists a{\in}A.\ y < a$
unfolding *less-cSup-iff*[*OF assms*] **.**
qed (*auto elim*!: *allE*[*of - Sup A*] *simp add*: *not-le*[*symmetric*] *cSup-upper assms*)

lemma (**in** *conditionally-complete-linorder*) *le-cSUP-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq Sup\ (f\ `\ A) \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y < f\ i)$
using *le-cSup-iff* [*of f ` A*] **by** *simp*

lemma *le-cSup-iff-less*:
fixes $x :: 'a :: \{conditionally\text{-}complete\text{-}linorder, dense\text{-}linorder\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-above* $(f`A) \Longrightarrow x \leq (SUP\ i{\in}A.\ f\ i) \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y \leq f\ i)$
by (*simp add*: *le-cSUP-iff*)
(*blast intro*: *less-imp-le less-trans less-le-trans dest*: *dense*)

lemma *le-Sup-iff-less*:
fixes $x :: 'a :: \{complete\text{-}linorder, dense\text{-}linorder\}$
shows $x \leq (SUP\ i{\in}A.\ f\ i) \longleftrightarrow (\forall y{<}x.\ \exists i{\in}A.\ y \leq f\ i)$ (**is** *?lhs = ?rhs*)
unfolding *le-SUP-iff*
by (*blast intro*: *less-imp-le less-trans less-le-trans dest*: *dense*)

lemma (**in** *conditionally-complete-linorder*) *cInf-le-iff*:
assumes $A \neq \{\}$ *bdd-below A*
shows $Inf\ A \leq x \longleftrightarrow (\forall y{>}x.\ \exists a{\in}A.\ y > a)$
proof *safe*
fix *y* **assume** $x \geq Inf\ A\ y > x$
then have $y > Inf\ A$ **by** *auto*

then show $\exists\, a{\in}A.\ y > a$
unfolding *cInf-less-iff*[*OF assms*] .
qed (*auto elim*!: *allE*[*of - Inf A*] *simp add*: *not-le*[*symmetric*] *cInf-lower assms*)

lemma (**in** *conditionally-complete-linorder*) *cINF-le-iff*:
$A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow$ *Inf* $(f\ `\ A) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ y > f\ i)$
using *cInf-le-iff* [*of f ` A*] **by** *simp*

lemma *cInf-le-iff-less*:
fixes $x :: 'a :: \{$*conditionally-complete-linorder*, *dense-linorder*$\}$
shows $A \neq \{\} \Longrightarrow$ *bdd-below* $(f`A) \Longrightarrow (INF\ i{\in}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
by (*simp add*: *cINF-le-iff*)
(*blast intro*: *less-imp-le less-trans le-less-trans dest*: *dense*)

lemma *Inf-le-iff-less*:
fixes $x :: 'a :: \{$*complete-linorder*, *dense-linorder*$\}$
shows $(INF\ i{\in}A.\ f\ i) \leq x \longleftrightarrow (\forall\, y{>}x.\ \exists\, i{\in}A.\ f\ i \leq y)$
unfolding *INF-le-iff*
by (*blast intro*: *less-imp-le less-trans le-less-trans dest*: *dense*)

lemma *SUP-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(SUP\ i \in A.\ SUP\ j \in B.\ f\ i\ j) = (SUP\ p \in A \times B.\ f\ (fst\ p)\ (snd\ p))$
by (*rule antisym*) (*auto intro*!: *SUP-least SUP-upper2*)

lemma *INF-pair*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(INF\ i \in A.\ INF\ j \in B.\ f\ i\ j) = (INF\ p \in A \times B.\ f\ (fst\ p)\ (snd\ p))$
by (*rule antisym*) (*auto intro*!: *INF-greatest INF-lower2*)

lemma *INF-Sigma*:
fixes $f :: - \Rightarrow - \Rightarrow -$ :: *complete-lattice*
shows $(INF\ i \in A.\ INF\ j \in B\ i.\ f\ i\ j) = (INF\ p \in Sigma\ A\ B.\ f\ (fst\ p)\ (snd\ p))$
by (*rule antisym*) (*auto intro*!: *INF-greatest INF-lower2*)

37.0.1 *Liminf* and *Limsup*

definition *Liminf* :: *'a filter* $\Rightarrow$ $('a \Rightarrow 'b) \Rightarrow$ *'b* :: *complete-lattice* **where**
Liminf F f $= (SUP\ P{\in}\{P.\ eventually\ P\ F\}.\ INF\ x{\in}\{x.\ P\ x\}.\ f\ x)$

definition *Limsup* :: *'a filter* $\Rightarrow$ $('a \Rightarrow 'b) \Rightarrow$ *'b* :: *complete-lattice* **where**
Limsup F f $= (INF\ P{\in}\{P.\ eventually\ P\ F\}.\ SUP\ x{\in}\{x.\ P\ x\}.\ f\ x)$

abbreviation *liminf* $\equiv$ *Liminf sequentially*

abbreviation *limsup* $\equiv$ *Limsup sequentially*

lemma *Liminf-eqI*:

$(\bigwedge P.\ eventually\ P\ F \Longrightarrow Inf\ (f\ `\ (Collect\ P)) \leq x) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.\ eventually\ P\ F \Longrightarrow Inf\ (f\ `\ (Collect\ P)) \leq y) \Longrightarrow x \leq y) \Longrightarrow Liminf\ F\ f = x$
unfolding *Liminf-def* **by** (*auto intro*!: *SUP-eqI*)

lemma *Limsup-eqI*:
$(\bigwedge P.\ eventually\ P\ F \Longrightarrow x \leq Sup\ (f\ `\ (Collect\ P))) \Longrightarrow$
$(\bigwedge y.\ (\bigwedge P.\ eventually\ P\ F \Longrightarrow y \leq Sup\ (f\ `\ (Collect\ P))) \Longrightarrow y \leq x) \Longrightarrow Limsup\ F\ f = x$
unfolding *Limsup-def* **by** (*auto intro*!: *INF-eqI*)

lemma *liminf-SUP-INF*: $liminf\ f = (SUP\ n.\ INF\ m{\in}\{n..\}.\ f\ m)$
unfolding *Liminf-def eventually-sequentially*
by (*rule SUP-eq*) (*auto simp*: *atLeast-def intro*!: *INF-mono*)

lemma *limsup-INF-SUP*: $limsup\ f = (INF\ n.\ SUP\ m{\in}\{n..\}.\ f\ m)$
unfolding *Limsup-def eventually-sequentially*
by (*rule INF-eq*) (*auto simp*: *atLeast-def intro*!: *SUP-mono*)

lemma *mem-limsup-iff*: $x \in limsup\ A \longleftrightarrow (\exists_F\ n\ in\ sequentially.\ x \in A\ n)$
by (*simp add*: *Limsup-def*) (*metis* (*mono-tags*) *eventually-mono not-frequently*)

lemma *mem-liminf-iff*: $x \in liminf\ A \longleftrightarrow (\forall_F\ n\ in\ sequentially.\ x \in A\ n)$
by (*simp add*: *Liminf-def*) (*metis* (*mono-tags*) *eventually-mono*)

lemma *Limsup-const*:
assumes *ntriv*: $\neg\ trivial\text{-}limit\ F$
shows $Limsup\ F\ (\lambda x.\ c) = c$
proof −
have ∗: $\bigwedge P.\ Ex\ P \longleftrightarrow P \neq (\lambda x.\ False)$ **by** *auto*
have $\bigwedge P.\ eventually\ P\ F \Longrightarrow (SUP\ x \in \{x.\ P\ x\}.\ c) = c$
using *ntriv* **by** (*intro SUP-const*) (*auto simp*: *eventually-False* ∗)
then show *?thesis*
apply (*auto simp add*: *Limsup-def*)
apply (*rule INF-const*)
apply *auto*
using *eventually-True* **apply** *blast*
done
qed

lemma *Liminf-const*:
assumes *ntriv*: $\neg\ trivial\text{-}limit\ F$
shows $Liminf\ F\ (\lambda x.\ c) = c$
proof −
have ∗: $\bigwedge P.\ Ex\ P \longleftrightarrow P \neq (\lambda x.\ False)$ **by** *auto*
have $\bigwedge P.\ eventually\ P\ F \Longrightarrow (INF\ x \in \{x.\ P\ x\}.\ c) = c$
using *ntriv* **by** (*intro INF-const*) (*auto simp*: *eventually-False* ∗)
then show *?thesis*
apply (*auto simp add*: *Liminf-def*)

apply (*rule SUP-const*)
apply *auto*
using *eventually-True* **apply** *blast*
done
qed

lemma *Liminf-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Liminf F f* ≤ *Liminf F g*
unfolding *Liminf-def*
proof (*safe intro*!: *SUP-mono*)
fix *P* **assume** *eventually P F*
with *ev* **have** *eventually* (λ*x*. *f x* ≤ *g x* ∧ *P x*) *F* (**is** *eventually ?Q F*) **by** (*rule eventually-conj*)
then show ∃ *Q*∈{*P*. *eventually P F*}. *Inf* (*f* ‘ (*Collect P*)) ≤ *Inf* (*g* ‘ (*Collect Q*))
by (*intro bexI*[*of* - *?Q*]) (*auto intro*!: *INF-mono*)
qed

lemma *Liminf-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *F*
shows *Liminf F f* = *Liminf F g*
by (*intro antisym Liminf-mono eventually-mono*[*OF assms*]) *auto*

lemma *Limsup-mono*:
assumes *ev*: *eventually* (λ*x*. *f x* ≤ *g x*) *F*
shows *Limsup F f* ≤ *Limsup F g*
unfolding *Limsup-def*
proof (*safe intro*!: *INF-mono*)
fix *P* **assume** *eventually P F*
with *ev* **have** *eventually* (λ*x*. *f x* ≤ *g x* ∧ *P x*) *F* (**is** *eventually ?Q F*) **by** (*rule eventually-conj*)
then show ∃ *Q*∈{*P*. *eventually P F*}. *Sup* (*f* ‘ (*Collect Q*)) ≤ *Sup* (*g* ‘ (*Collect P*))
by (*intro bexI*[*of* - *?Q*]) (*auto intro*!: *SUP-mono*)
qed

lemma *Limsup-eq*:
assumes *eventually* (λ*x*. *f x* = *g x*) *net*
shows *Limsup net f* = *Limsup net g*
by (*intro antisym Limsup-mono eventually-mono*[*OF assms*]) *auto*

lemma *Liminf-bot*[*simp*]: *Liminf bot f* = *top*
unfolding *Liminf-def top-unique*[*symmetric*]
by (*rule SUP-upper2*[**where** *i*=λ*x*. *False*]) *simp-all*

lemma *Limsup-bot*[*simp*]: *Limsup bot f* = *bot*
unfolding *Limsup-def bot-unique*[*symmetric*]
by (*rule INF-lower2*[**where** *i*=λ*x*. *False*]) *simp-all*

lemma *Liminf-le-Limsup*:
assumes *ntriv*: ¬ *trivial-limit F*
shows *Liminf F f* ≤ *Limsup F f*
unfolding *Limsup-def Liminf-def*
apply (*rule SUP-least*)
apply (*rule INF-greatest*)
proof *safe*
fix *P Q* **assume** *eventually P F eventually Q F*
then have *eventually* (λ*x*. *P x* ∧ *Q x*) *F* (**is** *eventually ?C F*) **by** (*rule eventually-conj*)
then have *not-False*: (λ*x*. *P x* ∧ *Q x*) ≠ (λ*x*. *False*)
using *ntriv* **by** (*auto simp add*: *eventually-False*)
have *Inf* (*f* ‘ (*Collect P*)) ≤ *Inf* (*f* ‘ (*Collect ?C*))
by (*rule INF-mono*) *auto*
also have ... ≤ *Sup* (*f* ‘ (*Collect ?C*))
using *not-False* **by** (*intro INF-le-SUP*) *auto*
also have ... ≤ *Sup* (*f* ‘ (*Collect Q*))
by (*rule SUP-mono*) *auto*
finally show *Inf* (*f* ‘ (*Collect P*)) ≤ *Sup* (*f* ‘ (*Collect Q*)) **.**
qed

lemma *Liminf-bounded*:
assumes *le*: *eventually* (λ*n*. *C* ≤ *X n*) *F*
shows *C* ≤ *Liminf F X*
using *Liminf-mono*[*OF le*] *Liminf-const*[*of F C*]
by (*cases F* = *bot*) *simp-all*

lemma *Limsup-bounded*:
assumes *le*: *eventually* (λ*n*. *X n* ≤ *C*) *F*
shows *Limsup F X* ≤ *C*
using *Limsup-mono*[*OF le*] *Limsup-const*[*of F C*]
by (*cases F* = *bot*) *simp-all*

lemma *le-Limsup*:
assumes *F*: *F* ≠ *bot* **and** *x*: ∀ *F x in F*. *l* ≤ *f x*
shows *l* ≤ *Limsup F f*
using *F Liminf-bounded*[*of l f F*] *Liminf-le-Limsup*[*of F f*] *order.trans x* **by** *blast*

lemma *Liminf-le*:
assumes *F*: *F* ≠ *bot* **and** *x*: ∀ *F x in F*. *f x* ≤ *l*
shows *Liminf F f* ≤ *l*
using *F Liminf-le-Limsup Limsup-bounded order.trans x* **by** *blast*

lemma *le-Liminf-iff*:
fixes *X* :: - ⇒ - :: *complete-linorder*
shows *C* ≤ *Liminf F X* ⟷ (∀ *y*<*C*. *eventually* (λ*x*. *y* < *X x*) *F*)
proof −
have *eventually* (λ*x*. *y* < *X x*) *F*

```
    if eventually P F y < Inf (X ‘ (Collect P)) for y P
    using that by (auto elim!: eventually-mono dest: less-INF-D)
  moreover
  have ∃ P. eventually P F ∧ y < Inf (X ‘ (Collect P))
    if y < C and y: ∀ y<C. eventually (λx. y < X x) F for y P
  proof (cases ∃ z. y < z ∧ z < C)
    case True
    then obtain z where z: y < z ∧ z < C ..
    moreover from z have z ≤ Inf (X ‘ {x. z < X x})
      by (auto intro!: INF-greatest)
    ultimately show ?thesis
      using y by (intro exI[of - λx. z < X x]) auto
  next
    case False
    then have C ≤ Inf (X ‘ {x. y < X x})
      by (intro INF-greatest) auto
    with ‹y < C› show ?thesis
      using y by (intro exI[of - λx. y < X x]) auto
  qed
  ultimately show ?thesis
    unfolding Liminf-def le-SUP-iff by auto
qed

lemma Limsup-le-iff:
  fixes X :: - ⇒ - :: complete-linorder
  shows C ≥ Limsup F X ⟷ (∀ y>C. eventually (λx. y > X x) F)
proof −
  { fix y P assume eventually P F y > Sup (X ‘ (Collect P))
    then have eventually (λx. y > X x) F
      by (auto elim!: eventually-mono dest: SUP-lessD) }
  moreover
  { fix y P assume y > C and y: ∀ y>C. eventually (λx. y > X x) F
    have ∃ P. eventually P F ∧ y > Sup (X ‘ (Collect P))
    proof (cases ∃ z. C < z ∧ z < y)
      case True
      then obtain z where z: C < z ∧ z < y ..
      moreover from z have z ≥ Sup (X ‘ {x. X x < z})
        by (auto intro!: SUP-least)
      ultimately show ?thesis
        using y by (intro exI[of - λx. z > X x]) auto
    next
      case False
      then have C ≥ Sup (X ‘ {x. X x < y})
        by (intro SUP-least) (auto simp: not-less)
      with ‹y > C› show ?thesis
        using y by (intro exI[of - λx. y > X x]) auto
    qed }
  ultimately show ?thesis
    unfolding Limsup-def INF-le-iff by auto
```

qed

lemma *less-LiminfD*:
y < *Liminf F* (*f* :: - ⇒ *'a* :: *complete-linorder*) ⟹ *eventually* (λ*x*. *f x* > *y*) *F*
using *le-Liminf-iff*[*of Liminf F f F f*] **by** *simp*

lemma *Limsup-lessD*:
y > *Limsup F* (*f* :: - ⇒ *'a* :: *complete-linorder*) ⟹ *eventually* (λ*x*. *f x* < *y*) *F*
using *Limsup-le-iff*[*of F f Limsup F f*] **by** *simp*

lemma *lim-imp-Liminf*:
fixes *f* :: *'a* ⇒ - :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: ¬ *trivial-limit F*
assumes *lim*: (*f* ⟶ *f0*) *F*
shows *Liminf F f* = *f0*
proof (*intro Liminf-eqI*)
fix *P* **assume** *P*: *eventually P F*
then have *eventually* (λ*x*. *Inf* (*f* ‘ (*Collect P*)) ≤ *f x*) *F*
by *eventually-elim* (*auto intro*!: *INF-lower*)
then show *Inf* (*f* ‘ (*Collect P*)) ≤ *f0*
by (*rule tendsto-le*[*OF ntriv lim tendsto-const*])
next
fix *y* **assume** *upper*: ⋀*P*. *eventually P F* ⟹ *Inf* (*f* ‘ (*Collect P*)) ≤ *y*
show *f0* ≤ *y*
proof *cases*
assume ∃ *z*. *y* < *z* ∧ *z* < *f0*
then obtain *z* **where** *y* < *z* ∧ *z* < *f0* **..**
moreover have *z* ≤ *Inf* (*f* ‘ {*x*. *z* < *f x*})
by (*rule INF-greatest*) *simp*
ultimately show *?thesis*
using *lim*[*THEN topological-tendstoD*, *THEN upper*, *of* {*z* <..}] **by** *auto*
next
assume *discrete*: ¬ (∃ *z*. *y* < *z* ∧ *z* < *f0*)
show *?thesis*
proof (*rule classical*)
assume ¬ *f0* ≤ *y*
then have *eventually* (λ*x*. *y* < *f x*) *F*
using *lim*[*THEN topological-tendstoD*, *of* {*y* <..}] **by** *auto*
then have *eventually* (λ*x*. *f0* ≤ *f x*) *F*
using *discrete* **by** (*auto elim*!: *eventually-mono*)
then have *Inf* (*f* ‘ {*x*. *f0* ≤ *f x*}) ≤ *y*
by (*rule upper*)
moreover have *f0* ≤ *Inf* (*f* ‘ {*x*. *f0* ≤ *f x*})
by (*intro INF-greatest*) *simp*
ultimately show *f0* ≤ *y* **by** *simp*
qed
qed
qed

lemma *lim-imp-Limsup*:
fixes $f :: 'a \Rightarrow -$ :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: $\neg$ *trivial-limit F*
assumes *lim*: $(f \longrightarrow f0)\ F$
shows *Limsup F f* = *f0*
proof (*intro Limsup-eqI*)
fix *P* **assume** *P*: *eventually P F*
then have *eventually* $(\lambda x.\ f\ x \leq Sup\ (f\ `\ (Collect\ P)))\ F$
by *eventually-elim* (*auto intro*!: *SUP-upper*)
then show *f0* $\leq$ *Sup* (*f* ` (*Collect P*))
by (*rule tendsto-le*[*OF ntriv tendsto-const lim*])
next
fix *y* **assume** *lower*: $\bigwedge P.$ *eventually P F* $\Longrightarrow$ *y* $\leq$ *Sup* (*f* ` (*Collect P*))
show *y* $\leq$ *f0*
proof (*cases* $\exists\, z.$ *f0* $<$ *z* $\wedge$ *z* $<$ *y*)
case *True*
then obtain *z* **where** *f0* $<$ *z* $\wedge$ *z* $<$ *y* **..**
moreover have *Sup* (*f* ` {*x*. *f x* $<$ *z*}) $\leq$ *z*
by (*rule SUP-least*) *simp*
ultimately show *?thesis*
using *lim*[*THEN topological-tendstoD*, *THEN lower*, *of* {..$<$ *z*}] **by** *auto*
next
case *False*
show *?thesis*
proof (*rule classical*)
assume $\neg$ *y* $\leq$ *f0*
then have *eventually* $(\lambda x.\ f\ x < y)\ F$
using *lim*[*THEN topological-tendstoD*, *of* {..$<$ *y*}] **by** *auto*
then have *eventually* $(\lambda x.\ f\ x \leq f0)\ F$
using *False* **by** (*auto elim*!: *eventually-mono simp*: *not-less*)
then have *y* $\leq$ *Sup* (*f* ` {*x*. *f x* $\leq$ *f0*})
by (*rule lower*)
moreover have *Sup* (*f* ` {*x*. *f x* $\leq$ *f0*}) $\leq$ *f0*
by (*intro SUP-least*) *simp*
ultimately show *y* $\leq$ *f0* **by** *simp*
qed
qed
qed

lemma *Liminf-eq-Limsup*:
fixes *f0* :: *'a* :: {*complete-linorder*,*linorder-topology*}
assumes *ntriv*: $\neg$ *trivial-limit F*
and *lim*: *Liminf F f* = *f0 Limsup F f* = *f0*
shows $(f \longrightarrow f0)\ F$
proof (*rule order-tendstoI*)
fix *a* **assume** *f0* $<$ *a*
with *assms* **have** *Limsup F f* $<$ *a* **by** *simp*
then obtain *P* **where** *eventually P F Sup* (*f* ` (*Collect P*)) $<$ *a*
unfolding *Limsup-def INF-less-iff* **by** *auto*

then show *eventually* $(\lambda x.\ f\ x < a)\ F$
by (*auto elim*!: *eventually-mono dest*: *SUP-lessD*)
next
fix a **assume** $a < f0$
with *assms* **have** $a <$ *Liminf* $F\ f$ **by** *simp*
then obtain P **where** *eventually* $P\ F\ a <$ *Inf* $(f\ `\ (Collect\ P))$
unfolding *Liminf-def less-SUP-iff* **by** *auto*
then show *eventually* $(\lambda x.\ a < f\ x)\ F$
by (*auto elim*!: *eventually-mono dest*: *less-INF-D*)
qed

lemma *tendsto-iff-Liminf-eq-Limsup*:
fixes *f0* :: $'a$ :: {*complete-linorder*,*linorder-topology*}
shows ¬ *trivial-limit* $F \Longrightarrow (f \longrightarrow f0)\ F \longleftrightarrow$ (*Liminf* $F\ f = f0 \wedge$ *Limsup* $F\ f = f0$)
by (*metis Liminf-eq-Limsup lim-imp-Limsup lim-imp-Liminf*)

lemma *liminf-subseq-mono*:
fixes X :: *nat* ⇒ $'a$ :: *complete-linorder*
assumes *strict-mono* r
shows *liminf* $X \leq$ *liminf* $(X \circ r)$
proof−
have $\bigwedge n.$ $(INF\ m{\in}\{n..\}.\ X\ m) \leq (INF\ m{\in}\{n..\}.\ (X \circ r)\ m)$
proof (*safe intro*!: *INF-mono*)
fix $n\ m$ :: *nat* **assume** $n \leq m$ **then show** $\exists\, ma{\in}\{n..\}.\ X\ ma \leq (X \circ r)\ m$
using *seq-suble*[*OF* ‹*strict-mono r*›, *of m*] **by** (*intro bexI*[*of - r m*]) *auto*
qed
then show *?thesis* **by** (*auto intro*!: *SUP-mono simp*: *liminf-SUP-INF comp-def*)
qed

lemma *limsup-subseq-mono*:
fixes X :: *nat* ⇒ $'a$ :: *complete-linorder*
assumes *strict-mono* r
shows *limsup* $(X \circ r) \leq$ *limsup* X
proof−
have $(SUP\ m{\in}\{n..\}.\ (X \circ r)\ m) \leq (SUP\ m{\in}\{n..\}.\ X\ m)$ **for** n
proof (*safe intro*!: *SUP-mono*)
fix m :: *nat*
assume $n \leq m$
then show $\exists\, ma{\in}\{n..\}.\ (X \circ r)\ m \leq X\ ma$
using *seq-suble*[*OF* ‹*strict-mono r*›, *of m*] **by** (*intro bexI*[*of - r m*]) *auto*
qed
then show *?thesis*
by (*auto intro*!: *INF-mono simp*: *limsup-INF-SUP comp-def*)
qed

lemma *continuous-on-imp-continuous-within*:
continuous-on $s\ f \Longrightarrow t \subseteq s \Longrightarrow x \in s \Longrightarrow$ *continuous* $(at\ x\ within\ t)\ f$
unfolding *continuous-on-eq-continuous-within*

by (*auto simp*: *continuous-within intro*: *tendsto-within-subset*)

lemma *Liminf-compose-continuous-mono*:
fixes f :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: $F \neq bot$
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
proof −
have ∗: ∃ *x*. *P x* **if** *eventually P F* **for** *P*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed
have *f* (*SUP P*∈{*P*. *eventually P F*}. *Inf* (*g* ‘ *Collect P*)) =
Sup (*f* ‘ (λ*P*. *Inf* (*g* ‘ *Collect P*)) ‘ {*P*. *eventually P F*})
using *am continuous-on-imp-continuous-within* [*OF c*]
by (*rule continuous-at-Sup-mono*) (*auto intro*: *eventually-True*)
then have *f* (*Liminf F g*) = (*SUP P* ∈ {*P*. *eventually P F*}. *f* (*Inf* (*g* ‘ *Collect P*)))
by (*simp add*: *Liminf-def image-comp*)
also have ... = (*SUP P* ∈ {*P*. *eventually P F*}. *Inf* (*f* ‘ (*g* ‘ *Collect P*)))
using ∗ *continuous-at-Inf-mono* [*OF am continuous-on-imp-continuous-within* [*OF c*]]
by *auto*
finally show *?thesis* **by** (*auto simp*: *Liminf-def image-comp*)
qed

lemma *Limsup-compose-continuous-mono*:
fixes f :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *mono f* **and** *F*: $F \neq bot$
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
proof −
have ∗: ∃ *x*. *P x* **if** *eventually P F* **for** *P*
proof (*rule ccontr*)
assume ¬ *?thesis*
then have *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed
have *f* (*INF P*∈{*P*. *eventually P F*}. *Sup* (*g* ‘ *Collect P*)) =
Inf (*f* ‘ (λ*P*. *Sup* (*g* ‘ *Collect P*)) ‘ {*P*. *eventually P F*})
using *am continuous-on-imp-continuous-within* [*OF c*]
by (*rule continuous-at-Inf-mono*) (*auto intro*: *eventually-True*)
then have *f* (*Limsup F g*) = (*INF P* ∈ {*P*. *eventually P F*}. *f* (*Sup* (*g* ‘ *Collect P*)))
by (*simp add*: *Limsup-def image-comp*)

also have ... = (*INF P* ∈ {*P*. *eventually P F*}. *Sup* (*f* ' (*g* ' *Collect P*)))
using ∗ *continuous-at-Sup-mono* [*OF am continuous-on-imp-continuous-within* [*OF c*]]
by *auto*
finally show *?thesis* **by** (*auto simp*: *Limsup-def image-comp*)
qed

lemma *Liminf-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*,*linorder-topology*} ⇒ *'b*::{*complete-linorder*,*linorder-topology*}
assumes *c*: *continuous-on UNIV f*
and *am*: *antimono f*
and *F*: *F* ≠ *bot*
shows *Liminf F* (λ*n*. *f* (*g n*)) = *f* (*Limsup F g*)
proof −
have ∗: ∃ *x*. *P x* **if** *eventually P F* **for** *P*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed

have *f* (*INF P*∈{*P*. *eventually P F*}. *Sup* (*g* ' *Collect P*)) =
Sup (*f* ' (λ*P*. *Sup* (*g* ' *Collect P*)) ' {*P*. *eventually P F*})
using *am continuous-on-imp-continuous-within* [*OF c*]
by (*rule continuous-at-Inf-antimono*) (*auto intro*: *eventually-True*)
then have *f* (*Limsup F g*) = (*SUP P* ∈ {*P*. *eventually P F*}. *f* (*Sup* (*g* ' *Collect P*)))
by (*simp add*: *Limsup-def image-comp*)
also have ... = (*SUP P* ∈ {*P*. *eventually P F*}. *Inf* (*f* ' (*g* ' *Collect P*)))
using ∗ *continuous-at-Sup-antimono* [*OF am continuous-on-imp-continuous-within* [*OF c*]]
by *auto*
finally show *?thesis*
by (*auto simp*: *Liminf-def image-comp*)
qed

lemma *Limsup-compose-continuous-antimono*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *c*: *continuous-on UNIV f* **and** *am*: *antimono f* **and** *F*: *F* ≠ *bot*
shows *Limsup F* (λ*n*. *f* (*g n*)) = *f* (*Liminf F g*)
proof −
have ∗: ∃ *x*. *P x* **if** *eventually P F* **for** *P*
proof (*rule ccontr*)
assume ¬ (∃ *x*. *P x*) **then have** *P* = (λ*x*. *False*)
by *auto*
with ‹*eventually P F*› *F* **show** *False*
by *auto*
qed

have *f* (*SUP P∈{P. eventually P F}. Inf* (*g ‘ Collect P*)) =
Inf (*f ‘* (*λP. Inf* (*g ‘ Collect P*)) *‘ {P. eventually P F}*)
using *am continuous-on-imp-continuous-within* [*OF c*]
by (*rule continuous-at-Sup-antimono*) (*auto intro*: *eventually-True*)
then have *f* (*Liminf F g*) = (*INF P ∈ {P. eventually P F}. f* (*Inf* (*g ‘ Collect P*)))
by (*simp add*: *Liminf-def image-comp*)
also have *...* = (*INF P ∈ {P. eventually P F}. Sup* (*f ‘* (*g ‘ Collect P*)))
using ∗ *continuous-at-Inf-antimono* [*OF am continuous-on-imp-continuous-within* [*OF c*]]
by *auto*
finally show *?thesis*
by (*auto simp*: *Limsup-def image-comp*)
qed

lemma *Liminf-filtermap-le*: *Liminf* (*filtermap f F*) *g* ≤ *Liminf F* (*λx. g* (*f x*))
apply (*cases F* = *bot*, *simp*)
by (*subst Liminf-def*)
(*auto simp add*: *INF-lower Liminf-bounded eventually-filtermap eventually-mono intro*!: *SUP-least*)

lemma *Limsup-filtermap-ge*: *Limsup* (*filtermap f F*) *g* ≥ *Limsup F* (*λx. g* (*f x*))
apply (*cases F* = *bot*, *simp*)
by (*subst Limsup-def*)
(*auto simp add*: *SUP-upper Limsup-bounded eventually-filtermap eventually-mono intro*!: *INF-greatest*)

lemma *Liminf-least*: (*⋀P. eventually P F ⟹* (*INF x∈Collect P. f x*) ≤ *x*) *⟹ Liminf F f* ≤ *x*
by (*auto intro*!: *SUP-least simp*: *Liminf-def*)

lemma *Limsup-greatest*: (*⋀P. eventually P F ⟹ x* ≤ (*SUP x∈Collect P. f x*)) *⟹ Limsup F f* ≥ *x*
by (*auto intro*!: *INF-greatest simp*: *Limsup-def*)

lemma *Liminf-filtermap-ge*: *inj f ⟹ Liminf* (*filtermap f F*) *g* ≥ *Liminf F* (*λx. g* (*f x*))
apply (*cases F* = *bot*, *simp*)
apply (*rule Liminf-least*)
subgoal for *P*
by (*auto simp*: *eventually-filtermap the-inv-f-f*
intro!: *Liminf-bounded INF-lower2 eventually-mono*[*of P*])
done

lemma *Limsup-filtermap-le*: *inj f ⟹ Limsup* (*filtermap f F*) *g* ≤ *Limsup F* (*λx. g* (*f x*))
apply (*cases F* = *bot*, *simp*)
apply (*rule Limsup-greatest*)
subgoal for *P*

by (*auto simp*: *eventually-filtermap the-inv-f-f*
intro!: *Limsup-bounded SUP-upper2 eventually-mono*[*of P*])
done

lemma *Liminf-filtermap-eq*: *inj f* ⟹ *Liminf* (*filtermap f F*) *g* = *Liminf F* (λ*x*. *g* (*f x*))
using *Liminf-filtermap-le*[*of f F g*] *Liminf-filtermap-ge*[*of f F g*]
by *simp*

lemma *Limsup-filtermap-eq*: *inj f* ⟹ *Limsup* (*filtermap f F*) *g* = *Limsup F* (λ*x*. *g* (*f x*))
using *Limsup-filtermap-le*[*of f F g*] *Limsup-filtermap-ge*[*of F g f*]
by *simp*

37.1 More Limits

lemma *convergent-limsup-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *limsup X* = *lim X*
by (*auto simp*: *convergent-def limI lim-imp-Limsup*)

lemma *convergent-liminf-cl*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows *convergent X* ⟹ *liminf X* = *lim X*
by (*auto simp*: *convergent-def limI lim-imp-Liminf*)

lemma *lim-increasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≥ *f m*
obtains *l* **where** *f* ⟶ (*l*::*'a*::{*complete-linorder*,*linorder-topology*})
proof
show *f* ⟶ (*SUP n*. *f n*)
using *assms*
by (*intro increasing-tendsto*)
(*auto simp*: *SUP-upper eventually-sequentially less-SUP-iff intro*: *less-le-trans*)
qed

lemma *lim-decreasing-cl*:
assumes ⋀*n m*. *n* ≥ *m* ⟹ *f n* ≤ *f m*
obtains *l* **where** *f* ⟶ (*l*::*'a*::{*complete-linorder*,*linorder-topology*})
proof
show *f* ⟶ (*INF n*. *f n*)
using *assms*
by (*intro decreasing-tendsto*)
(*auto simp*: *INF-lower eventually-sequentially INF-less-iff intro*: *le-less-trans*)
qed

lemma *compact-complete-linorder*:
fixes *X* :: *nat* ⇒ *'a*::{*complete-linorder*,*linorder-topology*}
shows ∃ *l r*. *strict-mono r* ∧ (*X* ∘ *r*) ⟶ *l*

proof −
obtain r **where** *strict-mono* r **and** *mono*: *monoseq* $(X \circ r)$
using *seq-monosub*[*of* X]
unfolding *comp-def*
by *auto*
then have $(\forall n\ m.\ m \leq n \longrightarrow (X \circ r)\ m \leq (X \circ r)\ n) \vee (\forall n\ m.\ m \leq n \longrightarrow (X \circ r)\ n \leq (X \circ r)\ m)$
by (*auto simp add*: *monoseq-def*)
then obtain l **where** $(X \circ r) \longrightarrow l$
using *lim-increasing-cl*[*of* $X \circ r$] *lim-decreasing-cl*[*of* $X \circ r$]
by *auto*
then show *?thesis*
using ‹*strict-mono* r› **by** *auto*
qed

lemma *tendsto-Limsup*:
fixes $f :: - \Rightarrow 'a :: \{$*complete-linorder*,*linorder-topology*$\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Limsup\ F\ f)\ F$
by (*subst tendsto-iff-Liminf-eq-Limsup*) *auto*

lemma *tendsto-Liminf*:
fixes $f :: - \Rightarrow 'a :: \{$*complete-linorder*,*linorder-topology*$\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ f = Liminf\ F\ f \Longrightarrow (f \longrightarrow Liminf\ F\ f)\ F$
by (*subst tendsto-iff-Liminf-eq-Limsup*) *auto*

end

38 Extended real number line

theory *Extended-Real*
imports *Complex-Main Extended-Nat Liminf-Limsup*
begin

This should be part of *HOL−Library.Extended-Nat* or *HOL−Library.Order-Continuity*, but then the AFP-entry *Jinja-Thread* fails, as it does overload certain named from *Complex-Main*.

lemma *incseq-sumI2*:
fixes $f :: 'i \Rightarrow nat \Rightarrow 'a$::*ordered-comm-monoid-add*
shows $(\bigwedge n.\ n \in A \Longrightarrow mono\ (f\ n)) \Longrightarrow mono\ (\lambda i.\ \sum n{\in}A.\ f\ n\ i)$
unfolding *incseq-def* **by** (*auto intro*: *sum-mono*)

lemma *incseq-sumI*:
fixes $f :: nat \Rightarrow 'a$::*ordered-comm-monoid-add*
assumes $\bigwedge i.\ 0 \leq f\ i$
shows *incseq* $(\lambda i.\ sum\ f\ \{..< i\})$
proof (*intro incseq-SucI*)
fix n
have $sum\ f\ \{..< n\} + 0 \leq sum\ f\ \{..<n\} + f\ n$
using *assms* **by** (*rule add-left-mono*)

then show *sum f* {..< *n*} ≤ *sum f* {..< *Suc n*}
by *auto*
qed

lemma *continuous-at-left-imp-sup-continuous*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *mono f* ⋀*x*. *continuous* (*at-left x*) *f*
shows *sup-continuous f*
unfolding *sup-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'a* **assume** *incseq M* **then show** *f* (*SUP i*. *M i*) = (*SUP i*. *f* (*M i*))
using *continuous-at-Sup-mono* [*OF assms*, *of range M*] **by** (*simp add*: *image-comp*)
qed

lemma *sup-continuous-at-left*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
assumes *f*: *sup-continuous f*
shows *continuous* (*at-left x*) *f*
proof *cases*
assume *x* = *bot* **then show** *?thesis*
by (*simp add*: *trivial-limit-at-left-bot*)
next
assume *x*: *x* ≠ *bot*
show *?thesis*
unfolding *continuous-within*
proof (*intro tendsto-at-left-sequentially*[*of bot*])
fix *S* :: *nat* ⇒ *'a* **assume** *S*: *incseq S* **and** *S-x*: *S* ⟶ *x*
from *S-x* **have** *x-eq*: *x* = (*SUP i*. *S i*)
by (*rule LIMSEQ-unique*) (*intro LIMSEQ-SUP S*)
show (λ*n*. *f* (*S n*)) ⟶ *f x*
unfolding *x-eq sup-continuousD*[*OF f S*]
using *S sup-continuous-mono*[*OF f*] **by** (*intro LIMSEQ-SUP*) (*auto simp*: *mono-def*)
qed (*insert x*, *auto simp*: *bot-less*)
qed

lemma *sup-continuous-iff-at-left*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
shows *sup-continuous f* ⟷ (∀ *x*. *continuous* (*at-left x*) *f*) ∧ *mono f*
using *continuous-at-left-imp-sup-continuous sup-continuous-at-left sup-continuous-mono*
by *blast*

lemma *continuous-at-right-imp-inf-continuous*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*} ⇒ *'b*::{*complete-linorder*, *linorder-topology*}
assumes *mono f* ⋀*x*. *continuous* (*at-right x*) *f*

shows *inf-continuous f*
unfolding *inf-continuous-def*
proof *safe*
fix *M* :: *nat* ⇒ *'a*
assume *decseq M*
then show *f* (*INF i. M i*) = (*INF i. f* (*M i*))
using *continuous-at-Inf-mono* [*OF assms*, *of range M*]
by (*simp add*: *image-comp*)
qed

lemma *inf-continuous-at-right*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
assumes *f*: *inf-continuous f*
shows *continuous* (*at-right x*) *f*
proof *cases*
assume *x* = *top* **then show** *?thesis*
by (*simp add*: *trivial-limit-at-right-top*)
next
assume *x*: *x* ≠ *top*
show *?thesis*
unfolding *continuous-within*
proof (*intro tendsto-at-right-sequentially*[*of - top*])
fix *S* :: *nat* ⇒ *'a*
assume *S*: *decseq S* **and** *S-x*: *S* ⟶ *x*
then have *x-eq*: *x* = (*INF i. S i*)
using *INF-Lim* **by** *blast*
show (*λn. f* (*S n*)) ⟶ *f x*
unfolding *x-eq inf-continuousD*[*OF f S*]
using *S inf-continuous-mono*[*OF f*] **by** (*intro LIMSEQ-INF*) (*auto simp*:
mono-def antimono-def)
qed (*insert x*, *auto simp*: *less-top*)
qed

lemma *inf-continuous-iff-at-right*:
fixes *f* :: *'a*::{*complete-linorder*, *linorder-topology*, *first-countable-topology*} ⇒
'b::{*complete-linorder*, *linorder-topology*}
shows *inf-continuous f* ⟷ (∀ *x. continuous* (*at-right x*) *f*) ∧ *mono f*
using *continuous-at-right-imp-inf-continuous inf-continuous-at-right inf-continuous-mono*
by *blast*

instantiation *enat* :: *linorder-topology*
begin

definition *open-enat* :: *enat set* ⇒ *bool* **where**
open-enat = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
proof qed (*rule open-enat-def*)

end

lemma *open-enat*: *open* {*enat n*}
proof (*cases n*)
  case *0*
  then have {*enat n*} = {..< *eSuc 0*}
    by (*auto simp*: *enat-0*)
  then show *?thesis*
    by *simp*
next
  case (*Suc n'*)
  then have {*enat n*} = {*enat n'* <..< *enat* (*Suc n*)}
    using *enat-iless* **by** (*fastforce simp*: *set-eq-iff*)
  then show *?thesis*
    by *simp*
qed

lemma *open-enat-iff*:
  fixes *A* :: *enat set*
  shows *open A* ⟷ (∞ ∈ *A* ⟶ (∃ *n*::*nat*. {*n* <..} ⊆ *A*))
proof *safe*
  assume ∞ ∉ *A*
  then have *A* = (⋃ *n*∈{*n*. *enat n* ∈ *A*}. {*enat n*})
    by (*simp add*: *set-eq-iff*) (*metis not-enat-eq*)
  moreover have *open* ...
    by (*auto intro*: *open-enat*)
  ultimately show *open A*
    by *simp*
next
  fix *n* **assume** {*enat n* <..} ⊆ *A*
  then have *A* = (⋃ *n*∈{*n*. *enat n* ∈ *A*}. {*enat n*}) ∪ {*enat n* <..}
    using *enat-ile leI* **by** (*simp add*: *set-eq-iff*) *blast*
  moreover have *open* ...
    by (*intro open-Un open-UN ballI open-enat open-greaterThan*)
  ultimately show *open A*
    by *simp*
next
  assume *open A* ∞ ∈ *A*
  then have *generate-topology* (*range lessThan* ∪ *range greaterThan*) *A* ∞ ∈ *A*
    unfolding *open-enat-def* **by** *auto*
  then show ∃ *n*::*nat*. {*n* <..} ⊆ *A*
  proof *induction*
    case (*Int A B*)
    then obtain *n m* **where** {*enat n*<..} ⊆ *A* {*enat m*<..} ⊆ *B*
      by *auto*
    then have {*enat* (*max n m*) <..} ⊆ *A* ∩ *B*
      by (*auto simp*: *subset-eq Ball-def max-def simp flip*: *enat-ord-code*(*1*))
    then show *?case*

```
    by auto
  next
    case (UN K)
    then obtain k where k ∈ K ∞ ∈ k
      by auto
    with UN.IH[OF this] show ?case
      by auto
  qed auto
qed

lemma nhds-enat: nhds x = (if x = ∞ then INF i. principal {enat i..} else principal
{x})
proof auto
  show nhds ∞ = (INF i. principal {enat i..})
  proof (rule antisym)
    show nhds ∞ ≤ (INF i. principal {enat i..})
      unfolding nhds-def
     using Ioi-le-Ico by (intro INF-greatest INF-lower) (auto simp: open-enat-iff)
    show (INF i. principal {enat i..}) ≤ nhds ∞
      unfolding nhds-def
    by (intro INF-greatest) (force intro: INF-lower2[of Suc -] simp add: open-enat-iff
Suc-ile-eq)
  qed
  show nhds (enat i) = principal {enat i} for i
    by (simp add: nhds-discrete-open open-enat)
qed

instance enat :: topological-comm-monoid-add
proof
  have [simp]: enat i ≤ aa ⟹ enat i ≤ aa + ba for aa ba i
    by (rule order-trans[OF - add-mono[of aa aa 0 ba]]) auto
  then have [simp]: enat i ≤ ba ⟹ enat i ≤ aa + ba for aa ba i
    by (metis add.commute)
  fix a b :: enat
  have ∀F x in INF m n. principal ({enat n..} × {enat m..}). enat i ≤ fst x +
snd x
      ∀F x in INF n. principal ({enat n..} × {enat j}). enat i ≤ fst x + snd x
      ∀F x in INF n. principal ({enat j} × {enat n..}). enat i ≤ fst x + snd x
    for i j
    by (auto intro!: eventually-INF1[of i] simp: eventually-principal)
  then show ((λx. fst x + snd x) ⟶ a + b) (nhds a ×F nhds b)
    by (auto simp: nhds-enat filterlim-INF prod-filter-INF1 prod-filter-INF2
        filterlim-principal principal-prod-principal eventually-principal)
qed
```

For more lemmas about the extended real numbers see `~~/src/HOL/Analysis/Extended_Real_Limits.thy`.

38.1 Definition and basic properties

datatype *ereal* = *ereal real* | *PInfty* | *MInfty*

instantiation *ereal* :: *uminus*
begin

fun *uminus-ereal* **where**
  − (*ereal r*) = *ereal* (− *r*)
| − *PInfty* = *MInfty*
| − *MInfty* = *PInfty*

instance ..

end

instantiation *ereal* :: *infinity*
begin

definition (∞::*ereal*) = *PInfty*
instance ..

end

declare [[*coercion ereal* :: *real* ⇒ *ereal*]]

lemma *ereal-uminus-uminus*[*simp*]:
  fixes *a* :: *ereal*
  shows − (− *a*) = *a*
  by (*cases a*) *simp-all*

lemma
  shows *PInfty-eq-infinity*[*simp*]: *PInfty* = ∞
    and *MInfty-eq-minfinity*[*simp*]: *MInfty* = −∞
    and *MInfty-neq-PInfty*[*simp*]: ∞ ≠ − (∞::*ereal*) −∞ ≠ (∞::*ereal*)
    and *MInfty-neq-ereal*[*simp*]: *ereal r* ≠ −∞ −∞ ≠ *ereal r*
    and *PInfty-neq-ereal*[*simp*]: *ereal r* ≠ ∞ ∞ ≠ *ereal r*
    and *PInfty-cases*[*simp*]: (*case* ∞ *of ereal r* ⇒ *f r* | *PInfty* ⇒ *y* | *MInfty* ⇒ *z*)
= *y*
    and *MInfty-cases*[*simp*]: (*case* −∞ *of ereal r* ⇒ *f r* | *PInfty* ⇒ *y* | *MInfty* ⇒
z) = *z*
  by (*simp-all add*: *infinity-ereal-def*)

declare
  PInfty-eq-infinity[*code-post*]
  MInfty-eq-minfinity[*code-post*]

lemma [*code-unfold*]:
  ∞ = *PInfty*
  − *PInfty* = *MInfty*

by *simp-all*

lemma *inj-ereal*[*simp*]: *inj-on ereal A*
unfolding *inj-on-def* **by** *auto*

lemma *ereal-cases*[*cases type*: *ereal*]:
obtains (*real*) *r* **where** $x = ereal\ r$
| (*PInf*) $x = \infty$
| (*MInf*) $x = -\infty$
by (*cases x*) *auto*

lemmas *ereal2-cases* = *ereal-cases*[*case-product ereal-cases*]
lemmas *ereal3-cases* = *ereal2-cases*[*case-product ereal-cases*]

lemma *ereal-all-split*: $\bigwedge P.\ (\forall x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \wedge (\forall x.\ P\ (ereal\ x)) \wedge P\ (-\infty)$
by (*metis ereal-cases*)

lemma *ereal-ex-split*: $\bigwedge P.\ (\exists x{::}ereal.\ P\ x) \longleftrightarrow P\ \infty \vee (\exists x.\ P\ (ereal\ x)) \vee P\ (-\infty)$
by (*metis ereal-cases*)

lemma *ereal-uminus-eq-iff*[*simp*]:
fixes *a b* :: *ereal*
shows $-a = -b \longleftrightarrow a = b$
by (*cases rule*: *ereal2-cases*[*of a b*]) *simp-all*

function *real-of-ereal* :: *ereal* ⇒ *real* **where**
real-of-ereal (*ereal r*) = *r*
| *real-of-ereal* ∞ = *0*
| *real-of-ereal* (−∞) = *0*
by (*auto intro*: *ereal-cases*)
termination by *standard* (*rule wf-on-bot*)

lemma *real-of-ereal*[*simp*]:
real-of-ereal (− *x* :: *ereal*) = − (*real-of-ereal x*)
by (*cases x*) *simp-all*

lemma *range-ereal*[*simp*]: *range ereal* = *UNIV* − {∞, −∞}
proof *safe*
fix *x*
assume $x \notin range\ ereal\ x \neq \infty$
then show $x = -\infty$
by (*cases x*) *auto*
qed *auto*

lemma *ereal-range-uminus*[*simp*]: *range uminus* = (*UNIV*::*ereal set*)
proof *safe*
fix *x* :: *ereal*

show $x \in$ *range uminus*
by (*intro image-eqI*[*of* - - $-x$]) *auto*
qed *auto*

instantiation *ereal* :: *abs*
begin

function *abs-ereal* **where**
$|$*ereal r*$|$ = *ereal* $|r|$
$| |-\infty| = (\infty$::*ereal*)
$| |\infty| = (\infty$::*ereal*)
by (*auto intro*: *ereal-cases*)
termination proof qed (*rule wf-on-bot*)

instance ..

end

lemma *abs-eq-infinity-cases*[*elim*!]:
fixes x :: *ereal*
assumes $|x| = \infty$
obtains $x = \infty \mid x = -\infty$
using *assms* **by** (*cases x*) *auto*

lemma *abs-neq-infinity-cases*[*elim*!]:
fixes x :: *ereal*
assumes $|x| \neq \infty$
obtains r **where** $x =$ *ereal r*
using *assms* **by** (*cases x*) *auto*

lemma *abs-ereal-uminus*[*simp*]:
fixes x :: *ereal*
shows $|-x| = |x|$
by (*cases x*) *auto*

lemma *ereal-infinity-cases*:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a \neq -\infty \Longrightarrow |a| \neq \infty$
by *auto*

38.1.1 Addition

instantiation *ereal* :: {*one*,*comm-monoid-add*,*zero-neq-one*}
begin

definition *0* = *ereal 0*
definition *1* = *ereal 1*

function *plus-ereal* **where**

```
  ereal r + ereal p = ereal (r + p)
| ∞ + a = (∞::ereal)
| a + ∞ = (∞::ereal)
| ereal r + −∞ = −∞
| −∞ + ereal p = −(∞::ereal)
| −∞ + −∞ = −(∞::ereal)
proof goal-cases
  case prems: (1 P x)
  then obtain a b where x = (a, b)
    by (cases x) auto
  with prems show P
   by (cases rule: ereal2-cases[of a b]) auto
qed auto
termination by standard (rule wf-on-bot)

lemma Infty-neq-0[simp]:
  (∞::ereal) ≠ 0 0 ≠ (∞::ereal)
  −(∞::ereal) ≠ 0 0 ≠ −(∞::ereal)
  by (simp-all add: zero-ereal-def)

lemma ereal-eq-0[simp]:
  ereal r = 0 ⟷ r = 0
  0 = ereal r ⟷ r = 0
  unfolding zero-ereal-def by simp-all

lemma ereal-eq-1[simp]:
  ereal r = 1 ⟷ r = 1
  1 = ereal r ⟷ r = 1
  unfolding one-ereal-def by simp-all

instance
proof
  fix a b c :: ereal
  show 0 + a = a
    by (cases a) (simp-all add: zero-ereal-def)
  show a + b = b + a
    by (cases rule: ereal2-cases[of a b]) simp-all
  show a + b + c = a + (b + c)
    by (cases rule: ereal3-cases[of a b c]) simp-all
  show 0 ≠ (1::ereal)
    by (simp add: one-ereal-def zero-ereal-def)
qed

end

lemma ereal-0-plus [simp]: ereal 0 + x = x
  and plus-ereal-0 [simp]: x + ereal 0 = x
  by(simp-all flip: zero-ereal-def)
```

instance *ereal* :: *numeral* **..**

lemma *real-of-ereal-0*[*simp*]: *real-of-ereal* (*0*::*ereal*) = *0*
unfolding *zero-ereal-def* **by** *simp*

lemma *abs-ereal-zero*[*simp*]: |*0*| = (*0*::*ereal*)
unfolding *zero-ereal-def abs-ereal.simps* **by** *simp*

lemma *ereal-uminus-zero*[*simp*]: − *0* = (*0*::*ereal*)
by (*simp add*: *zero-ereal-def*)

lemma *ereal-uminus-zero-iff*[*simp*]:
fixes *a* :: *ereal*
shows −*a* = *0* ⟷ *a* = *0*
by (*cases a*) *simp-all*

lemma *ereal-plus-eq-PInfty*[*simp*]:
fixes *a b* :: *ereal*
shows *a* + *b* = ∞ ⟷ *a* = ∞ ∨ *b* = ∞
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-plus-eq-MInfty*[*simp*]:
fixes *a b* :: *ereal*
shows *a* + *b* = −∞ ⟷ (*a* = −∞ ∨ *b* = −∞) ∧ *a* ≠ ∞ ∧ *b* ≠ ∞
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-add-cancel-left*:
fixes *a b* :: *ereal*
assumes *a* ≠ −∞
shows *a* + *b* = *a* + *c* ⟷ *a* = ∞ ∨ *b* = *c*
using *assms* **by** (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-add-cancel-right*:
fixes *a b* :: *ereal*
assumes *a* ≠ −∞
shows *b* + *a* = *c* + *a* ⟷ *a* = ∞ ∨ *b* = *c*
using *assms* **by** (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-real*: *ereal* (*real-of-ereal x*) = (*if* |*x*| = ∞ *then 0 else x*)
by *auto*

lemma *real-of-ereal-add*:
fixes *a b* :: *ereal*
shows *real-of-ereal* (*a* + *b*) =
(*if* (|*a*| = ∞) ∧ (|*b*| = ∞) ∨ (|*a*| ≠ ∞) ∧ (|*b*| ≠ ∞) *then real-of-ereal a* + *real-of-ereal b else 0*)
by *auto*

38.1.2 Linear order on *ereal*

instantiation *ereal* :: *linorder*
begin

function *less-ereal*
where
  ereal x < *ereal y* ⟷ *x* < *y*
| (∞::*ereal*) < *a* ⟷ *False*
| *a* < −(∞::*ereal*) ⟷ *False*
| *ereal x* < ∞ ⟷ *True*
| −∞ < *ereal r* ⟷ *True*
| −∞ < (∞::*ereal*) ⟷ *True*
proof *goal-cases*
  case *prems*: (*1 P x*)
  then obtain *a b* **where** *x* = (*a*,*b*) **by** (*cases x*) *auto*
  with *prems* **show** *P* **by** (*cases rule*: *ereal2-cases*[*of a b*]) *auto*
qed *simp-all*
termination by (*relation* {}) *simp*

definition *x* ≤ (*y*::*ereal*) ⟷ *x* < *y* ∨ *x* = *y*

lemma *ereal-infty-less*[*simp*]:
  fixes *x* :: *ereal*
  shows *x* < ∞ ⟷ (*x* ≠ ∞)
    −∞ < *x* ⟷ (*x* ≠ −∞)
  by (*cases x*, *simp-all*)+

lemma *ereal-infty-less-eq*[*simp*]:
  fixes *x* :: *ereal*
  shows ∞ ≤ *x* ⟷ *x* = ∞
   and *x* ≤ −∞ ⟷ *x* = −∞
  by (*auto simp*: *less-eq-ereal-def*)

lemma *ereal-less*[*simp*]:
  ereal r < *0* ⟷ (*r* < *0*)
  0 < *ereal r* ⟷ (*0* < *r*)
  ereal r < *1* ⟷ (*r* < *1*)
  1 < *ereal r* ⟷ (*1* < *r*)
  0 < (∞::*ereal*)
  −(∞::*ereal*) < *0*
  by (*simp-all add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-less-eq*[*simp*]:
  x ≤ (∞::*ereal*)
  −(∞::*ereal*) ≤ *x*
  ereal r ≤ *ereal p* ⟷ *r* ≤ *p*
  ereal r ≤ *0* ⟷ *r* ≤ *0*
  0 ≤ *ereal r* ⟷ *0* ≤ *r*
  ereal r ≤ *1* ⟷ *r* ≤ *1*

$1 \leq \mathit{ereal}\ r \longleftrightarrow 1 \leq r$
by (*auto simp*: *less-eq-ereal-def zero-ereal-def one-ereal-def*)

lemma *ereal-infty-less-eq2*:
$a \leq b \Longrightarrow a = \infty \Longrightarrow b = (\infty::\mathit{ereal})$
$a \leq b \Longrightarrow b = -\infty \Longrightarrow a = -(\infty::\mathit{ereal})$
by *simp-all*

instance
proof
fix $x\ y\ z :: \mathit{ereal}$
show $x \leq x$
by (*cases x*) *simp-all*
show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*
show $x \leq y \vee y \leq x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*
assume $x \leq y$
then show $y \leq x \Longrightarrow x = y$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*
show $y \leq z \Longrightarrow x \leq z$
using ‹$x \leq y$›
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*
qed

end

lemma *ereal-dense2*: $x < y \Longrightarrow \exists\, z.\ x < \mathit{ereal}\ z \wedge \mathit{ereal}\ z < y$
using *lt-ex gt-ex dense* **by** (*cases x y rule*: *ereal2-cases*) *auto*

instance *ereal* :: *dense-linorder*
by *standard* (*blast dest*: *ereal-dense2*)

instance *ereal* :: *ordered-comm-monoid-add*
proof
fix $a\ b\ c :: \mathit{ereal}$
assume $a \leq b$
then show $c + a \leq c + b$
by (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*
qed

lemma *ereal-one-not-less-zero-ereal*[*simp*]: $\neg\ 1 < (0::\mathit{ereal})$
by (*simp add*: *zero-ereal-def*)

lemma *real-of-ereal-positive-mono*:
fixes $x\ y :: \mathit{ereal}$
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow y \neq \infty \Longrightarrow \mathit{real\text{-}of\text{-}ereal}\ x \leq \mathit{real\text{-}of\text{-}ereal}\ y$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*

lemma *ereal-MInfty-lessI*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq -\infty \Longrightarrow -\infty < a$
by *simp*

lemma *ereal-less-PInfty*[*intro*, *simp*]:
fixes a :: *ereal*
shows $a \neq \infty \Longrightarrow a < \infty$
by *simp*

lemma *ereal-less-ereal-Ex*:
fixes a b :: *ereal*
shows $x < ereal\ r \longleftrightarrow x = -\infty \vee (\exists\, p.\ p < r \wedge x = ereal\ p)$
by (*cases x*) *auto*

lemma *less-PInf-Ex-of-nat*: $x \neq \infty \longleftrightarrow (\exists\, n{::}nat.\ x < ereal\ (real\ n))$
proof (*cases x*)
case (*real r*)
then show *?thesis*
using *reals-Archimedean2*[*of r*] **by** *simp*
qed *simp-all*

lemma *ereal-add-strict-mono2*:
fixes a b c d :: *ereal*
assumes $a < b$ **and** $c < d$
shows $a + c < b + d$
using *assms*
by (*cases a*; *force simp*: *elim*: *less-ereal.elims*)

lemma *ereal-minus-le-minus*[*simp*]:
fixes a b :: *ereal*
shows $-a \leq -b \longleftrightarrow b \leq a$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-minus-less-minus*[*simp*]:
fixes a b :: *ereal*
shows $-a < -b \longleftrightarrow b < a$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-le-real-iff*:
$x \leq real\text{-}of\text{-}ereal\ y \longleftrightarrow (|y| \neq \infty \longrightarrow ereal\ x \leq y) \wedge (|y| = \infty \longrightarrow x \leq 0)$
by (*cases y*) *auto*

lemma *real-le-ereal-iff*:
$real\text{-}of\text{-}ereal\ y \leq x \longleftrightarrow (|y| \neq \infty \longrightarrow y \leq ereal\ x) \wedge (|y| = \infty \longrightarrow 0 \leq x)$
by (*cases y*) *auto*

lemma *ereal-less-real-iff*:
$x < real\text{-}of\text{-}ereal\ y \longleftrightarrow (|y| \neq \infty \longrightarrow ereal\ x < y) \wedge (|y| = \infty \longrightarrow x < 0)$

by (*cases y*) *auto*

lemma *real-less-ereal-iff*:
real-of-ereal y $< x \longleftrightarrow (|y| \neq \infty \longrightarrow y <$ *ereal x*$) \wedge (|y| = \infty \longrightarrow 0 < x)$
by (*cases y*) *auto*

To help with inferences like $[\![a <$ *ereal x*; $x < y]\!] \Longrightarrow a <$ *ereal y*, where x and y are real.

lemma *le-ereal-le*: $a \leq$ *ereal x* $\Longrightarrow x \leq y \Longrightarrow a \leq$ *ereal y*
using *ereal-less-eq*(*3*) *order.trans* **by** *blast*

lemma *le-ereal-less*: $a \leq$ *ereal x* $\Longrightarrow x < y \Longrightarrow a <$ *ereal y*
by (*simp add*: *le-less-trans*)

lemma *less-ereal-le*: $a <$ *ereal x* $\Longrightarrow x \leq y \Longrightarrow a <$ *ereal y*
using *ereal-less-ereal-Ex* **by** *auto*

lemma *ereal-le-le*: *ereal y* $\leq a \Longrightarrow x \leq y \Longrightarrow$ *ereal x* $\leq a$
by (*simp add*: *order-subst2*)

lemma *ereal-le-less*: *ereal y* $\leq a \Longrightarrow x < y \Longrightarrow$ *ereal x* $< a$
by (*simp add*: *dual-order.strict-trans1*)

lemma *ereal-less-le*: *ereal y* $< a \Longrightarrow x \leq y \Longrightarrow$ *ereal x* $< a$
using *ereal-less-eq*(*3*) *le-less-trans* **by** *blast*

lemma *real-of-ereal-pos*:
fixes *x* :: *ereal*
shows $0 \leq x \Longrightarrow 0 \leq$ *real-of-ereal x*
by (*cases x*) *auto*

lemmas *real-of-ereal-ord-simps* =
ereal-le-real-iff real-le-ereal-iff ereal-less-real-iff real-less-ereal-iff

lemma *abs-ereal-ge0*[*simp*]: $0 \leq x \Longrightarrow |x ::$ *ereal*$| = x$
by (*cases x*) *auto*

lemma *abs-ereal-less0*[*simp*]: $x < 0 \Longrightarrow |x ::$ *ereal*$| = -x$
by (*cases x*) *auto*

lemma *abs-ereal-pos*[*simp*]: $0 \leq |x ::$ *ereal*$|$
by (*cases x*) *auto*

lemma *ereal-abs-leI*:
fixes *x y* :: *ereal*
shows $[\![x \leq y; -x \leq y]\!] \Longrightarrow |x| \leq y$
by(*cases x y rule*: *ereal2-cases*)(*simp-all*)

lemma *ereal-abs-add*:

fixes a b::*ereal*
shows $abs(a+b) \leq abs\ a + abs\ b$
by (*cases rule*: *ereal2-cases*[*of* a b]) (*auto*)

lemma *real-of-ereal-le-0*[*simp*]: *real-of-ereal* $(x :: ereal) \leq 0 \longleftrightarrow x \leq 0 \vee x = \infty$
by (*cases* x) *auto*

lemma *abs-real-of-ereal*[*simp*]: $|$*real-of-ereal* $(x :: ereal)| =$ *real-of-ereal* $|x|$
by (*cases* x) *auto*

lemma *zero-less-real-of-ereal*:
fixes $x ::$ *ereal*
shows $0 <$ *real-of-ereal* $x \longleftrightarrow 0 < x \wedge x \neq \infty$
by (*cases* x) *auto*

lemma *ereal-0-le-uminus-iff*[*simp*]:
fixes $a ::$ *ereal*
shows $0 \leq - a \longleftrightarrow a \leq 0$
by (*cases rule*: *ereal2-cases*[*of* a]) *auto*

lemma *ereal-uminus-le-0-iff*[*simp*]:
fixes $a ::$ *ereal*
shows $- a \leq 0 \longleftrightarrow 0 \leq a$
by (*cases rule*: *ereal2-cases*[*of* a]) *auto*

lemma *ereal-add-strict-mono*:
fixes a b c $d ::$ *ereal*
assumes $a \leq b$
and $0 \leq a$
and $a \neq \infty$
and $c < d$
shows $a + c < b + d$
using *assms*
by (*cases rule*: *ereal3-cases*[*case-product ereal-cases*, *of* a b c d]) *auto*

lemma *ereal-less-add*:
fixes a b $c ::$ *ereal*
shows $|a| \neq \infty \Longrightarrow c < b \Longrightarrow a + c < a + b$
by (*cases rule*: *ereal2-cases*[*of* b c]) *auto*

lemma *ereal-uminus-eq-reorder*: $- a = b \longleftrightarrow a = (-b$::*ereal*$)$
by *auto*

lemma *ereal-uminus-less-reorder*: $- a < b \longleftrightarrow -b < a$
and *ereal-less-uminus-reorder*: $a < - b \longleftrightarrow b < - a$
and *ereal-uminus-le-reorder*: $- a \leq b \longleftrightarrow -b \leq a$ **for** a::*ereal*
using *ereal-minus-le-minus ereal-minus-less-minus* **by** *fastforce*+

lemmas *ereal-uminus-reorder* =

ereal-uminus-eq-reorder ereal-uminus-less-reorder ereal-uminus-le-reorder

lemma *ereal-bot*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \leq ereal\ B$
shows $x = -\infty$
proof (*cases x*)
case (*real r*)
with *assms*[*of r − 1*] **show** *?thesis*
by *auto*
next
case *PInf*
with *assms*[*of 0*] **show** *?thesis*
by *auto*
next
case *MInf*
then show *?thesis*
by *simp*
qed

lemma *ereal-top*:
fixes x :: *ereal*
assumes $\bigwedge B.\ x \geq ereal\ B$
shows $x = \infty$
proof (*cases x*)
case (*real r*)
with *assms*[*of r + 1*] **show** *?thesis*
by *auto*
next
case *MInf*
with *assms*[*of 0*] **show** *?thesis*
by *auto*
next
case *PInf*
then show *?thesis*
by *simp*
qed

lemma
shows *ereal-max*[*simp*]: *ereal* (*max x y*) = *max* (*ereal x*) (*ereal y*)
and *ereal-min*[*simp*]: *ereal* (*min x y*) = *min* (*ereal x*) (*ereal y*)
by (*simp-all add*: *min-def max-def*)

lemma *ereal-max-0*: *max 0* (*ereal r*) = *ereal* (*max 0 r*)
by (*auto simp*: *zero-ereal-def*)

lemma
fixes f :: *nat* ⇒ *ereal*
shows *ereal-incseq-uminus*[*simp*]: *incseq* (λ*x*. − *f x*) ⟷ *decseq f*

and *ereal-decseq-uminus*[*simp*]: *decseq* $(\lambda x.\ -f\ x) \longleftrightarrow$ *incseq f*
unfolding *decseq-def incseq-def* **by** *auto*

lemma *incseq-ereal*: *incseq f* $\Longrightarrow$ *incseq* $(\lambda x.\ ereal\ (f\ x))$
unfolding *incseq-def* **by** *auto*

lemma *sum-ereal*[*simp*]: $(\sum x{\in}A.\ ereal\ (f\ x)) = ereal\ (\sum x{\in}A.\ f\ x)$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *sum-list-ereal* [*simp*]: *sum-list* (*map* $(\lambda x.\ ereal\ (f\ x))$ *xs*) = *ereal* (*sum-list* (*map f xs*))
by (*induction xs*) *simp-all*

lemma *sum-Pinfty*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\sum x{\in}P.\ f\ x) = \infty \longleftrightarrow$ *finite P* $\wedge$ $(\exists\, i{\in}P.\ f\ i = \infty)$
proof *safe*
assume $*$: *sum f P* $= \infty$
show *finite P*
by (*metis* $*$ *Infty-neq-0*(*2*) *sum.infinite*)
show $\exists\, i{\in}P.\ f\ i = \infty$
proof (*rule ccontr*)
assume $\neg$ *?thesis*
then have $\bigwedge i.\ i \in P \Longrightarrow f\ i \neq \infty$
by *auto*
with ‹*finite P*› **have** *sum f P* $\neq \infty$
by *induct auto*
with $*$ **show** *False*
by *auto*
qed
next
fix *i*
assume *finite P* **and** $i \in P$ **and** $f\ i = \infty$
then show *sum f P* $= \infty$
proof *induct*
case (*insert x A*)
show *?case* **using** *insert* **by** (*cases* $x = i$) *auto*
qed *simp*
qed

lemma *sum-Inf*:
fixes $f :: 'a \Rightarrow ereal$
shows $|sum\ f\ A| = \infty \longleftrightarrow$ *finite A* $\wedge$ $(\exists\, i{\in}A.\ |f\ i| = \infty)$
proof
assume $*$: $|sum\ f\ A| = \infty$
have *finite A*
by (*rule ccontr*) (*insert* $*$, *auto*)
moreover have $\exists\, i{\in}A.\ |f\ i| = \infty$
proof (*rule ccontr*)

```
    assume ¬ ?thesis
    then have ∀ i∈A. ∃ r. f i = ereal r
      by auto
    then obtain r where ∀ x∈A. f x = ereal (r x)
      by metis
    with ∗ show False
      by auto
  qed
  ultimately show finite A ∧ (∃ i∈A. |f i| = ∞)
    by auto
next
  assume finite A ∧ (∃ i∈A. |f i| = ∞)
  then obtain i where finite A i ∈ A and |f i| = ∞
    by auto
  then show |sum f A| = ∞
  proof induct
    case (insert j A)
    then show ?case
      by (cases rule: ereal3-cases[of f i f j sum f A]) auto
  qed simp
qed

lemma sum-real-of-ereal:
  fixes f :: 'i ⇒ ereal
  assumes ⋀x. x ∈ S ⟹ |f x| ≠ ∞
  shows (∑ x∈S. real-of-ereal (f x)) = real-of-ereal (sum f S)
proof −
  have ∀ x∈S. ∃ r. f x = ereal r
    using assms by blast
  then obtain r where ∀ x∈S. f x = ereal (r x)
    by metis
  then show ?thesis
    by simp
qed
```

38.1.3 Multiplication

```
instantiation ereal :: {comm-monoid-mult,sgn}
begin

function sgn-ereal :: ereal ⇒ ereal where
  sgn (ereal r) = ereal (sgn r)
| sgn (∞::ereal) = 1
| sgn (−∞::ereal) = −1
by (auto intro: ereal-cases)
termination by standard (rule wf-on-bot)

function times-ereal where
  ereal r ∗ ereal p = ereal (r ∗ p)
```

| *ereal r* $*$ ∞ = (*if r = 0 then 0 else if r > 0 then* ∞ *else* $-\infty$)
| ∞ $*$ *ereal r* = (*if r = 0 then 0 else if r > 0 then* ∞ *else* $-\infty$)
| *ereal r* $*$ $-\infty$ = (*if r = 0 then 0 else if r > 0 then* $-\infty$ *else* ∞)
| $-\infty$ $*$ *ereal r* = (*if r = 0 then 0 else if r > 0 then* $-\infty$ *else* ∞)
| (∞::*ereal*) $*$ ∞ = ∞
| $-$(∞::*ereal*) $*$ ∞ = $-\infty$
| (∞::*ereal*) $*$ $-\infty$ = $-\infty$
| $-$(∞::*ereal*) $*$ $-\infty$ = ∞
proof *goal-cases*
  case *prems*: (*1 P x*)
  then obtain *a b* **where** *x* = (*a*, *b*)
    by (*cases x*) *auto*
  with *prems* **show** *P*
    by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*
qed *simp-all*
termination by (*relation* {}) *simp*

instance
proof
  fix *a b c* :: *ereal*
  show *1* $*$ *a* = *a*
    by (*cases a*) (*simp-all add*: *one-ereal-def*)
  show *a* $*$ *b* = *b* $*$ *a*
    by (*cases rule*: *ereal2-cases*[*of a b*]) *simp-all*
  show *a* $*$ *b* $*$ *c* = *a* $*$ (*b* $*$ *c*)
    by (*cases rule*: *ereal3-cases*[*of a b c*])
      (*simp-all add*: *zero-ereal-def zero-less-mult-iff*)
qed

end

lemma [*simp*]:
  shows *ereal-1-times*: *ereal 1* $*$ *x* = *x*
  and *times-ereal-1*: *x* $*$ *ereal 1* = *x*
by(*simp-all flip*: *one-ereal-def*)

lemma *one-not-le-zero-ereal*[*simp*]: $\neg$ (*1* $\leq$ (*0*::*ereal*))
  by (*simp add*: *one-ereal-def zero-ereal-def*)

lemma *real-ereal-1*[*simp*]: *real-of-ereal* (*1*::*ereal*) = *1*
  unfolding *one-ereal-def* **by** *simp*

lemma *real-of-ereal-le-1*:
  fixes *a* :: *ereal*
  shows *a* $\leq$ *1* $\Longrightarrow$ *real-of-ereal a* $\leq$ *1*
  by (*cases a*) (*auto simp*: *one-ereal-def*)

lemma *abs-ereal-one*[*simp*]: |*1*| = (*1*::*ereal*)
  unfolding *one-ereal-def* **by** *simp*

lemma *ereal-mult-zero*[*simp*]:
fixes *a* :: *ereal*
shows *a* ∗ *0* = *0*
by (*cases a*) (*simp-all add*: *zero-ereal-def*)

lemma *ereal-zero-mult*[*simp*]:
fixes *a* :: *ereal*
shows *0* ∗ *a* = *0*
by (*metis ereal-mult-zero mult.commute*)

lemma *ereal-m1-less-0*[*simp*]: −(*1*::*ereal*) < *0*
by (*simp add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-times*[*simp*]:
1 ≠ (∞::*ereal*) (∞::*ereal*) ≠ *1*
1 ≠ −(∞::*ereal*) −(∞::*ereal*) ≠ *1*
by (*auto simp*: *one-ereal-def*)

lemma *ereal-plus-1*[*simp*]:
1 + *ereal r* = *ereal* (*r* + *1*)
ereal r + *1* = *ereal* (*r* + *1*)
1 + −(∞::*ereal*) = −∞
−(∞::*ereal*) + *1* = −∞
unfolding *one-ereal-def* **by** *auto*

lemma *ereal-zero-times*[*simp*]:
fixes *a b* :: *ereal*
shows *a* ∗ *b* = *0* ⟷ *a* = *0* ∨ *b* = *0*
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-eq-PInfty*[*simp*]:
a ∗ *b* = (∞::*ereal*) ⟷
(*a* = ∞ ∧ *b* > *0*) ∨ (*a* > *0* ∧ *b* = ∞) ∨ (*a* = −∞ ∧ *b* < *0*) ∨ (*a* < *0* ∧ *b* = −∞)
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-eq-MInfty*[*simp*]:
a ∗ *b* = −(∞::*ereal*) ⟷
(*a* = ∞ ∧ *b* < *0*) ∨ (*a* < *0* ∧ *b* = ∞) ∨ (*a* = −∞ ∧ *b* > *0*) ∨ (*a* > *0* ∧ *b* = −∞)
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-abs-mult*: |*x* ∗ *y* :: *ereal*| = |*x*| ∗ |*y*|
by (*cases x y rule*: *ereal2-cases*) (*auto simp*: *abs-mult*)

lemma *ereal-0-less-1*[*simp*]: *0* < (*1*::*ereal*)
by (*simp add*: *zero-ereal-def one-ereal-def*)

lemma *ereal-mult-minus-left*[*simp*]:
fixes *a b* :: *ereal*
shows $-a * b = -(a * b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-minus-right*[*simp*]:
fixes *a b* :: *ereal*
shows $a * -b = -(a * b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-mult-infty*[*simp*]:
$a * (\infty::ereal) = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
by (*cases a*) *auto*

lemma *ereal-infty-mult*[*simp*]:
$(\infty::ereal) * a = (if\ a = 0\ then\ 0\ else\ if\ 0 < a\ then\ \infty\ else\ -\infty)$
by (*cases a*) *auto*

lemma *ereal-mult-strict-right-mono*:
assumes $a < b$
and $0 < c$
and $c < (\infty::ereal)$
shows $a * c < b * c$
using *assms*
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*auto simp*: *zero-le-mult-iff*)

lemma *ereal-mult-strict-left-mono*:
$a < b \Longrightarrow 0 < c \Longrightarrow c < (\infty::ereal) \Longrightarrow c * a < c * b$
using *ereal-mult-strict-right-mono*
by (*simp add*: *mult.commute*[*of c*])

lemma *ereal-mult-right-mono*:
fixes *a b c* :: *ereal*
assumes $a \leq b\ 0 \leq c$
shows $a * c \leq b * c$
proof (*cases* $c = 0$)
case *False*
with *assms* **show** *?thesis*
by (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*
qed *auto*

lemma *ereal-mult-left-mono*:
fixes *a b c* :: *ereal*
shows $a \leq b \Longrightarrow 0 \leq c \Longrightarrow c * a \leq c * b$
by (*simp add*: *ereal-mult-right-mono mult.commute*)

lemma *ereal-mult-mono*:
fixes *a b c d*::*ereal*
assumes $b \geq 0\ c \geq 0\ a \leq b\ c \leq d$

shows $a * c \leq b * d$
by (*metis ereal-mult-right-mono mult.commute order-trans assms*)

lemma *ereal-mult-mono'*:
fixes $a\ b\ c\ d$::*ereal*
assumes $a \geq 0\ c \geq 0\ a \leq b\ c \leq d$
shows $a * c \leq b * d$
by (*metis ereal-mult-right-mono mult.commute order-trans assms*)

lemma *ereal-mult-mono-strict*:
fixes $a\ b\ c\ d$::*ereal*
assumes $b > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
proof −
have $c < \infty$ **using** ‹$c < d$›
by *auto*
then have $a * c < b * c$
by (*metis ereal-mult-strict-left-mono*[*OF assms*(*3*) *assms*(*2*)] *mult.commute*)
moreover have $b * c \leq b * d$
using *assms*(*1*,*4*) *ereal-mult-left-mono* **by** *force*
ultimately show *?thesis* **by** *simp*
qed

lemma *ereal-mult-mono-strict'*:
fixes $a\ b\ c\ d$::*ereal*
assumes $a > 0\ c > 0\ a < b\ c < d$
shows $a * c < b * d$
using *assms ereal-mult-mono-strict* **by** *auto*

lemma *zero-less-one-ereal*[*simp*]: $0 \leq (1$::*ereal*$)$
by (*simp add*: *one-ereal-def zero-ereal-def*)

lemma *ereal-0-le-mult*[*simp*]: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow 0 \leq a * (b :: ereal)$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-right-distrib*:
fixes $r\ a\ b$ :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow r * (a + b) = r * a + r * b$
by (*cases rule*: *ereal3-cases*[*of r a b*]) (*simp-all add*: *field-simps*)

lemma *ereal-left-distrib*:
fixes $r\ a\ b$ :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow (a + b) * r = a * r + b * r$
by (*cases rule*: *ereal3-cases*[*of r a b*]) (*simp-all add*: *field-simps*)

lemma *ereal-mult-le-0-iff*:
fixes $a\ b$ :: *ereal*
shows $a * b \leq 0 \longleftrightarrow (0 \leq a \wedge b \leq 0) \vee (a \leq 0 \wedge 0 \leq b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *mult-le-0-iff*)

lemma *ereal-zero-le-0-iff*:
fixes *a b* :: *ereal*
shows $0 \leq a * b \longleftrightarrow (0 \leq a \wedge 0 \leq b) \vee (a \leq 0 \wedge b \leq 0)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *zero-le-mult-iff*)

lemma *ereal-mult-less-0-iff*:
fixes *a b* :: *ereal*
shows $a * b < 0 \longleftrightarrow (0 < a \wedge b < 0) \vee (a < 0 \wedge 0 < b)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *mult-less-0-iff*)

lemma *ereal-zero-less-0-iff*:
fixes *a b* :: *ereal*
shows $0 < a * b \longleftrightarrow (0 < a \wedge 0 < b) \vee (a < 0 \wedge b < 0)$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*simp-all add*: *zero-less-mult-iff*)

lemma *ereal-left-mult-cong*:
fixes *a b c* :: *ereal*
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow a * c = b * d$
by (*cases* $c = 0$) *simp-all*

lemma *ereal-right-mult-cong*:
fixes *a b c* :: *ereal*
shows $c = d \Longrightarrow (d \neq 0 \Longrightarrow a = b) \Longrightarrow c * a = d * b$
by (*cases* $c = 0$) *simp-all*

lemma *ereal-distrib*:
fixes *a b c* :: *ereal*
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $(a + b) * c = a * c + b * c$
using *assms*
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *field-simps*)

lemma *numeral-eq-ereal* [*simp*]: *numeral w* = *ereal* (*numeral w*)
proof (*induct w rule*: *num-induct*)
case *One*
then show *?case*
by *simp*
next
case (*inc x*)
then show *?case*
by (*simp add*: *inc numeral-inc*)
qed

lemma *m1-ereal-less-iff* [*simp*]:
$((-1::ereal) < numeral\ a) \longleftrightarrow ((-1::real) < numeral\ a)$
by (*simp add*: *one-ereal-def*)

lemma *m1-ereal-le-iff* [*simp*]:
((-1::*ereal*) $\leq$ *numeral a*) $\longleftrightarrow$ ((-1::*real*) $\leq$ *numeral a*)
by (*simp add*: *one-ereal-def*)

lemma *m1-ereal-eq-iff* [*simp*]:
((-1::*ereal*) = *numeral a*) $\longleftrightarrow$ ((-1::*real*) = *numeral a*)
by (*simp add*: *one-ereal-def*)

lemma *ereal-less-m1-iff* [*simp*]:
(*numeral a* $<$ (-1::*ereal*)) $\longleftrightarrow$ (*numeral a* $<$ (-1::*real*))
by (*simp add*: *one-ereal-def*)

lemma *ereal-le-m1-iff* [*simp*]:
(*numeral a* $\leq$ (-1::*ereal*)) $\longleftrightarrow$ (*numeral a* $\leq$ (-1::*real*))
by (*simp add*: *one-ereal-def*)

lemma *ereal-eq-m1-iff* [*simp*]:
(*numeral a* = (-1::*ereal*)) $\longleftrightarrow$ (*numeral a* = (-1::*real*))
by (*simp add*: *one-ereal-def*)

lemma *distrib-left-ereal-nn*:
$c \geq 0 \Longrightarrow (x + y) * ereal\ c = x * ereal\ c + y * ereal\ c$
by(*cases x y rule*: *ereal2-cases*)(*simp-all add*: *ring-distribs*)

lemma *sum-ereal-right-distrib*:
fixes $f :: 'a \Rightarrow ereal$
shows $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow r * sum\ f\ A = (\sum n{\in}A.\ r * f\ n)$
by (*induct A rule*: *infinite-finite-induct*) (*auto simp*: *ereal-right-distrib sum-nonneg*)

lemma *sum-ereal-left-distrib*:
$(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow sum\ f\ A * r = (\sum n{\in}A.\ f\ n * r :: ereal)$
using *sum-ereal-right-distrib*[*of A f r*] **by** (*simp add*: *mult-ac*)

lemma *sum-distrib-right-ereal*:
$c \geq 0 \Longrightarrow sum\ f\ A * ereal\ c = (\sum x{\in}A.\ f\ x * c :: ereal)$
by(*subst sum-comp-morphism*[**where** $h{=}\lambda x.\ x * ereal\ c$, *symmetric*])(*simp-all add*: *distrib-left-ereal-nn*)

lemma *ereal-le-epsilon*:
fixes $x\ y :: ereal$
assumes $\bigwedge e.\ 0 < e \Longrightarrow x \leq y + e$
shows $x \leq y$
proof (*cases* $x = -\infty \vee x = \infty \vee y = -\infty \vee y = \infty$)
case *True*
then show *?thesis*
using *assms*[*of 1*] **by** *auto*
next
case *False*

then obtain *p q* **where** *x = ereal p y = ereal q*
by (*metis MInfty-eq-minfinity ereal.distinct(3) uminus-ereal.elims*)
then show *?thesis*
by (*metis assms field-le-epsilon ereal-less(2) ereal-less-eq(3) plus-ereal.simps(1)*)
qed

lemma *ereal-le-epsilon2*:
fixes *x y* :: *ereal*
assumes $\bigwedge e$::*real*. *0 < e* $\Longrightarrow$ *x ≤ y + ereal e*
shows *x ≤ y*
proof (*rule ereal-le-epsilon*)
show $\bigwedge \varepsilon$::*ereal*. *0 < ε* $\Longrightarrow$ *x ≤ y + ε*
using *assms less-ereal.elims(2) zero-less-real-of-ereal* **by** *fastforce*
qed

lemma *ereal-le-real*:
fixes *x y* :: *ereal*
assumes $\bigwedge z$. *x ≤ ereal z* $\Longrightarrow$ *y ≤ ereal z*
shows *y ≤ x*
by (*metis assms ereal-bot ereal-cases ereal-infty-less-eq(2) ereal-less-eq(1) linorder-le-cases*)

lemma *prod-ereal-0*:
fixes *f* :: *'a* ⇒ *ereal*
shows $(\prod i{\in}A.\ f\ i) = 0 \longleftrightarrow$ *finite A* ∧ $(\exists\, i{\in}A.\ f\ i = 0)$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *prod-ereal-pos*:
fixes *f* :: *'a* ⇒ *ereal*
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i$
shows $0 \le (\prod i{\in}I.\ f\ i)$
using *assms*
by (*induction I rule*: *infinite-finite-induct*) *auto*

lemma *prod-PInf*:
fixes *f* :: *'a* ⇒ *ereal*
assumes $\bigwedge i.\ i \in I \Longrightarrow 0 \le f\ i$
shows $(\prod i{\in}I.\ f\ i) = \infty \longleftrightarrow$ *finite I* ∧ $(\exists\, i{\in}I.\ f\ i = \infty)$ ∧ $(\forall\, i{\in}I.\ f\ i \neq 0)$
using *assms*
proof (*induction I rule*: *infinite-finite-induct*)
case (*insert i I*)
then have *pos*: *0 ≤ f i 0 ≤ prod f I*
by (*auto intro*!: *prod-ereal-pos*)
from *insert* **have** $(\prod j{\in}$*insert i I*. *f j*$) = \infty \longleftrightarrow$ *prod f I ∗ f i = ∞*
by *auto*
also have ... $\longleftrightarrow$ *(prod f I = ∞ ∨ f i = ∞) ∧ f i ≠ 0 ∧ prod f I ≠ 0*
using *prod-ereal-pos*[*of I f*] *pos*
by (*cases rule*: *ereal2-cases*[*of f i prod f I*]) *auto*
also have ... $\longleftrightarrow$ *finite* (*insert i I*) ∧ $(\exists j{\in}$*insert i I*. *f j = ∞*$)$ ∧ $(\forall j{\in}$*insert i I*. *f j ≠ 0*$)$

using *insert* **by** (*auto simp*: *prod-ereal-0*)
finally show *?case* **.**
qed *auto*

lemma *prod-ereal*: $(\prod i{\in}A.\ ereal\ (f\ i)) = ereal\ (prod\ f\ A)$
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *one-ereal-def*)

38.1.4 Power

lemma *ereal-power*[*simp*]: $(ereal\ x)\ \hat{}\ n = ereal\ (x\hat{}n)$
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-PInf*[*simp*]: $(\infty{::}ereal)\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ \infty)$
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-uminus*[*simp*]:
fixes $x :: ereal$
shows $(-x)\ \hat{}\ n = (if\ even\ n\ then\ x\ \hat{}\ n\ else\ -(x\hat{}n))$
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *ereal-power-numeral*[*simp*]:
$(numeral\ num :: ereal)\ \hat{}\ n = ereal\ (numeral\ num\ \hat{}\ n)$
by (*induct n*) (*auto simp*: *one-ereal-def*)

lemma *zero-le-power-ereal*[*simp*]:
fixes $a :: ereal$
assumes $0 \le a$
shows $0 \le a\ \hat{}\ n$
using *assms* **by** (*induct n*) (*auto simp*: *ereal-zero-le-0-iff*)

38.1.5 Subtraction

lemma *ereal-minus-minus-image*[*simp*]:
fixes $S :: ereal\ set$
shows $uminus\ `\ uminus\ `\ S = S$
by (*auto simp*: *image-iff*)

lemma *ereal-uminus-lessThan*[*simp*]:
fixes $a :: ereal$
shows $uminus\ `\ \{..<a\} = \{-a<..\}$
by (*force simp*: *ereal-uminus-less-reorder*)

lemma *ereal-uminus-greaterThan*[*simp*]: $uminus\ `\ \{(a{::}ereal)<..\} = \{..<-a\}$
by (*metis ereal-uminus-lessThan ereal-uminus-uminus ereal-minus-minus-image*)

instantiation *ereal* :: *minus*
begin

definition $x - y = x + -(y{::}ereal)$
instance ..

end

lemma *ereal-minus*[*simp*]:
ereal r − ereal p = ereal (r − p)
−∞ − ereal r = −∞
ereal r −∞ = −∞
(∞::ereal) − x = ∞
−(∞::ereal) −∞ = −∞
x − −y = x + y
x − 0 = x
0 − x = −x
by (*simp-all add*: *minus-ereal-def*)

lemma *ereal-x-minus-x*[*simp*]: *x − x = (if |x| = ∞ then ∞ else 0::ereal)*
by *auto*

lemma *ereal-eq-minus-iff*:
fixes *x y z* :: *ereal*
shows *x = z − y ⟷*
(|y| ≠ ∞ ⟶ x + y = z) ∧
(y = −∞ ⟶ x = ∞) ∧
(y = ∞ ⟶ z = ∞ ⟶ x = ∞) ∧
(y = ∞ ⟶ z ≠ ∞ ⟶ x = −∞)
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-eq-minus*:
fixes *x y z* :: *ereal*
shows *|y| ≠ ∞ ⟹ x = z − y ⟷ x + y = z*
by (*auto simp*: *ereal-eq-minus-iff*)

lemma *ereal-less-minus-iff*:
fixes *x y z* :: *ereal*
shows *x < z − y ⟷*
(y = ∞ ⟶ z = ∞ ∧ x ≠ ∞) ∧
(y = −∞ ⟶ x ≠ ∞) ∧
(|y| ≠ ∞⟶ x + y < z)
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-less-minus*:
fixes *x y z* :: *ereal*
shows *|y| ≠ ∞ ⟹ x < z − y ⟷ x + y < z*
by (*auto simp*: *ereal-less-minus-iff*)

lemma *ereal-le-minus-iff*:
fixes *x y z* :: *ereal*
shows *x ≤ z − y ⟷ (y = ∞ ⟶ z ≠ ∞ ⟶ x = −∞) ∧ (|y| ≠ ∞ ⟶ x + y ≤ z)*
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-le-minus*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x \leq z - y \longleftrightarrow x + y \leq z$
by (*auto simp*: *ereal-le-minus-iff*)

lemma *ereal-minus-less-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y < z \longleftrightarrow y \neq -\infty \wedge (y = \infty \longrightarrow x \neq \infty \wedge z \neq -\infty) \wedge (y \neq \infty \longrightarrow x < z + y)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-minus-less*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y < z \longleftrightarrow x < z + y$
by (*auto simp*: *ereal-minus-less-iff*)

lemma *ereal-minus-le-iff*:
fixes $x\ y\ z :: ereal$
shows $x - y \leq z \longleftrightarrow$
$(y = -\infty \longrightarrow z = \infty) \wedge$
$(y = \infty \longrightarrow x = \infty \longrightarrow z = \infty) \wedge$
$(|y| \neq \infty \longrightarrow x \leq z + y)$
by (*cases rule*: *ereal3-cases*[*of x y z*]) *auto*

lemma *ereal-minus-le*:
fixes $x\ y\ z :: ereal$
shows $|y| \neq \infty \Longrightarrow x - y \leq z \longleftrightarrow x \leq z + y$
by (*auto simp*: *ereal-minus-le-iff*)

lemma *ereal-minus-eq-minus-iff*:
fixes $a\ b\ c :: ereal$
shows $a - b = a - c \longleftrightarrow$
$b = c \vee a = \infty \vee (a = -\infty \wedge b \neq -\infty \wedge c \neq -\infty)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) *auto*

lemma *ereal-add-le-add-iff*:
fixes $a\ b\ c :: ereal$
shows $c + a \leq c + b \longleftrightarrow$
$a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *field-simps*)

lemma *ereal-add-le-add-iff2*:
fixes $a\ b\ c :: ereal$
shows $a + c \leq b + c \longleftrightarrow a \leq b \vee c = \infty \vee (c = -\infty \wedge a \neq \infty \wedge b \neq \infty)$
by (*metis* (*no-types*, *lifting*) *add.commute ereal-add-le-add-iff*)

lemma *ereal-mult-le-mult-iff*:
fixes $a\ b\ c :: ereal$

shows $|c| \neq \infty \Longrightarrow c * a \leq c * b \longleftrightarrow (0 < c \longrightarrow a \leq b) \wedge (c < 0 \longrightarrow b \leq a)$
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *mult-le-cancel-left*)

lemma *ereal-minus-mono*:
fixes $A\ B\ C\ D$ :: *ereal* **assumes** $A \leq B\ D \leq C$
shows $A - C \leq B - D$
using *assms*
by (*cases rule*: *ereal3-cases*[*case-product ereal-cases*, *of A B C D*]) *simp-all*

lemma *ereal-mono-minus-cancel*:
fixes $a\ b\ c$ :: *ereal*
shows $c - a \leq c - b \Longrightarrow 0 \leq c \Longrightarrow c < \infty \Longrightarrow b \leq a$
by (*cases a b c rule*: *ereal3-cases*) *auto*

lemma *real-of-ereal-minus*:
fixes $a\ b$ :: *ereal*
shows *real-of-ereal* $(a - b) =$ (*if* $|a| = \infty \vee |b| = \infty$ *then 0 else real-of-ereal a* $-$ *real-of-ereal b*)
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *real-of-ereal-minus'*: $|x| = \infty \longleftrightarrow |y| = \infty \Longrightarrow$ *real-of-ereal* $x -$ *real-of-ereal* $y =$ *real-of-ereal* $(x - y ::$ *ereal*$)$
by(*subst real-of-ereal-minus*) *auto*

lemma *ereal-diff-positive*:
fixes $a\ b$ :: *ereal* **shows** $a \leq b \Longrightarrow 0 \leq b - a$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-between*:
fixes $x\ e$ :: *ereal*
assumes $|x| \neq \infty$ **and** $0 < e$
shows $x - e < x$
and $x < x + e$
using *assms* **by** (*cases x*, *cases e*, *auto*)+

lemma *ereal-minus-eq-PInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = \infty \longleftrightarrow y = -\infty \vee x = \infty$
by (*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-diff-add-eq-diff-diff-swap*:
fixes $x\ y\ z$ :: *ereal*
shows $|y| \neq \infty \Longrightarrow x - (y + z) = x - y - z$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *ereal-diff-add-assoc2*:
fixes $x\ y\ z$ :: *ereal*
shows $x + y - z = x - z + y$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *ereal-add-uminus-conv-diff*: **fixes** $x\ y\ z$ :: *ereal* **shows** $-x + y = y - x$
by (*simp add*: *add.commute minus-ereal-def*)

lemma *ereal-minus-diff-eq*:
fixes $x\ y$ :: *ereal*
shows $[\![x = \infty \longrightarrow y \neq \infty; x = -\infty \longrightarrow y \neq -\infty]\!] \Longrightarrow -(x - y) = y - x$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ediff-le-self* [*simp*]: $x - y \leq (x :: enat)$
by(*cases x y rule*: *enat.exhaust*[*case-product enat.exhaust*]) *simp-all*

lemma *ereal-abs-diff*:
fixes *a b*::*ereal*
shows $abs(a-b) \leq abs\ a + abs\ b$
by (*cases rule*: *ereal2-cases*[*of a b*]) (*auto*)

38.1.6 Division

instantiation *ereal* :: *inverse*
begin

function *inverse-ereal* **where**
inverse (*ereal r*) = (*if r* = *0 then* ∞ *else ereal* (*inverse r*))
| *inverse* (∞::*ereal*) = *0*
| *inverse* ($-\infty$::*ereal*) = *0*
by (*auto intro*: *ereal-cases*)
termination by (*relation* {}) *simp*

definition *x div y* = *x* $*$ *inverse* (*y* :: *ereal*)

instance ..

end

lemma *real-of-ereal-inverse*[*simp*]:
fixes *a* :: *ereal*
shows *real-of-ereal* (*inverse a*) = *1 / real-of-ereal a*
by (*cases a*) (*auto simp*: *inverse-eq-divide*)

lemma *ereal-inverse*[*simp*]:
inverse (*0*::*ereal*) = ∞
inverse (*1*::*ereal*) = *1*
by (*simp-all add*: *one-ereal-def zero-ereal-def*)

lemma *ereal-divide*[*simp*]:
ereal r / ereal p = (*if p* = *0 then ereal r* $*$ ∞ *else ereal* (*r / p*))
unfolding *divide-ereal-def* **by** (*auto simp*: *divide-real-def*)

lemma *ereal-divide-same*[*simp*]:
fixes x :: *ereal*
shows x / x = (*if* $|x| = \infty \vee x = 0$ *then* 0 *else* 1)
by (*cases* x) (*simp-all add*: *divide-real-def divide-ereal-def one-ereal-def*)

lemma *ereal-inv-inv*[*simp*]:
fixes x :: *ereal*
shows *inverse* (*inverse* x) = (*if* $x \neq -\infty$ *then* x *else* ∞)
by (*cases* x) *auto*

lemma *ereal-inverse-minus*[*simp*]:
fixes x :: *ereal*
shows *inverse* (− x) = (*if* $x = 0$ *then* ∞ *else* −*inverse* x)
by (*cases* x) *simp-all*

lemma *ereal-uminus-divide*[*simp*]:
fixes x y :: *ereal*
shows − x / y = − (x / y)
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-divide-Infty*[*simp*]:
fixes x :: *ereal*
shows x / ∞ = 0 x / $-\infty$ = 0
unfolding *divide-ereal-def* **by** *simp-all*

lemma *ereal-divide-one*[*simp*]: x / 1 = (x::*ereal*)
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-divide-ereal*[*simp*]: ∞ / *ereal* r = (*if* 0 ≤ r *then* ∞ *else* $-\infty$)
unfolding *divide-ereal-def* **by** *simp*

lemma *ereal-inverse-nonneg-iff*: 0 ≤ *inverse* (x :: *ereal*) ⟷ 0 ≤ $x \vee x = -\infty$
by (*cases* x) *auto*

lemma *inverse-ereal-ge0I*: 0 ≤ (x :: *ereal*) ⟹ 0 ≤ *inverse* x
by(*cases* x) *simp-all*

lemma *zero-le-divide-ereal*[*simp*]:
fixes a :: *ereal*
assumes 0 ≤ a **and** 0 ≤ b
shows 0 ≤ a / b
by (*simp add*: *assms divide-ereal-def ereal-inverse-nonneg-iff*)

lemma *ereal-le-divide-pos*:
fixes x y z :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y \leq z$ / $x \longleftrightarrow x * y \leq z$
by (*cases rule*: *ereal3-cases*[*of* x y z]) (*auto simp*: *field-simps*)

lemma *ereal-divide-le-pos*:

fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow z \leq x * y$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-le-divide-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow y \leq z\ /\ x \longleftrightarrow z \leq x * y$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-divide-le-neg*:
fixes $x\ y\ z$ :: *ereal*
shows $x < 0 \Longrightarrow x \neq -\infty \Longrightarrow z\ /\ x \leq y \longleftrightarrow x * y \leq z$
by (*cases rule*: *ereal3-cases*[*of x y z*]) (*auto simp*: *field-simps*)

lemma *ereal-inverse-antimono-strict*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x < y \Longrightarrow inverse\ y < inverse\ x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*

lemma *ereal-inverse-antimono*:
fixes $x\ y$ :: *ereal*
shows $0 \leq x \Longrightarrow x \leq y \Longrightarrow inverse\ y \leq inverse\ x$
by (*cases rule*: *ereal2-cases*[*of x y*]) *auto*

lemma *inverse-inverse-Pinfty-iff*[*simp*]:
fixes x :: *ereal*
shows $inverse\ x = \infty \longleftrightarrow x = 0$
by (*cases x*) *auto*

lemma *ereal-inverse-eq-0*:
fixes x :: *ereal*
shows $inverse\ x = 0 \longleftrightarrow x = \infty \vee x = -\infty$
by (*cases x*) *auto*

lemma *ereal-0-gt-inverse*:
fixes x :: *ereal*
shows $0 < inverse\ x \longleftrightarrow x \neq \infty \wedge 0 \leq x$
by (*cases x*) *auto*

lemma *ereal-inverse-le-0-iff*:
fixes x :: *ereal*
shows $inverse\ x \leq 0 \longleftrightarrow x < 0 \vee x = \infty$
by(*cases x*) *auto*

lemma *ereal-divide-eq-0-iff*: $x\ /\ y = 0 \longleftrightarrow x = 0 \vee |y :: ereal| = \infty$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-mult-less-right*:
fixes $a\ b\ c$ :: *ereal*

assumes $b * a < c * a$ $0 < a$ $a < \infty$
shows $b < c$
using *assms*
by (*metis order.asym ereal-mult-strict-left-mono linorder-neqE mult.commute*)

lemma *ereal-mult-divide*:
fixes a b :: *ereal*
shows $0 < b \Longrightarrow b < \infty \Longrightarrow b * (a \mathbin{/} b) = a$
by (*cases a b rule*: *ereal2-cases*) *auto*

lemma *ereal-power-divide*:
fixes x y :: *ereal*
shows $y \neq 0 \Longrightarrow (x \mathbin{/} y) \mathbin{\hat{}} n = x \mathbin{\hat{}} n \mathbin{/} y \mathbin{\hat{}} n$
by (*cases rule*: *ereal2-cases* [*of x y*])
(*auto simp*: *one-ereal-def zero-ereal-def power-divide zero-le-power-eq*)

lemma *divide-pos-0-ereal* [*simp*]: $x > 0 \Longrightarrow x \mathbin{/} (0 :: ereal) = \infty$
by (*simp add*: *divide-ereal-def*)

lemma *divide-pos-0-ereal'* [*simp*]: $x > 0 \Longrightarrow x \mathbin{/} (ereal\ 0) = \infty$
by (*simp add*: *divide-ereal-def*)

lemma *ereal-le-mult-one-interval*:
fixes x y :: *ereal*
assumes *y*: $y \neq -\infty$
assumes *z*: $\bigwedge z.\ 0 < z \Longrightarrow z < 1 \Longrightarrow z * x \leq y$
shows $x \leq y$
proof (*cases x*)
case *PInf*
with *z*[*of 1 / 2*] **show** $x \leq y$
by (*simp add*: *one-ereal-def*)
next
case *r*: (*real r*)
show $x \leq y$
proof (*cases y*)
case *p*: (*real p*)
have $r \leq p$
proof (*rule field-le-mult-one-interval*)
fix z :: *real*
assume $0 < z$ **and** $z < 1$
with *z*[*of ereal z*] **show** $z * r \leq p$
using p r **by** (*auto simp*: *zero-le-mult-iff one-ereal-def*)
qed
then show $x \leq y$
using p r **by** *simp*
qed (*use y* **in** *simp-all*)
qed *simp*

lemma *ereal-divide-right-mono*[*simp*]:

fixes $x\ y\ z :: ereal$
assumes $x \leq y$
and $0 < z$
shows $x\ /\ z \leq y\ /\ z$
using *assms* **by** (*cases x y z rule*: *ereal3-cases*) (*auto intro*: *divide-right-mono*)

lemma *ereal-divide-left-mono*[*simp*]:
fixes $x\ y\ z :: ereal$
assumes $y \leq x$
and $0 < z$
and $0 < x * y$
shows $z\ /\ x \leq z\ /\ y$
using *assms*
by (*cases x y z rule*: *ereal3-cases*)
(*auto intro*: *divide-left-mono simp*: *field-simps zero-less-mult-iff mult-less-0-iff split*: *if-split-asm*)

lemma *ereal-divide-zero-left*[*simp*]:
fixes $a :: ereal$
shows $0\ /\ a = 0$
using *ereal-divide-eq-0-iff* **by** *blast*

lemma *ereal-times-divide-eq-left*[*simp*]:
fixes $a\ b\ c :: ereal$
shows $b\ /\ c * a = b * a\ /\ c$
by (*metis divide-ereal-def mult.assoc mult.commute*)

lemma *ereal-times-divide-eq*: $a * (b\ /\ c :: ereal) = a * b\ /\ c$
by (*metis ereal-times-divide-eq-left mult.commute*)

lemma *ereal-inverse-real* [*simp*]: $|z| \neq \infty \Longrightarrow z \neq 0 \Longrightarrow ereal\ (inverse\ (real\text{-}of\text{-}ereal\ z)) = inverse\ z$
by *auto*

lemma *ereal-inverse-mult*:
$a \neq 0 \Longrightarrow b \neq 0 \Longrightarrow inverse\ (a * (b::ereal)) = inverse\ a * inverse\ b$
by (*cases a*; *cases b*) *auto*

lemma *inverse-eq-infinity-iff-eq-zero* [*simp*]:
$1/(x::ereal) = \infty \longleftrightarrow x = 0$
by (*simp add*: *divide-ereal-def*)

lemma *ereal-distrib-left*:
fixes $a\ b\ c :: ereal$
assumes $a \neq \infty \vee b \neq -\infty$
and $a \neq -\infty \vee b \neq \infty$
and $|c| \neq \infty$
shows $c * (a + b) = c * a + c * b$
by (*metis assms ereal-distrib mult.commute*)

lemma *ereal-distrib-minus-left*:
fixes $a\ b\ c :: ereal$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $c * (a - b) = c * a - c * b$
using *assms ereal-distrib-left ereal-uminus-eq-reorder minus-ereal-def* **by** *auto*

lemma *ereal-distrib-minus-right*:
fixes $a\ b\ c :: ereal$
assumes $a \neq \infty \vee b \neq \infty$
and $a \neq -\infty \vee b \neq -\infty$
and $|c| \neq \infty$
shows $(a - b) * c = a * c - b * c$
by (*metis assms ereal-distrib-minus-left mult.commute*)

38.2 Complete lattice

instantiation *ereal* :: *lattice*
begin

definition [*simp*]: $sup\ x\ y = (max\ x\ y :: ereal)$
definition [*simp*]: $inf\ x\ y = (min\ x\ y :: ereal)$
instance by *standard simp-all*

end

instantiation *ereal* :: *complete-lattice*
begin

definition $bot = (-\infty :: ereal)$
definition $top = (\infty :: ereal)$

definition $Sup\ S = (SOME\ x :: ereal.\ (\forall\, y{\in}S.\ y \leq x) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ y \leq z) \longrightarrow x \leq z))$
definition $Inf\ S = (SOME\ x :: ereal.\ (\forall\, y{\in}S.\ x \leq y) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ z \leq y) \longrightarrow z \leq x))$

lemma *ereal-complete-Sup*:
fixes $S :: ereal\ set$
shows $\exists\, x.\ (\forall\, y{\in}S.\ y \leq x) \wedge (\forall\, z.\ (\forall\, y{\in}S.\ y \leq z) \longrightarrow x \leq z)$
proof (*cases* $\exists\, x.\ \forall\, a{\in}S.\ a \leq ereal\ x$)
case *True*
then obtain y **where** y: $a \leq ereal\ y$ **if** $a{\in}S$ **for** a
by *auto*
then have $\infty \notin S$
by *force*
show *?thesis*

```
  proof (cases S ≠ {−∞} ∧ S ≠ {})
    case True
    with ‹∞ ∉ S› obtain x where x: x ∈ S |x| ≠ ∞
      by auto
    obtain s where s: ∀ x∈ereal −` S. x ≤ s (∀ x∈ereal −` S. x ≤ z) ⟹ s ≤ z
for z
    proof (atomize-elim, rule complete-real)
      show ∃ x. x ∈ ereal −` S
        using x by auto
      show ∃ z. ∀ x∈ereal −` S. x ≤ z
        by (auto dest: y intro!: exI[of - y])
    qed
    show ?thesis
    proof (safe intro!: exI[of - ereal s])
      fix y
      assume y ∈ S
      with s ‹∞ ∉ S› show y ≤ ereal s
        by (cases y) auto
    next
      fix z
      assume ∀ y∈S. y ≤ z
      with ‹S ≠ {−∞} ∧ S ≠ {}› show ereal s ≤ z
        by (cases z) (auto intro!: s)
    qed
  next
    case False
    then show ?thesis
      by (auto intro!: exI[of - −∞])
  qed
next
  case False
  then show ?thesis
    by (fastforce intro!: exI[of - ∞] ereal-top intro: order-trans dest: less-imp-le
simp: not-le)
qed

lemma ereal-complete-uminus-eq:
  fixes S :: ereal set
  shows (∀ y∈uminus`S. y ≤ x) ∧ (∀ z. (∀ y∈uminus`S. y ≤ z) ⟶ x ≤ z)
    ⟷ (∀ y∈S. −x ≤ y) ∧ (∀ z. (∀ y∈S. z ≤ y) ⟶ z ≤ −x)
  by simp (metis ereal-minus-le-minus ereal-uminus-uminus)

lemma ereal-complete-Inf:
  ∃ x. (∀ y∈S::ereal set. x ≤ y) ∧ (∀ z. (∀ y∈S. z ≤ y) ⟶ z ≤ x)
  using ereal-complete-Sup[of uminus ` S]
  unfolding ereal-complete-uminus-eq
  by auto

instance
```

proof
show *Sup* {} = (*bot*::*ereal*)
using *ereal-bot* **by** (*auto simp*: *bot-ereal-def Sup-ereal-def*)
show *Inf* {} = (*top*::*ereal*)
unfolding *top-ereal-def Inf-ereal-def*
using *ereal-infty-less-eq*(*1*) *ereal-less-eq*(*1*) **by** *blast*
show $\bigwedge x$::*ereal*. $\bigwedge A$. $x \in A \Longrightarrow$ *Inf* $A \leq x$
$\bigwedge A$ *z*. ($\bigwedge x$::*ereal*. $x \in A \Longrightarrow z \leq x$) $\Longrightarrow z \leq$ *Inf* *A*
by (*auto intro*: *someI2-ex ereal-complete-Inf simp*: *Inf-ereal-def*)
show $\bigwedge x$::*ereal*. $\bigwedge A$. $x \in A \Longrightarrow x \leq$ *Sup* *A*
$\bigwedge A$ *z*. ($\bigwedge x$::*ereal*. $x \in A \Longrightarrow x \leq z$) $\Longrightarrow$ *Sup* $A \leq z$
by (*auto intro*: *someI2-ex ereal-complete-Sup simp*: *Sup-ereal-def*)
qed

end

instance *ereal* :: *complete-linorder* **..**

instance *ereal* :: *linear-continuum*
proof
show $\exists$ *a b*::*ereal*. $a \neq b$
using *zero-neq-one* **by** *blast*
qed

lemma *min-PInf* [*simp*]: *min* (∞::*ereal*) $x = x$
by (*metis min-top top-ereal-def*)

lemma *min-PInf2* [*simp*]: *min* x (∞::*ereal*) $= x$
by (*metis min-top2 top-ereal-def*)

lemma *max-PInf* [*simp*]: *max* (∞::*ereal*) $x = \infty$
by (*metis max-top top-ereal-def*)

lemma *max-PInf2* [*simp*]: *max* x (∞::*ereal*) $= \infty$
by (*metis max-top2 top-ereal-def*)

lemma *min-MInf* [*simp*]: *min* ($-\infty$::*ereal*) $x = -\infty$
by (*metis min-bot bot-ereal-def*)

lemma *min-MInf2* [*simp*]: *min* x ($-\infty$::*ereal*) $= -\infty$
by (*metis min-bot2 bot-ereal-def*)

lemma *max-MInf* [*simp*]: *max* ($-\infty$::*ereal*) $x = x$
by (*metis max-bot bot-ereal-def*)

lemma *max-MInf2* [*simp*]: *max* x ($-\infty$::*ereal*) $= x$
by (*metis max-bot2 bot-ereal-def*)

38.3 Extended real intervals

lemma *real-greaterThanLessThan-infinity-eq*:
real-of-ereal ‘ {*N*::*ereal*<..<∞} =
(*if N* = ∞ *then* {} *else if N* = −∞ *then UNIV else* {*real-of-ereal N*<..})
by (*force simp*: *real-less-ereal-iff intro*!: *image-eqI*[**where** *x*=*ereal* -] *elim*!: *less-ereal.elims*)

lemma *real-greaterThanLessThan-minus-infinity-eq*:
real-of-ereal ‘ {−∞<..<*N*::*ereal*} =
(*if N* = ∞ *then UNIV else if N* = −∞ *then* {} *else* {..<*real-of-ereal N*})
proof −
have *real-of-ereal* ‘ {−∞<..<*N*::*ereal*} = *uminus* ‘ *real-of-ereal* ‘ {−*N*<..<∞}
by (*auto simp*: *ereal-uminus-less-reorder intro*!: *image-eqI*[**where** *x*=−*x* **for** *x*])
also note *real-greaterThanLessThan-infinity-eq*
finally show *?thesis* **by** (*auto intro*!: *image-eqI*[**where** *x*=−*x* **for** *x*])
qed

lemma *real-greaterThanLessThan-inter*:
real-of-ereal ‘ {*N*<..<*M*::*ereal*} = *real-of-ereal* ‘ {−∞<..<*M*} ∩ *real-of-ereal* ‘ {*N*<..<∞}
by (*force elim*!: *less-ereal.elims*)

lemma *real-atLeastGreaterThan-eq*: *real-of-ereal* ‘ {*N*<..<*M*::*ereal*} =
(*if N* = ∞ *then* {} *else*
if N = −∞ *then*
(*if M* = ∞ *then UNIV*
else if M = −∞ *then* {}
else {..< *real-of-ereal M*})
else if M = −∞ *then* {}
else if M = ∞ *then* {*real-of-ereal N*<..}
else {*real-of-ereal N* <..< *real-of-ereal M*})
proof (*cases M* = −∞ ∨ *M* = ∞ ∨ *N* = −∞ ∨ *N* = ∞)
case *True*
then show *?thesis*
by (*auto simp*: *real-greaterThanLessThan-minus-infinity-eq real-greaterThanLessThan-infinity-eq*
)
next
case *False*
then obtain *p q* **where** *M* = *ereal p N* = *ereal q*
by (*metis MInfty-eq-minfinity ereal.distinct*(*3*) *uminus-ereal.elims*)
moreover have ⋀*x*. ⟦*q* < *x*; *x* < *p*⟧ ⟹ *x* ∈ *real-of-ereal* ‘ {*ereal q*<..<*ereal p*}
by (*metis greaterThanLessThan-iff imageI less-ereal.simps*(*1*) *real-of-ereal.simps*(*1*))
ultimately show *?thesis*
by (*auto elim*!: *less-ereal.elims*)
qed

lemma *real-image-ereal-ivl*:
fixes *a b*::*ereal*
shows
real-of-ereal ‘ {*a*<..<*b*} =

(*if* $a < b$ *then* (*if* $a = -\infty$ *then if* $b = \infty$ *then UNIV else* {..<*real-of-ereal* b}
else if $b = \infty$ *then* {*real-of-ereal* a<..} *else* {*real-of-ereal* a <..< *real-of-ereal* b})
else {})
by (*cases* a; *cases* b; *simp add*: *real-atLeastGreaterThan-eq not-less*)

lemma fixes a b c::*ereal*
shows *not-inftyI*: $a < b \Longrightarrow b < c \Longrightarrow abs\ b \neq \infty$
by *force*

context
fixes r s t::*real*
begin

lemma *interval-Ioo-neq-Ioi*: {r<..<s} $\neq$ {t<..}
by (*simp add*: *set-eq-iff*) (*meson linordered-field-no-ub nless-le order-less-trans*)

lemma *interval-Ioo-neq-Iio*: {r<..<s} $\neq$ {..<t}
by (*simp add*: *set-eq-iff*) (*meson linordered-field-no-lb order-less-irrefl order-less-trans*)

lemma *interval-neq-ioo-UNIV*: {r<..<s} $\neq$ *UNIV*
and *interval-Ioi-neq-UNIV*: {r<..} $\neq$ *UNIV*
and *interval-Iio-neq-UNIV*: {..<r} $\neq$ *UNIV*
by *auto*

lemma *interval-Ioi-neq-Iio*: {r<..} $\neq$ {..<s}
by (*simp add*: *set-eq-iff*) (*meson lt-ex order-less-irrefl order-less-trans*)

lemma *interval-empty-neq-Ioi*: {} $\neq$ {r<..}
and *interval-empty-neq-Iio*: {} $\neq$ {..<r}
by (*auto simp*: *set-eq-iff linordered-field-no-ub linordered-field-no-lb*)

end

lemmas *interval-neqs* = *interval-Ioo-neq-Ioi interval-Ioo-neq-Iio*
interval-neq-ioo-UNIV interval-Ioi-neq-Iio
interval-Ioi-neq-UNIV interval-Iio-neq-UNIV
interval-empty-neq-Ioi interval-empty-neq-Iio

lemma *greaterThanLessThan-eq-iff*:
fixes r s t u::*real*
shows ({r<..<s} = {t<..<u}) = ($r \geq s \wedge u \leq t \vee r = t \wedge s = u$)
by (*metis cInf-greaterThanLessThan cSup-greaterThanLessThan greaterThanLessThan-empty-iff not-le*)

lemma *real-of-ereal-image-greaterThanLessThan-iff*:
real-of-ereal ' {a <..< b} = *real-of-ereal* ' {c <..< d} $\longleftrightarrow$ ($a \geq b \wedge c \geq d \vee a = c \wedge b = d$)
unfolding *real-atLeastGreaterThan-eq*
by (*cases* a; *cases* b; *cases* c; *cases* d;

simp add: *greaterThanLessThan-eq-iff interval-neqs interval-neqs*[*symmetric*])

lemma *uminus-image-real-of-ereal-image-greaterThanLessThan*:
uminus ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{-u <..< -l\}$
by (*force simp*: *algebra-simps ereal-less-uminus-reorder*
ereal-uminus-less-reorder intro: *image-eqI*[**where** $x{=}-x$ **for** x])

lemma *add-image-real-of-ereal-image-greaterThanLessThan*:
$(+)$ c ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{c + l <..< c + u\}$
apply *safe*
subgoal for x
using *ereal-less-add*[*of c*]
by (*force simp*: *real-of-ereal-add add.commute*)
subgoal for - x
by (*force simp*: *add.commute real-of-ereal-minus ereal-minus-less ereal-less-minus*
intro: *image-eqI*[**where** $x{=}x - c$])
done

lemma *add2-image-real-of-ereal-image-greaterThanLessThan*:
$(\lambda x.\ x + c)$ ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{l + c <..< u + c\}$
using *add-image-real-of-ereal-image-greaterThanLessThan*[*of c l u*]
by (*metis add.commute image-cong*)

lemma *minus-image-real-of-ereal-image-greaterThanLessThan*:
$(-)$ c ‘ *real-of-ereal* ‘ $\{l <..< u\}$ = *real-of-ereal* ‘ $\{c - u <..< c - l\}$
(**is** *?l* = *?r*)
proof −
have *?l* = $(+)$ c ‘ *uminus* ‘ *real-of-ereal* ‘ $\{l <..< u\}$ **by** *auto*
also note *uminus-image-real-of-ereal-image-greaterThanLessThan*
also note *add-image-real-of-ereal-image-greaterThanLessThan*
finally show *?thesis* **by** (*simp add*: *minus-ereal-def*)
qed

lemma *real-ereal-bound-lemma-up*:
assumes $s \in$ *real-of-ereal* ‘ $\{a<..<b\}$
assumes $t \notin$ *real-of-ereal* ‘ $\{a<..<b\}$
assumes $s \leq t$
shows $b \neq \infty$
proof (*cases b*)
case *PInf*
then show *?thesis*
using *assms*
by (*metis UNIV-I empty-iff greaterThan-iff order-less-le-trans real-image-ereal-ivl*)
qed *auto*

lemma *real-ereal-bound-lemma-down*:
assumes *s*: $s \in$ *real-of-ereal* ‘ $\{a<..<b\}$
and *t*: $t \notin$ *real-of-ereal* ‘ $\{a<..<b\}$
and $t \leq s$

shows $a \neq -\infty$
by (*metis UNIV-I assms empty-iff lessThan-iff order-le-less-trans real-greaterThanLessThan-minus-infinity-eq*)

38.4 Topological space

instantiation *ereal* :: *linear-continuum-topology*
begin

definition *open-ereal* :: *ereal set* ⇒ *bool* **where**
open-ereal-generated: *open-ereal* = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
by *standard* (*simp add*: *open-ereal-generated*)

end

lemma *continuous-on-ereal*[*continuous-intros*]:
assumes *f*: *continuous-on s f* **shows** *continuous-on s* (λ*x*. *ereal* (*f x*))
by (*rule continuous-on-compose2* [*OF continuous-onI-mono*[*of ereal UNIV*] *f*]) *auto*

lemma *tendsto-ereal*[*tendsto-intros*, *simp*, *intro*]: (*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ereal* (*f x*)) ⟶ *ereal x*) *F*
using *isCont-tendsto-compose*[*of x ereal f F*] *continuous-on-ereal*[*of UNIV* λ*x*. *x*]
by (*simp add*: *continuous-on-eq-continuous-at*)

lemma *tendsto-uminus-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes (*f* ⟶ *x*) *F*
shows ((λ*x*. − *f x*::*ereal*) ⟶ − *x*) *F*
proof (*rule tendsto-compose*[*OF order-tendstoI assms*])
show ⋀*a*. *a* < − *x* ⟹ ∀⇩*F* *x in at x*. *a* < − *x*
by (*metis ereal-less-uminus-reorder eventually-at-topological lessThan-iff open-lessThan*)
show ⋀*a*. − *x* < *a* ⟹ ∀⇩*F* *x in at x*. − *x* < *a*
by (*metis ereal-uminus-reorder*(*2*) *eventually-at-topological greaterThan-iff open-greaterThan*)
qed

lemma *at-infty-ereal-eq-at-top*: *at* ∞ = *filtermap ereal at-top*
proof −
have ⋀*P b*. ∀ *z*. *b* ≤ *z* ∧ *b* ≠ *z* ⟶ *P* (*ereal z*) ⟹ ∃ *N*. ∀ *n*≥*N*. *P* (*ereal n*)
by (*metis gt-ex order-less-le order-less-le-trans*)
then show *?thesis*
unfolding *filter-eq-iff eventually-at-filter eventually-at-top-linorder eventually-filtermap top-ereal-def*[*symmetric*]
apply (*subst eventually-nhds-top*[*of 0*])
apply (*auto simp*: *top-ereal-def less-le ereal-all-split ereal-ex-split*)

```
    done
qed

lemma ereal-Lim-uminus: (f ⟶ f0) net ⟷ ((λx. − f x::ereal) ⟶ − f0) net
  using tendsto-uminus-ereal[of f f0 net] tendsto-uminus-ereal[of λx. − f x − f0 net]
  by auto

lemma ereal-divide-less-iff: 0 < (c::ereal) ⟹ c < ∞ ⟹ a / c < b ⟷ a < b * c
  by (cases a b c rule: ereal3-cases) (auto simp: field-simps)

lemma ereal-less-divide-iff: 0 < (c::ereal) ⟹ c < ∞ ⟹ a < b / c ⟷ a * c < b
  by (cases a b c rule: ereal3-cases) (auto simp: field-simps)

lemma tendsto-cmult-ereal[tendsto-intros, simp, intro]:
  assumes c: |c| ≠ ∞ and f: (f ⟶ x) F
  shows ((λx. c * f x::ereal) ⟶ c * x) F
proof −
  have ∗: ((λx. c * f x::ereal) ⟶ c * x) F if 0 < c c < ∞ for c :: ereal
    using that
    apply (intro tendsto-compose[OF - f])
    apply (auto intro!: order-tendstoI simp: eventually-at-topological)
     apply (rule-tac x={a/c <..} in exI)
     apply (auto split: ereal.split simp: ereal-divide-less-iff mult.commute) []
    apply (rule-tac x={..< a/c} in exI)
    apply (auto split: ereal.split simp: ereal-less-divide-iff mult.commute) []
    done
  have ((0 < c ∧ c < ∞) ∨ (−∞ < c ∧ c < 0) ∨ c = 0)
    using c by (cases c) auto
  then show ?thesis
  proof (elim disjE conjE)
    assume −∞ < c c < 0
    then have 0 < − c − c < ∞
      by (auto simp: ereal-uminus-reorder ereal-less-uminus-reorder[of 0])
    then have ((λx. (− c) * f x) ⟶ (− c) * x) F
      by (rule ∗)
    from tendsto-uminus-ereal[OF this] show ?thesis
      by simp
  qed (auto intro!: ∗)
qed

lemma tendsto-cmult-ereal-not-0[tendsto-intros, simp, intro]:
  assumes x ≠ 0 and f: (f ⟶ x) F
  shows ((λx. c * f x::ereal) ⟶ c * x) F
proof cases
  assume |c| = ∞
```

show *?thesis*
proof (*rule filterlim-cong*[*THEN iffD1*, *OF refl refl* - *tendsto-const*])
have $0 < x \vee x < 0$
using ‹$x \neq 0$› **by** (*auto simp*: *neq-iff*)
then show *eventually* ($\lambda x'$. $c * x = c * f\, x'$) *F*
proof
assume $0 < x$ **from** *order-tendstoD*(*1*)[*OF f this*] **show** *?thesis*
by *eventually-elim* (*use* ‹*0*<*x*› ‹$|c| = \infty$› **in** *auto*)
next
assume $x < 0$ **from** *order-tendstoD*(*2*)[*OF f this*] **show** *?thesis*
by *eventually-elim* (*use* ‹*x*<*0*› ‹$|c| = \infty$› **in** *auto*)
qed
qed
qed (*rule tendsto-cmult-ereal*[*OF* - *f*])

lemma *tendsto-cadd-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $y \neq -\infty$ $x \neq -\infty$ **and** *f*: ($f \longrightarrow x$) *F*
shows ((λx. $f\, x + y$::*ereal*) $\longrightarrow x + y$) *F*
apply (*intro tendsto-compose*[*OF* - *f*])
apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
apply (*rule-tac x*={$a - y$ <..} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-minus-less-iff c*) []
apply (*rule-tac x*={..< $a - y$} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-less-minus-iff c*) []
done

lemma *tendsto-add-left-ereal*[*tendsto-intros*, *simp*, *intro*]:
assumes *c*: $|y| \neq \infty$ **and** *f*: ($f \longrightarrow x$) *F*
shows ((λx. $f\, x + y$::*ereal*) $\longrightarrow x + y$) *F*
apply (*intro tendsto-compose*[*OF* - *f*])
apply (*auto intro*!: *order-tendstoI simp*: *eventually-at-topological*)
apply (*rule-tac x*={$a - y$ <..} **in** *exI*)
apply (*insert c*, *auto split*: *ereal.split simp*: *ereal-minus-less-iff*) []
apply (*rule-tac x*={..< $a - y$} **in** *exI*)
apply (*auto split*: *ereal.split simp*: *ereal-less-minus-iff c*) []
done

lemma *continuous-at-ereal*[*continuous-intros*]: *continuous F f* $\Longrightarrow$ *continuous F* (λx. *ereal* (*f x*))
unfolding *continuous-def* **by** *auto*

lemma *ereal-Sup*:
assumes $*$: $|SUP\ a{\in}A.\ ereal\ a| \neq \infty$
shows *ereal* (*Sup A*) = (*SUP a*∈*A*. *ereal a*)
proof (*rule continuous-at-Sup-mono*)
obtain *r* **where** *r*: *ereal r* = (*SUP a*∈*A*. *ereal a*) $A \neq \{\}$
using $*$ **by** (*force simp*: *bot-ereal-def*)
then show *bdd-above A* $A \neq \{\}$
by (*auto intro*!: *SUP-upper bdd-aboveI*[*of* - *r*] *simp flip*: *ereal-less-eq*)

qed (*auto simp*: *mono-def continuous-at-imp-continuous-at-within continuous-at-ereal*)

lemma *ereal-SUP*: $|SUP\ a{\in}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (SUP\ a{\in}A.\ f\ a) = (SUP\ a{\in}A.\ ereal\ (f\ a))$
by (*simp add*: *ereal-Sup image-comp*)

lemma *ereal-Inf*:
assumes $*$: $|INF\ a{\in}A.\ ereal\ a| \neq \infty$
shows $ereal\ (Inf\ A) = (INF\ a{\in}A.\ ereal\ a)$
proof (*rule continuous-at-Inf-mono*)
obtain r **where** r: $ereal\ r = (INF\ a{\in}A.\ ereal\ a)\ A \neq \{\}$
using $*$ **by** (*force simp*: *top-ereal-def*)
then show *bdd-below A* $A \neq \{\}$
by (*auto intro*!: *INF-lower bdd-belowI*[*of - r*] *simp flip*: *ereal-less-eq*)
qed (*auto simp*: *mono-def continuous-at-imp-continuous-at-within continuous-at-ereal*)

lemma *ereal-Inf'*:
assumes $*$: *bdd-below A* $A \neq \{\}$
shows $ereal\ (Inf\ A) = (INF\ a{\in}A.\ ereal\ a)$
proof (*rule ereal-Inf*)
from $*$ **obtain** $l\ u$ **where** $x \in A \Longrightarrow l \leq x\ u \in A$ **for** x
by (*auto simp*: *bdd-below-def*)
then have $l \leq (INF\ x{\in}A.\ ereal\ x)\ (INF\ x{\in}A.\ ereal\ x) \leq u$
by (*auto intro*!: *INF-greatest INF-lower*)
then show $|INF\ a{\in}A.\ ereal\ a| \neq \infty$
by *auto*
qed

lemma *ereal-INF*: $|INF\ a{\in}A.\ ereal\ (f\ a)| \neq \infty \Longrightarrow ereal\ (INF\ a{\in}A.\ f\ a) = (INF\ a{\in}A.\ ereal\ (f\ a))$
by (*simp add*: *ereal-Inf image-comp*)

lemma *ereal-Sup-uminus-image-eq*: *Sup* (*uminus* ' *S*::*ereal set*) = − *Inf S*
by (*auto intro*!: *SUP-eqI*
simp: *Ball-def*[*symmetric*] *ereal-uminus-le-reorder le-Inf-iff*
intro!: *complete-lattice-class.Inf-lower2*)

lemma *ereal-SUP-uminus-eq*:
fixes $f :: 'a \Rightarrow ereal$
shows $(SUP\ x{\in}S.\ uminus\ (f\ x)) = -\ (INF\ x{\in}S.\ f\ x)$
using *ereal-Sup-uminus-image-eq* [*of f* ' *S*] **by** (*simp add*: *image-comp*)

lemma *ereal-inj-on-uminus*[*intro*, *simp*]: *inj-on uminus* (*A* :: *ereal set*)
by (*auto intro*!: *inj-onI*)

lemma *ereal-Inf-uminus-image-eq*: *Inf* (*uminus* ' *S*::*ereal set*) = − *Sup S*
using *ereal-Sup-uminus-image-eq*[*of uminus* ' *S*] **by** *simp*

lemma *ereal-INF-uminus-eq*:

fixes $f :: 'a \Rightarrow ereal$
shows $(INF\ x{\in}S.\ -\ f\ x) = -\ (SUP\ x{\in}S.\ f\ x)$
using *ereal-Inf-uminus-image-eq* [*of f ' S*] **by** (*simp add*: *image-comp*)

lemma *ereal-SUP-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |Sup\ (f\ `\ A)| \neq \infty$
using *SUP-upper2*[*of - A l f*] *SUP-least*[*of A f u*]
by (*cases Sup* (*f ' A*)) *auto*

lemma *ereal-INF-not-infty*:
fixes $f :: - \Rightarrow ereal$
shows $A \neq \{\} \Longrightarrow l \neq -\infty \Longrightarrow u \neq \infty \Longrightarrow \forall\, a{\in}A.\ l \leq f\ a \wedge f\ a \leq u \Longrightarrow |Inf\ (f\ `\ A)| \neq \infty$
using *INF-lower2*[*of - A f u*] *INF-greatest*[*of A l f*]
by (*cases Inf* (*f ' A*)) *auto*

lemma *ereal-image-uminus-shift*:
fixes $X\ Y :: ereal\ set$
shows $uminus\ `\ X = Y \longleftrightarrow X = uminus\ `\ Y$
by (*metis ereal-minus-minus-image*)

lemma *Sup-eq-MInfty*:
fixes $S :: ereal\ set$
shows $Sup\ S = -\infty \longleftrightarrow S = \{\} \vee S = \{-\infty\}$
unfolding *bot-ereal-def*[*symmetric*] **by** *auto*

lemma *Inf-eq-PInfty*:
fixes $S :: ereal\ set$
shows $Inf\ S = \infty \longleftrightarrow S = \{\} \vee S = \{\infty\}$
using *Sup-eq-MInfty*[*of uminus'S*]
unfolding *ereal-Sup-uminus-image-eq ereal-image-uminus-shift* **by** *simp*

lemma *Inf-eq-MInfty*:
fixes $S :: ereal\ set$
shows $-\infty \in S \Longrightarrow Inf\ S = -\infty$
unfolding *bot-ereal-def*[*symmetric*] **by** *auto*

lemma *Sup-eq-PInfty*:
fixes $S :: ereal\ set$
shows $\infty \in S \Longrightarrow Sup\ S = \infty$
unfolding *top-ereal-def*[*symmetric*] **by** *auto*

lemma *not-MInfty-nonneg*[*simp*]: $0 \leq (x{::}ereal) \Longrightarrow x \neq -\infty$
by *auto*

lemma *Sup-ereal-close*:
fixes $e :: ereal$

assumes $0 < e$
and S: $|Sup\ S| \neq \infty\ S \neq \{\}$
shows $\exists x{\in}S.\ Sup\ S - e < x$
using *assms* **by** (*cases e*) (*auto intro*!: *less-Sup-iff*[*THEN iffD1*])

lemma *Inf-ereal-close*:
fixes $e :: ereal$
assumes $|Inf\ X| \neq \infty$
and $0 < e$
shows $\exists x{\in}X.\ x < Inf\ X + e$
by (*meson Inf-less-iff assms ereal-between*(*2*))

lemma *SUP-PInfty*:
$(\bigwedge n{::}nat.\ \exists i{\in}A.\ ereal\ (real\ n) \leq f\ i) \Longrightarrow (SUP\ i{\in}A.\ f\ i :: ereal) = \infty$
by (*meson SUP-upper2 less-PInf-Ex-of-nat linorder-not-less*)

lemma *SUP-nat-Infty*: $(SUP\ i.\ ereal\ (real\ i)) = \infty$
by (*rule SUP-PInfty*) *auto*

lemma *SUP-ereal-add-left*:
assumes $I \neq \{\}\ c \neq -\infty$
shows $(SUP\ i{\in}I.\ f\ i + c :: ereal) = (SUP\ i{\in}I.\ f\ i) + c$
proof (*cases* $(SUP\ i{\in}I.\ f\ i) = -\infty$)
case *True*
then have $\bigwedge i.\ i \in I \Longrightarrow f\ i = -\infty$
unfolding *Sup-eq-MInfty* **by** *auto*
with *True* **show** *?thesis*
by (*cases c*) (*auto simp*: ‹$I \neq \{\}$›)
next
case *False*
then show *?thesis*
by (*subst continuous-at-Sup-mono*[**where** $f{=}\lambda x.\ x + c$])
(*auto simp*: *continuous-at-imp-continuous-at-within continuous-at mono-def add-mono* ‹$I \neq \{\}$› ‹$c \neq -\infty$› *image-comp*)
qed

lemma *SUP-ereal-add-right*:
fixes $c :: ereal$
shows $I \neq \{\} \Longrightarrow c \neq -\infty \Longrightarrow (SUP\ i{\in}I.\ c + f\ i) = c + (SUP\ i{\in}I.\ f\ i)$
using *SUP-ereal-add-left*[*of I c f*] **by** (*simp add*: *add.commute*)

lemma *SUP-ereal-minus-right*:
assumes $I \neq \{\}\ c \neq -\infty$
shows $(SUP\ i{\in}I.\ c - f\ i :: ereal) = c - (INF\ i{\in}I.\ f\ i)$
using *SUP-ereal-add-right*[*OF assms*, *of* $\lambda i.\ -f\ i$]
by (*simp add*: *ereal-SUP-uminus-eq minus-ereal-def*)

lemma *SUP-ereal-minus-left*:

assumes $I \neq \{\}$ $c \neq \infty$
shows $(SUP\ i{\in}I.\ f\ i - c :: ereal) = (SUP\ i{\in}I.\ f\ i) - c$
using *SUP-ereal-add-left*[*OF* ‹$I \neq \{\}$›, *of* $-c$ f] **by** (*simp add*: ‹$c \neq \infty$› *minus-ereal-def*)

lemma *INF-ereal-minus-right*:
assumes $I \neq \{\}$ **and** $|c| \neq \infty$
shows $(INF\ i{\in}I.\ c - f\ i) = c - (SUP\ i{\in}I.\ f\ i :: ereal)$
proof −
have $*$: $(-\ c) + b = -\ (c - b)$ **for** b
using ‹$|c| \neq \infty$› **by** (*cases c b rule*: *ereal2-cases*) *auto*
show *?thesis*
using *SUP-ereal-add-right*[*OF* ‹$I \neq \{\}$›, *of* $-c$ f] ‹$|c| \neq \infty$›
by (*auto simp*: $*$ *ereal-SUP-uminus-eq*)
qed

lemma *SUP-ereal-le-addI*:
fixes $f :: 'i \Rightarrow ereal$
assumes $\bigwedge i.\ f\ i + y \leq z$ **and** $y \neq -\infty$
shows $Sup\ (f\ `\ UNIV) + y \leq z$
by (*metis SUP-ereal-add-left SUP-least UNIV-not-empty assms*)

lemma *SUP-combine*:
fixes $f :: 'a :: semilattice\text{-}sup \Rightarrow 'a :: semilattice\text{-}sup \Rightarrow 'b :: complete\text{-}lattice$
assumes *mono*: $\bigwedge a\ b\ c\ d.\ a \leq b \Longrightarrow c \leq d \Longrightarrow f\ a\ c \leq f\ b\ d$
shows $(SUP\ i{\in}UNIV.\ SUP\ j{\in}UNIV.\ f\ i\ j) = (SUP\ i.\ f\ i\ i)$
proof (*rule antisym*)
show $(SUP\ i\ j.\ f\ i\ j) \leq (SUP\ i.\ f\ i\ i)$
by (*rule SUP-least SUP-upper2*[**where** *i*=*sup i j* **for** *i j*] *UNIV-I mono sup-ge1 sup-ge2*)+
show $(SUP\ i.\ f\ i\ i) \leq (SUP\ i\ j.\ f\ i\ j)$
by (*rule SUP-least SUP-upper2 UNIV-I mono order-refl*)+
qed

lemma *SUP-ereal-add*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *inc*: *incseq f incseq g*
and *pos*: $\bigwedge i.\ f\ i \neq -\infty$ $\bigwedge i.\ g\ i \neq -\infty$
shows $(SUP\ i.\ f\ i + g\ i) = Sup\ (f\ `\ UNIV) + Sup\ (g\ `\ UNIV)$
proof −
have $\bigwedge i\ j\ k\ l.\ [\![i \leq j;\ k \leq l]\!] \Longrightarrow f\ i + g\ k \leq f\ j + g\ l$
by (*meson add-mono inc incseq-def*)
then have $(SUP\ i.\ f\ i + g\ i) = (SUP\ i\ j.\ f\ i + g\ j)$
by (*simp add*: *SUP-combine*)
also have $... = (SUP\ i\ j.\ g\ j + f\ i)$
by (*simp add*: *add.commute*)
also have $... = (SUP\ i.\ Sup\ (range\ g) + f\ i)$
by (*simp add*: *SUP-ereal-add-left pos*(*1*))
also have $... = (SUP\ i.\ f\ i + Sup\ (range\ g))$

```
    by (simp add: add.commute)
  also have ... =  Sup (f ` UNIV) + Sup (g ` UNIV)
    by (simp add: SUP-eq-iff SUP-ereal-add-left pos(2))
  finally show ?thesis .
qed

lemma INF-eq-minf: (INF i∈I. f i::ereal) ≠ −∞ ⟷ (∃ b>−∞. ∀ i∈I. b ≤ f i)
  unfolding bot-ereal-def[symmetric] INF-eq-bot-iff by (auto simp: not-less)

lemma INF-ereal-add-left:
  assumes I ≠ {} c ≠ −∞ ⋀x. x ∈ I ⟹ 0 ≤ f x
  shows (INF i∈I. f i + c :: ereal) = (INF i∈I. f i) + c
proof −
  have (INF i∈I. f i) ≠ −∞
    unfolding INF-eq-minf using assms by (intro exI[of - 0]) auto
  then show ?thesis
    by (subst continuous-at-Inf-mono[where f=λx. x + c])
     (auto simp: mono-def add-mono ‹I ≠ {}› ‹c ≠ −∞› continuous-at-imp-continuous-at-within
        continuous-at image-comp)
qed

lemma INF-ereal-add-right:
  assumes I ≠ {} c ≠ −∞ ⋀x. x ∈ I ⟹ 0 ≤ f x
  shows (INF i∈I. c + f i :: ereal) = c + (INF i∈I. f i)
  using INF-ereal-add-left[OF assms] by (simp add: ac-simps)

lemma INF-ereal-add-directed:
  fixes f g :: 'a ⇒ ereal
  assumes nonneg: ⋀i. i ∈ I ⟹ 0 ≤ f i ⋀i. i ∈ I ⟹ 0 ≤ g i
  assumes directed: ⋀i j. i ∈ I ⟹ j ∈ I ⟹ ∃ k∈I. f i + g j ≥ f k + g k
  shows (INF i∈I. f i + g i) = (INF i∈I. f i) + (INF i∈I. g i)
proof (cases  I = {})
  case False
  show ?thesis
  proof (rule antisym)
    show (INF i∈I. f i) + (INF i∈I. g i) ≤ (INF i∈I. f i + g i)
      by (rule INF-greatest; intro add-mono INF-lower)
  next
    have (INF i∈I. f i + g i) ≤ (INF i∈I. (INF j∈I. f i + g j))
      using directed by (intro INF-greatest) (blast intro: INF-lower2)
    also have ... = (INF i∈I. f i + (INF i∈I. g i))
      using nonneg ‹I ≠ {}› by (auto simp: INF-ereal-add-right)
    also have ... = (INF i∈I. f i) + (INF i∈I. g i)
      using nonneg by (intro INF-ereal-add-left ‹I ≠ {}›) (auto simp: INF-eq-minf
intro!: exI[of - 0])
    finally show (INF i∈I. f i + g i) ≤ (INF i∈I. f i) + (INF i∈I. g i) .
  qed
qed (simp add: top-ereal-def)
```

lemma *INF-ereal-add*:
fixes $f :: nat \Rightarrow ereal$
assumes *decseq f decseq g*
and *fin*: $\bigwedge i.\ f\ i \neq \infty\ \bigwedge i.\ g\ i \neq \infty$
shows $(INF\ i.\ f\ i + g\ i) = Inf\ (f\ `\ UNIV) + Inf\ (g\ `\ UNIV)$
proof −
have *INF-less*: $(INF\ i.\ f\ i) < \infty\ (INF\ i.\ g\ i) < \infty$
using *assms* **unfolding** *INF-less-iff* **by** *auto*
have $*$: $-\ ((-\ a) + (-\ b)) = a + b$ **if** $a \neq \infty\ b \neq \infty$ **for** $a\ b :: ereal$
using *that* **by** (*cases a b rule*: *ereal2-cases*) *auto*
have $(INF\ i.\ f\ i + g\ i) = (INF\ i.\ -\ ((-\ f\ i) + (-\ g\ i)))$
by (*simp add*: *fin* $*$)
also have $\ldots = Inf\ (f\ `\ UNIV) + Inf\ (g\ `\ UNIV)$
unfolding *ereal-INF-uminus-eq*
using *assms INF-less*
by (*subst SUP-ereal-add*) (*auto simp*: *ereal-SUP-uminus-eq fin* $*$)
finally show *?thesis* **.**
qed

lemma *SUP-ereal-add-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *incseq f incseq g*
and $\bigwedge i.\ 0 \leq f\ i\ \bigwedge i.\ 0 \leq g\ i$
shows $(SUP\ i.\ f\ i + g\ i) = Sup\ (f\ `\ UNIV) + Sup\ (g\ `\ UNIV)$
by (*simp add*: *SUP-ereal-add assms*)

lemma *SUP-ereal-sum*:
fixes $f\ g :: 'a \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge n.\ n \in A \Longrightarrow incseq\ (f\ n)$
and *pos*: $\bigwedge n\ i.\ n \in A \Longrightarrow 0 \leq f\ n\ i$
shows $(SUP\ i.\ \sum n{\in}A.\ f\ n\ i) = (\sum n{\in}A.\ Sup\ ((f\ n)\ `\ UNIV))$
using *assms*
by (*induction A rule*: *infinite-finite-induct*) (*auto simp*: *incseq-sumI2 sum-nonneg SUP-ereal-add-pos*)

lemma *SUP-ereal-mult-left*:
fixes $f :: 'a \Rightarrow ereal$
assumes $I \neq \{\}$
assumes *f*: $\bigwedge i.\ i \in I \Longrightarrow 0 \leq f\ i$ **and** *c*: $0 \leq c$
shows $(SUP\ i{\in}I.\ c * f\ i) = c * (SUP\ i{\in}I.\ f\ i)$
proof (*cases* $(SUP\ i \in I.\ f\ i) = 0$)
case *True*
then have $\bigwedge i.\ i \in I \Longrightarrow f\ i = 0$
by (*metis SUP-upper f antisym*)
with *True* **show** *?thesis*
by *simp*
next
case *False*
then show *?thesis*

```
    by (subst continuous-at-Sup-mono[where f=λx. c * x])
       (auto simp: mono-def continuous-at continuous-at-imp-continuous-at-within
‹I ≠ {}› image-comp
             intro!: ereal-mult-left-mono c)
qed

lemma countable-approach:
  fixes x :: ereal
  assumes x ≠ −∞
  shows ∃f. incseq f ∧ (∀i::nat. f i < x) ∧ (f ⟶ x)
proof (cases x)
  case (real r)
  moreover have (λn. r − inverse (real (Suc n))) ⟶ r − 0
    by (intro tendsto-intros LIMSEQ-inverse-real-of-nat)
  ultimately show ?thesis
    by (intro exI[of - λn. x − inverse (Suc n)]) (auto simp: incseq-def)
next
  case PInf with LIMSEQ-SUP[of λn::nat. ereal (real n)] show ?thesis
    by (intro exI[of - λn. ereal (real n)]) (auto simp: incseq-def SUP-nat-Infty)
qed (simp add: assms)

lemma Sup-countable-SUP:
  assumes A ≠ {}
  shows ∃f::nat ⇒ ereal. incseq f ∧ range f ⊆ A ∧ Sup A = (SUP i. f i)
proof cases
  assume Sup A = −∞
  with ‹A ≠ {}› have A = {−∞}
    by (auto simp: Sup-eq-MInfty)
  then show ?thesis
    by (auto intro!: exI[of - λ-. −∞] simp: bot-ereal-def)
next
  assume Sup A ≠ −∞
  then obtain l where incseq l and l: l i < Sup A and l-Sup: l ⟶ Sup A
for i :: nat
    by (auto dest: countable-approach)

  have ∃f. ∀n. (f n ∈ A ∧ l n ≤ f n) ∧ (f n ≤ f (Suc n)) (is ∃f. ?P f)
  proof (rule dependent-nat-choice)
    show ∃x. x ∈ A ∧ l 0 ≤ x
      using l[of 0] by (auto simp: less-Sup-iff)
  next
    fix x n assume x ∈ A ∧ l n ≤ x
    moreover from l[of Suc n] obtain y where y ∈ A l (Suc n) < y
      by (auto simp: less-Sup-iff)
    ultimately show ∃y. (y ∈ A ∧ l (Suc n) ≤ y) ∧ x ≤ y
      by (auto intro!: exI[of - max x y] split: split-max)
  qed
  then obtain f where f: ?P f ..
  then have range f ⊆ A incseq f
```

by (*auto simp*: *incseq-Suc-iff*)
then have (*SUP i. f i*) = *Sup A*
by (*meson LIMSEQ-SUP LIMSEQ-le Sup-subset-mono f l-Sup order-class.order-eq-iff*)
then show *?thesis*
by (*metis ‹incseq f› ‹range f ⊆ A›*)
qed

lemma *Inf-countable-INF*:
assumes *A* ≠ {} **shows** ∃*f*::*nat* ⇒ *ereal. decseq f* ∧ *range f* ⊆ *A* ∧ *Inf A* = (*INF i. f i*)
proof −
obtain *f* **where** *incseq f range f* ⊆ *uminus‘A Sup* (*uminus‘A*) = (*SUP i. f i*)
using *Sup-countable-SUP*[*of uminus ‘ A*] ‹*A* ≠ {}› **by** *auto*
then show *?thesis*
by (*intro exI*[*of - λx. − f x*])
(*auto simp*: *ereal-Sup-uminus-image-eq ereal-INF-uminus-eq eq-commute*[*of − -*])
qed

lemma *SUP-countable-SUP*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ereal. range f* ⊆ *g‘A* ∧ *Sup* (*g ‘ A*) = *Sup* (*f ‘ UNIV*)
using *Sup-countable-SUP* [*of g‘A*] **by** *auto*

38.5 Relation to *enat*

definition *ereal-of-enat n* = (*case n of enat n* ⇒ *ereal* (*real n*) | ∞ ⇒ ∞)

declare [[*coercion ereal-of-enat* :: *enat* ⇒ *ereal*]]
declare [[*coercion* (λ*n. ereal* (*real n*)) :: *nat* ⇒ *ereal*]]

lemma *ereal-of-enat-simps*[*simp*]:
ereal-of-enat (*enat n*) = *ereal n*
ereal-of-enat ∞ = ∞
by (*simp-all add*: *ereal-of-enat-def*)

lemma *ereal-of-enat-le-iff*[*simp*]: *ereal-of-enat m* ≤ *ereal-of-enat n* ⟷ *m* ≤ *n*
by (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-less-iff*[*simp*]: *ereal-of-enat m* < *ereal-of-enat n* ⟷ *m* < *n*
by (*cases m n rule*: *enat2-cases*) *auto*

lemma *numeral-le-ereal-of-enat-iff*[*simp*]: *numeral m* ≤ *ereal-of-enat n* ⟷ *numeral m* ≤ *n*
by (*cases n*) (*auto*)

lemma *numeral-less-ereal-of-enat-iff*[*simp*]: *numeral m* < *ereal-of-enat n* ⟷ *numeral m* < *n*
by (*cases n*) *auto*

lemma *ereal-of-enat-ge-zero-cancel-iff*[*simp*]: $0 \leq$ *ereal-of-enat* $n \longleftrightarrow 0 \leq n$
by (*cases n*) (*auto simp flip*: *enat-0*)

lemma *ereal-of-enat-gt-zero-cancel-iff*[*simp*]: $0 <$ *ereal-of-enat* $n \longleftrightarrow 0 < n$
by (*cases n*) (*auto simp flip*: *enat-0*)

lemma *ereal-of-enat-zero*[*simp*]: *ereal-of-enat* $0 = 0$
by (*auto simp flip*: *enat-0*)

lemma *ereal-of-enat-inf*[*simp*]: *ereal-of-enat* $n = \infty \longleftrightarrow n = \infty$
by (*cases n*) *auto*

lemma *ereal-of-enat-add*: *ereal-of-enat* $(m + n) =$ *ereal-of-enat* $m +$ *ereal-of-enat* n
by (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-sub*:
assumes $n \leq m$
shows *ereal-of-enat* $(m - n) =$ *ereal-of-enat* $m -$ *ereal-of-enat* n
using *assms* **by** (*cases m n rule*: *enat2-cases*) *auto*

lemma *ereal-of-enat-mult*:
ereal-of-enat $(m * n) =$ *ereal-of-enat* $m *$ *ereal-of-enat* n
by (*cases m n rule*: *enat2-cases*) *auto*

lemmas *ereal-of-enat-pushin* = *ereal-of-enat-add ereal-of-enat-sub ereal-of-enat-mult*
lemmas *ereal-of-enat-pushout* = *ereal-of-enat-pushin*[*symmetric*]

lemma *ereal-of-enat-nonneg*: *ereal-of-enat* $n \geq 0$
by *simp*

lemma *ereal-of-enat-Sup*:
assumes $A \neq \{\}$ **shows** *ereal-of-enat* $(Sup\ A) = (SUP\ a \in A.$ *ereal-of-enat* $a)$
proof (*intro antisym mono-Sup*)
show *ereal-of-enat* $(Sup\ A) \leq (SUP\ a \in A.$ *ereal-of-enat* $a)$
proof *cases*
assume *finite A*
with ‹$A \neq \{\}$› **obtain** a **where** $a \in A$ *ereal-of-enat* $(Sup\ A) =$ *ereal-of-enat* a
using *Max-in*[*of A*] **by** (*auto simp*: *Sup-enat-def simp del*: *Max-in*)
then show *?thesis*
by (*auto intro*: *SUP-upper*)
next
assume ¬ *finite A*
have [*simp*]: $(SUP\ a \in A.$ *ereal-of-enat* $a) = top$
unfolding *SUP-eq-top-iff*
proof *safe*
fix $x ::$ *ereal* **assume** $x < top$
then obtain $n ::$ *nat* **where** $x < n$

using *less-PInf-Ex-of-nat top-ereal-def* **by** *auto*
obtain *a* **where** $a \in A - enat\ `\{..\ n\}$
by (*metis* ‹¬ *finite A*› *all-not-in-conv finite-Diff2 finite-atMost finite-imageI finite.emptyI*)
then have $a \in A$ *ereal n* $\leq$ *ereal-of-enat a*
by (*auto simp*: *image-iff Ball-def*)
(*metis enat-iless enat-ord-simps*(*1*) *ereal-of-enat-less-iff ereal-of-enat-simps*(*1*) *less-le not-less*)
with ‹*x* < *n*› **show** $\exists\, i{\in}A.\ x <$ *ereal-of-enat i*
by (*auto intro*!: *bexI*[*of* - *a*])
qed
show *?thesis*
by *simp*
qed
qed (*simp add*: *mono-def*)

lemma *ereal-of-enat-SUP*:
$A \neq \{\} \Longrightarrow$ *ereal-of-enat* (*SUP a*∈*A. f a*) = (*SUP a* ∈ *A. ereal-of-enat* (*f a*))
by (*simp add*: *ereal-of-enat-Sup image-comp*)

38.6 Limits on *ereal*

lemma *open-PInfty*: *open A* $\Longrightarrow \infty \in A \Longrightarrow (\exists\, x.\ \{$*ereal x*<..$\} \subseteq A)$
unfolding *open-ereal-generated*
proof (*induct rule*: *generate-topology.induct*)
case (*Int A B*)
then obtain *x z* **where** $\infty \in A \Longrightarrow \{$*ereal x* <..$\} \subseteq A\ \infty \in B \Longrightarrow \{$*ereal z* <..$\} \subseteq B$
by *auto*
with *Int* **show** *?case*
by (*intro exI*[*of* - *max x z*]) *fastforce*
next
case (*Basis S*)
moreover have $x \neq \infty \Longrightarrow \exists\, t.\ x \leq$ *ereal t* **for** *x*
by (*cases x*) *auto*
ultimately show *?case*
by (*auto split*: *ereal.split*)
qed (*fastforce simp*: *vimage-Union*)+

lemma *open-MInfty*: *open A* $\Longrightarrow -\infty \in A \Longrightarrow (\exists\, x.\ \{..<$*ereal x*$\} \subseteq A)$
unfolding *open-ereal-generated*
proof (*induct rule*: *generate-topology.induct*)
case (*Int A B*)
then obtain *x z* **where** $-\infty \in A \Longrightarrow \{..<$ *ereal x*$\} \subseteq A\ -\infty \in B \Longrightarrow \{..<$ *ereal z*$\} \subseteq B$
by *auto*
with *Int* **show** *?case*
by (*intro exI*[*of* - *min x z*]) *fastforce*
next

case (*Basis S*)
moreover have $x \neq -\infty \Longrightarrow \exists\, t.\ ereal\ t \leq x$ **for** x
by (*cases x*) *auto*
ultimately show *?case*
by (*auto split*: *ereal.split*)
qed (*fastforce simp*: *vimage-Union*)+

lemma *open-ereal-vimage*: *open S* $\Longrightarrow$ *open* (*ereal* $-$ ' *S*)
by (*intro open-vimage continuous-intros*)

lemma *open-ereal*: *open S* $\Longrightarrow$ *open* (*ereal* ' *S*)
unfolding *open-generated-order*[**where** *'a=real*]
proof (*induct rule*: *generate-topology.induct*)
case (*Basis S*)
moreover have $\bigwedge x.\ ereal$ ' $\{..< x\} = \{ -\infty <..<$ *ereal x* $\}$
using *ereal-less-ereal-Ex* **by** *auto*
moreover have $\bigwedge x.\ ereal$ ' $\{x <..\} = \{$ *ereal x* $<..< \infty \}$
using *less-ereal.elims*(*2*) **by** *fastforce*
ultimately show *?case*
by *auto*
qed (*auto simp*: *image-Union image-Int*)

lemma *open-image-real-of-ereal*:
fixes *X*::*ereal set*
assumes *open X*
assumes *infty*: $\infty \notin X$ $-\infty \notin X$
shows *open* (*real-of-ereal* ' *X*)
proof $-$
have *real-of-ereal* ' *X* = *ereal* $-$ ' *X*
using *infty ereal-real* **by** (*force simp*: *set-eq-iff*)
thus *?thesis*
by (*auto intro*!: *open-ereal-vimage assms*)
qed

lemma *eventually-finite*:
fixes *x* :: *ereal*
assumes $|x| \neq \infty$ $(f \longrightarrow x)$ *F*
shows *eventually* $(\lambda x.\ |f\ x| \neq \infty)$ *F*
proof $-$
have $(f \longrightarrow$ *ereal* (*real-of-ereal x*)) *F*
using *assms* **by** (*cases x*) *auto*
then have *eventually* $(\lambda x.\ f\ x \in$ *ereal* ' *UNIV*) *F*
by (*rule topological-tendstoD*) (*auto intro*: *open-ereal*)
also have $(\lambda x.\ f\ x \in$ *ereal* ' *UNIV*$) = (\lambda x.\ |f\ x| \neq \infty)$
by *auto*
finally show *?thesis* **.**
qed

lemma *open-ereal-def*:
  open A $\longleftrightarrow$ *open* (*ereal* $-$‘ *A*) $\wedge$ ($\infty \in A \longrightarrow (\exists x.\ \{ereal\ x <..\} \subseteq A)) \wedge (-\infty \in A \longrightarrow (\exists x.\ \{..<ereal\ x\} \subseteq A)$)
  (**is** *open A* $\longleftrightarrow$ *?rhs*)
proof
  assume *open A*
  then show *?rhs*
    using *open-PInfty open-MInfty open-ereal-vimage* **by** *auto*
next
  assume *?rhs*
  then obtain *x y* **where** *A*: *open* (*ereal* $-$‘ *A*) $\infty \in A \Longrightarrow \{ereal\ x<..\} \subseteq A$ $-\infty \in A \Longrightarrow \{..<\ ereal\ y\} \subseteq A$
    by *auto*
  have $*$: *A* = *ereal* ‘ (*ereal* $-$‘ *A*) $\cup$ (*if* $\infty \in A$ *then* {*ereal x*<..} *else* {}) $\cup$ (*if* $-\infty \in A$ *then* {..< *ereal y*} *else* {})
    using *A*(*2*,*3*) **by** *auto*
  from *open-ereal*[*OF A*(*1*)] **show** *open A*
    by (*subst* $*$) (*auto simp*: *open-Un*)
qed

lemma *open-PInfty2*:
  assumes *open A* **and** $\infty \in A$
  obtains *x* **where** {*ereal x*<..} $\subseteq A$
  using *open-PInfty*[*OF assms*] **by** *auto*

lemma *open-MInfty2*:
  assumes *open A* **and** $-\infty \in A$
  obtains *x* **where** {..<*ereal x*} $\subseteq A$
  using *open-MInfty*[*OF assms*] **by** *auto*

lemma *ereal-openE*:
  assumes *open A*
  obtains *x y* **where** *open* (*ereal* $-$‘ *A*)
    and $\infty \in A \Longrightarrow \{ereal\ x<..\} \subseteq A$
    and $-\infty \in A \Longrightarrow \{..<ereal\ y\} \subseteq A$
  using *assms open-ereal-def* **by** *auto*

lemmas *open-ereal-lessThan* = *open-lessThan*[**where** *'a*=*ereal*]
lemmas *open-ereal-greaterThan* = *open-greaterThan*[**where** *'a*=*ereal*]
lemmas *ereal-open-greaterThanLessThan* = *open-greaterThanLessThan*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atLeast* = *closed-atLeast*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atMost* = *closed-atMost*[**where** *'a*=*ereal*]
lemmas *closed-ereal-atLeastAtMost* = *closed-atLeastAtMost*[**where** *'a*=*ereal*]
lemmas *closed-ereal-singleton* = *closed-singleton*[**where** *'a*=*ereal*]

lemma *ereal-open-cont-interval*:
  fixes *S* :: *ereal set*
  assumes *open S*
    and $x \in S$

and $|x| \neq \infty$
obtains e **where** $e > 0$ **and** $\{x-e <..< x+e\} \subseteq S$
proof −
from ‹*open S*›
have *open* (*ereal* −‘ *S*)
by (*rule ereal-openE*)
then obtain e **where** $e > 0$ **and** e: *dist y* (*real-of-ereal x*) $< e \Longrightarrow$ *ereal y* $\in S$
for y
using *assms* **unfolding** *open-dist* **by** *force*
show *thesis*
proof (*intro that subsetI*)
show $0 <$ *ereal e*
using ‹$0 < e$› **by** *auto*
fix y
assume $y \in \{x -$ *ereal e*$<..<x +$ *ereal e*$\}$
with *assms* **obtain** t **where** $y =$ *ereal t dist t* (*real-of-ereal x*) $< e$
by (*cases y*) (*auto simp*: *dist-real-def*)
then show $y \in S$
using *e*[*of t*] **by** *auto*
qed
qed

lemma *ereal-open-cont-interval2*:
fixes S :: *ereal set*
assumes *open S* **and** $x \in S$ **and** $|x| \neq \infty$
obtains a b **where** $a < x$ **and** $x < b$ **and** $\{a <..< b\} \subseteq S$
by (*meson assms ereal-between ereal-open-cont-interval*)

38.6.1 Convergent sequences

lemma *lim-real-of-ereal*[*simp*]:
assumes *lim*: (*f* ⟶ *ereal x*) *net*
shows ((λ*x*. *real-of-ereal* (*f x*)) ⟶ *x*) *net*
proof (*intro topological-tendstoI*)
fix *S*
assume *open S* **and** $x \in S$
then have *S*: *open S ereal x* ∈ *ereal* ‘ *S*
by (*simp-all add*: *inj-image-mem-iff*)
show *eventually* (λ*x*. *real-of-ereal* (*f x*) ∈ *S*) *net*
by (*auto intro*: *eventually-mono* [*OF lim*[*THEN topological-tendstoD*, *OF open-ereal*, *OF S*]])
qed

lemma *lim-ereal*[*simp*]: ((λ*n*. *ereal* (*f n*)) ⟶ *ereal x*) *net* ⟷ (*f* ⟶ *x*) *net*
by (*auto dest*!: *lim-real-of-ereal*)

lemma *convergent-real-imp-convergent-ereal*:
assumes *convergent a*
shows *convergent* (λ*n*. *ereal* (*a n*)) **and** *lim* (λ*n*. *ereal* (*a n*)) = *ereal* (*lim a*)

proof −
from *assms* **obtain** *L* **where** *L*: $a \longrightarrow L$ **unfolding** *convergent-def* **..**
hence *lim*: $(\lambda n.\ ereal\ (a\ n)) \longrightarrow ereal\ L$ **using** *lim-ereal* **by** *auto*
thus *convergent* $(\lambda n.\ ereal\ (a\ n))$ **unfolding** *convergent-def* **..**
thus *lim* $(\lambda n.\ ereal\ (a\ n)) = ereal\ (lim\ a)$ **using** *lim L limI* **by** *metis*
qed

lemma *tendsto-PInfty*: $(f \longrightarrow \infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
proof −
{ fix *l* :: *ereal*
assume $\forall r.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F$
from *this*[*THEN spec*, *of real-of-ereal l*]
have $l \neq \infty \Longrightarrow eventually\ (\lambda x.\ l < f\ x)\ F$
by (*cases l*) (*auto elim*: *eventually-mono*)
}
then show *?thesis*
by (*auto simp*: *order-tendsto-iff*)
qed

lemma *tendsto-PInfty'*: $(f \longrightarrow \infty)\ F = (\forall r{>}c.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F)$
proof −
{ fix *r* :: *real*
assume $\forall r{>}c.\ eventually\ (\lambda x.\ ereal\ r < f\ x)\ F$
then have *eventually* $(\lambda x.\ ereal\ r < f\ x)\ F$
if $r > c$ **for** *r* **using** *that* **by** *blast*
then have *eventually* $(\lambda x.\ ereal\ r < f\ x)\ F$
by (*smt* (*verit*, *del-insts*) *ereal-less-le eventually-mono gt-ex*)
} then show *?thesis*
using *tendsto-PInfty* **by** *blast*
qed

lemma *tendsto-PInfty-eq-at-top*:
$((\lambda z.\ ereal\ (f\ z)) \longrightarrow \infty)\ F \longleftrightarrow (LIM\ z\ F.\ f\ z :> at\text{-}top)$
unfolding *tendsto-PInfty filterlim-at-top-dense* **by** *simp*

lemma *tendsto-MInfty*: $(f \longrightarrow -\infty)\ F \longleftrightarrow (\forall r.\ eventually\ (\lambda x.\ f\ x < ereal\ r)\ F)$
unfolding *tendsto-def*
proof *safe*
fix *S* :: *ereal set*
assume *open S* $-\infty \in S$
from *open-MInfty*[*OF this*] **obtain** *B* **where** $\{..{<}ereal\ B\} \subseteq S$ **..**
moreover
assume $\forall r{::}real.\ eventually\ (\lambda z.\ f\ z < r)\ F$
then have *eventually* $(\lambda z.\ f\ z \in \{..< B\})\ F$
by *auto*
ultimately show *eventually* $(\lambda z.\ f\ z \in S)\ F$
by (*auto elim*!: *eventually-mono*)

next
fix x
assume $\forall S.$ *open* $S \longrightarrow -\infty \in S \longrightarrow$ *eventually* $(\lambda x.\ f\ x \in S)\ F$
from *this*[*rule-format*, *of* $\{..< ereal\ x\}$] **show** *eventually* $(\lambda y.\ f\ y < ereal\ x)\ F$
by *auto*
qed

lemma *tendsto-MInfty'*: $(f \longrightarrow -\infty)\ F = (\forall r{<}c.$ *eventually* $(\lambda x.\ ereal\ r > f\ x)\ F)$
proof (*subst tendsto-MInfty*, *intro iffI allI impI*)
assume *A*: $\forall r{<}c.$ *eventually* $(\lambda x.\ ereal\ r > f\ x)\ F$
fix $r :: real$
from *A* **have** *A*: *eventually* $(\lambda x.\ ereal\ r > f\ x)\ F$ **if** $r < c$ **for** r **using** *that* **by** *blast*
show *eventually* $(\lambda x.\ ereal\ r > f\ x)\ F$
proof (*cases* $r < c$)
case *False*
hence *B*: $ereal\ r \geq ereal\ (c - 1)$ **by** *simp*
have $c > c - 1$ **by** *simp*
from *A*[*OF this*] **show** *eventually* $(\lambda x.\ ereal\ r > f\ x)\ F$
by *eventually-elim* (*erule less-le-trans*[*OF* - *B*])
qed (*simp add*: *A*)
qed *simp*

lemma *Lim-PInfty*: $f \longrightarrow \infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ f\ n \geq ereal\ B)$
unfolding *tendsto-PInfty eventually-sequentially*
proof *safe*
fix r
assume $\forall r.\ \exists N.\ \forall n{\geq}N.\ ereal\ r \leq f\ n$
then obtain N **where** $\forall n{\geq}N.\ ereal\ (r + 1) \leq f\ n$
by *blast*
moreover have $ereal\ r < ereal\ (r + 1)$
by *auto*
ultimately show $\exists N.\ \forall n{\geq}N.\ ereal\ r < f\ n$
by (*blast intro*: *less-le-trans*)
qed (*blast intro*: *less-imp-le*)

lemma *Lim-MInfty*: $f \longrightarrow -\infty \longleftrightarrow (\forall B.\ \exists N.\ \forall n{\geq}N.\ ereal\ B \geq f\ n)$
unfolding *tendsto-MInfty eventually-sequentially*
proof *safe*
fix r
assume $\forall r.\ \exists N.\ \forall n{\geq}N.\ f\ n \leq ereal\ r$
then obtain N **where** $\forall n{\geq}N.\ f\ n \leq ereal\ (r - 1)$
by *blast*
moreover have $ereal\ (r - 1) < ereal\ r$
by *auto*
ultimately show $\exists N.\ \forall n{\geq}N.\ f\ n < ereal\ r$
by (*blast intro*: *le-less-trans*)
qed (*blast intro*: *less-imp-le*)

lemma *Lim-bounded-PInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ f\ n \leq ereal\ B) \Longrightarrow l \neq \infty$
using *LIMSEQ-le-const2*[*of f l ereal B*] **by** *auto*

lemma *Lim-bounded-MInfty*: $f \longrightarrow l \Longrightarrow (\bigwedge n.\ ereal\ B \leq f\ n) \Longrightarrow l \neq -\infty$
using *LIMSEQ-le-const*[*of f l ereal B*] **by** *auto*

lemma *tendsto-zero-erealI*:
assumes $\bigwedge e.\ e > 0 \Longrightarrow eventually\ (\lambda x.\ |f\ x| < ereal\ e)\ F$
shows $(f \longrightarrow 0)\ F$
proof (*subst filterlim-cong*[*OF refl refl*])
from *assms*[*OF zero-less-one*] **show** *eventually* ($\lambda x.\ f\ x = ereal\ (real\text{-}of\text{-}ereal\ (f\ x))$) *F*
by *eventually-elim* (*auto simp*: *ereal-real*)
hence *eventually* ($\lambda x.\ abs\ (real\text{-}of\text{-}ereal\ (f\ x)) < e$) *F* **if** $e > 0$ **for** *e* **using** *assms*[*OF that*]
by *eventually-elim* (*simp add*: *real-less-ereal-iff that*)
hence (($\lambda x.\ real\text{-}of\text{-}ereal\ (f\ x)$) $\longrightarrow 0$) *F* **unfolding** *tendsto-iff*
by (*auto simp*: *tendsto-iff dist-real-def*)
thus (($\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))$) $\longrightarrow 0$) *F* **by** (*simp add*: *zero-ereal-def*)
qed

lemma *Lim-bounded-PInfty2*: $f \longrightarrow l \Longrightarrow \forall\, n{\geq}N.\ f\ n \leq ereal\ B \Longrightarrow l \neq \infty$
using *LIMSEQ-le-const2*[*of f l ereal B*] **by** *fastforce*

lemma *real-of-ereal-mult*[*simp*]:
fixes *a b* :: *ereal*
shows *real-of-ereal* ($a * b$) = *real-of-ereal a* $*$ *real-of-ereal b*
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *real-of-ereal-eq-0*:
fixes *x* :: *ereal*
shows *real-of-ereal x* = *0* $\longleftrightarrow x = \infty \vee x = -\infty \vee x = 0$
by (*cases x*) *auto*

lemma *tendsto-ereal-realD*:
fixes *f* :: *'a* $\Rightarrow$ *ereal*
assumes $x \neq 0$
and *tendsto*: (($\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))$) $\longrightarrow x$) *net*
shows ($f \longrightarrow x$) *net*
proof (*intro topological-tendstoI*)
fix *S*
assume *S*: *open S x* $\in S$
with ‹$x \neq 0$› **have** *open* ($S - \{0\}$) $x \in S - \{0\}$
by *auto*
from *tendsto*[*THEN topological-tendstoD*, *OF this*]
show *eventually* ($\lambda x.\ f\ x \in S$) *net*
by (*rule eventually-rev-mp*) (*auto simp*: *ereal-real*)
qed

lemma *tendsto-ereal-realI*:
fixes $f :: 'a \Rightarrow ereal$
assumes x: $|x| \neq \infty$ **and** *tendsto*: $(f \longrightarrow x)$ *net*
shows $((\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x))) \longrightarrow x)$ *net*
proof (*intro topological-tendstoI*)
fix S
assume *open* S **and** $x \in S$
with x **have** *open* $(S - \{\infty, -\infty\})$ $x \in S - \{\infty, -\infty\}$
by *auto*
from *tendsto*[*THEN topological-tendstoD*, *OF this*]
show *eventually* $(\lambda x.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ x)) \in S)$ *net*
by (*elim eventually-mono*) (*auto simp*: *ereal-real*)
qed

lemma *ereal-mult-cancel-left*:
fixes $a\ b\ c :: ereal$
shows $a * b = a * c \longleftrightarrow (|a| = \infty \wedge 0 < b * c) \vee a = 0 \vee b = c$
by (*cases rule*: *ereal3-cases*[*of a b c*]) (*simp-all add*: *zero-less-mult-iff*)

lemma *tendsto-add-ereal*:
fixes $x\ y :: ereal$
assumes x: $|x| \neq \infty$ **and** y: $|y| \neq \infty$
assumes f: $(f \longrightarrow x)$ F **and** g: $(g \longrightarrow y)$ F
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)$ F
proof −
from x **obtain** r **where** x': $x = ereal\ r$ **by** (*cases x*) *auto*
with f **have** $((\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i)) \longrightarrow r)$ F **by** *simp*
moreover
from y **obtain** p **where** y': $y = ereal\ p$ **by** (*cases y*) *auto*
with g **have** $((\lambda i.\ real\text{-}of\text{-}ereal\ (g\ i)) \longrightarrow p)$ F **by** *simp*
ultimately have $((\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i) + real\text{-}of\text{-}ereal\ (g\ i)) \longrightarrow r + p)$ F
by (*rule tendsto-add*)
moreover
from *eventually-finite*[*OF x f*] *eventually-finite*[*OF y g*]
have *eventually* $(\lambda x.\ f\ x + g\ x = ereal\ (real\text{-}of\text{-}ereal\ (f\ x) + real\text{-}of\text{-}ereal\ (g\ x)))$
F
by *eventually-elim auto*
ultimately show *?thesis*
by (*simp add*: x' y' *cong*: *filterlim-cong*)
qed

lemma *tendsto-add-ereal-nonneg*:
fixes $x\ y :: ereal$
assumes $x \neq -\infty$ $y \neq -\infty$ $(f \longrightarrow x)$ F $(g \longrightarrow y)$ F
shows $((\lambda x.\ f\ x + g\ x) \longrightarrow x + y)$ F
proof (*cases* $x = \infty \vee y = \infty$)
case *True*
moreover

```
  { fix y :: ereal and f g :: 'a ⇒ ereal assume y ≠ −∞ (f ⟶ ∞) F (g ⟶
y) F
    then obtain y' where −∞ < y' y' < y
      using dense[of −∞ y] by auto
    have ((λx. f x + g x) ⟶ ∞) F
    proof (rule tendsto-sandwich)
      have ∀F x in F. y' < g x
        using order-tendstoD(1)[OF ‹(g ⟶ y) F› ‹y' < y›] by auto
      then show ∀F x in F. f x + y' ≤ f x + g x
        by eventually-elim (auto intro!: add-mono)
      show ∀F n in F. f n + g n ≤ ∞ ((λn. ∞) ⟶ ∞) F
        by auto
      show ((λx. f x + y') ⟶ ∞) F
        using tendsto-cadd-ereal[of y' ∞ f F] ‹(f ⟶ ∞) F› ‹−∞ < y'› by auto
    qed }
  note this[of y f g] this[of x g f]
  ultimately show ?thesis
    using assms by (auto simp: add-ac)
next
  case False
  with assms tendsto-add-ereal[of x y f F g]
  show ?thesis
    by auto
qed

lemma ereal-inj-affinity:
  fixes m t :: ereal
  assumes |m| ≠ ∞
    and m ≠ 0
    and |t| ≠ ∞
  shows inj-on (λx. m * x + t) A
  using assms
  by (cases rule: ereal2-cases[of m t])
     (auto intro!: inj-onI simp: ereal-add-cancel-right ereal-mult-cancel-left)

lemma ereal-PInfty-eq-plus[simp]:
  fixes a b :: ereal
  shows ∞ = a + b ⟷ a = ∞ ∨ b = ∞
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-MInfty-eq-plus[simp]:
  fixes a b :: ereal
  shows −∞ = a + b ⟷ (a = −∞ ∧ b ≠ ∞) ∨ (b = −∞ ∧ a ≠ ∞)
  by (cases rule: ereal2-cases[of a b]) auto

lemma ereal-less-divide-pos:
  fixes x y :: ereal
  shows x > 0 ⟹ x ≠ ∞ ⟹ y < z / x ⟷ x * y < z
  by (simp add: ereal-less-divide-iff mult.commute)
```

lemma *ereal-divide-less-pos*:
fixes $x\ y\ z$ :: *ereal*
shows $x > 0 \Longrightarrow x \neq \infty \Longrightarrow y\ /\ x < z \longleftrightarrow y < x * z$
by (*simp add*: *ereal-divide-less-iff mult.commute*)

lemma *ereal-divide-eq*:
fixes $a\ b\ c$ :: *ereal*
shows $b \neq 0 \Longrightarrow |b| \neq \infty \Longrightarrow a\ /\ b = c \longleftrightarrow a = b * c$
by (*metis ereal-divide-same ereal-times-divide-eq mult.commute*
mult.right-neutral)

lemma *ereal-inverse-not-MInfty*[*simp*]: *inverse* (a::*ereal*) $\neq -\infty$
by (*cases a*) *auto*

lemma *ereal-mult-m1*[*simp*]: $x * ereal\ (-1) = -x$
by (*cases x*) *auto*

lemma *ereal-real'*:
assumes $|x| \neq \infty$
shows *ereal* (*real-of-ereal x*) $= x$
using *assms* **by** *auto*

lemma *real-ereal-id*: *real-of-ereal* $\circ$ *ereal* $= id$
by *auto*

lemma *open-image-ereal*: *open*(*UNIV*$-\{\ \infty\ ,\ (-\infty$ :: *ereal*)$\}$)
by (*metis range-ereal open-ereal open-UNIV*)

lemma *ereal-le-distrib*:
fixes $a\ b\ c$ :: *ereal*
shows $c * (a + b) \leq c * a + c * b$
by (*cases rule*: *ereal3-cases*[*of a b c*])
(*auto simp*: *field-simps not-le mult-le-0-iff mult-less-0-iff*)

lemma *ereal-pos-distrib*:
fixes $a\ b\ c$ :: *ereal*
assumes $0 \leq c$
and $c \neq \infty$
shows $c * (a + b) = c * a + c * b$
using *assms*
by (*cases rule*: *ereal3-cases*[*of a b c*])
(*auto simp*: *field-simps not-le mult-le-0-iff mult-less-0-iff*)

lemma *ereal-LimI-finite*:
fixes x :: *ereal*
assumes $|x| \neq \infty$
and $\bigwedge r.\ 0 < r \Longrightarrow \exists N.\ \forall n \geq N.\ u\ n < x + r \wedge x < u\ n + r$
shows $u \longrightarrow x$

proof (*rule topological-tendstoI*, *unfold eventually-sequentially*)
obtain *rx* **where** *rx*: $x = ereal\ rx$
using *assms* **by** (*cases x*) *auto*
fix *S*
assume *open S* **and** $x \in S$
then have *open* (*ereal* −‘ *S*)
unfolding *open-ereal-def* **by** *auto*
with ‹$x \in S$› **obtain** *r* **where** $0 < r$ **and** *dist*: $dist\ y\ rx < r \Longrightarrow ereal\ y \in S$ **for** *y*
unfolding *open-dist rx* **by** *auto*
then obtain *n*
where *upper*: $u\ N < x + ereal\ r$
and *lower*: $x < u\ N + ereal\ r$
if $n \leq N$ **for** *N*
using *assms*(*2*)[*of ereal r*] **by** *auto*
show $\exists N.\ \forall n{\geq}N.\ u\ n \in S$
proof (*safe intro*!: *exI*[*of* - *n*])
fix *N*
assume $n \leq N$
from *upper*[*OF this*] *lower*[*OF this*] *assms* ‹$0 < r$›
have $u\ N \notin \{\infty,(-\infty)\}$
by *auto*
then obtain *ra* **where** *ra-def*: $(u\ N) = ereal\ ra$
by (*cases u N*) *auto*
then have $rx < ra + r$ **and** $ra < rx + r$
using *rx assms* ‹$0 < r$› *lower*[*OF* ‹$n \leq N$›] *upper*[*OF* ‹$n \leq N$›]
by *auto*
then have *dist* (*real-of-ereal* (*u N*)) *rx* < *r*
using *rx ra-def*
by (*auto simp*: *dist-real-def abs-diff-less-iff field-simps*)
from *dist*[*OF this*] **show** $u\ N \in S$
using ‹$u\ N\ \notin \{\infty, -\infty\}$›
by (*auto simp*: *ereal-real split*: *if-split-asm*)
qed
qed

lemma *tendsto-obtains-N*:
assumes $f \longrightarrow f0$ *open S f0* $\in S$
obtains *N* **where** $\forall n{\geq}N.\ f\ n \in S$
using *assms lim-explicit* **by** *blast*

lemma *ereal-LimI-finite-iff*:
fixes *x* :: *ereal*
assumes $|x| \neq \infty$
shows $u \longrightarrow x \longleftrightarrow (\forall r.\ 0 < r \longrightarrow (\exists N.\ \forall n{\geq}N.\ u\ n < x + r \wedge x < u\ n + r))$
(**is** *?lhs* ⟷ *?rhs*)
proof
assume *lim*: $u \longrightarrow x$

```
  {
    fix r :: ereal
    assume r > 0
    then obtain N where ∀ n≥N. u n ∈ {x − r <..< x + r}
      using lim ereal-between[of x r] assms ‹r > 0› tendsto-obtains-N[of u x {x −
r <..< x + r}]
      by auto
    then have ∃ N. ∀ n≥N. u n < x + r ∧ x < u n + r
      using ereal-minus-less[of r x]
      by (cases r) auto
  }
  then show ?rhs
    by auto
next
  assume ?rhs
  then show u ⟶ x
    using ereal-LimI-finite[of x] assms by auto
qed

lemma ereal-Limsup-uminus:
  fixes f :: 'a ⇒ ereal
  shows Limsup net (λx. − (f x)) = − Liminf net f
  unfolding Limsup-def Liminf-def ereal-SUP-uminus-eq ereal-INF-uminus-eq ..

lemma liminf-bounded-iff:
  fixes x :: nat ⇒ ereal
  shows C ≤ liminf x ⟷ (∀ B<C. ∃ N. ∀ n≥N. B < x n)
  (is ?lhs ⟷ ?rhs)
  unfolding le-Liminf-iff eventually-sequentially ..

lemma Liminf-add-le:
  fixes f g :: - ⇒ ereal
  assumes F: F ≠ bot
  assumes ev: eventually (λx. 0 ≤ f x) F eventually (λx. 0 ≤ g x) F
  shows Liminf F f + Liminf F g ≤ Liminf F (λx. f x + g x)
  unfolding Liminf-def
proof (subst SUP-ereal-add-left[symmetric])
  let ?F = {P. eventually P F}
  let ?INF = λP g. Inf (g ' (Collect P))
  show ?F ≠ {}
    by (auto intro: eventually-True)
  show (SUP P∈?F. ?INF P g) ≠ −∞
    unfolding bot-ereal-def[symmetric] SUP-bot-conv INF-eq-bot-iff
    by (auto intro!: exI[of - 0] ev simp: bot-ereal-def)
  have (SUP P∈?F. ?INF P f + (SUP P∈?F. ?INF P g)) ≤ (SUP P∈?F. (SUP
P'∈?F. ?INF P f + ?INF P' g))
  proof (safe intro!: SUP-mono bexI[of - λx. P x ∧ 0 ≤ f x for P])
    fix P let ?P' = λx. P x ∧ 0 ≤ f x
    assume eventually P F
```

```
    with ev show eventually ?P' F
      by eventually-elim auto
    have ?INF P f + (SUP P∈?F. ?INF P g) ≤ ?INF ?P' f + (SUP P∈?F. ?INF
P g)
      by (intro add-mono INF-mono) auto
    also have ... = (SUP P'∈?F. ?INF ?P' f + ?INF P' g)
    proof (rule SUP-ereal-add-right[symmetric])
      show Inf (f ` {x. P x ∧ 0 ≤ f x}) ≠ −∞
        unfolding bot-ereal-def[symmetric] INF-eq-bot-iff
        by (auto intro!: exI[of - 0] ev simp: bot-ereal-def)
    qed fact
    finally show ?INF P f + (SUP P∈?F. ?INF P g) ≤ (SUP P'∈?F. ?INF ?P'
f + ?INF P' g) .
  qed
  also have ... ≤ (SUP P∈?F. INF x∈Collect P. f x + g x)
  proof (safe intro!: SUP-least)
    fix P Q assume ∗: eventually P F eventually Q F
    show ?INF P f + ?INF Q g ≤ (SUP P∈?F. INF x∈Collect P. f x + g x)
    proof (rule SUP-upper2)
      show (λx. P x ∧ Q x) ∈ ?F
        using ∗ by (auto simp: eventually-conj)
      show ?INF P f + ?INF Q g ≤ (INF x∈{x. P x ∧ Q x}. f x + g x)
        by (intro INF-greatest add-mono) (auto intro: INF-lower)
    qed
  qed
  finally show (SUP P∈?F. ?INF P f + (SUP P∈?F. ?INF P g)) ≤ (SUP P∈?F.
INF x∈Collect P. f x + g x) .
qed

lemma Sup-ereal-mult-right':
  assumes nonempty: Y ≠ {}
  and x: x ≥ 0
  shows (SUP i∈Y. f i) ∗ ereal x = (SUP i∈Y. f i ∗ ereal x) (is ?lhs = ?rhs)
proof(cases x = 0)
  case True thus ?thesis by(auto simp: nonempty zero-ereal-def[symmetric])
next
  case False
  show ?thesis
  proof(rule antisym)
    show ?rhs ≤ ?lhs
      by(rule SUP-least)(simp add: ereal-mult-right-mono SUP-upper x)
  next
    have ?lhs / ereal x = (SUP i∈Y. f i) ∗ (ereal x / ereal x) by(simp only:
ereal-times-divide-eq)
    also have ... = (SUP i∈Y. f i) using False by simp
    also have ... ≤ ?rhs / x
    proof(rule SUP-least)
      fix i
      assume i ∈ Y
```

have $f\ i = f\ i * (ereal\ x\ /\ ereal\ x)$ **using** *False* **by** *simp*
also have $\ldots = f\ i * x\ /\ x$ **by**(*simp only*: *ereal-times-divide-eq*)
also from ‹$i \in Y$› **have** $f\ i * x \leq ?rhs$ **by**(*rule SUP-upper*)
hence $f\ i * x\ /\ x \leq ?rhs\ /\ x$ **using** x *False* **by** *simp*
finally show $f\ i \leq ?rhs\ /\ x$ **.**
qed
finally have $(?lhs\ /\ x) * x \leq (?rhs\ /\ x) * x$
by(*rule ereal-mult-right-mono*)(*simp add*: x)
also have $\ldots = ?rhs$ **using** *False ereal-divide-eq mult.commute* **by** *force*
also have $(?lhs\ /\ x) * x = ?lhs$ **using** *False ereal-divide-eq mult.commute* **by** *force*
finally show $?lhs \leq ?rhs$ **.**
qed
qed

lemma *Sup-ereal-mult-left'*:
$[\![\ Y \neq \{\};\ x \geq 0\ ]\!] \Longrightarrow ereal\ x * (SUP\ i{\in}Y.\ f\ i) = (SUP\ i{\in}Y.\ ereal\ x * f\ i)$
by (*smt* (*verit*) *Sup.SUP-cong Sup-ereal-mult-right' mult.commute*)

lemma *sup-continuous-add*[*order-continuous-intros*]:
fixes $f\ g :: 'a{::}complete\text{-}lattice \Rightarrow ereal$
assumes *nn*: $\bigwedge x.\ 0 \leq f\ x\ \bigwedge x.\ 0 \leq g\ x$ **and** *cont*: *sup-continuous f sup-continuous g*
shows *sup-continuous* $(\lambda x.\ f\ x + g\ x)$
unfolding *sup-continuous-def*
proof *safe*
fix $M :: nat \Rightarrow 'a$ **assume** *incseq M*
then show $f\ (SUP\ i.\ M\ i) + g\ (SUP\ i.\ M\ i) = (SUP\ i.\ f\ (M\ i) + g\ (M\ i))$
using *SUP-ereal-add-pos*[*of* $\lambda i.\ f\ (M\ i)$ $\lambda i.\ g\ (M\ i)$] *nn*
cont[*THEN sup-continuous-mono*] *cont*[*THEN sup-continuousD*]
by (*auto simp*: *mono-def*)
qed

lemma *sup-continuous-mult-right*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ f\ x * c :: ereal)$
by (*cases c*) (*auto simp*: *sup-continuous-def fun-eq-iff Sup-ereal-mult-right'*)

lemma *sup-continuous-mult-left*[*order-continuous-intros*]:
$0 \leq c \Longrightarrow c < \infty \Longrightarrow$ *sup-continuous f* $\Longrightarrow$ *sup-continuous* $(\lambda x.\ c * f\ x :: ereal)$
using *sup-continuous-mult-right*[*of c f*] **by** (*simp add*: *mult-ac*)

lemma *sup-continuous-ereal-of-enat*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f*
shows *sup-continuous* $(\lambda x.\ ereal\text{-}of\text{-}enat\ (f\ x))$
by (*metis UNIV-not-empty ereal-of-enat-SUP f sup-continuous-compose sup-continuous-def*)

38.6.2 Sums

lemma *sums-ereal-positive*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $f\ sums\ (SUP\ n.\ \sum i{<}n.\ f\ i)$
by (*simp add*: *LIMSEQ-SUP assms incseq-sumI sums-def*)

lemma *summable-ereal-pos*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows *summable f*
using *sums-ereal-positive*[*of f*, *OF assms*]
unfolding *summable-def*
by *auto*

lemma *sums-ereal*: $(\lambda x.\ ereal\ (f\ x))\ sums\ ereal\ x \longleftrightarrow f\ sums\ x$
unfolding *sums-def* **by** *simp*

lemma *suminf-ereal-eq-SUP*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \le f\ i$
shows $(\sum x.\ f\ x) = (SUP\ n.\ \sum i{<}n.\ f\ i)$
using *sums-ereal-positive*[*of f*, *OF assms*, *THEN sums-unique*]
by *simp*

lemma *suminf-bound*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) \le x\ \bigwedge n.\ 0 \le f\ n$
shows $suminf\ f \le x$
by (*simp add*: *SUP-least assms suminf-ereal-eq-SUP*)

lemma *suminf-bound-add*:
fixes $f :: nat \Rightarrow ereal$
assumes $\forall N.\ (\sum n{<}N.\ f\ n) + y \le x$
and $\bigwedge n.\ 0 \le f\ n$
and $y \neq -\infty$
shows $suminf\ f + y \le x$
by (*simp add*: *SUP-ereal-le-addI assms suminf-ereal-eq-SUP*)

lemma *suminf-upper*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$
shows $(\sum n{<}N.\ f\ n) \le (\sum n.\ f\ n)$
unfolding *suminf-ereal-eq-SUP* [*OF assms*]
by (*auto intro*: *complete-lattice-class.SUP-upper*)

lemma *suminf-0-le*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge n.\ 0 \le f\ n$

shows $0 \leq (\sum n.\ f\ n)$
using *suminf-upper*[*of f 0*, *OF assms*]
by *simp*

lemma *suminf-le-pos*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge N.\ f\ N \leq g\ N$
and $\bigwedge N.\ 0 \leq f\ N$
shows *suminf f* $\leq$ *suminf g*
by (*meson assms order-trans suminf-le summable-ereal-pos*)

lemma *suminf-half-series-ereal*: $(\sum n.\ (1/2 :: ereal) \ \hat{}\ Suc\ n) = 1$
using *sums-ereal*[*THEN iffD2*, *OF power-half-series*, *THEN sums-unique*, *symmetric*]
by (*simp add*: *one-ereal-def*)

lemma *suminf-add-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i\ \bigwedge i.\ 0 \leq g\ i$
shows $(\sum i.\ f\ i + g\ i) =$ *suminf f* + *suminf g*
proof −
have $(SUP\ n.\ \sum i{<}n.\ f\ i + g\ i) = (SUP\ n.\ sum\ f\ \{..{<}n\}) + (SUP\ n.\ sum\ g\ \{..{<}n\})$
unfolding *sum.distrib*
by (*intro assms add-nonneg-nonneg SUP-ereal-add-pos incseq-sumI sum-nonneg ballI*)
with *assms* **show** *?thesis*
by (*simp add*: *suminf-ereal-eq-SUP*)
qed

lemma *suminf-cmult-ereal*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$ **and** $0 \leq a$
shows $(\sum i.\ a * f\ i) = a *$ *suminf f*
by (*simp add*: *assms sum-nonneg suminf-ereal-eq-SUP sum-ereal-right-distrib flip*: *SUP-ereal-mult-left*)

lemma *suminf-PInfty*:
fixes $f :: nat \Rightarrow ereal$
assumes $\bigwedge i.\ 0 \leq f\ i$
and *suminf f* $\neq \infty$
shows $f\ i \neq \infty$
proof −
from *suminf-upper*[*of f Suc i*, *OF assms*(*1*)] *assms*(*2*)
have $(\sum i{<}Suc\ i.\ f\ i) \neq \infty$
by *auto*
then show *?thesis*
unfolding *sum-Pinfty* **by** *simp*
qed

lemma *suminf-PInfty-fun*:
assumes $\bigwedge i.\ 0 \le f\ i$
and $suminf\ f \neq \infty$
shows $\exists f'.\ f = (\lambda x.\ ereal\ (f'\ x))$
proof −
have $\forall\, i.\ \exists\, r.\ f\ i = ereal\ r$
by (*metis abs-ereal-ge0 abs-neq-infinity-cases assms suminf-PInfty*)
then show *?thesis*
by *metis*
qed

lemma *summable-ereal*:
assumes $\bigwedge i.\ 0 \le f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows *summable f*
proof −
have $0 \le (\sum i.\ ereal\ (f\ i))$
using *assms* **by** (*intro suminf-0-le*) *auto*
with *assms* **obtain** *r* **where** *r*: $(\sum i.\ ereal\ (f\ i)) = ereal\ r$
by (*cases* $\sum i.\ ereal\ (f\ i)$) *auto*
from *summable-ereal-pos*[*of* $\lambda x.\ ereal\ (f\ x)$]
have *summable* $(\lambda x.\ ereal\ (f\ x))$
using *assms* **by** *auto*
from *summable-sums*[*OF this*]
have $(\lambda x.\ ereal\ (f\ x))$ *sums* $(\sum x.\ ereal\ (f\ x))$
by *auto*
then show *summable f*
unfolding *r sums-ereal summable-def* **..**
qed

lemma *suminf-ereal*:
assumes $\bigwedge i.\ 0 \le f\ i$
and $(\sum i.\ ereal\ (f\ i)) \neq \infty$
shows $(\sum i.\ ereal\ (f\ i)) = ereal\ (suminf\ f)$
proof (*rule sums-unique*[*symmetric*])
from *summable-ereal*[*OF assms*]
show $(\lambda x.\ ereal\ (f\ x))$ *sums* $(ereal\ (suminf\ f))$
unfolding *sums-ereal*
using *assms*
by (*intro summable-sums summable-ereal*)
qed

lemma *suminf-ereal-minus*:
fixes $f\ g :: nat \Rightarrow ereal$
assumes *ord*: $\bigwedge i.\ g\ i \le f\ i\ \bigwedge i.\ 0 \le g\ i$
and *fin*: $suminf\ f \neq \infty\ suminf\ g \neq \infty$
shows $(\sum i.\ f\ i - g\ i) = suminf\ f - suminf\ g$
proof −

have *0*: $0 \leq f\ i$ **for** i
using *ord order-trans* **by** *blast*
moreover
obtain f' **where** [*simp*]: $f = (\lambda x.\ ereal\ (f'\ x))$
using *0 fin(1) suminf-PInfty-fun* **by** *presburger*
obtain g' **where** [*simp*]: $g = (\lambda x.\ ereal\ (g'\ x))$
using *fin(2) ord(2) suminf-PInfty-fun* **by** *presburger*
have $0 \leq f\ i - g\ i$ **for** i
using *ord(1)* **by** *auto*
moreover
have $suminf\ (\lambda i.\ f\ i - g\ i) \leq suminf\ f$
using *assms* **by** (*auto intro*!: *suminf-le-pos simp*: *field-simps*)
then have $suminf\ (\lambda i.\ f\ i - g\ i) \neq \infty$
using *fin* **by** *auto*
ultimately show *?thesis*
using *assms* ‹$\bigwedge i.\ 0 \leq f\ i$›
apply *simp*
apply (*subst* (*1 2 3*) *suminf-ereal*)
apply (*auto intro*!: *suminf-diff*[*symmetric*] *summable-ereal*)
done
qed

lemma *suminf-ereal-PInf* [*simp*]: $(\sum x.\ \infty :: ereal) = \infty$
by (*metis ereal-less-eq(1) suminf-PInfty*)

lemma *summable-real-of-ereal*:
fixes $f :: nat \Rightarrow ereal$
assumes *f*: $\bigwedge i.\ 0 \leq f\ i$
and *fin*: $(\sum i.\ f\ i) \neq \infty$
shows $summable\ (\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i))$
proof (*rule summable-def*[*THEN iffD2*])
have $0 \leq (\sum i.\ f\ i)$
using *assms* **by** (*auto intro*: *suminf-0-le*)
with *fin* **obtain** *r* **where** *r*: $ereal\ r = (\sum i.\ f\ i)$
by (*cases* $(\sum i.\ f\ i)$) *auto*
have *fin*: $|f\ i| \neq \infty$ **for** i
by (*simp add*: *assms(2) f suminf-PInfty*)
have $(\lambda i.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ i)))\ sums\ (\sum i.\ ereal\ (real\text{-}of\text{-}ereal\ (f\ i)))$
using *f*
by (*auto intro*!: *summable-ereal-pos simp*: *ereal-le-real-iff zero-ereal-def*)
also have $\ldots = ereal\ r$
using *fin r* **by** (*auto simp*: *ereal-real*)
finally show $\exists r.\ (\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i))\ sums\ r$
by (*auto simp*: *sums-ereal*)
qed

lemma *suminf-SUP-eq*:
fixes $f :: nat \Rightarrow nat \Rightarrow ereal$
assumes $\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)$

```
    and ⋀n i. 0 ≤ f n i
  shows (∑ i. SUP n. f n i) = (SUP n. ∑ i. f n i)
proof −
  have ∗: ⋀n. (∑ i<n. SUP k. f k i) = (SUP k. ∑ i<n. f k i)
    using assms
    by (auto intro!: SUP-ereal-sum [symmetric])
  show ?thesis
    using assms
    by (auto simp: suminf-ereal-eq-SUP SUP-upper2 ∗ intro!: SUP-commute)
qed

lemma suminf-sum-ereal:
  fixes f :: - ⇒ - ⇒ ereal
  assumes nonneg: ⋀i a. a ∈ A ⟹ 0 ≤ f i a
  shows (∑ i. ∑ a∈A. f i a) = (∑ a∈A. ∑ i. f i a)
  using nonneg
by (induction A rule: infinite-finite-induct; simp add: suminf-add-ereal sum-nonneg)

lemma suminf-ereal-eq-0:
  fixes f :: nat ⇒ ereal
  assumes nneg: ⋀i. 0 ≤ f i
  shows (∑ i. f i) = 0 ⟷ (∀ i. f i = 0)
proof
  assume (∑ i. f i) = 0
  {
    fix i
    assume f i ≠ 0
    with nneg have 0 < f i
      by (auto simp: less-le)
    also have f i = (∑ j. if j = i then f i else 0)
      by (subst suminf-finite[where N={i}]) auto
    also have ... ≤ (∑ i. f i)
      using nneg
      by (auto intro!: suminf-le-pos)
    finally have False
      using ‹(∑ i. f i) = 0› by auto
  }
  then show ∀ i. f i = 0
    by auto
qed simp

lemma suminf-ereal-offset-le:
  fixes f :: nat ⇒ ereal
  assumes f: ⋀i. 0 ≤ f i
  shows (∑ i. f (i + k)) ≤ suminf f
proof −
  have (λn. ∑ i<n. f (i + k)) ⟶ (∑ i. f (i + k))
    using summable-sums[OF summable-ereal-pos]
    by (simp add: sums-def atLeast0LessThan f)
```

```
  moreover have (λn. ∑ i<n. f i) ⟶ (∑ i. f i)
    using summable-sums[OF summable-ereal-pos]
    by (simp add: sums-def atLeast0LessThan f)
  then have (λn. ∑ i<n + k. f i) ⟶ (∑ i. f i)
    by (rule LIMSEQ-ignore-initial-segment)
  ultimately show ?thesis
  proof (rule LIMSEQ-le, safe intro!: exI[of - k])
    fix n assume k ≤ n
    have (∑ i<n. f (i + k)) = (∑ i<n. (f ∘ plus k) i)
      by (simp add: ac-simps)
    also have ... = (∑ i∈(plus k) ` {..<n}. f i)
      by (rule sum.reindex [symmetric]) simp
    also have ... ≤ sum f {..<n + k}
      by (intro sum-mono2) (auto simp: f)
    finally show (∑ i<n. f (i + k)) ≤ sum f {..<n + k} .
  qed
qed

lemma sums-suminf-ereal: f sums x ⟹ (∑ i. ereal (f i)) = ereal x
  by (metis sums-ereal sums-unique)

lemma suminf-ereal': summable f ⟹ (∑ i. ereal (f i)) = ereal (∑ i. f i)
  by (metis sums-ereal sums-unique summable-def)

lemma suminf-ereal-finite: summable f ⟹ (∑ i. ereal (f i)) ≠ ∞
  by (auto simp: summable-def simp flip: sums-ereal sums-unique)

lemma suminf-ereal-finite-neg:
  assumes summable f
  shows (∑ x. ereal (f x)) ≠ −∞
  by (simp add: assms suminf-ereal')

lemma SUP-ereal-add-directed:
  fixes f g :: 'a ⇒ ereal
  assumes nonneg: ⋀i. i ∈ I ⟹ 0 ≤ f i ⋀i. i ∈ I ⟹ 0 ≤ g i
  assumes directed: ⋀i j. i ∈ I ⟹ j ∈ I ⟹ ∃k∈I. f i + g j ≤ f k + g k
  shows (SUP i∈I. f i + g i) = (SUP i∈I. f i) + (SUP i∈I. g i)
proof cases
  assume I = {} then show ?thesis
    by (simp add: bot-ereal-def)
next
  assume I ≠ {}
  show ?thesis
  proof (rule antisym)
    show (SUP i∈I. f i + g i) ≤ (SUP i∈I. f i) + (SUP i∈I. g i)
      by (rule SUP-least; intro add-mono SUP-upper)
  next
    have bot < (SUP i∈I. g i)
      using ‹I ≠ {}› nonneg(2) by (auto simp: bot-ereal-def less-SUP-iff)
```

```
    then have (SUP i∈I. f i) + (SUP i∈I. g i) = (SUP i∈I. f i + (SUP i∈I. g
i))
      by (intro SUP-ereal-add-left[symmetric] ‹I ≠ {}›) auto
    also have ... = (SUP i∈I. (SUP j∈I. f i + g j))
      using nonneg(1) ‹I ≠ {}› by (simp add: SUP-ereal-add-right)
    also have ... ≤ (SUP i∈I. f i + g i)
      using directed by (intro SUP-least) (blast intro: SUP-upper2)
    finally show (SUP i∈I. f i) + (SUP i∈I. g i) ≤ (SUP i∈I. f i + g i) .
  qed
qed

lemma SUP-ereal-sum-directed:
  fixes f g :: 'a ⇒ 'b ⇒ ereal
  assumes I ≠ {}
  assumes directed: ⋀N i j. N ⊆ A ⟹ i ∈ I ⟹ j ∈ I ⟹ ∃k∈I. ∀n∈N. f n i
≤ f n k ∧ f n j ≤ f n k
  assumes nonneg: ⋀n i. i ∈ I ⟹ n ∈ A ⟹ 0 ≤ f n i
  shows (SUP i∈I. ∑ n∈A. f n i) = (∑ n∈A. SUP i∈I. f n i)
proof −
  have N ⊆ A ⟹ (SUP i∈I. ∑ n∈N. f n i) = (∑ n∈N. SUP i∈I. f n i) for N
  proof (induction N rule: infinite-finite-induct)
    case (insert n N)
     have (SUP i∈I. f n i + (∑ l∈N. f l i)) = (SUP i∈I. f n i) + (SUP i∈I.
∑ l∈N. f l i)
    proof (rule SUP-ereal-add-directed)
      fix i assume i ∈ I then show 0 ≤ f n i 0 ≤ (∑ l∈N. f l i)
        using insert by (auto intro!: sum-nonneg nonneg)
    next
      fix i j assume i ∈ I j ∈ I
      from directed[OF insert(4) this]
      show ∃k∈I. f n i + (∑ l∈N. f l j) ≤ f n k + (∑ l∈N. f l k)
        by (auto intro!: add-mono sum-mono)
    qed
    with insert show ?case
      by simp
  qed (simp-all add: SUP-constant ‹I ≠ {}›)
  from this[of A] show ?thesis by simp
qed

lemma suminf-SUP-eq-directed:
  fixes f :: - ⇒ nat ⇒ ereal
  assumes I ≠ {}
  assumes directed: ⋀N i j. i ∈ I ⟹ j ∈ I ⟹ finite N ⟹ ∃k∈I. ∀n∈N. f i n
≤ f k n ∧ f j n ≤ f k n
  assumes nonneg: ⋀n i. 0 ≤ f n i
  shows (∑ i. SUP n∈I. f n i) = (SUP n∈I. ∑ i. f n i)
proof (subst (1 2) suminf-ereal-eq-SUP)
  show ⋀n i. 0 ≤ f n i ⋀i. 0 ≤ (SUP n∈I. f n i)
    using ‹I ≠ {}› nonneg by (auto intro: SUP-upper2)
```

show (*SUP n.* $\sum i{<}n$*. SUP n∈I. f n i*) = (*SUP n∈I. SUP j.* $\sum i{<}j$*. f n i*)
by (*auto simp*: *finite-subset SUP-commute SUP-ereal-sum-directed assms*)
qed

lemma *ereal-dense3*:
fixes *x y* :: *ereal*
shows $x < y \Longrightarrow \exists r{::}rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
proof (*cases x y rule*: *ereal2-cases*, *simp-all*)
fix *r q* :: *real*
assume $r < q$
from *Rats-dense-in-real*[*OF this*] **show** $\exists x.\ r < real\text{-}of\text{-}rat\ x \wedge real\text{-}of\text{-}rat\ x < q$
by (*fastforce simp*: *Rats-def*)
next
fix *r* :: *real*
show $\exists x.\ r < real\text{-}of\text{-}rat\ x\ \exists x.\ real\text{-}of\text{-}rat\ x < r$
using *gt-ex*[*of r*] *lt-ex*[*of r*] *Rats-dense-in-real*
by (*auto simp*: *Rats-def*)
qed

lemma *continuous-within-ereal*[*intro*, *simp*]: $x \in A \Longrightarrow$ *continuous* (*at x within A*) *ereal*
using *continuous-on-eq-continuous-within*[*of A ereal*]
by (*auto intro*: *continuous-on-ereal continuous-on-id*)

lemma *ereal-open-uminus*:
fixes *S* :: *ereal set*
assumes *open S*
shows *open* (*uminus ‘ S*)
using ‹*open S*›[*unfolded open-generated-order*]
proof *induct*
have *range uminus* = (*UNIV* :: *ereal set*)
by (*auto simp*: *image-iff ereal-uminus-eq-reorder*)
then show *open* (*range uminus* :: *ereal set*)
by *simp*
qed (*auto simp*: *image-Union image-Int*)

lemma *ereal-uminus-complement*:
fixes *S* :: *ereal set*
shows *uminus ‘* (− *S*) = − *uminus ‘ S*
by (*auto intro*!: *bij-image-Compl-eq surjI*[*of - uminus*] *simp*: *bij-betw-def*)

lemma *ereal-closed-uminus*:
fixes *S* :: *ereal set*
assumes *closed S*
shows *closed* (*uminus ‘ S*)
using *assms*
unfolding *closed-def ereal-uminus-complement*[*symmetric*]
by (*rule ereal-open-uminus*)

lemma *ereal-open-affinity-pos*:
fixes S :: *ereal set*
assumes *open* S
and m: $m \neq \infty$ $0 < m$
and t: $|t| \neq \infty$
shows *open* $((\lambda x.\ m * x + t)\ `\ S)$
proof −
have *continuous-on UNIV* $(\lambda x.\ inverse\ m * (x + -\ t))$
using m t
by (*intro continuous-at-imp-continuous-on ballI continuous-at*[*THEN iffD2*]; *force*)
then have *open* $((\lambda x.\ inverse\ m * (x + -t)) -`\ S)$
using ‹*open S*› *open-vimage* **by** *blast*
also have $(\lambda x.\ inverse\ m * (x + -t)) -`\ S = (\lambda x.\ (x - t)\ /\ m) -`\ S$
using m t **by** (*auto simp*: *divide-ereal-def mult.commute minus-ereal-def simp flip*: *uminus-ereal.simps*)
also have $(\lambda x.\ (x - t)\ /\ m) -`\ S = (\lambda x.\ m * x + t)\ `\ S$
using m t
by (*simp add*: *set-eq-iff image-iff*)
(*metis abs-ereal-less0 abs-ereal-uminus ereal-divide-eq ereal-eq-minus ereal-minus*(7,8) *ereal-minus-less-minus ereal-mult-eq-PInfty ereal-uminus-uminus ereal-zero-mult*)
finally show *?thesis* **.**
qed

lemma *ereal-open-affinity*:
fixes S :: *ereal set*
assumes *open* S
and m: $|m| \neq \infty$ $m \neq 0$
and t: $|t| \neq \infty$
shows *open* $((\lambda x.\ m * x + t)\ `\ S)$
proof *cases*
assume $0 < m$
then show *?thesis*
using *ereal-open-affinity-pos*[*OF* ‹*open S*› - - *t*, *of m*] m
by *auto*
next
assume ¬ $0 < m$ **then**
have $0 < -m$
using ‹$m \neq 0$›
by (*cases m*) *auto*
then have m: $-m \neq \infty$ $0 < -m$
using ‹$|m| \neq \infty$›
by (*auto simp*: *ereal-uminus-eq-reorder*)
from *ereal-open-affinity-pos*[*OF ereal-open-uminus*[*OF* ‹*open S*›] m t] **show** *?thesis*
unfolding *image-image* **by** *simp*
qed

lemma *open-uminus-iff*:

fixes S :: *ereal set*
shows *open* (*uminus ' S*) $\longleftrightarrow$ *open S*
using *ereal-open-uminus*[*of S*] *ereal-open-uminus*[*of uminus ' S*]
by *auto*

lemma *ereal-Liminf-uminus*:
fixes f :: $'a \Rightarrow$ *ereal*
shows *Liminf net* ($\lambda x.\ -\ (f\ x)$) = $-$ *Limsup net f*
using *ereal-Limsup-uminus*[*of* - ($\lambda x.\ -\ (f\ x)$)] **by** *auto*

lemma *Liminf-PInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes $\neg$ *trivial-limit net*
shows ($f \longrightarrow \infty$) *net* $\longleftrightarrow$ *Liminf net f* = ∞
unfolding *tendsto-iff-Liminf-eq-Limsup*[*OF assms*]
using *Liminf-le-Limsup*[*OF assms*, *of f*]
by *auto*

lemma *Limsup-MInfty*:
fixes f :: $'a \Rightarrow$ *ereal*
assumes $\neg$ *trivial-limit net*
shows ($f \longrightarrow -\infty$) *net* $\longleftrightarrow$ *Limsup net f* = $-\infty$
unfolding *tendsto-iff-Liminf-eq-Limsup*[*OF assms*]
using *Liminf-le-Limsup*[*OF assms*, *of f*]
by *auto*

lemma *convergent-ereal*: — RENAME
fixes X :: *nat* $\Rightarrow$ $'a$ :: {*complete-linorder*,*linorder-topology*}
shows *convergent X* $\longleftrightarrow$ *limsup X* = *liminf X*
using *tendsto-iff-Liminf-eq-Limsup*[*of sequentially*]
by (*auto simp*: *convergent-def*)

lemma *limsup-le-liminf-real*:
fixes X :: *nat* $\Rightarrow$ *real* **and** L :: *real*
assumes *1*: *limsup X* $\leq$ *L* **and** *2*: *L* $\leq$ *liminf X*
shows *X* $\longrightarrow$ *L*
proof $-$
from *1 2* **have** *limsup X* $\leq$ *liminf X* **by** *auto*
hence *3*: *limsup X* = *liminf X*
by (*simp add*: *Liminf-le-Limsup order-class.order.antisym*)
hence *4*: *convergent* ($\lambda n.$ *ereal* (*X n*))
by (*subst convergent-ereal*)
hence *limsup X* = *lim* ($\lambda n.$ *ereal*(*X n*))
by (*rule convergent-limsup-cl*)
also from *1 2 3* **have** *limsup X* = *L* **by** *auto*
finally have *lim* ($\lambda n.$ *ereal*(*X n*)) = *L* **..**
hence ($\lambda n.$ *ereal* (*X n*)) $\longrightarrow$ *L*
using *4 convergent-LIMSEQ-iff* **by** *force*
thus *?thesis* **by** *simp*

qed

lemma *liminf-PInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow \infty \longleftrightarrow liminf\ X = \infty$
by (*metis Liminf-PInfty trivial-limit-sequentially*)

lemma *limsup-MInfty*:
fixes $X :: nat \Rightarrow ereal$
shows $X \longrightarrow -\infty \longleftrightarrow limsup\ X = -\infty$
by (*metis Limsup-MInfty trivial-limit-sequentially*)

lemma *SUP-eq-LIMSEQ*:
assumes *mono f*
shows $(SUP\ n.\ ereal\ (f\ n)) = ereal\ x \longleftrightarrow f \longrightarrow x$
proof
have *inc*: *incseq* $(\lambda i.\ ereal\ (f\ i))$
using ‹*mono f*› **unfolding** *mono-def incseq-def* **by** *auto*
{
assume $f \longrightarrow x$
then have $(\lambda i.\ ereal\ (f\ i)) \longrightarrow ereal\ x$
by *auto*
from *SUP-Lim*[*OF inc this*] **show** $(SUP\ n.\ ereal\ (f\ n)) = ereal\ x$ **.**
next
assume $(SUP\ n.\ ereal\ (f\ n)) = ereal\ x$
with *LIMSEQ-SUP*[*OF inc*] **show** $f \longrightarrow x$ **by** *auto*
}
qed

lemma *liminf-ereal-cminus*:
fixes $f :: nat \Rightarrow ereal$
assumes $c \neq -\infty$
shows $liminf\ (\lambda x.\ c - f\ x) = c - limsup\ f$
proof (*cases c*)
case *PInf*
then show *?thesis*
by (*simp add*: *Liminf-const*)
next
case (*real r*)
then show *?thesis*
by (*simp add*: *liminf-SUP-INF limsup-INF-SUP INF-ereal-minus-right SUP-ereal-minus-right*)
qed (*use* ‹$c \neq -\infty$› **in** *simp*)

38.6.3 Continuity

lemma *continuous-at-of-ereal*:
$|x0 :: ereal| \neq \infty \Longrightarrow continuous\ (at\ x0)\ real\text{-}of\text{-}ereal$
unfolding *continuous-at*
by (*rule lim-real-of-ereal*) (*simp add*: *ereal-real*)

lemma *nhds-ereal*: *nhds* (*ereal r*) = *filtermap ereal* (*nhds r*)
by (*simp add*: *filtermap-nhds-open-map open-ereal continuous-at-of-ereal*)

lemma *at-ereal*: *at* (*ereal r*) = *filtermap ereal* (*at r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma *at-left-ereal*: *at-left* (*ereal r*) = *filtermap ereal* (*at-left r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma *at-right-ereal*: *at-right* (*ereal r*) = *filtermap ereal* (*at-right r*)
by (*simp add*: *filter-eq-iff eventually-at-filter nhds-ereal eventually-filtermap*)

lemma
shows *at-left-PInf*: *at-left* ∞ = *filtermap ereal at-top*
and *at-right-MInf*: *at-right* ($-\infty$) = *filtermap ereal at-bot*
unfolding *filter-eq-iff eventually-filtermap eventually-at-top-dense eventually-at-bot-dense*
eventually-at-left[*OF ereal-less*(*5*)] *eventually-at-right*[*OF ereal-less*(*6*)]
by (*auto simp*: *ereal-all-split ereal-ex-split*)

lemma *ereal-tendsto-simps1*:
((f ∘ *real-of-ereal*) ⟶ y) (*at-left* (*ereal x*)) ⟷ (f ⟶ y) (*at-left x*)
((f ∘ *real-of-ereal*) ⟶ y) (*at-right* (*ereal x*)) ⟷ (f ⟶ y) (*at-right x*)
((f ∘ *real-of-ereal*) ⟶ y) (*at-left* (∞::*ereal*)) ⟷ (f ⟶ y) *at-top*
((f ∘ *real-of-ereal*) ⟶ y) (*at-right* ($-\infty$::*ereal*)) ⟷ (f ⟶ y) *at-bot*
unfolding *tendsto-compose-filtermap at-left-ereal at-right-ereal at-left-PInf at-right-MInf*
by (*auto simp*: *filtermap-filtermap filtermap-ident*)

lemma *ereal-tendsto-simps2*:
((*ereal* ∘ f) ⟶ *ereal a*) F ⟷ (f ⟶ a) F
((*ereal* ∘ f) ⟶ ∞) F ⟷ (*LIM x F*. f x :> *at-top*)
((*ereal* ∘ f) ⟶ $-\infty$) F ⟷ (*LIM x F*. f x :> *at-bot*)
unfolding *tendsto-PInfty filterlim-at-top-dense tendsto-MInfty filterlim-at-bot-dense*
using *lim-ereal* **by** (*simp-all add*: *comp-def*)

lemma *inverse-infty-ereal-tendsto-0*: *inverse* $-\infty\rightarrow$ (*0*::*ereal*)
proof −
have ∗∗: ((λ*x*. *ereal* (*inverse x*)) ⟶ *ereal 0*) *at-infinity*
by (*intro tendsto-intros tendsto-inverse-0*)
then have ((λ*x*. *if x* = *0 then* ∞ *else ereal* (*inverse x*)) ⟶ *0*) *at-top*
proof (*rule filterlim-mono-eventually*)
show *nhds* (*ereal 0*) ≤ *nhds 0*
by (*simp add*: *zero-ereal-def*)
show (*at-top*::*real filter*) ≤ *at-infinity*
by (*simp add*: *at-top-le-at-infinity*)
qed *auto*
then show *?thesis*
unfolding *at-infty-ereal-eq-at-top tendsto-compose-filtermap*[*symmetric*] *comp-def*
by *auto*

qed

lemma *inverse-ereal-tendsto-pos*:
fixes x :: *ereal* **assumes** $0 < x$
shows *inverse* $-x\rightarrow$ *inverse* x
proof (*cases* x)
case (*real* r)
with ‹$0 < x$› **have** $**$: $(\lambda x.$ *ereal* (*inverse* x)) $-r\rightarrow$ *ereal* (*inverse* r)
by (*auto intro*!: *tendsto-inverse*)
from *real* ‹$0 < x$› **show** *?thesis*
by (*auto simp*: *at-ereal tendsto-compose-filtermap*[*symmetric*] *eventually-at-filter*
intro!: *Lim-transform-eventually*[*OF* $**$] *t1-space-nhds*)
qed (*insert* ‹$0 < x$›, *auto intro*!: *inverse-infty-ereal-tendsto-0*)

lemma *inverse-ereal-tendsto-at-right-0*: (*inverse* $\longrightarrow \infty$) (*at-right* (*0*::*ereal*))
unfolding *tendsto-compose-filtermap*[*symmetric*] *at-right-ereal zero-ereal-def*
by (*subst filterlim-cong*[*OF refl refl*, **where** $g{=}\lambda x.$ *ereal* (*inverse* x)])
(*auto simp*: *eventually-at-filter tendsto-PInfty-eq-at-top filterlim-inverse-at-top-right*)

lemmas *ereal-tendsto-simps* = *ereal-tendsto-simps1 ereal-tendsto-simps2*

lemma *continuous-at-iff-ereal*:
fixes f :: $'a$::*t2-space* $\Rightarrow$ *real*
shows *continuous* (*at x0 within s*) $f \longleftrightarrow$ *continuous* (*at x0 within s*) (*ereal* $\circ$ f)
unfolding *continuous-within comp-def lim-ereal* **..**

lemma *continuous-on-iff-ereal*:
fixes f :: $'a$::*t2-space* => *real*
assumes *open A*
shows *continuous-on A f* $\longleftrightarrow$ *continuous-on A* (*ereal* $\circ$ f)
unfolding *continuous-on-def comp-def lim-ereal* **..**

lemma *continuous-on-real*: *continuous-on* (*UNIV* $-$ $\{\infty, -\infty$::*ereal*$\}$) *real-of-ereal*
using *continuous-at-of-ereal continuous-on-eq-continuous-at open-image-ereal*
by *auto*

lemma *continuous-on-iff-real*:
fixes f :: $'a$::*t2-space* $\Rightarrow$ *ereal*
assumes $\bigwedge x.\ x \in A \Longrightarrow |f\ x| \neq \infty$
shows *continuous-on A f* $\longleftrightarrow$ *continuous-on A* (*real-of-ereal* $\circ$ f)
proof
assume *L*: *continuous-on A f*
have f ‘ $A \subseteq$ *UNIV* $-$ $\{\infty, -\infty\}$
using *assms* **by** *force*
then show *continuous-on A* (*real-of-ereal* $\circ$ f)
by (*meson L continuous-on-compose continuous-on-real continuous-on-subset*)
next
assume *R*: *continuous-on A* (*real-of-ereal* $\circ$ f)
then have *continuous-on A* (*ereal* $\circ$ (*real-of-ereal* $\circ$ f))

by (*meson continuous-at-iff-ereal continuous-on-eq-continuous-within*)
then show *continuous-on A f*
using *assms ereal-real'* **by** *auto*
qed

lemma *continuous-uminus-ereal* [*continuous-intros*]: *continuous-on* (*A* :: *ereal set*) *uminus*
unfolding *continuous-on-def*
by (*intro ballI tendsto-uminus-ereal*[*of* λ*x*. *x*::*ereal*]) *simp*

lemma *ereal-uminus-atMost* [*simp*]: *uminus* ' {..(*a*::*ereal*)} = {−*a*..}
proof (*intro equalityI subsetI*)
fix *x* :: *ereal* **assume** *x* ∈ {−*a*..}
hence −(−*x*) ∈ *uminus* ' {..*a*} **by** (*intro imageI*) (*simp add*: *ereal-uminus-le-reorder*)
thus *x* ∈ *uminus* ' {..*a*} **by** *simp*
qed *auto*

lemma *continuous-on-inverse-ereal* [*continuous-intros*]:
continuous-on {*0*::*ereal* ..} *inverse*
unfolding *continuous-on-def*
proof *clarsimp*
fix *x* :: *ereal* **assume** *0* ≤ *x*
moreover have *at 0 within* {*0* ..} = *at-right* (*0*::*ereal*)
by (*auto simp*: *filter-eq-iff eventually-at-filter le-less*)
moreover have *at x within* {*0* ..} = *at x* **if** *0* < *x*
using *that* **by** (*intro at-within-nhd*[*of* - {*0*<..}]) *auto*
ultimately show (*inverse* ⟶ *inverse x*) (*at x within* {*0*..})
by (*auto simp*: *le-less inverse-ereal-tendsto-at-right-0 inverse-ereal-tendsto-pos*)
qed

lemma *continuous-inverse-ereal-nonpos*: *continuous-on* ({..<*0*} :: *ereal set*) *inverse*
proof (*subst continuous-on-cong*[*OF refl*])
have *continuous-on* {(*0*::*ereal*)<..} *inverse*
by (*rule continuous-on-subset*[*OF continuous-on-inverse-ereal*]) *auto*
thus *continuous-on* {..<(*0*::*ereal*)} (*uminus* ∘ *inverse* ∘ *uminus*)
by (*intro continuous-intros*) *simp-all*
qed *simp*

lemma *tendsto-inverse-ereal*:
assumes (*f* ⟶ (*c* :: *ereal*)) *F*
assumes *eventually* (λ*x*. *f x* ≥ *0*) *F*
shows ((λ*x*. *inverse* (*f x*)) ⟶ *inverse c*) *F*
by (*cases F* = *bot*)
(*auto intro*!: *tendsto-lowerbound assms*
continuous-on-tendsto-compose[*OF continuous-on-inverse-ereal*])

38.6.4 liminf and limsup

lemma *Limsup-ereal-mult-right*:
assumes $F \neq bot$ $(c::real) \geq 0$
shows $Limsup\ F\ (\lambda n.\ f\ n * ereal\ c) = Limsup\ F\ f * ereal\ c$
proof (*rule Limsup-compose-continuous-mono*)
from *assms* **show** *continuous-on UNIV* $(\lambda a.\ a * ereal\ c)$
using *tendsto-cmult-ereal*[*of ereal c* $\lambda x.\ x$]
by (*force simp*: *continuous-on-def mult-ac*)
qed (*use assms* **in** ‹*auto simp*: *mono-def ereal-mult-right-mono*›)

lemma *Liminf-ereal-mult-right*:
assumes $F \neq bot$ $(c::real) \geq 0$
shows $Liminf\ F\ (\lambda n.\ f\ n * ereal\ c) = Liminf\ F\ f * ereal\ c$
proof (*rule Liminf-compose-continuous-mono*)
from *assms* **show** *continuous-on UNIV* $(\lambda a.\ a * ereal\ c)$
using *tendsto-cmult-ereal*[*of ereal c* $\lambda x.\ x$]
by (*force simp*: *continuous-on-def mult-ac*)
qed (*use assms* **in** ‹*auto simp*: *mono-def ereal-mult-right-mono*›)

lemma *Liminf-ereal-mult-left*:
assumes $F \neq bot$ $(c::real) \geq 0$
shows $Liminf\ F\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * Liminf\ F\ f$
using *Liminf-ereal-mult-right*[*OF assms*] **by** (*subst* (*1 2*) *mult.commute*)

lemma *Limsup-ereal-mult-left*:
assumes $F \neq bot$ $(c::real) \geq 0$
shows $Limsup\ F\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * Limsup\ F\ f$
using *Limsup-ereal-mult-right*[*OF assms*] **by** (*subst* (*1 2*) *mult.commute*)

lemma *limsup-ereal-mult-right*:
$(c::real) \geq 0 \Longrightarrow limsup\ (\lambda n.\ f\ n * ereal\ c) = limsup\ f * ereal\ c$
by (*rule Limsup-ereal-mult-right*) *simp-all*

lemma *limsup-ereal-mult-left*:
$(c::real) \geq 0 \Longrightarrow limsup\ (\lambda n.\ ereal\ c * f\ n) = ereal\ c * limsup\ f$
by (*simp add*: *Limsup-ereal-mult-left*)

lemma *Limsup-add-ereal-right*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ g\ n + (c :: ereal)) = Limsup\ F\ g + c$
by (*rule Limsup-compose-continuous-mono*) (*auto simp*: *mono-def add-mono continuous-on-def*)

lemma *Limsup-add-ereal-left*:
$F \neq bot \Longrightarrow abs\ c \neq \infty \Longrightarrow Limsup\ F\ (\lambda n.\ (c :: ereal) + g\ n) = c + Limsup\ F\ g$
by (*subst* (*1 2*) *add.commute*) (*rule Limsup-add-ereal-right*)

lemma *Liminf-add-ereal-right*:

F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Liminf F* (λ*n*. *g n* + (*c* :: *ereal*)) = *Liminf F g* + *c*
by (*rule Liminf-compose-continuous-mono*) (*auto simp*: *mono-def add-mono continuous-on-def*)

lemma *Liminf-add-ereal-left*:
F ≠ *bot* ⟹ *abs c* ≠ ∞ ⟹ *Liminf F* (λ*n*. (*c* :: *ereal*) + *g n*) = *c* + *Liminf F g*
by (*subst* (*1 2*) *add.commute*) (*rule Liminf-add-ereal-right*)

lemma
assumes *F* ≠ *bot*
assumes *nonneg*: *eventually* (λ*x*. *f x* ≥ (*0*::*ereal*)) *F*
shows *Liminf-inverse-ereal*: *Liminf F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Limsup F f*)
and *Limsup-inverse-ereal*: *Limsup F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Liminf F f*)
proof −
define *inv* **where** [*abs-def*]: *inv x* = (*if x* ≤ *0 then* ∞ *else inverse x*) **for** *x* :: *ereal*
have *continuous-on* ({..*0*} ∪ {*0*..}) *inv* **unfolding** *inv-def*
by (*intro continuous-on-If*) (*auto intro*!: *continuous-intros*)
also have {..*0*} ∪ {*0*..} = (*UNIV* :: *ereal set*) **by** *auto*
finally have *cont*: *continuous-on UNIV inv* **.**
have *antimono*: *antimono inv* **unfolding** *inv-def antimono-def*
by (*auto intro*!: *ereal-inverse-antimono*)

have *Liminf F* (λ*x*. *inverse* (*f x*)) = *Liminf F* (λ*x*. *inv* (*f x*)) **using** *nonneg*
by (*auto intro*!: *Liminf-eq elim*!: *eventually-mono simp*: *inv-def*)
also have ... = *inv* (*Limsup F f*)
by (*simp add*: *assms*(*1*) *Liminf-compose-continuous-antimono*[*OF cont antimono*])
also from *assms* **have** *Limsup F f* ≥ *0* **by** (*intro le-Limsup*) *simp-all*
hence *inv* (*Limsup F f*) = *inverse* (*Limsup F f*) **by** (*simp add*: *inv-def*)
finally show *Liminf F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Limsup F f*) **.**

have *Limsup F* (λ*x*. *inverse* (*f x*)) = *Limsup F* (λ*x*. *inv* (*f x*)) **using** *nonneg*
by (*auto intro*!: *Limsup-eq elim*!: *eventually-mono simp*: *inv-def*)
also have ... = *inv* (*Liminf F f*)
by (*simp add*: *assms*(*1*) *Limsup-compose-continuous-antimono*[*OF cont antimono*])
also from *assms* **have** *Liminf F f* ≥ *0* **by** (*intro Liminf-bounded*) *simp-all*
hence *inv* (*Liminf F f*) = *inverse* (*Liminf F f*) **by** (*simp add*: *inv-def*)
finally show *Limsup F* (λ*x*. *inverse* (*f x*)) = *inverse* (*Liminf F f*) **.**
qed

lemma *ereal-diff-le-mono-left*: ⟦ *x* ≤ *z*; *0* ≤ *y* ⟧ ⟹ *x* − *y* ≤ (*z* :: *ereal*)
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *neg-0-less-iff-less-erea* [*simp*]: *0* < − *a* ⟷ (*a* :: *ereal*) < *0*
by(*cases a*) *simp-all*

lemma *not-infty-ereal*: $|x| \neq \infty \longleftrightarrow (\exists\, x'.\ x = ereal\ x')$
by *auto*

lemma *neq-PInf-trans*: **fixes** $x\ y :: ereal$ **shows** $[\![\ y \neq \infty;\ x \leq y\ ]\!] \Longrightarrow x \neq \infty$
by *auto*

lemma *mult-2-ereal*: $ereal\ 2 * x = x + x$
by(*cases x*) *simp-all*

lemma *ereal-diff-le-self*: $0 \leq y \Longrightarrow x - y \leq (x :: ereal)$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-le-add-self*: $0 \leq y \Longrightarrow x \leq x + (y :: ereal)$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-le-add-self2*: $0 \leq y \Longrightarrow x \leq y + (x :: ereal)$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-diff-nonpos*:
fixes $a\ b :: ereal$ **shows** $[\![\ a \leq b;\ a = \infty \Longrightarrow b \neq \infty;\ a = -\infty \Longrightarrow b \neq -\infty\ ]\!]$
$\Longrightarrow a - b \leq 0$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *minus-ereal-0* [*simp*]: $x - ereal\ 0 = x$
by(*simp flip*: *zero-ereal-def*)

lemma *ereal-diff-eq-0-iff*: **fixes** $a\ b :: ereal$
shows $(|a| = \infty \Longrightarrow |b| \neq \infty) \Longrightarrow a - b = 0 \longleftrightarrow a = b$
by(*cases a b rule*: *ereal2-cases*) *simp-all*

lemma *SUP-ereal-eq-0-iff-nonneg*:
fixes $f :: - \Rightarrow ereal$ **and** A
assumes *nonneg*: $\forall\, x{\in}A.\ f\ x \geq 0$
and $A{:}A \neq \{\}$
shows $(SUP\ x{\in}A.\ f\ x) = 0 \longleftrightarrow (\forall\, x{\in}A.\ f\ x = 0)$ (**is** *?lhs* $\longleftrightarrow$ -)
proof(*intro iffI ballI*)
fix x
assume *?lhs* $x \in A$
from ‹$x \in A$› **have** $f\ x \leq (SUP\ x{\in}A.\ f\ x)$ **by**(*rule SUP-upper*)
with ‹*?lhs*› **show** $f\ x = 0$ **using** *nonneg* ‹$x \in A$› **by** *auto*
qed (*simp add*: A)

lemma *ereal-divide-le-posI*:
fixes $x\ y\ z :: ereal$
shows $x > 0 \Longrightarrow z \neq -\infty \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
by (*cases rule*: *ereal3-cases*[*of x y z*])(*auto simp*: *field-simps split*: *if-split-asm*)

lemma *add-diff-eq-ereal*:

fixes $x\ y\ z$ :: *ereal*
shows $x + (y - z) = x + y - z$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *ereal-diff-gr0*:
fixes $a\ b$ :: *ereal*
shows $a < b \Longrightarrow 0 < b - a$
by (*cases rule*: *ereal2-cases*[*of a b*]) *auto*

lemma *ereal-minus-minus*:
fixes $x\ y\ z$ :: *ereal* **shows**
$(|y| = \infty \Longrightarrow |z| \neq \infty) \Longrightarrow x - (y - z) = x + z - y$
by(*cases x y z rule*: *ereal3-cases*) *simp-all*

lemma *diff-diff-commute-ereal*:
fixes $x\ y\ z$ :: *ereal*
shows $x - y - z = x - z - y$
by (*metis add-diff-eq-ereal ereal-add-uminus-conv-diff*)

lemma *ereal-diff-eq-MInfty-iff*:
fixes $x\ y$ :: *ereal*
shows $x - y = -\infty \longleftrightarrow x = -\infty \wedge y \neq -\infty \vee y = \infty \wedge |x| \neq \infty$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *ereal-diff-add-inverse*:
fixes $x\ y$ :: *ereal*
shows $|x| \neq \infty \Longrightarrow x + y - x = y$
by(*cases x y rule*: *ereal2-cases*) *simp-all*

lemma *tendsto-diff-ereal*:
fixes $x\ y$ :: *ereal*
assumes *x*: $|x| \neq \infty$ **and** *y*: $|y| \neq \infty$
assumes *f*: $(f \longrightarrow x)\ F$ **and** *g*: $(g \longrightarrow y)\ F$
shows $((\lambda x.\ f\ x - g\ x) \longrightarrow x - y)\ F$
proof −
from *x* **obtain** *r* **where** *x'*: $x = ereal\ r$ **by** (*cases x*) *auto*
with *f* **have** $((\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i)) \longrightarrow r)\ F$ **by** *simp*
moreover
from *y* **obtain** *p* **where** *y'*: $y = ereal\ p$ **by** (*cases y*) *auto*
with *g* **have** $((\lambda i.\ real\text{-}of\text{-}ereal\ (g\ i)) \longrightarrow p)\ F$ **by** *simp*
ultimately have $((\lambda i.\ real\text{-}of\text{-}ereal\ (f\ i) - real\text{-}of\text{-}ereal\ (g\ i)) \longrightarrow r - p)\ F$
by (*rule tendsto-diff*)
moreover
from *eventually-finite*[*OF x f*] *eventually-finite*[*OF y g*]
have *eventually* $(\lambda x.\ f\ x - g\ x = ereal\ (real\text{-}of\text{-}ereal\ (f\ x) - real\text{-}of\text{-}ereal\ (g\ x)))$
F
by *eventually-elim auto*
ultimately show *?thesis*
by (*simp add*: *x' y' cong*: *filterlim-cong*)

qed

lemma *continuous-on-diff-ereal*:
continuous-on A f $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ $(\bigwedge x.\ x \in A \Longrightarrow |f\ x| \neq \infty) \Longrightarrow$ $(\bigwedge x.\ x \in A \Longrightarrow |g\ x| \neq \infty) \Longrightarrow$ *continuous-on A* $(\lambda z.\ f\ z - g\ z :: ereal)$
by (*auto simp*: *tendsto-diff-ereal continuous-on-def*)

38.6.5 Tests for code generator

A small list of simple arithmetic expressions.

value $-\infty$ :: *ereal*
value $|-\infty|$ :: *ereal*
value *4 + 5 / 4 − ereal 2* :: *ereal*
value *ereal 3* $< \infty$
value *real-of-ereal* $(\infty :: ereal) = 0$

end

39 Indicator Function

theory *Indicator-Function*
imports *Complex-Main Disjoint-Sets*
begin

definition *indicator S x* = *of-bool* $(x \in S)$

Type constrained version

abbreviation *indicat-real* :: *'a set* $\Rightarrow$ *'a* $\Rightarrow$ *real* **where** *indicat-real S* $\equiv$ *indicator S*

lemma *indicator-simps*[*simp*]:
$x \in S \Longrightarrow$ *indicator S x* = *1*
$x \notin S \Longrightarrow$ *indicator S x* = *0*
unfolding *indicator-def* **by** *auto*

lemma *indicator-pos-le*[*intro*, *simp*]: $(0 :: 'a :: linordered\text{-}semidom) \leq$ *indicator S x*
and *indicator-le-1*[*intro*, *simp*]: *indicator S x* $\leq (1 :: 'a :: linordered\text{-}semidom)$
unfolding *indicator-def* **by** *auto*

lemma *indicator-abs-le-1*: $|indicator\ S\ x| \leq (1 :: 'a :: linordered\text{-}idom)$
unfolding *indicator-def* **by** *auto*

lemma *indicator-eq-0-iff*: *indicator A x* $= (0 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow x \notin A$
by (*auto simp*: *indicator-def*)

lemma *indicator-eq-1-iff*: *indicator A x* $= (1 :: 'a :: zero\text{-}neq\text{-}one) \longleftrightarrow x \in A$
by (*auto simp*: *indicator-def*)

lemma *indicator-UNIV* [*simp*]: *indicator UNIV* = (λ*x*. *1*)
by *auto*

lemma *indicator-leI*:
(*x* ∈ *A* ⟹ *y* ∈ *B*) ⟹ (*indicator A x* :: *'a*::*linordered-nonzero-semiring*) ≤ *indicator B y*
by (*auto simp*: *indicator-def*)

lemma *split-indicator*: *P* (*indicator S x*) ⟷ ((*x* ∈ *S* ⟶ *P 1*) ∧ (*x* ∉ *S* ⟶ *P 0*))
unfolding *indicator-def* **by** *auto*

lemma *split-indicator-asm*: *P* (*indicator S x*) ⟷ (¬ (*x* ∈ *S* ∧ ¬ *P 1* ∨ *x* ∉ *S* ∧ ¬ *P 0*))
unfolding *indicator-def* **by** *auto*

lemma *indicator-inter-arith*: *indicator* (*A* ∩ *B*) *x* = *indicator A x* ∗ (*indicator B x*::*'a*::*semiring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-union-arith*:
indicator (*A* ∪ *B*) *x* = *indicator A x* + *indicator B x* − *indicator A x* ∗ (*indicator B x* :: *'a*::*ring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-inter-min*: *indicator* (*A* ∩ *B*) *x* = *min* (*indicator A x*) (*indicator B x*::*'a*::*linordered-semidom*)
and *indicator-union-max*: *indicator* (*A* ∪ *B*) *x* = *max* (*indicator A x*) (*indicator B x*::*'a*::*linordered-semidom*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-disj-union*:
A ∩ *B* = {} ⟹ *indicator* (*A* ∪ *B*) *x* = (*indicator A x* + *indicator B x* :: *'a*::*linordered-semidom*)
by (*auto split*: *split-indicator*)

lemma *indicator-compl*: *indicator* (− *A*) *x* = *1* − (*indicator A x* :: *'a*::*ring-1*)
and *indicator-diff*: *indicator* (*A* − *B*) *x* = *indicator A x* ∗ (*1* − *indicator B x* ::*'a*::*ring-1*)
unfolding *indicator-def* **by** (*auto simp*: *min-def max-def*)

lemma *indicator-times*:
indicator (*A* × *B*) *x* = *indicator A* (*fst x*) ∗ (*indicator B* (*snd x*) :: *'a*::*semiring-1*)
unfolding *indicator-def* **by** (*cases x*) *auto*

lemma *indicator-sum*:
indicator (*A* <+> *B*) *x* = (*case x of Inl x* ⇒ *indicator A x* | *Inr x* ⇒ *indicator B x*)
unfolding *indicator-def* **by** (*cases x*) *auto*

lemma *indicator-image*: *inj f* ⟹ *indicator* (*f* ' *X*) (*f x*) = (*indicator X x*::-::*zero-neq-one*)
by (*auto simp*: *indicator-def inj-def*)

lemma *indicator-vimage*: *indicator* (*f* −' *A*) *x* = *indicator A* (*f x*)
by (*auto split*: *split-indicator*)

lemma *mult-indicator-cong*:
fixes *f g* :: - ⇒ *'a* :: *semiring-1*
shows (⋀*x*. *x* ∈ *A* ⟹ *f x* = *g x*) ⟹ *indicator A x* ∗ *f x* = *indicator A x* ∗ *g x*
by (*auto simp*: *indicator-def*)

lemma
fixes *f* :: *'a* ⇒ *'b*::*semiring-1*
assumes *finite A*
shows *sum-mult-indicator*[*simp*]: (∑ *x* ∈ *A*. *f x* ∗ *indicator B x*) = (∑ *x* ∈ *A* ∩ *B*. *f x*)
and *sum-indicator-mult*[*simp*]: (∑ *x* ∈ *A*. *indicator B x* ∗ *f x*) = (∑ *x* ∈ *A* ∩ *B*. *f x*)
unfolding *indicator-def*
using *assms* **by** (*auto intro*!: *sum.mono-neutral-cong-right split*: *if-split-asm*)

lemma *sum-indicator-eq-card*:
assumes *finite A*
shows (∑ *x* ∈ *A*. *indicator B x*) = *card* (*A Int B*)
using *sum-mult-indicator* [*OF assms*, *of* λ*x*. *1*::*nat*]
unfolding *card-eq-sum* **by** *simp*

lemma *sum-indicator-scaleR*[*simp*]:
finite A ⟹
(∑ *x* ∈ *A*. *indicator* (*B x*) (*g x*) ∗$_R$ *f x*) = (∑ *x* ∈ {*x*∈*A*. *g x* ∈ *B x*}. *f x* :: *'a*::*real-vector*)
by (*auto intro*!: *sum.mono-neutral-cong-right split*: *if-split-asm simp*: *indicator-def*)

lemma *LIMSEQ-indicator-incseq*:
assumes *incseq A*
shows (λ*i*. *indicator* (*A i*) *x* :: *'a*::{*topological-space*,*zero-neq-one*}) ⟶ *indicator* (⋃ *i*. *A i*) *x*
proof (*cases* ∃ *i*. *x* ∈ *A i*)
case *True*
then obtain *i* **where** *x* ∈ *A i*
by *auto*
then have ∗:
⋀*n*. (*indicator* (*A* (*n* + *i*)) *x* :: *'a*) = *1*
(*indicator* (⋃ *i*. *A i*) *x* :: *'a*) = *1*
using *incseqD*[*OF* ‹*incseq A*›, *of i n* + *i* **for** *n*] ‹*x* ∈ *A i*› **by** (*auto simp*: *indicator-def*)
show *?thesis*

by (*rule LIMSEQ-offset*[*of* - *i*]) (*use* ∗ **in** *simp*)
next
case *False*
then show *?thesis* **by** (*simp add*: *indicator-def*)
qed

lemma *LIMSEQ-indicator-UN*:
(λ*k*. *indicator* (⋃ *i*<*k*. *A i*) *x* :: *'a*::{*topological-space*,*zero-neq-one*}) ⟶ *indicator* (⋃ *i*. *A i*) *x*
proof −
have (λ*k*. *indicator* (⋃ *i*<*k*. *A i*) *x*::*'a*) ⟶ *indicator* (⋃ *k*. ⋃ *i*<*k*. *A i*) *x*
by (*intro LIMSEQ-indicator-incseq*) (*auto simp*: *incseq-def intro*: *less-le-trans*)
also have (⋃ *k*. ⋃ *i*<*k*. *A i*) = (⋃ *i*. *A i*)
by *auto*
finally show *?thesis* **.**
qed

lemma *LIMSEQ-indicator-decseq*:
assumes *decseq A*
shows (λ*i*. *indicator* (*A i*) *x* :: *'a*::{*topological-space*,*zero-neq-one*}) ⟶ *indicator* (⋂ *i*. *A i*) *x*
proof (*cases* ∃ *i*. *x* ∉ *A i*)
case *True*
then obtain *i* **where** *x* ∉ *A i*
by *auto*
then have ∗:
⋀*n*. (*indicator* (*A* (*n* + *i*)) *x* :: *'a*) = *0*
(*indicator* (⋂ *i*. *A i*) *x* :: *'a*) = *0*
using *decseqD*[*OF* ‹*decseq A*›, *of i n* + *i* **for** *n*] ‹*x* ∉ *A i*› **by** (*auto simp*: *indicator-def*)
show *?thesis*
by (*rule LIMSEQ-offset*[*of* - *i*]) (*use* ∗ **in** *simp*)
next
case *False*
then show *?thesis* **by** (*simp add*: *indicator-def*)
qed

lemma *LIMSEQ-indicator-INT*:
(λ*k*. *indicator* (⋂ *i*<*k*. *A i*) *x* :: *'a*::{*topological-space*,*zero-neq-one*}) ⟶ *indicator* (⋂ *i*. *A i*) *x*
proof −
have (λ*k*. *indicator* (⋂ *i*<*k*. *A i*) *x*::*'a*) ⟶ *indicator* (⋂ *k*. ⋂ *i*<*k*. *A i*) *x*
by (*intro LIMSEQ-indicator-decseq*) (*auto simp*: *decseq-def intro*: *less-le-trans*)
also have (⋂ *k*. ⋂ *i*<*k*. *A i*) = (⋂ *i*. *A i*)
by *auto*
finally show *?thesis* **.**
qed

lemma *indicator-add*:

$A \cap B = \{\} \Longrightarrow (\mathit{indicator}\ A\ x ::\text{-}::\mathit{monoid\text{-}add}) + \mathit{indicator}\ B\ x = \mathit{indicator}\ (A \cup B)\ x$
unfolding *indicator-def* **by** *auto*

lemma *of-real-indicator*: $\mathit{of\text{-}real}\ (\mathit{indicator}\ A\ x) = \mathit{indicator}\ A\ x$
by (*simp split*: *split-indicator*)

lemma *real-of-nat-indicator*: $\mathit{real}\ (\mathit{indicator}\ A\ x :: \mathit{nat}) = \mathit{indicator}\ A\ x$
by (*simp split*: *split-indicator*)

lemma *abs-indicator*: $|\mathit{indicator}\ A\ x :: 'a::\mathit{linordered\text{-}idom}| = \mathit{indicator}\ A\ x$
by (*simp split*: *split-indicator*)

lemma *mult-indicator-subset*:
$A \subseteq B \Longrightarrow \mathit{indicator}\ A\ x * \mathit{indicator}\ B\ x = (\mathit{indicator}\ A\ x :: 'a::\mathit{comm\text{-}semiring\text{-}1})$
by (*auto split*: *split-indicator simp*: *fun-eq-iff*)

lemma *indicator-times-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\mathit{comm\text{-}ring\text{-}1}$
shows $\mathit{indicator}\ S\ x * f\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)$ $f\ x * \mathit{indicator}\ S\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)$
by *auto*

lemma *indicator-scaleR-eq-if*:
fixes $f :: 'a \Rightarrow 'b::\mathit{real\text{-}vector}$
shows $\mathit{indicator}\ S\ x *_R f\ x = (\mathit{if}\ x \in S\ \mathit{then}\ f\ x\ \mathit{else}\ 0)$
by *simp*

lemma *indicator-sums*:
assumes $\bigwedge i\ j.\ i \neq j \Longrightarrow A\ i \cap A\ j = \{\}$
shows $(\lambda i.\ \mathit{indicator}\ (A\ i)\ x::\mathit{real})\ \mathit{sums}\ \mathit{indicator}\ (\bigcup i.\ A\ i)\ x$
proof (*cases* $\exists i.\ x \in A\ i$)
case *True*
then obtain *i* **where** *i*: $x \in A\ i$ **..**
with *assms* **have** $(\lambda i.\ \mathit{indicator}\ (A\ i)\ x::\mathit{real})\ \mathit{sums}\ (\sum i \in \{i\}.\ \mathit{indicator}\ (A\ i)\ x)$
by (*intro sums-finite*) (*auto split*: *split-indicator*)
also have $(\sum i \in \{i\}.\ \mathit{indicator}\ (A\ i)\ x) = \mathit{indicator}\ (\bigcup i.\ A\ i)\ x$
using *i* **by** (*auto split*: *split-indicator*)
finally show *?thesis* **.**
next
case *False*
then show *?thesis* **by** *simp*
qed

The indicator function of the union of a disjoint family of sets is the sum over all the individual indicators.

lemma *indicator-UN-disjoint*:
$\mathit{finite}\ A \Longrightarrow \mathit{disjoint\text{-}family\text{-}on}\ f\ A \Longrightarrow \mathit{indicator}\ (\bigcup (f\ `\ A))\ x = (\sum y \in A.\ \mathit{indicator}\ (f\ y)\ x)$

by (*induct A rule*: *finite-induct*)
(*auto simp*: *disjoint-family-on-def indicator-def split*: *if-splits split-of-bool-asm*)

end

40 The type of non-negative extended real numbers

theory *Extended-Nonnegative-Real*
imports *Extended-Real Indicator-Function*
begin

lemma *ereal-ineq-diff-add*:
assumes $b \neq (-\infty::ereal)$ $a \geq b$
shows $a = b + (a-b)$
by (*metis add.commute assms ereal-eq-minus-iff ereal-minus-le-iff ereal-plus-eq-PInfty*)

lemma *Limsup-const-add*:
fixes $c :: 'a::\{complete\text{-}linorder, linorder\text{-}topology, topological\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\}$
shows $F \neq bot \Longrightarrow Limsup\ F\ (\lambda x.\ c + f\ x) = c + Limsup\ F\ f$
by (*intro Limsup-compose-continuous-mono monoI add-mono continuous-intros*) *auto*

lemma *Liminf-const-add*:
fixes $c :: 'a::\{complete\text{-}linorder, linorder\text{-}topology, topological\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\}$
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ c + f\ x) = c + Liminf\ F\ f$
by (*intro Liminf-compose-continuous-mono monoI add-mono continuous-intros*) *auto*

lemma *Liminf-add-const*:
fixes $c :: 'a::\{complete\text{-}linorder, linorder\text{-}topology, topological\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\}$
shows $F \neq bot \Longrightarrow Liminf\ F\ (\lambda x.\ f\ x + c) = Liminf\ F\ f + c$
by (*intro Liminf-compose-continuous-mono monoI add-mono continuous-intros*) *auto*

lemma *sums-offset*:
fixes $f\ g :: nat \Rightarrow 'a :: \{t2\text{-}space, topological\text{-}comm\text{-}monoid\text{-}add\}$
assumes $(\lambda n.\ f\ (n + i))\ sums\ l$ **shows** $f\ sums\ (l + (\sum j{<}i.\ f\ j))$
proof −
have $(\lambda k.\ (\sum n{<}k.\ f\ (n + i)) + (\sum j{<}i.\ f\ j)) \longrightarrow l + (\sum j{<}i.\ f\ j)$
using *assms* **by** (*auto intro*!: *tendsto-add simp*: *sums-def*)
moreover have $(\sum j{<}k + i.\ f\ j) = (\sum n{<}k.\ f\ (n + i)) + (\sum j{<}i.\ f\ j)$ **for** $k :: nat$
proof −
have $(\sum j{<}k + i.\ f\ j) = (\sum j{=}i..{<}k + i.\ f\ j) + (\sum j{=}0..{<}i.\ f\ j)$

by (*subst sum.union-disjoint*[*symmetric*]) (*auto intro*!: *sum.cong*)
also have $(\sum j{=}i..{<}k + i.\ f\ j) = (\sum j{\in}(\lambda n.\ n + i)`\{0..{<}k\}.\ f\ j)$
unfolding *image-add-atLeastLessThan* **by** *simp*
finally show *?thesis*
by (*auto simp*: *inj-on-def atLeast0LessThan sum.reindex*)
qed
ultimately have $(\lambda k.\ (\sum n{<}k + i.\ f\ n)) \longrightarrow l + (\sum j{<}i.\ f\ j)$
by *simp*
then show *?thesis*
unfolding *sums-def* **by** (*rule LIMSEQ-offset*)
qed

lemma *suminf-offset*:
fixes *f g* :: *nat* ⇒ *'a* :: {*t2-space*, *topological-comm-monoid-add*}
shows *summable* $(\lambda j.\ f\ (j + i)) \Longrightarrow$ *suminf f* $= (\sum j.\ f\ (j + i)) + (\sum j{<}i.\ f\ j)$
by (*intro sums-unique*[*symmetric*] *sums-offset summable-sums*)

lemma *eventually-at-left-1*: $(\bigwedge z{::}real.\ 0 < z \Longrightarrow z < 1 \Longrightarrow P\ z) \Longrightarrow$ *eventually P* (*at-left 1*)
by (*subst eventually-at-left*[*of 0*]) (*auto intro*: *exI*[*of - 0*])

lemma *mult-eq-1*:
fixes *a b* :: *'a* :: {*ordered-semiring*, *comm-monoid-mult*}
shows $0 \le a \Longrightarrow a \le 1 \Longrightarrow b \le 1 \Longrightarrow a * b = 1 \longleftrightarrow (a = 1 \wedge b = 1)$
by (*metis mult.left-neutral eq-iff mult.commute mult-right-mono*)

lemma *ereal-add-diff-cancel*:
fixes *a b* :: *ereal*
shows $|b| \neq \infty \Longrightarrow (a + b) - b = a$
by (*cases a b rule*: *ereal2-cases*) *auto*

lemma *add-top*:
fixes *x* :: *'a*::{*order-top*, *ordered-comm-monoid-add*}
shows $0 \le x \Longrightarrow x + top = top$
by (*intro top-le add-increasing order-refl*)

lemma *top-add*:
fixes *x* :: *'a*::{*order-top*, *ordered-comm-monoid-add*}
shows $0 \le x \Longrightarrow top + x = top$
by (*intro top-le add-increasing2 order-refl*)

lemma *le-lfp*: *mono f* $\Longrightarrow x \le$ *lfp f* $\Longrightarrow$ *f x* $\le$ *lfp f*
by (*subst lfp-unfold*) (*auto dest*: *monoD*)

lemma *lfp-transfer*:
assumes α: *sup-continuous* α **and** *f*: *sup-continuous f* **and** *mg*: *mono g*
assumes *bot*: α *bot* $\le$ *lfp g* **and** *eq*: $\bigwedge x.\ x \le$ *lfp f* $\Longrightarrow \alpha\ (f\ x) = g\ (\alpha\ x)$
shows α (*lfp f*) = *lfp g*
proof (*rule antisym*)

note *mf* = *sup-continuous-mono*[*OF f*]
have *f-le-lfp*: (*f* ^^ *i*) *bot* ≤ *lfp f* **for** *i*
by (*induction i*) (*auto intro*: *le-lfp mf*)

have *α* ((*f* ^^ *i*) *bot*) ≤ *lfp g* **for** *i*
by (*induction i*) (*auto simp*: *bot eq f-le-lfp intro*!: *le-lfp mg*)
then show *α* (*lfp f*) ≤ *lfp g*
unfolding *sup-continuous-lfp*[*OF f*]
by (*simp add*: *SUP-least α*[*THEN sup-continuousD*] *mf mono-funpow*)
show *lfp g* ≤ *α* (*lfp f*)
by (*rule lfp-lowerbound*) (*simp add*: *eq*[*symmetric*] *lfp-fixpoint*[*OF mf*])
qed

lemma *sup-continuous-applyD*: *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *f x h*)
using *sup-continuous-apply*[*THEN sup-continuous-compose*] **.**

lemma *sup-continuous-SUP*[*order-continuous-intros*]:
fixes *M* :: - ⇒ - ⇒ *'a*::*complete-lattice*
assumes *M*: ⋀*i*. *i* ∈ *I* ⟹ *sup-continuous* (*M i*)
shows *sup-continuous* (*SUP i*∈*I*. *M i*)
unfolding *sup-continuous-def* **by** (*auto simp add*: *sup-continuousD* [*OF M*] *image-comp intro*: *SUP-commute*)

lemma *sup-continuous-apply-SUP*[*order-continuous-intros*]:
fixes *M* :: - ⇒ - ⇒ *'a*::*complete-lattice*
shows (⋀*i*. *i* ∈ *I* ⟹ *sup-continuous* (*M i*)) ⟹ *sup-continuous* (λ*x*. *SUP i*∈*I*. *M i x*)
unfolding *SUP-apply*[*symmetric*] **by** (*rule sup-continuous-SUP*)

lemma *sup-continuous-lfp'*[*order-continuous-intros*]:
assumes *1*: *sup-continuous f*
assumes *2*: ⋀*g*. *sup-continuous g* ⟹ *sup-continuous* (*f g*)
shows *sup-continuous* (*lfp f*)
proof −
have *sup-continuous* ((*f* ^^ *i*) *bot*) **for** *i*
proof (*induction i*)
case (*Suc i*) **then show** *?case*
by (*auto intro*!: *2*)
qed (*simp add*: *bot-fun-def sup-continuous-const*)
then show *?thesis*
unfolding *sup-continuous-lfp*[*OF 1*] **by** (*intro order-continuous-intros*)
qed

lemma *sup-continuous-lfp''*[*order-continuous-intros*]:
assumes *1*: ⋀*s*. *sup-continuous* (*f s*)
assumes *2*: ⋀*g*. *sup-continuous g* ⟹ *sup-continuous* (λ*s*. *f s* (*g s*))
shows *sup-continuous* (λ*x*. *lfp* (*f x*))
proof −
have *sup-continuous* (λ*x*. (*f x* ^^ *i*) *bot*) **for** *i*

proof (*induction i*)
case (*Suc i*) **then show** *?case*
by (*auto intro*!: *2*)
qed (*simp add*: *bot-fun-def sup-continuous-const*)
then show *?thesis*
unfolding *sup-continuous-lfp*[*OF 1*] **by** (*intro order-continuous-intros*)
qed

lemma *mono-INF-fun*:
(⋀*x y*. *mono* (*F x y*)) ⟹ *mono* (λ*z x*. *INF y* ∈ *X x*. *F x y z* :: *'a* :: *complete-lattice*)
by (*auto intro*!: *INF-mono*[*OF bexI*] *simp*: *le-fun-def mono-def*)

lemma *continuous-on-cmult-ereal*:
|*c*::*ereal*| ≠ ∞ ⟹ *continuous-on A f* ⟹ *continuous-on A* (λ*x*. *c* ∗ *f x*)
using *tendsto-cmult-ereal*[*of c f f x at x within A* **for** *x*]
by (*auto simp*: *continuous-on-def simp del*: *tendsto-cmult-ereal*)

lemma *real-of-nat-Sup*:
assumes *A* ≠ {} *bdd-above A*
shows *of-nat* (*Sup A*) = (*SUP a*∈*A*. *of-nat a* :: *real*)
proof (*intro antisym*)
show (*SUP a*∈*A*. *of-nat a*::*real*) ≤ *of-nat* (*Sup A*)
using *assms* **by** (*intro cSUP-least of-nat-mono*) (*auto intro*: *cSup-upper*)
have *Sup A* ∈ *A*
using *assms* **by** (*auto simp*: *Sup-nat-def bdd-above-nat*)
then show *of-nat* (*Sup A*) ≤ (*SUP a*∈*A*. *of-nat a*::*real*)
by (*intro cSUP-upper bdd-above-image-mono assms*) (*auto simp*: *mono-def*)
qed

lemma (**in** *complete-lattice*) *SUP-sup-const1*:
I ≠ {} ⟹ (*SUP i*∈*I*. *sup c* (*f i*)) = *sup c* (*SUP i*∈*I*. *f i*)
using *SUP-sup-distrib*[*of* λ-. *c I f*] **by** *simp*

lemma (**in** *complete-lattice*) *SUP-sup-const2*:
I ≠ {} ⟹ (*SUP i*∈*I*. *sup* (*f i*) *c*) = *sup* (*SUP i*∈*I*. *f i*) *c*
using *SUP-sup-distrib*[*of f I* λ-. *c*] **by** *simp*

lemma *one-less-of-natD*:
assumes (*1*::*'a*::*linordered-semidom*) < *of-nat n* **shows** *1* < *n*
by (*cases n*) (*use assms* **in** *auto*)

40.1 Defining the extended non-negative reals

Basic definitions and type class setup

typedef *ennreal* = {*x* :: *ereal*. *0* ≤ *x*}
morphisms *enn2ereal e2ennreal'*
by *auto*

definition *e2ennreal x = e2ennreal′ (max 0 x)*

lemma *enn2ereal-range*: *e2ennreal ‘ {0..} = UNIV*
proof −
  have ∃ *y*≥*0*. *x = e2ennreal y* **for** *x*
    by (*cases x*) (*auto simp*: *e2ennreal-def max-absorb2*)
  then show *?thesis*
    by (*auto simp*: *image-iff Bex-def*)
qed

lemma *type-definition-ennreal′*: *type-definition enn2ereal e2ennreal {x. 0 ≤ x}*
  using *type-definition-ennreal*
  by (*auto simp*: *type-definition-def e2ennreal-def max-absorb2*)

setup-lifting *type-definition-ennreal′*

declare [[*coercion e2ennreal*]]

instantiation *ennreal* :: *complete-linorder*
begin

lift-definition *top-ennreal* :: *ennreal* **is** *top* **by** (*rule top-greatest*)
lift-definition *bot-ennreal* :: *ennreal* **is** *0* **by** (*rule order-refl*)
lift-definition *sup-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *sup* **by** (*rule le-supI1*)
lift-definition *inf-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** *inf* **by** (*rule le-infI*)

lift-definition *Inf-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *Inf*
  by (*rule Inf-greatest*)

lift-definition *Sup-ennreal* :: *ennreal set* ⇒ *ennreal* **is** *sup 0* ∘ *Sup*
  by *auto*

lift-definition *less-eq-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** (≤) **.**
lift-definition *less-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *bool* **is** (<) **.**

instance
  by *standard*
    (*transfer*; *auto simp*: *Inf-lower Inf-greatest Sup-upper Sup-least le-max-iff-disj max.absorb1*)+

end

lemma *pcr-ennreal-enn2ereal*[*simp*]: *pcr-ennreal* (*enn2ereal x*) *x*
  by (*simp add*: *ennreal.pcr-cr-eq cr-ennreal-def*)

lemma *rel-fun-eq-pcr-ennreal*: *rel-fun* (=) *pcr-ennreal f g* ⟷ *f = enn2ereal* ∘ *g*
  by (*auto simp*: *rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def*)

instantiation *ennreal* :: *infinity*

begin

definition *infinity-ennreal* :: *ennreal*
  where [*simp*]: ∞ = (*top*::*ennreal*)

instance ..

end

instantiation *ennreal* :: {*semiring-1-no-zero-divisors*, *comm-semiring-1*}
begin

lift-definition *one-ennreal* :: *ennreal* **is** *1* **by** *simp*
lift-definition *zero-ennreal* :: *ennreal* **is** *0* **by** *simp*
lift-definition *plus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** (+) **by** *simp*
lift-definition *times-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** (∗) **by** *simp*

instance
  by *standard* (*transfer*; *auto simp*: *field-simps ereal-right-distrib*)+

end

instantiation *ennreal* :: *minus*
begin

lift-definition *minus-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal* **is** λ*a b*. *max 0* (*a* − *b*)
  by *simp*

instance ..

end

instance *ennreal* :: *numeral* **..**

instantiation *ennreal* :: *inverse*
begin

lift-definition *inverse-ennreal* :: *ennreal* ⇒ *ennreal* **is** *inverse*
  by (*rule inverse-ereal-ge0I*)

definition *divide-ennreal* :: *ennreal* ⇒ *ennreal* ⇒ *ennreal*
  where *x div y* = *x* ∗ *inverse* (*y* :: *ennreal*)

instance ..

end

lemma *ennreal-zero-less-one*: *0* < (*1*::*ennreal*) — TODO: remove

by *transfer auto*

instance *ennreal* :: *dioid*
proof (*standard*; *transfer*)
fix *a b* :: *ereal* **assume** *0* ≤ *a 0* ≤ *b* **then show** (*a* ≤ *b*) = (∃ *c*∈*Collect* ((≤) *0*). *b* = *a* + *c*)
unfolding *ereal-ex-split Bex-def*
by (*cases a b rule*: *ereal2-cases*) (*auto intro*!: *exI*[*of* - *real-of-ereal* (*b* − *a*)])
qed

instance *ennreal* :: *ordered-comm-semiring*
by *standard*
(*transfer*; *auto intro*: *add-mono mult-mono mult-ac ereal-left-distrib ereal-mult-left-mono*)+

instance *ennreal* :: *linordered-nonzero-semiring*
proof
fix *a b*::*ennreal*
show *a* < *b* ⟹ *a* + *1* < *b* + *1*
by *transfer* (*simp add*: *add-right-mono ereal-add-cancel-right less-le*)
qed (*transfer*; *simp*)

instance *ennreal* :: *strict-ordered-ab-semigroup-add*
proof
fix *a b c d* :: *ennreal* **show** *a* < *b* ⟹ *c* < *d* ⟹ *a* + *c* < *b* + *d*
by *transfer* (*auto intro*!: *ereal-add-strict-mono*)
qed

declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *e2ennreal-neg*: *x* ≤ *0* ⟹ *e2ennreal x* = *0*
unfolding *zero-ennreal-def e2ennreal-def* **by** (*simp add*: *max-absorb1*)

lemma *e2ennreal-mono*: *x* ≤ *y* ⟹ *e2ennreal x* ≤ *e2ennreal y*
by (*cases 0* ≤ *x 0* ≤ *y rule*: *bool.exhaust*[*case-product bool.exhaust*])
(*auto simp*: *e2ennreal-neg less-eq-ennreal.abs-eq eq-onp-def*)

lemma *enn2ereal-nonneg*[*simp*]: *0* ≤ *enn2ereal x*
using *ennreal.enn2ereal*[*of x*] **by** *simp*

lemma *ereal-ennreal-cases*:
obtains *b* **where** *0* ≤ *a a* = *enn2ereal b* | *a* < *0*
using *e2ennreal'-inverse*[*of a*, *symmetric*] **by** (*cases 0* ≤ *a*) (*auto intro*: *enn2ereal-nonneg*)

lemma *rel-fun-liminf*[*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal liminf liminf*
proof −
have ∀ *x y*. *rel-fun* (=) *pcr-ennreal x y* ⟶ *pcr-ennreal* (*sup 0* (*liminf x*)) (*liminf y*) ⟹
∀ *x y*. *rel-fun* (=) *pcr-ennreal x y* ⟶ *pcr-ennreal* (*liminf x*) (*liminf y*)

by (*auto simp*: *comp-def Liminf-bounded rel-fun-eq-pcr-ennreal*)
moreover have *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* (λ*x*. *sup 0* (*liminf x*)) *liminf*
unfolding *liminf-SUP-INF*[*abs-def*] **by** (*transfer-prover-start*, *transfer-step*+; *simp*)
ultimately show *?thesis*
by (*simp add*: *rel-fun-def*)
qed

lemma *rel-fun-limsup*[*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal limsup limsup*
proof −
have [*simp*]: *max 0* (*SUP x*∈{*n*..}. *enn2ereal* (*y x*)) = (*SUP x*∈{*n*..}. *enn2ereal* (*y x*)) **for** *n*::*nat* **and** *y*
by (*meson SUP-upper atLeast-iff enn2ereal-nonneg max.absorb2 nle-le order-trans*)
have *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal* (λ*x*. *INF n*. *sup 0* (*SUP i*∈{*n*..}. *x i*)) *limsup*
unfolding *limsup-INF-SUP*[*abs-def*] **by** (*transfer-prover-start*, *transfer-step*+; *simp*)
moreover
have ⋀*x y*. ⟦*rel-fun* (=) *pcr-ennreal x y*;
pcr-ennreal (*INF n*::*nat*. *max 0* (*Sup* (*x* ‘ {*n*..}))) (*INF n*. *Sup* (*y* ‘ {*n*..}))⟧
⟹ *pcr-ennreal* (*INF n*. *Sup* (*x* ‘ {*n*..})) (*INF n*. *Sup* (*y* ‘ {*n*..}))
by (*auto simp*: *comp-def rel-fun-eq-pcr-ennreal*)
ultimately show *?thesis*
by (*simp add*: *limsup-INF-SUP rel-fun-def*)
qed

lemma *sum-enn2ereal*[*simp*]: (⋀*i*. *i* ∈ *I* ⟹ 0 ≤ *f i*) ⟹ (∑ *i*∈*I*. *enn2ereal* (*f i*)) = *enn2ereal* (*sum f I*)
by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *sum-nonneg zero-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *transfer-e2ennreal-sum* [*transfer-rule*]:
rel-fun (*rel-fun* (=) *pcr-ennreal*) (*rel-fun* (=) *pcr-ennreal*) *sum sum*
by (*auto intro*!: *rel-funI simp*: *rel-fun-eq-pcr-ennreal comp-def*)

lemma *enn2ereal-of-nat*[*simp*]: *enn2ereal* (*of-nat n*) = *ereal n*
by (*induction n*) (*auto simp*: *zero-ennreal.rep-eq one-ennreal.rep-eq plus-ennreal.rep-eq*)

lemma *enn2ereal-numeral*[*simp*]: *enn2ereal* (*numeral a*) = *numeral a*
by (*metis enn2ereal-of-nat numeral-eq-ereal of-nat-numeral*)

lemma *transfer-numeral*[*transfer-rule*]: *pcr-ennreal* (*numeral a*) (*numeral a*)
by (*metis enn2ereal-numeral pcr-ennreal-enn2ereal*)

40.2 Cancellation simprocs

lemma *ennreal-add-left-cancel*: $a + b = a + c \longleftrightarrow a = (\infty::ennreal) \vee b = c$
unfolding *infinity-ennreal-def* **by** *transfer* (*simp add*: *top-ereal-def ereal-add-cancel-left*)

lemma *ennreal-add-left-cancel-le*: $a + b \leq a + c \longleftrightarrow a = (\infty::ennreal) \vee b \leq c$
unfolding *infinity-ennreal-def* **by** *transfer* (*simp add*: *ereal-add-le-add-iff top-ereal-def disj-commute*)

lemma *ereal-add-left-cancel-less*:
fixes *a b c* :: *ereal*
shows $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow a + b < a + c \longleftrightarrow a \neq \infty \wedge b < c$
by (*cases a b c rule*: *ereal3-cases*) *auto*

lemma *ennreal-add-left-cancel-less*: $a + b < a + c \longleftrightarrow a \neq (\infty::ennreal) \wedge b < c$
unfolding *infinity-ennreal-def*
by *transfer* (*simp add*: *top-ereal-def ereal-add-left-cancel-less*)

ML ‹
structure Cancel-Ennreal-Common =
struct
(∗ *copied from src/HOL/Tools/nat-numeral-simprocs.ML* ∗)
fun find-first-t - - [] = raise TERM(*find-first-t*, [])
| *find-first-t past u* (*t*::*terms*) =
if u aconv t then (*rev past* @ *terms*)
else find-first-t (*t*::*past*) *u terms*

fun dest-summing (*Const* (***const-name*** ‹*Groups.plus*›, -) $ *t* $ *u*, *ts*) =
dest-summing (*t*, *dest-summing* (*u*, *ts*))
| *dest-summing* (*t*, *ts*) = *t* :: *ts*

val mk-sum = Arith-Data.long-mk-sum
fun dest-sum t = dest-summing (*t*, [])
val find-first = find-first-t []
val trans-tac = Numeral-Simprocs.trans-tac
val norm-ss =
HOL-basic-ss |> *Simplifier.simpset-map* ***context***
(*Simplifier.add-simps* @{*thms ac-simps add-0-left add-0-right*})
fun norm-tac ctxt = ALLGOALS (*simp-tac* (*put-simpset norm-ss ctxt*))
fun simplify-meta-eq ctxt cancel-th th =
Arith-Data.simplify-meta-eq [] *ctxt*
([*th*, *cancel-th*] *MRS trans*)
fun mk-eq (*a*, *b*) = *HOLogic.mk-Trueprop* (*HOLogic.mk-eq* (*a*, *b*))
end

structure Eq-Ennreal-Cancel = ExtractCommonTermFun
(*open Cancel-Ennreal-Common*
val mk-bal = HOLogic.mk-eq
val dest-bal = HOLogic.dest-bin ***const-name*** ‹*HOL.eq*› ***typ*** ‹*ennreal*›
fun simp-conv - - = SOME @{*thm ennreal-add-left-cancel*}

```
)

structure Le-Ennreal-Cancel = ExtractCommonTermFun
(open Cancel-Ennreal-Common
  val mk-bal = HOLogic.mk-binrel const-name ‹Orderings.less-eq›
  val dest-bal = HOLogic.dest-bin const-name ‹Orderings.less-eq› typ ‹ennreal›
  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel-le}
)

structure Less-Ennreal-Cancel = ExtractCommonTermFun
(open Cancel-Ennreal-Common
  val mk-bal = HOLogic.mk-binrel const-name ‹Orderings.less›
  val dest-bal = HOLogic.dest-bin const-name ‹Orderings.less› typ ‹ennreal›
  fun simp-conv - - = SOME @{thm ennreal-add-left-cancel-less}
)
›

simproc-setup ennreal-eq-cancel
  ((l::ennreal) + m = n | (l::ennreal) = m + n) =
  ‹K (fn ctxt => fn ct => Eq-Ennreal-Cancel.proc ctxt (Thm.term-of ct))›

simproc-setup ennreal-le-cancel
  ((l::ennreal) + m ≤ n | (l::ennreal) ≤ m + n) =
  ‹K (fn ctxt => fn ct => Le-Ennreal-Cancel.proc ctxt (Thm.term-of ct))›

simproc-setup ennreal-less-cancel
  ((l::ennreal) + m < n | (l::ennreal) < m + n) =
  ‹K (fn ctxt => fn ct => Less-Ennreal-Cancel.proc ctxt (Thm.term-of ct))›
```

40.3 Order with top

lemma *ennreal-zero-less-top*[*simp*]: *0 < (top::ennreal)*
  by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-one-less-top*[*simp*]: *1 < (top::ennreal)*
  by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-zero-neq-top*[*simp*]: *0 ≠ (top::ennreal)*
  by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-top-neq-zero*[*simp*]: *(top::ennreal) ≠ 0*
  by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-top-neq-one*[*simp*]: *top ≠ (1::ennreal)*
  by *transfer* (*simp add*: *top-ereal-def one-ereal-def flip*: *ereal-max*)

lemma *ennreal-one-neq-top*[*simp*]: *1 ≠ (top::ennreal)*
  by *transfer* (*simp add*: *top-ereal-def one-ereal-def flip*: *ereal-max*)

lemma *ennreal-add-less-top*[*simp*]:
fixes *a b* :: *ennreal*
shows $a + b < top \longleftrightarrow a < top \wedge b < top$
by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-add-eq-top*[*simp*]:
fixes *a b* :: *ennreal*
shows $a + b = top \longleftrightarrow a = top \vee b = top$
by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-sum-less-top*[*simp*]:
fixes *f* :: *'a* ⇒ *ennreal*
shows *finite I* ⟹ $(\sum i{\in}I.\ f\ i) < top \longleftrightarrow (\forall\, i{\in}I.\ f\ i < top)$
by (*induction I rule*: *finite-induct*) *auto*

lemma *ennreal-sum-eq-top*[*simp*]:
fixes *f* :: *'a* ⇒ *ennreal*
shows *finite I* ⟹ $(\sum i{\in}I.\ f\ i) = top \longleftrightarrow (\exists\, i{\in}I.\ f\ i = top)$
by (*induction I rule*: *finite-induct*) *auto*

lemma *ennreal-mult-eq-top-iff*:
fixes *a b* :: *ennreal*
shows $a * b = top \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
by *transfer* (*auto simp*: *top-ereal-def*)

lemma *ennreal-top-eq-mult-iff*:
fixes *a b* :: *ennreal*
shows $top = a * b \longleftrightarrow (a = top \wedge b \neq 0) \vee (b = top \wedge a \neq 0)$
using *ennreal-mult-eq-top-iff*[*of a b*] **by** *auto*

lemma *ennreal-mult-less-top*:
fixes *a b* :: *ennreal*
shows $a * b < top \longleftrightarrow (a = 0 \vee b = 0 \vee (a < top \wedge b < top))$
by *transfer* (*auto simp add*: *top-ereal-def*)

lemma *top-power-ennreal*: $top\ \hat{}\ n = (if\ n = 0\ then\ 1\ else\ top :: ennreal)$
by (*induction n*) (*simp-all add*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-prod-eq-0*[*simp*]:
fixes *f* :: *'a* ⇒ *ennreal*
shows $(prod\ f\ A = 0) = (finite\ A \wedge (\exists\, i{\in}A.\ f\ i = 0))$
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *ennreal-prod-eq-top*:
fixes *f* :: *'a* ⇒ *ennreal*
shows $(\prod i{\in}I.\ f\ i) = top \longleftrightarrow (finite\ I \wedge ((\forall\, i{\in}I.\ f\ i \neq 0) \wedge (\exists\, i{\in}I.\ f\ i = top)))$
by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-top-mult*: $top * a = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$

by (*simp add*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-mult-top*: $a * top = (if\ a = 0\ then\ 0\ else\ top :: ennreal)$
by (*simp add*: *ennreal-mult-eq-top-iff*)

lemma *enn2ereal-eq-top-iff*[*simp*]: $enn2ereal\ x = \infty \longleftrightarrow x = top$
by *transfer* (*simp add*: *top-ereal-def*)

lemma *enn2ereal-top*[*simp*]: $enn2ereal\ top = \infty$
by *transfer* (*simp add*: *top-ereal-def*)

lemma *e2ennreal-infty*[*simp*]: $e2ennreal\ \infty = top$
by (*simp add*: *top-ennreal.abs-eq top-ereal-def*)

lemma *ennreal-top-minus*[*simp*]: $top - x = (top::ennreal)$
by *transfer* (*auto simp*: *top-ereal-def max-def*)

lemma *minus-top-ennreal*: $x - top = (if\ x = top\ then\ top\ else\ 0:: ennreal)$
by *transfer* (*use ereal-eq-minus-iff top-ereal-def* **in** *force*)

lemma *bot-ennreal*: $bot = (0::ennreal)$
by *transfer rule*

lemma *ennreal-of-nat-neq-top*[*simp*]: $of\text{-}nat\ i \neq (top::ennreal)$
by (*induction i*) *auto*

lemma *numeral-eq-of-nat*: $(numeral\ a::ennreal) = of\text{-}nat\ (numeral\ a)$
by *simp*

lemma *of-nat-less-top*: $of\text{-}nat\ i < (top::ennreal)$
using *less-le-trans*[*of of-nat i of-nat* (*Suc i*) *top*::*ennreal*]
by *simp*

lemma *top-neq-numeral*[*simp*]: $top \neq (numeral\ i::ennreal)$
using *of-nat-less-top*[*of numeral i*] **by** *simp*

lemma *ennreal-numeral-less-top*[*simp*]: $numeral\ i < (top::ennreal)$
using *of-nat-less-top*[*of numeral i*] **by** *simp*

lemma *ennreal-add-bot*[*simp*]: $bot + x = (x::ennreal)$
by *transfer simp*

lemma *add-top-right-ennreal* [*simp*]: $x + top = (top :: ennreal)$
by (*cases x*) *auto*

lemma *add-top-left-ennreal* [*simp*]: $top + x = (top :: ennreal)$
by (*cases x*) *auto*

lemma *ennreal-top-mult-left* [*simp*]: $x \neq 0 \Longrightarrow x * top = (top :: ennreal)$

by (*subst ennreal-mult-eq-top-iff*) *auto*

lemma *ennreal-top-mult-right* [*simp*]: $x \neq 0 \Longrightarrow top * x = (top :: ennreal)$
by (*subst ennreal-mult-eq-top-iff*) *auto*

lemma *power-top-ennreal* [*simp*]: $n > 0 \Longrightarrow top \mathbin{\hat{}} n = (top :: ennreal)$
by (*induction n*) *auto*

lemma *power-eq-top-ennreal-iff*: $x \mathbin{\hat{}} n = top \longleftrightarrow x = (top :: ennreal) \wedge n > 0$
by (*induction n*) (*auto simp*: *ennreal-mult-eq-top-iff*)

lemma *ennreal-mult-le-mult-iff*: $c \neq 0 \Longrightarrow c \neq top \Longrightarrow c * a \leq c * b \longleftrightarrow a \leq (b :: ennreal)$
including *ennreal.lifting*
by (*transfer*, *subst ereal-mult-le-mult-iff*) (*auto simp*: *top-ereal-def*)

lemma *power-mono-ennreal*: $x \leq y \Longrightarrow x \mathbin{\hat{}} n \leq (y \mathbin{\hat{}} n :: ennreal)$
by (*induction n*) (*auto intro*!: *mult-mono*)

instance *ennreal* :: *semiring-char-0*
proof (*standard*, *safe intro*!: *linorder-injI*)
have $*$: $1 + of\text{-}nat\ k \neq (0::ennreal)$ **for** k
using *add-pos-nonneg*[*OF zero-less-one*, *of of-nat k* :: *ennreal*] **by** *auto*
fix $x\ y$ :: *nat* **assume** $x < y$ *of-nat* $x = (of\text{-}nat\ y::ennreal)$ **then show** *False*
by (*auto simp add*: *less-iff-Suc-add* $*$)
qed

40.4 Arithmetic

lemma *ennreal-minus-zero*[*simp*]: $a - (0::ennreal) = a$
by *transfer* (*auto simp*: *max-def*)

lemma *ennreal-add-diff-cancel-right*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (x + y) - y = x$
by *transfer* (*metis ereal-eq-minus-iff max-absorb2 not-MInfty-nonneg top-ereal-def*)

lemma *ennreal-add-diff-cancel-left*[*simp*]:
fixes $x\ y\ z$ :: *ennreal* **shows** $y \neq top \Longrightarrow (y + x) - y = x$
by (*simp add*: *add.commute*)

lemma
fixes $a\ b$ :: *ennreal*
shows $a - b = 0 \Longrightarrow a \leq b$
by *transfer* (*metis ereal-diff-gr0 le-cases max.absorb2 not-less*)

lemma *ennreal-minus-cancel*:
fixes $a\ b\ c$ :: *ennreal*
shows $c \neq top \Longrightarrow a \leq c \Longrightarrow b \leq c \Longrightarrow c - a = c - b \Longrightarrow a = b$

by (*metis ennreal-add-diff-cancel-left ennreal-add-diff-cancel-right ennreal-add-eq-top less-eqE*)

lemma *sup-const-add-ennreal*:
fixes *a b c* :: *ennreal*
shows *sup* (*c* + *a*) (*c* + *b*) = *c* + *sup a b*
by *transfer* (*metis add-left-mono le-cases sup.absorb2 sup.orderE*)

lemma *ennreal-diff-add-assoc*:
fixes *a b c* :: *ennreal*
shows *a* ≤ *b* ⟹ *c* + *b* − *a* = *c* + (*b* − *a*)
by (*metis add.left-commute ennreal-add-diff-cancel-left ennreal-add-eq-top ennreal-top-minus less-eqE*)

lemma *mult-divide-eq-ennreal*:
fixes *a b* :: *ennreal*
shows *b* ≠ *0* ⟹ *b* ≠ *top* ⟹ (*a* ∗ *b*) / *b* = *a*
unfolding *divide-ennreal-def*
apply *transfer*
by (*metis abs-ereal-ge0 divide-ereal-def ereal-divide-eq ereal-times-divide-eq top-ereal-def*)

lemma *divide-mult-eq*: *a* ≠ *0* ⟹ *a* ≠ ∞ ⟹ *x* ∗ *a* / (*b* ∗ *a*) = *x* / (*b*::*ennreal*)
unfolding *divide-ennreal-def infinity-ennreal-def*
apply *transfer*
subgoal for *a b c*
by (*cases a b c rule*: *ereal3-cases*) (*auto simp*: *top-ereal-def*)
done

lemma *ennreal-mult-divide-eq*:
fixes *a b* :: *ennreal*
shows *b* ≠ *0* ⟹ *b* ≠ *top* ⟹ (*a* ∗ *b*) / *b* = *a*
by (*fact mult-divide-eq-ennreal*)

lemma *ennreal-add-diff-cancel*:
fixes *a b* :: *ennreal*
shows *b* ≠ ∞ ⟹ (*a* + *b*) − *b* = *a*
by *simp*

lemma *ennreal-minus-eq-0*:
a − *b* = *0* ⟹ *a* ≤ (*b*::*ennreal*)
by *transfer* (*metis ereal-diff-gr0 le-cases max.absorb2 not-less*)

lemma *ennreal-mono-minus-cancel*:
fixes *a b c* :: *ennreal*
shows *a* − *b* ≤ *a* − *c* ⟹ *a* < *top* ⟹ *b* ≤ *a* ⟹ *c* ≤ *a* ⟹ *c* ≤ *b*
by *transfer*
(*auto simp add*: *ereal-diff-positive top-ereal-def dest*: *ereal-mono-minus-cancel*)

lemma *ennreal-mono-minus*:

fixes *a b c* :: *ennreal*
shows $c \leq b \Longrightarrow a - b \leq a - c$
by *transfer* (*meson ereal-minus-mono max.mono order-refl*)

lemma *ennreal-minus-pos-iff*:
fixes *a b* :: *ennreal*
shows $a < top \vee b < top \Longrightarrow 0 < a - b \Longrightarrow b < a$
by *transfer* (*use add.left-neutral ereal-minus-le-iff less-irrefl not-less* **in** *fastforce*)

lemma *ennreal-inverse-top*[*simp*]: *inverse top* = (*0*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-inverse-zero*[*simp*]: *inverse 0* = (*top*::*ennreal*)
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0*)

lemma *ennreal-top-divide*: *top* / (*x*::*ennreal*) = (*if x* = *top then 0 else top*)
unfolding *divide-ennreal-def*
by *transfer* (*simp add*: *top-ereal-def ereal-inverse-eq-0 ereal-0-gt-inverse*)

lemma *ennreal-zero-divide*[*simp*]: *0* / (*x*::*ennreal*) = *0*
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-divide-zero*[*simp*]: *x* / (*0*::*ennreal*) = (*if x* = *0 then 0 else top*)
by (*simp add*: *divide-ennreal-def ennreal-mult-top*)

lemma *ennreal-divide-top*[*simp*]: *x* / (*top*::*ennreal*) = *0*
by (*simp add*: *divide-ennreal-def ennreal-top-mult*)

lemma *ennreal-times-divide*: *a* ∗ (*b* / *c*) = *a* ∗ *b* / (*c*::*ennreal*)
unfolding *divide-ennreal-def*
by *transfer* (*simp add*: *divide-ereal-def*[*symmetric*] *ereal-times-divide-eq*)

lemma *ennreal-zero-less-divide*: *0* < *a* / *b* ⟷ (*0* < *a* ∧ *b* < (*top*::*ennreal*))
unfolding *divide-ennreal-def*
by *transfer* (*auto simp*: *ereal-zero-less-0-iff top-ereal-def ereal-0-gt-inverse*)

lemma *add-divide-distrib-ennreal*: (*a* + *b*) / *c* = *a* / *c* + *b* / (*c* :: *ennreal*)
by (*simp add*: *divide-ennreal-def ring-distribs*)

lemma *divide-right-mono-ennreal*:
fixes *a b c* :: *ennreal*
shows *a* ≤ *b* ⟹ *a* / *c* ≤ *b* / *c*
unfolding *divide-ennreal-def* **by** (*intro mult-mono*) *auto*

lemma *ennreal-mult-strict-right-mono*: (*a*::*ennreal*) < *c* ⟹ *0* < *b* ⟹ *b* < *top*
⟹ *a* ∗ *b* < *c* ∗ *b*
by *transfer* (*auto intro*!: *ereal-mult-strict-right-mono*)

lemma *ennreal-indicator-less*[*simp*]:

indicator A x $\leq$ (*indicator B x*::*ennreal*) $\longleftrightarrow$ ($x \in A \longrightarrow x \in B$)
by (*simp add*: *indicator-def not-le*)

lemma *ennreal-inverse-positive*: *0* $<$ *inverse x* $\longleftrightarrow$ (*x*::*ennreal*) $\neq$ *top*
by *transfer* (*simp add*: *ereal-0-gt-inverse top-ereal-def*)

lemma *ennreal-inverse-mult'*: (($0 < b \vee a < top$) $\wedge$ ($0 < a \vee b < top$)) $\Longrightarrow$ *inverse* (*a* $*$ *b*::*ennreal*) = *inverse a* $*$ *inverse b*
apply *transfer*
subgoal for *a b*
by (*cases a b rule*: *ereal2-cases*) (*auto simp*: *top-ereal-def*)
done

lemma *ennreal-inverse-mult*: *a* $<$ *top* $\Longrightarrow$ *b* $<$ *top* $\Longrightarrow$ *inverse* (*a* $*$ *b*::*ennreal*) = *inverse a* $*$ *inverse b*
by (*simp add*: *ennreal-inverse-mult'*)

lemma *ennreal-inverse-1*[*simp*]: *inverse* (*1*::*ennreal*) = *1*
by *transfer simp*

lemma *ennreal-inverse-eq-0-iff*[*simp*]: *inverse* (*a*::*ennreal*) = *0* $\longleftrightarrow$ *a* = *top*
by (*metis ennreal-inverse-positive not-gr-zero*)

lemma *ennreal-inverse-eq-top-iff*[*simp*]: *inverse* (*a*::*ennreal*) = *top* $\longleftrightarrow$ *a* = *0*
by *transfer* (*simp add*: *top-ereal-def*)

lemma *ennreal-divide-eq-0-iff*[*simp*]: (*a*::*ennreal*) / *b* = *0* $\longleftrightarrow$ ($a = 0 \vee b = top$)
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-divide-eq-top-iff*: (*a*::*ennreal*) / *b* = *top* $\longleftrightarrow$ (($a \neq 0 \wedge b = 0$) $\vee$ ($a = top \wedge b \neq top$))
by (*auto simp add*: *divide-ennreal-def ennreal-mult-eq-top-iff*)

lemma *one-divide-one-divide-ennreal*[*simp*]: *1* / (*1* / *c*) = (*c*::*ennreal*)
including *ennreal.lifting*
unfolding *divide-ennreal-def*
by *transfer auto*

lemma *ennreal-mult-left-cong*:
((*a*::*ennreal*) $\neq$ *0* $\Longrightarrow$ *b* = *c*) $\Longrightarrow$ *a* $*$ *b* = *a* $*$ *c*
by (*cases a* = *0*) *simp-all*

lemma *ennreal-mult-right-cong*:
((*a*::*ennreal*) $\neq$ *0* $\Longrightarrow$ *b* = *c*) $\Longrightarrow$ *b* $*$ *a* = *c* $*$ *a*
by (*cases a* = *0*) *simp-all*

lemma *ennreal-zero-less-mult-iff*: *0* $<$ *a* $*$ *b* $\longleftrightarrow$ *0* $<$ *a* $\wedge$ *0* $<$ (*b*::*ennreal*)
using *not-gr-zero* **by** *fastforce*

lemma *less-diff-eq-ennreal*:
fixes *a b c* :: *ennreal*
shows $b < top \vee c < top \Longrightarrow a < b - c \longleftrightarrow a + c < b$
apply *transfer*
subgoal for *a b c*
by (*cases a b c rule*: *ereal3-cases*) (*auto split*: *split-max*)
done

lemma *diff-add-cancel-ennreal*:
fixes *a b* :: *ennreal* **shows** $a \leq b \Longrightarrow b - a + a = b$
unfolding *infinity-ennreal-def*
by *transfer* (*metis* (*no-types*) *add.commute ereal-diff-positive ereal-ineq-diff-add max-def not-MInfty-nonneg*)

lemma *ennreal-diff-self*[*simp*]: $a \neq top \Longrightarrow a - a = (0::ennreal)$
by (*meson ennreal-minus-pos-iff less-imp-neq not-gr-zero top.not-eq-extremum*)

lemma *ennreal-minus-mono*:
fixes *a b c* :: *ennreal*
shows $a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq c - d$
by *transfer* (*meson ereal-minus-mono max.mono order-refl*)

lemma *ennreal-minus-eq-top*[*simp*]: $a - (b::ennreal) = top \longleftrightarrow a = top$
by (*metis add-top diff-add-cancel-ennreal ennreal-mono-minus ennreal-top-minus zero-le*)

lemma *ennreal-divide-self*[*simp*]: $a \neq 0 \Longrightarrow a < top \Longrightarrow a \mathbin{/} a = (1::ennreal)$
by (*metis mult-1 mult-divide-eq-ennreal top.not-eq-extremum*)

40.5 Coercion from *real* to *ennreal*

lift-definition *ennreal* :: *real* ⇒ *ennreal* **is** *sup 0* ∘ *ereal*
by *simp*

declare [[*coercion ennreal*]]

lemma *ennreal-cong*: $x = y \Longrightarrow ennreal\ x = ennreal\ y$
by *simp*

lemma *ennreal-cases*[*cases type*: *ennreal*]:
fixes *x* :: *ennreal*
obtains (*real*) *r* :: *real* **where** $0 \leq r\ x = ennreal\ r \mid (top)\ x = top$
apply *transfer*
subgoal for *x thesis*
by (*cases x*) (*auto simp*: *top-ereal-def*)
done

lemmas *ennreal2-cases* = *ennreal-cases*[*case-product ennreal-cases*]
lemmas *ennreal3-cases* = *ennreal-cases*[*case-product ennreal2-cases*]

lemma *ennreal-neq-top*[*simp*]: *ennreal r* ≠ *top*
by *transfer* (*simp add*: *top-ereal-def zero-ereal-def flip*: *ereal-max*)

lemma *top-neq-ennreal*[*simp*]: *top* ≠ *ennreal r*
using *ennreal-neq-top*[*of r*] **by** (*auto simp del*: *ennreal-neq-top*)

lemma *ennreal-less-top*[*simp*]: *ennreal x* < *top*
by *transfer* (*simp add*: *top-ereal-def max-def*)

lemma *ennreal-neg*: *x* ≤ *0* ⟹ *ennreal x* = *0*
by *transfer* (*simp add*: *max.absorb1*)

lemma *ennreal-inj*[*simp*]:
0 ≤ *a* ⟹ *0* ≤ *b* ⟹ *ennreal a* = *ennreal b* ⟷ *a* = *b*
by (*transfer fixing*: *a b*) (*auto simp*: *max-absorb2*)

lemma *ennreal-le-iff*[*simp*]: *0* ≤ *y* ⟹ *ennreal x* ≤ *ennreal y* ⟷ *x* ≤ *y*
by (*auto simp*: *ennreal-def zero-ereal-def less-eq-ennreal.abs-eq eq-onp-def split*: *split-max*)

lemma *le-ennreal-iff*: *0* ≤ *r* ⟹ *x* ≤ *ennreal r* ⟷ (∃ *q*≥*0*. *x* = *ennreal q* ∧ *q* ≤ *r*)
by (*cases x*) (*auto simp*: *top-unique*)

lemma *ennreal-less-iff*: *0* ≤ *r* ⟹ *ennreal r* < *ennreal q* ⟷ *r* < *q*
unfolding *not-le*[*symmetric*] **by** *auto*

lemma *ennreal-eq-zero-iff*[*simp*]: *0* ≤ *x* ⟹ *ennreal x* = *0* ⟷ *x* = *0*
by *transfer* (*auto simp*: *max-absorb2*)

lemma *ennreal-less-zero-iff*[*simp*]: *0* < *ennreal x* ⟷ *0* < *x*
by *transfer* (*auto simp*: *max-def*)

lemma *ennreal-lessI*: *0* < *q* ⟹ *r* < *q* ⟹ *ennreal r* < *ennreal q*
by (*cases 0* ≤ *r*) (*auto simp*: *ennreal-less-iff ennreal-neg*)

lemma *ennreal-leI*: *x* ≤ *y* ⟹ *ennreal x* ≤ *ennreal y*
by (*cases 0* ≤ *y*) (*auto simp*: *ennreal-neg*)

lemma *enn2ereal-ennreal*[*simp*]: *0* ≤ *x* ⟹ *enn2ereal* (*ennreal x*) = *x*
by *transfer* (*simp add*: *max-absorb2*)

lemma *e2ennreal-enn2ereal*[*simp*]: *e2ennreal* (*enn2ereal x*) = *x*
by (*simp add*: *e2ennreal-def max-absorb2 ennreal.enn2ereal-inverse*)

lemma *enn2ereal-e2ennreal*: *x* ≥ *0* ⟹ *enn2ereal* (*e2ennreal x*) = *x*
by (*metis e2ennreal-enn2ereal ereal-ennreal-cases not-le*)

lemma *e2ennreal-ereal* [*simp*]: *e2ennreal* (*ereal x*) = *ennreal x*
by (*metis e2ennreal-def enn2ereal-inverse ennreal.rep-eq sup-ereal-def*)

lemma *ennreal-0*[*simp*]: *ennreal 0 = 0*
by (*simp add*: *ennreal-def zero-ennreal.abs-eq*)

lemma *ennreal-1*[*simp*]: *ennreal 1 = 1*
by *transfer* (*simp add*: *max-absorb2*)

lemma *ennreal-eq-0-iff*: *ennreal x = 0* $\longleftrightarrow$ *x* $\leq$ *0*
by (*cases 0* $\leq$ *x*) (*auto simp*: *ennreal-neg*)

lemma *ennreal-le-iff2*: *ennreal x* $\leq$ *ennreal y* $\longleftrightarrow$ ((*0* $\leq$ *y* $\wedge$ *x* $\leq$ *y*) $\vee$ (*x* $\leq$ *0* $\wedge$ *y* $\leq$ *0*))
by (*cases 0* $\leq$ *y*) (*auto simp*: *ennreal-eq-0-iff ennreal-neg*)

lemma *ennreal-eq-1*[*simp*]: *ennreal x = 1* $\longleftrightarrow$ *x = 1*
by (*cases 0* $\leq$ *x*) (*auto simp*: *ennreal-neg simp flip*: *ennreal-1*)

lemma *ennreal-le-1*[*simp*]: *ennreal x* $\leq$ *1* $\longleftrightarrow$ *x* $\leq$ *1*
by (*cases 0* $\leq$ *x*) (*auto simp*: *ennreal-neg simp flip*: *ennreal-1*)

lemma *ennreal-ge-1*[*simp*]: *ennreal x* $\geq$ *1* $\longleftrightarrow$ *x* $\geq$ *1*
by (*cases 0* $\leq$ *x*) (*auto simp*: *ennreal-neg simp flip*: *ennreal-1*)

lemma *one-less-ennreal*[*simp*]: *1* $<$ *ennreal x* $\longleftrightarrow$ *1* $<$ *x*
by (*meson ennreal-le-1 linorder-not-le*)

lemma *ennreal-plus*[*simp*]:
0 $\leq$ *a* $\Longrightarrow$ *0* $\leq$ *b* $\Longrightarrow$ *ennreal* (*a* + *b*) = *ennreal a* + *ennreal b*
by (*transfer fixing*: *a b*) (*auto simp*: *max-absorb2*)

lemma *add-mono-ennreal*: *x* $<$ *ennreal y* $\Longrightarrow$ *x'* $<$ *ennreal y'* $\Longrightarrow$ *x* + *x'* $<$ *ennreal* (*y* + *y'*)
by (*metis* (*full-types*) *add-strict-mono ennreal-less-zero-iff ennreal-plus less-le not-less zero-le*)

lemma *sum-ennreal*[*simp*]: ($\bigwedge$*i*. *i* $\in$ *I* $\Longrightarrow$ *0* $\leq$ *f i*) $\Longrightarrow$ ($\sum$ *i*$\in$*I*. *ennreal* (*f i*)) = *ennreal* (*sum f I*)
by (*induction I rule*: *infinite-finite-induct*) (*auto simp*: *sum-nonneg*)

lemma *sum-list-ennreal*[*simp*]:
assumes $\bigwedge$*x*. *x* $\in$ *set xs* $\Longrightarrow$ *f x* $\geq$ *0*
shows *sum-list* (*map* (λ*x*. *ennreal* (*f x*)) *xs*) = *ennreal* (*sum-list* (*map f xs*))
using *assms*
proof (*induction xs*)
case (*Cons x xs*)
from *Cons* **have** ($\sum$ *x*$\leftarrow$*x* # *xs*. *ennreal* (*f x*)) = *ennreal* (*f x*) + *ennreal* (*sum-list* (*map f xs*))

by *simp*
also from *Cons.prems* **have** ... = *ennreal* (*f x* + *sum-list* (*map f xs*))
by (*intro ennreal-plus* [*symmetric*] *sum-list-nonneg*) *auto*
finally show *?case* **by** *simp*
qed *simp-all*

lemma *ennreal-of-nat-eq-real-of-nat*: *of-nat i* = *ennreal* (*of-nat i*)
by (*induction i*) *simp-all*

lemma *of-nat-le-ennreal-iff*[*simp*]: 0 ≤ *r* ⟹ *of-nat i* ≤ *ennreal r* ⟷ *of-nat i* ≤ *r*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-le-of-nat-iff*[*simp*]: *ennreal r* ≤ *of-nat i* ⟷ *r* ≤ *of-nat i*
by (*simp add*: *ennreal-of-nat-eq-real-of-nat*)

lemma *ennreal-indicator*: *ennreal* (*indicator A x*) = *indicator A x*
by (*auto split*: *split-indicator*)

lemma *ennreal-numeral*[*simp*]: *ennreal* (*numeral n*) = *numeral n*
using *ennreal-of-nat-eq-real-of-nat*[*of numeral n*] **by** *simp*

lemma *ennreal-less-numeral-iff* [*simp*]: *ennreal n* < *numeral w* ⟷ *n* < *numeral w*
by (*metis ennreal-less-iff ennreal-numeral less-le not-less zero-less-numeral*)

lemma *numeral-less-ennreal-iff* [*simp*]: *numeral w* < *ennreal n* ⟷ *numeral w* < *n*
using *ennreal-less-iff zero-le-numeral* **by** *fastforce*

lemma *numeral-le-ennreal-iff* [*simp*]: *numeral n* ≤ *ennreal m* ⟷ *numeral n* ≤ *m*
by (*metis not-le ennreal-less-numeral-iff*)

lemma *min-ennreal*: 0 ≤ *x* ⟹ 0 ≤ *y* ⟹ *min* (*ennreal x*) (*ennreal y*) = *ennreal* (*min x y*)
by (*auto split*: *split-min*)

lemma *ennreal-half*[*simp*]: *ennreal* (1/2) = *inverse* 2
by *transfer auto*

lemma *ennreal-minus*: 0 ≤ *q* ⟹ *ennreal r* − *ennreal q* = *ennreal* (*r* − *q*)
by *transfer* (*simp add*: *zero-ereal-def flip*: *ereal-max*)

lemma *ennreal-minus-top*[*simp*]: *ennreal a* − *top* = 0
by (*simp add*: *minus-top-ennreal*)

lemma *e2eenreal-enn2ereal-diff* [*simp*]:
e2ennreal(*enn2ereal x* − *enn2ereal y*) = *x* − *y* **for** *x y*
by (*cases x*, *cases y*, *auto simp add*: *ennreal-minus e2ennreal-neg*)

lemma *ennreal-mult*: $0 \leq a \Longrightarrow 0 \leq b \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
by *transfer* (*simp add*: *max-absorb2*)

lemma *ennreal-mult'*: $0 \leq a \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
by (*cases* $0 \leq b$) (*auto simp*: *ennreal-mult ennreal-neg mult-nonneg-nonpos*)

lemma *indicator-mult-ennreal*: $indicator\ A\ x * ennreal\ r = ennreal\ (indicator\ A\ x * r)$
by (*simp split*: *split-indicator*)

lemma *ennreal-mult''*: $0 \leq b \Longrightarrow ennreal\ (a * b) = ennreal\ a * ennreal\ b$
by (*cases* $0 \leq a$) (*auto simp*: *ennreal-mult ennreal-neg mult-nonpos-nonneg*)

lemma *numeral-mult-ennreal*: $0 \leq x \Longrightarrow numeral\ b * ennreal\ x = ennreal\ (numeral\ b * x)$
by (*simp add*: *ennreal-mult*)

lemma *ennreal-power*: $0 \leq r \Longrightarrow ennreal\ r\ \hat{}\ n = ennreal\ (r\ \hat{}\ n)$
by (*induction n*) (*auto simp*: *ennreal-mult*)

lemma *power-eq-top-ennreal*: $x\ \hat{}\ n = top \longleftrightarrow (n \neq 0 \wedge (x{::}ennreal) = top)$
using *not-gr-zero power-eq-top-ennreal-iff* **by** *force*

lemma *inverse-ennreal*: $0 < r \Longrightarrow inverse\ (ennreal\ r) = ennreal\ (inverse\ r)$
by *transfer* (*simp add*: *max.absorb2*)

lemma *divide-ennreal*: $0 \leq r \Longrightarrow 0 < q \Longrightarrow ennreal\ r\ /\ ennreal\ q = ennreal\ (r\ /\ q)$
by (*simp add*: *divide-ennreal-def inverse-ennreal ennreal-mult*[*symmetric*] *inverse-eq-divide*)

lemma *ennreal-inverse-power*: $inverse\ (x\ \hat{}\ n :: ennreal) = inverse\ x\ \hat{}\ n$
proof (*cases x rule*: *ennreal-cases*)
case *top* **with** *power-eq-top-ennreal*[*of x n*] **show** *?thesis*
by (*cases* $n = 0$) *auto*
next
case (*real r*) **then show** *?thesis*
proof (*cases* $x = 0$)
case *False* **then show** *?thesis*
by (*smt* (*verit*, *best*) *ennreal-0 ennreal-power inverse-ennreal*
inverse-nonnegative-iff-nonnegative power-inverse real zero-less-power)
qed (*simp add*: *top-power-ennreal*)
qed

lemma *power-divide-distrib-ennreal* [*algebra-simps*]:
$(x\ /\ y)\ \hat{}\ n = x\ \hat{}\ n\ /\ (y\ \hat{}\ n :: ennreal)$
by (*simp add*: *divide-ennreal-def ennreal-inverse-power power-mult-distrib*)

lemma *ennreal-divide-numeral*: $0 \leq x \Longrightarrow$ *ennreal x / numeral b = ennreal* (*x / numeral b*)
by (*subst divide-ennreal*[*symmetric*]) *auto*

lemma *prod-ennreal*: $(\bigwedge i.\ i \in A \Longrightarrow 0 \leq f\ i) \Longrightarrow (\prod i{\in}A.\ ennreal\ (f\ i)) = ennreal\ (prod\ f\ A)$
by (*induction A rule*: *infinite-finite-induct*)
(*auto simp*: *ennreal-mult prod-nonneg*)

lemma *prod-mono-ennreal*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \leq (g\ x :: ennreal)$
shows *prod f A* $\leq$ *prod g A*
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *mult-mono*)

lemma *mult-right-ennreal-cancel*: $a * ennreal\ c = b * ennreal\ c \longleftrightarrow (a = b \vee c \leq 0)$
by (*metis ennreal-eq-0-iff mult-divide-eq-ennreal mult-eq-0-iff top-neq-ennreal*)

lemma *ennreal-le-epsilon*:
$(\bigwedge e{::}real.\ y < top \Longrightarrow 0 < e \Longrightarrow x \leq y + ennreal\ e) \Longrightarrow x \leq y$
apply (*cases y rule*: *ennreal-cases*)
apply (*cases x rule*: *ennreal-cases*)
apply (*auto simp flip*: *ennreal-plus simp add*: *top-unique intro*: *zero-less-one field-le-epsilon*)
done

lemma *ennreal-rat-dense*:
fixes *x y* :: *ennreal*
shows $x < y \Longrightarrow \exists\, r{::}rat.\ x < real\text{-}of\text{-}rat\ r \wedge real\text{-}of\text{-}rat\ r < y$
proof *transfer*
fix *x y* :: *ereal* **assume** *xy*: $0 \leq x\ 0 \leq y\ x < y$
moreover
from *ereal-dense3*[*OF* ‹*x* < *y*›]
obtain *r* **where** *r*: *x* < *ereal* (*real-of-rat r*) *ereal* (*real-of-rat r*) < *y*
by *auto*
then have $0 \leq r$
using *le-less-trans*[*OF* ‹0 ≤ *x*› ‹*x* < *ereal* (*real-of-rat r*)›] **by** *auto*
with *r* **show** $\exists\, r.\ x < (sup\ 0 \circ ereal)\ (real\text{-}of\text{-}rat\ r) \wedge (sup\ 0 \circ ereal)\ (real\text{-}of\text{-}rat\ r) < y$
by (*intro exI*[*of* - *r*]) (*auto simp*: *max-absorb2*)
qed

lemma *ennreal-Ex-less-of-nat*: (*x*::*ennreal*) < *top* $\Longrightarrow \exists\, n.\ x <$ *of-nat n*
by (*cases x rule*: *ennreal-cases*)
(*auto simp*: *ennreal-of-nat-eq-real-of-nat ennreal-less-iff reals-Archimedean2*)

40.6 Coercion from *ennreal* to *real*

definition *enn2real x = real-of-ereal (enn2ereal x)*

lemma *enn2real-nonneg*[*simp*]: *0 ≤ enn2real x*
by (*auto simp*: *enn2real-def intro*!: *real-of-ereal-pos enn2ereal-nonneg*)

lemma *enn2real-mono*: *a ≤ b ⟹ b < top ⟹ enn2real a ≤ enn2real b*
by (*auto simp add*: *enn2real-def less-eq-ennreal.rep-eq intro*!: *real-of-ereal-positive-mono enn2ereal-nonneg*)

lemma *enn2real-of-nat*[*simp*]: *enn2real (of-nat n) = n*
by (*auto simp*: *enn2real-def*)

lemma *enn2real-ennreal*[*simp*]: *0 ≤ r ⟹ enn2real (ennreal r) = r*
by (*simp add*: *enn2real-def*)

lemma *ennreal-enn2real*[*simp*]: *r < top ⟹ ennreal (enn2real r) = r*
by (*cases r rule*: *ennreal-cases*) *auto*

lemma *real-of-ereal-enn2ereal*[*simp*]: *real-of-ereal (enn2ereal x) = enn2real x*
by (*simp add*: *enn2real-def*)

lemma *enn2real-top*[*simp*]: *enn2real top = 0*
unfolding *enn2real-def top-ennreal.rep-eq top-ereal-def* **by** *simp*

lemma *enn2real-0*[*simp*]: *enn2real 0 = 0*
unfolding *enn2real-def zero-ennreal.rep-eq* **by** *simp*

lemma *enn2real-1*[*simp*]: *enn2real 1 = 1*
unfolding *enn2real-def one-ennreal.rep-eq* **by** *simp*

lemma *enn2real-numeral*[*simp*]: *enn2real (numeral n) = (numeral n)*
unfolding *enn2real-def* **by** *simp*

lemma *enn2real-mult*: *enn2real (a * b) = enn2real a * enn2real b*
unfolding *enn2real-def*
by (*simp del*: *real-of-ereal-enn2ereal add*: *times-ennreal.rep-eq*)

lemma *enn2real-leI*: *0 ≤ B ⟹ x ≤ ennreal B ⟹ enn2real x ≤ B*
by (*cases x rule*: *ennreal-cases*) (*auto simp*: *top-unique*)

lemma *enn2real-positive-iff*: *0 < enn2real x ⟷ (0 < x ∧ x < top)*
by (*cases x rule*: *ennreal-cases*) *auto*

lemma *enn2real-eq-posreal-iff*[*simp*]: *c > 0 ⟹ enn2real x = c ⟷ x = c*
by (*cases x*) *auto*

lemma *ennreal-enn2real-if*: *ennreal (enn2real r) = (if r = top then 0 else r)*
by(*auto intro*!: *ennreal-enn2real simp add*: *less-top*)

40.7 Coercion from *enat* to *ennreal*

definition *ennreal-of-enat* :: *enat* ⇒ *ennreal*
where
ennreal-of-enat n = (*case n of* ∞ ⇒ *top* | *enat n* ⇒ *of-nat n*)

declare [[*coercion ennreal-of-enat*]]
declare [[*coercion of-nat* :: *nat* ⇒ *ennreal*]]

lemma *ennreal-of-enat-infty*[*simp*]: *ennreal-of-enat* ∞ = ∞
by (*simp add*: *ennreal-of-enat-def*)

lemma *ennreal-of-enat-enat*[*simp*]: *ennreal-of-enat* (*enat n*) = *of-nat n*
by (*simp add*: *ennreal-of-enat-def*)

lemma *ennreal-of-enat-0*[*simp*]: *ennreal-of-enat 0* = *0*
using *ennreal-of-enat-enat*[*of 0*] **unfolding** *enat-0* **by** *simp*

lemma *ennreal-of-enat-1*[*simp*]: *ennreal-of-enat 1* = *1*
using *ennreal-of-enat-enat*[*of 1*] **unfolding** *enat-1* **by** *simp*

lemma *ennreal-top-neq-of-nat*[*simp*]: (*top*::*ennreal*) ≠ *of-nat i*
using *ennreal-of-nat-neq-top*[*of i*] **by** *metis*

lemma *ennreal-of-enat-inj*[*simp*]: *ennreal-of-enat i* = *ennreal-of-enat j* ⟷ *i* = *j*
by (*cases i j rule*: *enat.exhaust*[*case-product enat.exhaust*]) *auto*

lemma *ennreal-of-enat-le-iff*[*simp*]: *ennreal-of-enat m* ≤ *ennreal-of-enat n* ⟷ *m* ≤ *n*
by (*auto simp*: *ennreal-of-enat-def top-unique split*: *enat.split*)

lemma *of-nat-less-ennreal-of-nat*[*simp*]: *of-nat n* ≤ *ennreal-of-enat x* ⟷ *of-nat n* ≤ *x*
by (*cases x*) (*auto simp*: *of-nat-eq-enat*)

lemma *ennreal-of-enat-Sup*: *ennreal-of-enat* (*Sup X*) = (*SUP x*∈*X*. *ennreal-of-enat x*)
proof −
have *ennreal-of-enat* (*Sup X*) ≤ (*SUP x* ∈ *X*. *ennreal-of-enat x*)
unfolding *Sup-enat-def*
proof (*clarsimp*, *intro conjI impI*)
fix *x* **assume** *finite X X* ≠ {}
then show *ennreal-of-enat* (*Max X*) ≤ (*SUP x* ∈ *X*. *ennreal-of-enat x*)
by (*intro SUP-upper Max-in*)
next
assume *infinite X X* ≠ {}
have ∃ *y*∈*X*. *r* < *ennreal-of-enat y* **if** *r*: *r* < *top* **for** *r*
proof −
obtain *n* **where** *n*: *r* < *of-nat n*
using *ennreal-Ex-less-of-nat*[*OF r*] **..**

have $\neg$ ($X \subseteq$ *enat* ' {.. n})
using ‹*infinite X*› **by** (*auto dest*: *finite-subset*)
then obtain x **where** x: $x \in X$ $x \notin$ *enat* ' {..n}
by *blast*
then have *of-nat* $n \leq x$
by (*cases x*) (*auto simp*: *of-nat-eq-enat*)
with x **show** *?thesis*
by (*auto intro*!: *bexI*[*of - x*] *less-le-trans*[*OF n*])
qed
then have (*SUP* $x \in X$. *ennreal-of-enat x*) = *top*
by *simp*
then show *top* $\leq$ (*SUP* $x \in X$. *ennreal-of-enat x*)
unfolding *top-unique* **by** *simp*
qed
then show *?thesis*
by (*auto intro*!: *antisym Sup-least intro*: *Sup-upper*)
qed

lemma *ennreal-of-enat-eSuc*[*simp*]: *ennreal-of-enat* (*eSuc x*) = *1* + *ennreal-of-enat x*
by (*cases x*) (*auto simp*: *eSuc-enat*)

lemma *ennreal-of-enat-plus*[*simp*]: ‹*ennreal-of-enat* (*a*+*b*) = *ennreal-of-enat a* + *ennreal-of-enat b*›
proof (*induct a*)
case (*enat nat*)
with *enat.simps* **show** *?case*
by (*smt* (*verit*, *del-insts*) *add.commute add-top-left-ennreal enat.exhaust enat-defs*(*4*) *ennreal-of-enat-def of-nat-add*)
qed *auto*

lemma *sum-ennreal-of-enat*[*simp*]: ($\sum i \in I$. *ennreal-of-enat* (*f i*)) = *ennreal-of-enat* (*sum f I*)
by (*induct I rule*: *infinite-finite-induct*) (*auto simp*: *sum-nonneg*)

40.8 Topology on *ennreal*

lemma *enn2ereal-Iio*: *enn2ereal* $-$ ' {..<*a*} = (*if 0* $\leq$ *a then* {..< *e2ennreal a*} *else* {})
using *enn2ereal-nonneg*
by (*cases a rule*: *ereal-ennreal-cases*)
(*auto simp add*: *vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq e2ennreal-def*
simp del: *enn2ereal-nonneg*
intro: *le-less-trans less-imp-le*)

lemma *enn2ereal-Ioi*: *enn2ereal* $-$ ' {*a* <..} = (*if 0* $\leq$ *a then* {*e2ennreal a* <..}

else UNIV)
by (*cases a rule*: *ereal-ennreal-cases*)
(*auto simp add*: *vimage-def set-eq-iff ennreal.enn2ereal-inverse less-ennreal.rep-eq e2ennreal-def*
intro: *less-le-trans*)

instantiation *ennreal* :: *linear-continuum-topology*
begin

definition *open-ennreal* :: *ennreal set* ⇒ *bool*
where (*open* :: *ennreal set* ⇒ *bool*) = *generate-topology* (*range lessThan* ∪ *range greaterThan*)

instance
proof
show ∃ *a b*::*ennreal*. *a* ≠ *b*
using *zero-neq-one* **by** (*intro exI*)
show ⋀*x y*::*ennreal*. *x* < *y* ⟹ ∃ *z*>*x*. *z* < *y*
proof *transfer*
fix *x y* :: *ereal*
assume ∗: *0* ≤ *x*
assume *x* < *y*
from *dense*[*OF this*] **obtain** *z* **where** *x* < *z* ∧ *z* < *y* **..**
with ∗ **show** ∃ *z*∈*Collect* ((≤) *0*). *x* < *z* ∧ *z* < *y*
by (*intro bexI*[*of* - *z*]) *auto*
qed
qed (*rule open-ennreal-def*)

end

lemma *continuous-on-e2ennreal*: *continuous-on A e2ennreal*
proof (*rule continuous-on-subset*)
show *continuous-on* ({*0*..} ∪ {..*0*}) *e2ennreal*
proof (*rule continuous-on-closed-Un*)
show *continuous-on* {*0* ..} *e2ennreal*
by (*simp add*: *continuous-onI-mono e2ennreal-mono enn2ereal-range*)
show *continuous-on* {.. *0*} *e2ennreal*
by (*metis atMost-iff continuous-on-cong continuous-on-const e2ennreal-neg*)
qed *auto*
qed *auto*

lemma *continuous-at-e2ennreal*: *continuous* (*at x within A*) *e2ennreal*
using *continuous-on-e2ennreal continuous-on-imp-continuous-within top.extremum*
by *blast*

lemma *continuous-on-enn2ereal*: *continuous-on UNIV enn2ereal*
by (*rule continuous-on-generate-topology*[*OF open-generated-order*])
(*auto simp add*: *enn2ereal-Iio enn2ereal-Ioi*)

lemma *continuous-at-enn2ereal*: *continuous* (*at x within A*) *enn2ereal*
by (*meson UNIV-I continuous-at-imp-continuous-at-within*
continuous-on-enn2ereal continuous-on-eq-continuous-within)

lemma *sup-continuous-e2ennreal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *e2ennreal* (*f x*))
proof (*rule sup-continuous-compose*[*OF - f*])
show *sup-continuous e2ennreal*
by (*simp add*: *continuous-at-e2ennreal continuous-at-left-imp-sup-continuous e2ennreal-mono mono-def*)
qed

lemma *sup-continuous-enn2ereal*[*order-continuous-intros*]:
assumes *f*: *sup-continuous f* **shows** *sup-continuous* (λ*x*. *enn2ereal* (*f x*))
proof (*rule sup-continuous-compose*[*OF - f*])
show *sup-continuous enn2ereal*
by (*simp add*: *continuous-at-enn2ereal continuous-at-left-imp-sup-continuous less-eq-ennreal.rep-eq mono-def*)
qed

lemma *sup-continuous-mult-left-ennreal′*:
fixes *c* :: *ennreal*
shows *sup-continuous* (λ*x*. *c* ∗ *x*)
unfolding *sup-continuous-def*
by *transfer* (*auto simp*: *SUP-ereal-mult-left max.absorb2 SUP-upper2*)

lemma *sup-continuous-mult-left-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *c* ∗ *f x* :: *ennreal*)
by (*rule sup-continuous-compose*[*OF sup-continuous-mult-left-ennreal′*])

lemma *sup-continuous-mult-right-ennreal*[*order-continuous-intros*]:
sup-continuous f ⟹ *sup-continuous* (λ*x*. *f x* ∗ *c* :: *ennreal*)
using *sup-continuous-mult-left-ennreal*[*of f c*] **by** (*simp add*: *mult.commute*)

lemma *sup-continuous-divide-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *′a*::*complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous* (λ*x*. *f x* / *c*)
unfolding *divide-ennreal-def* **by** (*rule sup-continuous-mult-right-ennreal*)

lemma *transfer-enn2ereal-continuous-on* [*transfer-rule*]:
rel-fun (=) (*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *continuous-on continuous-on*
proof −
have *continuous-on A f* **if** *continuous-on A* (λ*x*. *enn2ereal* (*f x*)) **for** *A* **and** *f* :: *′a* ⇒ *ennreal*
using *continuous-on-compose2*[*OF continuous-on-e2ennreal*[*of* {*0*..}] *that*]
by (*auto simp*: *ennreal.enn2ereal-inverse subset-eq e2ennreal-def max-absorb2*)
moreover
have *continuous-on A* (λ*x*. *enn2ereal* (*f x*)) **if** *continuous-on A f* **for** *A* **and** *f* :: *′a* ⇒ *ennreal*

using *continuous-on-compose2*[*OF continuous-on-enn2ereal that*] **by** *auto*
ultimately
show *?thesis*
by (*auto simp add*: *rel-fun-def ennreal.pcr-cr-eq cr-ennreal-def*)
qed

lemma *transfer-sup-continuous*[*transfer-rule*]:
(*rel-fun* (*rel-fun* (=) *pcr-ennreal*) (=)) *sup-continuous sup-continuous*
proof (*safe intro*!: *rel-funI dest*!: *rel-fun-eq-pcr-ennreal*[*THEN iffD1*])
show *sup-continuous* (*enn2ereal* ∘ *f*) ⟹ *sup-continuous f* **for** *f* :: *'a* ⇒ -
using *sup-continuous-e2ennreal*[*of enn2ereal* ∘ *f*] **by** *simp*
show *sup-continuous f* ⟹ *sup-continuous* (*enn2ereal* ∘ *f*) **for** *f* :: *'a* ⇒ -
using *sup-continuous-enn2ereal*[*of f*] **by** (*simp add*: *comp-def*)
qed

lemma *continuous-on-ennreal*[*tendsto-intros*]:
continuous-on A f ⟹ *continuous-on A* (λ*x*. *ennreal* (*f x*))
by *transfer* (*auto intro*!: *continuous-on-max continuous-on-const continuous-on-ereal*)

lemma *tendsto-ennrealD*:
assumes *lim*: ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
assumes ∗: ∀ *x in F*. *0* ≤ *f x* **and** *x*: *0* ≤ *x*
shows (*f* ⟶ *x*) *F*
proof −
have ((λ*x*. *enn2ereal* (*ennreal* (*f x*))) ⟶ *enn2ereal* (*ennreal x*)) *F*
⟷ (*f* ⟶ *enn2ereal* (*ennreal x*)) *F*
using ∗ *eventually-mono*
by (*intro tendsto-cong*) *fastforce*
then show *?thesis*
using *assms*(*1*) *continuous-at-enn2ereal isCont-tendsto-compose x* **by** *fastforce*
qed

lemma *tendsto-ennreal-iff* [*simp*]:
‹((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F* ⟷ (*f* ⟶ *x*) *F*› (**is** ‹*?P* ⟷ *?Q*›)
if ‹∀ *x in F*. *0* ≤ *f x*› ‹*0* ≤ *x*›
proof
assume ‹*?P*›
then show ‹*?Q*›
using *that* **by** (*rule tendsto-ennrealD*)
next
assume ‹*?Q*›
have ‹*continuous-on UNIV ereal*›
using *continuous-on-ereal* [*of - id*] **by** *simp*
then have ‹*continuous-on UNIV* (*e2ennreal* ∘ *ereal*)›
by (*rule continuous-on-compose*) (*simp-all add*: *continuous-on-e2ennreal*)
then have ‹((λ*x*. (*e2ennreal* ∘ *ereal*) (*f x*)) ⟶ (*e2ennreal* ∘ *ereal*) *x*) *F*›
using ‹*?Q*› **by** (*rule continuous-on-tendsto-compose*) *simp-all*
then show ‹*?P*›
by (*simp flip*: *e2ennreal-ereal*)

qed

lemma *tendsto-enn2ereal-iff*[*simp*]: $((\lambda i.\ enn2ereal\ (f\ i)) \longrightarrow enn2ereal\ x)\ F \longleftrightarrow (f \longrightarrow x)\ F$
using *continuous-on-enn2ereal*[*THEN continuous-on-tendsto-compose*, *of f x F*]
continuous-on-e2ennreal[*THEN continuous-on-tendsto-compose*, *of* $\lambda x.$ *enn2ereal* $(f\ x)$ *enn2ereal x F UNIV*]
by *auto*

lemma *ennreal-tendsto-0-iff*: $(\bigwedge n.\ f\ n \geq 0) \Longrightarrow ((\lambda n.\ ennreal\ (f\ n)) \longrightarrow 0) \longleftrightarrow (f \longrightarrow 0)$
by (*metis* (*mono-tags*) *ennreal-0 eventuallyI order-refl tendsto-ennreal-iff*)

lemma *continuous-on-add-ennreal*:
fixes *f g* :: *'a*::*topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A g* ⟹ *continuous-on A* $(\lambda x.\ f\ x + g\ x)$
by (*transfer fixing*: *A*) (*auto intro*!: *tendsto-add-ereal-nonneg simp*: *continuous-on-def*)

lemma *continuous-on-inverse-ennreal*[*continuous-intros*]:
fixes *f* :: *'a*::*topological-space* ⇒ *ennreal*
shows *continuous-on A f* ⟹ *continuous-on A* $(\lambda x.\ inverse\ (f\ x))$
proof (*transfer fixing*: *A*)
show *pred-fun top* $((\leq)\ 0)\ f$ ⟹ *continuous-on A* $(\lambda x.\ inverse\ (f\ x))$ **if** *continuous-on A f*
for *f* :: *'a* ⇒ *ereal*
using *continuous-on-compose2*[*OF continuous-on-inverse-ereal that*] **by** (*auto simp*: *subset-eq*)
qed

instance *ennreal* :: *topological-comm-monoid-add*
proof
show $((\lambda x.\ fst\ x + snd\ x) \longrightarrow a + b)\ (nhds\ a \times_F nhds\ b)$ **for** *a b* :: *ennreal*
using *continuous-on-add-ennreal*[*of UNIV fst snd*]
using *tendsto-at-iff-tendsto-nhds*[*symmetric*, *of* λx::(*ennreal* × *ennreal*). *fst x + snd x*]
by (*auto simp*: *continuous-on-eq-continuous-at*)
(*simp add*: *isCont-def nhds-prod*[*symmetric*])
qed

lemma *sup-continuous-add-ennreal*[*order-continuous-intros*]:
fixes *f g* :: *'a*::*complete-lattice* ⇒ *ennreal*
shows *sup-continuous f* ⟹ *sup-continuous g* ⟹ *sup-continuous* $(\lambda x.\ f\ x + g\ x)$
by *transfer* (*auto intro*!: *sup-continuous-add*)

lemma *ennreal-suminf-lessD*: $(\sum i.\ f\ i :: ennreal) < x \Longrightarrow f\ i < x$
using *le-less-trans*[*OF sum-le-suminf*[*OF summableI*, *of* {*i*} *f*]] **by** *simp*

lemma *sums-ennreal*[*simp*]: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow 0 \le x \Longrightarrow (\lambda i.\ ennreal\ (f\ i))\ sums\ ennreal\ x \longleftrightarrow f\ sums\ x$
unfolding *sums-def* **by** (*simp add*: *always-eventually sum-nonneg*)

lemma *summable-suminf-not-top*: $(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow summable\ f$
using *summable-sums*[*OF summableI*, *of* $\lambda i.\ ennreal\ (f\ i)$]
by (*cases* $\sum i.\ ennreal\ (f\ i)$ *rule*: *ennreal-cases*)
(*auto simp*: *summable-def*)

lemma *suminf-ennreal*[*simp*]:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
by (*rule sums-unique*[*symmetric*]) (*simp add*: *summable-suminf-not-top suminf-nonneg summable-sums*)

lemma *sums-enn2ereal*[*simp*]: $(\lambda i.\ enn2ereal\ (f\ i))\ sums\ enn2ereal\ x \longleftrightarrow f\ sums\ x$
unfolding *sums-def* **by** (*simp add*: *always-eventually sum-nonneg*)

lemma *suminf-enn2ereal*[*simp*]: $(\sum i.\ enn2ereal\ (f\ i)) = enn2ereal\ (suminf\ f)$
by (*metis summableI summable-sums sums-enn2ereal sums-unique*)

lemma *transfer-e2ennreal-suminf* [*transfer-rule*]: *rel-fun* (*rel-fun* (=) *pcr-ennreal*) *pcr-ennreal suminf suminf*
by (*auto simp*: *rel-funI rel-fun-eq-pcr-ennreal comp-def*)

lemma *ennreal-suminf-cmult*[*simp*]: $(\sum i.\ r * f\ i) = r * (\sum i.\ f\ i :: ennreal)$
by *transfer* (*auto intro*!: *suminf-cmult-ereal*)

lemma *ennreal-suminf-multc*[*simp*]: $(\sum i.\ f\ i * r) = (\sum i.\ f\ i :: ennreal) * r$
using *ennreal-suminf-cmult*[*of r f*] **by** (*simp add*: *ac-simps*)

lemma *ennreal-suminf-divide*[*simp*]: $(\sum i.\ f\ i\ /\ r) = (\sum i.\ f\ i :: ennreal)\ /\ r$
by (*simp add*: *divide-ennreal-def*)

lemma *ennreal-suminf-neq-top*: $summable\ f \Longrightarrow (\bigwedge i.\ 0 \le f\ i) \Longrightarrow (\sum i.\ ennreal\ (f\ i)) \neq top$
using *sums-ennreal*[*of f suminf f*]
by (*simp add*: *suminf-nonneg flip*: *sums-unique summable-sums-iff del*: *sums-ennreal*)

lemma *suminf-ennreal-eq*:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow f\ sums\ x \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ x$
using *suminf-nonneg*[*of f*] *sums-unique*[*of f x*]
by (*intro sums-unique*[*symmetric*]) (*auto simp*: *summable-sums-iff*)

lemma *ennreal-suminf-bound-add*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge N.\ (\sum n{<}N.\ f\ n) + y \le x) \Longrightarrow suminf\ f + y \le x$

by *transfer* (*auto intro*!: *suminf-bound-add*)

lemma *ennreal-suminf-SUP-eq-directed*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
assumes $*$: $\bigwedge N\ i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow finite\ N \Longrightarrow \exists k{\in}I.\ \forall n{\in}N.\ f\ i\ n \leq f\ k\ n \wedge f\ j\ n \leq f\ k\ n$
shows $(\sum n.\ SUP\ i{\in}I.\ f\ i\ n) = (SUP\ i{\in}I.\ \sum n.\ f\ i\ n)$
proof *cases*
assume $I \neq \{\}$
then obtain i **where** $i \in I$ **by** *auto*
from $*$ **show** *?thesis*
by (*transfer fixing*: I)
(*auto simp*: *max-absorb2 SUP-upper2*[*OF* ‹$i \in I$›] *suminf-nonneg summable-ereal-pos* ‹$I \neq \{\}$›
intro!: *suminf-SUP-eq-directed*)
qed (*simp add*: *bot-ennreal*)

lemma *INF-ennreal-add-const*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ f\ i + c) = (INF\ i.\ f\ i) + c$
using *continuous-at-Inf-mono*[*of* $\lambda x.\ x + c$ *f'UNIV*]
using *continuous-add*[*of at-right* (*Inf* (*range f*)), *of* $\lambda x.\ x$ $\lambda x.\ c$]
by (*auto simp*: *mono-def image-comp*)

lemma *INF-ennreal-const-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $(INF\ i.\ c + f\ i) = c + (INF\ i.\ f\ i)$
using *INF-ennreal-add-const*[*of f c*] **by** (*simp add*: *ac-simps*)

lemma *SUP-mult-left-ennreal*: $c * (SUP\ i{\in}I.\ f\ i) = (SUP\ i{\in}I.\ c * f\ i :: ennreal)$
proof *cases*
assume $I \neq \{\}$ **then show** *?thesis*
by *transfer* (*auto simp add*: *SUP-ereal-mult-left max-absorb2 SUP-upper2*)
qed (*simp add*: *bot-ennreal*)

lemma *SUP-mult-right-ennreal*: $(SUP\ i{\in}I.\ f\ i) * c = (SUP\ i{\in}I.\ f\ i * c :: ennreal)$
using *SUP-mult-left-ennreal* **by** (*simp add*: *mult.commute*)

lemma *SUP-divide-ennreal*: $(SUP\ i{\in}I.\ f\ i)\ /\ c = (SUP\ i{\in}I.\ f\ i\ /\ c :: ennreal)$
using *SUP-mult-right-ennreal* **by** (*simp add*: *divide-ennreal-def*)

lemma *ennreal-SUP-of-nat-eq-top*: $(SUP\ x.\ of\text{-}nat\ x :: ennreal) = top$
proof (*intro antisym top-greatest le-SUP-iff*[*THEN iffD2*] *allI impI*)
fix $y :: ennreal$ **assume** $y < top$
then obtain r **where** $y = ennreal\ r$
by (*cases y rule*: *ennreal-cases*) *auto*
then show $\exists i{\in}UNIV.\ y < of\text{-}nat\ i$
using *reals-Archimedean2*[*of max 1 r*] *zero-less-one*
by (*simp add*: *ennreal-Ex-less-of-nat*)

qed

lemma *ennreal-SUP-eq-top*:
fixes $f :: 'a \Rightarrow ennreal$
assumes $\bigwedge n.\ \exists i{\in}I.\ of\text{-}nat\ n \leq f\ i$
shows $(SUP\ i \in I.\ f\ i) = top$
proof −
have $(SUP\ x.\ of\text{-}nat\ x :: ennreal) \leq (SUP\ i \in I.\ f\ i)$
using *assms* **by** (*auto intro*!: *SUP-least intro*: *SUP-upper2*)
then show *?thesis*
by (*auto simp*: *ennreal-SUP-of-nat-eq-top top-unique*)
qed

lemma *ennreal-INF-const-minus*:
fixes $f :: 'a \Rightarrow ennreal$
shows $I \neq \{\} \Longrightarrow (SUP\ x{\in}I.\ c - f\ x) = c - (INF\ x{\in}I.\ f\ x)$
by (*transfer fixing*: *I*)
(*simp add*: *sup-max*[*symmetric*] *SUP-sup-const1 SUP-ereal-minus-right del*: *sup-ereal-def*)

lemma *of-nat-Sup-ennreal*:
assumes $A \neq \{\}$ *bdd-above A*
shows *of-nat* (*Sup A*) = (*SUP a*∈*A*. *of-nat a* :: *ennreal*)
proof (*intro antisym*)
show (*SUP a*∈*A*. *of-nat a*::*ennreal*) ≤ *of-nat* (*Sup A*)
by (*intro SUP-least of-nat-mono*) (*auto intro*: *cSup-upper assms*)
have *Sup A* ∈ *A*
using *assms* **by** (*auto simp*: *Sup-nat-def bdd-above-nat*)
then show *of-nat* (*Sup A*) ≤ (*SUP a*∈*A*. *of-nat a*::*ennreal*)
by (*intro SUP-upper*)
qed

lemma *ennreal-tendsto-const-minus*:
fixes $g :: 'a \Rightarrow ennreal$
assumes *ae*: $\forall_F\ x\ in\ F.\ g\ x \leq c$
assumes *g*: $((\lambda x.\ c - g\ x) \longrightarrow 0)\ F$
shows $(g \longrightarrow c)\ F$
proof (*cases c rule*: *ennreal-cases*)
case *top* **with** *tendsto-unique*[*OF - g*, *of top*] **show** *?thesis*
by (*cases F* = *bot*) *auto*
next
case (*real r*)
then have $\forall x.\ \exists q{\geq}0.\ g\ x \leq c \longrightarrow (g\ x = ennreal\ q \wedge q \leq r)$
by (*auto simp*: *le-ennreal-iff*)
then obtain *f* **where** ∗: $0 \leq f\ x\ g\ x = ennreal\ (f\ x)\ f\ x \leq r$ **if** $g\ x \leq c$ **for** *x*
by *metis*
from *ae* **have** *ae2*: $\forall_F\ x\ in\ F.\ c - g\ x = ennreal\ (r - f\ x) \wedge f\ x \leq r \wedge g\ x = ennreal\ (f\ x) \wedge 0 \leq f\ x$
proof *eventually-elim*

fix x **assume** $g\ x \leq c$ **with** $*[of\ x]$ ‹$0 \leq r$› **show** $c - g\ x = ennreal\ (r - f\ x) \wedge f\ x \leq r \wedge g\ x = ennreal\ (f\ x) \wedge 0 \leq f\ x$
by (*auto simp*: *real ennreal-minus*)
qed
with g **have** $((\lambda x.\ ennreal\ (r - f\ x)) \longrightarrow ennreal\ 0)\ F$
by (*auto simp add*: *tendsto-cong eventually-conj-iff*)
with *ae2* **have** $((\lambda x.\ r - f\ x) \longrightarrow 0)\ F$
by (*subst* (*asm*) *tendsto-ennreal-iff*) (*auto elim*: *eventually-mono*)
then have $(f \longrightarrow r)\ F$
by (*rule Lim-transform2*[*OF tendsto-const*])
with *ae2* **have** $((\lambda x.\ ennreal\ (f\ x)) \longrightarrow ennreal\ r)\ F$
by (*subst tendsto-ennreal-iff*) (*auto elim*: *eventually-mono simp*: *real*)
with *ae2* **show** *?thesis*
by (*auto simp*: *real tendsto-cong eventually-conj-iff*)
qed

lemma *ennreal-SUP-add*:
fixes $f\ g :: nat \Rightarrow ennreal$
shows $incseq\ f \Longrightarrow incseq\ g \Longrightarrow (SUP\ i.\ f\ i + g\ i) = Sup\ (f\ `\ UNIV) + Sup\ (g\ `\ UNIV)$
unfolding *incseq-def le-fun-def*
by *transfer*
(*simp add*: *SUP-ereal-add incseq-def le-fun-def max-absorb2 SUP-upper2*)

lemma *ennreal-SUP-sum*:
fixes $f :: 'a \Rightarrow nat \Rightarrow ennreal$
shows $(\bigwedge i.\ i \in I \Longrightarrow incseq\ (f\ i)) \Longrightarrow (SUP\ n.\ \sum i{\in}I.\ f\ i\ n) = (\sum i{\in}I.\ SUP\ n.\ f\ i\ n)$
unfolding *incseq-def*
by *transfer*
(*simp add*: *SUP-ereal-sum incseq-def SUP-upper2 max-absorb2 sum-nonneg*)

lemma *ennreal-liminf-minus*:
fixes $f :: nat \Rightarrow ennreal$
shows $(\bigwedge n.\ f\ n \leq c) \Longrightarrow liminf\ (\lambda n.\ c - f\ n) = c - limsup\ f$
apply *transfer*
apply (*simp add*: *ereal-diff-positive liminf-ereal-cminus*)
by (*metis max.absorb2 ereal-diff-positive Limsup-bounded eventually-sequentiallyI*)

lemma *ennreal-continuous-on-cmult*:
$(c{::}ennreal) < top \Longrightarrow continuous\text{-}on\ A\ f \Longrightarrow continuous\text{-}on\ A\ (\lambda x.\ c * f\ x)$
by (*transfer fixing*: *A*) (*auto intro*: *continuous-on-cmult-ereal*)

lemma *ennreal-tendsto-cmult*:
$(c{::}ennreal) < top \Longrightarrow (f \longrightarrow x)\ F \Longrightarrow ((\lambda x.\ c * f\ x) \longrightarrow c * x)\ F$
by (*rule continuous-on-tendsto-compose*[**where** *g*=*f*, *OF ennreal-continuous-on-cmult*, **where** *s*=*UNIV*])
(*auto simp*: *continuous-on-id*)

lemma *tendsto-ennrealI*[*intro*, *simp*, *tendsto-intros*]:
(*f* ⟶ *x*) *F* ⟹ ((λ*x*. *ennreal* (*f x*)) ⟶ *ennreal x*) *F*
by (*auto simp*: *ennreal-def*
intro!: *continuous-on-tendsto-compose*[*OF continuous-on-e2ennreal*[*of UNIV*]] *tendsto-max*)

lemma *tendsto-enn2erealI* [*tendsto-intros*]:
assumes (*f* ⟶ *l*) *F*
shows ((λ*i*. *enn2ereal*(*f i*)) ⟶ *enn2ereal l*) *F*
using *tendsto-enn2ereal-iff assms* **by** *auto*

lemma *tendsto-e2ennrealI* [*tendsto-intros*]:
assumes (*f* ⟶ *l*) *F*
shows ((λ*i*. *e2ennreal*(*f i*)) ⟶ *e2ennreal l*) *F*
proof −
have ∗: *e2ennreal* (*max x 0*) = *e2ennreal x* **for** *x*
by (*simp add*: *e2ennreal-def max.commute*)
have ((λ*i*. *max* (*f i*) *0*) ⟶ *max l 0*) *F*
using *assms* **by** (*intro tendsto-intros*) *auto*
then have ((λ*i*. *enn2ereal*(*e2ennreal* (*max* (*f i*) *0*))) ⟶ *enn2ereal* (*e2ennreal* (*max l 0*))) *F*
by (*subst enn2ereal-e2ennreal*, *auto*)+
then have ((λ*i*. *e2ennreal* (*max* (*f i*) *0*)) ⟶ *e2ennreal* (*max l 0*)) *F*
using *tendsto-enn2ereal-iff* **by** *auto*
then show *?thesis*
unfolding ∗ **by** *auto*
qed

lemma *ennreal-suminf-minus*:
fixes *f g* :: *nat* ⇒ *ennreal*
shows (⋀*i*. *g i* ≤ *f i*) ⟹ *suminf f* ≠ *top* ⟹ *suminf g* ≠ *top* ⟹ (∑ *i*. *f i* − *g i*) = *suminf f* − *suminf g*
by *transfer*
(*auto simp add*: *ereal-diff-positive suminf-le-pos top-ereal-def intro*!: *suminf-ereal-minus*)

lemma *ennreal-Sup-countable-SUP*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ennreal*. *incseq f* ∧ *range f* ⊆ *A* ∧ *Sup A* = (*SUP i*. *f i*)
unfolding *incseq-def*
apply *transfer*
subgoal for *A*
using *Sup-countable-SUP*[*of A*]
by (*force simp add*: *incseq-def*[*symmetric*] *SUP-upper2 image-subset-iff Sup-upper2 cong*: *conj-cong*)
done

lemma *ennreal-Inf-countable-INF*:
A ≠ {} ⟹ ∃*f*::*nat* ⇒ *ennreal*. *decseq f* ∧ *range f* ⊆ *A* ∧ *Inf A* = (*INF i*. *f i*)
unfolding *decseq-def*

apply *transfer*
subgoal for *A*
using *Inf-countable-INF*[*of A*] **by** (*simp flip*: *decseq-def*) *blast*
done

lemma *ennreal-SUP-countable-SUP*:
$A \neq \{\} \Longrightarrow \exists f{::}nat \Rightarrow ennreal.\ range\ f \subseteq g`A \wedge Sup\ (g\ `\ A) = Sup\ (f\ `\ UNIV)$
using *ennreal-Sup-countable-SUP* [*of g`A*] **by** *auto*

lemma *of-nat-tendsto-top-ennreal*: $(\lambda n{::}nat.\ of\text{-}nat\ n :: ennreal) \longrightarrow top$
using *LIMSEQ-SUP*[*of of-nat* :: *nat* ⇒ *ennreal*]
by (*simp add*: *ennreal-SUP-of-nat-eq-top incseq-def*)

lemma *SUP-sup-continuous-ennreal*:
fixes *f* :: *ennreal* ⇒ *'a*::*complete-lattice*
assumes *f*: *sup-continuous f* **and** $I \neq \{\}$
shows $(SUP\ i{\in}I.\ f\ (g\ i)) = f\ (SUP\ i{\in}I.\ g\ i)$
proof (*rule antisym*)
show $(SUP\ i{\in}I.\ f\ (g\ i)) \leq f\ (SUP\ i{\in}I.\ g\ i)$
by (*rule mono-SUP*[*OF sup-continuous-mono*[*OF f*]])
from *ennreal-Sup-countable-SUP*[*of g`I*] ‹$I \neq \{\}$›
obtain *M* :: *nat* ⇒ *ennreal* **where** *incseq M* **and** *M*: *range M* ⊆ *g* ` *I* **and** *eq*: $(SUP\ i \in I.\ g\ i) = (SUP\ i.\ M\ i)$
by *auto*
have $f\ (SUP\ i \in I.\ g\ i) = (SUP\ i \in range\ M.\ f\ i)$
unfolding *eq sup-continuousD*[*OF f* ‹*mono M*›] **by** (*simp add*: *image-comp*)
also have ... ≤ $(SUP\ i \in I.\ f\ (g\ i))$
by (*smt* (*verit*) *M SUP-le-iff dual-order.refl image-iff subsetD*)
finally show $f\ (SUP\ i \in I.\ g\ i) \leq (SUP\ i \in I.\ f\ (g\ i))$.
qed

lemma *ennreal-suminf-SUP-eq*:
fixes *f* :: *nat* ⇒ *nat* ⇒ *ennreal*
shows $(\bigwedge i.\ incseq\ (\lambda n.\ f\ n\ i)) \Longrightarrow (\sum i.\ SUP\ n.\ f\ n\ i) = (SUP\ n.\ \sum i.\ f\ n\ i)$
by (*metis ennreal-suminf-SUP-eq-directed incseqD nat-le-linear*)

lemma *ennreal-SUP-add-left*:
fixes *c* :: *ennreal*
shows $I \neq \{\} \Longrightarrow (SUP\ i{\in}I.\ f\ i + c) = (SUP\ i{\in}I.\ f\ i) + c$
apply *transfer*
apply (*simp add*: *SUP-ereal-add-left*)
by (*metis SUP-upper all-not-in-conv add-increasing2 max.absorb2 max.bounded-iff*)

lemma *ennreal-SUP-const-minus*:
fixes *f* :: *'a* ⇒ *ennreal*
shows $I \neq \{\} \Longrightarrow c < top \Longrightarrow (INF\ x{\in}I.\ c - f\ x) = c - (SUP\ x{\in}I.\ f\ x)$
apply (*transfer fixing*: *I*)
unfolding *ex-in-conv*[*symmetric*]
apply (*auto simp add*: *SUP-upper2 sup-absorb2 simp flip*: *sup-ereal-def*)

apply (*subst INF-ereal-minus-right*[*symmetric*])
apply (*auto simp del*: *sup-ereal-def simp add*: *sup-INF*)
done

lemma *isCont-ennreal*[*simp*]: ‹*isCont ennreal x*›
unfolding *continuous-within tendsto-def*
using *tendsto-ennrealI topological-tendstoD*
by (*blast intro*: *sequentially-imp-eventually-within*)

lemma *isCont-ennreal-of-enat*[*simp*]: ‹*isCont ennreal-of-enat x*›
proof −
have *continuous-at-open*:
— Copied lemma from `HOL-Analysis` to avoid dependency.
continuous (*at x*) *f* ⟷ (∀ *t*. *open t* ∧ *f x* ∈ *t* −−> (∃ *s*. *open s* ∧ *x* ∈ *s* ∧ (∀ *x'* ∈ *s*. (*f x'*) ∈ *t*))) **for** *f* :: ‹*enat* ⇒ *'z*::*topological-space*›
unfolding *continuous-within-topological* [*of x UNIV f*]
unfolding *imp-conjL*
by (*intro all-cong imp-cong ex-cong conj-cong refl*) *auto*
show *?thesis*
proof (*subst continuous-at-open*, *intro allI impI*, *cases* ‹*x* = ∞›)
case *True*

fix *t* **assume** ‹*open t* ∧ *ennreal-of-enat x* ∈ *t*›
then have ‹∃ *y*<∞. {*y* <.. ∞} ⊆ *t*›
by (*rule-tac open-left*[**where** *y*=*0*]) (*auto simp*: *True*)
then obtain *y* **where** ‹{*y*<..} ⊆ *t*› **and** ‹*y* ≠ ∞›
by *fastforce*
from ‹*y* ≠ ∞›
obtain *x'* **where** *x'y*: ‹*ennreal-of-enat x'* > *y*› **and** ‹*x'* ≠ ∞›
by (*metis enat.simps*(*3*) *ennreal-Ex-less-of-nat ennreal-of-enat-enat infinity-ennreal-def top.not-eq-extremum*)
define *s* **where** ‹*s* = {*x'*<..}›
have ‹*open s*›
by (*simp add*: *s-def*)
moreover have ‹*x* ∈ *s*›
by (*simp add*: ‹*x'* ≠ ∞› *s-def True*)
moreover have ‹*ennreal-of-enat z* ∈ *t*› **if** ‹*z* ∈ *s*› **for** *z*
by (*metis x'y* ‹{*y*<..} ⊆ *t*› *ennreal-of-enat-le-iff greaterThan-iff le-less-trans less-imp-le not-less s-def subsetD that*)
ultimately show ‹∃ *s*. *open s* ∧ *x* ∈ *s* ∧ (∀ *z*∈*s*. *ennreal-of-enat z* ∈ *t*)›
by *auto*
next
case *False*
fix *t* **assume** *asm*: ‹*open t* ∧ *ennreal-of-enat x* ∈ *t*›
define *s* **where** ‹*s* = {*x*}›
have ‹*open s*›
using *False open-enat-iff s-def* **by** *blast*

then show ‹$\exists s.\ open\ s \wedge x \in s \wedge (\forall z{\in}s.\ ennreal\text{-}of\text{-}enat\ z \in t)$›
using *asm s-def* **by** *blast*
qed
qed

40.9 Approximation lemmas

lemma *INF-approx-ennreal*:
fixes *x*::*ennreal* **and** *e*::*real*
assumes $e > 0$
assumes $x = (INF\ i \in A.\ f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A.\ f\ i < x + e$
using *INF-less-iff assms* **by** *fastforce*

lemma *SUP-approx-ennreal*:
fixes *x*::*ennreal* **and** *e*::*real*
assumes $e > 0\ A \neq \{\}$
assumes *SUP*: $x = (SUP\ i \in A.\ f\ i)$
assumes $x \neq \infty$
shows $\exists i \in A.\ x < f\ i + e$
proof −
have $x < x + e$
using ‹$0{<}e$› ‹$x \neq \infty$› **by** (*cases x*) *auto*
also have $x + e = (SUP\ i \in A.\ f\ i + e)$
unfolding *SUP ennreal-SUP-add-left*[*OF* ‹$A \neq \{\}$›] **..**
finally show *?thesis*
unfolding *less-SUP-iff* **.**
qed

lemma *ennreal-approx-SUP*:
fixes *x*::*ennreal*
assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow f\ i \leq x$
assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists i \in A.\ x \leq f\ i + e$
shows $x = (SUP\ i \in A.\ f\ i)$
proof (*rule antisym*)
show $x \leq (SUP\ i{\in}A.\ f\ i)$
proof (*rule ennreal-le-epsilon*)
fix *e* :: *real* **assume** $0 < e$
from *approx*[*OF this*] **obtain** *i* **where** $i \in A$ **and** $*$: $x \leq f\ i + ennreal\ e$
by *blast*
from $*$ **have** $x \leq f\ i + e$
by *simp*
also have $\ldots \leq (SUP\ i{\in}A.\ f\ i) + e$
by (*intro add-mono* ‹$i \in A$› *SUP-upper order-refl*)
finally show $x \leq (SUP\ i{\in}A.\ f\ i) + e$ **.**
qed
qed (*intro SUP-least f-bound*)

lemma *ennreal-approx-INF*:
fixes *x*::*ennreal*
assumes *f-bound*: $\bigwedge i.\ i \in A \Longrightarrow x \le f\ i$
assumes *approx*: $\bigwedge e.\ (e{::}real) > 0 \Longrightarrow \exists\, i \in A.\ f\ i \le x + e$
shows $x = (INF\ i \in A.\ f\ i)$
proof (*rule antisym*)
show $(INF\ i{\in}A.\ f\ i) \le x$
proof (*rule ennreal-le-epsilon*)
fix $e :: real$ **assume** $0 < e$
from *approx*[*OF this*] **obtain** i **where** $i{\in}A\ f\ i \le x + ennreal\ e$
by *blast*
then have $(INF\ i{\in}A.\ f\ i) \le f\ i$
by (*intro INF-lower*)
also have $\ldots \le x + e$
by *fact*
finally show $(INF\ i{\in}A.\ f\ i) \le x + e$ **.**
qed
qed (*intro INF-greatest f-bound*)

lemma *ennreal-approx-unit*:
$(\bigwedge a{::}ennreal.\ 0 < a \Longrightarrow a < 1 \Longrightarrow a * z \le y) \Longrightarrow z \le y$
using *SUP-mult-right-ennreal*[*of* $\lambda x.\ x\ \{0 <..< 1\}\ z$]
by (*smt* (*verit*) *SUP-least Sup-greaterThanLessThan greaterThanLessThan-iff image-ident mult-1 zero-less-one*)

lemma *suminf-ennreal2*:
$(\bigwedge i.\ 0 \le f\ i) \Longrightarrow summable\ f \Longrightarrow (\sum i.\ ennreal\ (f\ i)) = ennreal\ (\sum i.\ f\ i)$
using *suminf-ennreal-eq* **by** *blast*

lemma *less-top-ennreal*: $x < top \longleftrightarrow (\exists\, r{\ge}0.\ x = ennreal\ r)$
by (*cases x*) *auto*

lemma *enn2real-less-iff*[*simp*]: $x < top \Longrightarrow enn2real\ x < c \longleftrightarrow x < c$
using *ennreal-less-iff less-top-ennreal* **by** *auto*

lemma *enn2real-le-iff*[*simp*]: $[\![x < top;\ c > 0]\!] \Longrightarrow enn2real\ x \le c \longleftrightarrow x \le c$
by (*cases x*) *auto*

lemma *enn2real-less*:
assumes $enn2real\ e < r\ e \neq top$ **shows** $e < ennreal\ r$
using *enn2real-less-iff assms top.not-eq-extremum* **by** *blast*

lemma *enn2real-le*:
assumes $enn2real\ e \le r\ e \neq top$ **shows** $e \le ennreal\ r$
by (*metis assms enn2real-less ennreal-enn2real-if eq-iff less-le*)

lemma *tendsto-top-iff-ennreal*:
fixes $f :: {'a} \Rightarrow ennreal$
shows $(f \longrightarrow top)\ F \longleftrightarrow (\forall\, l{\ge}0.\ eventually\ (\lambda x.\ ennreal\ l < f\ x)\ F)$

by (*auto simp*: *less-top-ennreal order-tendsto-iff*)

lemma *ennreal-tendsto-top-eq-at-top*:
((λz. *ennreal* (*f z*)) $\longrightarrow$ *top*) *F* $\longleftrightarrow$ (*LIM z F*. *f z* :> *at-top*)
unfolding *filterlim-at-top-dense tendsto-top-iff-ennreal*
using *ennreal-less-iff eventually-mono allE*[*of* - *max 0* -]
by (*smt* (*verit*) *linorder-not-less order-refl order-trans*)

lemma *tendsto-0-if-Limsup-eq-0-ennreal*:
fixes *f* :: - $\Rightarrow$ *ennreal*
shows *Limsup F f* = *0* $\Longrightarrow$ (*f* $\longrightarrow$ *0*) *F*
using *Liminf-le-Limsup*[*of F f*] *tendsto-iff-Liminf-eq-Limsup*[*of F f 0*]
by (*cases F* = *bot*) *auto*

lemma *diff-le-self-ennreal*[*simp*]: *a* − *b* $\leq$ (*a*::*ennreal*)
by (*cases a b rule*: *ennreal2-cases*) (*auto simp*: *ennreal-minus*)

lemma *ennreal-ineq-diff-add*: *b* $\leq$ *a* $\Longrightarrow$ *a* = *b* + (*a* − *b*::*ennreal*)
by *transfer* (*auto simp*: *ereal-diff-positive max.absorb2 ereal-ineq-diff-add*)

lemma *ennreal-mult-strict-left-mono*: (*a*::*ennreal*) < *c* $\Longrightarrow$ *0* < *b* $\Longrightarrow$ *b* < *top* $\Longrightarrow$ *b* ∗ *a* < *b* ∗ *c*
by *transfer* (*auto intro*!: *ereal-mult-strict-left-mono*)

lemma *ennreal-between*: *0* < *e* $\Longrightarrow$ *0* < *x* $\Longrightarrow$ *x* < *top* $\Longrightarrow$ *x* − *e* < (*x*::*ennreal*)
by *transfer* (*auto intro*!: *ereal-between*)

lemma *minus-less-iff-ennreal*: *b* < *top* $\Longrightarrow$ *b* $\leq$ *a* $\Longrightarrow$ *a* − *b* < *c* $\longleftrightarrow$ *a* < *c* + (*b*::*ennreal*)
by *transfer*
(*auto simp*: *top-ereal-def ereal-minus-less le-less*)

lemma *tendsto-zero-ennreal*:
assumes *ev*: $\bigwedge r$. *0* < *r* $\Longrightarrow$ $\forall_F$ *x in F*. *f x* < *ennreal r*
shows (*f* $\longrightarrow$ *0*) *F*
proof (*rule order-tendstoI*)
fix *e*::*ennreal* **assume** *e* > *0*
obtain *e'*::*real* **where** *e'* > *0 ennreal e'* < *e*
using ‹*0* < *e*› *dense*[*of 0 if e* = *top then 1 else* (*enn2real e*)]
by (*cases e*) (*auto simp*: *ennreal-less-iff*)
from *ev*[*OF* ‹*e'* > *0*›] **show** $\forall_F$ *x in F*. *f x* < *e*
by *eventually-elim* (*insert* ‹*ennreal e'* < *e*›, *auto*)
qed *simp*

lifting-update *ennreal.lifting*
lifting-forget *ennreal.lifting*

40.10 *ennreal* theorems

lemma *neq-top-trans*: **fixes** $x\ y :: ennreal$ **shows** $[\![y \neq top; x \leq y]\!] \Longrightarrow x \neq top$
by (*auto simp*: *top-unique*)

lemma *diff-diff-ennreal*: **fixes** $a\ b :: ennreal$ **shows** $a \leq b \Longrightarrow b \neq \infty \Longrightarrow b - (b - a) = a$
by (*cases a b rule*: *ennreal2-cases*) (*auto simp*: *ennreal-minus top-unique*)

lemma *ennreal-less-one-iff*[*simp*]: $ennreal\ x < 1 \longleftrightarrow x < 1$
by (*cases* $0 \leq x$) (*auto simp*: *ennreal-neg ennreal-less-iff simp flip*: *ennreal-1*)

lemma *SUP-const-minus-ennreal*:
fixes $f :: 'a \Rightarrow ennreal$ **shows** $I \neq \{\} \Longrightarrow (SUP\ x{\in}I.\ c - f\ x) = c - (INF\ x{\in}I.\ f\ x)$
including *ennreal.lifting*
by (*transfer fixing*: *I*)
(*simp add*: *SUP-sup-distrib*[*symmetric*] *SUP-ereal-minus-right*
flip: *sup-ereal-def*)

lemma *zero-minus-ennreal*[*simp*]: $0 - (a::ennreal) = 0$
including *ennreal.lifting*
by *transfer* (*simp split*: *split-max*)

lemma *diff-diff-commute-ennreal*:
fixes $a\ b\ c :: ennreal$ **shows** $a - b - c = a - c - b$
by (*cases a b c rule*: *ennreal3-cases*) (*simp-all add*: *ennreal-minus field-simps*)

lemma *diff-gr0-ennreal*: $b < (a::ennreal) \Longrightarrow 0 < a - b$
including *ennreal.lifting* **by** *transfer* (*auto simp*: *ereal-diff-gr0 ereal-diff-positive split*: *split-max*)

lemma *divide-le-posI-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $x > 0 \Longrightarrow z \leq x * y \Longrightarrow z\ /\ x \leq y$
by (*cases x y z rule*: *ennreal3-cases*)
(*auto simp*: *divide-ennreal ennreal-mult*[*symmetric*] *field-simps top-unique*)

lemma *add-diff-eq-ennreal*:
fixes $x\ y\ z :: ennreal$
shows $z \leq y \Longrightarrow x + (y - z) = x + y - z$
using *ennreal-diff-add-assoc* **by** *auto*

lemma *add-diff-inverse-ennreal*:
fixes $x\ y :: ennreal$ **shows** $x \leq y \Longrightarrow x + (y - x) = y$
by (*cases x*) (*simp-all add*: *top-unique add-diff-eq-ennreal*)

lemma *add-diff-eq-iff-ennreal*[*simp*]:
fixes $x\ y :: ennreal$ **shows** $x + (y - x) = y \longleftrightarrow x \leq y$
by (*metis ennreal-ineq-diff-add le-iff-add*)

lemma *add-diff-le-ennreal*: $a + b - c \leq a + (b - c::ennreal)$
apply (*cases a b c rule*: *ennreal3-cases*)
subgoal for a' b' c'
by (*cases* $0 \leq b' - c'$) (*simp-all add*: *ennreal-minus top-add ennreal-neg flip*: *ennreal-plus*)
apply (*simp-all add*: *top-add flip*: *ennreal-plus*)
done

lemma *diff-eq-0-ennreal*: $a < top \Longrightarrow a \leq b \Longrightarrow a - b = (0::ennreal)$
using *ennreal-minus-pos-iff gr-zeroI not-less* **by** *blast*

lemma *diff-diff-ennreal'*: **fixes** x y $z :: ennreal$ **shows** $z \leq y \Longrightarrow y - z \leq x \Longrightarrow x - (y - z) = x + z - y$
by (*cases x*; *cases y*; *cases z*)
(*auto simp add*: *top-add add-top minus-top-ennreal ennreal-minus top-unique simp flip*: *ennreal-plus*)

lemma *diff-diff-ennreal''*: **fixes** x y $z :: ennreal$
shows $z \leq y \Longrightarrow x - (y - z) = (if\ y - z \leq x\ then\ x + z - y\ else\ 0)$
by (*cases x*; *cases y*; *cases z*)
(*auto simp add*: *top-add add-top minus-top-ennreal ennreal-minus top-unique ennreal-neg simp flip*: *ennreal-plus*)

lemma *power-less-top-ennreal*: **fixes** $x :: ennreal$ **shows** $x \hat{} n < top \longleftrightarrow x < top \vee n = 0$
using *power-eq-top-ennreal top.not-eq-extremum*
by *blast*

lemma *ennreal-divide-times*: $(a / b) * c = a * (c / b :: ennreal)$
by (*simp add*: *mult.commute ennreal-times-divide*)

lemma *diff-less-top-ennreal*: $a - b < top \longleftrightarrow a < (top :: ennreal)$
by (*cases a*; *cases b*) (*auto simp*: *ennreal-minus*)

lemma *divide-less-ennreal*: $b \neq 0 \Longrightarrow b < top \Longrightarrow a / b < c \longleftrightarrow a < (c * b :: ennreal)$
by (*cases a*; *cases b*; *cases c*)
(*auto simp*: *divide-ennreal ennreal-mult*[*symmetric*] *ennreal-less-iff field-simps ennreal-top-mult ennreal-top-divide*)

lemma *one-less-numeral*[*simp*]: $1 < (numeral\ n::ennreal) \longleftrightarrow (num.One < n)$
by *simp*

lemma *divide-eq-1-ennreal*: $a / b = (1::ennreal) \longleftrightarrow (b \neq top \wedge b \neq 0 \wedge b = a)$
by (*cases a* ; *cases b*; *cases b = 0*) (*auto simp*: *ennreal-top-divide divide-ennreal split*: *if-split-asm*)

lemma *ennreal-mult-cancel-left*: $(a * b = a * c) = (a = top \wedge b \neq 0 \wedge c \neq 0 \vee a = 0 \vee b = (c::ennreal))$
by (*cases a*; *cases b*; *cases c*) (*auto simp*: *ennreal-mult*[*symmetric*] *ennreal-mult-top ennreal-top-mult*)

lemma *ennreal-minus-if*: *ennreal a* − *ennreal b* = *ennreal* (*if* $0 \leq b$ *then* (*if* $b \leq a$ *then* $a - b$ *else* 0) *else a*)
by (*auto simp*: *ennreal-minus ennreal-neg*)

lemma *ennreal-plus-if*: *ennreal a* + *ennreal b* = *ennreal* (*if* $0 \leq a$ *then* (*if* $0 \leq b$ *then* $a + b$ *else a*) *else b*)
by (*auto simp*: *ennreal-neg*)

lemma *ennreal-diff-le-mono-left*: $a \leq b \Longrightarrow a - c \leq (b::ennreal)$
using *ennreal-mono-minus*[*of* 0 *c a*, *THEN order-trans*, *of b*] **by** *simp*

lemma *ennreal-minus-le-iff*: $a - b \leq c \longleftrightarrow (a \leq b + (c::ennreal) \wedge (a = top \wedge b = top \longrightarrow c = top))$
by (*cases a*; *cases b*; *cases c*)
(*auto simp*: *top-unique top-add add-top ennreal-minus simp flip*: *ennreal-plus*)

lemma *ennreal-le-minus-iff*: $a \leq b - c \longleftrightarrow (a + c \leq (b::ennreal) \vee (a = 0 \wedge b \leq c))$
by (*cases a*; *cases b*; *cases c*)
(*auto simp*: *top-unique top-add add-top ennreal-minus ennreal-le-iff2*
simp flip: *ennreal-plus*)

lemma *diff-add-eq-diff-diff-swap-ennreal*: $x - (y + z :: ennreal) = x - y - z$
by (*cases x*; *cases y*; *cases z*)
(*auto simp*: *ennreal-minus-if add-top top-add simp flip*: *ennreal-plus*)

lemma *diff-add-assoc2-ennreal*: $b \leq a \Longrightarrow (a - b + c::ennreal) = a + c - b$
by (*cases a*; *cases b*; *cases c*)
(*auto simp add*: *ennreal-minus-if ennreal-plus-if add-top top-add top-unique simp del*: *ennreal-plus*)

lemma *diff-gt-0-iff-gt-ennreal*: $0 < a - b \longleftrightarrow (a = top \wedge b = top \vee b < (a::ennreal))$
by (*cases a*; *cases b*) (*auto simp*: *ennreal-minus-if ennreal-less-iff*)

lemma *diff-eq-0-iff-ennreal*: $(a - b::ennreal) = 0 \longleftrightarrow (a < top \wedge a \leq b)$
by (*cases a*) (*auto simp*: *ennreal-minus-eq-0 diff-eq-0-ennreal*)

lemma *add-diff-self-ennreal*: $a + (b - a::ennreal) = (if\ a \leq b\ then\ b\ else\ a)$
by (*auto simp*: *diff-eq-0-iff-ennreal less-top*)

lemma *diff-add-self-ennreal*: $(b - a + a::ennreal) = (if\ a \leq b\ then\ b\ else\ a)$
by (*auto simp*: *diff-add-cancel-ennreal diff-eq-0-iff-ennreal less-top*)

lemma *ennreal-minus-cancel-iff*:

fixes *a b c* :: *ennreal*
shows $a - b = a - c \longleftrightarrow (b = c \vee (a \leq b \wedge a \leq c) \vee a = top)$
by (*cases a*; *cases b*; *cases c*) (*auto simp*: *ennreal-minus-if*)

The next lemma is wrong for $a = top$, for $b = c = 1$ for instance.

lemma *ennreal-right-diff-distrib*:
fixes *a b c* :: *ennreal*
assumes $a \neq top$
shows $a * (b - c) = a * b - a * c$
apply (*cases a*; *cases b*; *cases c*)
apply (*use assms* **in** ‹*auto simp add*: *ennreal-mult-top ennreal-minus ennreal-mult'* [*symmetric*]›)
apply (*simp add*: *algebra-simps*)
done

lemma *SUP-diff-ennreal*:
$c < top \Longrightarrow (SUP\ i{\in}I.\ f\ i - c :: ennreal) = (SUP\ i{\in}I.\ f\ i) - c$
by (*auto intro*!: *SUP-eqI ennreal-minus-mono SUP-least intro*: *SUP-upper simp*: *ennreal-minus-cancel-iff ennreal-minus-le-iff less-top*[*symmetric*])

lemma *ennreal-SUP-add-right*:
fixes *c* :: *ennreal* **shows** $I \neq \{\} \Longrightarrow c + (SUP\ i{\in}I.\ f\ i) = (SUP\ i{\in}I.\ c + f\ i)$
using *ennreal-SUP-add-left*[*of I f c*] **by** (*simp add*: *add.commute*)

lemma *SUP-add-directed-ennreal*:
fixes *f g* :: - ⇒ *ennreal*
assumes *directed*: $\bigwedge i\ j.\ i \in I \Longrightarrow j \in I \Longrightarrow \exists k{\in}I.\ f\ i + g\ j \leq f\ k + g\ k$
shows $(SUP\ i{\in}I.\ f\ i + g\ i) = (SUP\ i{\in}I.\ f\ i) + (SUP\ i{\in}I.\ g\ i)$
proof (*cases* $I = \{\}$)
case *False*
show *?thesis*
proof (*rule antisym*)
show $(SUP\ i{\in}I.\ f\ i + g\ i) \leq (SUP\ i{\in}I.\ f\ i) + (SUP\ i{\in}I.\ g\ i)$
by (*rule SUP-least*; *intro add-mono SUP-upper*)
next
have $(SUP\ i{\in}I.\ f\ i) + (SUP\ i{\in}I.\ g\ i) = (SUP\ i{\in}I.\ f\ i + (SUP\ i{\in}I.\ g\ i))$
by (*intro ennreal-SUP-add-left*[*symmetric*] ‹$I \neq \{\}$›)
also have $\ldots = (SUP\ i{\in}I.\ (SUP\ j{\in}I.\ f\ i + g\ j))$
using ‹$I \neq \{\}$› **by** (*simp add*: *ennreal-SUP-add-right*)
also have $\ldots \leq (SUP\ i{\in}I.\ f\ i + g\ i)$
using *directed* **by** (*intro SUP-least*) (*blast intro*: *SUP-upper2*)
finally show $(SUP\ i{\in}I.\ f\ i) + (SUP\ i{\in}I.\ g\ i) \leq (SUP\ i{\in}I.\ f\ i + g\ i)$.
qed
qed (*simp add*: *bot-ereal-def*)

lemma *enn2real-eq-0-iff*: $enn2real\ x = 0 \longleftrightarrow x = 0 \vee x = top$
by (*cases x*) *auto*

lemma *continuous-on-diff-ennreal*:
continuous-on A f $\Longrightarrow$ *continuous-on A g* $\Longrightarrow$ $(\bigwedge x.\ x \in A \Longrightarrow f\ x \neq top) \Longrightarrow$ $(\bigwedge x.\ x \in A \Longrightarrow g\ x \neq top) \Longrightarrow$ *continuous-on A* $(\lambda z.\ f\ z - g\ z :: ennreal)$
including *ennreal.lifting*
proof (*transfer fixing*: *A*, *simp add*: *top-ereal-def*)
fix *f g* :: $'a \Rightarrow ereal$ **assume** $\forall x.\ 0 \leq f\ x\ \forall x.\ 0 \leq g\ x$ *continuous-on A f continuous-on A g*
moreover assume $f\ x \neq \infty\ g\ x \neq \infty$ **if** $x \in A$ **for** *x*
ultimately show *continuous-on A* $(\lambda z.\ max\ 0\ (f\ z - g\ z))$
by (*intro continuous-on-max continuous-on-const continuous-on-diff-ereal*) *auto*
qed

lemma *tendsto-diff-ennreal*:
$(f \longrightarrow x)\ F \Longrightarrow (g \longrightarrow y)\ F \Longrightarrow x \neq top \Longrightarrow y \neq top \Longrightarrow ((\lambda z.\ f\ z - g\ z :: ennreal) \longrightarrow x - y)\ F$
using *continuous-on-tendsto-compose*[**where** $f{=}\lambda x.\ fst\ x - snd\ x :: ennreal$ **and** $s{=}\{(x,\ y).\ x \neq top \wedge y \neq top\}$ **and** $g{=}\lambda x.\ (f\ x,\ g\ x)$ **and** $l{=}(x,\ y)$ **and** $F{=}F$,
OF continuous-on-diff-ennreal]
by (*auto simp*: *tendsto-Pair eventually-conj-iff less-top order-tendstoD continuous-on-fst continuous-on-snd continuous-on-id*)

declare *lim-real-of-ereal* [*tendsto-intros*]

lemma *tendsto-enn2real* [*tendsto-intros*]:
assumes $(u \longrightarrow ennreal\ l)\ F\ l \geq 0$
shows $((\lambda n.\ enn2real\ (u\ n)) \longrightarrow l)\ F$
unfolding *enn2real-def*
by (*metis assms enn2ereal-ennreal lim-real-of-ereal tendsto-enn2erealI*)

end

41 Logarithm of Natural Numbers

theory *Log-Nat*
imports *Complex-Main*
begin

41.1 Preliminaries

lemma *divide-nat-diff-div-nat-less-one*:
$real\ x\ /\ real\ b - real\ (x\ div\ b) < 1$ **for** *x b* :: *nat*
proof (*cases* $b = 0$)
case *True*
then show *?thesis*
by *simp*
next
case *False*
then have $real\ (x\ div\ b) + real\ (x\ mod\ b)\ /\ real\ b - real\ (x\ div\ b) < 1$
by (*simp add*: *field-simps*)

then show *?thesis*
by (*metis of-nat-of-nat-div-aux*)
qed

41.2 Floorlog

definition *floorlog* :: *nat* ⇒ *nat* ⇒ *nat*
where *floorlog b a* = (*if a* > *0* ∧ *b* > *1 then nat* ⌊*log b a*⌋ + *1 else 0*)

lemma *floorlog-mono*: *x* ≤ *y* ⟹ *floorlog b x* ≤ *floorlog b y*
by (*auto simp*: *floorlog-def floor-mono nat-mono*)

lemma *floorlog-bounds*:
b ^ (*floorlog b x* − *1*) ≤ *x* ∧ *x* < *b* ^ (*floorlog b x*) **if** *x* > *0 b* > *1*
proof
show *b* ^ (*floorlog b x* − *1*) ≤ *x*
proof −
have *b* ^ *nat* ⌊*log b x*⌋ = *b powr* ⌊*log b x*⌋
using *powr-realpow*[*symmetric*, *of b nat* ⌊*log b x*⌋] ‹*x* > *0*› ‹*b* > *1*›
by *simp*
also have ... ≤ *b powr log b x* **using** ‹*b* > *1*› **by** *simp*
also have ... = *real-of-int x* **using** ‹*0* < *x*› ‹*b* > *1*› **by** *simp*
finally have *b* ^ *nat* ⌊*log b x*⌋ ≤ *real-of-int x* **by** *simp*
then show *?thesis*
using ‹*0* < *x*› ‹*b* > *1*› *of-nat-le-iff*
by (*fastforce simp add*: *floorlog-def*)
qed
show *x* < *b* ^ (*floorlog b x*)
proof −
have *x* ≤ *b powr* (*log b x*) **using** ‹*x* > *0*› ‹*b* > *1*› **by** *simp*
also have ... < *b powr* (⌊*log b x*⌋ + *1*)
using *that* **by** (*intro powr-less-mono*) *auto*
also have ... = *b* ^ *nat* (⌊*log b* (*real-of-int x*)⌋ + *1*)
using *that* **by** (*simp flip*: *powr-realpow*)
finally
have *x* < *b* ^ *nat* (⌊*log b* (*int x*)⌋ + *1*)
by (*rule of-nat-less-imp-less*)
then show *?thesis*
using ‹*x* > *0*› ‹*b* > *1*› **by** (*simp add*: *floorlog-def nat-add-distrib*)
qed
qed

lemma *floorlog-power* [*simp*]:
floorlog b (*a* ∗ *b* ^ *c*) = *floorlog b a* + *c* **if** *a* > *0 b* > *1*
proof −
have ⌊*log b a* + *real c*⌋ = ⌊*log b a*⌋ + *c* **by** *arith*
then show *?thesis* **using** *that*
by (*auto simp*: *floorlog-def log-mult powr-realpow*[*symmetric*] *nat-add-distrib*)
qed

lemma *floor-log-add-eqI*:
  $\lfloor log\ b\ (a + r)\rfloor = \lfloor log\ b\ a\rfloor$ **if** $b > 1\ a \geq 1\ 0 \leq r\ r < 1$
    for $a\ b :: nat$ **and** $r :: real$
proof (*rule floor-eq2*)
  have $log\ b\ a \leq log\ b\ (a + r)$ **using** *that* **by** *force*
  then show $\lfloor log\ b\ a\rfloor \leq log\ b\ (a + r)$ **by** *arith*
next
  define l::*int* **where** $l = int\ b\ \hat{}\ (nat\ \lfloor log\ b\ a\rfloor + 1)$
  have *l-def-real*: $l = b\ powr\ (\lfloor log\ b\ a\rfloor + 1)$
    using *that* **by** (*simp add*: *l-def powr-add powr-real-of-int*)
  have $a < l$
  proof −
    have $a = b\ powr\ (log\ b\ a)$ **using** *that* **by** *simp*
    also have $\ldots < b\ powr\ floor\ ((log\ b\ a) + 1)$
      using *that*(*1*) **by** *auto*
    also have $\ldots = l$
      using *that* **by** (*simp add*: *l-def powr-real-of-int powr-add*)
    finally show *?thesis* **by** *simp*
  qed
  then have $a + r < l$ **using** *that* **by** *simp*
  then have $log\ b\ (a + r) < log\ b\ l$ **using** *that* **by** *simp*
  also have $\ldots = real\text{-}of\text{-}int\ \lfloor log\ b\ a\rfloor + 1$
    using *that* **by** (*simp add*: *l-def-real*)
  finally show $log\ b\ (a + r) < real\text{-}of\text{-}int\ \lfloor log\ b\ a\rfloor + 1$ **.**
qed

lemma *floor-log-div*:
  $\lfloor log\ b\ x\rfloor = \lfloor log\ b\ (x\ div\ b)\rfloor + 1$ **if** $b > 1\ x > 0\ x\ div\ b > 0$
    for $b\ x :: nat$
proof−
  have $\lfloor log\ b\ x\rfloor = \lfloor log\ b\ (x\ /\ b * b)\rfloor$ **using** *that* **by** *simp*
  also have $\ldots = \lfloor log\ b\ (x\ /\ b) + log\ b\ b\rfloor$
    using *that* **by** (*subst log-mult*) *auto*
  also have $\ldots = \lfloor log\ b\ (x\ /\ b)\rfloor + 1$ **using** *that* **by** *simp*
  also have $\lfloor log\ b\ (x\ /\ b)\rfloor = \lfloor log\ b\ (x\ div\ b + (x\ /\ b - x\ div\ b))\rfloor$ **by** *simp*
  also have $\ldots = \lfloor log\ b\ (x\ div\ b)\rfloor$
    using *that of-nat-div-le-of-nat divide-nat-diff-div-nat-less-one*
    by (*intro floor-log-add-eqI*) *auto*
  finally show *?thesis* **.**
qed

lemma *compute-floorlog* [*code*]:
  floorlog b x = (*if x* > *0* ∧ *b* > *1 then floorlog b* (*x div b*) + *1 else 0*)
  by (*auto simp*: *floorlog-def floor-log-div*[*of b x*] *div-eq-0-iff nat-add-distrib*
    intro!: *floor-eq2*)

lemma *floor-log-eq-if*:
  $\lfloor log\ b\ x\rfloor = \lfloor log\ b\ y\rfloor$ **if** $x\ div\ b = y\ div\ b\ b > 1\ x > 0\ x\ div\ b \geq 1$

for *b x y* :: *nat*
proof −
have *y* > *0* **using** *that* **by** (*auto intro*: *ccontr*)
thus *?thesis* **using** *that* **by** (*simp add*: *floor-log-div*)
qed

lemma *floorlog-eq-if*:
floorlog b x = *floorlog b y* **if** *x div b* = *y div b b* > *1 x* > *0 x div b* ≥ *1*
for *b x y* :: *nat*
proof −
have *y* > *0* **using** *that* **by** (*auto intro*: *ccontr*)
then show *?thesis* **using** *that*
by (*auto simp add*: *floorlog-def eq-nat-nat-iff intro*: *floor-log-eq-if*)
qed

lemma *floorlog-leD*:
floorlog b x ≤ *w* ⟹ *b* > *1* ⟹ *x* < *b* ^ *w*
by (*metis floorlog-bounds leD linorder-neqE-nat order.strict-trans power-strict-increasing-iff zero-less-one zero-less-power*)

lemma *floorlog-leI*:
x < *b* ^ *w* ⟹ *0* ≤ *w* ⟹ *b* > *1* ⟹ *floorlog b x* ≤ *w*
by (*drule less-imp-of-nat-less*[**where** *'a*=*real*])
(*auto simp*: *floorlog-def Suc-le-eq nat-less-iff floor-less-iff log-of-power-less*)

lemma *floorlog-eq-zero-iff*:
floorlog b x = *0* ⟷ *b* ≤ *1* ∨ *x* ≤ *0*
by (*auto simp*: *floorlog-def*)

lemma *floorlog-le-iff*:
floorlog b x ≤ *w* ⟷ *b* ≤ *1* ∨ *b* > *1* ∧ *0* ≤ *w* ∧ *x* < *b* ^ *w*
using *floorlog-leD*[*of b x w*] *floorlog-leI*[*of x b w*]
by (*auto simp*: *floorlog-eq-zero-iff*[*THEN iffD2*])

lemma *floorlog-ge-SucI*:
Suc w ≤ *floorlog b x* **if** *b* ^ *w* ≤ *x b* > *1*
using *that le-log-of-power*[*of b w x*] *power-not-zero*
by (*force simp*: *floorlog-def Suc-le-eq powr-realpow not-less Suc-nat-eq-nat-zadd1 zless-nat-eq-int-zless int-add-floor less-floor-iff*
simp del: *floor-add2*)

lemma *floorlog-geI*:
w ≤ *floorlog b x* **if** *b* ^ (*w* − *1*) ≤ *x b* > *1*
using *floorlog-ge-SucI*[*of b w* − *1 x*] *that*
by *auto*

lemma *floorlog-geD*:
b ^ (*w* − *1*) ≤ *x* **if** *w* ≤ *floorlog b x w* > *0*
proof −

```
  have b > 1 0 < x
    using that by (auto simp: floorlog-def split: if-splits)
  have b ^ (w − 1) ≤ x if b ^ w ≤ x
  proof −
    have b ^ (w − 1) ≤ b ^ w
      using ‹b > 1›
      by (auto intro!: power-increasing)
    also note that
    finally show ?thesis .
  qed
  moreover have b ^ nat ⌊log (real b) (real x)⌋ ≤ x (is ?l ≤ -)
  proof −
    have 0 ≤ log (real b) (real x)
      using ‹b > 1› ‹0 < x›
      by auto
    then have ?l ≤ b powr log (real b) (real x)
      using ‹b > 1›
      by (auto simp flip: powr-realpow intro!: powr-mono of-nat-floor)
    also have ... = x using ‹b > 1› ‹0 < x›
      by auto
    finally show ?thesis
      unfolding of-nat-le-iff .
  qed
  ultimately show ?thesis
    using that
    by (auto simp: floorlog-def le-nat-iff le-floor-iff le-log-iff powr-realpow
        split: if-splits elim!: le-SucE)
qed
```

41.3

```
definition ceillog2 :: nat ⇒ nat where
  ceillog2 n = (if n = 0 then 0 else nat ⌈log 2 (real n)⌉)

lemma ceillog2-0 [simp]: ceillog2 0 = 0
  and ceillog2-Suc-0 [simp]: ceillog2 (Suc 0) = 0
  and ceillog2-2 [simp]: ceillog2 2 = 1
  by (auto simp: ceillog2-def)

lemma ceillog2-le1-eq-0 [simp]: n ≤ 1 ⟹ ceillog2 n = 0
  by (cases n) auto

lemma ceillog2-2-power [simp]: ceillog2 (2 ^ n) = n
  by (auto simp: ceillog2-def)

lemma ceillog2-ge-log:
  assumes n > 0
  shows   real (ceillog2 n) ≥ log 2 (real n)
proof −
```

```
  have real-of-int ⌈log 2 (real n)⌉ ≥ log 2 (real n)
    by linarith
  thus ?thesis
    using assms unfolding ceillog2-def by auto
qed

lemma ceillog2-less-log:
  assumes n > 0
  shows   real (ceillog2 n) < log 2 (real n) + 1
proof −
  have real-of-int ⌈log 2 (real n)⌉ < log 2 (real n) + 1
    by linarith
  thus ?thesis
    using assms unfolding ceillog2-def by auto
qed

lemma ceillog2-le-iff:
  assumes n > 0
  shows   ceillog2 n ≤ l ⟷ n ≤ 2 ^ l
proof −
  have ceillog2 n ≤ l ⟷ real n ≤ 2 ^ l
    unfolding ceillog2-def using assms by (auto simp: log-le-iff powr-realpow)
  also have 2 ^ l = real (2 ^ l)
    by simp
  also have real n ≤ real (2 ^ l) ⟷ n ≤ 2 ^ l
    by linarith
  finally show ?thesis .
qed

lemma ceillog2-ge-iff:
  assumes n > 0
  shows   ceillog2 n ≥ l ⟷ 2 ^ l < 2 ∗ n
proof −
  have −1 < (0 :: real)
    by auto
  also have ... ≤ log 2 (real n)
    using assms by auto
  finally have ceillog2 n ≥ l ⟷ real l − 1 < log 2 (real n)
    unfolding ceillog2-def using assms by (auto simp: le-nat-iff le-ceiling-iff)
  also have ... ⟷ real l < log 2 (real (2 ∗ n))
    using assms by (auto simp: log-mult)
  also have ... ⟷ 2 ^ l < real (2 ∗ n)
    using assms by (subst less-log-iff) (auto simp: powr-realpow)
  also have 2 ^ l = real (2 ^ l)
    by simp
  also have real (2 ^ l) < real (2 ∗ n) ⟷ 2 ^ l < 2 ∗ n
    by linarith
  finally show ?thesis .
qed
```

lemma *le-two-power-ceillog2*: $n \leq 2 \mathbin{\hat{}} \mathit{ceillog2}\ n$
using *neq0-conv ceillog2-le-iff* **by** *blast*

lemma *two-power-ceillog2-gt*:
assumes $n > 0$
shows $2 * n > 2 \mathbin{\hat{}} \mathit{ceillog2}\ n$
using *ceillog2-ge-iff*[*of n ceillog2 n*] *assms* **by** *simp*

lemma *ceillog2-eqI*:
assumes $n \leq 2 \mathbin{\hat{}} l\ 2 \mathbin{\hat{}} l < 2 * n$
shows $\mathit{ceillog2}\ n = l$
by (*metis Suc-leI assms bot-nat-0.not-eq-extremum ceillog2-ge-iff ceillog2-le-iff le-antisym mult-is-0*
not-less-eq-eq)

lemma *ceillog2-rec-even*:
assumes $k > 0$
shows $\mathit{ceillog2}\ (2 * k) = \mathit{Suc}\ (\mathit{ceillog2}\ k)$
by (*rule ceillog2-eqI*) (*auto simp*: *le-two-power-ceillog2 two-power-ceillog2-gt assms*)

lemma *ceillog2-mono*:
assumes $m \leq n$
shows $\mathit{ceillog2}\ m \leq \mathit{ceillog2}\ n$
proof (*cases* $m = 0$)
case *False*
have $\lceil \mathit{log}\ 2\ (\mathit{real}\ m) \rceil \leq \lceil \mathit{log}\ 2\ (\mathit{real}\ n) \rceil$
by (*intro ceiling-mono*) (*use False assms* **in** *auto*)
hence $\mathit{nat} \lceil \mathit{log}\ 2\ (\mathit{real}\ m) \rceil \leq \mathit{nat} \lceil \mathit{log}\ 2\ (\mathit{real}\ n) \rceil$
by *linarith*
thus *?thesis* **using** *False assms*
unfolding *ceillog2-def* **by** *simp*
qed *auto*

lemma *ceillog2-rec-odd*:
assumes $k > 0$
shows $\mathit{ceillog2}\ (\mathit{Suc}\ (2 * k)) = \mathit{Suc}\ (\mathit{ceillog2}\ (\mathit{Suc}\ k))$
proof −
have $2 \mathbin{\hat{}} \mathit{ceillog2}\ (\mathit{Suc}\ (2 * k)) > \mathit{Suc}\ (2 * k)$
by (*metis assms diff-Suc-1 dvd-triv-left le-two-power-ceillog2 mult-pos-pos nat-power-eq-Suc-0-iff*
order-less-le pos2 semiring-parity-class.even-mask-iff)
then have $\mathit{ceillog2}\ (2 * k + 2) \leq \mathit{ceillog2}\ (2 * k + 1)$
by (*simp add*: *ceillog2-le-iff*)
moreover have $\mathit{ceillog2}\ (2 * k + 2) \geq \mathit{ceillog2}\ (2 * k + 1)$
by (*rule ceillog2-mono*) *auto*
ultimately have $\mathit{ceillog2}\ (2 * k + 2) = \mathit{ceillog2}\ (2 * k + 1)$
by (*rule antisym*)
also have $2 * k + 2 = 2 * \mathit{Suc}\ k$

```
    by simp
  also have ceillog2 (2 * Suc k) = Suc (ceillog2 (Suc k))
    by (rule ceillog2-rec-even) auto
  finally show ?thesis
    by simp
qed


lemma ceillog2-rec:
  ceillog2 n = (if n ≤ 1 then 0 else 1 + ceillog2 ((n + 1) div 2))
proof (cases n ≤ 1)
  case True
  thus ?thesis
    by (cases n) auto
next
  case False
  thus ?thesis
   by (cases even n) (auto elim!: evenE oddE simp: ceillog2-rec-even ceillog2-rec-odd)
qed

lemma funpow-div2-ceillog2-le-1:
  ((λn. (n + 1) div 2) ^^ ceillog2 n) n ≤ 1
proof (induction n rule: less-induct)
  case (less n)
  show ?case
  proof (cases n ≤ 1)
    case True
    thus ?thesis by (cases n) auto
  next
    case False
    have ((λn. (n + 1) div 2) ^^ Suc (ceillog2 ((n + 1) div 2))) n ≤ 1
      using less.IH[of (n+1) div 2] False by (subst funpow-Suc-right) auto
    also have Suc (ceillog2 ((n + 1) div 2)) = ceillog2 n
      using False by (subst ceillog2-rec[of n]) auto
    finally show ?thesis .
  qed
qed


fun ceillog2-aux :: nat ⇒ nat ⇒ nat where
  ceillog2-aux acc n = (if n ≤ 1 then acc else ceillog2-aux (acc + 1) ((n + 1) div
2))

lemmas [simp del] = ceillog2-aux.simps

lemma ceillog2-aux-correct: ceillog2-aux acc n = ceillog2 n + acc
proof (induction acc n rule: ceillog2-aux.induct)
  case (1 acc n)
  show ?case
```

proof (*cases* $n \leq 1$)
case *False*
thus *?thesis* **using** *ceillog2-rec*[*of n*] *1.IH*
by (*auto simp*: *ceillog2-aux.simps*[*of acc n*])
qed (*auto simp*: *ceillog2-aux.simps*[*of acc n*])
qed

lemma *ceillog2-code* [*code*]: *ceillog2 n* = *ceillog2-aux 0 n*
by (*simp add*: *ceillog2-aux-correct*)

41.4 Bitlen

definition *bitlen* :: *int* ⇒ *int*
where *bitlen a* = *floorlog 2* (*nat a*)

lemma *bitlen-alt-def*:
bitlen a = (*if a* > *0 then* ⌊*log 2 a*⌋ + *1 else 0*)
by (*simp add*: *bitlen-def floorlog-def*)

lemma *bitlen-zero* [*simp*]:
bitlen 0 = *0*
by (*auto simp*: *bitlen-def floorlog-def*)

lemma *bitlen-nonneg*:
0 ≤ *bitlen x*
by (*simp add*: *bitlen-def*)

lemma *bitlen-bounds*:
2 ^ *nat* (*bitlen x* − *1*) ≤ *x* ∧ *x* < *2* ^ *nat* (*bitlen x*) **if** *x* > *0*
proof −
from *that* **have** *bitlen x* ≥ *1* **by** (*auto simp*: *bitlen-alt-def*)
with *that floorlog-bounds*[*of nat x 2*] **show** *?thesis*
by (*auto simp add*: *bitlen-def le-nat-iff nat-less-iff nat-diff-distrib*)
qed

lemma *bitlen-pow2* [*simp*]:
bitlen (*b* ∗ *2* ^ *c*) = *bitlen b* + *c* **if** *b* > *0*
using *that* **by** (*simp add*: *bitlen-def nat-mult-distrib nat-power-eq*)

lemma *compute-bitlen* [*code*]:
bitlen x = (*if x* > *0 then bitlen* (*x div 2*) + *1 else 0*)
by (*simp add*: *bitlen-def nat-div-distrib compute-floorlog*)

lemma *bitlen-eq-zero-iff*:
bitlen x = *0* ⟷ *x* ≤ *0*
by (*auto simp add*: *bitlen-alt-def*)
(*metis compute-bitlen add.commute bitlen-alt-def bitlen-nonneg less-add-same-cancel2*
not-less zero-less-one)

lemma *bitlen-div*:
1 ≤ real-of-int m / 2^nat (bitlen m − 1)
and *real-of-int m / 2^nat (bitlen m − 1) < 2* **if** *0 < m*
proof −
let *?B = 2^nat (bitlen m − 1)*

have *?B ≤ m* **using** *bitlen-bounds*[*OF ‹0 <m›*] **..**
then have *1 * ?B ≤ real-of-int m*
unfolding *of-int-le-iff*[*symmetric*] **by** *auto*
then show *1 ≤ real-of-int m / ?B* **by** *auto*

from *that* **have** *0 ≤ bitlen m − 1* **by** (*auto simp*: *bitlen-alt-def*)

have *m < 2^nat(bitlen m)* **using** *bitlen-bounds*[*OF that*] **..**
also from *that* **have** *... = 2^nat(bitlen m − 1 + 1)*
by (*auto simp*: *bitlen-def*)
also have *... = ?B * 2*
unfolding *nat-add-distrib*[*OF ‹0 ≤ bitlen m − 1› zero-le-one*] **by** *auto*
finally have *real-of-int m < 2 * ?B*
by (*metis* (*full-types*) *mult.commute power.simps*(*2*) *of-int-less-numeral-power-cancel-iff*)
then have *real-of-int m / ?B < 2 * ?B / ?B*
by (*rule divide-strict-right-mono*) *auto*
then show *real-of-int m / ?B < 2* **by** *auto*
qed

lemma *bitlen-le-iff-floorlog*:
bitlen x ≤ w ⟷ w ≥ 0 ∧ floorlog 2 (nat x) ≤ nat w
by (*auto simp*: *bitlen-def*)

lemma *bitlen-le-iff-power*:
bitlen x ≤ w ⟷ w ≥ 0 ∧ x < 2 ^ nat w
by (*auto simp*: *bitlen-le-iff-floorlog floorlog-le-iff*)

lemma *less-power-nat-iff-bitlen*:
x < 2 ^ w ⟷ bitlen (int x) ≤ w
using *bitlen-le-iff-power*[*of x w*]
by *auto*

lemma *bitlen-ge-iff-power*:
w ≤ bitlen x ⟷ w ≤ 0 ∨ 2 ^ (nat w − 1) ≤ x
unfolding *bitlen-def*
by (*auto simp flip*: *nat-le-iff intro*: *floorlog-geI dest*: *floorlog-geD*)

lemma *bitlen-twopow-add-eq*:
bitlen (2 ^ w + b) = w + 1 **if** *0 ≤ b b < 2 ^ w*
by (*auto simp*: *that nat-add-distrib bitlen-le-iff-power bitlen-ge-iff-power intro*!: *antisym*)

end

42 Various algebraic structures combined with a lattice

theory *Lattice-Algebras*
imports *Complex-Main*
begin

class *semilattice-inf-ab-group-add* = *ordered-ab-group-add* + *semilattice-inf*
begin

lemma *add-inf-distrib-left*: *a* + *inf b c* = *inf* (*a* + *b*) (*a* + *c*) (**is** *?L=?R*)
proof (*intro order.antisym*)
show *?R* ≤ *?L*
by (*metis add-commute diff-le-eq inf-greatest inf-le1 inf-le2*)
qed *simp*

lemma *add-inf-distrib-right*: *inf a b* + *c* = *inf* (*a* + *c*) (*b* + *c*)
using *add-commute add-inf-distrib-left* **by** *presburger*

end

class *semilattice-sup-ab-group-add* = *ordered-ab-group-add* + *semilattice-sup*
begin

lemma *add-sup-distrib-left*: *a* + *sup b c* = *sup* (*a* + *b*) (*a* + *c*) (**is** *?L* = *?R*)
proof (*rule order.antisym*)
show *?L* ≤ *?R*
by (*metis add-commute le-diff-eq sup.bounded-iff sup-ge1 sup-ge2*)
qed *simp*

lemma *add-sup-distrib-right*: *sup a b* + *c* = *sup* (*a* + *c*) (*b* + *c*)
proof −
have *c* + *sup a b* = *sup* (*c*+*a*) (*c*+*b*)
by (*simp add*: *add-sup-distrib-left*)
then show *?thesis*
by (*simp add*: *add.commute*)
qed

end

class *lattice-ab-group-add* = *ordered-ab-group-add* + *lattice*
begin

subclass *semilattice-inf-ab-group-add* ..
subclass *semilattice-sup-ab-group-add* ..

lemmas *add-sup-inf-distribs* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

lemma *inf-eq-neg-sup*: *inf a b* = − *sup* (− *a*) (− *b*)
proof (*rule inf-unique*)
fix *a b c* :: *'a*
show − *sup* (− *a*) (− *b*) ≤ *a*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
show − *sup* (−*a*) (−*b*) ≤ *b*
by (*rule add-le-imp-le-right* [*of* - *sup* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-sup-distrib-left*)
assume *a* ≤ *b a* ≤ *c*
then show *a* ≤ − *sup* (−*b*) (−*c*)
by (*subst neg-le-iff-le* [*symmetric*]) (*simp add*: *le-supI*)
qed

lemma *sup-eq-neg-inf*: *sup a b* = − *inf* (− *a*) (− *b*)
proof (*rule sup-unique*)
fix *a b c* :: *'a*
show *a* ≤ − *inf* (− *a*) (− *b*)
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
show *b* ≤ − *inf* (− *a*) (− *b*)
by (*rule add-le-imp-le-right* [*of* - *inf* (*uminus a*) (*uminus b*)])
(*simp*, *simp add*: *add-inf-distrib-left*)
show − *inf* (− *a*) (− *b*) ≤ *c* **if** *a* ≤ *c b* ≤ *c*
using *that* **by** (*subst neg-le-iff-le* [*symmetric*]) (*simp add*: *le-infI*)
qed

lemma *neg-inf-eq-sup*: − *inf a b* = *sup* (− *a*) (− *b*)
by (*simp add*: *inf-eq-neg-sup*)

lemma *diff-inf-eq-sup*: *a* − *inf b c* = *a* + *sup* (− *b*) (− *c*)
using *neg-inf-eq-sup* [*of b c*, *symmetric*] **by** *simp*

lemma *neg-sup-eq-inf*: − *sup a b* = *inf* (− *a*) (− *b*)
by (*simp add*: *sup-eq-neg-inf*)

lemma *diff-sup-eq-inf*: *a* − *sup b c* = *a* + *inf* (− *b*) (− *c*)
using *neg-sup-eq-inf* [*of b c*, *symmetric*] **by** *simp*

lemma *add-eq-inf-sup*: *a* + *b* = *sup a b* + *inf a b*
proof −
have *0* = − *inf 0* (*a* − *b*) + *inf* (*a* − *b*) *0*
by (*simp add*: *inf-commute*)
then have *0* = *sup 0* (*b* − *a*) + *inf* (*a* − *b*) *0*
by (*simp add*: *inf-eq-neg-sup*)
then have *0* = (− *a* + *sup a b*) + (*inf a b* + (− *b*))

by (*simp only*: *add-sup-distrib-left add-inf-distrib-right*) *simp*
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

42.1 Positive Part, Negative Part, Absolute Value

definition *nprt* :: *'a* ⇒ *'a*
where *nprt x* = *inf x 0*

definition *pprt* :: *'a* ⇒ *'a*
where *pprt x* = *sup x 0*

lemma *pprt-neg*: *pprt* (− *x*) = − *nprt x*
proof −
have *sup* (− *x*) *0* = *sup* (− *x*) (− *0*)
by (*simp only*: *minus-zero*)
also have ... = − *inf x 0*
by (*simp only*: *neg-inf-eq-sup*)
finally have *sup* (− *x*) *0* = − *inf x 0* **.**
then show *?thesis*
by (*simp only*: *pprt-def nprt-def*)
qed

lemma *nprt-neg*: *nprt* (− *x*) = − *pprt x*
proof −
from *pprt-neg* **have** *pprt* (− (− *x*)) = − *nprt* (− *x*) **.**
then have *pprt x* = − *nprt* (− *x*) **by** *simp*
then show *?thesis* **by** *simp*
qed

lemma *prts*: *a* = *pprt a* + *nprt a*
by (*simp add*: *pprt-def nprt-def flip*: *add-eq-inf-sup*)

lemma *zero-le-pprt*[*simp*]: *0* ≤ *pprt a*
by (*simp add*: *pprt-def*)

lemma *nprt-le-zero*[*simp*]: *nprt a* ≤ *0*
by (*simp add*: *nprt-def*)

lemma *le-eq-neg*: *a* ≤ − *b* ⟷ *a* + *b* ≤ *0*
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
show *?rhs*
by (*rule add-le-imp-le-right*[*of* - *uminus b* -]) (*simp add*: *add.assoc* ‹*?lhs*›)
next
assume *?rhs*
show *?lhs*

by (*rule add-le-imp-le-right*[*of* - *b* -]) (*simp add*: ‹*?rhs*›)
qed

lemma *pprt-0*[*simp*]: *pprt 0 = 0* **by** (*simp add*: *pprt-def*)
lemma *nprt-0*[*simp*]: *nprt 0 = 0* **by** (*simp add*: *nprt-def*)

lemma *pprt-eq-id* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow pprt\ x = x$
by (*simp add*: *pprt-def sup-absorb1*)

lemma *nprt-eq-id* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow nprt\ x = x$
by (*simp add*: *nprt-def inf-absorb1*)

lemma *pprt-eq-0* [*simp*, *no-atp*]: $x \leq 0 \Longrightarrow pprt\ x = 0$
by (*simp add*: *pprt-def sup-absorb2*)

lemma *nprt-eq-0* [*simp*, *no-atp*]: $0 \leq x \Longrightarrow nprt\ x = 0$
by (*simp add*: *nprt-def inf-absorb2*)

lemma *sup-0-imp-0*:
assumes $sup\ a\ (-\ a) = 0$
shows $a = 0$
proof −
have *pos*: $0 \leq a$ **if** $sup\ a\ (-\ a) = 0$ **for** $a :: 'a$
proof −
from *that* **have** $sup\ a\ (-\ a) + a = a$
by *simp*
then have $sup\ (a + a)\ 0 = a$
by (*simp add*: *add-sup-distrib-right*)
then have $sup\ (a + a)\ 0 \leq a$
by *simp*
then show *?thesis*
by (*blast intro*: *order-trans inf-sup-ord*)
qed
from *assms* **have** $**$: $sup\ (-a)\ (-(-a)) = 0$
by (*simp add*: *sup-commute*)
from *pos*[*OF assms*] *pos*[*OF* $**$] **show** $a = 0$
by *simp*
qed

lemma *inf-0-imp-0*: $inf\ a\ (-\ a) = 0 \Longrightarrow a = 0$
by (*metis local.neg-0-equal-iff-equal neg-inf-eq-sup sup-0-imp-0*)

lemma *inf-0-eq-0* [*simp*]: $inf\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*metis inf-0-imp-0 inf.idem minus-zero*)

lemma *sup-0-eq-0* [*simp*]: $sup\ a\ (-\ a) = 0 \longleftrightarrow a = 0$
by (*metis minus-zero sup.idem sup-0-imp-0*)

lemma *zero-le-double-add-iff-zero-le-single-add* [*simp*]: $0 \leq a + a \longleftrightarrow 0 \leq a$

(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
show *?rhs* **if** *?lhs*
proof −
from *that* **have** *a*: *inf* (*a* + *a*) *0* = *0*
by (*simp add*: *inf-commute inf-absorb1*)
have *inf a 0* + *inf a 0* = *inf* (*inf* (*a* + *a*) *0*) *a* (**is** *?l* = -)
by (*simp add*: *add-sup-inf-distribs inf-aci*)
then have *?l* = *0* + *inf a 0*
by (*simp add*: *a*, *simp add*: *inf-commute*)
then have *inf a 0* = *0*
by (*simp only*: *add-right-cancel*)
then show *?thesis*
unfolding *le-iff-inf* **by** (*simp add*: *inf-commute*)
qed
show *?lhs* **if** *?rhs*
by (*simp add*: *add-mono*[*OF that that*, *simplified*])
qed

lemma *double-zero* [*simp*]: *a* + *a* = *0* $\longleftrightarrow$ *a* = *0*
using *add-nonneg-eq-0-iff order.eq-iff* **by** *auto*

lemma *zero-less-double-add-iff-zero-less-single-add* [*simp*]: *0* < *a* + *a* $\longleftrightarrow$ *0* < *a*
by (*meson le-less-trans less-add-same-cancel2 less-le-not-le*
zero-le-double-add-iff-zero-le-single-add)

lemma *double-add-le-zero-iff-single-add-le-zero* [*simp*]: *a* + *a* ≤ *0* $\longleftrightarrow$ *a* ≤ *0*
proof −
have *a* + *a* ≤ *0* $\longleftrightarrow$ *0* ≤ − (*a* + *a*)
by (*subst le-minus-iff*) *simp*
moreover have ... $\longleftrightarrow$ *a* ≤ *0*
by (*simp only*: *minus-add-distrib zero-le-double-add-iff-zero-le-single-add*) *simp*
ultimately show *?thesis*
by *blast*
qed

lemma *double-add-less-zero-iff-single-less-zero* [*simp*]: *a* + *a* < *0* $\longleftrightarrow$ *a* < *0*
proof −
have *a* + *a* < *0* $\longleftrightarrow$ *0* < − (*a* + *a*)
by (*subst less-minus-iff*) *simp*
moreover have ... $\longleftrightarrow$ *a* < *0*
by (*simp only*: *minus-add-distrib zero-less-double-add-iff-zero-less-single-add*)
simp
ultimately show *?thesis*
by *blast*
qed

declare *neg-inf-eq-sup* [*simp*]
and *neg-sup-eq-inf* [*simp*]

and *diff-inf-eq-sup* [*simp*]
and *diff-sup-eq-inf* [*simp*]

lemma *le-minus-self-iff*: $a \leq - a \longleftrightarrow a \leq 0$
proof −
from *add-le-cancel-left* [*of uminus a plus a a zero*]
have $a \leq - a \longleftrightarrow a + a \leq 0$
by (*simp flip*: *add.assoc*)
then show *?thesis*
by *simp*
qed

lemma *minus-le-self-iff*: $- a \leq a \longleftrightarrow 0 \leq a$
proof −
have $- a \leq a \longleftrightarrow 0 \leq a + a$
using *add-le-cancel-left* [*of uminus a zero plus a a*]
by (*simp flip*: *add.assoc*)
then show *?thesis*
by *simp*
qed

lemma *zero-le-iff-zero-nprt*: $0 \leq a \longleftrightarrow nprt\ a = 0$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *le-zero-iff-zero-pprt*: $a \leq 0 \longleftrightarrow pprt\ a = 0$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *le-zero-iff-pprt-id*: $0 \leq a \longleftrightarrow pprt\ a = a$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-commute*)

lemma *zero-le-iff-nprt-id*: $a \leq 0 \longleftrightarrow nprt\ a = a$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-commute*)

lemma *pprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow pprt\ a \leq pprt\ b$
unfolding *le-iff-sup* **by** (*simp add*: *pprt-def sup-aci sup-assoc* [*symmetric*, *of a*])

lemma *nprt-mono* [*simp*, *no-atp*]: $a \leq b \Longrightarrow nprt\ a \leq nprt\ b$
unfolding *le-iff-inf* **by** (*simp add*: *nprt-def inf-aci inf-assoc* [*symmetric*, *of a*])

end

lemmas *add-sup-inf-distribs* =
add-inf-distrib-right add-inf-distrib-left add-sup-distrib-right add-sup-distrib-left

class *lattice-ab-group-add-abs* = *lattice-ab-group-add* + *abs* +
assumes *abs-lattice*: $|a| = sup\ a\ (- a)$
begin

lemma *abs-prts*: $|a| = pprt\ a - nprt\ a$
proof −
have $0 \leq |a|$
proof −
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$
by (*auto simp add*: *abs-lattice*)
show *?thesis*
by (*rule add-mono* [*OF a b*, *simplified*])
qed
then have $0 \leq sup\ a\ (-\ a)$
unfolding *abs-lattice* **.**
then have $sup\ (sup\ a\ (-\ a))\ 0 = sup\ a\ (-\ a)$
by (*rule sup-absorb1*)
then show *?thesis*
by (*simp add*: *add-sup-inf-distribs ac-simps pprt-def nprt-def abs-lattice*)
qed

subclass *ordered-ab-group-add-abs*
proof
have *abs-ge-zero* [*simp*]: $0 \leq |a|$ **for** *a*
proof −
have *a*: $a \leq |a|$ **and** *b*: $-\ a \leq |a|$
by (*auto simp add*: *abs-lattice*)
show $0 \leq |a|$
by (*rule add-mono* [*OF a b*, *simplified*])
qed
have *abs-leI*: $a \leq b \Longrightarrow -\ a \leq b \Longrightarrow |a| \leq b$ **for** *a b*
by (*simp add*: *abs-lattice le-supI*)
fix *a b*
show $0 \leq |a|$
by *simp*
show $a \leq |a|$
by (*auto simp add*: *abs-lattice*)
show $|-a| = |a|$
by (*simp add*: *abs-lattice sup-commute*)
show $-\ a \leq b \Longrightarrow |a| \leq b$ **if** $a \leq b$
using *that* **by** (*rule abs-leI*)
show $|a + b| \leq |a| + |b|$
proof −
have *g*: $|a| + |b| = sup\ (a + b)\ (sup\ (-\ a - b)\ (sup\ (-\ a + b)\ (a + (-\ b))))$
(**is** - = *sup ?m ?n*)
by (*simp add*: *abs-lattice add-sup-inf-distribs ac-simps*)
have *a*: $a + b \leq sup\ ?m\ ?n$
by *simp*
have *b*: $-\ a - b \leq ?n$
by *simp*
have *c*: $?n \leq sup\ ?m\ ?n$
by *simp*
from *b c* **have** *d*: $-\ a - b \leq sup\ ?m\ ?n$

```
    by (rule order-trans)
  have e: − a − b = − (a + b)
    by simp
  from a d e have |a + b| ≤ sup ?m ?n
    by (metis abs-leI)
  with g[symmetric] show ?thesis by simp
 qed
qed

end

lemma sup-eq-if:
 fixes a :: 'a::{lattice-ab-group-add,linorder}
 shows sup a (− a) = (if a < 0 then − a else a)
 using add-le-cancel-right [of a a − a, symmetric, simplified]
  and add-le-cancel-right [of −a a a, symmetric, simplified]
 by (auto simp: sup-max max.absorb1 max.absorb2)

lemma abs-if-lattice:
 fixes a :: 'a::{lattice-ab-group-add-abs,linorder}
 shows |a| = (if a < 0 then − a else a)
 by auto

lemma estimate-by-abs:
 fixes a b c :: 'a::lattice-ab-group-add-abs
 assumes a + b ≤ c
 shows a ≤ c + |b|
proof −
 from assms have a ≤ c + (− b)
  by (simp add: algebra-simps)
 have − b ≤ |b|
  by (rule abs-ge-minus-self)
 then have c + (− b) ≤ c + |b|
  by (rule add-left-mono)
 with ‹a ≤ c + (− b)› show ?thesis
  by (rule order-trans)
qed

class lattice-ring = ordered-ring + lattice-ab-group-add-abs
begin

subclass semilattice-inf-ab-group-add ..
subclass semilattice-sup-ab-group-add ..

end

lemma abs-le-mult:
 fixes a b :: 'a::lattice-ring
 shows |a * b| ≤ |a| * |b|
```

proof −
let *?x* = *pprt a* ∗ *pprt b* − *pprt a* ∗ *nprt b* − *nprt a* ∗ *pprt b* + *nprt a* ∗ *nprt b*
let *?y* = *pprt a* ∗ *pprt b* + *pprt a* ∗ *nprt b* + *nprt a* ∗ *pprt b* + *nprt a* ∗ *nprt b*
have *a*: |*a*| ∗ |*b*| = *?x*
by (*simp only*: *abs-prts*[*of a*] *abs-prts*[*of b*] *algebra-simps*)
have *bh*: *u* = *a* ⟹ *v* = *b* ⟹
u ∗ *v* = *pprt a* ∗ *pprt b* + *pprt a* ∗ *nprt b* +
nprt a ∗ *pprt b* + *nprt a* ∗ *nprt b* **for** *u v* :: *'a*
by (*metis add.commute combine-common-factor distrib-left prts*)
note *b* = *this*[*OF refl*[*of a*] *refl*[*of b*]]
have *xy*: − *?x* ≤ *?y*
apply *simp*
by (*meson add-increasing2 diff-le-eq neg-le-0-iff-le nprt-le-zero order.trans split-mult-pos-le zero-le-pprt*)
have *yx*: *?y* ≤ *?x*
apply *simp*
by (*metis add-decreasing2 diff-0 diff-mono diff-zero mult-nonpos-nonneg mult-right-mono-neg mult-zero-left nprt-le-zero zero-le-pprt*)
show *?thesis*
proof (*rule abs-leI*)
show *a* ∗ *b* ≤ |*a*| ∗ |*b*|
by (*simp only*: *a b yx*)
show − (*a* ∗ *b*) ≤ |*a*| ∗ |*b*|
by (*metis a bh minus-le-iff xy*)
qed
qed

instance *lattice-ring* ⊆ *ordered-ring-abs*
proof
fix *a b* :: *'a*::*lattice-ring*
assume *a*: (*0* ≤ *a* ∨ *a* ≤ *0*) ∧ (*0* ≤ *b* ∨ *b* ≤ *0*)
show |*a* ∗ *b*| = |*a*| ∗ |*b*|
proof −
have *s*: (*0* ≤ *a* ∗ *b*) ∨ (*a* ∗ *b* ≤ *0*)
by (*metis a split-mult-neg-le split-mult-pos-le*)
have *mulprts*: *a* ∗ *b* = (*pprt a* + *nprt a*) ∗ (*pprt b* + *nprt b*)
by (*simp flip*: *prts*)
show *?thesis*
proof (*cases 0* ≤ *a* ∗ *b*)
case *True*
then show *?thesis*
using *a split-mult-neg-le* **by** *fastforce*
next
case *False*
with *s* **have** *a* ∗ *b* ≤ *0*
by *simp*
then show *?thesis*
using *a split-mult-pos-le* **by** *fastforce*
qed

```
  qed
qed

lemma mult-le-prts:
  fixes a b :: 'a::lattice-ring
  assumes a1 ≤ a
    and a ≤ a2
    and b1 ≤ b
    and b ≤ b2
  shows a * b ≤
    pprt a2 * pprt b2 + pprt a1 * nprt b2 + nprt a2 * pprt b1 + nprt a1 * nprt
b1
proof −
  have a * b = (pprt a + nprt a) * (pprt b + nprt b)
    by (subst prts[symmetric])+ simp
  then have a * b = pprt a * pprt b + pprt a * nprt b + nprt a * pprt b + nprt
a * nprt b
    by (simp add: algebra-simps)
  moreover have pprt a * pprt b ≤ pprt a2 * pprt b2
    by (simp-all add: assms mult-mono)
  moreover have pprt a * nprt b ≤ pprt a1 * nprt b2
  proof −
    have pprt a * nprt b ≤ pprt a * nprt b2
      by (simp add: mult-left-mono assms)
    moreover have pprt a * nprt b2 ≤ pprt a1 * nprt b2
      by (simp add: mult-right-mono-neg assms)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * pprt b ≤ nprt a2 * pprt b1
  proof −
    have nprt a * pprt b ≤ nprt a2 * pprt b
      by (simp add: mult-right-mono assms)
    moreover have nprt a2 * pprt b ≤ nprt a2 * pprt b1
      by (simp add: mult-left-mono-neg assms)
    ultimately show ?thesis
      by simp
  qed
  moreover have nprt a * nprt b ≤ nprt a1 * nprt b1
  proof −
    have nprt a * nprt b ≤ nprt a * nprt b1
      by (simp add: mult-left-mono-neg assms)
    moreover have nprt a * nprt b1 ≤ nprt a1 * nprt b1
      by (simp add: mult-right-mono-neg assms)
    ultimately show ?thesis
      by simp
  qed
  ultimately show ?thesis
    by − (rule add-mono | simp)+
```

qed

lemma *mult-ge-prts*:
fixes *a b* :: *'a*::*lattice-ring*
assumes *a1* ≤ *a*
and *a* ≤ *a2*
and *b1* ≤ *b*
and *b* ≤ *b2*
shows *a* ∗ *b* ≥
nprt a1 ∗ *pprt b2* + *nprt a2* ∗ *nprt b2* + *pprt a1* ∗ *pprt b1* + *pprt a2* ∗ *nprt b1*
proof −
from *assms* **have** *a1*: − *a2* ≤ −*a*
by *auto*
from *assms* **have** *a2*: − *a* ≤ −*a1*
by *auto*
from *mult-le-prts*[*of* − *a2* − *a* − *a1 b1 b b2*,
OF a1 a2 assms(*3*) *assms*(*4*), *simplified nprt-neg pprt-neg*]
have *le*: − (*a* ∗ *b*) ≤
− *nprt a1* ∗ *pprt b2* + − *nprt a2* ∗ *nprt b2* +
− *pprt a1* ∗ *pprt b1* + − *pprt a2* ∗ *nprt b1*
by *simp*
then have − (− *nprt a1* ∗ *pprt b2* + − *nprt a2* ∗ *nprt b2* +
− *pprt a1* ∗ *pprt b1* + − *pprt a2* ∗ *nprt b1*) ≤ *a* ∗ *b*
by (*simp only*: *minus-le-iff*)
then show *?thesis*
by (*simp add*: *algebra-simps*)
qed

instance *int* :: *lattice-ring*
proof
show |*k*| = *sup k* (− *k*) **for** *k* :: *int*
by (*auto simp add*: *sup-int-def*)
qed

instance *real* :: *lattice-ring*
proof
show |*a*| = *sup a* (− *a*) **for** *a* :: *real*
by (*auto simp add*: *sup-real-def*)
qed

end

43 Floating-Point Numbers

theory *Float*
imports *Log-Nat Lattice-Algebras*
begin

definition *float* = {*m* ∗ *2 powr e* | (*m* :: *int*) (*e* :: *int*). *True*}

typedef *float* = *float*
  morphisms *real-of-float float-of*
  unfolding *float-def* **by** *auto*

setup-lifting *type-definition-float*

declare *real-of-float* [*code-unfold*]

lemmas *float-of-inject*[*simp*]

declare [[*coercion real-of-float* :: *float* ⇒ *real*]]

lemma *real-of-float-eq*: *f1* = *f2* ⟷ *real-of-float f1* = *real-of-float f2* **for** *f1 f2* :: *float*
  unfolding *real-of-float-inject* **..**

declare *real-of-float-inverse*[*simp*] *float-of-inverse* [*simp*]
declare *real-of-float* [*simp*]

43.1 Real operations preserving the representation as floating point number

lemma *floatI*: *m* ∗ *2 powr e* = *x* ⟹ *x* ∈ *float* **for** *m e* :: *int*
  by (*auto simp*: *float-def*)

lemma *zero-float*[*simp*]: *0* ∈ *float*
  by (*auto simp*: *float-def*)

lemma *one-float*[*simp*]: *1* ∈ *float*
  by (*intro floatI*[*of 1 0*]) *simp*

lemma *numeral-float*[*simp*]: *numeral i* ∈ *float*
  by (*intro floatI*[*of numeral i 0*]) *simp*

lemma *neg-numeral-float*[*simp*]: − *numeral i* ∈ *float*
  by (*intro floatI*[*of* − *numeral i 0*]) *simp*

lemma *real-of-int-float*[*simp*]: *real-of-int x* ∈ *float* **for** *x* :: *int*
  by (*intro floatI*[*of x 0*]) *simp*

lemma *real-of-nat-float*[*simp*]: *real x* ∈ *float* **for** *x* :: *nat*
  by (*intro floatI*[*of x 0*]) *simp*

lemma *two-powr-int-float*[*simp*]: *2 powr* (*real-of-int i*) ∈ *float* **for** *i* :: *int*
  by (*intro floatI*[*of 1 i*]) *simp*

lemma *two-powr-nat-float*[*simp*]: *2 powr* (*real i*) ∈ *float* **for** *i* :: *nat*

by (*intro floatI*[*of 1 i*]) *simp*

lemma *two-powr-minus-int-float*[*simp*]: *2 powr − (real-of-int i) ∈ float* **for** *i* :: *int*
by (*intro floatI*[*of 1 −i*]) *simp*

lemma *two-powr-minus-nat-float*[*simp*]: *2 powr − (real i) ∈ float* **for** *i* :: *nat*
by (*intro floatI*[*of 1 −i*]) *simp*

lemma *two-powr-numeral-float*[*simp*]: *2 powr numeral i ∈ float*
by (*intro floatI*[*of 1 numeral i*]) *simp*

lemma *two-powr-neg-numeral-float*[*simp*]: *2 powr − numeral i ∈ float*
by (*intro floatI*[*of 1 − numeral i*]) *simp*

lemma *two-pow-float*[*simp*]: *2 ^ n ∈ float*
by (*intro floatI*[*of 1 n*]) (*simp add*: *powr-realpow*)

lemma *plus-float*[*simp*]: *r ∈ float ⟹ p ∈ float ⟹ r + p ∈ float*
unfolding *float-def*
proof (*safe*, *simp*)
have ∗: ∃(*m*::*int*) (*e*::*int*). *m1 ∗ 2 powr e1 + m2 ∗ 2 powr e2 = m ∗ 2 powr e*
if *e1 ≤ e2* **for** *e1 m1 e2 m2* :: *int*
proof −
from *that* **have** *m1 ∗ 2 powr e1 + m2 ∗ 2 powr e2 = (m1 + m2 ∗ 2 ^ nat (e2 − e1)) ∗ 2 powr e1*
by (*simp add*: *powr-diff field-simps flip*: *powr-realpow*)
then show *?thesis*
by *blast*
qed
fix *e1 m1 e2 m2* :: *int*
consider *e2 ≤ e1* | *e1 ≤ e2* **by** (*rule linorder-le-cases*)
then show ∃(*m*::*int*) (*e*::*int*). *m1 ∗ 2 powr e1 + m2 ∗ 2 powr e2 = m ∗ 2 powr e*
proof *cases*
case *1*
from ∗[*OF this*, *of m2 m1*] **show** *?thesis*
by (*simp add*: *ac-simps*)
next
case *2*
then show *?thesis* **by** (*rule* ∗)
qed
qed

lemma *uminus-float*[*simp*]: *x ∈ float ⟹ −x ∈ float*
by (*simp add*: *float-def*) (*metis mult-minus-left of-int-minus*)

lemma *times-float*[*simp*]: *x ∈ float ⟹ y ∈ float ⟹ x ∗ y ∈ float*
apply (*clarsimp simp*: *float-def mult-ac*)

by (*metis mult.assoc of-int-mult of-int-add powr-add*)

lemma *minus-float*[*simp*]: $x \in$ *float* $\Longrightarrow y \in$ *float* $\Longrightarrow x - y \in$ *float*
using *plus-float* [*of* $x - y$] **by** *simp*

lemma *abs-float*[*simp*]: $x \in$ *float* $\Longrightarrow |x| \in$ *float*
by (*cases x rule*: *linorder-cases*[*of 0*]) *auto*

lemma *sgn-of-float*[*simp*]: $x \in$ *float* $\Longrightarrow$ *sgn* $x \in$ *float*
by (*simp add*: *sgn-real-def*)

lemma *div-power-2-float*[*simp*]: $x \in$ *float* $\Longrightarrow x$ / 2^$d \in$ *float*
by (*simp add*: *float-def*) (*metis of-int-diff of-int-of-nat-eq powr-diff powr-realpow zero-less-numeral times-divide-eq-right*)

lemma *div-power-2-int-float*[*simp*]: $x \in$ *float* $\Longrightarrow x$ / (2::*int*)^$d \in$ *float*
by *simp*

lemma *div-numeral-Bit0-float*[*simp*]:
assumes x / *numeral* $n \in$ *float*
shows x / (*numeral* (*Num.Bit0* n)) $\in$ *float*
proof −
have (x / *numeral* n) / 2^1 $\in$ *float*
by (*intro assms div-power-2-float*)
also have (x / *numeral* n) / 2^1 = x / (*numeral* (*Num.Bit0* n))
by (*induct* n) *auto*
finally show *?thesis* **.**
qed

lemma *div-neg-numeral-Bit0-float*[*simp*]:
assumes x / *numeral* $n \in$ *float*
shows x / (− *numeral* (*Num.Bit0* n)) $\in$ *float*
using *assms* **by** *force*

lemma *power-float*[*simp*]:
assumes $a \in$ *float*
shows a ^ $b \in$ *float*
proof −
from *assms* **obtain** m e :: *int* **where** $a = m * 2$ *powr* e
by (*auto simp*: *float-def*)
then show *?thesis*
by (*intro floatI*[**where** m=m^b **and** $e = e$*b])
(*auto simp*: *powr-powr power-mult-distrib simp flip*: *powr-realpow*)
qed

lift-definition *Float* :: *int* ⇒ *int* ⇒ *float* **is** λ(m::*int*) (e::*int*). $m * 2$ *powr* e
by *simp*
declare *Float.rep-eq*[*simp*]

code-datatype *Float*

lemma *compute-real-of-float*[*code*]:
real-of-float (*Float m e*) = (*if e* ≥ *0 then m* ∗ *2* ^ *nat e else m* / *2* ^ (*nat* (−*e*)))
by (*simp add*: *powr-int*)

43.2 Arithmetic operations on floating point numbers

instantiation *float* :: {*ring-1*,*linorder*,*linordered-ring*,*linordered-idom*,*numeral*,*equal*}
begin

lift-definition *zero-float* :: *float* **is** *0* **by** *simp*
declare *zero-float.rep-eq*[*simp*]

lift-definition *one-float* :: *float* **is** *1* **by** *simp*
declare *one-float.rep-eq*[*simp*]

lift-definition *plus-float* :: *float* ⇒ *float* ⇒ *float* **is** (+) **by** *simp*
declare *plus-float.rep-eq*[*simp*]

lift-definition *times-float* :: *float* ⇒ *float* ⇒ *float* **is** (∗) **by** *simp*
declare *times-float.rep-eq*[*simp*]

lift-definition *minus-float* :: *float* ⇒ *float* ⇒ *float* **is** (−) **by** *simp*
declare *minus-float.rep-eq*[*simp*]

lift-definition *uminus-float* :: *float* ⇒ *float* **is** *uminus* **by** *simp*
declare *uminus-float.rep-eq*[*simp*]

lift-definition *abs-float* :: *float* ⇒ *float* **is** *abs* **by** *simp*
declare *abs-float.rep-eq*[*simp*]

lift-definition *sgn-float* :: *float* ⇒ *float* **is** *sgn* **by** *simp*
declare *sgn-float.rep-eq*[*simp*]

lift-definition *equal-float* :: *float* ⇒ *float* ⇒ *bool* **is** (=) :: *real* ⇒ *real* ⇒ *bool* **.**

lift-definition *less-eq-float* :: *float* ⇒ *float* ⇒ *bool* **is** (≤) **.**
declare *less-eq-float.rep-eq*[*simp*]

lift-definition *less-float* :: *float* ⇒ *float* ⇒ *bool* **is** (<) **.**
declare *less-float.rep-eq*[*simp*]

instance
by *standard* (*transfer*; *fastforce simp add*: *field-simps intro*: *mult-left-mono mult-right-mono*)+

end

lemma *real-of-float* [*simp*]: *real-of-float* (*of-nat n*) = *of-nat n*

by (*induct n*) *simp-all*

lemma *real-of-float-of-int-eq* [*simp*]: *real-of-float* (*of-int z*) = *of-int z*
by (*cases z rule*: *int-diff-cases*) (*simp-all add*: *of-rat-diff*)

lemma *Float-0-eq-0*[*simp*]: *Float 0 e* = *0*
by *transfer simp*

lemma *real-of-float-power*[*simp*]: *real-of-float* (*f^n*) = *real-of-float f^n* **for** *f* :: *float*
by (*induct n*) *simp-all*

lemma *real-of-float-min*: *real-of-float* (*min x y*) = *min* (*real-of-float x*) (*real-of-float y*)
and *real-of-float-max*: *real-of-float* (*max x y*) = *max* (*real-of-float x*) (*real-of-float y*)
for *x y* :: *float*
by (*simp-all add*: *min-def max-def*)

instance *float* :: *unbounded-dense-linorder*
proof
fix *a b* :: *float*
show ∃ *c*. *a* < *c*
by (*metis Float.real-of-float less-float.rep-eq reals-Archimedean2*)
show ∃ *c*. *c* < *a*
by (*metis add-0 add-strict-right-mono neg-less-0-iff-less zero-less-one*)
show ∃ *c*. *a* < *c* ∧ *c* < *b* **if** *a* < *b*
apply (*rule exI*[*of* - (*a* + *b*) ∗ *Float 1* (− *1*)])
using *that*
apply *transfer*
apply (*simp add*: *powr-minus*)
done
qed

instantiation *float* :: *lattice-ab-group-add*
begin

definition *inf-float* :: *float* ⇒ *float* ⇒ *float*
where *inf-float a b* = *min a b*

definition *sup-float* :: *float* ⇒ *float* ⇒ *float*
where *sup-float a b* = *max a b*

instance
by *standard* (*transfer*; *simp add*: *inf-float-def sup-float-def real-of-float-min real-of-float-max*)+

end

lemma *float-numeral*[*simp*]: *real-of-float* (*numeral x* :: *float*) = *numeral x*
by (*metis of-int-numeral real-of-float-of-int-eq*)

lemma *transfer-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (*numeral* :: - ⇒ *real*) (*numeral* :: - ⇒ *float*)
by (*simp add*: *rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma *float-neg-numeral*[*simp*]: *real-of-float* (− *numeral x* :: *float*) = − *numeral x*
by *simp*

lemma *transfer-neg-numeral* [*transfer-rule*]:
rel-fun (=) *pcr-float* (− *numeral* :: - ⇒ *real*) (− *numeral* :: - ⇒ *float*)
by (*simp add*: *rel-fun-def float.pcr-cr-eq cr-float-def*)

lemma *float-of-numeral*: *numeral k* = *float-of* (*numeral k*)
and *float-of-neg-numeral*: − *numeral k* = *float-of* (− *numeral k*)
unfolding *real-of-float-eq* **by** *simp-all*

43.3 Quickcheck

instantiation *float* :: *exhaustive*
begin

definition *exhaustive-float* **where**
exhaustive-float f d =
Quickcheck-Exhaustive.exhaustive (λ*x*. *Quickcheck-Exhaustive.exhaustive* (λ*y*. *f* (*Float x y*)) *d*) *d*

instance ..

end

context
includes *term-syntax*
begin

definition [*code-unfold*]:
valtermify-float x y = *Code-Evaluation.valtermify Float* {·} *x* {·} *y*

end

instantiation *float* :: *full-exhaustive*
begin

definition
full-exhaustive-float f d =
Quickcheck-Exhaustive.full-exhaustive
(λ*x*. *Quickcheck-Exhaustive.full-exhaustive* (λ*y*. *f* (*valtermify-float x y*)) *d*) *d*

instance ..

end

instantiation *float* :: *random*
begin

definition *Quickcheck-Random.random i* =
scomp (*Quickcheck-Random.random* (*2* ^ *nat-of-natural i*))
(λ*man. scomp* (*Quickcheck-Random.random i*) (λ*exp. Pair* (*valtermify-float man exp*)))

instance ..

end

43.4 Represent floats as unique mantissa and exponent

lemma *int-induct-abs*[*case-names less*]:
fixes *j* :: *int*
assumes *H*: ⋀*n*. (⋀*i*. |*i*| < |*n*| ⟹ *P i*) ⟹ *P n*
shows *P j*
proof (*induct nat* |*j*| *arbitrary*: *j rule*: *less-induct*)
case *less*
show *?case* **by** (*rule H*[*OF less*]) *simp*
qed

lemma *int-cancel-factors*:
fixes *n* :: *int*
assumes *1* < *r*
shows *n* = *0* ∨ (∃ *k i*. *n* = *k* ∗ *r* ^ *i* ∧ ¬ *r dvd k*)
proof (*induct n rule*: *int-induct-abs*)
case (*less n*)
have ∃ *k i*. *n* = *k* ∗ *r* ^ *Suc i* ∧ ¬ *r dvd k* **if** *n* ≠ *0 n* = *m* ∗ *r* **for** *m*
proof −
from *that* **have** |*m* | < |*n*|
using ‹*1* < *r*› **by** (*simp add*: *abs-mult*)
from *less*[*OF this*] *that* **show** *?thesis* **by** *auto*
qed
then show *?case*
by (*metis dvd-def monoid-mult-class.mult.right-neutral mult.commute power-0*)
qed

lemma *mult-powr-eq-mult-powr-iff-asym*:
fixes *m1 m2 e1 e2* :: *int*
assumes *m1*: ¬ *2 dvd m1*
and *e1* ≤ *e2*
shows *m1* ∗ *2 powr e1* = *m2* ∗ *2 powr e2* ⟷ *m1* = *m2* ∧ *e1* = *e2*
(**is** *?lhs* ⟷ *?rhs*)
proof

```
  show ?rhs if eq: ?lhs
  proof −
    have m1 ≠ 0
      using m1 unfolding dvd-def by auto
    from ‹e1 ≤ e2› eq have m1 = m2 * 2 powr nat (e2 − e1)
      by (simp add: powr-diff field-simps)
    also have ... = m2 * 2^nat (e2 − e1)
      by (simp add: powr-realpow)
    finally have m1-eq: m1 = m2 * 2^nat (e2 − e1)
      by linarith
    with m1 have m1 = m2
      by (cases nat (e2 − e1)) (auto simp add: dvd-def)
    then show ?thesis
      using eq ‹m1 ≠ 0› by (simp add: powr-inj)
  qed
  show ?lhs if ?rhs
    using that by simp
qed

lemma mult-powr-eq-mult-powr-iff:
  ¬ 2 dvd m1 ⟹ ¬ 2 dvd m2 ⟹ m1 * 2 powr e1 = m2 * 2 powr e2 ⟷ m1 =
m2 ∧ e1 = e2
  for m1 m2 e1 e2 :: int
  using mult-powr-eq-mult-powr-iff-asym[of m1 e1 e2 m2]
  using mult-powr-eq-mult-powr-iff-asym[of m2 e2 e1 m1]
  by (cases e1 e2 rule: linorder-le-cases) auto

lemma floatE-normed:
  assumes x: x ∈ float
  obtains (zero) x = 0
   | (powr) m e :: int where x = m * 2 powr e ¬ 2 dvd m x ≠ 0
proof −
  have ∃(m::int) (e::int). x = m * 2 powr e ∧ ¬ (2::int) dvd m if x ≠ 0
  proof −
    from x obtain m e :: int where x: x = m * 2 powr e
      by (auto simp: float-def)
    with ‹x ≠ 0› int-cancel-factors[of 2 m] obtain k i where m = k * 2 ^ i ¬ 2
dvd k
      by auto
    with ‹¬ 2 dvd k› x have x = real-of-int k * 2 powr real-of-int (e + int i) ∧
odd k
      by (simp add: powr-add powr-realpow)
    then show ?thesis
      by blast
  qed
  with that show thesis by blast
qed

lemma float-normed-cases:
```

```
  fixes f :: float
  obtains (zero) f = 0
   | (powr) m e :: int where real-of-float f = m * 2 powr e ¬ 2 dvd m f ≠ 0
proof (atomize-elim, induct f)
  case (float-of y)
  then show ?case
    by (cases rule: floatE-normed) (auto simp: zero-float-def)
qed

definition mantissa :: float ⇒ int
  where mantissa f =
    fst (SOME p::int × int. (f = 0 ∧ fst p = 0 ∧ snd p = 0) ∨
      (f ≠ 0 ∧ real-of-float f = real-of-int (fst p) * 2 powr real-of-int (snd p) ∧ ¬
2 dvd fst p))

definition exponent :: float ⇒ int
  where exponent f =
    snd (SOME p::int × int. (f = 0 ∧ fst p = 0 ∧ snd p = 0) ∨
      (f ≠ 0 ∧ real-of-float f = real-of-int (fst p) * 2 powr real-of-int (snd p) ∧ ¬
2 dvd fst p))

lemma exponent-0[simp]: exponent 0 = 0 (is ?E)
  and mantissa-0[simp]: mantissa 0 = 0 (is ?M)
proof −
  have ⋀p::int × int. fst p = 0 ∧ snd p = 0 ⟷ p = (0, 0)
    by auto
  then show ?E ?M
    by (auto simp add: mantissa-def exponent-def zero-float-def)
qed

lemma mantissa-exponent: real-of-float f = mantissa f * 2 powr exponent f (is
?E)
  and mantissa-not-dvd: f ≠ 0 ⟹ ¬ 2 dvd mantissa f (is - ⟹ ?D)
proof cases
  assume [simp]: f ≠ 0
  have f = mantissa f * 2 powr exponent f ∧ ¬ 2 dvd mantissa f
  proof (cases f rule: float-normed-cases)
    case zero
    then show ?thesis by simp
  next
    case (powr m e)
    then have ∃ p::int × int. (f = 0 ∧ fst p = 0 ∧ snd p = 0) ∨
      (f ≠ 0 ∧ real-of-float f = real-of-int (fst p) * 2 powr real-of-int (snd p) ∧ ¬
2 dvd fst p)
      by auto
    then show ?thesis
      unfolding exponent-def mantissa-def
      by (rule someI2-ex) simp
  qed
```

```
  then show ?E ?D by auto
qed simp

lemma mantissa-noteq-0: f ≠ 0 ⟹ mantissa f ≠ 0
  using mantissa-not-dvd[of f] by auto

lemma mantissa-eq-zero-iff: mantissa x = 0 ⟷ x = 0
  (is ?lhs ⟷ ?rhs)
proof
  show ?rhs if ?lhs
  proof −
    from that have z: 0 = real-of-float x
      using mantissa-exponent by simp
    show ?thesis
      by (simp add: zero-float-def z)
  qed
  show ?lhs if ?rhs
    using that by simp
qed

lemma mantissa-pos-iff: 0 < mantissa x ⟷ 0 < x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma mantissa-nonneg-iff: 0 ≤ mantissa x ⟷ 0 ≤ x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma mantissa-neg-iff: 0 > mantissa x ⟷ 0 > x
  by (auto simp: mantissa-exponent algebra-split-simps)

lemma
  fixes m e :: int
  defines f ≡ float-of (m * 2 powr e)
  assumes dvd: ¬ 2 dvd m
  shows mantissa-float: mantissa f = m (is ?M)
    and exponent-float: m ≠ 0 ⟹ exponent f = e (is - ⟹ ?E)
proof cases
  assume m = 0
  with dvd show mantissa f = m by auto
next
  assume m ≠ 0
  then have f-not-0: f ≠ 0 by (simp add: f-def zero-float-def)
  from mantissa-exponent[of f] have m * 2 powr e = mantissa f * 2 powr exponent
f
    by (auto simp add: f-def)
  then show ?M ?E
    using mantissa-not-dvd[OF f-not-0] dvd
    by (auto simp: mult-powr-eq-mult-powr-iff)
qed
```

43.5 Compute arithmetic operations

lemma *Float-mantissa-exponent*: *Float* (*mantissa f*) (*exponent f*) = *f*
unfolding *real-of-float-eq mantissa-exponent*[*of f*] **by** *simp*

lemma *Float-cases* [*cases type*: *float*]:
fixes *f* :: *float*
obtains (*Float*) *m e* :: *int* **where** *f* = *Float m e*
using *Float-mantissa-exponent*[*symmetric*]
by (*atomize-elim*) *auto*

lemma *denormalize-shift*:
assumes *f-def*: *f* = *Float m e*
and *not-0*: *f* ≠ *0*
obtains *i* **where** *m* = *mantissa f* ∗ *2* ^ *i e* = *exponent f* − *i*
proof
from *mantissa-exponent*[*of f*] *f-def*
have *m* ∗ *2 powr e* = *mantissa f* ∗ *2 powr exponent f*
by *simp*
then have *eq*: *m* = *mantissa f* ∗ *2 powr* (*exponent f* − *e*)
by (*simp add*: *powr-diff field-simps*)
moreover
have *e* ≤ *exponent f*
proof (*rule ccontr*)
assume ¬ *e* ≤ *exponent f*
then have *pos*: *exponent f* < *e* **by** *simp*
then have *2 powr* (*exponent f* − *e*) = *2 powr* − *real-of-int* (*e* − *exponent f*)
by *simp*
also have ... = *1* / *2*^*nat* (*e* − *exponent f*)
using *pos* **by** (*simp flip*: *powr-realpow add*: *powr-diff*)
finally have *m* ∗ *2*^*nat* (*e* − *exponent f*) = *real-of-int* (*mantissa f*)
using *eq* **by** *simp*
then have *mantissa f* = *m* ∗ *2*^*nat* (*e* − *exponent f*)
by *linarith*
with ‹*exponent f* < *e*› **have** *2 dvd mantissa f*
by (*force intro*: *dvdI*[**where** *k*=*m* ∗ *2*^(*nat* (*e*−*exponent f*)) *div 2*])
then show *False* **using** *mantissa-not-dvd*[*OF not-0*] **by** *simp*
qed
ultimately have *real-of-int m* = *mantissa f* ∗ *2*^*nat* (*exponent f* − *e*)
by (*simp flip*: *powr-realpow*)
with ‹*e* ≤ *exponent f*›
show *m* = *mantissa f* ∗ *2* ^ *nat* (*exponent f* − *e*)
by *linarith*
show *e* = *exponent f* − *nat* (*exponent f* − *e*)
using ‹*e* ≤ *exponent f*› **by** *auto*
qed

context
begin

qualified lemma *compute-float-zero*[*code-unfold*, *code*]: *0 = Float 0 0*
by *transfer simp*

qualified lemma *compute-float-one*[*code-unfold*, *code*]: *1 = Float 1 0*
by *transfer simp*

lift-definition *normfloat* :: *float* ⇒ *float* **is** λ*x*. *x* **.**
lemma *normloat-id*[*simp*]: *normfloat x = x* **by** *transfer rule*

qualified lemma *compute-normfloat*[*code*]:
normfloat (*Float m e*) =
(*if m mod 2 = 0* ∧ *m* ≠ *0 then normfloat* (*Float* (*m div 2*) (*e* + *1*))
else if m = 0 then 0 else Float m e)
by *transfer* (*auto simp add*: *powr-add zmod-eq-0-iff*)

qualified lemma *compute-float-numeral*[*code-abbrev*]: *Float* (*numeral k*) *0* = *numeral k*
by *transfer simp*

qualified lemma *compute-float-neg-numeral*[*code-abbrev*]: *Float* (− *numeral k*) *0* = − *numeral k*
by *transfer simp*

qualified lemma *compute-float-uminus*[*code*]: − *Float m1 e1* = *Float* (− *m1*) *e1*
by *transfer simp*

qualified lemma *compute-float-times*[*code*]: *Float m1 e1* ∗ *Float m2 e2* = *Float* (*m1* ∗ *m2*) (*e1* + *e2*)
by *transfer* (*simp add*: *field-simps powr-add*)

qualified lemma *compute-float-plus*[*code*]:
Float m1 e1 + *Float m2 e2* =
(*if m1 = 0 then Float m2 e2*
else if m2 = 0 then Float m1 e1
else if e1 ≤ *e2 then Float* (*m1* + *m2* ∗ *2^nat* (*e2* − *e1*)) *e1*
else Float (*m2* + *m1* ∗ *2^nat* (*e1* − *e2*)) *e2*)
by *transfer* (*simp add*: *field-simps powr-realpow*[*symmetric*] *powr-diff*)

qualified lemma *compute-float-minus*[*code*]: *f* − *g* = *f* + (−*g*) **for** *f g* :: *float*
by *simp*

qualified lemma *compute-float-sgn*[*code*]:
sgn (*Float m1 e1*) = (*if 0* < *m1 then 1 else if m1* < *0 then* −*1 else 0*)
by *transfer* (*simp add*: *sgn-mult*)

lift-definition *is-float-pos* :: *float* ⇒ *bool* **is** (<) *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-pos*[*code*]: *is-float-pos* (*Float m e*) ⟷ *0* < *m*
by *transfer* (*auto simp add*: *zero-less-mult-iff not-le*[*symmetric*, *of - 0*])

lift-definition *is-float-nonneg* :: *float* ⇒ *bool* **is** (≤) *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-nonneg*[*code*]: *is-float-nonneg* (*Float m e*) ⟷ *0* ≤ *m*
by *transfer* (*auto simp add*: *zero-le-mult-iff not-less*[*symmetric*, *of - 0*])

lift-definition *is-float-zero* :: *float* ⇒ *bool* **is** (=) *0* :: *real* ⇒ *bool* **.**

qualified lemma *compute-is-float-zero*[*code*]: *is-float-zero* (*Float m e*) ⟷ *0* = *m*
by *transfer* (*auto simp add*: *is-float-zero-def*)

qualified lemma *compute-float-abs*[*code*]: |*Float m e*| = *Float* |*m*| *e*
by *transfer* (*simp add*: *abs-mult*)

qualified lemma *compute-float-eq*[*code*]: *equal-class.equal f g* = *is-float-zero* (*f* − *g*)
by *transfer simp*

end

43.6 Lemmas for types *real*, *nat*, *int*

lemmas *real-of-ints* =
of-int-add
of-int-minus
of-int-diff
of-int-mult
of-int-power
of-int-numeral of-int-neg-numeral

lemmas *int-of-reals* = *real-of-ints*[*symmetric*]

43.7 Rounding Real Numbers

definition *round-down* :: *int* ⇒ *real* ⇒ *real*
where *round-down prec x* = ⌊*x* ∗ *2 powr prec*⌋ ∗ *2 powr* −*prec*

definition *round-up* :: *int* ⇒ *real* ⇒ *real*
where *round-up prec x* = ⌈*x* ∗ *2 powr prec*⌉ ∗ *2 powr* −*prec*

lemma *round-down-float*[*simp*]: *round-down prec x* ∈ *float*
unfolding *round-down-def*
by (*auto intro*!: *times-float simp flip*: *of-int-minus*)

lemma *round-up-float*[*simp*]: *round-up prec x* ∈ *float*
unfolding *round-up-def*
by (*auto intro*!: *times-float simp flip*: *of-int-minus*)

lemma *round-up*: *x* ≤ *round-up prec x*

by (*simp add*: *powr-minus-divide le-divide-eq round-up-def ceiling-correct*)

lemma *round-down*: *round-down prec x* $\leq$ *x*
by (*simp add*: *powr-minus-divide divide-le-eq round-down-def*)

lemma *round-up-0*[*simp*]: *round-up p 0* = *0*
unfolding *round-up-def* **by** *simp*

lemma *round-down-0*[*simp*]: *round-down p 0* = *0*
unfolding *round-down-def* **by** *simp*

lemma *round-up-diff-round-down*: *round-up prec x* − *round-down prec x* $\leq$ *2 powr* −*prec*
proof −
have *round-up prec x* − *round-down prec x* = (⌈*x* ∗ *2 powr prec*⌉ − ⌊*x* ∗ *2 powr prec*⌋) ∗ *2 powr* −*prec*
by (*simp add*: *round-up-def round-down-def field-simps*)
also have ... $\leq$ *1* ∗ *2 powr* −*prec*
by (*rule mult-mono*)
(*auto simp flip*: *of-int-diff simp*: *ceiling-diff-floor-le-1*)
finally show *?thesis* **by** *simp*
qed

lemma *round-down-shift*: *round-down p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-down* (*p* + *k*) *x*
unfolding *round-down-def*
by (*simp add*: *powr-add powr-mult field-simps powr-diff*)
(*simp flip*: *powr-add*)

lemma *round-up-shift*: *round-up p* (*x* ∗ *2 powr k*) = *2 powr k* ∗ *round-up* (*p* + *k*) *x*
unfolding *round-up-def*
by (*simp add*: *powr-add powr-mult field-simps powr-diff*)
(*simp flip*: *powr-add*)

lemma *round-up-uminus-eq*: *round-up p* (−*x*) = − *round-down p x*
and *round-down-uminus-eq*: *round-down p* (−*x*) = − *round-up p x*
by (*auto simp*: *round-up-def round-down-def ceiling-def*)

lemma *round-up-mono*: *x* $\leq$ *y* ⟹ *round-up p x* $\leq$ *round-up p y*
by (*auto intro*!: *ceiling-mono simp*: *round-up-def*)

lemma *round-up-le1*:
assumes *x* $\leq$ *1 prec* $\geq$ *0*
shows *round-up prec x* $\leq$ *1*
proof −
have *real-of-int* ⌈*x* ∗ *2 powr prec*⌉ $\leq$ *real-of-int* ⌈*2 powr real-of-int prec*⌉
using *assms* **by** (*auto intro*!: *ceiling-mono*)
also have ... = *2 powr prec* **using** *assms* **by** (*auto simp*: *powr-int intro*!:

exI[**where** *x=2^nat prec*])
finally show *?thesis*
by (*simp add*: *round-up-def*) (*simp add*: *powr-minus inverse-eq-divide*)
qed

lemma *round-up-less1*:
assumes $x < 1 / 2$ $p > 0$
shows *round-up p x* < 1
proof −
have $x * 2$ *powr* $p < 1 / 2 *$ 2 *powr p*
using *assms* **by** *simp*
also have $\ldots \leq 2$ *powr* $p - 1$ **using** ‹$p > 0$›
by (*auto simp*: *powr-diff powr-int field-simps self-le-power*)
finally show *?thesis* **using** ‹$p > 0$›
by (*simp add*: *round-up-def field-simps powr-minus powr-int ceiling-less-iff*)
qed

lemma *round-down-ge1*:
assumes *x*: $x \geq 1$
assumes *prec*: $p \geq -$ *log* 2 *x*
shows $1 \leq$ *round-down p x*
proof *cases*
assume *nonneg*: $0 \leq p$
have *2 powr p* = *real-of-int* $\lfloor 2$ *powr real-of-int p*$\rfloor$
using *nonneg* **by** (*auto simp*: *powr-int*)
also have $\ldots \leq$ *real-of-int* $\lfloor x * 2$ *powr p*$\rfloor$
using *assms* **by** (*auto intro*!: *floor-mono*)
finally show *?thesis*
by (*simp add*: *round-down-def*) (*simp add*: *powr-minus inverse-eq-divide*)
next
assume *neg*: $\neg\ 0 \leq p$
have $x =$ *2 powr* (*log 2 x*)
using *x* **by** *simp*
also have *2 powr* (*log 2 x*) $\geq$ *2 powr* $-\ p$
using *prec* **by** *auto*
finally have *x-le*: $x \geq$ *2 powr* $-p$ **.**

from *neg* **have** *2 powr real-of-int p* $\leq$ *2 powr 0*
by (*intro powr-mono*) *auto*
also have $\ldots \leq \lfloor$*2 powr 0*::*real*$\rfloor$ **by** *simp*
also have $\ldots \leq \lfloor x * 2$ *powr* (*real-of-int p*)$\rfloor$
unfolding *of-int-le-iff*
using *x x-le* **by** (*intro floor-mono*) (*simp add*: *powr-minus-divide field-simps*)
finally show *?thesis*
using *prec x*
by (*simp add*: *round-down-def powr-minus-divide pos-le-divide-eq*)
qed

lemma *round-up-le0*: $x \leq 0 \Longrightarrow$ *round-up p x* ≤ 0

unfolding *round-up-def*
by (*auto simp*: *field-simps mult-le-0-iff zero-le-mult-iff*)

43.8 Rounding Floats

definition *div-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *div-twopow x n* = *x div* (*2* ^ *n*)

definition *mod-twopow* :: *int* ⇒ *nat* ⇒ *int*
where [*simp*]: *mod-twopow x n* = *x mod* (*2* ^ *n*)

lemma *compute-div-twopow*[*code*]:
div-twopow x n = (*if x* = *0* ∨ *x* = −*1* ∨ *n* = *0 then x else div-twopow* (*x div 2*) (*n* − *1*))
by (*cases n*) (*auto simp*: *zdiv-zmult2-eq div-eq-minus1*)

lemma *compute-mod-twopow*[*code*]:
mod-twopow x n = (*if n* = *0 then 0 else x mod 2* + *2* ∗ *mod-twopow* (*x div 2*) (*n* − *1*))
by (*cases n*) (*auto simp*: *zmod-zmult2-eq*)

lift-definition *float-up* :: *int* ⇒ *float* ⇒ *float* **is** *round-up* **by** *simp*
declare *float-up.rep-eq*[*simp*]

lemma *round-up-correct*: *round-up e f* − *f* ∈ {*0*..*2 powr* −*e*}
unfolding *atLeastAtMost-iff*
proof
have *round-up e f* − *f* ≤ *round-up e f* − *round-down e f*
using *round-down* **by** *simp*
also have ... ≤ *2 powr* −*e*
using *round-up-diff-round-down* **by** *simp*
finally show *round-up e f* − *f* ≤ *2 powr* − (*real-of-int e*)
by *simp*
qed (*simp add*: *algebra-simps round-up*)

lemma *float-up-correct*: *real-of-float* (*float-up e f*) − *real-of-float f* ∈ {*0*..*2 powr* −*e*}
by *transfer* (*rule round-up-correct*)

lift-definition *float-down* :: *int* ⇒ *float* ⇒ *float* **is** *round-down* **by** *simp*
declare *float-down.rep-eq*[*simp*]

lemma *round-down-correct*: *f* − (*round-down e f*) ∈ {*0*..*2 powr* −*e*}
unfolding *atLeastAtMost-iff*
proof
have *f* − *round-down e f* ≤ *round-up e f* − *round-down e f*
using *round-up* **by** *simp*
also have ... ≤ *2 powr* −*e*
using *round-up-diff-round-down* **by** *simp*

```
  finally show f − round-down e f ≤ 2 powr − (real-of-int e)
    by simp
qed (simp add: algebra-simps round-down)

lemma float-down-correct: real-of-float f − real-of-float (float-down e f) ∈ {0..2
powr −e}
  by transfer (rule round-down-correct)

context
begin

qualified lemma compute-float-down[code]:
  float-down p (Float m e) =
    (if p + e < 0 then Float (div-twopow m (nat (−(p + e)))) (−p) else Float m e)
proof (cases p + e < 0)
  case True
  then have real-of-int ((2::int) ^ nat (−(p + e))) = 2 powr (−(p + e))
    using powr-realpow[of 2 nat (−(p + e))] by simp
  also have ... = 1 / 2 powr p / 2 powr e
    unfolding powr-minus-divide of-int-minus by (simp add: powr-add)
  finally show ?thesis
    using ‹p + e < 0›
    apply transfer
   apply (simp add: round-down-def field-simps flip: floor-divide-of-int-eq powr-add)
    apply (metis (no-types, opaque-lifting) Float.rep-eq
      add.inverse-inverse compute-real-of-float diff-minus-eq-add
      floor-divide-of-int-eq int-of-reals(1) linorder-not-le
      minus-add-distrib of-int-eq-numeral-power-cancel-iff)
    done
next
  case False
  then have r: real-of-int e + real-of-int p = real (nat (e + p))
    by simp
  have r: ⌊(m * 2 powr e) * 2 powr real-of-int p⌋ = (m * 2 powr e) * 2 powr
real-of-int p
    by (auto intro: exI[where x=m*2^nat (e+p)]
        simp add: ac-simps powr-add[symmetric] r powr-realpow)
  with ‹¬ p + e < 0› show ?thesis
    by transfer (auto simp add: round-down-def field-simps powr-add powr-minus)
qed

lemma abs-round-down-le: |f − (round-down e f)| ≤ 2 powr −e
  using round-down-correct[of f e] by simp

lemma abs-round-up-le: |f − (round-up e f)| ≤ 2 powr −e
  using round-up-correct[of e f] by simp

lemma round-down-nonneg: 0 ≤ s ⟹ 0 ≤ round-down p s
  by (auto simp: round-down-def)
```

lemma *ceil-divide-floor-conv*:
assumes $b \neq 0$
shows $\lceil real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \rceil =$
$(if\ b\ dvd\ a\ then\ a\ div\ b\ else\ \lfloor real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \rfloor + 1)$
proof (*cases b dvd a*)
case *True*
then show *?thesis*
by (*simp add*: *ceiling-def floor-divide-of-int-eq dvd-neg-div*
flip: *of-int-minus divide-minus-left*)
next
case *False*
then have $a\ mod\ b \neq 0$
by *auto*
then have *ne*: $real\text{-}of\text{-}int\ (a\ mod\ b)\ /\ real\text{-}of\text{-}int\ b \neq 0$
using ‹$b \neq 0$› **by** *auto*
have $\lceil real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \rceil = \lfloor real\text{-}of\text{-}int\ a\ /\ real\text{-}of\text{-}int\ b \rfloor + 1$
by (*metis add-cancel-left-right ceiling-altdef floor-divide-of-int-eq ne of-int-div-aux*)
then show *?thesis*
using ‹$\neg$ *b dvd a*› **by** *simp*
qed

qualified lemma *compute-float-up*[*code*]: $float\text{-}up\ p\ x = -\ float\text{-}down\ p\ (-x)$
by *transfer* (*simp add*: *round-down-uminus-eq*)

end

lemma *bitlen-Float*:
fixes *m e*
defines [*THEN meta-eq-to-obj-eq*]: $f \equiv Float\ m\ e$
shows $bitlen\ |mantissa\ f| + exponent\ f = (if\ m = 0\ then\ 0\ else\ bitlen\ |m| + e)$
proof (*cases m = 0*)
case *True*
then show *?thesis* **by** (*simp add*: *f-def bitlen-alt-def*)
next
case *False*
then have $f \neq 0$
unfolding *real-of-float-eq* **by** (*simp add*: *f-def*)
then have $mantissa\ f \neq 0$
by (*simp add*: *mantissa-eq-zero-iff*)
moreover
obtain *i* **where** $m = mantissa\ f * 2\ \widehat{}\ i\ e = exponent\ f - int\ i$
by (*rule f-def*[*THEN denormalize-shift*, *OF* ‹$f \neq 0$›])
ultimately show *?thesis* **by** (*simp add*: *abs-mult*)
qed

lemma *float-gt1-scale*:
assumes $1 \leq Float\ m\ e$

```
  shows 0 ≤ e + (bitlen m − 1)
proof −
  have 0 < Float m e using assms by auto
  then have 0 < m using powr-gt-zero[of 2 e]
    by (auto simp: zero-less-mult-iff)
  then have m ≠ 0 by auto
  show ?thesis
  proof (cases 0 ≤ e)
    case True
    then show ?thesis
      using ‹0 < m› by (simp add: bitlen-alt-def)
  next
    case False
    have (1::int) < 2 by simp
    let ?S = 2^(nat (−e))
    have inverse (2 ^ nat (− e)) = 2 powr e
      using assms False powr-realpow[of 2 nat (−e)]
      by (auto simp: powr-minus field-simps)
    then have 1 ≤ real-of-int m * inverse ?S
      using assms False powr-realpow[of 2 nat (−e)]
      by (auto simp: powr-minus)
    then have 1 * ?S ≤ real-of-int m * inverse ?S * ?S
      by (rule mult-right-mono) auto
    then have ?S ≤ real-of-int m
      unfolding mult.assoc by auto
    then have ?S ≤ m
      unfolding of-int-le-iff[symmetric] by auto
    from this bitlen-bounds[OF ‹0 < m›, THEN conjunct2]
    have nat (−e) < (nat (bitlen m))
      unfolding power-strict-increasing-iff[OF ‹1 < 2›, symmetric]
      by (rule order-le-less-trans)
    then have −e < bitlen m
      using False by auto
    then show ?thesis
      by auto
  qed
qed
```

43.9 Truncating Real Numbers

```
definition truncate-down::nat ⇒ real ⇒ real
  where truncate-down prec x = round-down (prec − ⌊log 2 |x|⌋) x

lemma truncate-down: truncate-down prec x ≤ x
  using round-down by (simp add: truncate-down-def)

lemma truncate-down-le: x ≤ y ⟹ truncate-down prec x ≤ y
  by (rule order-trans[OF truncate-down])
```

lemma *truncate-down-zero*[*simp*]: *truncate-down prec 0 = 0*
by (*simp add*: *truncate-down-def*)

lemma *truncate-down-float*[*simp*]: *truncate-down p x ∈ float*
by (*auto simp*: *truncate-down-def*)

definition *truncate-up*::*nat ⇒ real ⇒ real*
where *truncate-up prec x = round-up* (*prec − ⌊log 2 |x|⌋*) *x*

lemma *truncate-up*: *x ≤ truncate-up prec x*
using *round-up* **by** (*simp add*: *truncate-up-def*)

lemma *truncate-up-le*: *x ≤ y ⟹ x ≤ truncate-up prec y*
by (*rule order-trans*[*OF - truncate-up*])

lemma *truncate-up-zero*[*simp*]: *truncate-up prec 0 = 0*
by (*simp add*: *truncate-up-def*)

lemma *truncate-up-uminus-eq*: *truncate-up prec* (*−x*) = *− truncate-down prec x*
and *truncate-down-uminus-eq*: *truncate-down prec* (*−x*) = *− truncate-up prec x*
by (*auto simp*: *truncate-up-def round-up-def truncate-down-def round-down-def ceiling-def*)

lemma *truncate-up-float*[*simp*]: *truncate-up p x ∈ float*
by (*auto simp*: *truncate-up-def*)

lemma *mult-powr-eq*: *0 < b ⟹ b ≠ 1 ⟹ 0 < x ⟹ x * b powr y = b powr* (*y + log b x*)
by (*simp-all add*: *powr-add*)

lemma *truncate-down-pos*:
assumes *x > 0*
shows *truncate-down p x > 0*
proof −
have *0 ≤ log 2 x − real-of-int ⌊log 2 x⌋*
by (*simp add*: *algebra-simps*)
moreover have *0 ≤ real p − real-of-int ⌊log 2 x⌋ + log 2 x*
by *linarith*
ultimately show *?thesis*
using *assms*
by (*auto simp*: *truncate-down-def round-down-def mult-powr-eq*
intro!: *ge-one-powr-ge-zero mult-pos-pos*)
qed

lemma *truncate-down-nonneg*: *0 ≤ y ⟹ 0 ≤ truncate-down prec y*
by (*auto simp*: *truncate-down-def round-down-def*)

lemma *truncate-down-ge1*: *1 ≤ x ⟹ 1 ≤ truncate-down p x*
apply (*auto simp*: *truncate-down-def algebra-simps intro*!: *round-down-ge1*)

apply *linarith*
done

lemma *truncate-up-nonpos*: $x \leq 0 \Longrightarrow$ *truncate-up prec* $x \leq 0$
by (*auto simp*: *truncate-up-def round-up-def intro*!: *mult-nonpos-nonneg*)

lemma *truncate-up-le1*:
assumes $x \leq 1$
shows *truncate-up p* $x \leq 1$
proof −
consider $x \leq 0 \mid x > 0$
by *arith*
then show *?thesis*
proof *cases*
case *1*
with *truncate-up-nonpos*[*OF this*, *of p*] **show** *?thesis*
by *simp*
next
case *2*
then have *le*: $\lfloor log\ 2\ |x| \rfloor \leq 0$
using *assms* **by** (*auto simp*: *log-less-iff*)
from *assms* **have** $0 \leq int\ p$ **by** *simp*
from *add-mono*[*OF this le*]
show *?thesis*
using *assms* **by** (*simp add*: *truncate-up-def round-up-le1 add-mono*)
qed
qed

lemma *truncate-down-shift-int*:
truncate-down p (*x* ∗ *2 powr real-of-int k*) = *truncate-down p x* ∗ *2 powr k*
by (*cases x* = *0*)
(*simp-all add*: *algebra-simps abs-mult log-mult truncate-down-def*
round-down-shift[*of - - k*, *simplified*])

lemma *truncate-down-shift-nat*: *truncate-down p* (*x* ∗ *2 powr real k*) = *truncate-down p x* ∗ *2 powr k*
by (*metis of-int-of-nat-eq truncate-down-shift-int*)

lemma *truncate-up-shift-int*: *truncate-up p* (*x* ∗ *2 powr real-of-int k*) = *truncate-up p x* ∗ *2 powr k*
by (*cases x* = *0*)
(*simp-all add*: *algebra-simps abs-mult log-mult truncate-up-def*
round-up-shift[*of - - k*, *simplified*])

lemma *truncate-up-shift-nat*: *truncate-up p* (*x* ∗ *2 powr real k*) = *truncate-up p x* ∗ *2 powr k*
by (*metis of-int-of-nat-eq truncate-up-shift-int*)

43.10 Truncating Floats

lift-definition *float-round-up* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-up*
by (*simp add*: *truncate-up-def*)

lemma *float-round-up*: *real-of-float x* ≤ *real-of-float* (*float-round-up prec x*)
using *truncate-up* **by** *transfer simp*

lemma *float-round-up-zero*[*simp*]: *float-round-up prec 0* = *0*
by *transfer simp*

lift-definition *float-round-down* :: *nat* ⇒ *float* ⇒ *float* **is** *truncate-down*
by (*simp add*: *truncate-down-def*)

lemma *float-round-down*: *real-of-float* (*float-round-down prec x*) ≤ *real-of-float x*
using *truncate-down* **by** *transfer simp*

lemma *float-round-down-zero*[*simp*]: *float-round-down prec 0* = *0*
by *transfer simp*

lemmas *float-round-up-le* = *order-trans*[*OF - float-round-up*]
and *float-round-down-le* = *order-trans*[*OF float-round-down*]

lemma *minus-float-round-up-eq*: − *float-round-up prec x* = *float-round-down prec* (− *x*)
and *minus-float-round-down-eq*: − *float-round-down prec x* = *float-round-up prec* (− *x*)
by (*transfer*; *simp add*: *truncate-down-uminus-eq truncate-up-uminus-eq*)+

context
begin

qualified lemma *compute-float-round-down*[*code*]:
float-round-down prec (*Float m e*) =
(*let d* = *bitlen* |*m*| − *int prec* − *1 in*
if 0 < *d then Float* (*div-twopow m* (*nat d*)) (*e* + *d*)
else Float m e)
using *Float.compute-float-down*[*of Suc prec* − *bitlen* |*m*| − *e m e*, *symmetric*]
by *transfer*
(*simp add*: *field-simps abs-mult log-mult bitlen-alt-def truncate-down-def*
cong del: *if-weak-cong*)

qualified lemma *compute-float-round-up*[*code*]:
float-round-up prec x = − *float-round-down prec* (−*x*)
by *transfer* (*simp add*: *truncate-down-uminus-eq*)

end

lemma *truncate-up-nonneg-mono*:
assumes *0* ≤ *x x* ≤ *y*

```
  shows truncate-up prec x ≤ truncate-up prec y
proof −
  consider ⌊log 2 x⌋ = ⌊log 2 y⌋ | ⌊log 2 x⌋ ≠ ⌊log 2 y⌋ 0 < x | x ≤ 0
    by arith
  then show ?thesis
  proof cases
    case 1
    then show ?thesis
      using assms
      by (auto simp: truncate-up-def round-up-def intro!: ceiling-mono)
  next
    case 2
    from assms ‹0 < x› have log 2 x ≤ log 2 y
      by auto
    with ‹⌊log 2 x⌋ ≠ ⌊log 2 y⌋›
    have logless: log 2 x < log 2 y
      by linarith
    have flogless: ⌊log 2 x⌋ < ⌊log 2 y⌋
      using ‹⌊log 2 x⌋ ≠ ⌊log 2 y⌋› ‹log 2 x ≤ log 2 y› by linarith
    have truncate-up prec x =
     real-of-int ⌈x * 2 powr real-of-int (int prec − ⌊log 2 x⌋ )⌉ * 2 powr − real-of-int
(int prec − ⌊log 2 x⌋)
      using assms by (simp add: truncate-up-def round-up-def)
    also have ⌈x * 2 powr real-of-int (int prec − ⌊log 2 x⌋)⌉ ≤ (2 ^ (Suc prec))
    proof (simp only: ceiling-le-iff)
      have x * 2 powr real-of-int (int prec − ⌊log 2 x⌋) ≤
        x * (2 powr real (Suc prec) / (2 powr log 2 x))
        using real-of-int-floor-add-one-ge[of log 2 x] assms
        by (auto simp: algebra-simps simp flip: powr-diff intro!: mult-left-mono)
     then show x * 2 powr real-of-int (int prec − ⌊log 2 x⌋) ≤ real-of-int ((2::int)
^ (Suc prec))
        using ‹0 < x› by (simp add: powr-realpow powr-add)
    qed
    then have real-of-int ⌈x * 2 powr real-of-int (int prec − ⌊log 2 x⌋)⌉ ≤ 2 powr
int (Suc prec)
      by (auto simp: powr-realpow powr-add)
        (metis power-Suc of-int-le-numeral-power-cancel-iff)
    also
     have 2 powr − real-of-int (int prec − ⌊log 2 x⌋) ≤ 2 powr − real-of-int (int
prec − ⌊log 2 y⌋ + 1)
      using logless flogless by (auto intro!: floor-mono)
    also have 2 powr real-of-int (int (Suc prec)) ≤
        2 powr (log 2 y + real-of-int (int prec − ⌊log 2 y⌋ + 1))
      using assms ‹0 < x›
      by (auto simp: algebra-simps)
    finally have truncate-up prec x ≤
      2 powr (log 2 y + real-of-int (int prec − ⌊log 2 y⌋ + 1)) * 2 powr − real-of-int
(int prec − ⌊log 2 y⌋ + 1)
      by simp
```

```
    also have ... = 2 powr (log 2 y + real-of-int (int prec − ⌊log 2 y⌋) − real-of-int
(int prec − ⌊log 2 y⌋))
      by (subst powr-add[symmetric]) simp
    also have ... = y
      using ‹0 < x› assms
      by (simp add: powr-add)
    also have ... ≤ truncate-up prec y
      by (rule truncate-up)
    finally show ?thesis .
  next
    case 3
    then show ?thesis
      using assms
      by (auto intro!: truncate-up-le)
  qed
qed

lemma truncate-up-switch-sign-mono:
  assumes x ≤ 0 0 ≤ y
  shows truncate-up prec x ≤ truncate-up prec y
proof −
  note truncate-up-nonpos[OF ‹x ≤ 0›]
  also note truncate-up-le[OF ‹0 ≤ y›]
  finally show ?thesis .
qed

lemma truncate-down-switch-sign-mono:
  assumes x ≤ 0
    and 0 ≤ y
    and x ≤ y
  shows truncate-down prec x ≤ truncate-down prec y
proof −
  note truncate-down-le[OF ‹x ≤ 0›]
  also note truncate-down-nonneg[OF ‹0 ≤ y›]
  finally show ?thesis .
qed

lemma truncate-down-nonneg-mono:
  assumes 0 ≤ x x ≤ y
  shows truncate-down prec x ≤ truncate-down prec y
proof −
  consider x ≤ 0 | ⌊log 2 |x|⌋ = ⌊log 2 |y|⌋ |
    0 < x ⌊log 2 |x|⌋ ≠ ⌊log 2 |y|⌋
    by arith
  then show ?thesis
  proof cases
    case 1
    with assms have x = 0 0 ≤ y by simp-all
    then show ?thesis
```

```
    by (auto intro!: truncate-down-nonneg)
  next
    case 2
    then show ?thesis
      using assms
      by (auto simp: truncate-down-def round-down-def intro!: floor-mono)
  next
    case 3
    from ‹0 < x› have log 2 x ≤ log 2 y 0 < y 0 ≤ y
      using assms by auto
    with ‹⌊log 2 |x|⌋ ≠ ⌊log 2 |y|⌋›
    have logless: log 2 x < log 2 y and flogless: ⌊log 2 x⌋ < ⌊log 2 y⌋
      unfolding atomize-conj abs-of-pos[OF ‹0 < x›] abs-of-pos[OF ‹0 < y›]
      by (metis floor-less-cancel linorder-cases not-le)
    have 2 powr prec ≤ y * 2 powr real prec / (2 powr log 2 y)
      using ‹0 < y› by simp
    also have ... ≤ y * 2 powr real (Suc prec) / (2 powr (real-of-int ⌊log 2 y⌋ +
1))
      using ‹0 ≤ y› ‹0 ≤ x› assms(2)
      by (auto intro!: powr-mono divide-left-mono
          simp: of-nat-diff powr-add powr-diff)
    also have ... = y * 2 powr real (Suc prec) / (2 powr real-of-int ⌊log 2 y⌋ * 2)
      by (auto simp: powr-add)
    finally have (2 ^ prec) ≤ ⌊y * 2 powr real-of-int (int (Suc prec) − ⌊log 2 |y|⌋
− 1)⌋
      using ‹0 ≤ y›
      by (auto simp: powr-diff le-floor-iff powr-realpow powr-add)
    then have (2 ^ (prec)) * 2 powr − real-of-int (int prec − ⌊log 2 |y|⌋) ≤
truncate-down prec y
      by (auto simp: truncate-down-def round-down-def)
    moreover have x ≤ (2 ^ prec) * 2 powr − real-of-int (int prec − ⌊log 2 |y|⌋)
    proof −
      have x = 2 powr (log 2 |x|) using ‹0 < x› by simp
      also have ... ≤ (2 ^ (Suc prec )) * 2 powr − real-of-int (int prec − ⌊log 2
|x|⌋)
        using real-of-int-floor-add-one-ge[of log 2 |x|] ‹0 < x›
      by (auto simp flip: powr-realpow powr-add simp: algebra-simps powr-mult-base
le-powr-iff)
      also
      have 2 powr − real-of-int (int prec − ⌊log 2 |x|⌋) ≤ 2 powr − real-of-int (int
prec − ⌊log 2 |y|⌋ + 1)
        using logless flogless ‹x > 0› ‹y > 0›
        by (auto intro!: floor-mono)
      finally show ?thesis
        by (auto simp flip: powr-realpow simp: powr-diff assms)
    qed
    ultimately show ?thesis
      by (metis dual-order.trans truncate-down)
  qed
```

qed

lemma *truncate-down-eq-truncate-up*: *truncate-down p x* = − *truncate-up p* (−*x*)
and *truncate-up-eq-truncate-down*: *truncate-up p x* = − *truncate-down p* (−*x*)
by (*auto simp*: *truncate-up-uminus-eq truncate-down-uminus-eq*)

lemma *truncate-down-mono*: *x* ≤ *y* ⟹ *truncate-down p x* ≤ *truncate-down p y*
by (*smt* (*verit*) *truncate-down-nonneg-mono truncate-up-nonneg-mono truncate-up-uminus-eq*)

lemma *truncate-up-mono*: *x* ≤ *y* ⟹ *truncate-up p x* ≤ *truncate-up p y*
by (*simp add*: *truncate-up-eq-truncate-down truncate-down-mono*)

lemma *truncate-up-nonneg*: *0* ≤ *truncate-up p x* **if** *0* ≤ *x*
by (*simp add*: *that truncate-up-le*)

lemma *truncate-up-pos*: *0* < *truncate-up p x* **if** *0* < *x*
by (*meson less-le-trans that truncate-up*)

lemma *truncate-up-less-zero-iff*[*simp*]: *truncate-up p x* < *0* ⟷ *x* < *0*
by (*smt* (*verit*) *truncate-down-pos truncate-down-uminus-eq truncate-up-nonneg*)

lemma *truncate-up-nonneg-iff*[*simp*]: *truncate-up p x* ≥ *0* ⟷ *x* ≥ *0*
using *truncate-up-less-zero-iff*[*of p x*] *truncate-up-nonneg*[*of x*]
by *linarith*

lemma *truncate-down-less-zero-iff*[*simp*]: *truncate-down p x* < *0* ⟷ *x* < *0*
by (*metis le-less-trans not-less-iff-gr-or-eq truncate-down truncate-down-pos truncate-down-zero*)

lemma *truncate-down-nonneg-iff*[*simp*]: *truncate-down p x* ≥ *0* ⟷ *x* ≥ *0*
using *truncate-down-less-zero-iff*[*of p x*] *truncate-down-nonneg*[*of x p*]
by *linarith*

lemma *truncate-down-eq-zero-iff*[*simp*]: *truncate-down prec x* = *0* ⟷ *x* = *0*
by (*metis not-less-iff-gr-or-eq truncate-down-less-zero-iff truncate-down-pos truncate-down-zero*)

lemma *truncate-up-eq-zero-iff*[*simp*]: *truncate-up prec x* = *0* ⟷ *x* = *0*
by (*metis not-less-iff-gr-or-eq truncate-up-less-zero-iff truncate-up-pos truncate-up-zero*)

43.11 Approximation of positive rationals

lemma *div-mult-twopow-eq*: *a div* ((*2*::*nat*) ^ *n*) *div b* = *a div* (*b* ∗ *2* ^ *n*) **for** *a b* :: *nat*
by (*cases b* = *0*) (*simp-all add*: *div-mult2-eq*[*symmetric*] *ac-simps*)

lemma *real-div-nat-eq-floor-of-divide*: *a div b* = *real-of-int* ⌊*a* / *b*⌋ **for** *a b* :: *nat*
by (*simp add*: *floor-divide-of-nat-eq* [*of a b*])

definition *rat-precision prec x y* =
(*let d* = *bitlen x* − *bitlen y*
in int prec − *d* + (*if Float* (*abs x*) *0* < *Float* (*abs y*) *d then 1 else 0*))

lemma *floor-log-divide-eq*:
assumes *i* > *0 j* > *0 p* > *1*
shows ⌊*log p* (*i* / *j*)⌋ = *floor* (*log p i*) − *floor* (*log p j*) −
(*if i* ≥ *j* ∗ *p powr* (*floor* (*log p i*) − *floor* (*log p j*)) *then 0 else 1*)
proof −
let *?l* = *log p*
let *?fl* = λ*x*. *floor* (*?l x*)
have ⌊*?l* (*i* / *j*)⌋ = ⌊*?l i* − *?l j*⌋ **using** *assms*
by (*auto simp*: *log-divide*)
also have ... = *floor* (*real-of-int* (*?fl i* − *?fl j*) + (*?l i* − *?fl i* − (*?l j* − *?fl j*)))
(**is** - = *floor* (- + *?r*))
by (*simp add*: *algebra-simps*)
also note *floor-add2*
also note ‹*p* > *1*›
note *powr* = *powr-le-cancel-iff*[*symmetric*, *OF* ‹*1* < *p*›, *THEN iffD2*]
note *powr-strict* = *powr-less-cancel-iff*[*symmetric*, *OF* ‹*1* < *p*›, *THEN iffD2*]
have *floor ?r* = (*if i* ≥ *j* ∗ *p powr* (*?fl i* − *?fl j*) *then 0 else* −*1*) (**is** - = *?if*)
using *assms*
apply *simp*
by (*smt* (*verit*, *ccfv-SIG*) *floor-less-iff floor-uminus-of-int le-log-iff mult-powr-eq*
of-int-1 real-of-int-floor-add-one-gt zero-le-floor)
finally
show *?thesis* **by** *simp*
qed

lemma *truncate-down-rat-precision*:
truncate-down prec (*real x* / *real y*) = *round-down* (*rat-precision prec x y*) (*real x* / *real y*)
and *truncate-up-rat-precision*:
truncate-up prec (*real x* / *real y*) = *round-up* (*rat-precision prec x y*) (*real x* / *real y*)
unfolding *truncate-down-def truncate-up-def rat-precision-def*
by (*cases x*; *cases y*) (*auto simp*: *floor-log-divide-eq algebra-simps bitlen-alt-def*)

lift-definition *lapprox-posrat* :: *nat* ⇒ *nat* ⇒ *nat* ⇒ *float*
is λ*prec* (*x*::*nat*) (*y*::*nat*). *truncate-down prec* (*x* / *y*)
by *simp*

context
begin

qualified lemma *compute-lapprox-posrat*[*code*]:
lapprox-posrat prec x y =
(*let*
l = *rat-precision prec x y*;

```
    d = if 0 ≤ l then x * 2^nat l div y else x div 2^nat (− l) div y
   in normfloat (Float d (− l)))
   unfolding div-mult-twopow-eq
   by transfer
    (simp add: round-down-def powr-int real-div-nat-eq-floor-of-divide field-simps
Let-def
      truncate-down-rat-precision del: two-powr-minus-int-float)

end

lift-definition rapprox-posrat :: nat ⇒ nat ⇒ nat ⇒ float
 is λprec (x::nat) (y::nat). truncate-up prec (x / y)
 by simp

context
begin

qualified lemma compute-rapprox-posrat[code]:
 fixes prec x y
 defines l ≡ rat-precision prec x y
 shows rapprox-posrat prec x y =
  (let
    l = l;
    (r, s) = if 0 ≤ l then (x * 2^nat l, y) else (x, y * 2^nat(−l));
    d = r div s;
    m = r mod s
   in normfloat (Float (d + (if m = 0 ∨ y = 0 then 0 else 1)) (− l)))
proof (cases y = 0)
 assume y = 0
 then show ?thesis by transfer simp
next
 assume y ≠ 0
 show ?thesis
 proof (cases 0 ≤ l)
  case True
  define x' where x' = x * 2 ^ nat l
  have int x * 2 ^ nat l = x'
   by (simp add: x'-def)
  moreover have real x * 2 powr l = real x'
   by (simp flip: powr-realpow add: ‹0 ≤ l› x'-def)
  ultimately show ?thesis
   using ceil-divide-floor-conv[of y x'] powr-realpow[of 2 nat l] ‹0 ≤ l› ‹y ≠ 0›
    l-def[symmetric, THEN meta-eq-to-obj-eq]
   apply transfer
   apply (auto simp add: round-up-def truncate-up-rat-precision)
   apply (metis floor-divide-of-int-eq of-int-of-nat-eq)
   done
 next
  case False
```

```
    define y' where y' = y * 2 ^ nat (- l)
    from ‹y ≠ 0› have y' ≠ 0 by (simp add: y'-def)
    have int y * 2 ^ nat (- l) = y'
      by (simp add: y'-def)
    moreover have real x * real-of-int (2::int) powr real-of-int l / real y = x /
real y'
    using ‹¬ 0 ≤ l› by (simp flip: powr-realpow add: powr-minus y'-def field-simps)
    ultimately show ?thesis
      using ceil-divide-floor-conv[of y' x] ‹¬ 0 ≤ l› ‹y' ≠ 0› ‹y ≠ 0›
        l-def[symmetric, THEN meta-eq-to-obj-eq]
      apply transfer
    apply (auto simp add: round-up-def ceil-divide-floor-conv truncate-up-rat-precision)
      apply (metis floor-divide-of-int-eq of-int-of-nat-eq)
      done
  qed
qed

end

lemma rat-precision-pos:
  assumes 0 ≤ x
    and 0 < y
    and 2 * x < y
  shows rat-precision n (int x) (int y) > 0
proof −
  have 0 < x ⟹ log 2 x + 1 = log 2 (2 * x)
    by (simp add: log-mult)
  then have bitlen (int x) < bitlen (int y)
    using assms
    by (simp add: bitlen-alt-def)
      (auto intro!: floor-mono simp add: one-add-floor)
  then show ?thesis
    using assms
    by (auto intro!: pos-add-strict simp add: field-simps rat-precision-def)
qed

lemma rapprox-posrat-less1:
  0 ≤ x ⟹ 0 < y ⟹ 2 * x < y ⟹ real-of-float (rapprox-posrat n x y) < 1
  by transfer (simp add: rat-precision-pos round-up-less1 truncate-up-rat-precision)

lift-definition lapprox-rat :: nat ⇒ int ⇒ int ⇒ float is
  λprec (x::int) (y::int). truncate-down prec (x / y)
  by simp

context
begin

qualified lemma compute-lapprox-rat[code]:
  lapprox-rat prec x y =
```

```
  (if y = 0 then 0
   else if 0 ≤ x then
    (if 0 < y then lapprox-posrat prec (nat x) (nat y)
     else − (rapprox-posrat prec (nat x) (nat (−y))))
     else
      (if 0 < y
       then − (rapprox-posrat prec (nat (−x)) (nat y))
       else lapprox-posrat prec (nat (−x)) (nat (−y))))
 by transfer (simp add: truncate-up-uminus-eq)

lift-definition rapprox-rat :: nat ⇒ int ⇒ int ⇒ float is
 λprec (x::int) (y::int). truncate-up prec (x / y)
 by simp

lemma rapprox-rat = rapprox-posrat
 by transfer auto

lemma lapprox-rat = lapprox-posrat
 by transfer auto

qualified lemma compute-rapprox-rat[code]:
 rapprox-rat prec x y = − lapprox-rat prec (−x) y
 by transfer (simp add: truncate-down-uminus-eq)

qualified lemma compute-truncate-down[code]:
 truncate-down p (Ratreal r) = (let (a, b) = quotient-of r in lapprox-rat p a b)
 by transfer (auto split: prod.split simp: of-rat-divide dest!: quotient-of-div)

qualified lemma compute-truncate-up[code]:
 truncate-up p (Ratreal r) = (let (a, b) = quotient-of r in rapprox-rat p a b)
 by transfer (auto split: prod.split simp:  of-rat-divide dest!: quotient-of-div)

end
```

43.12 Division

```
definition real-divl prec a b = truncate-down prec (a / b)

definition real-divr prec a b = truncate-up prec (a / b)

lift-definition float-divl :: nat ⇒ float ⇒ float ⇒ float is real-divl
 by (simp add: real-divl-def)

context
begin

qualified lemma compute-float-divl[code]:
 float-divl prec (Float m1 s1) (Float m2 s2) = lapprox-rat prec m1 m2 * Float 1
(s1 − s2)
```

apply *transfer*
unfolding *real-divl-def of-int-1 mult-1 truncate-down-shift-int*[*symmetric*]
apply (*simp add*: *powr-diff powr-minus*)
done

lift-definition *float-divr* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *real-divr*
by (*simp add*: *real-divr-def*)

qualified lemma *compute-float-divr*[*code*]:
float-divr prec x y = − *float-divl prec* (−*x*) *y*
by *transfer* (*simp add*: *real-divr-def real-divl-def truncate-down-uminus-eq*)

end

43.13 Approximate Addition

definition *plus-down prec x y* = *truncate-down prec* (*x* + *y*)

definition *plus-up prec x y* = *truncate-up prec* (*x* + *y*)

lemma *float-plus-down-float*[*intro*, *simp*]: *x* ∈ *float* ⟹ *y* ∈ *float* ⟹ *plus-down p x y* ∈ *float*
by (*simp add*: *plus-down-def*)

lemma *float-plus-up-float*[*intro*, *simp*]: *x* ∈ *float* ⟹ *y* ∈ *float* ⟹ *plus-up p x y* ∈ *float*
by (*simp add*: *plus-up-def*)

lift-definition *float-plus-down* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-down* **..**

lift-definition *float-plus-up* :: *nat* ⇒ *float* ⇒ *float* ⇒ *float* **is** *plus-up* **..**

lemma *plus-down*: *plus-down prec x y* ≤ *x* + *y*
and *plus-up*: *x* + *y* ≤ *plus-up prec x y*
by (*auto simp*: *plus-down-def truncate-down plus-up-def truncate-up*)

lemma *float-plus-down*: *real-of-float* (*float-plus-down prec x y*) ≤ *x* + *y*
and *float-plus-up*: *x* + *y* ≤ *real-of-float* (*float-plus-up prec x y*)
by (*transfer*; *rule plus-down plus-up*)+

lemmas *plus-down-le* = *order-trans*[*OF plus-down*]
and *plus-up-le* = *order-trans*[*OF* - *plus-up*]
and *float-plus-down-le* = *order-trans*[*OF float-plus-down*]
and *float-plus-up-le* = *order-trans*[*OF* - *float-plus-up*]

lemma *compute-plus-up*[*code*]: *plus-up p x y* = − *plus-down p* (−*x*) (−*y*)
using *truncate-down-uminus-eq*[*of p x* + *y*]
by (*auto simp*: *plus-down-def plus-up-def*)

lemma *truncate-down-log2-eqI*:
assumes $\lfloor log\ 2\ |x| \rfloor = \lfloor log\ 2\ |y| \rfloor$
assumes $\lfloor x * 2\ powr\ (p - \lfloor log\ 2\ |x| \rfloor) \rfloor = \lfloor y * 2\ powr\ (p - \lfloor log\ 2\ |x| \rfloor) \rfloor$
shows *truncate-down p x = truncate-down p y*
using *assms* **by** (*auto simp*: *truncate-down-def round-down-def*)

lemma *sum-neq-zeroI*:
$|a| \geq k \Longrightarrow |b| < k \Longrightarrow a + b \neq 0$
$|a| > k \Longrightarrow |b| \leq k \Longrightarrow a + b \neq 0$
for *a k* :: *real*
by *auto*

lemma *abs-real-le-2-powr-bitlen*[*simp*]: $|real\text{-}of\text{-}int\ m2| < 2\ powr\ real\text{-}of\text{-}int\ (bitlen\ |m2|)$
proof (*cases m2 = 0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have $|m2| < 2\ \widehat{}\ nat\ (bitlen\ |m2|)$
using *bitlen-bounds*[*of* $|m2|$]
by (*auto simp*: *powr-add bitlen-nonneg*)
then show *?thesis*
by (*metis bitlen-nonneg powr-int of-int-abs of-int-less-numeral-power-cancel-iff zero-less-numeral*)
qed

lemma *floor-sum-times-2-powr-sgn-eq*:
fixes *ai p q* :: *int*
and *a b* :: *real*
assumes *a * 2 powr p = ai*
and *b-le-1*: $|b * 2\ powr\ (p + 1)| \leq 1$
and *leqp*: $q \leq p$
shows $\lfloor (a + b) * 2\ powr\ q \rfloor = \lfloor (2 * ai + sgn\ b) * 2\ powr\ (q - p - 1) \rfloor$
proof −
consider $b = 0 \mid b > 0 \mid b < 0$ **by** *arith*
then show *?thesis*
proof *cases*
case *1*
then show *?thesis*
by (*simp flip*: *assms*(*1*) *powr-add add*: *algebra-simps powr-mult-base*)
next
case *2*
then have $b * 2\ powr\ p < |b * 2\ powr\ (p + 1)|$
by *simp*
also note *b-le-1*
finally have *b-less-1*: $b * 2\ powr\ real\text{-}of\text{-}int\ p < 1$.

from *b-less-1* ‹$b > 0$› **have** *floor-eq*: $\lfloor b * 2\ powr\ real\text{-}of\text{-}int\ p \rfloor = 0\ \lfloor sgn\ b\ /$

2⌋ = 0
by (*simp-all add*: *floor-eq-iff*)

have *⌊(a + b) * 2 powr q⌋ = ⌊(a + b) * 2 powr p * 2 powr (q − p)⌋*
by (*simp add*: *algebra-simps flip*: *powr-realpow powr-add*)
also have *... = ⌊(ai + b * 2 powr p) * 2 powr (q − p)⌋*
by (*simp add*: *assms algebra-simps*)
also have *... = ⌊(ai + b * 2 powr p) / real-of-int ((2::int) ^ nat (p − q))⌋*
using *assms*
by (*simp add*: *algebra-simps divide-powr-uminus flip*: *powr-realpow powr-add*)
also have *... = ⌊ai / real-of-int ((2::int) ^ nat (p − q))⌋*
by (*simp del*: *of-int-power add*: *floor-divide-real-eq-div floor-eq*)
finally have *⌊(a + b) * 2 powr real-of-int q⌋ = ⌊real-of-int ai / real-of-int ((2::int) ^ nat (p − q))⌋* **.**
moreover
have *⌊(2 * ai + (sgn b)) * 2 powr (real-of-int (q − p) − 1)⌋ =*
⌊real-of-int ai / real-of-int ((2::int) ^ nat (p − q))⌋
proof −
have *⌊(2 * ai + sgn b) * 2 powr (real-of-int (q − p) − 1)⌋ = ⌊(ai + sgn b / 2) * 2 powr (q − p)⌋*
by (*subst powr-diff*) (*simp add*: *field-simps*)
also have *... = ⌊(ai + sgn b / 2) / real-of-int ((2::int) ^ nat (p − q))⌋*
using *leqp* **by** (*simp flip*: *powr-realpow add*: *powr-diff*)
also have *... = ⌊ai / real-of-int ((2::int) ^ nat (p − q))⌋*
by (*simp del*: *of-int-power add*: *floor-divide-real-eq-div floor-eq*)
finally show *?thesis* **.**
qed
ultimately show *?thesis* **by** *simp*
next
case *3*
then have *floor-eq*: *⌊b * 2 powr (real-of-int p + 1)⌋ = −1*
using *b-le-1*
by (*auto simp*: *floor-eq-iff algebra-simps pos-divide-le-eq*[*symmetric*] *abs-if divide-powr-uminus*
intro!: *mult-neg-pos split*: *if-split-asm*)
have *⌊(a + b) * 2 powr q⌋ = ⌊(2∗a + 2∗b) * 2 powr p * 2 powr (q − p − 1)⌋*
by (*simp add*: *algebra-simps powr-mult-base flip*: *powr-realpow powr-add*)
also have *... = ⌊(2 * (a * 2 powr p) + 2 * b * 2 powr p) * 2 powr (q − p − 1)⌋*
by (*simp add*: *algebra-simps*)
also have *... = ⌊(2 * ai + b * 2 powr (p + 1)) / 2 powr (1 − q + p)⌋*
using *assms* **by** (*simp add*: *algebra-simps powr-mult-base divide-powr-uminus*)
also have *... = ⌊(2 * ai + b * 2 powr (p + 1)) / real-of-int ((2::int) ^ nat (p − q + 1))⌋*
using *assms* **by** (*simp add*: *algebra-simps flip*: *powr-realpow*)
also have *... = ⌊(2 * ai − 1) / real-of-int ((2::int) ^ nat (p − q + 1))⌋*
using ‹*b < 0*› *assms*
by (*simp add*: *floor-divide-of-int-eq floor-eq floor-divide-real-eq-div*
del: *of-int-mult of-int-power of-int-diff*)

```
    also have ... = ⌊(2 * ai - 1) * 2 powr (q - p - 1)⌋
    using assms by (simp add: algebra-simps divide-powr-uminus flip: powr-realpow)
    finally show ?thesis
      using ‹b < 0› by simp
  qed
qed

lemma log2-abs-int-add-less-half-sgn-eq:
  fixes ai :: int
    and b :: real
  assumes |b| ≤ 1/2
    and ai ≠ 0
  shows ⌊log 2 |real-of-int ai + b|⌋ = ⌊log 2 |ai + sgn b / 2|⌋
proof (cases b = 0)
  case True
  then show ?thesis by simp
next
  case False
  define k where k = ⌊log 2 |ai|⌋
  then have ⌊log 2 |ai|⌋ = k
    by simp
  then have k: 2 powr k ≤ |ai| |ai| < 2 powr (k + 1)
    by (simp-all add: floor-log-eq-powr-iff ‹ai ≠ 0›)
  have k ≥ 0
    using assms by (auto simp: k-def)
  define r where r = |ai| - 2 ^ nat k
  have r: 0 ≤ r r < 2 powr k
    using ‹k ≥ 0› k
    by (auto simp: r-def k-def algebra-simps powr-add abs-if powr-int)
  then have r ≤ (2::int) ^ nat k - 1
    using ‹k ≥ 0› by (auto simp: powr-int)
  from this[simplified of-int-le-iff[symmetric]] ‹0 ≤ k›
  have r-le: r ≤ 2 powr k - 1
    by (auto simp: algebra-simps powr-int)
      (metis of-int-1 of-int-add of-int-le-numeral-power-cancel-iff)

  have |ai| = 2 powr k + r
    using ‹k ≥ 0› by (auto simp: k-def r-def simp flip: powr-realpow)

  have pos: |b| < 1 ⟹ 0 < 2 powr k + (r + b) for b :: real
    using ‹0 ≤ k› ‹ai ≠ 0›
    by (auto simp add: r-def powr-realpow[symmetric] abs-if sgn-if algebra-simps
      split: if-split-asm)
  have less: |sgn ai * b| < 1
    and less': |sgn (sgn ai * b) / 2| < 1
    using ‹|b| ≤ -› by (auto simp: abs-if sgn-if split: if-split-asm)

  have floor-eq: ⋀b::real. |b| ≤ 1 / 2 ⟹
      ⌊log 2 (1 + (r + b) / 2 powr k)⌋ = (if r = 0 ∧ b < 0 then -1 else 0)
```

```
    using ‹k ≥ 0› r r-le
    by (auto simp: floor-log-eq-powr-iff powr-minus-divide field-simps sgn-if)

  from ‹real-of-int |ai| = -› have |ai + b| = 2 powr k + (r + sgn ai * b)
    using ‹|b| ≤ -› ‹0 ≤ k› r
    by (auto simp add: sgn-if abs-if)
  also have ⌊log 2 ...⌋ = ⌊log 2 (2 powr k + r + sgn (sgn ai * b) / 2)⌋
  proof -
    have 2 powr k + (r + (sgn ai) * b) = 2 powr k * (1 + (r + sgn ai * b) / 2
powr k)
      by (simp add: field-simps)
    also have ⌊log 2 ...⌋ = k + ⌊log 2 (1 + (r + sgn ai * b) / 2 powr k)⌋
      using pos[OF less]
      by (subst log-mult) (simp-all add: log-mult powr-mult field-simps)
    also
    let ?if = if r = 0 ∧ sgn ai * b < 0 then −1 else 0
    have ⌊log 2 (1 + (r + sgn ai * b) / 2 powr k)⌋ = ?if
      using ‹|b| ≤ -›
      by (intro floor-eq) (auto simp: abs-mult sgn-if)
    also
    have ... = ⌊log 2 (1 + (r + sgn (sgn ai * b) / 2) / 2 powr k)⌋
      by (subst floor-eq) (auto simp: sgn-if)
    also have k + ... = ⌊log 2 (2 powr k * (1 + (r + sgn (sgn ai * b) / 2) / 2
powr k))⌋
      unfolding int-add-floor
      using pos[OF less'] ‹|b| ≤ -›
      by (simp add: field-simps add-log-eq-powr del: floor-add2)
    also have 2 powr k * (1 + (r + sgn (sgn ai * b) / 2) / 2 powr k) =
        2 powr k + r + sgn (sgn ai * b) / 2
      by (simp add: sgn-if field-simps)
    finally show ?thesis .
  qed
  also have 2 powr k + r + sgn (sgn ai * b) / 2 = |ai + sgn b / 2|
    unfolding ‹real-of-int |ai| = -›[symmetric] using ‹ai ≠ 0›
    by (auto simp: abs-if sgn-if algebra-simps)
  finally show ?thesis .
qed

context
begin

qualified lemma compute-far-float-plus-down:
  fixes m1 e1 m2 e2 :: int
    and p :: nat
  defines k1 ≡ Suc p − nat (bitlen |m1|)
  assumes H: bitlen |m2| ≤ e1 − e2 − k1 − 2 m1 ≠ 0 m2 ≠ 0 e1 ≥ e2
  shows float-plus-down p (Float m1 e1) (Float m2 e2) =
    float-round-down p (Float (m1 * 2 ^ (Suc (Suc k1)) + sgn m2) (e1 − int k1
− 2))
```

proof −
let *?a* = *real-of-float* (*Float m1 e1*)
let *?b* = *real-of-float* (*Float m2 e2*)
let *?sum* = *?a* + *?b*
let *?shift* = *real-of-int e2* − *real-of-int e1* + *real k1* + *1*
let *?m1* = *m1* ∗ *2* ^ *Suc k1*
let *?m2* = *m2* ∗ *2 powr ?shift*
let *?m2'* = *sgn m2* / *2*
let *?e* = *e1* − *int k1* − *1*

have *sum-eq*: *?sum* = (*?m1* + *?m2*) ∗ *2 powr ?e*
by (*auto simp flip*: *powr-add powr-mult powr-realpow simp*: *powr-mult-base algebra-simps*)

have |*?m2*| ∗ *2* < *2 powr* (*bitlen* |*m2*| + *?shift* + *1*)
by (*auto simp*: *field-simps powr-add powr-mult-base powr-diff abs-mult*)
also have ... ≤ *2 powr 0*
using *H* **by** (*intro powr-mono*) *auto*
finally have *abs-m2-less-half*: |*?m2*| < *1* / *2*
by *simp*

then have |*real-of-int m2*| < *2 powr* −(*?shift* + *1*)
unfolding *powr-minus-divide* **by** (*auto simp*: *bitlen-alt-def field-simps powr-mult-base abs-mult*)
also have ... ≤ *2 powr real-of-int* (*e1* − *e2* − *2*)
by *simp*
finally have *b-less-quarter*: |*?b*| < *1/4* ∗ *2 powr real-of-int e1*
by (*simp add*: *powr-add field-simps powr-diff abs-mult*)
also have *1/4* < |*real-of-int m1*| / *2* **using** ‹*m1* ≠ *0*› **by** *simp*
finally have *b-less-half-a*: |*?b*| < *1/2* ∗ |*?a*|
by (*simp add*: *algebra-simps powr-mult-base abs-mult*)
then have *a-half-less-sum*: |*?a*| / *2* < |*?sum*|
by (*auto simp*: *field-simps abs-if split*: *if-split-asm*)

from *b-less-half-a* **have** |*?b*| < |*?a*| |*?b*| ≤ |*?a*|
by *simp-all*

have |*real-of-float* (*Float m1 e1*)| ≥ *1/4* ∗ *2 powr real-of-int e1*
using ‹*m1* ≠ *0*›
by (*auto simp*: *powr-add powr-int bitlen-nonneg divide-right-mono abs-mult*)
then have *?sum* ≠ *0* **using** *b-less-quarter*
by (*rule sum-neq-zeroI*)
then have *?m1* + *?m2* ≠ *0*
unfolding *sum-eq* **by** (*simp add*: *abs-mult zero-less-mult-iff*)

have |*real-of-int ?m1*| ≥ *2* ^ *Suc k1* |*?m2'*| < *2* ^ *Suc k1*
using ‹*m1* ≠ *0*› ‹*m2* ≠ *0*› **by** (*auto simp*: *sgn-if less-1-mult abs-mult simp del*: *power.simps*)
then have *sum'-nz*: *?m1* + *?m2'* ≠ *0*

by (*intro sum-neq-zeroI*)

have $\lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ m1\ e1) + real\text{-}of\text{-}float\ (Float\ m2\ e2)| \rfloor = \lfloor log\ 2\ |?m1 + ?m2| \rfloor + ?e$
using ‹$?m1 + ?m2 \neq 0$›
unfolding *floor-add*[*symmetric*] *sum-eq*
by (*simp add*: *abs-mult log-mult*) *linarith*
also have $\lfloor log\ 2\ |?m1 + ?m2| \rfloor = \lfloor log\ 2\ |?m1 + sgn\ (real\text{-}of\text{-}int\ m2 * 2\ powr\ ?shift) / 2| \rfloor$
using *abs-m2-less-half* ‹$m1 \neq 0$›
by (*intro log2-abs-int-add-less-half-sgn-eq*) (*auto simp*: *abs-mult*)
also have $sgn\ (real\text{-}of\text{-}int\ m2 * 2\ powr\ ?shift) = sgn\ m2$
by (*auto simp*: *sgn-if zero-less-mult-iff less-not-sym*)
also
have $|?m1 + ?m2'| * 2\ powr\ ?e = |?m1 * 2 + sgn\ m2| * 2\ powr\ (?e - 1)$
by (*auto simp*: *field-simps powr-minus*[*symmetric*] *powr-diff powr-mult-base*)
then have $\lfloor log\ 2\ |?m1 + ?m2'| \rfloor + ?e = \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ (?m1 * 2 + sgn\ m2)\ (?e - 1))| \rfloor$
using ‹$?m1 + ?m2' \neq 0$›
unfolding *floor-add-int*
by (*simp add*: *log-add-eq-powr abs-mult-pos del*: *floor-add2*)
finally
have $\lfloor log\ 2\ |?sum| \rfloor = \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ (?m1*2 + sgn\ m2)\ (?e - 1))| \rfloor$
.
then have *plus-down p* (*Float m1 e1*) (*Float m2 e2*) =
truncate-down p (*Float* (*?m1*2 + sgn m2*) (*?e − 1*))
unfolding *plus-down-def*
proof (*rule truncate-down-log2-eqI*)
let $?f = (int\ p - \lfloor log\ 2\ |real\text{-}of\text{-}float\ (Float\ m1\ e1) + real\text{-}of\text{-}float\ (Float\ m2\ e2)| \rfloor)$
let $?ai = m1 * 2\ \widehat{}\ (Suc\ k1)$
have $\lfloor (?a + ?b) * 2\ powr\ real\text{-}of\text{-}int\ ?f \rfloor = \lfloor (real\text{-}of\text{-}int\ (2 * ?ai) + sgn\ ?b) * 2\ powr\ real\text{-}of\text{-}int\ (?f - - ?e - 1) \rfloor$
proof (*rule floor-sum-times-2-powr-sgn-eq*)
show $?a * 2\ powr\ real\text{-}of\text{-}int\ (-?e) = real\text{-}of\text{-}int\ ?ai$
by (*simp add*: *powr-add powr-realpow*[*symmetric*] *powr-diff*)
show $|?b * 2\ powr\ real\text{-}of\text{-}int\ (-?e + 1)| \leq 1$
using *abs-m2-less-half*
by (*simp add*: *abs-mult powr-add*[*symmetric*] *algebra-simps powr-mult-base*)
next
have $e1 + \lfloor log\ 2\ |real\text{-}of\text{-}int\ m1| \rfloor - 1 = \lfloor log\ 2\ |?a| \rfloor - 1$
using ‹$m1 \neq 0$›
by (*simp add*: *int-add-floor algebra-simps log-mult abs-mult del*: *floor-add2*)
also have $\ldots \leq \lfloor log\ 2\ |?a + ?b| \rfloor$
using *a-half-less-sum* ‹$m1 \neq 0$› ‹$?sum \neq 0$›
unfolding *floor-diff-of-int*[*symmetric*]
by (*auto simp add*: *log-minus-eq-powr powr-minus-divide intro*!: *floor-mono*)
finally
have $int\ p - \lfloor log\ 2\ |?a + ?b| \rfloor \leq p - (bitlen\ |m1|) - e1 + 2$

```
      by (auto simp: algebra-simps bitlen-alt-def ‹m1 ≠ 0›)
    also have ... ≤ - ?e
      using bitlen-nonneg[of |m1|] by (simp add: k1-def)
    finally show ?f ≤ - ?e by simp
  qed
  also have sgn ?b = sgn m2
    using powr-gt-zero[of 2 e2]
    by (auto simp add: sgn-if zero-less-mult-iff simp del: powr-gt-zero)
  also have ⌊(real-of-int (2 * ?m1) + real-of-int (sgn m2)) * 2 powr real-of-int
(?f - - ?e - 1)⌋ =
      ⌊Float (?m1 * 2 + sgn m2) (?e - 1) * 2 powr ?f⌋
    by (simp flip: powr-add powr-realpow add: algebra-simps)
  finally
  show ⌊(?a + ?b) * 2 powr ?f⌋ = ⌊real-of-float (Float (?m1 * 2 + sgn m2) (?e
- 1)) * 2 powr ?f⌋ .
 qed
 then show ?thesis
  by transfer (simp add: plus-down-def ac-simps Let-def)
qed

lemma compute-float-plus-down-naive: float-plus-down p x y = float-round-down
p (x + y)
 by transfer (auto simp: plus-down-def)

qualified lemma compute-float-plus-down[code]:
 fixes p::nat and m1 e1 m2 e2::int
 shows float-plus-down p (Float m1 e1) (Float m2 e2) =
  (if m1 = 0 then float-round-down p (Float m2 e2)
  else if m2 = 0 then float-round-down p (Float m1 e1)
  else
    (if e1 ≥ e2 then
      (let k1 = Suc p - nat (bitlen |m1|) in
        if bitlen |m2| > e1 - e2 - k1 - 2
        then float-round-down p ((Float m1 e1) + (Float m2 e2))
         else float-round-down p (Float (m1 * 2 ^ (Suc (Suc k1)) + sgn m2) (e1
- int k1 - 2)))
    else float-plus-down p (Float m2 e2) (Float m1 e1)))
proof -
 {
  assume bitlen |m2| ≤ e1 - e2 - (Suc p - nat (bitlen |m1|)) - 2 m1 ≠ 0 m2
≠ 0 e1 ≥ e2
  note compute-far-float-plus-down[OF this]
 }
 then show ?thesis
  by transfer (simp add: Let-def plus-down-def ac-simps)
qed

qualified lemma compute-float-plus-up[code]: float-plus-up p x y = - float-plus-down
p (-x) (-y)
```

using *truncate-down-uminus-eq*[*of p x + y*]
by *transfer* (*simp add*: *plus-down-def plus-up-def ac-simps*)

lemma *mantissa-zero*: *mantissa 0 = 0*
by (*fact mantissa-0*)

qualified lemma *compute-float-less*[*code*]: *a < b ⟷ is-float-pos* (*float-plus-down 0 b* (*− a*))
using *truncate-down*[*of 0 b − a*] *truncate-down-pos*[*of b − a 0*]
by *transfer* (*auto simp*: *plus-down-def*)

qualified lemma *compute-float-le*[*code*]: *a ≤ b ⟷ is-float-nonneg* (*float-plus-down 0 b* (*− a*))
using *truncate-down*[*of 0 b − a*] *truncate-down-nonneg*[*of b − a 0*]
by *transfer* (*auto simp*: *plus-down-def*)

end

lemma *plus-down-mono*: *plus-down p a b ≤ plus-down p c d* **if** *a + b ≤ c + d*
by (*auto simp*: *plus-down-def intro*!: *truncate-down-mono that*)

lemma *plus-up-mono*: *plus-up p a b ≤ plus-up p c d* **if** *a + b ≤ c + d*
by (*auto simp*: *plus-up-def intro*!: *truncate-up-mono that*)

43.14 Approximate Multiplication

lemma *mult-mono-nonpos-nonneg*: *a * b ≤ c * d*
if *a ≤ c a ≤ 0 0 ≤ d d ≤ b* **for** *a b c d*::*'a*::*ordered-ring*
by (*meson dual-order.trans mult-left-mono-neg mult-right-mono that*)

lemma *mult-mono-nonneg-nonpos*: *b * a ≤ d * c*
if *a ≤ c c ≤ 0 0 ≤ d d ≤ b* **for** *a b c d*::*'a*::*ordered-ring*
by (*meson dual-order.trans mult-right-mono-neg mult-left-mono that*)

lemma *mult-mono-nonpos-nonpos*: *a * b ≤ c * d*
if *a ≥ c a ≤ 0 b ≥ d d ≤ 0* **for** *a b c d*::*real*
by (*meson dual-order.trans mult-left-mono-neg mult-right-mono-neg that*)

lemma *mult-float-mono1*:
shows *a ≤ b ⟹ ab ≤ bb ⟹*
aa ≤ a ⟹
b ≤ ba ⟹
ac ≤ ab ⟹
bb ≤ bc ⟹
plus-down prec (*nprt aa * pprt bc*)
(*plus-down prec* (*nprt ba * nprt bc*)
(*plus-down prec* (*pprt aa * pprt ac*)
(*pprt ba * nprt ac*)))
≤ plus-down prec (*nprt a * pprt bb*)

(*plus-down prec* (*nprt b* ∗ *nprt bb*)
(*plus-down prec* (*pprt a* ∗ *pprt ab*)
(*pprt b* ∗ *nprt ab*)))
by (*smt* (*verit*, *best*) *mult-mono plus-down-mono add-mono nprt-mono nprt-le-zero zero-le-pprt*
pprt-mono mult-mono-nonpos-nonneg mult-mono-nonpos-nonpos mult-mono-nonneg-nonpos)

lemma *mult-float-mono2*:
shows $a \leq b \Longrightarrow$
$ab \leq bb \Longrightarrow$
$aa \leq a \Longrightarrow$
$b \leq ba \Longrightarrow$
$ac \leq ab \Longrightarrow$
$bb \leq bc \Longrightarrow$
plus-up prec (*pprt b* ∗ *pprt bb*)
(*plus-up prec* (*pprt a* ∗ *nprt bb*)
(*plus-up prec* (*nprt b* ∗ *pprt ab*)
(*nprt a* ∗ *nprt ab*)))
≤ *plus-up prec* (*pprt ba* ∗ *pprt bc*)
(*plus-up prec* (*pprt aa* ∗ *nprt bc*)
(*plus-up prec* (*nprt ba* ∗ *pprt ac*)
(*nprt aa* ∗ *nprt ac*)))
by (*smt* (*verit*, *best*) *plus-up-mono add-mono mult-mono nprt-mono nprt-le-zero zero-le-pprt pprt-mono*
mult-mono-nonpos-nonneg mult-mono-nonpos-nonpos mult-mono-nonneg-nonpos)

43.15 Approximate Power

lemma *div2-less-self*[*termination-simp*]: *odd n* ⟹ *n div 2* < *n* **for** *n* :: *nat*
by (*simp add*: *odd-pos*)

fun *power-down* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-down p x 0 = *1*
| *power-down p x* (*Suc n*) =
(*if odd n then truncate-down* (*Suc p*) ((*power-down p x* (*Suc n div 2*))^2)
else truncate-down (*Suc p*) (*x* ∗ *power-down p x n*))

fun *power-up* :: *nat* ⇒ *real* ⇒ *nat* ⇒ *real*
where
power-up p x 0 = *1*
| *power-up p x* (*Suc n*) =
(*if odd n then truncate-up p* ((*power-up p x* (*Suc n div 2*))^2)
else truncate-up p (*x* ∗ *power-up p x n*))

lift-definition *power-up-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-up*
by (*induct-tac rule*: *power-up.induct*) *simp-all*

lift-definition *power-down-fl* :: *nat* ⇒ *float* ⇒ *nat* ⇒ *float* **is** *power-down*

by (*induct-tac rule*: *power-down.induct*) *simp-all*

lemma *power-float-transfer*[*transfer-rule*]:
(*rel-fun pcr-float* (*rel-fun* (=) *pcr-float*)) (^) (^)
unfolding *power-def*
by *transfer-prover*

lemma *compute-power-up-fl*[*code*]:
power-up-fl p x 0 = 1
power-up-fl p x (*Suc n*) =
(*if odd n then float-round-up p* ((*power-up-fl p x* (*Suc n div 2*))^2)
else float-round-up p (*x* ∗ *power-up-fl p x n*))
and *compute-power-down-fl*[*code*]:
power-down-fl p x 0 = 1
power-down-fl p x (*Suc n*) =
(*if odd n then float-round-down* (*Suc p*) ((*power-down-fl p x* (*Suc n div 2*))^2)
else float-round-down (*Suc p*) (*x* ∗ *power-down-fl p x n*))
unfolding *atomize-conj* **by** *transfer simp*

lemma *power-down-pos*: *0 < x* ⟹ *0 < power-down p x n*
by (*induct p x n rule*: *power-down.induct*)
(*auto simp del*: *odd-Suc-div-two intro*!: *truncate-down-pos*)

lemma *power-down-nonneg*: *0* ≤ *x* ⟹ *0* ≤ *power-down p x n*
by (*induct p x n rule*: *power-down.induct*)
(*auto simp del*: *odd-Suc-div-two intro*!: *truncate-down-nonneg mult-nonneg-nonneg*)

lemma *power-down*: *0* ≤ *x* ⟹ *power-down p x n* ≤ *x* ^ *n*
proof (*induct p x n rule*: *power-down.induct*)
case (*2 p x n*)
have *?case* **if** *odd n*
proof −
from *that 2* **have** (*power-down p x* (*Suc n div 2*)) ^ *2* ≤ (*x* ^ (*Suc n div 2*)) ^ *2*
by (*auto intro*: *power-mono power-down-nonneg simp del*: *odd-Suc-div-two*)
also have ... = *x* ^ (*Suc n div 2* ∗ *2*)
by (*simp flip*: *power-mult*)
also have *Suc n div 2* ∗ *2* = *Suc n*
using ‹*odd n*› **by** *presburger*
finally show *?thesis*
using *that* **by** (*auto intro*!: *truncate-down-le simp del*: *odd-Suc-div-two*)
qed
then show *?case*
by (*auto intro*!: *truncate-down-le mult-left-mono 2 mult-nonneg-nonneg power-down-nonneg*)
qed *simp*

lemma *power-up*: *0* ≤ *x* ⟹ *x* ^ *n* ≤ *power-up p x n*
proof (*induct p x n rule*: *power-up.induct*)
case (*2 p x n*)

```
  have ?case if odd n
  proof −
    from that even-Suc have Suc n = Suc n div 2 * 2
      by presburger
    then have x ^ Suc n ≤ (x ^ (Suc n div 2))^2
      by (simp flip: power-mult)
    also from that 2 have ... ≤ (power-up p x (Suc n div 2))^2
      by (auto intro: power-mono simp del: odd-Suc-div-two)
    finally show ?thesis
      using that by (auto intro!: truncate-up-le simp del: odd-Suc-div-two)
  qed
  then show ?case
    by (auto intro!: truncate-up-le mult-left-mono 2)
qed simp

lemmas power-up-le = order-trans[OF - power-up]
  and power-up-less = less-le-trans[OF - power-up]
  and power-down-le = order-trans[OF power-down]

lemma power-down-fl: 0 ≤ x ⟹ power-down-fl p x n ≤ x ^ n
  by transfer (rule power-down)

lemma power-up-fl: 0 ≤ x ⟹ x ^ n ≤ power-up-fl p x n
  by transfer (rule power-up)

lemma real-power-up-fl: real-of-float (power-up-fl p x n) = power-up p x n
  by transfer simp

lemma real-power-down-fl: real-of-float (power-down-fl p x n) = power-down p x
n
  by transfer simp

lemmas [simp del] = power-down.simps(2) power-up.simps(2)

lemmas power-down-simp = power-down.simps(2)
lemmas power-up-simp = power-up.simps(2)

lemma power-down-even-nonneg: even n ⟹ 0 ≤ power-down p x n
  by (induct p x n rule: power-down.induct)
   (auto simp: power-down-simp simp del: odd-Suc-div-two intro!: truncate-down-nonneg
)

lemma power-down-eq-zero-iff[simp]: power-down prec b n = 0 ⟷ b = 0 ∧ n ≠
0
proof (induction n arbitrary: b rule: less-induct)
  case (less x)
  then show ?case
    using power-down-simp[of - - x − 1]
    by (cases x) (auto simp add: div2-less-self)
```

qed

lemma *power-down-nonneg-iff*[*simp*]:
power-down prec b n ≥ *0* ⟷ *even n* ∨ *b* ≥ *0*
proof (*induction n arbitrary*: *b rule*: *less-induct*)
case (*less x*)
show *?case*
using *less*(*1*)[*of x* − *1 b*] *power-down-simp*[*of - - x* − *1*]
by (*cases x*) (*auto simp*: *algebra-split-simps zero-le-mult-iff*)
qed

lemma *power-down-neg-iff*[*simp*]:
power-down prec b n < *0* ⟷
b < *0* ∧ *odd n*
using *power-down-nonneg-iff*[*of prec b n*] **by** (*auto simp del*: *power-down-nonneg-iff*)

lemma *power-down-nonpos-iff*[*simp*]:
notes [*simp del*] = *power-down-neg-iff power-down-eq-zero-iff*
shows *power-down prec b n* ≤ *0* ⟷ *b* < *0* ∧ *odd n* ∨ *b* = *0* ∧ *n* ≠ *0*
using *power-down-neg-iff*[*of prec b n*] *power-down-eq-zero-iff*[*of prec b n*]
by *auto*

lemma *power-down-mono*:
power-down prec a n ≤ *power-down prec b n*
if ((*0* ≤ *a* ∧ *a* ≤ *b*)∨(*odd n* ∧ *a* ≤ *b*) ∨ (*even n* ∧ *a* ≤ *0* ∧ *b* ≤ *a*))
using *that*
proof (*induction n arbitrary*: *a b rule*: *less-induct*)
case (*less i*)
show *?case*
proof (*cases i*)
case *j*: (*Suc j*)
note *IH* = *less*[*unfolded j even-Suc not-not*]
note [*simp del*] = *power-down.simps*
show *?thesis*
proof *cases*
assume [*simp*]: *even j*
have *a* ∗ *power-down prec a j* ≤ *b* ∗ *power-down prec b j*
by (*metis IH*(*1*) *IH*(*2*) ‹*even j*› *lessI linear mult-mono mult-mono′ mult-mono-nonpos-nonneg power-down-even-nonneg*)
then have *truncate-down* (*Suc prec*) (*a* ∗ *power-down prec a j*) ≤ *truncate-down* (*Suc prec*) (*b* ∗ *power-down prec b j*)
by (*auto intro*!: *truncate-down-mono simp*: *abs-le-square-iff*[*symmetric*] *abs-real-def*)
then show *?thesis*
unfolding *j*
by (*simp add*: *power-down-simp*)
next
assume [*simp*]: *odd j*
have *power-down prec 0* (*Suc* (*j div 2*)) ≤ − *power-down prec b* (*Suc* (*j div*

```
2))
      if b < 0 even (j div 2)
    by (metis even-Suc le-minus-iff Suc-neq-Zero neg-equal-zero power-down-eq-zero-iff
          power-down-nonpos-iff that)
    then have truncate-down (Suc prec) ((power-down prec a (Suc (j div 2)))^2)
      ≤ truncate-down (Suc prec) ((power-down prec b (Suc (j div 2)))^2)
    by (smt (verit) IH Suc-less-eq ‹odd j› div2-less-self mult-mono-nonpos-nonpos

        Suc-neq-Zero power2-eq-square power-down-neg-iff power-down-nonpos-iff
power-mono truncate-down-mono)
    then show ?thesis
      unfolding j by (simp add: power-down-simp)
  qed
 qed simp
qed

lemma power-up-even-nonneg: even n ⟹ 0 ≤ power-up p x n
 by (induct p x n rule: power-up.induct)
   (auto simp: power-up.simps simp del: odd-Suc-div-two)

lemma power-up-eq-zero-iff[simp]: power-up prec b n = 0 ⟷ b = 0 ∧ n ≠ 0
proof (induction n arbitrary: b rule: less-induct)
 case (less x)
 then show ?case
   using power-up-simp[of - - x − 1]
   by (cases x) (auto simp: algebra-split-simps zero-le-mult-iff div2-less-self)
qed

lemma power-up-nonneg-iff[simp]:
 power-up prec b n ≥ 0 ⟷ even n ∨ b ≥ 0
proof (induction n arbitrary: b rule: less-induct)
 case (less x)
 show ?case
   using less(1)[of x − 1 b] power-up-simp[of - - x − 1]
   by (cases x) (auto simp: algebra-split-simps zero-le-mult-iff)
qed

lemma power-up-neg-iff[simp]:
 power-up prec b n < 0 ⟷ b < 0 ∧ odd n
 using power-up-nonneg-iff[of prec b n] by (auto simp del: power-up-nonneg-iff)

lemma power-up-nonpos-iff[simp]:
 notes [simp del] = power-up-neg-iff power-up-eq-zero-iff
 shows power-up prec b n ≤ 0 ⟷ b < 0 ∧ odd n ∨ b = 0 ∧ n ≠ 0
 using power-up-neg-iff[of prec b n] power-up-eq-zero-iff[of prec b n]
 by auto

lemma power-up-mono:
 power-up prec a n ≤ power-up prec b n
```

if $((0 \leq a \wedge a \leq b) \vee (odd\ n \wedge a \leq b) \vee (even\ n \wedge a \leq 0 \wedge b \leq a))$
using *that*
proof (*induction n arbitrary*: *a b rule*: *less-induct*)
case (*less i*)
show *?case*
proof (*cases i*)
case *j*: (*Suc j*)
note *IH* = *less*[*unfolded j even-Suc not-not*]
note [*simp del*] = *power-up.simps*
show *?thesis*
proof *cases*
assume [*simp*]: *even j*
have *a* ∗ *power-up prec a j* ≤ *b* ∗ *power-up prec b j*
by (*metis IH*(*1*) *IH*(*2*) ‹*even j*› *lessI linear mult-mono mult-mono′ mult-mono-nonpos-nonneg power-up-even-nonneg*)
then have *truncate-up prec* (*a* ∗ *power-up prec a j*) ≤ *truncate-up prec* (*b* ∗ *power-up prec b j*)
by (*auto intro*!: *truncate-up-mono simp*: *abs-le-square-iff*[*symmetric*] *abs-real-def*)
then show *?thesis*
unfolding *j*
by (*simp add*: *power-up-simp*)
next
assume [*simp*]: *odd j*
have *power-up prec 0* (*Suc* (*j div 2*)) ≤ − *power-up prec b* (*Suc* (*j div 2*))
if *b* < *0 even* (*j div 2*)
by (*metis Suc-neq-Zero even-Suc neg-0-le-iff-le power-up-eq-zero-iff power-up-nonpos-iff that*)
then have *truncate-up prec* ((*power-up prec a* (*Suc* (*j div 2*)))^2)
≤ *truncate-up prec* ((*power-up prec b* (*Suc* (*j div 2*)))^2)
using *IH*
by (*auto intro*!: *truncate-up-mono intro*: *order-trans*[**where** *y*=*0*]
simp: *abs-le-square-iff*[*symmetric*] *abs-real-def*
div2-less-self)
then show *?thesis*
unfolding *j*
by (*simp add*: *power-up-simp*)
qed
qed *simp*
qed

43.16 Lemmas needed by Approximate

lemma *Float-num*[*simp*]:
real-of-float (*Float 1 0*) = *1*
real-of-float (*Float 1 1*) = *2*
real-of-float (*Float 1 2*) = *4*
real-of-float (*Float 1* (− *1*)) = *1/2*
real-of-float (*Float 1* (− *2*)) = *1/4*
real-of-float (*Float 1* (− *3*)) = *1/8*

real-of-float (*Float* (− *1*) *0*) = −*1*
real-of-float (*Float* (*numeral n*) *0*) = *numeral n*
real-of-float (*Float* (− *numeral n*) *0*) = − *numeral n*
using *two-powr-int-float*[*of 2*] *two-powr-int-float*[*of* −*1*] *two-powr-int-float*[*of* −*2*]
two-powr-int-float[*of* −*3*]
using *powr-realpow*[*of 2 2*] *powr-realpow*[*of 2 3*]
using *powr-minus*[*of 2*::*real 1*] *powr-minus*[*of 2*::*real 2*] *powr-minus*[*of 2*::*real 3*]
by *auto*

lemma *real-of-Float-int*[*simp*]: *real-of-float* (*Float n 0*) = *real n*
by *simp*

lemma *float-zero*[*simp*]: *real-of-float* (*Float 0 e*) = *0*
by *simp*

lemma *abs-div-2-less*: *a* ≠ *0* ⟹ *a* ≠ −*1* ⟹ |(*a*::*int*) *div 2*| < |*a*|
by *arith*

lemma *lapprox-rat*: *real-of-float* (*lapprox-rat prec x y*) ≤ *real-of-int x* / *real-of-int y*
by (*simp add*: *lapprox-rat.rep-eq truncate-down*)

lemma *mult-div-le*:
fixes *a b* :: *int*
assumes *b* > *0*
shows *a* ≥ *b* ∗ (*a div b*)
by (*smt* (*verit*, *ccfv-threshold*) *assms minus-div-mult-eq-mod mod-int-pos-iff mult.commute*)

lemma *lapprox-rat-nonneg*:
assumes *0* ≤ *x* **and** *0* ≤ *y*
shows *0* ≤ *real-of-float* (*lapprox-rat n x y*)
using *assms*
by *transfer* (*simp add*: *truncate-down-nonneg*)

lemma *rapprox-rat*: *real-of-int x* / *real-of-int y* ≤ *real-of-float* (*rapprox-rat prec x y*)
by (*simp add*: *rapprox-rat.rep-eq truncate-up*)

lemma *rapprox-rat-le1*:
assumes *0* ≤ *x 0* < *y x* ≤ *y*
shows *real-of-float* (*rapprox-rat n x y*) ≤ *1*
using *assms*
by *transfer* (*simp add*: *truncate-up-le1*)

lemma *rapprox-rat-nonneg-nonpos*: *0* ≤ *x* ⟹ *y* ≤ *0* ⟹ *real-of-float* (*rapprox-rat n x y*) ≤ *0*
by *transfer* (*simp add*: *truncate-up-nonpos divide-nonneg-nonpos*)

lemma *rapprox-rat-nonpos-nonneg*: *x* ≤ *0* ⟹ *0* ≤ *y* ⟹ *real-of-float* (*rapprox-rat*

n x y) ≤ *0*
by *transfer* (*simp add*: *truncate-up-nonpos divide-nonpos-nonneg*)

lemma *real-divl*: *real-divl prec x y* ≤ *x* / *y*
by (*simp add*: *real-divl-def truncate-down*)

lemma *real-divr*: *x* / *y* ≤ *real-divr prec x y*
by (*simp add*: *real-divr-def truncate-up*)

lemma *float-divl*: *real-of-float* (*float-divl prec x y*) ≤ *x* / *y*
by *transfer* (*rule real-divl*)

lemma *real-divl-lower-bound*: *0* ≤ *x* ⟹ *0* ≤ *y* ⟹ *0* ≤ *real-divl prec x y*
by (*simp add*: *real-divl-def truncate-down-nonneg*)

lemma *float-divl-lower-bound*: *0* ≤ *x* ⟹ *0* ≤ *y* ⟹ *0* ≤ *real-of-float* (*float-divl prec x y*)
by *transfer* (*rule real-divl-lower-bound*)

lemma *exponent-1*: *exponent 1* = *0*
using *exponent-float*[*of 1 0*] **by** (*simp add*: *one-float-def*)

lemma *mantissa-1*: *mantissa 1* = *1*
using *mantissa-float*[*of 1 0*] **by** (*simp add*: *one-float-def*)

lemma *bitlen-1*: *bitlen 1* = *1*
by (*simp add*: *bitlen-alt-def*)

lemma *float-upper-bound*: *x* ≤ *2 powr* (*bitlen* |*mantissa x*| + *exponent x*)
proof (*cases x* = *0*)
case *True*
then show *?thesis* **by** *simp*
next
case *False*
then have *mantissa x* ≠ *0*
using *mantissa-eq-zero-iff* **by** *auto*
have *x* = *mantissa x* ∗ *2 powr* (*exponent x*)
by (*rule mantissa-exponent*)
also have *mantissa x* ≤ |*mantissa x*|
by *simp*
also have ... ≤ *2 powr* (*bitlen* |*mantissa x*|)
using *bitlen-bounds*[*of* |*mantissa x*|] *bitlen-nonneg* ‹*mantissa x* ≠ *0*›
by (*auto simp del*: *of-int-abs simp add*: *powr-int*)
finally show *?thesis* **by** (*simp add*: *powr-add*)
qed

lemma *real-divl-pos-less1-bound*:
assumes *0* < *x x* ≤ *1*
shows *1* ≤ *real-divl prec 1 x*

using *assms*
by (*auto intro*!: *truncate-down-ge1 simp*: *real-divl-def*)

lemma *float-divl-pos-less1-bound*:
0 < real-of-float x ⟹ real-of-float x ≤ 1 ⟹ prec ≥ 1 ⟹
1 ≤ real-of-float (*float-divl prec 1 x*)
by *transfer* (*rule real-divl-pos-less1-bound*)

lemma *float-divr*: *real-of-float x / real-of-float y ≤ real-of-float* (*float-divr prec x y*)
by (*simp add*: *float-divr.rep-eq real-divr*)

lemma *real-divr-pos-less1-lower-bound*:
assumes *0 < x*
and *x ≤ 1*
shows *1 ≤ real-divr prec 1 x*
proof −
have *1 ≤ 1 / x*
using ‹*0 < x*› **and** ‹*x ≤ 1*› **by** *auto*
also have *... ≤ real-divr prec 1 x*
using *real-divr*[**where** *x = 1* **and** *y = x*] **by** *auto*
finally show *?thesis* **by** *auto*
qed

lemma *float-divr-pos-less1-lower-bound*: *0 < x ⟹ x ≤ 1 ⟹ 1 ≤ float-divr prec 1 x*
by *transfer* (*rule real-divr-pos-less1-lower-bound*)

lemma *real-divr-nonpos-pos-upper-bound*: *x ≤ 0 ⟹ 0 ≤ y ⟹ real-divr prec x y ≤ 0*
by (*simp add*: *real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonpos-pos-upper-bound*:
real-of-float x ≤ 0 ⟹ 0 ≤ real-of-float y ⟹ real-of-float (*float-divr prec x y*) *≤ 0*
by *transfer* (*rule real-divr-nonpos-pos-upper-bound*)

lemma *real-divr-nonneg-neg-upper-bound*: *0 ≤ x ⟹ y ≤ 0 ⟹ real-divr prec x y ≤ 0*
by (*simp add*: *real-divr-def truncate-up-nonpos divide-le-0-iff*)

lemma *float-divr-nonneg-neg-upper-bound*:
0 ≤ real-of-float x ⟹ real-of-float y ≤ 0 ⟹ real-of-float (*float-divr prec x y*) *≤ 0*
by *transfer* (*rule real-divr-nonneg-neg-upper-bound*)

lemma *Float-le-zero-iff*: *Float a b ≤ 0 ⟷ a ≤ 0*
by (*auto simp*: *zero-float-def mult-le-0-iff*)

lemma *real-of-float-pprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*pprt a*) = *pprt* (*real-of-float a*)
unfolding *pprt-def sup-float-def max-def sup-real-def* **by** *auto*

lemma *real-of-float-nprt*[*simp*]:
fixes *a* :: *float*
shows *real-of-float* (*nprt a*) = *nprt* (*real-of-float a*)
unfolding *nprt-def inf-float-def min-def inf-real-def* **by** *auto*

context
begin

lift-definition *int-floor-fl* :: *float* ⇒ *int* **is** *floor* **.**

qualified lemma *compute-int-floor-fl*[*code*]:
int-floor-fl (*Float m e*) = (*if* 0 ≤ *e then m* ∗ 2 ^ *nat e else m div* (2 ^ (*nat* (−*e*))))
apply *transfer*
by (*smt* (*verit*, *ccfv-threshold*) *Float.rep-eq compute-real-of-float floor-divide-of-int-eq*
floor-of-int of-int-1 of-int-add of-int-mult of-int-power)

lift-definition *floor-fl* :: *float* ⇒ *float* **is** λ*x*. *real-of-int* ⌊*x*⌋
by *simp*

qualified lemma *compute-floor-fl*[*code*]:
floor-fl (*Float m e*) = (*if* 0 ≤ *e then Float m e else Float* (*m div* (2 ^ (*nat* (−*e*)))) 0)
apply *transfer*
using *compute-int-floor-fl int-floor-fl.rep-eq powr-int* **by** *auto*

end

lemma *floor-fl*: *real-of-float* (*floor-fl x*) ≤ *real-of-float x*
by *transfer simp*

lemma *int-floor-fl*: *real-of-int* (*int-floor-fl x*) ≤ *real-of-float x*
by *transfer simp*

lemma *floor-pos-exp*: *exponent* (*floor-fl x*) ≥ 0
proof (*cases floor-fl x* = 0)
case *True*
then show *?thesis*
by (*simp add*: *floor-fl-def*)
next
case *False*
have *eq*: *floor-fl x* = *Float* ⌊*real-of-float x*⌋ 0
by *transfer simp*

obtain *i* **where** $\lfloor$*real-of-float x*$\rfloor$ = *mantissa* (*floor-fl x*) * *2* ^ *i 0* = *exponent* (*floor-fl x*) − *int i*
by (*rule denormalize-shift*[*OF eq False*])
then show *?thesis*
by *simp*
qed

lemma *compute-mantissa*[*code*]:
mantissa (*Float m e*) =
(*if m = 0 then 0 else if 2 dvd m then mantissa* (*normfloat* (*Float m e*)) *else m*)
by (*auto simp*: *mantissa-float Float.abs-eq simp flip*: *zero-float-def*)

lemma *compute-exponent*[*code*]:
exponent (*Float m e*) =
(*if m = 0 then 0 else if 2 dvd m then exponent* (*normfloat* (*Float m e*)) *else e*)
by (*auto simp*: *exponent-float Float.abs-eq simp flip*: *zero-float-def*)

lifting-update *Float.float.lifting*
lifting-forget *Float.float.lifting*

end

44 Pointwise instantiation of functions to algebra type classes

theory *Function-Algebras*
imports *Main*
begin

Pointwise operations

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

definition *f* + *g* = (λ*x*. *f x* + *g x*)
instance ..

end

lemma *plus-fun-apply* [*simp*]:
(*f* + *g*) *x* = *f x* + *g x*
by (*simp add*: *plus-fun-def*)

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

definition *0* = (λ*x*. *0*)
instance ..

end

lemma *zero-fun-apply* [*simp*]:
0 x = 0
by (*simp add*: *zero-fun-def*)

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

definition $f * g = (\lambda x.\ f\ x * g\ x)$
instance ..

end

lemma *times-fun-apply* [*simp*]:
$(f * g)\ x = f\ x * g\ x$
by (*simp add*: *times-fun-def*)

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

definition $1 = (\lambda x.\ 1)$
instance ..

end

lemma *one-fun-apply* [*simp*]:
1 x = 1
by (*simp add*: *one-fun-def*)

Additive structures

instance *fun* :: (*type*, *semigroup-add*) *semigroup-add*
by *standard* (*simp add*: *fun-eq-iff add.assoc*)

instance *fun* :: (*type*, *cancel-semigroup-add*) *cancel-semigroup-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*simp add*: *fun-eq-iff add.commute*)

instance *fun* :: (*type*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by *standard* (*simp-all add*: *fun-eq-iff diff-diff-eq*)

instance *fun* :: (*type*, *monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
by *standard simp*

instance *fun* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add* **..**

instance *fun* :: (*type*, *group-add*) *group-add*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *ab-group-add*) *ab-group-add*
by *standard simp-all*

Multiplicative structures

instance *fun* :: (*type*, *semigroup-mult*) *semigroup-mult*
by *standard* (*simp add*: *fun-eq-iff mult.assoc*)

instance *fun* :: (*type*, *ab-semigroup-mult*) *ab-semigroup-mult*
by *standard* (*simp add*: *fun-eq-iff mult.commute*)

instance *fun* :: (*type*, *monoid-mult*) *monoid-mult*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *comm-monoid-mult*) *comm-monoid-mult*
by *standard simp*

Misc

instance *fun* :: (*type*, *Rings.dvd*) *Rings.dvd* **..**

instance *fun* :: (*type*, *mult-zero*) *mult-zero*
by *standard* (*simp-all add*: *fun-eq-iff*)

instance *fun* :: (*type*, *zero-neq-one*) *zero-neq-one*
by *standard* (*simp add*: *fun-eq-iff*)

Ring structures

instance *fun* :: (*type*, *semiring*) *semiring*
by *standard* (*simp-all add*: *fun-eq-iff algebra-simps*)

instance *fun* :: (*type*, *comm-semiring*) *comm-semiring*
by *standard* (*simp add*: *fun-eq-iff algebra-simps*)

instance *fun* :: (*type*, *semiring-0*) *semiring-0* **..**

instance *fun* :: (*type*, *comm-semiring-0*) *comm-semiring-0* **..**

instance *fun* :: (*type*, *semiring-0-cancel*) *semiring-0-cancel* **..**

instance *fun* :: (*type*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel* **..**

instance *fun* :: (*type*, *semiring-1*) *semiring-1* **..**

lemma *numeral-fun*:
‹*numeral n* = (λ*x*::'*a*. *numeral n*)›

by (*induction n*) (*simp-all only*: *numeral.simps plus-fun-def*, *simp-all*)

lemma *numeral-fun-apply* [*simp*]:
‹*numeral n x* = *numeral n*›
by (*simp add*: *numeral-fun*)

lemma *of-nat-fun*: *of-nat n* = (λ*x*::′*a*. *of-nat n*)
proof −
have *comp*: *comp* = (λ*f g x*. *f* (*g x*))
by (*rule ext*)+ *simp*
have *plus-fun*: *plus* = (λ*f g x*. *f x* + *g x*)
by (*rule ext*, *rule ext*) (*fact plus-fun-def*)
have *of-nat n* = (*comp* (*plus* (*1*::′*b*)) ^^ *n*) (λ*x*::′*a*. *0*)
by (*simp add*: *of-nat-def plus-fun zero-fun-def one-fun-def comp*)
also have ... = *comp* ((*plus 1*) ^^ *n*) (λ*x*::′*a*. *0*)
by (*simp only*: *comp-funpow*)
finally show *?thesis* **by** (*simp add*: *of-nat-def comp*)
qed

lemma *of-nat-fun-apply* [*simp*]:
of-nat n x = *of-nat n*
by (*simp add*: *of-nat-fun*)

instance *fun* :: (*type*, *comm-semiring-1*) *comm-semiring-1* ..

instance *fun* :: (*type*, *semiring-1-cancel*) *semiring-1-cancel* ..

instance *fun* :: (*type*, *comm-semiring-1-cancel*) *comm-semiring-1-cancel*
by *standard* (*auto simp add*: *times-fun-def algebra-simps*)

instance *fun* :: (*type*, *semiring-char-0*) *semiring-char-0*
proof
from *inj-of-nat* **have** *inj* (λ*n* (*x*::′*a*). *of-nat n* :: ′*b*)
by (*rule inj-fun*)
then have *inj* (λ*n*. *of-nat n* :: ′*a* ⇒ ′*b*)
by (*simp add*: *of-nat-fun*)
then show *inj* (*of-nat* :: *nat* ⇒ ′*a* ⇒ ′*b*) .
qed

instance *fun* :: (*type*, *ring*) *ring* ..

instance *fun* :: (*type*, *comm-ring*) *comm-ring* ..

instance *fun* :: (*type*, *ring-1*) *ring-1* ..

instance *fun* :: (*type*, *comm-ring-1*) *comm-ring-1* ..

instance *fun* :: (*type*, *ring-char-0*) *ring-char-0* ..

Ordered structures

instance *fun* :: (*type*, *ordered-ab-semigroup-add*) *ordered-ab-semigroup-add*
by *standard* (*auto simp add*: *le-fun-def intro*: *add-left-mono*)

instance *fun* :: (*type*, *ordered-cancel-ab-semigroup-add*) *ordered-cancel-ab-semigroup-add*
..

instance *fun* :: (*type*, *ordered-ab-semigroup-add-imp-le*) *ordered-ab-semigroup-add-imp-le*
by *standard* (*simp add*: *le-fun-def*)

instance *fun* :: (*type*, *ordered-comm-monoid-add*) *ordered-comm-monoid-add* **..**

instance *fun* :: (*type*, *ordered-cancel-comm-monoid-add*) *ordered-cancel-comm-monoid-add*
..

instance *fun* :: (*type*, *ordered-ab-group-add*) *ordered-ab-group-add* **..**

instance *fun* :: (*type*, *ordered-semiring*) *ordered-semiring*
by *standard* (*auto simp add*: *le-fun-def intro*: *mult-left-mono mult-right-mono*)

instance *fun* :: (*type*, *dioid*) *dioid*
proof *standard*
fix *a b* :: *'a* ⇒ *'b*
show *a* ≤ *b* ⟷ (∃ *c*. *b* = *a* + *c*)
unfolding *le-fun-def plus-fun-def fun-eq-iff choice-iff*[*symmetric*, *of* λ*x c*. *b x* = *a x* + *c*]
by (*intro arg-cong*[**where** *f*=*All*] *ext canonically-ordered-monoid-add-class.le-iff-add*)
qed

instance *fun* :: (*type*, *ordered-comm-semiring*) *ordered-comm-semiring*
by *standard* (*fact mult-left-mono*)

instance *fun* :: (*type*, *ordered-cancel-semiring*) *ordered-cancel-semiring* **..**

instance *fun* :: (*type*, *ordered-cancel-comm-semiring*) *ordered-cancel-comm-semiring*
..

instance *fun* :: (*type*, *ordered-ring*) *ordered-ring* **..**

instance *fun* :: (*type*, *ordered-comm-ring*) *ordered-comm-ring* **..**

lemmas *func-plus* = *plus-fun-def*
lemmas *func-zero* = *zero-fun-def*
lemmas *func-times* = *times-fun-def*
lemmas *func-one* = *one-fun-def*

end

45 Pointwise instantiation of functions to division

theory *Function-Division*
imports *Function-Algebras*
begin

45.1 Syntactic with division

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

definition *inverse f* = *inverse* ∘ *f*

definition *f div g* = (λ*x*. *f x* / *g x*)

instance ..

end

lemma *inverse-fun-apply* [*simp*]:
inverse f x = *inverse* (*f x*)
by (*simp add*: *inverse-fun-def*)

lemma *divide-fun-apply* [*simp*]:
(*f* / *g*) *x* = *f x* / *g x*
by (*simp add*: *divide-fun-def*)

Unfortunately, we cannot lift this operations to algebraic type classes for division: being different from the constant zero function $f \neq 0$ is too weak as precondition. So we must introduce our own set of lemmas.

abbreviation *zero-free* :: ('*b* ⇒ '*a*::*field*) ⇒ *bool* **where**
zero-free f ≡ ¬ (∃ *x*. *f x* = *0*)

lemma *fun-left-inverse*:
fixes *f* :: '*b* ⇒ '*a*::*field*
shows *zero-free f* ⟹ *inverse f* ∗ *f* = *1*
by (*simp add*: *fun-eq-iff*)

lemma *fun-right-inverse*:
fixes *f* :: '*b* ⇒ '*a*::*field*
shows *zero-free f* ⟹ *f* ∗ *inverse f* = *1*
by (*simp add*: *fun-eq-iff*)

lemma *fun-divide-inverse*:
fixes *f g* :: '*b* ⇒ '*a*::*field*
shows *f* / *g* = *f* ∗ *inverse g*
by (*simp add*: *fun-eq-iff divide-inverse*)

Feel free to extend this.

Another possibility would be a reformulation of the division type classes to user a *zero-free* predicate rather than a direct $a \neq 0$ condition.

end

46 Functions as a real vector spaces

theory *Function-Real-Vectors*
imports
Complex-Main
Function-Algebras
begin

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

definition *scaleR-fun*::*real* ⇒ ('*a* ⇒ '*b*) ⇒ '*a* ⇒ '*b*
where *scaleR-fun* = (λ*c f*. (λ*x*. *c* ∗$_R$ *f x*))

lemma *scaleR-apply* [*simp*, *code*]: (*c* ∗$_R$ *f*) *x* = *c* ∗$_R$ (*f x*)
by (*simp add*: *scaleR-fun-def*)

instance ..

end

instance *fun* :: (*type*, *real-vector*) *real-vector*
by *standard*
(*auto simp*: *scaleR-fun-def algebra-simps*)

end

47 Lexicographic order on functions

theory *Fun-Lexorder*
imports *Main*
begin

definition *less-fun* :: ('*a*::*linorder* ⇒ '*b*::*linorder*) ⇒ ('*a* ⇒ '*b*) ⇒ *bool*
where
less-fun f g ⟷ (∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*))

lemma *less-funI*:
assumes ∃ *k*. *f k* < *g k* ∧ (∀ *k'* < *k*. *f k'* = *g k'*)
shows *less-fun f g*
using *assms* **by** (*simp add*: *less-fun-def*)

lemma *less-funE*:
assumes *less-fun f g*

obtains k **where** $f\ k < g\ k$ **and** $\bigwedge k'.\ k' < k \Longrightarrow f\ k' = g\ k'$
using *assms* **unfolding** *less-fun-def* **by** *blast*

lemma *less-fun-asym*:
assumes *less-fun f g*
shows ¬ *less-fun g f*
proof
from *assms* **obtain** *k1* **where** *k1*: $f\ k1 < g\ k1\ k' < k1 \Longrightarrow f\ k' = g\ k'$ **for** k'
by (*blast elim*!: *less-funE*)
assume *less-fun g f* **then obtain** *k2* **where** *k2*: $g\ k2 < f\ k2\ k' < k2 \Longrightarrow g\ k' = f\ k'$ **for** k'
by (*blast elim*!: *less-funE*)
show *False* **proof** (*cases k1 k2 rule*: *linorder-cases*)
case *equal* **with** *k1 k2* **show** *False* **by** *simp*
next
case *less* **with** *k2* **have** $g\ k1 = f\ k1$ **by** *simp*
with *k1* **show** *False* **by** *simp*
next
case *greater* **with** *k1* **have** $f\ k2 = g\ k2$ **by** *simp*
with *k2* **show** *False* **by** *simp*
qed
qed

lemma *less-fun-irrefl*:
¬ *less-fun f f*
proof
assume *less-fun f f*
then obtain *k* **where** *k*: $f\ k < f\ k$
by (*blast elim*!: *less-funE*)
then show *False* **by** *simp*
qed

lemma *less-fun-trans*:
assumes *less-fun f g* **and** *less-fun g h*
shows *less-fun f h*
proof (*rule less-funI*)
from ‹*less-fun f g*› **obtain** *k1* **where** *k1*: $f\ k1 < g\ k1\ k' < k1 \Longrightarrow f\ k' = g\ k'$ **for** k'
by (*blast elim*!: *less-funE*)
from ‹*less-fun g h*› **obtain** *k2* **where** *k2*: $g\ k2 < h\ k2\ k' < k2 \Longrightarrow g\ k' = h\ k'$ **for** k'
by (*blast elim*!: *less-funE*)
show $\exists k.\ f\ k < h\ k \wedge (\forall k' < k.\ f\ k' = h\ k')$
proof (*cases k1 k2 rule*: *linorder-cases*)
case *equal* **with** *k1 k2* **show** *?thesis* **by** (*auto simp add*: *exI* [*of* - *k2*])
next
case *less* **with** *k2* **have** $g\ k1 = h\ k1\ \bigwedge k'.\ k' < k1 \Longrightarrow g\ k' = h\ k'$ **by** *simp-all*
with *k1* **show** *?thesis* **by** (*auto intro*: *exI* [*of* - *k1*])
next

```
   case greater with k1 have f k2 = g k2 ⋀k'. k' < k2 ⟹ f k' = g k' by simp-all
    with k2 show ?thesis by (auto intro: exI [of - k2])
  qed
qed

lemma order-less-fun:
  class.order (λf g. less-fun f g ∨ f = g) less-fun
 by (rule order-strictI) (auto intro: less-fun-trans intro!: less-fun-irrefl less-fun-asym)

lemma less-fun-trichotomy:
  assumes finite {k. f k ≠ g k}
  shows less-fun f g ∨ f = g ∨ less-fun g f
proof −
  { define K where K = {k. f k ≠ g k}
    assume f ≠ g
    then obtain k' where f k' ≠ g k' by auto
    then have [simp]: K ≠ {} by (auto simp add: K-def)
    with assms have [simp]: finite K by (simp add: K-def)
    define q where q = Min K
    then have q ∈ K and ⋀k. k ∈ K ⟹ k ≥ q by auto
    then have ⋀k. ¬ k ≥ q ⟹ k ∉ K by blast
    then have ∗: ⋀k. k < q ⟹ f k = g k by (simp add: K-def)
    from ‹q ∈ K› have f q ≠ g q by (simp add: K-def)
    then have f q < g q ∨ f q > g q by auto
    with ∗ have less-fun f g ∨ less-fun g f
      by (auto intro!: less-funI)
  } then show ?thesis by blast
qed

end
```

48 The *going-to* filter

```
theory Going-To-Filter
  imports Complex-Main
begin

definition going-to-within :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a set ⇒ 'a filter
    (‹(‹open-block notation=‹mixfix going-to-within››(-)/ going'-to (-)/ within (-))›
[1000,60,60] 60)
  where f going-to F within A = inf (filtercomap f F) (principal A)

abbreviation going-to :: ('a ⇒ 'b) ⇒ 'b filter ⇒ 'a filter
    (infix ‹going'-to› 60)
    where f going-to F ≡ f going-to F within UNIV
```

The *going-to* filter is, in a sense, the opposite of *filtermap*. It corresponds to the intuition of, given a function $f : A \to B$ and a filter F on the range of B, looking at such values of x that $f(x)$ approaches F. This can be written

as *f going-to F*.

A classic example is the *at-infinity* filter, which describes the neigbourhood of infinity (i. e. all values sufficiently far away from the zero). This can also be written as *norm going-to at-top*.

Additionally, the *going-to* filter can be restricted with an optional 'within' parameter. For instance, if one would would want to consider the filter of complex numbers near infinity that do not lie on the negative real line, one could write *cmod going-to at-top within − complex-of-real ' {..0}*.

A third, less mathematical example lies in the complexity analysis of algorithms. Suppose we wanted to say that an algorithm on lists takes $O(n^2)$ time where n is the length of the input list. We can write this using the Landau symbols from the AFP, where the underlying filter is *length going-to sequentially*. If, on the other hand, we want to look the complexity of the algorithm on sorted lists, we could use the filter *length going-to sequentially within Collect sorted*.

lemma *going-to-def*: *f going-to F = filtercomap f F*
by (*simp add*: *going-to-within-def*)

lemma *eventually-going-toI* [*intro*]:
assumes *eventually P F*
shows *eventually* (*λx*. *P* (*f x*)) (*f going-to F*)
using *assms* **by** (*auto simp*: *going-to-def*)

lemma *filterlim-going-toI-weak* [*intro*]: *filterlim f F* (*f going-to F within A*)
unfolding *going-to-within-def*
by (*meson filterlim-filtercomap filterlim-iff inf-le1 le-filter-def*)

lemma *going-to-mono*: *F ≤ G ⟹ A ⊆ B ⟹ f going-to F within A ≤ f going-to G within B*
unfolding *going-to-within-def* **by** (*intro inf-mono filtercomap-mono*) *simp-all*

lemma *going-to-inf*:
f going-to (*inf F G*) *within A* = *inf* (*f going-to F within A*) (*f going-to G within A*)
by (*simp add*: *going-to-within-def filtercomap-inf inf-assoc inf-commute inf-left-commute*)

lemma *going-to-sup*:
f going-to (*sup F G*) *within A ≥ sup* (*f going-to F within A*) (*f going-to G within A*)
by (*auto simp*: *going-to-within-def intro*!: *inf.coboundedI1 filtercomap-sup filtercomap-mono*)

lemma *going-to-top* [*simp*]: *f going-to top within A* = *principal A*
by (*simp add*: *going-to-within-def*)

lemma *going-to-bot* [*simp*]: *f going-to bot within A* = *bot*
by (*simp add*: *going-to-within-def*)

lemma *going-to-principal*:
f going-to principal A within B = principal (f −‘ A ∩ B)
by (*simp add*: *going-to-within-def*)

lemma *going-to-within-empty* [*simp*]: *f going-to F within {} = bot*
by (*simp add*: *going-to-within-def*)

lemma *going-to-within-union* [*simp*]:
f going-to F within (A ∪ B) = sup (f going-to F within A) (f going-to F within B)
by (*simp add*: *going-to-within-def flip*: *inf-sup-distrib1*)

lemma *eventually-going-to-at-top-linorder*:
fixes *f* :: *'a ⇒ 'b* :: *linorder*
shows *eventually P (f going-to at-top within A) ⟷ (∃ C. ∀ x∈A. f x ≥ C ⟶ P x)*
unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-top-linorder **by** *fast*

lemma *eventually-going-to-at-bot-linorder*:
fixes *f* :: *'a ⇒ 'b* :: *linorder*
shows *eventually P (f going-to at-bot within A) ⟷ (∃ C. ∀ x∈A. f x ≤ C ⟶ P x)*
unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-bot-linorder **by** *fast*

lemma *eventually-going-to-at-top-dense*:
fixes *f* :: *'a ⇒ 'b* :: {*linorder*,*no-top*}
shows *eventually P (f going-to at-top within A) ⟷ (∃ C. ∀ x∈A. f x > C ⟶ P x)*
unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-top-dense **by** *fast*

lemma *eventually-going-to-at-bot-dense*:
fixes *f* :: *'a ⇒ 'b* :: {*linorder*,*no-bot*}
shows *eventually P (f going-to at-bot within A) ⟷ (∃ C. ∀ x∈A. f x < C ⟶ P x)*
unfolding *going-to-within-def eventually-filtercomap*
eventually-inf-principal eventually-at-bot-dense **by** *fast*

lemma *eventually-going-to-nhds*:
eventually P (f going-to nhds a within A) ⟷
(∃ S. open S ∧ a ∈ S ∧ (∀ x∈A. f x ∈ S ⟶ P x))
unfolding *going-to-within-def eventually-filtercomap eventually-inf-principal*
eventually-nhds **by** *fast*

lemma *eventually-going-to-at*:
eventually P (f going-to (at a within B) within A) ⟷

$(\exists S.\ open\ S \wedge a \in S \wedge (\forall x{\in}A.\ f\ x \in B \cap S - \{a\} \longrightarrow P\ x))$
unfolding *at-within-def going-to-inf eventually-inf-principal eventually-going-to-nhds going-to-principal* **by** *fast*

lemma *norm-going-to-at-top-eq*: *norm going-to at-top = at-infinity*
by (*simp add*: *eventually-at-infinity eventually-going-to-at-top-linorder filter-eq-iff*)

lemmas *at-infinity-altdef = norm-going-to-at-top-eq* [*symmetric*]

end

49 Big sum and product over function bodies

theory *Groups-Big-Fun*
imports
Main
begin

49.1 Abstract product

locale *comm-monoid-fun = comm-monoid*
begin

definition $G :: ('b \Rightarrow 'a) \Rightarrow 'a$
where
expand-set: $G\ g = comm\text{-}monoid\text{-}set.F\ f\ \mathbf{1}\ g\ \{a.\ g\ a \neq \mathbf{1}\}$

interpretation *F*: *comm-monoid-set f* **1**
..

lemma *expand-superset*:
assumes *finite A* **and** $\{a.\ g\ a \neq \mathbf{1}\} \subseteq A$
shows $G\ g = F.F\ g\ A$
using *F.mono-neutral-right assms expand-set* **by** *fastforce*

lemma *conditionalize*:
assumes *finite A*
shows $F.F\ g\ A = G\ (\lambda a.\ if\ a \in A\ then\ g\ a\ else\ \mathbf{1})$
using *assms*
by (*smt* (*verit*, *ccfv-threshold*) *Diff-iff F.mono-neutral-cong-right expand-set mem-Collect-eq subsetI*)

lemma *neutral* [*simp*]:
$G\ (\lambda a.\ \mathbf{1}) = \mathbf{1}$
by (*simp add*: *expand-set*)

lemma *update* [*simp*]:
assumes *finite* $\{a.\ g\ a \neq \mathbf{1}\}$

assumes $g\ a = \mathbf{1}$
shows $G\ (g(a := b)) = b * G\ g$
proof (*cases* $b = \mathbf{1}$)
case *True* **with** ‹$g\ a = \mathbf{1}$› **show** *?thesis*
by (*simp add*: *expand-set*) (*rule F.cong*, *auto*)
next
case *False*
moreover have $\{a'.\ a' \neq a \longrightarrow g\ a' \neq \mathbf{1}\} = insert\ a\ \{a.\ g\ a \neq \mathbf{1}\}$
by *auto*
moreover from ‹$g\ a = \mathbf{1}$› **have** $a \notin \{a.\ g\ a \neq \mathbf{1}\}$
by *simp*
moreover have $F.F\ (\lambda a'.\ if\ a' = a\ then\ b\ else\ g\ a')\ \{a.\ g\ a \neq \mathbf{1}\} = F.F\ g\ \{a.\ g\ a \neq \mathbf{1}\}$
by (*rule F.cong*) (*auto simp add*: ‹$g\ a = \mathbf{1}$›)
ultimately show *?thesis* **using** ‹*finite* $\{a.\ g\ a \neq \mathbf{1}\}$› **by** (*simp add*: *expand-set*)
qed

lemma *infinite* [*simp*]:
$\neg$ *finite* $\{a.\ g\ a \neq \mathbf{1}\} \Longrightarrow G\ g = \mathbf{1}$
by (*simp add*: *expand-set*)

lemma *cong* [*cong*]:
assumes $\bigwedge a.\ g\ a = h\ a$
shows $G\ g = G\ h$
using *assms* **by** (*simp add*: *expand-set*)

lemma *not-neutral-obtains-not-neutral*:
assumes $G\ g \neq \mathbf{1}$
obtains *a* **where** $g\ a \neq \mathbf{1}$
using *assms* **by** (*auto elim*: *F.not-neutral-contains-not-neutral simp add*: *expand-set*)

lemma *reindex-cong*:
assumes *bij l*
assumes $g \circ l = h$
shows $G\ g = G\ h$
proof −
from *assms* **have** *unfold*: $h = g \circ l$ **by** *simp*
from ‹*bij l*› **have** *inj l* **by** (*rule bij-is-inj*)
then have *inj-on l* $\{a.\ h\ a \neq \mathbf{1}\}$ **by** (*rule inj-on-subset*) *simp*
moreover from ‹*bij l*› **have** $\{a.\ g\ a \neq \mathbf{1}\} = l\ `\ \{a.\ h\ a \neq \mathbf{1}\}$
by (*auto simp add*: *image-Collect unfold elim*: *bij-pointE*)
moreover have $\bigwedge x.\ x \in \{a.\ h\ a \neq \mathbf{1}\} \Longrightarrow g\ (l\ x) = h\ x$
by (*simp add*: *unfold*)
ultimately have $F.F\ g\ \{a.\ g\ a \neq \mathbf{1}\} = F.F\ h\ \{a.\ h\ a \neq \mathbf{1}\}$
by (*rule F.reindex-cong*)
then show *?thesis* **by** (*simp add*: *expand-set*)
qed

lemma *distrib*:
assumes *finite* $\{a.\ g\ a \neq \mathbf{1}\}$ **and** *finite* $\{a.\ h\ a \neq \mathbf{1}\}$
shows $G\ (\lambda a.\ g\ a * h\ a) = G\ g * G\ h$
proof −
from *assms* **have** *finite* $(\{a.\ g\ a \neq \mathbf{1}\} \cup \{a.\ h\ a \neq \mathbf{1}\})$ **by** *simp*
moreover have $\{a.\ g\ a * h\ a \neq \mathbf{1}\} \subseteq \{a.\ g\ a \neq \mathbf{1}\} \cup \{a.\ h\ a \neq \mathbf{1}\}$
by *auto* (*drule sym*, *simp*)
ultimately show *?thesis*
using *assms*
by (*simp add*: *expand-superset* [*of* $\{a.\ g\ a \neq \mathbf{1}\} \cup \{a.\ h\ a \neq \mathbf{1}\}$] *F.distrib*)
qed

lemma *swap*:
assumes *finite C*
assumes *subset*: $\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\} \times \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\} \subseteq C$ (**is** *?A* × *?B* ⊆ *C*)
shows $G\ (\lambda a.\ G\ (g\ a)) = G\ (\lambda b.\ G\ (\lambda a.\ g\ a\ b))$
proof −
from ‹*finite C*› *subset*
have *finite* $(\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\} \times \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\})$
by (*rule rev-finite-subset*)
then have *fins*:
finite $\{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$ *finite* $\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\}$
by (*auto simp add*: *finite-cartesian-product-iff*)
have *subsets*: $\bigwedge a.\ \{b.\ g\ a\ b \neq \mathbf{1}\} \subseteq \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$
$\bigwedge b.\ \{a.\ g\ a\ b \neq \mathbf{1}\} \subseteq \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\}$
$\{a.\ F.F\ (g\ a)\ \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\} \neq \mathbf{1}\} \subseteq \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\}$
$\{a.\ F.F\ (\lambda aa.\ g\ aa\ a)\ \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\} \neq \mathbf{1}\} \subseteq \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$
by (*auto elim*: *F.not-neutral-contains-not-neutral*)
from *F.swap* **have**
$F.F\ (\lambda a.\ F.F\ (g\ a)\ \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\})\ \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\} =$
$F.F\ (\lambda b.\ F.F\ (\lambda a.\ g\ a\ b)\ \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\})\ \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$ **.**
with *subsets fins* **have** $G\ (\lambda a.\ F.F\ (g\ a)\ \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}) =$
$G\ (\lambda b.\ F.F\ (\lambda a.\ g\ a\ b)\ \{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\})$
by (*auto simp add*: *expand-superset* [*of* $\{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$]
expand-superset [*of* $\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\}$])
with *subsets fins* **show** *?thesis*
by (*auto simp add*: *expand-superset* [*of* $\{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\}$]
expand-superset [*of* $\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\}$])
qed

lemma *cartesian-product*:
assumes *finite C*
assumes *subset*: $\{a.\ \exists b.\ g\ a\ b \neq \mathbf{1}\} \times \{b.\ \exists a.\ g\ a\ b \neq \mathbf{1}\} \subseteq C$ (**is** *?A* × *?B* ⊆ *C*)
shows $G\ (\lambda a.\ G\ (g\ a)) = G\ (\lambda(a,\ b).\ g\ a\ b)$
proof −
from *subset* ‹*finite C*› **have** *fin-prod*: *finite* (*?A* × *?B*)
by (*rule finite-subset*)

from *fin-prod* **have** *finite ?A* **and** *finite ?B*
by (*auto simp add*: *finite-cartesian-product-iff*)
have ∗: $G\ (\lambda a.\ G\ (g\ a)) =$
$(F.F\ (\lambda a.\ F.F\ (g\ a)\ \{b.\ \exists\, a.\ g\ a\ b \neq \mathbf{1}\})\ \{a.\ \exists\, b.\ g\ a\ b \neq \mathbf{1}\})$
using ‹*finite ?A*› ‹*finite ?B*› *expand-superset*
by (*smt* (*verit*, *del-insts*) *Collect-mono local.cong not-neutral-obtains-not-neutral*)
have ∗∗: $\{p.\ (\mathit{case}\ p\ \mathit{of}\ (a,\ b) \Rightarrow g\ a\ b) \neq \mathbf{1}\} \subseteq ?A \times ?B$
by *auto*
show *?thesis*
using ‹*finite C*› *expand-superset*
using ∗ ∗∗ *F.cartesian-product fin-prod* **by** *force*
qed

lemma *cartesian-product2*:
assumes *fin*: *finite D*
assumes *subset*: $\{(a,\ b).\ \exists\, c.\ g\ a\ b\ c \neq \mathbf{1}\} \times \{c.\ \exists\, a\ b.\ g\ a\ b\ c \neq \mathbf{1}\} \subseteq D$ (**is** $?AB \times ?C \subseteq D$)
shows $G\ (\lambda(a,\ b).\ G\ (g\ a\ b)) = G\ (\lambda(a,\ b,\ c).\ g\ a\ b\ c)$
proof −
have *bij*: *bij* $(\lambda(a,\ b,\ c).\ ((a,\ b),\ c))$
by (*auto intro*!: *bijI injI simp add*: *image-def*)
have $\{p.\ \exists\, c.\ g\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)\ c \neq \mathbf{1}\} \times \{c.\ \exists\, p.\ g\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)\ c \neq \mathbf{1}\} \subseteq D$
by *auto* (*insert subset*, *blast*)
with *fin* **have** $G\ (\lambda p.\ G\ (g\ (\mathit{fst}\ p)\ (\mathit{snd}\ p))) = G\ (\lambda(p,\ c).\ g\ (\mathit{fst}\ p)\ (\mathit{snd}\ p)\ c)$
by (*rule cartesian-product*)
then have $G\ (\lambda(a,\ b).\ G\ (g\ a\ b)) = G\ (\lambda((a,\ b),\ c).\ g\ a\ b\ c)$
by (*auto simp add*: *split-def*)
also have $G\ (\lambda((a,\ b),\ c).\ g\ a\ b\ c) = G\ (\lambda(a,\ b,\ c).\ g\ a\ b\ c)$
using *bij* **by** (*rule reindex-cong* [*of* $\lambda(a,\ b,\ c).\ ((a,\ b),\ c)$]) (*simp add*: *fun-eq-iff*)
finally show *?thesis* **.**
qed

lemma *delta* [*simp*]:
$G\ (\lambda b.\ \mathit{if}\ b = a\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1}) = g\ a$
proof −
have $\{b.\ (\mathit{if}\ b = a\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1}) \neq \mathbf{1}\} \subseteq \{a\}$ **by** *auto*
then show *?thesis* **by** (*simp add*: *expand-superset* [*of* $\{a\}$])
qed

lemma *delta′* [*simp*]:
$G\ (\lambda b.\ \mathit{if}\ a = b\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1}) = g\ a$
proof −
have $(\lambda b.\ \mathit{if}\ a = b\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1}) = (\lambda b.\ \mathit{if}\ b = a\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1})$
by (*simp add*: *fun-eq-iff*)
then have $G\ (\lambda b.\ \mathit{if}\ a = b\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1}) = G\ (\lambda b.\ \mathit{if}\ b = a\ \mathit{then}\ g\ b\ \mathit{else}\ \mathbf{1})$
by (*simp cong del*: *cong*)
then show *?thesis* **by** *simp*
qed

end

49.2 Concrete sum

context *comm-monoid-add*
begin

sublocale *Sum-any*: *comm-monoid-fun plus 0*
  rewrites *comm-monoid-set.F plus 0 = sum*
  defines *Sum-any = Sum-any.G*
proof −
  show *comm-monoid-fun plus 0* **..**
  then interpret *Sum-any*: *comm-monoid-fun plus 0* **.**
  from *sum-def* **show** *comm-monoid-set.F plus 0 = sum* **by** (*auto intro*: *sym*)
qed

end

syntax (*ASCII*)
  -Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (‹(‹*indent=3 notation=‹binder SUM››SUM -. -*)› [*0*, *10*] *10*)
syntax
  -Sum-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-add* (‹(‹*indent=2 notation=‹binder* ∑*››*∑ *-. -*)› [*0*, *10*] *10*)
syntax-consts
  -Sum-any ⇌ *Sum-any*
translations
  ∑ *a. b* ⇌ *CONST Sum-any* (λ*a. b*)

lemma *Sum-any-left-distrib*:
  fixes *r* :: *'a* :: *semiring-0*
  assumes *finite* {*a. g a* ≠ *0*}
  shows *Sum-any g* ∗ *r* = (∑ *n. g n* ∗ *r*)
  by (*metis* (*mono-tags*, *lifting*) *Collect-mono Sum-any.expand-superset assms mult-zero-left sum-distrib-right*)

lemma *Sum-any-right-distrib*:
  fixes *r* :: *'a* :: *semiring-0*
  assumes *finite* {*a. g a* ≠ *0*}
  shows *r* ∗ *Sum-any g* = (∑ *n. r* ∗ *g n*)
  by (*metis* (*mono-tags*, *lifting*) *Collect-mono Sum-any.expand-superset assms mult-zero-right sum-distrib-left*)

lemma *Sum-any-product*:
  fixes *f g* :: *'b* ⇒ *'a*::*semiring-0*
  assumes *finite* {*a. f a* ≠ *0*} **and** *finite* {*b. g b* ≠ *0*}
  shows *Sum-any f* ∗ *Sum-any g* = (∑ *a.* ∑ *b. f a* ∗ *g b*)
proof −
  have ∀ *a.* (∑ *b. a* ∗ *g b*) = *a* ∗ *Sum-any g*

by (*simp add*: *Sum-any-right-distrib assms*(*2*))
then show *?thesis*
by (*simp add*: *Sum-any-left-distrib assms*(*1*))
qed

lemma *Sum-any-eq-zero-iff* [*simp*]:
fixes $f :: 'a \Rightarrow nat$
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows *Sum-any* $f = 0 \longleftrightarrow f = (\lambda\text{-}.\ 0)$
using *assms* **by** (*simp add*: *Sum-any.expand-set fun-eq-iff*)

49.3 Concrete product

context *comm-monoid-mult*
begin

sublocale *Prod-any*: *comm-monoid-fun times 1*
rewrites *comm-monoid-set.F times 1 = prod*
defines *Prod-any = Prod-any.G*
proof −
show *comm-monoid-fun times 1* **..**
then interpret *Prod-any*: *comm-monoid-fun times 1* **.**
from *prod-def* **show** *comm-monoid-set.F times 1 = prod* **by** (*auto intro*: *sym*)
qed

end

syntax (*ASCII*)
-Prod-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* (‹(‹*indent=4 notation=‹binder PROD››PROD -. -*)› [*0*, *10*] *10*)
syntax
-Prod-any :: *pttrn* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult* (‹(‹*indent=2 notation=‹binder* ∏ ››∏ *-. -*)› [*0*, *10*] *10*)
syntax-consts
-Prod-any == *Prod-any*
translations
∏ *a. b* == *CONST Prod-any* (λ*a. b*)

lemma *Prod-any-zero*:
fixes $f :: 'b \Rightarrow 'a$ :: *comm-semiring-1*
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $f\ a = 0$
shows $(\prod a.\ f\ a) = 0$
proof −
from ‹$f\ a = 0$› **have** $f\ a \neq 1$ **by** *simp*
with ‹$f\ a = 0$› **have** $\exists a.\ f\ a \neq 1 \wedge f\ a = 0$ **by** *blast*
with ‹*finite* $\{a.\ f\ a \neq 1\}$› **show** *?thesis*
by (*simp add*: *Prod-any.expand-set prod-zero*)
qed

lemma *Prod-any-not-zero*:
fixes $f :: 'b \Rightarrow 'a :: comm\text{-}semiring\text{-}1$
assumes *finite* $\{a.\ f\ a \neq 1\}$
assumes $(\prod a.\ f\ a) \neq 0$
shows $f\ a \neq 0$
using *assms Prod-any-zero* [*of f*] **by** *blast*

lemma *power-Sum-any*:
assumes *finite* $\{a.\ f\ a \neq 0\}$
shows $c\ \hat{}\ (\sum a.\ f\ a) = (\prod a.\ c\ \hat{}\ f\ a)$
proof −
have $\{a.\ c\ \hat{}\ f\ a \neq 1\} \subseteq \{a.\ f\ a \neq 0\}$
by (*auto intro*: *ccontr*)
with *assms* **show** *?thesis*
by (*simp add*: *Sum-any.expand-set Prod-any.expand-superset power-sum*)
qed

end

50 Infinite Type Class

The type class of infinite sets (orginally from the Incredible Proof Machine)

theory *Infinite-Typeclass*
imports *Complex-Main*
begin

class *infinite* =
assumes *infinite-UNIV*: *infinite* (*UNIV*::*'a set*)

begin

lemma *arb-element*: *finite* $Y \Longrightarrow \exists x :: 'a.\ x \notin Y$
using *ex-new-if-finite infinite-UNIV*
by *blast*

lemma *arb-finite-subset*: *finite* $Y \Longrightarrow \exists X :: 'a\ set.\ Y \cap X = \{\} \wedge$ *finite* $X \wedge n \leq$ *card* X
proof −
assume *fin*: *finite Y*
then obtain X **where** $X \subseteq UNIV - Y$ *finite* X $n \leq$ *card* X
using *infinite-UNIV*
by (*metis Compl-eq-Diff-UNIV finite-compl infinite-arbitrarily-large order-refl*)
then show *?thesis*
by *auto*
qed

lemma *arb-inj-on-finite-infinite*: *finite*$(A :: 'b\ set) \Longrightarrow \exists f :: 'b \Rightarrow 'a.$ *inj-on f A*

by (*meson arb-finite-subset card-le-inj infinite-imp-nonempty*)

lemma *arb-countable-map*: *finite* $Y \Longrightarrow \exists f :: (nat \Rightarrow 'a).\ inj\ f \wedge range\ f \subseteq UNIV - Y$
using *infinite-UNIV*
by (*auto simp*: *infinite-countable-subset*)

end

instance *nat* :: *infinite*
by (*intro-classes*) *simp*

instance *int* :: *infinite*
by (*intro-classes*) *simp*

instance *rat* :: *infinite*
proof
show *infinite* (*UNIV*::*rat set*)
by (*simp add*: *infinite-UNIV-char-0*)
qed

instance *real* :: *infinite*
proof
show *infinite* (*UNIV*::*real set*)
by (*simp add*: *infinite-UNIV-char-0*)
qed

instance *complex* :: *infinite*
proof
show *infinite* (*UNIV*::*complex set*)
by (*simp add*: *infinite-UNIV-char-0*)
qed

instance *option* :: (*infinite*) *infinite*
by *intro-classes* (*simp add*: *infinite-UNIV*)

instance *prod* :: (*infinite*, *type*) *infinite*
by *intro-classes* (*simp add*: *finite-prod infinite-UNIV*)

instance *list* :: (*type*) *infinite*
by *intro-classes* (*simp add*: *infinite-UNIV-listI*)

end

51 Algebraic operations on sets

theory *Set-Algebras*
imports *Main*
begin

This library lifts operations like addition and multiplication to sets. It was designed to support asymptotic calculations for the now-obsolete BigO theory, but has other uses.

instantiation *set* :: (*plus*) *plus*
begin

definition *plus-set* :: *'a::plus set* ⇒ *'a set* ⇒ *'a set*
  where *set-plus-def*: $A + B = \{c.\ \exists a{\in}A.\ \exists b{\in}B.\ c = a + b\}$

instance ..

end

instantiation *set* :: (*times*) *times*
begin

definition *times-set* :: *'a::times set* ⇒ *'a set* ⇒ *'a set*
  where *set-times-def*: $A * B = \{c.\ \exists a{\in}A.\ \exists b{\in}B.\ c = a * b\}$

instance ..

end

instantiation *set* :: (*zero*) *zero*
begin

definition *set-zero*[*simp*]: $(0::'a::zero\ set) = \{0\}$

instance ..

end

instantiation *set* :: (*one*) *one*
begin

definition *set-one*[*simp*]: $(1::'a::one\ set) = \{1\}$

instance ..

end

definition *elt-set-plus* :: *'a::plus* ⇒ *'a set* ⇒ *'a set* (**infixl** ‹+o› *70*)
  where $a +o B = \{c.\ \exists b{\in}B.\ c = a + b\}$

definition *elt-set-times* :: *'a::times* ⇒ *'a set* ⇒ *'a set* (**infixl** ‹*o› *80*)
  where $a *o B = \{c.\ \exists b{\in}B.\ c = a * b\}$

abbreviation (*input*) *elt-set-eq* :: *'a* ⇒ *'a set* ⇒ *bool* (**infix** ‹=o› *50*)
  where $x =o A \equiv x \in A$

instance *set* :: (*semigroup-add*) *semigroup-add*
by *standard* (*force simp add*: *set-plus-def add.assoc*)

instance *set* :: (*ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*force simp add*: *set-plus-def add.commute*)

instance *set* :: (*monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *set-plus-def*)

instance *set* :: (*comm-monoid-add*) *comm-monoid-add*
by *standard* (*simp-all add*: *set-plus-def*)

instance *set* :: (*semigroup-mult*) *semigroup-mult*
by *standard* (*force simp add*: *set-times-def mult.assoc*)

instance *set* :: (*ab-semigroup-mult*) *ab-semigroup-mult*
by *standard* (*force simp add*: *set-times-def mult.commute*)

instance *set* :: (*monoid-mult*) *monoid-mult*
by *standard* (*simp-all add*: *set-times-def*)

instance *set* :: (*comm-monoid-mult*) *comm-monoid-mult*
by *standard* (*simp-all add*: *set-times-def*)

lemma *sumset-empty* [*simp*]: $A + \{\} = \{\}$ $\{\} + A = \{\}$
by (*auto simp*: *set-plus-def*)

lemma *Un-set-plus*: $(A \cup B) + C = (A+C) \cup (B+C)$ **and** *set-plus-Un*: $C + (A \cup B) = (C+A) \cup (C+B)$
by (*auto simp*: *set-plus-def*)

lemma
fixes A :: *'a*::*comm-monoid-add set*
shows *insert-set-plus*: $(insert\ a\ A) + B = (A+B) \cup (((+)a)\ `\ B)$ **and** *set-plus-insert*: $B + (insert\ a\ A) = (B+A) \cup (((+)a)\ `\ B)$
using *add.commute* **by** (*auto simp*: *set-plus-def*)

lemma *set-add-0* [*simp*]:
fixes A :: *'a*::*comm-monoid-add set*
shows $\{0\} + A = A$
by (*metis comm-monoid-add-class.add-0 set-zero*)

lemma *set-add-0-right* [*simp*]:
fixes A :: *'a*::*comm-monoid-add set*
shows $A + \{0\} = A$
by (*metis add.comm-neutral set-zero*)

lemma *card-plus-sing*:

fixes $A :: 'a::ab\text{-}group\text{-}add\ set$
shows $card\ (A + \{a\}) = card\ A$
proof (*rule bij-betw-same-card*)
show *bij-betw* $((+)\ (-a))\ (A + \{a\})\ A$
by (*fastforce simp*: *set-plus-def bij-betw-def image-iff*)
qed

lemma *set-plus-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a + b \in C + D$
by (*auto simp add*: *set-plus-def*)

lemma *set-plus-elim*:
assumes $x \in A + B$
obtains $a\ b$ **where** $x = a + b$ **and** $a \in A$ **and** $b \in B$
using *assms* **unfolding** *set-plus-def* **by** *fast*

lemma *set-plus-intro2* [*intro*]: $b \in C \Longrightarrow a + b \in a +o C$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-rearrange*: $(a +o C) + (b +o D) = (a + b) +o (C + D)$
for $a\ b :: 'a::comm\text{-}monoid\text{-}add$
by (*auto simp*: *elt-set-plus-def set-plus-def*; *metis group-cancel.add1 group-cancel.add2*)

lemma *set-plus-rearrange2*: $a +o (b +o C) = (a + b) +o C$
for $a\ b :: 'a::semigroup\text{-}add$
by (*auto simp add*: *elt-set-plus-def add.assoc*)

lemma *set-plus-rearrange3*: $(a +o B) + C = a +o (B + C)$
for $a :: 'a::semigroup\text{-}add$
by (*auto simp add*: *elt-set-plus-def set-plus-def*; *metis add.assoc*)

theorem *set-plus-rearrange4*: $C + (a +o D) = a +o (C + D)$
for $a :: 'a::comm\text{-}monoid\text{-}add$
by (*metis add.commute set-plus-rearrange3*)

lemmas *set-plus-rearranges* = *set-plus-rearrange set-plus-rearrange2*
set-plus-rearrange3 set-plus-rearrange4

lemma *set-plus-mono* [*intro*!]: $C \subseteq D \Longrightarrow a +o C \subseteq a +o D$
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-plus-mono2* [*intro*]: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C + E \subseteq D + F$
for $C\ D\ E\ F :: 'a::plus\ set$
by (*auto simp add*: *set-plus-def*)

lemma *set-plus-mono3* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq C + D$
by (*auto simp add*: *elt-set-plus-def set-plus-def*)

lemma *set-plus-mono4* [*intro*]: $a \in C \Longrightarrow a +o D \subseteq D + C$
for $a :: 'a::comm\text{-}monoid\text{-}add$

by (*auto simp add*: *elt-set-plus-def set-plus-def ac-simps*)

lemma *set-plus-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a +o B \subseteq C + D$
using *order-subst2* **by** *blast*

lemma *set-plus-mono-b*: $C \subseteq D \Longrightarrow x \in a +o C \Longrightarrow x \in a +o D$
using *set-plus-mono* **by** *blast*

lemma *set-zero-plus* [*simp*]: $0 +o C = C$
for C :: *'a*::*comm-monoid-add set*
by (*auto simp add*: *elt-set-plus-def*)

lemma *set-zero-plus2*: $0 \in A \Longrightarrow B \subseteq A + B$
for A B :: *'a*::*comm-monoid-add set*
using *set-plus-intro* **by** *fastforce*

lemma *set-plus-imp-minus*: $a \in b +o C \Longrightarrow a - b \in C$
for a b :: *'a*::*ab-group-add*
by (*auto simp add*: *elt-set-plus-def ac-simps*)

lemma *set-minus-imp-plus*: $a - b \in C \Longrightarrow a \in b +o C$
for a b :: *'a*::*ab-group-add*
by (*metis add.commute diff-add-cancel set-plus-intro2*)

lemma *set-minus-plus*: $a - b \in C \longleftrightarrow a \in b +o C$
for a b :: *'a*::*ab-group-add*
by (*meson set-minus-imp-plus set-plus-imp-minus*)

lemma *set-times-intro* [*intro*]: $a \in C \Longrightarrow b \in D \Longrightarrow a * b \in C * D$
by (*auto simp add*: *set-times-def*)

lemma *set-times-elim*:
assumes $x \in A * B$
obtains a b **where** $x = a * b$ **and** $a \in A$ **and** $b \in B$
using *assms* **unfolding** *set-times-def* **by** *fast*

lemma *set-times-intro2* [*intro*!]: $b \in C \Longrightarrow a * b \in a *o C$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-rearrange*: $(a *o C) * (b *o D) = (a * b) *o (C * D)$
for a b :: *'a*::*comm-monoid-mult*
by (*auto simp add*: *elt-set-times-def set-times-def*; *metis mult.assoc mult.left-commute*)

lemma *set-times-rearrange2*: $a *o (b *o C) = (a * b) *o C$
for a b :: *'a*::*semigroup-mult*
by (*auto simp add*: *elt-set-times-def mult.assoc*)

lemma *set-times-rearrange3*: $(a *o B) * C = a *o (B * C)$
for a :: *'a*::*semigroup-mult*

by (*auto simp add*: *elt-set-times-def set-times-def*; *metis mult.assoc*)

theorem *set-times-rearrange4*: $C * (a *o D) = a *o (C * D)$
for $a :: 'a$::*comm-monoid-mult*
by (*metis mult.commute set-times-rearrange3*)

lemmas *set-times-rearranges* = *set-times-rearrange set-times-rearrange2*
set-times-rearrange3 set-times-rearrange4

lemma *set-times-mono* [*intro*]: $C \subseteq D \Longrightarrow a *o C \subseteq a *o D$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-mono2* [*intro*]: $C \subseteq D \Longrightarrow E \subseteq F \Longrightarrow C * E \subseteq D * F$
for *C D E F* :: $'a$::*times set*
by (*auto simp add*: *set-times-def*)

lemma *set-times-mono3* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq C * D$
by (*auto simp add*: *elt-set-times-def set-times-def*)

lemma *set-times-mono4* [*intro*]: $a \in C \Longrightarrow a *o D \subseteq D * C$
for $a :: 'a$::*comm-monoid-mult*
by (*auto simp add*: *elt-set-times-def set-times-def ac-simps*)

lemma *set-times-mono5*: $a \in C \Longrightarrow B \subseteq D \Longrightarrow a *o B \subseteq C * D$
by (*meson dual-order.trans set-times-mono set-times-mono3*)

lemma *set-one-times* [*simp*]: $1 *o C = C$
for $C :: 'a$::*comm-monoid-mult set*
by (*auto simp add*: *elt-set-times-def*)

lemma *set-times-plus-distrib*: $a *o (b +o C) = (a * b) +o (a *o C)$
for $a\ b :: 'a$::*semiring*
by (*auto simp add*: *elt-set-plus-def elt-set-times-def ring-distribs*)

lemma *set-times-plus-distrib2*: $a *o (B + C) = (a *o B) + (a *o C)$
for $a :: 'a$::*semiring*
by (*auto simp*: *set-plus-def elt-set-times-def*; *metis distrib-left*)

lemma *set-times-plus-distrib3*: $(a +o C) * D \subseteq a *o D + C * D$
for $a :: 'a$::*semiring*
using *distrib-right*
by (*fastforce simp add*: *elt-set-plus-def elt-set-times-def set-times-def set-plus-def*)

lemmas *set-times-plus-distribs* =
set-times-plus-distrib
set-times-plus-distrib2

lemma *set-neg-intro*: $a \in (-\ 1) *o C \Longrightarrow -\ a \in C$
for $a :: 'a$::*ring-1*

by (*auto simp add*: *elt-set-times-def*)

lemma *set-neg-intro2*: $a \in C \Longrightarrow - a \in (-1) *o C$
for $a :: 'a::ring\text{-}1$
by (*auto simp add*: *elt-set-times-def*)

lemma *set-plus-image*: $S + T = (\lambda(x, y).\ x + y)\ `\ (S \times T)$
by (*fastforce simp*: *set-plus-def image-iff*)

lemma *set-times-image*: $S * T = (\lambda(x, y).\ x * y)\ `\ (S \times T)$
by (*fastforce simp*: *set-times-def image-iff*)

lemma *finite-set-plus*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s + t)$
by (*simp add*: *set-plus-image*)

lemma *finite-set-times*: *finite* $s \Longrightarrow$ *finite* $t \Longrightarrow$ *finite* $(s * t)$
by (*simp add*: *set-times-image*)

lemma *set-sum-alt*:
assumes *fin*: *finite* I
shows *sum* $S\ I = \{sum\ s\ I\ |s.\ \forall i{\in}I.\ s\ i \in S\ i\}$
(**is** - = *?sum* I)
using *fin*
proof *induct*
case *empty*
then show *?case* **by** *simp*
next
case (*insert* $x\ F$)
have *sum* $S\ (insert\ x\ F) = S\ x + ?sum\ F$
using *insert.hyps* **by** *auto*
also have $\ldots = \{s\ x + sum\ s\ F\ |s.\ \forall\ i{\in}insert\ x\ F.\ s\ i \in S\ i\}$
unfolding *set-plus-def*
proof *safe*
fix $y\ s$
assume $y \in S\ x\ \forall i{\in}F.\ s\ i \in S\ i$
then show $\exists s'.\ y + sum\ s\ F = s'\ x + sum\ s'\ F \wedge (\forall i{\in}insert\ x\ F.\ s'\ i \in S\ i)$
using *insert.hyps*
by (*intro exI*[*of* - $\lambda i.$ *if* $i \in F$ *then* $s\ i$ *else* y]) (*auto simp add*: *set-plus-def*)
qed *auto*
finally show *?case*
using *insert.hyps* **by** *auto*
qed

lemma *sum-set-cond-linear*:
fixes $f :: 'a::comm\text{-}monoid\text{-}add\ set \Rightarrow 'b::comm\text{-}monoid\text{-}add\ set$
assumes [*intro*!]: $\bigwedge A\ B.\ P\ A \Longrightarrow P\ B \Longrightarrow P\ (A + B)\ P\ \{0\}$
and f: $\bigwedge A\ B.\ P\ A \Longrightarrow P\ B \Longrightarrow f\ (A + B) = f\ A + f\ B\ f\ \{0\} = \{0\}$
assumes *all*: $\bigwedge i.\ i \in I \Longrightarrow P\ (S\ i)$
shows $f\ (sum\ S\ I) = sum\ (f \circ S)\ I$

proof (*cases finite I*)
case *True*
from *this all* **show** *?thesis*
proof *induct*
case *empty*
then show *?case* **by** (*auto intro*!: *f*)
next
case (*insert x F*)
from ‹*finite F*› ‹$\bigwedge i.\ i \in$ *insert x F* $\Longrightarrow$ *P* (*S i*)› **have** *P* (*sum S F*)
by *induct auto*
with *insert* **show** *?case*
by (*simp*, *subst f*) *auto*
qed
next
case *False*
then show *?thesis* **by** (*auto intro*!: *f*)
qed

lemma *sum-set-linear*:
fixes *f* :: *'a*::*comm-monoid-add set* ⇒ *'b*::*comm-monoid-add set*
assumes $\bigwedge A\ B.\ f(A) + f(B) = f(A + B)\ f\ \{0\} = \{0\}$
shows *f* (*sum S I*) = *sum* (*f* ∘ *S*) *I*
using *sum-set-cond-linear*[*of* $\lambda x.$ *True f I S*] *assms* **by** *auto*

lemma *set-times-Un-distrib*:
$A * (B \cup C) = A * B \cup A * C$
$(A \cup B) * C = A * C \cup B * C$
by (*auto simp*: *set-times-def*)

lemma *set-times-UNION-distrib*:
$A * \bigcup (M\ `\ I) = (\bigcup i \in I.\ A * M\ i)$
$\bigcup (M\ `\ I) * A = (\bigcup i \in I.\ M\ i * A)$
by (*auto simp*: *set-times-def*)

end

52 Interval Type

theory *Interval*
imports
Complex-Main
Lattice-Algebras
Set-Algebras
begin

A type of non-empty, closed intervals.

typedef (**overloaded**) *'a interval* =
{(*a*::*'a*::*preorder*, *b*). $a \leq b$}
morphisms *bounds-of-interval Interval*

by *auto*

setup-lifting *type-definition-interval*

lift-definition *lower*::('*a*::*preorder*) *interval* ⇒ '*a* **is** *fst* .

lift-definition *upper*::('*a*::*preorder*) *interval* ⇒ '*a* **is** *snd* .

lemma *interval-eq-iff*: $a = b \longleftrightarrow lower\ a = lower\ b \wedge upper\ a = upper\ b$
by *transfer auto*

lemma *interval-eqI*: $lower\ a = lower\ b \Longrightarrow upper\ a = upper\ b \Longrightarrow a = b$
by (*auto simp*: *interval-eq-iff*)

lemma *lower-le-upper*[*simp*]: $lower\ i \leq upper\ i$
by *transfer auto*

lift-definition *set-of* :: '*a*::*preorder interval* ⇒ '*a set* **is** $\lambda x.\ \{fst\ x\ ..\ snd\ x\}$.

lemma *set-of-eq*: $set\text{-}of\ x = \{lower\ x\ ..\ upper\ x\}$
by *transfer simp*

context notes [[*typedef-overloaded*]] **begin**

lift-definition(*code-dt*) *Interval'*::'*a*::*preorder* ⇒ '*a*::*preorder* ⇒ '*a interval option*
is $\lambda a\ b.\ if\ a \leq b\ then\ Some\ (a,\ b)\ else\ None$
by *auto*

lemma *Interval'-split*:
$P\ (Interval'\ a\ b) \longleftrightarrow$
$(\forall\, ivl.\ a \leq b \longrightarrow lower\ ivl = a \longrightarrow upper\ ivl = b \longrightarrow P\ (Some\ ivl)) \wedge (\neg a \leq b \longrightarrow P\ None)$
by *transfer auto*

lemma *Interval'-split-asm*:
$P\ (Interval'\ a\ b) \longleftrightarrow$
$\neg((\exists\, ivl.\ a \leq b \wedge lower\ ivl = a \wedge upper\ ivl = b \wedge \neg P\ (Some\ ivl)) \vee (\neg a \leq b \wedge \neg P\ None))$
unfolding *Interval'-split*
by *auto*

lemmas *Interval'-splits* = *Interval'-split Interval'-split-asm*

lemma *Interval'-eq-Some*: $Interval'\ a\ b = Some\ i \Longrightarrow lower\ i = a \wedge upper\ i = b$
by (*simp split*: *Interval'-splits*)

end

instantiation *interval* :: ({*preorder*,*equal*}) *equal*

begin

definition *equal-class.equal a b* ≡ (*lower a* = *lower b*) ∧ (*upper a* = *upper b*)

instance proof qed (*simp add*: *equal-interval-def interval-eq-iff*)
end

instantiation *interval* :: (*preorder*) *ord* **begin**

definition *less-eq-interval* :: *'a interval* ⇒ *'a interval* ⇒ *bool*
where *less-eq-interval a b* ⟷ *lower b* ≤ *lower a* ∧ *upper a* ≤ *upper b*

definition *less-interval* :: *'a interval* ⇒ *'a interval* ⇒ *bool*
where *less-interval x y* = (*x* ≤ *y* ∧ ¬ *y* ≤ *x*)

instance proof qed
end

instantiation *interval* :: (*lattice*) *semilattice-sup*
begin

lift-definition *sup-interval* :: *'a interval* ⇒ *'a interval* ⇒ *'a interval*
is λ(*a*, *b*) (*c*, *d*). (*inf a c*, *sup b d*)
by (*auto simp*: *le-infI1 le-supI1*)

lemma *lower-sup*[*simp*]: *lower* (*sup A B*) = *inf* (*lower A*) (*lower B*)
by *transfer auto*

lemma *upper-sup*[*simp*]: *upper* (*sup A B*) = *sup* (*upper A*) (*upper B*)
by *transfer auto*

instance proof qed (*auto simp*: *less-eq-interval-def less-interval-def interval-eq-iff*)
end

lemma *set-of-interval-union*: *set-of A* ∪ *set-of B* ⊆ *set-of* (*sup A B*) **for** *A*::*'a*::*lattice interval*
by (*auto simp*: *set-of-eq*)

lemma *interval-union-commute*: *sup A B* = *sup B A* **for** *A*::*'a*::*lattice interval*
by (*auto simp add*: *interval-eq-iff inf.commute sup.commute*)

lemma *interval-union-mono1*: *set-of a* ⊆ *set-of* (*sup a A*) **for** *A* :: *'a*::*lattice interval*
using *set-of-interval-union* **by** *blast*

lemma *interval-union-mono2*: *set-of A* ⊆ *set-of* (*sup a A*) **for** *A* :: *'a*::*lattice interval*
using *set-of-interval-union* **by** *blast*

lift-definition *interval-of* :: *'a*::*preorder* ⇒ *'a interval* **is** λ*x*. (*x*, *x*)
  by *auto*

lemma *lower-interval-of*[*simp*]: *lower* (*interval-of a*) = *a*
  by *transfer auto*

lemma *upper-interval-of*[*simp*]: *upper* (*interval-of a*) = *a*
  by *transfer auto*

definition *width* :: *'a*::{*preorder*,*minus*} *interval* ⇒ *'a*
  where *width i* = *upper i* − *lower i*

instantiation *interval* :: (*ordered-ab-semigroup-add*) *ab-semigroup-add*
begin

lift-definition *plus-interval*::*'a interval* ⇒ *'a interval* ⇒ *'a interval*
  is λ(*a*, *b*). λ(*c*, *d*). (*a* + *c*, *b* + *d*)
  by (*auto intro*!: *add-mono*)
lemma *lower-plus*[*simp*]: *lower* (*plus A B*) = *plus* (*lower A*) (*lower B*)
  by *transfer auto*
lemma *upper-plus*[*simp*]: *upper* (*plus A B*) = *plus* (*upper A*) (*upper B*)
  by *transfer auto*

instance proof qed (*auto simp*: *interval-eq-iff less-eq-interval-def ac-simps*)
end

instance *interval* :: ({*ordered-ab-semigroup-add*, *lattice*}) *ordered-ab-semigroup-add*
proof qed (*auto simp*: *less-eq-interval-def intro*!: *add-mono*)

instantiation *interval* :: ({*preorder*,*zero*}) *zero*
begin

lift-definition *zero-interval*::*'a interval* **is** (*0*, *0*) **by** *auto*
lemma *lower-zero*[*simp*]: *lower 0* = *0*
  by *transfer auto*
lemma *upper-zero*[*simp*]: *upper 0* = *0*
  by *transfer auto*
instance proof qed
end

instance *interval* :: ({*ordered-comm-monoid-add*}) *comm-monoid-add*
proof qed (*auto simp*: *interval-eq-iff*)

instance *interval* :: ({*ordered-comm-monoid-add*,*lattice*}) *ordered-comm-monoid-add*
..

instantiation *interval* :: ({*ordered-ab-group-add*}) *uminus*
begin

lift-definition *uminus-interval*::*'a interval* ⇒ *'a interval* **is** λ(*a*, *b*). (−*b*, −*a*) **by** *auto*
lemma *lower-uminus*[*simp*]: *lower* (− *A*) = − *upper A*
by *transfer auto*
lemma *upper-uminus*[*simp*]: *upper* (− *A*) = − *lower A*
by *transfer auto*
instance ..
end

instantiation *interval* :: ({*ordered-ab-group-add*}) *minus*
begin

definition *minus-interval*::*'a interval* ⇒ *'a interval* ⇒ *'a interval*
where *minus-interval a b* = *a* + − *b*
lemma *lower-minus*[*simp*]: *lower* (*minus A B*) = *minus* (*lower A*) (*upper B*)
by (*auto simp*: *minus-interval-def*)
lemma *upper-minus*[*simp*]: *upper* (*minus A B*) = *minus* (*upper A*) (*lower B*)
by (*auto simp*: *minus-interval-def*)

instance ..
end

instantiation *interval* :: ({*times*, *linorder*}) *times*
begin

lift-definition *times-interval* :: *'a interval* ⇒ *'a interval* ⇒ *'a interval*
is λ(*a1*, *a2*). λ(*b1*, *b2*).
(*let x1* = *a1* ∗ *b1*; *x2* = *a1* ∗ *b2*; *x3* = *a2* ∗ *b1*; *x4* = *a2* ∗ *b2*
in (*min x1* (*min x2* (*min x3 x4*)), *max x1* (*max x2* (*max x3 x4*))))
by (*auto simp*: *Let-def intro*!: *min.coboundedI1 max.coboundedI1*)

lemma *lower-times*:
lower (*times A B*) = *Min* {*lower A* ∗ *lower B*, *lower A* ∗ *upper B*, *upper A* ∗ *lower B*, *upper A* ∗ *upper B*}
by *transfer* (*auto simp*: *Let-def*)

lemma *upper-times*:
upper (*times A B*) = *Max* {*lower A* ∗ *lower B*, *lower A* ∗ *upper B*, *upper A* ∗ *lower B*, *upper A* ∗ *upper B*}
by *transfer* (*auto simp*: *Let-def*)

instance ..
end

lemma *interval-eq-set-of-iff*: *X* = *Y* ⟷ *set-of X* = *set-of Y* **for** *X Y*::*'a*::*order interval*
by (*auto simp*: *set-of-eq interval-eq-iff*)

52.1 Membership

abbreviation (**in** *preorder*) *in-interval* (‹(‹*notation*=‹*infix* $\in_i$››-/ $\in_i$ -)› [*51*, *51*] *50*)
where *in-interval x X* ≡ *x* ∈ *set-of X*

lemma *in-interval-to-interval*[*intro*!]: *a* $\in_i$ *interval-of a*
by (*auto simp*: *set-of-eq*)

lemma *plus-in-intervalI*:
fixes *x y* :: *'a* :: *ordered-ab-semigroup-add*
shows *x* $\in_i$ *X* ⟹ *y* $\in_i$ *Y* ⟹ *x* + *y* $\in_i$ *X* + *Y*
by (*simp add*: *add-mono-thms-linordered-semiring*(*1*) *set-of-eq*)

lemma *connected-set-of*[*intro*, *simp*]:
connected (*set-of X*) **for** *X*::*'a*::*linear-continuum-topology interval*
by (*auto simp*: *set-of-eq*)

lemma *ex-sum-in-interval-lemma*: ∃ *xa*∈{*la* .. *ua*}. ∃ *xb*∈{*lb* .. *ub*}. *x* = *xa* + *xb*
if *la* ≤ *ua lb* ≤ *ub la* + *lb* ≤ *x x* ≤ *ua* + *ub*
ua − *la* ≤ *ub* − *lb*
for *la b c d*::*'a*::*linordered-ab-group-add*
proof −
define *wa* **where** *wa* = *ua* − *la*
define *wb* **where** *wb* = *ub* − *lb*
define *w* **where** *w* = *wa* + *wb*
define *d* **where** *d* = *x* − *la* − *lb*
define *da* **where** *da* = *max 0* (*min wa* (*d* − *wa*))
define *db* **where** *db* = *d* − *da*
from *that* **have** *nonneg*: *0* ≤ *wa 0* ≤ *wb 0* ≤ *w 0* ≤ *d d* ≤ *w*
by (*auto simp add*: *wa-def wb-def w-def d-def add.commute le-diff-eq*)
have *0* ≤ *db*
by (*auto simp*: *da-def nonneg db-def intro*!: *min.coboundedI2*)
have *x* = (*la* + *da*) + (*lb* + *db*)
by (*simp add*: *da-def db-def d-def*)
moreover
have *x* − *la* − *ub* ≤ *da*
using *that*
unfolding *da-def*
by (*intro max.coboundedI2*) (*auto simp*: *wa-def d-def diff-le-eq diff-add-eq*)
then have *db* ≤ *wb*
by (*auto simp*: *db-def d-def wb-def algebra-simps*)
with ‹*0* ≤ *db*› *that nonneg* **have** *lb* + *db* ∈ {*lb*..*ub*}
by (*auto simp*: *wb-def algebra-simps*)
moreover
have *da* ≤ *wa*
by (*auto simp*: *da-def nonneg*)
then have *la* + *da* ∈ {*la*..*ua*}
by (*auto simp*: *da-def wa-def algebra-simps*)
ultimately show *?thesis*

by *force*
qed

lemma *ex-sum-in-interval*: $\exists xa{\geq}la.\ xa \leq ua \wedge (\exists xb{\geq}lb.\ xb \leq ub \wedge x = xa + xb)$
if *a*: $la \leq ua$ **and** *b*: $lb \leq ub$ **and** *x*: $la + lb \leq x\ x \leq ua + ub$
for *la b c d*::*'a*::*linordered-ab-group-add*
proof −
from *linear* **consider** $ua - la \leq ub - lb \mid ub - lb \leq ua - la$
by *blast*
then show *?thesis*
proof *cases*
case *1*
from *ex-sum-in-interval-lemma*[*OF that 1*]
show *?thesis* **by** *auto*
next
case *2*
from *x* **have** $lb + la \leq x\ x \leq ub + ua$ **by** (*simp-all add*: *ac-simps*)
from *ex-sum-in-interval-lemma*[*OF b a this 2*]
show *?thesis* **by** *auto*
qed
qed

lemma *Icc-plus-Icc*:
$\{a\ ..\ b\} + \{c\ ..\ d\} = \{a + c\ ..\ b + d\}$
if $a \leq b\ c \leq d$
for *a b c d*::*'a*::*linordered-ab-group-add*
using *ex-sum-in-interval*[*OF that*]
by (*auto intro*: *add-mono simp*: *atLeastAtMost-iff Bex-def set-plus-def*)

lemma *set-of-plus*:
fixes *A* :: *'a*::*linordered-ab-group-add interval*
shows *set-of* $(A + B) =$ *set-of A* + *set-of B*
using *Icc-plus-Icc*[*of lower A upper A lower B upper B*]
by (*auto simp*: *set-of-eq*)

lemma *plus-in-intervalE*:
fixes *xy* :: *'a* :: *linordered-ab-group-add*
assumes $xy \in_i X + Y$
obtains *x y* **where** $xy = x + y\ x \in_i X\ y \in_i Y$
using *assms*
unfolding *set-of-plus set-plus-def*
by *auto*

lemma *set-of-uminus*: *set-of* $(-X) = \{-x \mid x.\ x \in$ *set-of X*$\}$
for *X* :: *'a* :: *ordered-ab-group-add interval*
by (*auto simp*: *set-of-eq simp*: *le-minus-iff minus-le-iff*
intro!: *exI*[**where** $x{=}{-}x$ **for** *x*])

lemma *uminus-in-intervalI*:
fixes $x :: 'a :: ordered\text{-}ab\text{-}group\text{-}add$
shows $x \in_i X \Longrightarrow -x \in_i -X$
by (*auto simp*: *set-of-uminus*)

lemma *uminus-in-intervalD*:
fixes $x :: 'a :: ordered\text{-}ab\text{-}group\text{-}add$
shows $x \in_i - X \Longrightarrow - x \in_i X$
by (*auto simp*: *set-of-uminus*)

lemma *minus-in-intervalI*:
fixes $x\ y :: 'a :: ordered\text{-}ab\text{-}group\text{-}add$
shows $x \in_i X \Longrightarrow y \in_i Y \Longrightarrow x - y \in_i X - Y$
by (*metis diff-conv-add-uminus minus-interval-def plus-in-intervalI uminus-in-intervalI*)

lemma *set-of-minus*: $set\text{-}of\ (X - Y) = \{x - y \mid x\ y\ .\ x \in set\text{-}of\ X \wedge y \in set\text{-}of\ Y\}$
for $X\ Y :: 'a :: linordered\text{-}ab\text{-}group\text{-}add\ interval$
unfolding *minus-interval-def set-of-plus set-of-uminus set-plus-def*
by *force*

lemma *times-in-intervalI*:
fixes $x\ y::'a::linordered\text{-}ring$
assumes $x \in_i X\ y \in_i Y$
shows $x * y \in_i X * Y$
proof −
define *X1* **where** $X1 \equiv lower\ X$
define *X2* **where** $X2 \equiv upper\ X$
define *Y1* **where** $Y1 \equiv lower\ Y$
define *Y2* **where** $Y2 \equiv upper\ Y$
from *assms* **have** *assms*: $X1 \le x\ x \le X2\ Y1 \le y\ y \le Y2$
by (*auto simp*: *X1-def X2-def Y1-def Y2-def set-of-eq*)
have $(X1 * Y1 \le x * y \vee X1 * Y2 \le x * y \vee X2 * Y1 \le x * y \vee X2 * Y2 \le x * y) \wedge$
$(X1 * Y1 \ge x * y \vee X1 * Y2 \ge x * y \vee X2 * Y1 \ge x * y \vee X2 * Y2 \ge x * y)$
proof (*cases x 0::'a rule*: *linorder-cases*)
case *x0*: *less*
show *?thesis*
proof (*cases* $y < 0$)
case *y0*: *True*
from *y0 x0 assms* **have** $x * y \le X1 * y$ **by** (*intro mult-right-mono-neg*, *auto*)
also from *x0 y0 assms* **have** $X1 * y \le X1 * Y1$ **by** (*intro mult-left-mono-neg*, *auto*)
finally have *1*: $x * y \le X1 * Y1$.
show *?thesis* **proof**(*cases* $X2 \le 0$)
case *True*
with *assms* **have** $X2 * Y2 \le X2 * y$ **by** (*auto intro*: *mult-left-mono-neg*)
also from *assms y0* **have** $... \le x * y$ **by** (*auto intro*: *mult-right-mono-neg*)

```
      finally have X2 * Y2 ≤ x * y.
      with 1 show ?thesis by auto
    next
      case False
      with assms have X2 * Y1 ≤ X2 * y by (auto intro: mult-left-mono)
      also from assms y0 have ... ≤ x * y by (auto intro: mult-right-mono-neg)
      finally have X2 * Y1 ≤ x * y.
      with 1 show ?thesis by auto
    qed
  next
    case False
    then have y0: y ≥ 0 by auto
    from x0 y0 assms have X1 * Y2 ≤ x * Y2 by (intro mult-right-mono, auto)
    also from y0 x0 assms have ... ≤ x * y by (intro mult-left-mono-neg, auto)
    finally have 1: X1 * Y2 ≤ x * y.
    show ?thesis
    proof(cases X2 ≤ 0)
      case X2: True
      from assms y0 have x * y ≤ X2 * y by (intro mult-right-mono)
    also from assms X2 have ... ≤ X2 * Y1 by (auto intro: mult-left-mono-neg)
      finally have x * y ≤ X2 * Y1.
      with 1 show ?thesis by auto
    next
      case X2: False
      from assms y0 have x * y ≤ X2 * y by (intro mult-right-mono)
      also from assms X2 have ... ≤ X2 * Y2 by (auto intro: mult-left-mono)
      finally have x * y ≤ X2 * Y2.
      with 1 show ?thesis by auto
    qed
  qed
next
  case [simp]: equal
  with assms show ?thesis by (cases Y2 ≤ 0, auto intro:mult-sign-intros)
next
  case x0: greater
  show ?thesis
  proof (cases y < 0)
    case y0: True
    from x0 y0 assms have X2 * Y1 ≤ X2 * y by (intro mult-left-mono, auto)
    also from y0 x0 assms have X2 * y ≤ x * y by (intro mult-right-mono-neg,
auto)
    finally have 1: X2 * Y1 ≤ x * y.
    show ?thesis
    proof(cases Y2 ≤ 0)
      case Y2: True
      from x0 assms have x * y ≤ x * Y2 by (auto intro: mult-left-mono)
    also from assms Y2 have ... ≤ X1 * Y2 by (auto intro: mult-right-mono-neg)
      finally have x * y ≤ X1 * Y2.
      with 1 show ?thesis by auto
```

```
      next
        case Y2: False
        from x0 assms have x * y ≤ x * Y2 by (auto intro: mult-left-mono)
        also from assms Y2 have ... ≤ X2 * Y2 by (auto intro: mult-right-mono)
        finally have x * y ≤ X2 * Y2.
        with 1 show ?thesis by auto
      qed
    next
      case y0: False
      from x0 y0 assms have x * y ≤ X2 * y by (intro mult-right-mono, auto)
      also from y0 x0 assms have ... ≤ X2 * Y2 by (intro mult-left-mono, auto)
      finally have 1: x * y ≤ X2 * Y2.
      show ?thesis
      proof(cases X1 ≤ 0)
        case True
        with assms have X1 * Y2 ≤ X1 * y by (auto intro: mult-left-mono-neg)
        also from assms y0 have ... ≤ x * y by (auto intro: mult-right-mono)
        finally have X1 * Y2 ≤ x * y.
        with 1 show ?thesis by auto
      next
        case False
        with assms have X1 * Y1 ≤ X1 * y by (auto intro: mult-left-mono)
        also from assms y0 have ... ≤ x * y by (auto intro: mult-right-mono)
        finally have X1 * Y1 ≤ x * y.
        with 1 show ?thesis by auto
      qed
    qed
  qed
  hence min:min (X1 * Y1) (min (X1 * Y2) (min (X2 * Y1) (X2 * Y2))) ≤ x
* y
    and max:x * y ≤ max (X1 * Y1) (max (X1 * Y2) (max (X2 * Y1) (X2 *
Y2)))
    by (auto simp:min-le-iff-disj le-max-iff-disj)
  show ?thesis using min max
    by (auto simp: Let-def X1-def X2-def Y1-def Y2-def set-of-eq lower-times up-
per-times)
qed

lemma times-in-intervalE:
  fixes xy :: 'a :: {linorder, real-normed-algebra, linear-continuum-topology}
    — TODO: linear continuum topology is pretty strong
  assumes xy ∈i X * Y
  obtains x y where xy = x * y x ∈i X y ∈i Y
proof −
  let ?mult = λ(x, y). x * y
  let ?XY = set-of X × set-of Y
  have cont: continuous-on ?XY ?mult
    by (auto intro!: tendsto-eq-intros simp: continuous-on-def split-beta')
  have conn: connected (?mult ' ?XY)
```

by (*rule connected-continuous-image*[*OF cont*]) *auto*
have *lower* (*X* ∗ *Y*) ∈ *?mult* ‘ *?XY upper* (*X* ∗ *Y*) ∈ *?mult* ‘ *?XY*
by (*auto simp*: *set-of-eq lower-times upper-times min-def max-def split*: *if-splits*)
from *connectedD-interval*[*OF conn this*, *of xy*] *assms*
obtain *x y* **where** *xy* = *x* ∗ *y x* ∈*i X y* ∈*i Y* **by** (*auto simp*: *set-of-eq*)
then show *?thesis* **..**
qed
thm *times-in-intervalE*[*of 1*::*real*]
lemma *set-of-times*: *set-of* (*X* ∗ *Y*) = {*x* ∗ *y* | *x y*. *x* ∈ *set-of X* ∧ *y* ∈ *set-of Y*}
for *X Y*::′*a* :: {*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
by (*auto intro*!: *times-in-intervalI elim*!: *times-in-intervalE*)

instance *interval* :: (*linordered-idom*) *cancel-semigroup-add*
proof qed (*auto simp*: *interval-eq-iff*)

lemma *interval-mul-commute*: *A* ∗ *B* = *B* ∗ *A* **for** *A B*:: ′*a*::*linordered-idom interval*
by (*simp add*: *interval-eq-iff lower-times upper-times ac-simps*)

lemma *interval-times-zero-right*[*simp*]: *A* ∗ *0* = *0* **for** *A* :: ′*a*::*linordered-ring interval*
by (*simp add*: *interval-eq-iff lower-times upper-times ac-simps*)

lemma *interval-times-zero-left*[*simp*]:
0 ∗ *A* = *0* **for** *A* :: ′*a*::*linordered-ring interval*
by (*simp add*: *interval-eq-iff lower-times upper-times ac-simps*)

instantiation *interval* :: ({*preorder*,*one*}) *one*
begin

lift-definition *one-interval*::′*a interval* **is** (*1*, *1*) **by** *auto*
lemma *lower-one*[*simp*]: *lower 1* = *1*
by *transfer auto*
lemma *upper-one*[*simp*]: *upper 1* = *1*
by *transfer auto*
instance proof qed
end

instance *interval* :: ({*one*, *preorder*, *linorder*, *times*}) *power*
proof qed

lemma *set-of-one*[*simp*]: *set-of* (*1*::′*a*::{*one*, *order*} *interval*) = {*1*}
by (*auto simp*: *set-of-eq*)

instance *interval* ::
({*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*}) *monoid-mult*
apply *standard*
unfolding *interval-eq-set-of-iff set-of-times*

subgoal
by (*auto simp*: *interval-eq-set-of-iff set-of-times*; *metis mult.assoc*)
by *auto*

lemma *one-times-ivl-left*[*simp*]: *1 * A = A* **for** *A* :: *'a*::*linordered-idom interval*
by (*simp add*: *interval-eq-iff lower-times upper-times ac-simps min-def max-def*)

lemma *one-times-ivl-right*[*simp*]: *A * 1 = A* **for** *A* :: *'a*::*linordered-idom interval*
by (*metis interval-mul-commute one-times-ivl-left*)

lemma *set-of-power-mono*: *a^n ∈ set-of (A^n)* **if** *a ∈ set-of A*
for *a* :: *'a*::*linordered-idom*
using *that*
by (*induction n*) (*auto intro*!: *times-in-intervalI*)

lemma *set-of-add-cong*:
set-of (A + B) = set-of (A' + B')
if *set-of A = set-of A' set-of B = set-of B'*
for *A* :: *'a*::*linordered-ab-group-add interval*
unfolding *set-of-plus that* **..**

lemma *set-of-add-inc-left*:
set-of (A + B) ⊆ set-of (A' + B)
if *set-of A ⊆ set-of A'*
for *A* :: *'a*::*linordered-ab-group-add interval*
unfolding *set-of-plus* **using** *that* **by** (*auto simp*: *set-plus-def*)

lemma *set-of-add-inc-right*:
set-of (A + B) ⊆ set-of (A + B')
if *set-of B ⊆ set-of B'*
for *A* :: *'a*::*linordered-ab-group-add interval*
using *set-of-add-inc-left*[*OF that*]
by (*simp add*: *add.commute*)

lemma *set-of-add-inc*:
set-of (A + B) ⊆ set-of (A' + B')
if *set-of A ⊆ set-of A' set-of B ⊆ set-of B'*
for *A* :: *'a*::*linordered-ab-group-add interval*
using *set-of-add-inc-left*[*OF that*(*1*)] *set-of-add-inc-right*[*OF that*(*2*)]
by *auto*

lemma *set-of-neg-inc*:
set-of (−A) ⊆ set-of (−A')
if *set-of A ⊆ set-of A'*
for *A* :: *'a*::*ordered-ab-group-add interval*
using *that*
unfolding *set-of-uminus*
by *auto*

lemma *set-of-sub-inc-left*:
set-of $(A - B) \subseteq$ *set-of* $(A' - B)$
if *set-of* $A \subseteq$ *set-of* A'
for A :: *'a*::*linordered-ab-group-add interval*
using *that*
unfolding *set-of-minus*
by *auto*

lemma *set-of-sub-inc-right*:
set-of $(A - B) \subseteq$ *set-of* $(A - B')$
if *set-of* $B \subseteq$ *set-of* B'
for A :: *'a*::*linordered-ab-group-add interval*
using *that*
unfolding *set-of-minus*
by *auto*

lemma *set-of-sub-inc*:
set-of $(A - B) \subseteq$ *set-of* $(A' - B')$
if *set-of* $A \subseteq$ *set-of* A' *set-of* $B \subseteq$ *set-of* B'
for A :: *'a*::*linordered-idom interval*
using *set-of-sub-inc-left*[*OF that*(*1*)] *set-of-sub-inc-right*[*OF that*(*2*)]
by *auto*

lemma *set-of-mul-inc-right*:
set-of $(A * B) \subseteq$ *set-of* $(A * B')$
if *set-of* $B \subseteq$ *set-of* B'
for A :: *'a*::*linordered-ring interval*
using *that*
apply *transfer*
apply (*clarsimp simp add*: *Let-def*)
by (*smt* (*verit*, *best*) *linorder-le-cases max.coboundedI1 max.coboundedI2 min.absorb1 min.coboundedI2 mult-left-mono mult-left-mono-neg*)

lemma *set-of-distrib-left*:
set-of $(B * (A1 + A2)) \subseteq$ *set-of* $(B * A1 + B * A2)$
for *A1* :: *'a*::*linordered-ring interval*
apply *transfer*
apply (*clarsimp simp*: *Let-def distrib-left distrib-right*)
apply (*intro conjI*)
apply (*metis add-mono min.cobounded1 min.left-commute*)
apply (*metis add-mono min.cobounded1 min.left-commute*)
apply (*metis add-mono min.cobounded1 min.left-commute*)
apply (*metis add-mono min.assoc min.cobounded2*)
apply (*meson add-mono order.trans max.cobounded1 max.cobounded2*)
apply (*meson add-mono order.trans max.cobounded1 max.cobounded2*)
apply (*meson add-mono order.trans max.cobounded1 max.cobounded2*)
apply (*meson add-mono order.trans max.cobounded1 max.cobounded2*)
done

lemma *set-of-distrib-right*:
set-of ((*A1* + *A2*) ∗ *B*) ⊆ *set-of* (*A1* ∗ *B* + *A2* ∗ *B*)
for *A1 A2 B* :: *'a*::{*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
unfolding *set-of-times set-of-plus set-plus-def*
using *distrib-right* **by** *blast*

lemma *set-of-mul-inc-left*:
set-of (*A* ∗ *B*) ⊆ *set-of* (*A'* ∗ *B*)
if *set-of A* ⊆ *set-of A'*
for *A* :: *'a*::{*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
using *that*
unfolding *set-of-times*
by *auto*

lemma *set-of-mul-inc*:
set-of (*A* ∗ *B*) ⊆ *set-of* (*A'* ∗ *B'*)
if *set-of A* ⊆ *set-of A' set-of B* ⊆ *set-of B'*
for *A* :: *'a*::{*linordered-ring*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
using *that* **unfolding** *set-of-times* **by** *auto*

lemma *set-of-pow-inc*:
set-of (*A*^*n*) ⊆ *set-of* (*A'*^*n*)
if *set-of A* ⊆ *set-of A'*
for *A* :: *'a*::{*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
using *that*
by (*induction n*, *simp-all add*: *set-of-mul-inc*)

lemma *set-of-distrib-right-left*:
set-of ((*A1* + *A2*) ∗ (*B1* + *B2*)) ⊆ *set-of* (*A1* ∗ *B1* + *A1* ∗ *B2* + *A2* ∗ *B1* + *A2* ∗ *B2*)
for *A1* :: *'a*::{*linordered-idom*, *real-normed-algebra*, *linear-continuum-topology*} *interval*
proof−
have *set-of* ((*A1* + *A2*) ∗ (*B1* + *B2*)) ⊆ *set-of* (*A1* ∗ (*B1* + *B2*) + *A2* ∗ (*B1* + *B2*))
by (*rule set-of-distrib-right*)
also have ... ⊆ *set-of* ((*A1* ∗ *B1* + *A1* ∗ *B2*) + *A2* ∗ (*B1* + *B2*))
by (*rule set-of-add-inc-left*[*OF set-of-distrib-left*])
also have ... ⊆ *set-of* ((*A1* ∗ *B1* + *A1* ∗ *B2*) + (*A2* ∗ *B1* + *A2* ∗ *B2*))
by (*rule set-of-add-inc-right*[*OF set-of-distrib-left*])
finally show *?thesis*
by (*simp add*: *add.assoc*)
qed

lemma *mult-bounds-enclose-zero1*:

*min (la * lb) (min (la * ub) (min (lb * ua) (ua * ub))) ≤ 0*
*0 ≤ max (la * lb) (max (la * ub) (max (lb * ua) (ua * ub)))*
if *la ≤ 0 0 ≤ ua*
for *la lb ua ub*:: *'a*::*linordered-idom*
subgoal by (*metis* (*no-types*, *opaque-lifting*) *that eq-iff min-le-iff-disj mult-zero-left mult-zero-right*
zero-le-mult-iff)
subgoal by (*metis that le-max-iff-disj mult-zero-right order-refl zero-le-mult-iff*)
done

lemma *mult-bounds-enclose-zero2*:
*min (la * lb) (min (la * ub) (min (lb * ua) (ua * ub))) ≤ 0*
*0 ≤ max (la * lb) (max (la * ub) (max (lb * ua) (ua * ub)))*
if *lb ≤ 0 0 ≤ ub*
for *la lb ua ub*:: *'a*::*linordered-idom*
using *mult-bounds-enclose-zero1*[*OF that*, *of la ua*]
by (*simp-all add*: *ac-simps*)

lemma *set-of-mul-contains-zero*:
*0 ∈ set-of (A * B)*
if *0 ∈ set-of A ∨ 0 ∈ set-of B*
for *A* :: *'a*::*linordered-idom interval*
using *that*
by (*auto simp*: *set-of-eq lower-times upper-times algebra-simps mult-le-0-iff*
mult-bounds-enclose-zero1 mult-bounds-enclose-zero2)

instance *interval* :: ({*linordered-semiring*, *zero*, *times*}) *mult-zero*
by (*standard*; *transfer*; *auto*)

lift-definition *min-interval*::*'a*::*linorder interval* ⇒ *'a interval* ⇒ *'a interval* **is**
λ(*l1*, *u1*). *λ*(*l2*, *u2*). (*min l1 l2*, *min u1 u2*)
by (*auto simp*: *min-def*)
lemma *lower-min-interval*[*simp*]: *lower* (*min-interval x y*) = *min* (*lower x*) (*lower y*)
by *transfer auto*
lemma *upper-min-interval*[*simp*]: *upper* (*min-interval x y*) = *min* (*upper x*) (*upper y*)
by *transfer auto*

lemma *min-intervalI*:
a ∈*i* *A* ⟹ *b* ∈*i* *B* ⟹ *min a b* ∈*i* *min-interval A B*
by (*auto simp*: *set-of-eq min-def*)

lift-definition *max-interval*::*'a*::*linorder interval* ⇒ *'a interval* ⇒ *'a interval* **is**
λ(*l1*, *u1*). *λ*(*l2*, *u2*). (*max l1 l2*, *max u1 u2*)
by (*auto simp*: *max-def*)
lemma *lower-max-interval*[*simp*]: *lower* (*max-interval x y*) = *max* (*lower x*) (*lower y*)
by *transfer auto*

lemma *upper-max-interval*[*simp*]: *upper* (*max-interval x y*) = *max* (*upper x*) (*upper y*)
by *transfer auto*

lemma *max-intervalI*:
a ∈$_i$ *A* ⟹ *b* ∈$_i$ *B* ⟹ *max a b* ∈$_i$ *max-interval A B*
by (*auto simp*: *set-of-eq max-def*)

lift-definition *abs-interval*::'*a*::*linordered-idom interval* ⇒ '*a interval* **is**
(λ(*l*,*u*). (*if l* < *0* ∧ *0* < *u then 0 else min* |*l*| |*u*|, *max* |*l*| |*u*|))
by *auto*

lemma *lower-abs-interval*[*simp*]:
lower (*abs-interval x*) = (*if lower x* < *0* ∧ *0* < *upper x then 0 else min* |*lower x*| |*upper x*|)
by *transfer auto*
lemma *upper-abs-interval*[*simp*]: *upper* (*abs-interval x*) = *max* |*lower x*| |*upper x*|
by *transfer auto*

lemma *in-abs-intervalI1*:
lx < *0* ⟹ *0* < *ux* ⟹ *0* ≤ *xa* ⟹ *xa* ≤ *max* (− *lx*) (*ux*) ⟹ *xa* ∈ *abs* ' {*lx*..*ux*}
for *xa*::'*a*::*linordered-idom*
by (*metis abs-minus-cancel abs-of-nonneg atLeastAtMost-iff image-eqI le-less le-max-iff-disj le-minus-iff neg-le-0-iff-le order-trans*)

lemma *in-abs-intervalI2*:
min (|*lx*|) |*ux*| ≤ *xa* ⟹ *xa* ≤ *max* |*lx*| |*ux*| ⟹ *lx* ≤ *ux* ⟹ *0* ≤ *lx* ∨ *ux* ≤ *0* ⟹
xa ∈ *abs* ' {*lx*..*ux*}
for *xa*::'*a*::*linordered-idom*
by (*force intro*: *image-eqI*[**where** *x*=−*xa*] *image-eqI*[**where** *x*=*xa*])

lemma *set-of-abs-interval*: *set-of* (*abs-interval x*) = *abs* ' *set-of x*
by (*auto simp*: *set-of-eq not-less intro*: *in-abs-intervalI1 in-abs-intervalI2 cong del*: *image-cong-simp*)

fun *split-domain* :: ('*a*::*preorder interval* ⇒ '*a interval list*) ⇒ '*a interval list* ⇒ '*a interval list list*
where *split-domain split* [] = [[]]
| *split-domain split* (*I*#*Is*) = (
let S = *split I*;
D = *split-domain split Is*
in concat (*map* (λ*d*. *map* (λ*s*. *s* # *d*) *S*) *D*)
)

context notes [[*typedef-overloaded*]] **begin**
lift-definition(*code-dt*) *split-interval*::'*a*::*linorder interval* ⇒ '*a* ⇒ ('*a interval* × '*a interval*)
is λ(*l*, *u*) *x*. ((*min l x*, *max l x*), (*min u x*, *max u x*))

```
  by (auto simp: min-def)
end

lemma split-domain-nonempty:
  assumes ⋀I. split I ≠ []
  shows split-domain split I ≠ []
  using last-in-set assms
  by (induction I, auto)

lemma lower-split-interval1: lower (fst (split-interval X m)) = min (lower X) m
  and lower-split-interval2: lower (snd (split-interval X m)) = min (upper X) m
  and upper-split-interval1: upper (fst (split-interval X m)) = max (lower X) m
  and upper-split-interval2: upper (snd (split-interval X m)) = max (upper X) m
  subgoal by transfer auto
  subgoal by transfer (auto simp: min.commute)
  subgoal by transfer auto
  subgoal by transfer auto
  done

lemma split-intervalD: split-interval X x = (A, B) ⟹ set-of X ⊆ set-of A ∪ set-of
B
  unfolding set-of-eq
  by transfer (auto simp: min-def max-def split: if-splits)

instantiation interval :: ({topological-space, preorder}) topological-space
begin

definition open-interval-def [code drop]: open (X::'a interval set) =
  (∀ x∈X.
     ∃ A B.
       open A ∧
       open B ∧
       lower x ∈ A ∧ upper x ∈ B ∧ Interval ' (A × B) ⊆ X)

instance
proof
  show open (UNIV :: ('a interval) set)
    unfolding open-interval-def by auto
next
  fix S T :: ('a interval) set
  assume open S open T
  show open (S ∩ T)
    unfolding open-interval-def
  proof (safe)
    fix x assume x ∈ S x ∈ T
    from ‹x ∈ S› ‹open S› obtain Sl Su where S:
      open Sl open Su lower x ∈ Sl upper x ∈ Su Interval ' (Sl × Su) ⊆ S
      by (auto simp: open-interval-def)
    from ‹x ∈ T› ‹open T› obtain Tl Tu where T:
```

open Tl open Tu lower x ∈ Tl upper x ∈ Tu Interval ‘ (Tl × Tu) ⊆ T
by (*auto simp*: *open-interval-def*)

let *?L = Sl ∩ Tl* **and** *?U = Su ∩ Tu*
have *open ?L ∧ open ?U ∧ lower x ∈ ?L ∧ upper x ∈ ?U ∧ Interval ‘ (?L × ?U) ⊆ S ∩ T*
using *S T* **by** (*auto simp add*: *open-Int*)
then show *∃ A B. open A ∧ open B ∧ lower x ∈ A ∧ upper x ∈ B ∧ Interval ‘ (A × B) ⊆ S ∩ T*
by *fast*
qed
qed (*unfold open-interval-def*, *fast*)

end

52.2 Quickcheck

lift-definition *Ivl*::*'a* ⇒ *'a*::*preorder* ⇒ *'a interval* **is** *λa b.* (*min a b*, *b*)
by (*auto simp*: *min-def*)

instantiation *interval* :: ({*exhaustive*,*preorder*}) *exhaustive*
begin

definition *exhaustive-interval*::(*'a interval* ⇒ (*bool* × *term list*) *option*)
⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
exhaustive-interval f d =
Quickcheck-Exhaustive.exhaustive (*λx. Quickcheck-Exhaustive.exhaustive* (*λy. f* (*Ivl x y*)) *d*) *d*

instance ..

end

context
includes *term-syntax*
begin

definition [*code-unfold*]:
valtermify-interval x y = *Code-Evaluation.valtermify* (*Ivl*::*'a*::{*preorder*,*typerep*}⇒-) {·} *x* {·} *y*

end

instantiation *interval* :: ({*full-exhaustive*,*preorder*,*typerep*}) *full-exhaustive*
begin

definition *full-exhaustive-interval*::
(*'a interval* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*)

```
    ⇒ natural ⇒ (bool × term list) option where
  full-exhaustive-interval f d =
    Quickcheck-Exhaustive.full-exhaustive
     (λx. Quickcheck-Exhaustive.full-exhaustive (λy. f (valtermify-interval x y)) d)
d

instance ..

end

instantiation interval :: ({random,preorder,typerep}) random
begin

definition random-interval ::
  natural
  ⇒ natural × natural
    ⇒ ('a interval × (unit ⇒ term)) × natural × natural where
  random-interval i =
  scomp (Quickcheck-Random.random i)
   (λman. scomp (Quickcheck-Random.random i) (λexp. Pair (valtermify-interval
man exp)))

instance ..

end

lifting-update interval.lifting
lifting-forget interval.lifting

end
```

53 Approximate Operations on Intervals of Floating Point Numbers

```
theory Interval-Float
  imports
    Interval
    Float
begin

definition mid :: float interval ⇒ float
  where mid i = (lower i + upper i) * Float 1 (−1)

lemma mid-in-interval: mid i ∈i i
  using lower-le-upper[of i]
  by (auto simp: mid-def set-of-eq powr-minus)

lemma mid-le: lower i ≤ mid i mid i ≤ upper i
```

using *mid-in-interval*
by (*auto simp*: *set-of-eq*)

definition *centered* :: *float interval* ⇒ *float interval*
where *centered i* = *i* − *interval-of* (*mid i*)

definition *split-float-interval x* = *split-interval x* ((*lower x* + *upper x*) ∗ *Float 1* (−*1*))

lemma *split-float-intervalD*: *split-float-interval X* = (*A*, *B*) ⟹ *set-of X* ⊆ *set-of A* ∪ *set-of B*
by (*auto dest*!: *split-intervalD simp*: *split-float-interval-def*)

lemma *split-float-interval-bounds*:
shows
lower-split-float-interval1: *lower* (*fst* (*split-float-interval X*)) = *lower X*
and *lower-split-float-interval2*: *lower* (*snd* (*split-float-interval X*)) = *mid X*
and *upper-split-float-interval1*: *upper* (*fst* (*split-float-interval X*)) = *mid X*
and *upper-split-float-interval2*: *upper* (*snd* (*split-float-interval X*)) = *upper X*
using *mid-le*[*of X*]
by (*auto simp*: *split-float-interval-def mid-def*[*symmetric*] *min-def max-def real-of-float-eq*
lower-split-interval1 lower-split-interval2
upper-split-interval1 upper-split-interval2)

lemmas *float-round-down-le*[*intro*] = *order-trans*[*OF float-round-down*]
and *float-round-up-ge*[*intro*] = *order-trans*[*OF - float-round-up*]

TODO: many of the lemmas should move to theories Float or Approximation (the latter should be based on type *interval*.

53.1 Intervals with Floating Point Bounds

context includes *interval.lifting* **begin**

lift-definition *round-interval* :: *nat* ⇒ *float interval* ⇒ *float interval*
is λ*p*. λ(*l*, *u*). (*float-round-down p l*, *float-round-up p u*)
by (*auto simp*: *intro*!: *float-round-down-le float-round-up-le*)

lemma *lower-round-ivl*[*simp*]: *lower* (*round-interval p x*) = *float-round-down p* (*lower x*)
by *transfer auto*
lemma *upper-round-ivl*[*simp*]: *upper* (*round-interval p x*) = *float-round-up p* (*upper x*)
by *transfer auto*

lemma *round-ivl-correct*: *set-of A* ⊆ *set-of* (*round-interval prec A*)
by (*auto simp*: *set-of-eq float-round-down-le float-round-up-le*)

lift-definition *truncate-ivl* :: *nat* ⇒ *real interval* ⇒ *real interval*

is $\lambda p.\ \lambda(l,\ u).\ (truncate\text{-}down\ p\ l,\ truncate\text{-}up\ p\ u)$
by (*auto intro*!: *truncate-down-le truncate-up-le*)

lemma *lower-truncate-ivl*[*simp*]: *lower* (*truncate-ivl p x*) = *truncate-down p* (*lower x*)
by *transfer auto*
lemma *upper-truncate-ivl*[*simp*]: *upper* (*truncate-ivl p x*) = *truncate-up p* (*upper x*)
by *transfer auto*

lemma *truncate-ivl-correct*: *set-of A* ⊆ *set-of* (*truncate-ivl prec A*)
by (*auto simp*: *set-of-eq intro*!: *truncate-down-le truncate-up-le*)

lift-definition *real-interval*::*float interval* ⇒ *real interval*
is $\lambda(l,\ u).\ (real\text{-}of\text{-}float\ l,\ real\text{-}of\text{-}float\ u)$
by *auto*

lemma *lower-real-interval*[*simp*]: *lower* (*real-interval x*) = *lower x*
by *transfer auto*
lemma *upper-real-interval*[*simp*]: *upper* (*real-interval x*) = *upper x*
by *transfer auto*

definition *set-of'* *x* = (*case x of None* ⇒ *UNIV* | *Some i* ⇒ *set-of* (*real-interval i*))

lemma *real-interval-min-interval*[*simp*]:
real-interval (*min-interval a b*) = *min-interval* (*real-interval a*) (*real-interval b*)
by (*auto simp*: *interval-eq-set-of-iff set-of-eq real-of-float-min*)

lemma *real-interval-max-interval*[*simp*]:
real-interval (*max-interval a b*) = *max-interval* (*real-interval a*) (*real-interval b*)
by (*auto simp*: *interval-eq-set-of-iff set-of-eq real-of-float-max*)

lemma *in-intervalI*:
x ∈$_i$ *X* **if** *lower X* ≤ *x x* ≤ *upper X*
using *that* **by** (*auto simp*: *set-of-eq*)

abbreviation *in-real-interval* (‹(‹*notation*=‹*infix* ∈$_r$››-/ ∈$_r$ -)› [*51*, *51*] *50*)
where *x* ∈$_r$ *X* ≡ *x* ∈$_i$ *real-interval X*

lemma *in-real-intervalI*:
x ∈$_r$ *X* **if** *lower X* ≤ *x x* ≤ *upper X* **for** *x*::*real* **and** *X*::*float interval*
using *that*
by (*intro in-intervalI*) *auto*

53.2 intros for *real-interval*

lemma *in-round-intervalI*: *x* ∈$_r$ *A* ⟹ *x* ∈$_r$ (*round-interval prec A*)
by (*auto simp*: *set-of-eq float-round-down-le float-round-up-le*)

lemma *zero-in-float-intervalI*: $0 \in_r 0$
by (*auto simp*: *set-of-eq*)

lemma *plus-in-float-intervalI*: $a + b \in_r A + B$ **if** $a \in_r A$ $b \in_r B$
using *that*
by (*auto simp*: *set-of-eq*)

lemma *minus-in-float-intervalI*: $a - b \in_r A - B$ **if** $a \in_r A$ $b \in_r B$
using *that*
by (*auto simp*: *set-of-eq*)

lemma *uminus-in-float-intervalI*: $-a \in_r -A$ **if** $a \in_r A$
using *that*
by (*auto simp*: *set-of-eq*)

lemma *real-interval-times*: *real-interval* $(A * B)$ = *real-interval* A $*$ *real-interval* B
by (*auto simp*: *interval-eq-iff lower-times upper-times min-def max-def*)

lemma *times-in-float-intervalI*: $a * b \in_r A * B$ **if** $a \in_r A$ $b \in_r B$
using *times-in-intervalI*[*OF that*]
by (*auto simp*: *real-interval-times*)

lemma *real-interval-abs*: *real-interval* (*abs-interval* A) = *abs-interval* (*real-interval* A)
by (*auto simp*: *interval-eq-iff min-def max-def*)

lemma *abs-in-float-intervalI*: *abs* $a \in_r$ *abs-interval* A **if** $a \in_r A$
by (*auto simp*: *set-of-abs-interval real-interval-abs intro*!: *imageI that*)

lemma *interval-of*[*intro*,*simp*]: $x \in_r$ *interval-of* x
by (*auto simp*: *set-of-eq*)

lemma *split-float-interval-realD*: *split-float-interval* $X = (A,\ B) \Longrightarrow x \in_r X \Longrightarrow x \in_r A \vee x \in_r B$
by (*auto simp*: *set-of-eq prod-eq-iff split-float-interval-bounds*)

53.3 bounds for lists

lemma *lower-Interval*: *lower* (*Interval* x) = *fst* x
and *upper-Interval*: *upper* (*Interval* x) = *snd* x
if *fst* $x \leq$ *snd* x
using *that*
by (*auto simp*: *lower-def upper-def Interval-inverse split-beta'*)

definition *all-in-i* :: *'a*::*preorder list* ⇒ *'a interval list* ⇒ *bool*
(**infix** ‹(*all'-in*$_i$)› *50*)
where x *all-in*$_i$ I = (*length* x = *length* $I \wedge$ ($\forall\, i <$ *length* I. x ! $i \in_i I$! i))

definition *all-in* :: *real list* ⇒ *float interval list* ⇒ *bool*
(**infix** ‹(*all'-in*)› *50*)
where *x all-in I* = (*length x* = *length I* ∧ (∀ *i* < *length I*. *x* ! *i* ∈$_r$ *I* ! *i*))

definition *all-subset* :: *'a*::*order interval list* ⇒ *'a interval list* ⇒ *bool*
(**infix** ‹(*all'-subset*)› *50*)
where *I all-subset J* = (*length I* = *length J* ∧ (∀ *i* < *length I*. *set-of* (*I*!*i*) ⊆ *set-of* (*J*!*i*)))

lemmas [*simp*] = *all-in-def all-subset-def*

lemma *all-subsetD*:
assumes *I all-subset J*
assumes *x all-in I*
shows *x all-in J*
using *assms*
by (*auto simp*: *set-of-eq*; *fastforce*)

lemma *round-interval-mono*: *set-of* (*round-interval prec X*) ⊆ *set-of* (*round-interval prec Y*)
if *set-of X* ⊆ *set-of Y*
using *that*
by *transfer*
(*auto simp*: *float-round-down.rep-eq float-round-up.rep-eq truncate-down-mono truncate-up-mono*)

lemma *Ivl-simps*[*simp*]: *lower* (*Ivl a b*) = *min a b upper* (*Ivl a b*) = *b*
subgoal by *transfer simp*
subgoal by *transfer simp*
done

lemma *set-of-subset-iff*: *set-of X* ⊆ *set-of Y* ⟷ *lower Y* ≤ *lower X* ∧ *upper X* ≤ *upper Y*
for *X Y*::*'a*::*linorder interval*
by (*auto simp*: *set-of-eq subset-iff*)

lemma *set-of-subset-iff'*:
set-of a ⊆ *set-of* (*b* :: *'a* :: *linorder interval*) ⟷ *a* ≤ *b*
unfolding *less-eq-interval-def set-of-subset-iff* **..**

lemma *bounds-of-interval-eq-lower-upper*:
bounds-of-interval ivl = (*lower ivl*, *upper ivl*) **if** *lower ivl* ≤ *upper ivl*
using *that*
by (*auto simp*: *lower.rep-eq upper.rep-eq*)

lemma *real-interval-Ivl*: *real-interval* (*Ivl a b*) = *Ivl a b*
by *transfer* (*auto simp*: *min-def*)

lemma *set-of-mul-contains-real-zero*:
0 $\in_r$ (*A* ∗ *B*) **if** *0* $\in_r$ *A* ∨ *0* $\in_r$ *B*
using *that set-of-mul-contains-zero*[*of A B*]
by (*auto simp*: *set-of-eq*)

fun *subdivide-interval* :: *nat* ⇒ *float interval* ⇒ *float interval list*
where *subdivide-interval 0 I* = [*I*]
| *subdivide-interval* (*Suc n*) *I* = (
let m = *mid I*
in (*subdivide-interval n* (*Ivl* (*lower I*) *m*)) @ (*subdivide-interval n* (*Ivl m* (*upper I*)))
)

lemma *subdivide-interval-length*:
shows *length* (*subdivide-interval n I*) = *2*^*n*
by(*induction n arbitrary*: *I*, *simp-all add*: *Let-def*)

lemma *lower-le-mid*: *lower x* ≤ *mid x real-of-float* (*lower x*) ≤ *mid x*
and *mid-le-upper*: *mid x* ≤ *upper x real-of-float* (*mid x*) ≤ *upper x*
unfolding *mid-def*
subgoal by *transfer* (*auto simp*: *powr-neg-one*)
subgoal by *transfer* (*auto simp*: *powr-neg-one*)
subgoal by *transfer* (*auto simp*: *powr-neg-one*)
subgoal by *transfer* (*auto simp*: *powr-neg-one*)
done

lemma *subdivide-interval-correct*:
list-ex (λ*i*. *x* $\in_r$ *i*) (*subdivide-interval n I*) **if** *x* $\in_r$ *I* **for** *x*::*real*
using *that*
proof(*induction n arbitrary*: *x I*)
case *0*
then show *?case* **by** *simp*
next
case (*Suc n*)
from ‹*x* $\in_r$ *I*› **consider** *x* $\in_r$ *Ivl* (*lower I*) (*mid I*) | *x* $\in_r$ *Ivl* (*mid I*) (*upper I*)
by (*cases x* ≤ *real-of-float* (*mid I*))
(*auto simp*: *set-of-eq min-def lower-le-mid mid-le-upper*)
from *this*[*case-names lower upper*] **show** *?case*
by *cases* (*use Suc.IH* **in** ‹*auto simp*: *Let-def*›)
qed

fun *interval-list-union* :: *'a*::*lattice interval list* ⇒ *'a interval*
where *interval-list-union* [] = *undefined*
| *interval-list-union* [*I*] = *I*
| *interval-list-union* (*I*#*Is*) = *sup I* (*interval-list-union Is*)

lemma *interval-list-union-correct*:
assumes *S* ≠ []
assumes *i* < *length S*

```
  shows set-of (S!i) ⊆ set-of (interval-list-union S)
  using assms
proof(induction S arbitrary: i)
  case (Cons a S i)
  thus ?case
  proof(cases S)
    fix b S'
    assume S = b # S'
    hence S ≠ []
      by simp
    show ?thesis
    proof(cases i)
      case 0
      show ?thesis
        apply(cases S)
        using interval-union-mono1
        by (auto simp add: 0)
    next
      case (Suc i-prev)
      hence i-prev < length S
        using Cons(3) by simp

      from Cons(1)[OF ‹S ≠ []› this] Cons(1)
      have set-of ((a # S) ! i) ⊆ set-of (interval-list-union S)
        by (simp add: ‹i = Suc i-prev›)
      also have ... ⊆ set-of (interval-list-union (a # S))
        using ‹S ≠ []›
        apply(cases S)
        using interval-union-mono2
        by auto
      finally show ?thesis .
    qed
  qed simp
qed simp

lemma split-domain-correct:
  fixes x :: real list
  assumes x all-in I
  assumes split-correct: ⋀x a I. x ∈r I ⟹ list-ex (λi::float interval. x ∈r i) (split
I)
  shows list-ex (λs. x all-in s) (split-domain split I)
  using assms(1)
proof(induction I arbitrary: x)
  case (Cons I Is x)
  have x ≠ []
    using Cons(2) by auto
  obtain x' xs where x-decomp: x = x' # xs
    using ‹x ≠ []› list.exhaust by auto
  hence x' ∈r I xs all-in Is
```

using *Cons*(*2*)
by *auto*
show *?case*
using *Cons*(*1*)[*OF* ‹*xs all-in Is*›]
split-correct[*OF* ‹$x' \in_r I$›]
apply (*auto simp add*: *list-ex-iff set-of-eq*)
by (*smt* (*verit*, *ccfv-SIG*) *One-nat-def Suc-pred* ‹$x \neq []$› *le-simps*(*3*) *length-greater-0-conv length-tl linorder-not-less list.sel*(*3*) *neq0-conv nth-Cons' x-decomp*)
qed *simp*

lift-definition(*code-dt*) *inverse-float-interval*::*nat* ⇒ *float interval* ⇒ *float interval option* **is**
λ*prec* (*l*, *u*). *if* (*0* < *l* ∨ *u* < *0*) *then Some* (*float-divl prec 1 u*, *float-divr prec 1 l*) *else None*
by (*auto intro*!: *order-trans*[*OF float-divl*] *order-trans*[*OF* - *float-divr*] *simp*: *divide-simps*)

lemma *inverse-float-interval-eq-Some-conv*:
defines *one* ≡ (*1*::*float*)
shows
inverse-float-interval p X = *Some R* ⟷
(*lower X* > *0* ∨ *upper X* < *0*) ∧
lower R = *float-divl p one* (*upper X*) ∧
upper R = *float-divr p one* (*lower X*)
by *clarsimp* (*transfer fixing*: *one*, *force simp*: *one-def split*: *if-splits*)

lemma *inverse-float-interval*:
inverse ‘ *set-of* (*real-interval X*) ⊆ *set-of* (*real-interval Y*)
if *inverse-float-interval p X* = *Some Y*
using *that*
apply (*clarsimp simp*: *set-of-eq inverse-float-interval-eq-Some-conv*)
by (*intro order-trans*[*OF float-divl*] *order-trans*[*OF* - *float-divr*] *conjI*)
(*auto simp*: *divide-simps*)

lemma *inverse-float-intervalI*:
$x \in_r X \Longrightarrow$ *inverse x* ∈ *set-of'* (*inverse-float-interval p X*)
using *inverse-float-interval*[*of p X*]
by (*auto simp*: *set-of'-def split*: *option.splits*)

lemma *inverse-float-interval-eqI*: *inverse-float-interval p X* = *Some IVL* ⟹ $x \in_r$ *X* ⟹ *inverse x* $\in_r$ *IVL*
using *inverse-float-intervalI*[*of x X p*]
by (*auto simp*: *set-of'-def*)

lemma *real-interval-abs-interval*[*simp*]:
real-interval (*abs-interval x*) = *abs-interval* (*real-interval x*)
by (*auto simp*: *interval-eq-set-of-iff set-of-eq real-of-float-max real-of-float-min*)

lift-definition *floor-float-interval*::*float interval* ⇒ *float interval* **is**
  λ(*l*, *u*). (*floor-fl l*, *floor-fl u*)
  by (*auto intro*!: *floor-mono simp*: *floor-fl.rep-eq*)

lemma *lower-floor-float-interval*[*simp*]: *lower* (*floor-float-interval x*) = *floor-fl* (*lower x*)
  by *transfer auto*
lemma *upper-floor-float-interval*[*simp*]: *upper* (*floor-float-interval x*) = *floor-fl* (*upper x*)
  by *transfer auto*

lemma *floor-float-intervalI*: ⌊*x*⌋ ∈$_r$ *floor-float-interval X* **if** *x* ∈$_r$ *X*
  using *that* **by** (*auto simp*: *set-of-eq floor-fl-def floor-mono*)

end

53.4 constants for code generation

definition *lowerF*::*float interval* ⇒ *float* **where** *lowerF* = *lower*
definition *upperF*::*float interval* ⇒ *float* **where** *upperF* = *upper*

end

54 Immutable Arrays with Code Generation

theory *IArray*
imports *Main*
begin

54.1 Fundamental operations

Immutable arrays are lists wrapped up in an additional constructor. There are no update operations. Hence code generation can safely implement this type by efficient target language arrays. Currently only SML is provided. Could be extended to other target languages and more operations.

context
begin

datatype *'a iarray* = *IArray 'a list*

qualified primrec *list-of* :: *'a iarray* ⇒ *'a list* **where**
list-of (*IArray xs*) = *xs*

qualified definition *of-fun* :: (*nat* ⇒ *'a*) ⇒ *nat* ⇒ *'a iarray* **where**
[*simp*]: *of-fun f n* = *IArray* (*map f* [*0*..<*n*])

qualified definition *sub* :: *'a iarray* ⇒ *nat* ⇒ *'a* (**infixl** ‹!!› *100*) **where**

[*simp*]: *as* !! *n* = *IArray.list-of as* ! *n*

qualified definition *length* :: *'a iarray* ⇒ *nat* **where**
[*simp*]: *length as* = *List.length* (*IArray.list-of as*)

qualified definition *all* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
[*simp*]: *all p as* ⟷ (∀ *a* ∈ *set* (*list-of as*). *p a*)

qualified definition *exists* :: (*'a* ⇒ *bool*) ⇒ *'a iarray* ⇒ *bool* **where**
[*simp*]: *exists p as* ⟷ (∃ *a* ∈ *set* (*list-of as*). *p a*)

lemma *of-fun-nth*:
IArray.of-fun f n !! *i* = *f i* **if** *i* < *n*
using *that* **by** (*simp add*: *map-nth*)

end

54.2 Generic code equations

lemma [*code*]:
size (*as* :: *'a iarray*) = *Suc* (*IArray.length as*)
by (*cases as*) *simp*

lemma [*code*]:
size-iarray f as = *Suc* (*size-list f* (*IArray.list-of as*))
by (*cases as*) *simp*

lemma [*code*]:
rec-iarray f as = *f* (*IArray.list-of as*)
by (*cases as*) *simp*

lemma [*code*]:
case-iarray f as = *f* (*IArray.list-of as*)
by (*cases as*) *simp*

lemma [*code*]:
set-iarray as = *set* (*IArray.list-of as*)
by (*cases as*) *auto*

lemma [*code*]:
map-iarray f as = *IArray* (*map f* (*IArray.list-of as*))
by (*cases as*) *auto*

lemma [*code*]:
rel-iarray r as bs = *list-all2 r* (*IArray.list-of as*) (*IArray.list-of bs*)
by (*cases as*, *cases bs*) *auto*

lemma *list-of-code* [*code*]:
IArray.list-of as = *map* (λ*n*. *as* !! *n*) [*0* ..< *IArray.length as*]

by (*cases as*) (*simp add*: *map-nth*)

lemma [*code*]:
HOL.equal as bs ⟷ *HOL.equal* (*IArray.list-of as*) (*IArray.list-of bs*)
by (*cases as*, *cases bs*) (*simp add*: *equal*)

lemma [*code*]:
IArray.all p = *Not* ∘ *IArray.exists* (*Not* ∘ *p*)
by (*simp add*: *fun-eq-iff*)

context
includes *term-syntax*
begin

lemma [*code*]:
Code-Evaluation.term-of (*as* :: *'a*::*typerep iarray*) =
Code-Evaluation.Const (*STR ''IArray.iarray.IArray''*) (*TYPEREP*(*'a list* ⇒ *'a iarray*)) <·> (*Code-Evaluation.term-of* (*IArray.list-of as*))
by (*subst term-of-anything*) *rule*

end

54.3 Auxiliary operations for code generation

context
begin

qualified primrec *tabulate* :: *integer* × (*integer* ⇒ *'a*) ⇒ *'a iarray* **where**
tabulate (*n*, *f*) = *IArray* (*map* (*f* ∘ *integer-of-nat*) [*0*..<*nat-of-integer n*])

lemma [*code*]:
IArray.of-fun f n = *IArray.tabulate* (*integer-of-nat n*, *f* ∘ *nat-of-integer*)
by *simp*

qualified primrec *sub'* :: *'a iarray* × *integer* ⇒ *'a* **where**
sub' (*as*, *n*) = *as* !! *nat-of-integer n*

lemma [*code*]:
IArray.sub' (*IArray as*, *n*) = *as* ! *nat-of-integer n*
by *simp*

lemma [*code*]:
as !! *n* = *IArray.sub'* (*as*, *integer-of-nat n*)
by *simp*

qualified definition *length'* :: *'a iarray* ⇒ *integer* **where**
[*simp*]: *length' as* = *integer-of-nat* (*List.length* (*IArray.list-of as*))

lemma [*code*]:

IArray.length′ (IArray as) = integer-of-nat (List.length as)
by *simp*

lemma [*code*]:
IArray.length as = nat-of-integer (IArray.length′ as)
by *simp*

qualified definition *exists-upto* :: *('a ⇒ bool) ⇒ integer ⇒ 'a iarray ⇒ bool*
where
[*simp*]: *exists-upto p k as ⟷ (∃ l. 0 ≤ l ∧ l < k ∧ p (sub′ (as, l)))*

lemma *exists-upto-of-nat*:
exists-upto p (of-nat n) as ⟷ (∃ m<n. p (as !! m))
including *integer.lifting* **by** (*simp*, *transfer*)
(*metis nat-int nat-less-iff of-nat-0-le-iff*)

lemma [*code*]:
exists-upto p k as ⟷ (if k ≤ 0 then False else
let l = k − 1 in p (sub′ (as, l)) ∨ exists-upto p l as)
proof (*cases k ≥ 1*)
case *False*
then have ‹*k ≤ 0*›
including *integer.lifting* **by** *transfer simp*
then show *?thesis*
by *simp*
next
case *True*
then have *less*: *k ≤ 0 ⟷ False*
by *simp*
define *n* **where** *n = nat-of-integer (k − 1)*
with *True* **have** *k*: *k − 1 = of-nat n k = of-nat (Suc n)*
by *simp-all*
show *?thesis* **unfolding** *less Let-def k(1)* **unfolding** *k(2) exists-upto-of-nat*
using *less-Suc-eq* **by** *auto*
qed

lemma [*code*]:
IArray.exists p as ⟷ exists-upto p (length′ as) as
including *integer.lifting* **by** (*simp*, *transfer*)
(*auto*, *metis in-set-conv-nth less-imp-of-nat-less nat-int of-nat-0-le-iff*)

end

54.4 Code Generation for SML

Note that arrays cannot be printed directly but only by turning them into lists first. Arrays could be converted back into lists for printing if they were wrapped up in an additional constructor.

code-reserved (*SML*) *Vector*

code-printing
type-constructor *iarray* ⇀ (*SML*) - *Vector.vector*
| **constant** *IArray* ⇀ (*SML*) *Vector.fromList*
| **constant** *IArray.all* ⇀ (*SML*) *Vector.all*
| **constant** *IArray.exists* ⇀ (*SML*) *Vector.exists*
| **constant** *IArray.tabulate* ⇀ (*SML*) *Vector.tabulate*
| **constant** *IArray.sub'* ⇀ (*SML*) *Vector.sub*
| **constant** *IArray.length'* ⇀ (*SML*) *Vector.length*

54.5 Code Generation for Haskell

We map *'a iarray*s in Isabelle/HOL to *Data.Array.IArray.array* in Haskell. Performance mapping to *Data.Array.Unboxed.Array* and *Data.Array.Array* is similar.

code-printing
code-module *IArray* ⇀ (*Haskell*) ‹
module IArray(*IArray*, *tabulate*, *of-list*, *sub*, *length*) *where* {

import Prelude (*Bool*(*True*, *False*), *not*, *Maybe*(*Nothing*, *Just*),
Integer, (+), (−), (<), *fromInteger*, *toInteger*, *map*, *seq*, (.));
import qualified Prelude;
import qualified Data.Array.IArray;
import qualified Data.Array.Base;
import qualified Data.Ix;

newtype IArray e = *IArray* (*Data.Array.IArray.Array Integer e*);

tabulate :: (*Integer*, (*Integer* −> *e*)) −> *IArray e*;
tabulate (*k*, *f*) = *IArray* (*Data.Array.IArray.array* (*0*, *k* − *1*) (*map* (*i* −> *let fi* = *f i in fi* ‘*seq*‘ (*i*, *fi*)) [*0*..*k* − *1*]));

of-list :: [*e*] −> *IArray e*;
of-list l = *IArray* (*Data.Array.IArray.listArray* (*0*, (*toInteger* . *Prelude.length*) *l* − *1*) *l*);

sub :: (*IArray e*, *Integer*) −> *e*;
sub (*IArray v*, *i*) = *v* ‘*Data.Array.Base.unsafeAt*‘ *fromInteger i*;

length :: *IArray e* −> *Integer*;
length (*IArray v*) = *toInteger* (*Data.Ix.rangeSize* (*Data.Array.IArray.bounds v*));

}› **for type-constructor** *iarray* **constant** *IArray IArray.tabulate IArray.sub' IArray.length'*

code-reserved (*Haskell*) *IArray-Impl*

code-printing
type-constructor *iarray* ⇀ (*Haskell*) *IArray.IArray* -

| **constant** *IArray* ⇀ (*Haskell*) *IArray.of'-list*
| **constant** *IArray.tabulate* ⇀ (*Haskell*) *IArray.tabulate*
| **constant** *IArray.sub'* ⇀ (*Haskell*) *IArray.sub*
| **constant** *IArray.length'* ⇀ (*Haskell*) *IArray.length*

end

55 Definition of Landau symbols

theory *Landau-Symbols*
imports
Complex-Main
begin

lemma *eventually-subst'*:
eventually (λ*x*. *f x* = *g x*) *F* ⟹ *eventually* (λ*x*. *P x* (*f x*)) *F* = *eventually* (λ*x*. *P x* (*g x*)) *F*
by (*rule eventually-subst*, *erule eventually-rev-mp*) *simp*

55.1 Definition of Landau symbols

Our Landau symbols are sign-oblivious, i.e. any function always has the same growth as its absolute. This has the advantage of making some cancelling rules for sums nicer, but introduces some problems in other places. Nevertheless, we found this definition more convenient to work with.

definition *bigo* :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=1 notation=‹mixfix bigo›*›*O*[-]'(-'))›)
where *bigo F g* = {*f*. (∃ *c*>*0*. *eventually* (λ*x*. *norm* (*f x*) ≤ *c* ∗ *norm* (*g x*)) *F*)}

definition *smallo* :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=1 notation=‹mixfix smallo›*›*o*[-]'(-'))›)
where *smallo F g* = {*f*. (∀ *c*>*0*. *eventually* (λ*x*. *norm* (*f x*) ≤ *c* ∗ *norm* (*g x*)) *F*)}

definition *bigomega* :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=1 notation=‹mixfix bigomega›*›Ω[-]'(-'))›)
where *bigomega F g* = {*f*. (∃ *c*>*0*. *eventually* (λ*x*. *norm* (*f x*) ≥ *c* ∗ *norm* (*g x*)) *F*)}

definition *smallomega* :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=1 notation=‹mixfix smallomega›*›ω[-]'(-'))›)
where *smallomega F g* = {*f*. (∀ *c*>*0*. *eventually* (λ*x*. *norm* (*f x*) ≥ *c* ∗ *norm* (*g x*)) *F*)}

definition *bigtheta* :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
(‹(‹*indent=1 notation=‹mixfix bigtheta›*›Θ[-]'(-'))›)
where *bigtheta F g* = *bigo F g* ∩ *bigomega F g*

abbreviation *bigo-at-top* (‹(‹*indent=2 notation=‹mixfix bigo›*› $O'(\text{-}')$)›)
where $O(g) \equiv$ *bigo at-top g*

abbreviation *smallo-at-top* (‹(‹*indent=2 notation=‹mixfix smallo›*› $o'(\text{-}')$)›)
where $o(g) \equiv$ *smallo at-top g*

abbreviation *bigomega-at-top* (‹(‹*indent=2 notation=‹mixfix bigomega›*› $\Omega'(\text{-}')$)›)
where $\Omega(g) \equiv$ *bigomega at-top g*

abbreviation *smallomega-at-top* (‹(‹*indent=2 notation=‹mixfix smallomega›*› $\omega'(\text{-}')$)›)
where $\omega(g) \equiv$ *smallomega at-top g*

abbreviation *bigtheta-at-top* (‹(‹*indent=2 notation=‹mixfix bigtheta›*› $\Theta'(\text{-}')$)›)
where $\Theta(g) \equiv$ *bigtheta at-top g*

The following is a set of properties that all Landau symbols satisfy.

named-theorems *landau-divide-simps*

locale *landau-symbol* =
fixes L :: *'a filter* ⇒ (*'a* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'a* ⇒ *'b*) *set*
and L' :: *'c filter* ⇒ (*'c* ⇒ (*'b* :: *real-normed-field*)) ⇒ (*'c* ⇒ *'b*) *set*
and Lr :: *'a filter* ⇒ (*'a* ⇒ *real*) ⇒ (*'a* ⇒ *real*) *set*
assumes *bot'*: $L\ bot\ f = UNIV$
assumes *filter-mono'*: $F1 \leq F2 \Longrightarrow L\ F2\ f \subseteq L\ F1\ f$
assumes *in-filtermap-iff*:
$f' \in L\ (filtermap\ h'\ F')\ g' \longleftrightarrow (\lambda x.\ f'\ (h'\ x)) \in L'\ F'\ (\lambda x.\ g'\ (h'\ x))$
assumes *filtercomap*:
$f' \in L\ F''\ g' \Longrightarrow (\lambda x.\ f'\ (h'\ x)) \in L'\ (filtercomap\ h'\ F'')\ (\lambda x.\ g'\ (h'\ x))$
assumes *sup*: $f \in L\ F1\ g \Longrightarrow f \in L\ F2\ g \Longrightarrow f \in L\ (sup\ F1\ F2)\ g$
assumes *in-cong*: $eventually\ (\lambda x.\ f\ x = g\ x)\ F \Longrightarrow f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$
assumes *cong*: $eventually\ (\lambda x.\ f\ x = g\ x)\ F \Longrightarrow L\ F\ (f) = L\ F\ (g)$
assumes *cong-bigtheta*: $f \in \Theta[F](g) \Longrightarrow L\ F\ (f) = L\ F\ (g)$
assumes *in-cong-bigtheta*: $f \in \Theta[F](g) \Longrightarrow f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$
assumes *cmult* [*simp*]: $c \neq 0 \Longrightarrow L\ F\ (\lambda x.\ c * f\ x) = L\ F\ (f)$
assumes *cmult-in-iff* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ c * f\ x) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
assumes *mult-left* [*simp*]: $f \in L\ F\ (g) \Longrightarrow (\lambda x.\ h\ x * f\ x) \in L\ F\ (\lambda x.\ h\ x * g\ x)$
assumes *inverse*: $eventually\ (\lambda x.\ f\ x \neq 0)\ F \Longrightarrow eventually\ (\lambda x.\ g\ x \neq 0)\ F \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x.\ inverse\ (g\ x)) \in L\ F\ (\lambda x.\ inverse\ (f\ x))$
assumes *subsetI*: $f \in L\ F\ (g) \Longrightarrow L\ F\ (f) \subseteq L\ F\ (g)$
assumes *plus-subset1*: $f \in o[F](g) \Longrightarrow L\ F\ (g) \subseteq L\ F\ (\lambda x.\ f\ x + g\ x)$
assumes *trans*: $f \in L\ F\ (g) \Longrightarrow g \in L\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
assumes *compose*: $f \in L\ F\ (g) \Longrightarrow filterlim\ h'\ F\ G \Longrightarrow (\lambda x.\ f\ (h'\ x)) \in L'\ G\ (\lambda x.\ g\ (h'\ x))$
assumes *norm-iff* [*simp*]: $(\lambda x.\ norm\ (f\ x)) \in Lr\ F\ (\lambda x.\ norm\ (g\ x)) \longleftrightarrow f \in L\ F\ (g)$
assumes *abs* [*simp*]: $Lr\ Fr\ (\lambda x.\ |fr\ x|) = Lr\ Fr\ fr$

assumes *abs-in-iff* [*simp*]: $(\lambda x.\ |fr\ x|) \in Lr\ Fr\ gr \longleftrightarrow fr \in Lr\ Fr\ gr$
begin

lemma *bot* [*simp*]: $f \in L\ bot\ g$ **by** (*simp add*: *bot'*)

lemma *filter-mono*: $F1 \leq F2 \Longrightarrow f \in L\ F2\ g \Longrightarrow f \in L\ F1\ g$
using *filter-mono'*[*of F1 F2*] **by** *blast*

lemma *cong-ex*:
$eventually\ (\lambda x.\ f1\ x = f2\ x)\ F \Longrightarrow eventually\ (\lambda x.\ g1\ x = g2\ x)\ F \Longrightarrow$
$f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
by (*subst cong*, *assumption*, *subst in-cong*, *assumption*, *rule refl*)

lemma *cong-ex-bigtheta*:
$f1 \in \Theta[F](f2) \Longrightarrow g1 \in \Theta[F](g2) \Longrightarrow f1 \in L\ F\ (g1) \longleftrightarrow f2 \in L\ F\ (g2)$
by (*subst cong-bigtheta*, *assumption*, *subst in-cong-bigtheta*, *assumption*, *rule refl*)

lemma *bigtheta-trans1*:
$f \in L\ F\ (g) \Longrightarrow g \in \Theta[F](h) \Longrightarrow f \in L\ F\ (h)$
by (*subst cong-bigtheta*[*symmetric*])

lemma *bigtheta-trans2*:
$f \in \Theta[F](g) \Longrightarrow g \in L\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
by (*subst in-cong-bigtheta*)

lemma *cmult'* [*simp*]: $c \neq 0 \Longrightarrow L\ F\ (\lambda x.\ f\ x * c) = L\ F\ (f)$
by (*subst mult.commute*) (*rule cmult*)

lemma *cmult-in-iff'* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ f\ x * c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
by (*subst mult.commute*) (*rule cmult-in-iff*)

lemma *cdiv* [*simp*]: $c \neq 0 \Longrightarrow L\ F\ (\lambda x.\ f\ x\ /\ c) = L\ F\ (f)$
using *cmult'*[*of inverse c F f*] **by** (*simp add*: *field-simps*)

lemma *cdiv-in-iff'* [*simp*]: $c \neq 0 \Longrightarrow (\lambda x.\ f\ x\ /\ c) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
using *cmult-in-iff'*[*of inverse c f*] **by** (*simp add*: *field-simps*)

lemma *uminus* [*simp*]: $L\ F\ (\lambda x.\ -g\ x) = L\ F\ (g)$ **using** *cmult*[*of −1*] **by** *simp*

lemma *uminus-in-iff* [*simp*]: $(\lambda x.\ -f\ x) \in L\ F\ (g) \longleftrightarrow f \in L\ F\ (g)$
using *cmult-in-iff*[*of −1*] **by** *simp*

lemma *const*: $c \neq 0 \Longrightarrow L\ F\ (\lambda\text{-}.\ c) = L\ F\ (\lambda\text{-}.\ 1)$
by (*subst* (*2*) *cmult*[*symmetric*]) *simp-all*

lemma *const'* [*simp*]: *NO-MATCH 1 c* $\Longrightarrow c \neq 0 \Longrightarrow L\ F\ (\lambda\text{-}.\ c) = L\ F\ (\lambda\text{-}.\ 1)$
by (*rule const*)

lemma *const-in-iff*: $c \neq 0 \Longrightarrow (\lambda\text{-}.\ c) \in L\ F\ (f) \longleftrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (f)$
using *cmult-in-iff'*[*of c* $\lambda\text{-}.\ 1$] **by** *simp*

lemma *const-in-iff'* [*simp*]: *NO-MATCH 1 c* $\Longrightarrow c \neq 0 \Longrightarrow (\lambda\text{-}.\ c) \in L\ F\ (f) \longleftrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (f)$
by (*rule const-in-iff*)

lemma *plus-subset2*: $g \in o[F](f) \Longrightarrow L\ F\ (f) \subseteq L\ F\ (\lambda x.\ f\ x + g\ x)$
by (*subst add.commute*) (*rule plus-subset1*)

lemma *mult-right* [*simp*]: $f \in L\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in L\ F\ (\lambda x.\ g\ x * h\ x)$
using *mult-left* **by** (*simp add*: *mult.commute*)

lemma *mult*: *f1* $\in L\ F\ (g1) \Longrightarrow$ *f2* $\in L\ F\ (g2) \Longrightarrow (\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x)$
by (*rule trans*, *erule mult-left*, *erule mult-right*)

lemma *inverse-cancel*:
assumes *eventually* $(\lambda x.\ f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
shows $(\lambda x.\ inverse\ (f\ x)) \in L\ F\ (\lambda x.\ inverse\ (g\ x)) \longleftrightarrow g \in L\ F\ (f)$
proof
assume $(\lambda x.\ inverse\ (f\ x)) \in L\ F\ (\lambda x.\ inverse\ (g\ x))$
from *inverse*[*OF - - this*] *assms* **show** $g \in L\ F\ (f)$ **by** *simp*
qed (*intro inverse assms*)

lemma *divide-right*:
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
assumes $f \in L\ F\ (g)$
shows $(\lambda x.\ f\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x)$
by (*subst* (*1 2*) *divide-inverse*) (*intro mult-right inverse assms*)

lemma *divide-right-iff*:
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
shows $(\lambda x.\ f\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x) \longleftrightarrow f \in L\ F\ (g)$
proof
assume $(\lambda x.\ f\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x)$
from *mult-right*[*OF this*, *of h*] *assms* **show** $f \in L\ F\ (g)$
by (*subst* (*asm*) *cong-ex*[*of - f F - g*]) (*auto elim*!: *eventually-mono*)
qed (*simp add*: *divide-right assms*)

lemma *divide-left*:
assumes *eventually* $(\lambda x.\ f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
assumes $g \in L\ F(f)$
shows $(\lambda x.\ h\ x\ /\ f\ x) \in L\ F\ (\lambda x.\ h\ x\ /\ g\ x)$
by (*subst* (*1 2*) *divide-inverse*) (*intro mult-left inverse assms*)

lemma *divide-left-iff*:

assumes *eventually* $(\lambda x.\ f\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
shows $(\lambda x.\ h\ x\ /\ f\ x) \in L\ F\ (\lambda x.\ h\ x\ /\ g\ x) \longleftrightarrow g \in L\ F\ (f)$
proof
assume *A*: $(\lambda x.\ h\ x\ /\ f\ x) \in L\ F\ (\lambda x.\ h\ x\ /\ g\ x)$
from *assms* **have** *B*: *eventually* $(\lambda x.\ h\ x\ /\ f\ x\ /\ h\ x = inverse\ (f\ x))\ F$
by *eventually-elim* (*simp add*: *divide-inverse*)
from *assms* **have** *C*: *eventually* $(\lambda x.\ h\ x\ /\ g\ x\ /\ h\ x = inverse\ (g\ x))\ F$
by *eventually-elim* (*simp add*: *divide-inverse*)
from *divide-right*[*OF assms*(*3*) *A*] *assms* **show** $g \in L\ F\ (f)$
by (*subst* (*asm*) *cong-ex*[*OF B C*]) (*simp add*: *inverse-cancel*)
qed (*simp add*: *divide-left assms*)

lemma *divide*:
assumes *eventually* $(\lambda x.\ g1\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ g2\ x \neq 0)\ F$
assumes $f1 \in L\ F\ (f2)$ $g2 \in L\ F\ (g1)$
shows $(\lambda x.\ f1\ x\ /\ g1\ x) \in L\ F\ (\lambda x.\ f2\ x\ /\ g2\ x)$
by (*subst* (*1 2*) *divide-inverse*) (*intro mult inverse assms*)

lemma *divide-eq1*:
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
shows $f \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x) \longleftrightarrow (\lambda x.\ f\ x * h\ x) \in L\ F\ (g)$
proof−
have $f \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x) \longleftrightarrow (\lambda x.\ f\ x * h\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x\ /\ h\ x)$
using *assms* **by** (*intro in-cong*) (*auto elim*: *eventually-mono*)
thus *?thesis* **by** (*simp only*: *divide-right-iff assms*)
qed

lemma *divide-eq2*:
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
shows $(\lambda x.\ f\ x\ /\ h\ x) \in L\ F\ (\lambda x.\ g\ x) \longleftrightarrow f \in L\ F\ (\lambda x.\ g\ x * h\ x)$
proof−
have $L\ F\ (\lambda x.\ g\ x) = L\ F\ (\lambda x.\ g\ x * h\ x\ /\ h\ x)$
using *assms* **by** (*intro cong*) (*auto elim*: *eventually-mono*)
thus *?thesis* **by** (*simp only*: *divide-right-iff assms*)
qed

lemma *inverse-eq1*:
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
shows $f \in L\ F\ (\lambda x.\ inverse\ (g\ x)) \longleftrightarrow (\lambda x.\ f\ x * g\ x) \in L\ F\ (\lambda\text{-}.\ 1)$
using *divide-eq1*[*of g F f* λ*-. 1*] **by** (*simp add*: *divide-inverse assms*)

lemma *inverse-eq2*:
assumes *eventually* $(\lambda x.\ f\ x \neq 0)\ F$
shows $(\lambda x.\ inverse\ (f\ x)) \in L\ F\ (g) \longleftrightarrow (\lambda x.\ 1) \in L\ F\ (\lambda x.\ f\ x * g\ x)$
using *divide-eq2*[*of f F* λ*-. 1 g*] **by** (*simp add*: *divide-inverse assms mult-ac*)

lemma *inverse-flip*:
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
assumes *eventually* $(\lambda x.\ h\ x \neq 0)\ F$
assumes $(\lambda x.\ inverse\ (g\ x)) \in L\ F\ (h)$
shows $(\lambda x.\ inverse\ (h\ x)) \in L\ F\ (g)$
using *assms* **by** (*simp add*: *divide-eq1 divide-eq2 inverse-eq-divide mult.commute*)

lemma *lift-trans*:
assumes $f \in L\ F\ (g)$
assumes $(\lambda x.\ t\ x\ (g\ x)) \in L\ F\ (h)$
assumes $\bigwedge f\ g.\ f \in L\ F\ (g) \Longrightarrow (\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (\lambda x.\ t\ x\ (g\ x))$
shows $(\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (h)$
by (*rule trans*[*OF assms*(*3*)[*OF assms*(*1*)] *assms*(*2*)])

lemma *lift-trans'*:
assumes $f \in L\ F\ (\lambda x.\ t\ x\ (g\ x))$
assumes $g \in L\ F\ (h)$
assumes $\bigwedge g\ h.\ g \in L\ F\ (h) \Longrightarrow (\lambda x.\ t\ x\ (g\ x)) \in L\ F\ (\lambda x.\ t\ x\ (h\ x))$
shows $f \in L\ F\ (\lambda x.\ t\ x\ (h\ x))$
by (*rule trans*[*OF assms*(*1*) *assms*(*3*)[*OF assms*(*2*)]])

lemma *lift-trans-bigtheta*:
assumes $f \in L\ F\ (g)$
assumes $(\lambda x.\ t\ x\ (g\ x)) \in \Theta[F](h)$
assumes $\bigwedge f\ g.\ f \in L\ F\ (g) \Longrightarrow (\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (\lambda x.\ t\ x\ (g\ x))$
shows $(\lambda x.\ t\ x\ (f\ x)) \in L\ F\ (h)$
using *cong-bigtheta*[*OF assms*(*2*)] *assms*(*3*)[*OF assms*(*1*)] **by** *simp*

lemma *lift-trans-bigtheta'*:
assumes $f \in L\ F\ (\lambda x.\ t\ x\ (g\ x))$
assumes $g \in \Theta[F](h)$
assumes $\bigwedge g\ h.\ g \in \Theta[F](h) \Longrightarrow (\lambda x.\ t\ x\ (g\ x)) \in \Theta[F](\lambda x.\ t\ x\ (h\ x))$
shows $f \in L\ F\ (\lambda x.\ t\ x\ (h\ x))$
using *cong-bigtheta*[*OF assms*(*3*)[*OF assms*(*2*)]] *assms*(*1*) **by** *simp*

lemma (**in** *landau-symbol*) *mult-in-1*:
assumes $f \in L\ F\ (\lambda\text{-}.\ 1)$ $g \in L\ F\ (\lambda\text{-}.\ 1)$
shows $(\lambda x.\ f\ x * g\ x) \in L\ F\ (\lambda\text{-}.\ 1)$
using *mult*[*OF assms*] **by** *simp*

lemma (**in** *landau-symbol*) *of-real-cancel*:
$(\lambda x.\ of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x.\ of\text{-}real\ (g\ x)) \Longrightarrow f \in Lr\ F\ g$
by (*subst* (*asm*) *norm-iff* [*symmetric*], *subst* (*asm*) (*1 2*) *norm-of-real*) *simp-all*

lemma (**in** *landau-symbol*) *of-real-iff*:
$(\lambda x.\ of\text{-}real\ (f\ x)) \in L\ F\ (\lambda x.\ of\text{-}real\ (g\ x)) \longleftrightarrow f \in Lr\ F\ g$
by (*subst norm-iff* [*symmetric*], *subst* (*1 2*) *norm-of-real*) *simp-all*

lemmas [*landau-divide-simps*] =

inverse-cancel divide-left-iff divide-eq1 divide-eq2 inverse-eq1 inverse-eq2

end

The symbols O and o and Ω and ω are dual, so for many rules, replacing O with Ω, o with ω, and $\leq$ with $\geq$ in a theorem yields another valid theorem. The following locale captures this fact.

locale *landau-pair* =
fixes $L\ l :: 'a\ filter \Rightarrow ('a \Rightarrow ('b :: real\text{-}normed\text{-}field)) \Rightarrow ('a \Rightarrow 'b)\ set$
fixes $L'\ l' :: 'c\ filter \Rightarrow ('c \Rightarrow ('b :: real\text{-}normed\text{-}field)) \Rightarrow ('c \Rightarrow 'b)\ set$
fixes $Lr\ lr :: 'a\ filter \Rightarrow ('a \Rightarrow real) \Rightarrow ('a \Rightarrow real)\ set$
and $R :: real \Rightarrow real \Rightarrow bool$
assumes *L-def*: $L\ F\ g = \{f.\ \exists\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F\}$
and *l-def*: $l\ F\ g = \{f.\ \forall\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F\}$
and *L'-def*: $L'\ F'\ g' = \{f.\ \exists\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g'\ x)))\ F'\}$
and *l'-def*: $l'\ F'\ g' = \{f.\ \forall\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g'\ x)))\ F'\}$
and *Lr-def*: $Lr\ F''\ g'' = \{f.\ \exists\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g''\ x)))\ F''\}$
and *lr-def*: $lr\ F''\ g'' = \{f.\ \forall\, c{>}0.\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g''\ x)))\ F''\}$
and *R*: $R = (\leq) \vee R = (\geq)$

interpretation *landau-o*:
landau-pair bigo smallo bigo smallo bigo smallo $(\leq)$
by *unfold-locales* (*auto simp*: *bigo-def smallo-def intro*!: *ext*)

interpretation *landau-omega*:
landau-pair bigomega smallomega bigomega smallomega bigomega smallomega $(\geq)$
by *unfold-locales* (*auto simp*: *bigomega-def smallomega-def intro*!: *ext*)

context *landau-pair*
begin

lemmas *R-E* = *disjE* [*OF R*, *case-names le ge*]

lemma *bigI*:
$c > 0 \Longrightarrow eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x)))\ F \Longrightarrow f \in L\ F\ (g)$
unfolding *L-def* **by** *blast*

lemma *bigE*:
assumes $f \in L\ F\ (g)$
obtains *c* **where** $c > 0\ eventually\ (\lambda x.\ R\ (norm\ (f\ x))\ (c * (norm\ (g\ x))))\ F$
using *assms* **unfolding** *L-def* **by** *blast*

lemma *smallI*:
(⋀*c*. *c* > *0* ⟹ *eventually* (λ*x*. *R* (*norm* (*f x*)) (*c* ∗ (*norm* (*g x*)))) *F*) ⟹ *f* ∈ *l F* (*g*)
unfolding *l-def* **by** *blast*

lemma *smallD*:
f ∈ *l F* (*g*) ⟹ *c* > *0* ⟹ *eventually* (λ*x*. *R* (*norm* (*f x*)) (*c* ∗ (*norm* (*g x*)))) *F*
unfolding *l-def* **by** *blast*

lemma *bigE-nonneg-real*:
assumes *f* ∈ *Lr F* (*g*) *eventually* (λ*x*. *f x* ≥ *0*) *F*
obtains *c* **where** *c* > *0 eventually* (λ*x*. *R* (*f x*) (*c* ∗ |*g x*|)) *F*
proof−
from *assms*(*1*) **obtain** *c* **where** *c*: *c* > *0 eventually* (λ*x*. *R* (*norm* (*f x*)) (*c* ∗ *norm* (*g x*))) *F*
by (*auto simp*: *Lr-def*)
from *c*(*2*) *assms*(*2*) **have** *eventually* (λ*x*. *R* (*f x*) (*c* ∗ |*g x*|)) *F*
by *eventually-elim simp*
from *c*(*1*) **and** *this* **show** *?thesis* **by** (*rule that*)
qed

lemma *smallD-nonneg-real*:
assumes *f* ∈ *lr F* (*g*) *eventually* (λ*x*. *f x* ≥ *0*) *F c* > *0*
shows *eventually* (λ*x*. *R* (*f x*) (*c* ∗ |*g x*|)) *F*
using *assms* **by** (*auto simp*: *lr-def dest*!: *spec*[*of - c*] *elim*: *eventually-elim2*)

lemma *small-imp-big*: *f* ∈ *l F* (*g*) ⟹ *f* ∈ *L F* (*g*)
by (*rule bigI*[*OF - smallD, of 1*]) *simp-all*

lemma *small-subset-big*: *l F* (*g*) ⊆ *L F* (*g*)
using *small-imp-big* **by** *blast*

lemma *R-refl* [*simp*]: *R x x* **using** *R* **by** *auto*

lemma *R-linear*: ¬*R x y* ⟹ *R y x*
using *R* **by** *auto*

lemma *R-trans* [*trans*]: *R a b* ⟹ *R b c* ⟹ *R a c*
using *R* **by** *auto*

lemma *R-mult-left-mono*: *R a b* ⟹ *c* ≥ *0* ⟹ *R* (*c*∗*a*) (*c*∗*b*)
using *R* **by** (*auto simp*: *mult-left-mono*)

lemma *R-mult-right-mono*: *R a b* ⟹ *c* ≥ *0* ⟹ *R* (*a*∗*c*) (*b*∗*c*)
using *R* **by** (*auto simp*: *mult-right-mono*)

lemma *big-trans*:
assumes *f* ∈ *L F* (*g*) *g* ∈ *L F* (*h*)

```
  shows   f ∈ L F (h)
proof−
  from assms obtain c d where ∗: 0 < c 0 < d
    and ∗∗: ∀F x in F. R (norm (f x)) (c ∗ norm (g x))
          ∀F x in F. R (norm (g x)) (d ∗ norm (h x))
    by (elim bigE)
  from ∗∗ have eventually (λx. R (norm (f x)) (c ∗ d ∗ (norm (h x)))) F
  proof eventually-elim
    fix x assume R (norm (f x)) (c ∗ (norm (g x)))
    also assume R (norm (g x)) (d ∗ (norm (h x)))
    with ‹0 < c› have R (c ∗ (norm (g x))) (c ∗ (d ∗ (norm (h x))))
      by (intro R-mult-left-mono) simp-all
    finally show R (norm (f x)) (c ∗ d ∗ (norm (h x)))
      by (simp add: algebra-simps)
  qed
  with ∗ show ?thesis by (intro bigI[of c∗d]) simp-all
qed

lemma big-small-trans:
  assumes f ∈ L F (g) g ∈ l F (h)
  shows   f ∈ l F (h)
proof (rule smallI)
  fix c :: real assume c: c > 0
  from assms(1) obtain d where d: d > 0 and ∗: ∀F x in F. R (norm (f x)) (d
∗ norm (g x))
    by (elim bigE)
  from assms(2) c d have eventually (λx. R (norm (g x)) (c ∗ inverse d ∗ norm
(h x))) F
    by (intro smallD) simp-all
  with ∗ show eventually (λx. R (norm (f x)) (c ∗ (norm (h x)))) F
  proof eventually-elim
    case (elim x)
    show ?case
     by (use elim(1) in ‹rule R-trans›) (use elim(2) R d in ‹auto simp: field-simps›)
  qed
qed

lemma small-big-trans:
  assumes f ∈ l F (g) g ∈ L F (h)
  shows   f ∈ l F (h)
proof (rule smallI)
  fix c :: real assume c: c > 0
  from assms(2) obtain d where d: d > 0 and ∗: ∀F x in F. R (norm (g x)) (d
∗ norm (h x))
    by (elim bigE)
  from assms(1) c d have eventually (λx. R (norm (f x)) (c ∗ inverse d ∗ norm
(g x))) F
    by (intro smallD) simp-all
  with ∗ show eventually (λx. R (norm (f x)) (c ∗ norm (h x))) F
```

by *eventually-elim* (*rotate-tac 2*, *erule R-trans*, *insert R c d*, *auto simp*: *field-simps*)
qed

lemma *small-trans*:
f ∈ *l F* (*g*) ⟹ *g* ∈ *l F* (*h*) ⟹ *f* ∈ *l F* (*h*)
by (*rule big-small-trans*[*OF small-imp-big*])

lemma *small-big-trans'*:
f ∈ *l F* (*g*) ⟹ *g* ∈ *L F* (*h*) ⟹ *f* ∈ *L F* (*h*)
by (*rule small-imp-big*[*OF small-big-trans*])

lemma *big-small-trans'*:
f ∈ *L F* (*g*) ⟹ *g* ∈ *l F* (*h*) ⟹ *f* ∈ *L F* (*h*)
by (*rule small-imp-big*[*OF big-small-trans*])

lemma *big-subsetI* [*intro*]: *f* ∈ *L F* (*g*) ⟹ *L F* (*f*) ⊆ *L F* (*g*)
by (*intro subsetI*) (*drule* (*1*) *big-trans*)

lemma *small-subsetI* [*intro*]: *f* ∈ *L F* (*g*) ⟹ *l F* (*f*) ⊆ *l F* (*g*)
by (*intro subsetI*) (*drule* (*1*) *small-big-trans*)

lemma *big-refl* [*simp*]: *f* ∈ *L F* (*f*)
by (*rule bigI*[*of 1*]) *simp-all*

lemma *small-refl-iff*: *f* ∈ *l F* (*f*) ⟷ *eventually* (λ*x*. *f x* = *0*) *F*
proof (*rule iffI*[*OF - smallI*])
assume *f*: *f* ∈ *l F f*
have (*1*/*2*::*real*) > *0* (*2*::*real*) > *0* **by** *simp-all*
from *smallD*[*OF f this*(*1*)] *smallD*[*OF f this*(*2*)]
show *eventually* (λ*x*. *f x* = *0*) *F* **by** *eventually-elim* (*insert R*, *auto*)
next
fix *c* :: *real* **assume** *c* > *0 eventually* (λ*x*. *f x* = *0*) *F*
from *this*(*2*) **show** *eventually* (λ*x*. *R* (*norm* (*f x*)) (*c* ∗ *norm* (*f x*))) *F*
by *eventually-elim simp-all*
qed

lemma *big-small-asymmetric*: *f* ∈ *L F* (*g*) ⟹ *g* ∈ *l F* (*f*) ⟹ *eventually* (λ*x*. *f x* = *0*) *F*
by (*drule* (*1*) *big-small-trans*) (*simp add*: *small-refl-iff*)

lemma *small-big-asymmetric*: *f* ∈ *l F* (*g*) ⟹ *g* ∈ *L F* (*f*) ⟹ *eventually* (λ*x*. *f x* = *0*) *F*
by (*drule* (*1*) *small-big-trans*) (*simp add*: *small-refl-iff*)

lemma *small-asymmetric*: *f* ∈ *l F* (*g*) ⟹ *g* ∈ *l F* (*f*) ⟹ *eventually* (λ*x*. *f x* = *0*) *F*
by (*drule* (*1*) *small-trans*) (*simp add*: *small-refl-iff*)

lemma *plus-aux*:
assumes $f \in o[F](g)$
shows $g \in L\ F\ (\lambda x.\ f\ x + g\ x)$
proof (*rule R-E*)
assume *R*: $R = (\leq)$
have *A*: $1/2 > (0::real)$ **by** *simp*
have *B*: $1/2 * (norm\ (g\ x)) \leq norm\ (f\ x + g\ x)$
if $norm\ (f\ x) \leq 1/2 * norm\ (g\ x)$ **for** *x*
proof −
from *that* **have** $1/2 * (norm\ (g\ x)) \leq (norm\ (g\ x)) - (norm\ (f\ x))$
by *simp*
also have $norm\ (g\ x) - norm\ (f\ x) \leq norm\ (f\ x + g\ x)$
by (*subst add.commute*) (*rule norm-diff-ineq*)
finally show *?thesis* **by** *simp*
qed
show $g \in L\ F\ (\lambda x.\ f\ x + g\ x)$
apply (*rule bigI*[*of 2*])
apply *simp*
apply (*use landau-o.smallD*[*OF assms A*] **in** *eventually-elim*)
apply (*use B* **in** ‹*simp add*: *R algebra-simps*›)
done
next
assume *R*: $R = (\lambda x\ y.\ x \geq y)$
show $g \in L\ F\ (\lambda x.\ f\ x + g\ x)$
proof (*rule bigI*[*of 1/2*])
show *eventually* $(\lambda x.\ R\ (norm\ (g\ x))\ (1/2 * norm\ (f\ x + g\ x)))\ F$
using *landau-o.smallD*[*OF assms zero-less-one*]
proof *eventually-elim*
case (*elim x*)
have $norm\ (f\ x + g\ x) \leq norm\ (f\ x) + norm\ (g\ x)$
by (*rule norm-triangle-ineq*)
also note *elim*
finally show *?case* **by** (*simp add*: *R*)
qed
qed *simp-all*
qed

end

lemma *summable-comparison-test-bigo*:
fixes $f :: nat \Rightarrow real$
assumes *summable* $(\lambda n.\ norm\ (g\ n))$ $f \in O(g)$
shows *summable f*
proof −
from ‹$f \in O(g)$› **obtain** *C* **where** *C*: *eventually* $(\lambda x.\ norm\ (f\ x) \leq C * norm\ (g\ x))$ *at-top*
by (*auto elim*: *landau-o.bigE*)
thus *?thesis*

by (*rule summable-comparison-test-ev*) (*insert assms*, *auto intro*: *summable-mult*)
qed

lemma *bigomega-iff-bigo*: $g \in \Omega[F](f) \longleftrightarrow f \in O[F](g)$
proof
assume $f \in O[F](g)$
then obtain c **where** $0 < c \; \forall_F \; x \; in \; F. \; norm \; (f \; x) \leq c * norm \; (g \; x)$
by (*rule landau-o.bigE*)
then show $g \in \Omega[F](f)$
by (*intro landau-omega.bigI*[*of inverse c*]) (*simp-all add*: *field-simps*)
next
assume $g \in \Omega[F](f)$
then obtain c **where** $0 < c \; \forall_F \; x \; in \; F. \; c * norm \; (f \; x) \leq norm \; (g \; x)$
by (*rule landau-omega.bigE*)
then show $f \in O[F](g)$
by (*intro landau-o.bigI*[*of inverse c*]) (*simp-all add*: *field-simps*)
qed

lemma *smallomega-iff-smallo*: $g \in \omega[F](f) \longleftrightarrow f \in o[F](g)$
proof
assume $f \in o[F](g)$
from *landau-o.smallD*[*OF this*, *of inverse c* **for** *c*]
show $g \in \omega[F](f)$ **by** (*intro landau-omega.smallI*) (*simp-all add*: *field-simps*)
next
assume $g \in \omega[F](f)$
from *landau-omega.smallD*[*OF this*, *of inverse c* **for** *c*]
show $f \in o[F](g)$ **by** (*intro landau-o.smallI*) (*simp-all add*: *field-simps*)
qed

context *landau-pair*
begin

lemma *big-mono*:
eventually ($\lambda x. \; R \; (norm \; (f \; x)) \; (norm \; (g \; x))$) $F \Longrightarrow f \in L \; F \; (g)$
by (*rule bigI*[*OF zero-less-one*]) *simp*

lemma *big-mult*:
assumes $f1 \in L \; F \; (g1) \; f2 \in L \; F \; (g2)$
shows $(\lambda x. \; f1 \; x * f2 \; x) \in L \; F \; (\lambda x. \; g1 \; x * g2 \; x)$
proof−
from *assms* **obtain** $c1 \; c2$ **where** $*$: $c1 > 0 \; c2 > 0$
and $**$: $\forall_F \; x \; in \; F. \; R \; (norm \; (f1 \; x)) \; (c1 * norm \; (g1 \; x))$
$\forall_F \; x \; in \; F. \; R \; (norm \; (f2 \; x)) \; (c2 * norm \; (g2 \; x))$
by (*elim bigE*)
from $*$ **have** $c1 * c2 > 0$ **by** *simp*
moreover have *eventually* ($\lambda x. \; R \; (norm \; (f1 \; x * f2 \; x)) \; (c1 * c2 * norm \; (g1 \; x * g2 \; x))) \; F$
using $**$

```
  proof eventually-elim
    case (elim x)
    show ?case
    proof (cases rule: R-E)
      case le
      have norm (f1 x) * norm (f2 x) ≤ (c1 * norm (g1 x)) * (c2 * norm (g2 x))
        using elim le * by (intro mult-mono mult-nonneg-nonneg) auto
      with le show ?thesis by (simp add: le norm-mult mult-ac)
    next
      case ge
      have (c1 * norm (g1 x)) * (c2 * norm (g2 x)) ≤ norm (f1 x) * norm (f2 x)
        using elim ge * by (intro mult-mono mult-nonneg-nonneg) auto
      with ge show ?thesis by (simp-all add: norm-mult mult-ac)
    qed
  qed
  ultimately show ?thesis by (rule bigI)
qed

lemma small-big-mult:
  assumes f1 ∈ l F (g1) f2 ∈ L F (g2)
  shows   (λx. f1 x * f2 x) ∈ l F (λx. g1 x * g2 x)
proof (rule smallI)
  fix c1 :: real assume c1: c1 > 0
  from assms(2) obtain c2 where c2: c2 > 0
    and *: ∀ F x in F. R (norm (f2 x)) (c2 * norm (g2 x)) by (elim bigE)
  from assms(1) c1 c2 have eventually (λx. R (norm (f1 x)) (c1 * inverse c2 *
norm (g1 x))) F
    by (auto intro!: smallD)
  with * show eventually (λx. R (norm (f1 x * f2 x)) (c1 * norm (g1 x * g2 x)))
F
  proof eventually-elim
    case (elim x)
    show ?case
    proof (cases rule: R-E)
      case le
      have norm (f1 x) * norm (f2 x) ≤ (c1 * inverse c2 * norm (g1 x)) * (c2 *
norm (g2 x))
        using elim le c1 c2 by (intro mult-mono mult-nonneg-nonneg) auto
      with le c2 show ?thesis by (simp add: le norm-mult field-simps)
    next
      case ge
      have norm (f1 x) * norm (f2 x) ≥ (c1 * inverse c2 * norm (g1 x)) * (c2 *
norm (g2 x))
        using elim ge c1 c2 by (intro mult-mono mult-nonneg-nonneg) auto
      with ge c2 show ?thesis by (simp add: ge norm-mult field-simps)
    qed
  qed
qed
```

lemma *big-small-mult*:
f1 ∈ *L F* (*g1*) ⟹ *f2* ∈ *l F* (*g2*) ⟹ (λ*x*. *f1 x* ∗ *f2 x*) ∈ *l F* (λ*x*. *g1 x* ∗ *g2 x*)
by (*subst* (*1 2*) *mult.commute*) (*rule small-big-mult*)

lemma *small-mult*: *f1* ∈ *l F* (*g1*) ⟹ *f2* ∈ *l F* (*g2*) ⟹ (λ*x*. *f1 x* ∗ *f2 x*) ∈ *l F* (λ*x*. *g1 x* ∗ *g2 x*)
by (*rule small-big-mult*, *assumption*, *rule small-imp-big*)

lemmas *mult* = *big-mult small-big-mult big-small-mult small-mult*

lemma *big-power*:
assumes *f* ∈ *L F* (*g*)
shows (λ*x*. *f x* ^ *m*) ∈ *L F* (λ*x*. *g x* ^ *m*)
using *assms* **by** (*induction m*) (*auto intro*: *mult*)

lemma (**in** *landau-pair*) *small-power*:
assumes *f* ∈ *l F* (*g*) *m* > *0*
shows (λ*x*. *f x* ^ *m*) ∈ *l F* (λ*x*. *g x* ^ *m*)
proof −
have (λ*x*. *f x* ∗ *f x* ^ (*m* − *1*)) ∈ *l F* (λ*x*. *g x* ∗ *g x* ^ (*m* − *1*))
by (*intro small-big-mult assms big-power*[*OF small-imp-big*])
thus *?thesis*
using *assms* **by** (*cases m*) (*simp-all add*: *mult-ac*)
qed

lemma *big-power-increasing*:
assumes (λ-. *1*) ∈ *L F f m* ≤ *n*
shows (λ*x*. *f x* ^ *m*) ∈ *L F* (λ*x*. *f x* ^ *n*)
proof −
have (λ*x*. *f x* ^ *m* ∗ *1* ^ (*n* − *m*)) ∈ *L F* (λ*x*. *f x* ^ *m* ∗ *f x* ^ (*n* − *m*))
using *assms* **by** (*intro mult big-power*) *auto*
also have (λ*x*. *f x* ^ *m* ∗ *f x* ^ (*n* − *m*)) = (λ*x*. *f x* ^ (*m* + (*n* − *m*)))
by (*subst power-add* [*symmetric*]) (*rule refl*)
also have *m* + (*n* − *m*) = *n*
using *assms* **by** *simp*
finally show *?thesis* **by** *simp*
qed

lemma *small-power-increasing*:
assumes (λ-. *1*) ∈ *l F f m* < *n*
shows (λ*x*. *f x* ^ *m*) ∈ *l F* (λ*x*. *f x* ^ *n*)
proof −
note [*trans*] = *small-big-trans*
have (λ*x*. *f x* ^ *m* ∗ *1*) ∈ *l F* (λ*x*. *f x* ^ *m* ∗ *f x*)
using *assms* **by** (*intro big-small-mult*) *auto*
also have (λ*x*. *f x* ^ *m* ∗ *f x*) = (λ*x*. *f x* ^ *Suc m*)
by (*simp add*: *mult-ac*)
also have ... ∈ *L F* (λ*x*. *f x* ^ *n*)
using *assms* **by** (*intro big-power-increasing*[*OF small-imp-big*]) *auto*

finally show *?thesis* **by** *simp*
qed

sublocale *big*: *landau-symbol L L' Lr*
proof
have *L*: $L = bigo \vee L = bigomega$
by (*rule R-E*) (*auto simp*: *bigo-def L-def bigomega-def fun-eq-iff*)
have *A*: $(\lambda x.\ c * f\ x) \in L\ F\ f$ **if** $c \neq 0$ **for** $c :: 'b$ **and** F **and** $f :: 'a \Rightarrow 'b$
using *that* **by** (*intro bigI*[*of norm c*]) (*simp-all add*: *norm-mult*)
show $L\ F\ (\lambda x.\ c * f\ x) = L\ F\ f$ **if** $c \neq 0$ **for** $c :: 'b$ **and** F **and** $f :: 'a \Rightarrow 'b$
using ‹$c \neq 0$› **and** *A*[*of c f*] **and** *A*[*of inverse c* $\lambda x.\ c * f\ x$]
by (*intro equalityI big-subsetI*) (*simp-all add*: *field-simps*)
show $((\lambda x.\ c * f\ x) \in L\ F\ g) = (f \in L\ F\ g)$ **if** $c \neq 0$
for $c :: 'b$ **and** F **and** $f\ g :: 'a \Rightarrow 'b$
proof −
from ‹$c \neq 0$› **and** *A*[*of c f*] **and** *A*[*of inverse c* $\lambda x.\ c * f\ x$]
have $(\lambda x.\ c * f\ x) \in L\ F\ f\ f \in L\ F\ (\lambda x.\ c * f\ x)$
by (*simp-all add*: *field-simps*)
then show *?thesis* **by** (*intro iffI*) (*erule* (*1*) *big-trans*)+
qed
show $(\lambda x.\ inverse\ (g\ x)) \in L\ F\ (\lambda x.\ inverse\ (f\ x))$
if ∗: $f \in L\ F\ (g)$ **and** ∗∗: *eventually* $(\lambda x.\ f\ x \neq 0)\ F$ *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
for $f\ g :: 'a \Rightarrow 'b$ **and** F
proof −
from ∗ **obtain** c **where** *c*: $c > 0$ **and** ∗∗∗: $\forall_F\ x\ in\ F.\ R\ (norm\ (f\ x))\ (c * norm\ (g\ x))$
by (*elim bigE*)
from ∗∗ ∗∗∗ **have** *eventually* $(\lambda x.\ R\ (norm\ (inverse\ (g\ x)))\ (c * norm\ (inverse\ (f\ x))))\ F$
by *eventually-elim* (*rule R-E*, *simp-all add*: *field-simps norm-inverse norm-divide c*)
with c **show** *?thesis* **by** (*rule bigI*)
qed
show $L\ F\ g \subseteq L\ F\ (\lambda x.\ f\ x + g\ x)$ **if** $f \in o[F](g)$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** F
using *plus-aux that* **by** (*blast intro*!: *big-subsetI*)
show $L\ F\ (f) = L\ F\ (g)$ **if** *eventually* $(\lambda x.\ f\ x = g\ x)\ F$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** F
unfolding *L-def* **by** (*subst eventually-subst'*[*OF that*]) (*rule refl*)
show $f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$ **if** *eventually* $(\lambda x.\ f\ x = g\ x)\ F$
for $f\ g\ h :: 'a \Rightarrow 'b$ **and** F
unfolding *L-def mem-Collect-eq*
by (*subst* (*1*) *eventually-subst'*[*OF that*]) (*rule refl*)
show $L\ F\ f \subseteq L\ F\ g$ **if** $f \in L\ F\ g$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** F
using *that* **by** (*rule big-subsetI*)
show $L\ F\ (f) = L\ F\ (g)$ **if** $f \in \Theta[F](g)$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** F
using *L that* **unfolding** *bigtheta-def*
by (*intro equalityI big-subsetI*) (*auto simp*: *bigomega-iff-bigo*)
show $f \in L\ F\ (h) \longleftrightarrow g \in L\ F\ (h)$ **if** $f \in \Theta[F](g)$ **for** $f\ g\ h :: 'a \Rightarrow 'b$ **and** F
by (*rule disjE*[*OF L*])

```
    (use that in ‹auto simp: bigtheta-def bigomega-iff-bigo intro: landau-o.big-trans›)
  show (λx. h x * f x) ∈ L F (λx. h x * g x) if f ∈ L F g for f g h :: 'a ⇒ 'b and
F
    using that by (intro big-mult) simp
  show f ∈ L F (h) if f ∈ L F g g ∈ L F h for f g h :: 'a ⇒ 'b and F
    using that by (rule big-trans)
  show (λx. f (h x)) ∈ L' G (λx. g (h x))
    if f ∈ L F g and filterlim h F G
    for f g :: 'a ⇒ 'b and h :: 'c ⇒ 'a and F G
    using that by (auto simp: L-def L'-def filterlim-iff)
  show f ∈ L (sup F G) g if f ∈ L F g f ∈ L G g
    for f g :: 'a ⇒ 'b and F G :: 'a filter
  proof −
    from that [THEN bigE] obtain c1 c2
      where *: c1 > 0 c2 > 0
        and **: ∀F x in F. R (norm (f x)) (c1 * norm (g x))
                ∀F x in G. R (norm (f x)) (c2 * norm (g x)) .
    define c where c = (if R c1 c2 then c2 else c1)
    from * have c: R c1 c R c2 c c > 0
      by (auto simp: c-def dest: R-linear)
    with ** have eventually (λx. R (norm (f x)) (c * norm (g x))) F
                 eventually (λx. R (norm (f x)) (c * norm (g x))) G
      by (force elim: eventually-mono intro: R-trans[OF - R-mult-right-mono])+
    with c show f ∈ L (sup F G) g
      by (auto simp: L-def eventually-sup)
  qed
  show ((λx. f (h x)) ∈ L' (filtercomap h F) (λx. g (h x))) if (f ∈ L F g)
    for f g :: 'a ⇒ 'b and h :: 'c ⇒ 'a and F G :: 'a filter
    using that unfolding L-def L'-def by auto
qed (auto simp: L-def Lr-def eventually-filtermap L'-def
        intro: filter-leD exI[of - 1::real])

sublocale small: landau-symbol l l' lr
proof
  have A: (λx. c * f x) ∈ L F f if c ≠ 0 for c :: 'b and f :: 'a ⇒ 'b and F
    using that by (intro bigI[of norm c]) (simp-all add: norm-mult)
  show l F (λx. c * f x) = l F f if c ≠ 0 for c :: 'b and f :: 'a ⇒ 'b and F
    using that and A[of c f] and A[of inverse c λx. c * f x]
    by (intro equalityI small-subsetI) (simp-all add: field-simps)
  show ((λx. c * f x) ∈ l F g) = (f ∈ l F g) if c ≠ 0 for c :: 'b and f g :: 'a ⇒ 'b
and F
  proof −
    from that and A[of c f] and A[of inverse c λx. c * f x]
    have (λx. c * f x) ∈ L F f f ∈ L F (λx. c * f x)
      by (simp-all add: field-simps)
    then show ?thesis
      by (intro iffI) (erule (1) big-small-trans)+
  qed
  show l F g ⊆ l F (λx. f x + g x) if f ∈ o[F](g) for f g :: 'a ⇒ 'b and F
```

using *plus-aux that* **by** (*blast intro*!: *small-subsetI*)
show $(\lambda x.\ inverse\ (g\ x)) \in l\ F\ (\lambda x.\ inverse\ (f\ x))$
if *A*: $f \in l\ F\ (g)$ **and** *B*: *eventually* $(\lambda x.\ f\ x \neq 0)\ F$ *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
for $f\ g :: 'a \Rightarrow 'b$ **and** *F*
proof (*rule smallI*)
fix *c* :: *real* **assume** *c*: $c > 0$
from *B smallD*[*OF A c*]
show *eventually* $(\lambda x.\ R\ (norm\ (inverse\ (g\ x)))\ (c * norm\ (inverse\ (f\ x))))\ F$
by *eventually-elim* (*rule R-E*, *simp-all add*: *field-simps norm-inverse norm-divide*)
qed
show $l\ F\ (f) = l\ F\ (g)$ **if** *eventually* $(\lambda x.\ f\ x = g\ x)\ F$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** *F*
unfolding *l-def* **by** (*subst eventually-subst'*[*OF that*]) (*rule refl*)
show $f \in l\ F\ (h) \longleftrightarrow g \in l\ F\ (h)$ **if** *eventually* $(\lambda x.\ f\ x = g\ x)\ F$ **for** $f\ g\ h :: 'a \Rightarrow 'b$ **and** *F*
unfolding *l-def mem-Collect-eq* **by** (*subst* (*1*) *eventually-subst'*[*OF that*]) (*rule refl*)
show $l\ F\ f \subseteq l\ F\ g$ **if** $f \in l\ F\ g$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** *F*
using *that* **by** (*intro small-subsetI small-imp-big*)
show $l\ F\ (f) = l\ F\ (g)$ **if** $f \in \Theta[F](g)$ **for** $f\ g :: 'a \Rightarrow 'b$ **and** *F*
proof −
have *L*: $L = bigo \vee L = bigomega$
by (*rule R-E*) (*auto simp*: *bigo-def L-def bigomega-def fun-eq-iff*)
with *that* **show** *?thesis* **unfolding** *bigtheta-def*
by (*intro equalityI small-subsetI*) (*auto simp*: *bigomega-iff-bigo*)
qed
show $f \in l\ F\ (h) \longleftrightarrow g \in l\ F\ (h)$ **if** $f \in \Theta[F](g)$ **for** $f\ g\ h :: 'a \Rightarrow 'b$ **and** *F*
proof −
have *l*: $l = smallo \vee l = smallomega$
by (*rule R-E*) (*auto simp*: *smallo-def l-def smallomega-def fun-eq-iff*)
show *?thesis*
by (*rule disjE*[*OF l*])
(*use that* **in** ‹*auto simp*: *bigtheta-def bigomega-iff-bigo smallomega-iff-smallo intro*: *landau-o.big-small-trans landau-o.small-big-trans*›)
qed
show $(\lambda x.\ h\ x * f\ x) \in l\ F\ (\lambda x.\ h\ x * g\ x)$ **if** $f \in l\ F\ g$ **for** $f\ g\ h :: 'a \Rightarrow 'b$ **and** *F*
using *that* **by** (*intro big-small-mult*) *simp*
show $f \in l\ F\ (h)$ **if** $f \in l\ F\ g\ g \in l\ F\ h$ **for** $f\ g\ h :: 'a \Rightarrow 'b$ **and** *F*
using *that* **by** (*rule small-trans*)
show $(\lambda x.\ f\ (h\ x)) \in l'\ G\ (\lambda x.\ g\ (h\ x))$ **if** $f \in l\ F\ g$ **and** *filterlim h F G*
for $f\ g :: 'a \Rightarrow 'b$ **and** $h :: 'c \Rightarrow 'a$ **and** *F G*
using *that* **by** (*auto simp*: *l-def l'-def filterlim-iff*)
show $((\lambda x.\ f\ (h\ x)) \in l'\ (filtercomap\ h\ F)\ (\lambda x.\ g\ (h\ x)))$ **if** $f \in l\ F\ g$
for $f\ g :: 'a \Rightarrow 'b$ **and** $h :: 'c \Rightarrow 'a$ **and** $F\ G :: 'a\ filter$
using *that* **unfolding** *l-def l'-def* **by** *auto*
qed (*auto simp*: *l-def lr-def eventually-filtermap l'-def eventually-sup intro*: *filter-leD*)

These rules allow chaining of Landau symbol propositions in Isar with "also".

lemma *big-mult-1*: $f \in L\ F\ (g) \Longrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (h) \Longrightarrow f \in L\ F\ (\lambda x.\ g\ x * h\ x)$
and *big-mult-1'*: $(\lambda\text{-}.\ 1) \in L\ F\ (g) \Longrightarrow f \in L\ F\ (h) \Longrightarrow f \in L\ F\ (\lambda x.\ g\ x * h\ x)$
and *small-mult-1*: $f \in l\ F\ (g) \Longrightarrow (\lambda\text{-}.\ 1) \in L\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x.\ g\ x * h\ x)$
and *small-mult-1'*: $(\lambda\text{-}.\ 1) \in L\ F\ (g) \Longrightarrow f \in l\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x.\ g\ x * h\ x)$
and *small-mult-1''*: $f \in L\ F\ (g) \Longrightarrow (\lambda\text{-}.\ 1) \in l\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x.\ g\ x * h\ x)$
and *small-mult-1'''*: $(\lambda\text{-}.\ 1) \in l\ F\ (g) \Longrightarrow f \in L\ F\ (h) \Longrightarrow f \in l\ F\ (\lambda x.\ g\ x * h\ x)$
by (*drule* (*1*) *big.mult big-small-mult small-big-mult*, *simp*)+

lemma *big-1-mult*: $f \in L\ F\ (g) \Longrightarrow h \in L\ F\ (\lambda\text{-}.\ 1) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in L\ F\ (g)$
and *big-1-mult'*: $h \in L\ F\ (\lambda\text{-}.\ 1) \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in L\ F\ (g)$
and *small-1-mult*: $f \in l\ F\ (g) \Longrightarrow h \in L\ F\ (\lambda\text{-}.\ 1) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult'*: $h \in L\ F\ (\lambda\text{-}.\ 1) \Longrightarrow f \in l\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult''*: $f \in L\ F\ (g) \Longrightarrow h \in l\ F\ (\lambda\text{-}.\ 1) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in l\ F\ (g)$
and *small-1-mult'''*: $h \in l\ F\ (\lambda\text{-}.\ 1) \Longrightarrow f \in L\ F\ (g) \Longrightarrow (\lambda x.\ f\ x * h\ x) \in l\ F\ (g)$
by (*drule* (*1*) *big.mult big-small-mult small-big-mult*, *simp*)+

lemmas *mult-1-trans* =
big-mult-1 big-mult-1' small-mult-1 small-mult-1' small-mult-1'' small-mult-1'''
big-1-mult big-1-mult' small-1-mult small-1-mult' small-1-mult'' small-1-mult'''

lemma *big-equal-iff-bigtheta*: $L\ F\ (f) = L\ F\ (g) \longleftrightarrow f \in \Theta[F](g)$
proof
have *L*: $L = bigo \vee L = bigomega$
by (*rule R-E*) (*auto simp*: *fun-eq-iff L-def bigo-def bigomega-def*)
fix $f\ g :: 'a \Rightarrow 'b$ **assume** $L\ F\ (f) = L\ F\ (g)$
with *big-refl*[*of f F*] *big-refl*[*of g F*] **have** $f \in L\ F\ (g)\ g \in L\ F\ (f)$ **by** *simp-all*
thus $f \in \Theta[F](g)$ **using** *L* **unfolding** *bigtheta-def* **by** (*auto simp*: *bigomega-iff-bigo*)
qed (*rule big.cong-bigtheta*)

lemma *big-prod*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in L\ F\ (g\ x)$
shows $(\lambda y.\ \prod x{\in}A.\ f\ x\ y) \in L\ F\ (\lambda y.\ \prod x{\in}A.\ g\ x\ y)$
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *big.mult*)

lemma *big-prod-in-1*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in L\ F\ (\lambda\text{-}.\ 1)$
shows $(\lambda y.\ \prod x{\in}A.\ f\ x\ y) \in L\ F\ (\lambda\text{-}.\ 1)$
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*!: *big.mult-in-1*)

end

context *landau-symbol*
begin

lemma *plus-absorb1*:
assumes $f \in o[F](g)$
shows $L\ F\ (\lambda x.\ f\ x + g\ x) = L\ F\ (g)$
proof (*intro equalityI*)
from *plus-subset1* **and** *assms* **show** $L\ F\ g \subseteq L\ F\ (\lambda x.\ f\ x + g\ x)$ **.**
from *landau-o.small.plus-subset1*[*OF assms*] **and** *assms* **have** $(\lambda x.\ -f\ x) \in o[F](\lambda x.\ f\ x + g\ x)$
by (*auto simp*: *landau-o.small.uminus-in-iff*)
from *plus-subset1*[*OF this*] **show** $L\ F\ (\lambda x.\ f\ x + g\ x) \subseteq L\ F\ (g)$ **by** *simp*
qed

lemma *plus-absorb2*: $g \in o[F](f) \Longrightarrow L\ F\ (\lambda x.\ f\ x + g\ x) = L\ F\ (f)$
using *plus-absorb1*[*of g F f*] **by** (*simp add*: *add.commute*)

lemma *diff-absorb1*: $f \in o[F](g) \Longrightarrow L\ F\ (\lambda x.\ f\ x - g\ x) = L\ F\ (g)$
by (*simp only*: *diff-conv-add-uminus plus-absorb1 landau-o.small.uminus uminus*)

lemma *diff-absorb2*: $g \in o[F](f) \Longrightarrow L\ F\ (\lambda x.\ f\ x - g\ x) = L\ F\ (f)$
by (*simp only*: *diff-conv-add-uminus plus-absorb2 landau-o.small.uminus-in-iff*)

lemmas *absorb* = *plus-absorb1 plus-absorb2 diff-absorb1 diff-absorb2*

end

lemma *bigthetaI* [*intro*]: $f \in O[F](g) \Longrightarrow f \in \Omega[F](g) \Longrightarrow f \in \Theta[F](g)$
unfolding *bigtheta-def bigomega-def* **by** *blast*

lemma *bigthetaD1* [*dest*]: $f \in \Theta[F](g) \Longrightarrow f \in O[F](g)$
and *bigthetaD2* [*dest*]: $f \in \Theta[F](g) \Longrightarrow f \in \Omega[F](g)$
unfolding *bigtheta-def bigo-def bigomega-def* **by** *blast+*

lemma *bigtheta-refl* [*simp*]: $f \in \Theta[F](f)$
unfolding *bigtheta-def* **by** *simp*

lemma *bigtheta-sym*: $f \in \Theta[F](g) \longleftrightarrow g \in \Theta[F](f)$
unfolding *bigtheta-def* **by** (*auto simp*: *bigomega-iff-bigo*)

lemmas *landau-flip* =
bigomega-iff-bigo[*symmetric*] *smallomega-iff-smallo*[*symmetric*]
bigomega-iff-bigo smallomega-iff-smallo bigtheta-sym

interpretation *landau-theta*: *landau-symbol bigtheta bigtheta bigtheta*

proof
fix $f\ g :: 'a \Rightarrow 'b$ **and** F
assume $f \in o[F](g)$
hence $O[F](g) \subseteq O[F](\lambda x.\ f\ x + g\ x)\ \Omega[F](g) \subseteq \Omega[F](\lambda x.\ f\ x + g\ x)$
by (*rule landau-o.big.plus-subset1 landau-omega.big.plus-subset1*)+
thus $\Theta[F](g) \subseteq \Theta[F](\lambda x.\ f\ x + g\ x)$ **unfolding** *bigtheta-def* **by** *blast*
next
fix $f\ g :: 'a \Rightarrow 'b$ **and** F
assume $f \in \Theta[F](g)$
thus *A*: $\Theta[F](f) = \Theta[F](g)$
apply (*subst* (*1 2*) *bigtheta-def*)
apply (*subst landau-o.big.cong-bigtheta landau-omega.big.cong-bigtheta*, *assumption*)+
apply (*rule refl*)
done
thus $\Theta[F](f) \subseteq \Theta[F](g)$ **by** *simp*
fix $h :: 'a \Rightarrow 'b$
show $f \in \Theta[F](h) \longleftrightarrow g \in \Theta[F](h)$ **by** (*subst* (*1 2*) *bigtheta-sym*) (*simp add*: *A*)
next
fix $f\ g\ h :: 'a \Rightarrow 'b$ **and** F
assume $f \in \Theta[F](g)\ g \in \Theta[F](h)$
thus $f \in \Theta[F](h)$ **unfolding** *bigtheta-def*
by (*blast intro*: *landau-o.big.trans landau-omega.big.trans*)
next
fix $f :: 'a \Rightarrow 'b$ **and** *F1 F2* :: *'a filter*
assume *F1* $\leq$ *F2*
thus $\Theta[F2](f) \subseteq \Theta[F1](f)$
by (*auto simp*: *bigtheta-def intro*: *landau-o.big.filter-mono landau-omega.big.filter-mono*)
qed (*auto simp*: *bigtheta-def landau-o.big.norm-iff*
landau-o.big.cmult landau-omega.big.cmult
landau-o.big.cmult-in-iff landau-omega.big.cmult-in-iff
landau-o.big.in-cong landau-omega.big.in-cong
landau-o.big.mult landau-omega.big.mult
landau-o.big.inverse landau-omega.big.inverse
landau-o.big.compose landau-omega.big.compose
landau-o.big.bot' landau-omega.big.bot'
landau-o.big.in-filtermap-iff landau-omega.big.in-filtermap-iff
landau-o.big.sup landau-omega.big.sup
landau-o.big.filtercomap landau-omega.big.filtercomap
dest: *landau-o.big.cong landau-omega.big.cong*)

lemmas *landau-symbols* =
landau-o.big.landau-symbol-axioms landau-o.small.landau-symbol-axioms
landau-omega.big.landau-symbol-axioms landau-omega.small.landau-symbol-axioms
landau-theta.landau-symbol-axioms

lemma *bigoI* [*intro*]:
assumes *eventually* $(\lambda x.\ (norm\ (f\ x)) \leq c * (norm\ (g\ x)))\ F$
shows $f \in O[F](g)$

proof (*rule landau-o.bigI*)
show *max 1 c > 0* **by** *simp*
have *c * (norm (g x)) ≤ max 1 c * (norm (g x))* **for** *x*
by (*simp add*: *mult-right-mono*)
with *assms* **show** *eventually* (λ*x*. (*norm* (*f x*)) ≤ *max 1 c * (norm* (*g x*))) *F*
by (*auto elim*!: *eventually-mono dest*: *order.trans*)
qed

lemma *smallomegaD* [*dest*]:
assumes *f* ∈ ω[*F*](*g*)
shows *eventually* (λ*x*. (*norm* (*f x*)) ≥ *c * (norm* (*g x*))) *F*
proof (*cases c > 0*)
case *False*
show *?thesis*
by (*intro always-eventually allI*, *rule order.trans*[*of - 0*])
(*insert False*, *auto intro*!: *mult-nonpos-nonneg*)
qed (*blast dest*: *landau-omega.smallD*[*OF assms*, *of c*])

lemma *bigthetaI'*:
assumes *c1 > 0 c2 > 0*
assumes *eventually* (λ*x*. *c1 * (norm* (*g x*)) ≤ (*norm* (*f x*)) ∧ (*norm* (*f x*)) ≤ *c2 * (norm* (*g x*))) *F*
shows *f* ∈ Θ[*F*](*g*)
apply (*rule bigthetaI*)
apply (*rule landau-o.bigI*[*OF assms*(*2*)]) **using** *assms*(*3*) **apply** (*eventually-elim*, *simp*)
apply (*rule landau-omega.bigI*[*OF assms*(*1*)]) **using** *assms*(*3*) **apply** (*eventually-elim*, *simp*)
done

lemma *bigthetaI-cong*: *eventually* (λ*x*. *f x* = *g x*) *F* ⟹ *f* ∈ Θ[*F*](*g*)
by (*intro bigthetaI'*[*of 1 1*]) (*auto elim*!: *eventually-mono*)

lemma (**in** *landau-symbol*) *ev-eq-trans1*:
f ∈ *L F* (λ*x*. *g x* (*h x*)) ⟹ *eventually* (λ*x*. *h x* = *h'* *x*) *F* ⟹ *f* ∈ *L F* (λ*x*. *g x* (*h'* *x*))
by (*rule bigtheta-trans1*[*OF - bigthetaI-cong*]) (*auto elim*!: *eventually-mono*)

lemma (**in** *landau-symbol*) *ev-eq-trans2*:
eventually (λ*x*. *f x* = *f'* *x*) *F* ⟹ (λ*x*. *g x* (*f'* *x*)) ∈ *L F* (*h*) ⟹ (λ*x*. *g x* (*f x*)) ∈ *L F* (*h*)
by (*rule bigtheta-trans2*[*OF bigthetaI-cong*]) (*auto elim*!: *eventually-mono*)

declare *landau-o.smallI landau-omega.bigI landau-omega.smallI* [*intro*]
declare *landau-o.bigE landau-omega.bigE* [*elim*]
declare *landau-o.smallD*

lemma (**in** *landau-symbol*) *bigtheta-trans1'*:

$f \in L\ F\ (g) \Longrightarrow h \in \Theta[F](g) \Longrightarrow f \in L\ F\ (h)$
by (*subst cong-bigtheta*[*symmetric*]) (*simp add*: *bigtheta-sym*)

lemma (**in** *landau-symbol*) *bigtheta-trans2'*:
$g \in \Theta[F](f) \Longrightarrow g \in L\ F\ (h) \Longrightarrow f \in L\ F\ (h)$
by (*rule bigtheta-trans2*, *subst bigtheta-sym*)

lemma *bigo-bigomega-trans*: $f \in O[F](g) \Longrightarrow h \in \Omega[F](g) \Longrightarrow f \in O[F](h)$
and *bigo-smallomega-trans*: $f \in O[F](g) \Longrightarrow h \in \omega[F](g) \Longrightarrow f \in o[F](h)$
and *smallo-bigomega-trans*: $f \in o[F](g) \Longrightarrow h \in \Omega[F](g) \Longrightarrow f \in o[F](h)$
and *smallo-smallomega-trans*: $f \in o[F](g) \Longrightarrow h \in \omega[F](g) \Longrightarrow f \in o[F](h)$
and *bigomega-bigo-trans*: $f \in \Omega[F](g) \Longrightarrow h \in O[F](g) \Longrightarrow f \in \Omega[F](h)$
and *bigomega-smallo-trans*: $f \in \Omega[F](g) \Longrightarrow h \in o[F](g) \Longrightarrow f \in \omega[F](h)$
and *smallomega-bigo-trans*: $f \in \omega[F](g) \Longrightarrow h \in O[F](g) \Longrightarrow f \in \omega[F](h)$
and *smallomega-smallo-trans*: $f \in \omega[F](g) \Longrightarrow h \in o[F](g) \Longrightarrow f \in \omega[F](h)$
by (*unfold bigomega-iff-bigo smallomega-iff-smallo*)
(*erule* (*1*) *landau-o.big-trans landau-o.big-small-trans landau-o.small-big-trans*
landau-o.big-trans landau-o.small-trans)+

lemmas *landau-trans-lift* [*trans*] =
landau-symbols[*THEN landau-symbol.lift-trans*]
landau-symbols[*THEN landau-symbol.lift-trans'*]
landau-symbols[*THEN landau-symbol.lift-trans-bigtheta*]
landau-symbols[*THEN landau-symbol.lift-trans-bigtheta'*]

lemmas *landau-mult-1-trans* [*trans*] =
landau-o.mult-1-trans landau-omega.mult-1-trans

lemmas *landau-trans* [*trans*] =
landau-symbols[*THEN landau-symbol.bigtheta-trans1*]
landau-symbols[*THEN landau-symbol.bigtheta-trans2*]
landau-symbols[*THEN landau-symbol.bigtheta-trans1'*]
landau-symbols[*THEN landau-symbol.bigtheta-trans2'*]
landau-symbols[*THEN landau-symbol.ev-eq-trans1*]
landau-symbols[*THEN landau-symbol.ev-eq-trans2*]

landau-o.big-trans landau-o.small-trans landau-o.small-big-trans landau-o.big-small-trans
landau-omega.big-trans landau-omega.small-trans
landau-omega.small-big-trans landau-omega.big-small-trans

bigo-bigomega-trans bigo-smallomega-trans smallo-bigomega-trans smallo-smallomega-trans
bigomega-bigo-trans bigomega-smallo-trans smallomega-bigo-trans smallomega-smallo-trans

lemma *bigtheta-inverse* [*simp*]:
shows $(\lambda x.\ inverse\ (f\ x)) \in \Theta[F](\lambda x.\ inverse\ (g\ x)) \longleftrightarrow f \in \Theta[F](g)$
proof −
have $(\lambda x.\ inverse\ (f\ x)) \in O[F](\lambda x.\ inverse\ (g\ x))$
if *A*: $f \in \Theta[F](g)$
for $f\ g :: 'a \Rightarrow 'b$ **and** *F*

proof −
from *A* **obtain** *c1 c2* :: *real* **where** ∗: *c1* > *0 c2* > *0*
and ∗∗: ∀ *F x in F*. *norm* (*f x*) ≤ *c1* ∗ *norm* (*g x*)
∀ *F x in F*. *c2* ∗ *norm* (*g x*) ≤ *norm* (*f x*)
unfolding *bigtheta-def* **by** (*elim landau-o.bigE landau-omega.bigE IntE*)
from ‹*c2* > *0*› **have** *c2*: *inverse c2* > *0* **by** *simp*
from ∗∗ **have** *eventually* (λ*x*. *norm* (*inverse* (*f x*)) ≤ *inverse c2* ∗ *norm* (*inverse* (*g x*))) *F*
proof *eventually-elim*
fix *x* **assume** *A*: *norm* (*f x*) ≤ *c1* ∗ *norm* (*g x*) *c2* ∗ *norm* (*g x*) ≤ *norm* (*f x*)
from *A* ∗ **have** *f x* = *0* ⟷ *g x* = *0*
by (*auto simp*: *field-simps mult-le-0-iff*)
with *A* ∗ **show** *norm* (*inverse* (*f x*)) ≤ *inverse c2* ∗ *norm* (*inverse* (*g x*))
by (*force simp*: *field-simps norm-inverse norm-divide*)
qed
with *c2* **show** *?thesis* **by** (*rule landau-o.bigI*)
qed
then show *?thesis*
unfolding *bigtheta-def*
by (*force simp*: *bigomega-iff-bigo bigtheta-sym*)
qed

lemma *bigtheta-divide*:
assumes *f1* ∈ Θ(*f2*) *g1* ∈ Θ(*g2*)
shows (λ*x*. *f1 x* / *g1 x*) ∈ Θ(λ*x*. *f2 x* / *g2 x*)
by (*subst* (*1 2*) *divide-inverse*, *intro landau-theta.mult*) (*simp-all add*: *bigtheta-inverse assms*)

lemma *eventually-nonzero-bigtheta*:
assumes *f* ∈ Θ[*F*](*g*)
shows *eventually* (λ*x*. *f x* ≠ *0*) *F* ⟷ *eventually* (λ*x*. *g x* ≠ *0*) *F*
proof −
have *eventually* (λ*x*. *g x* ≠ *0*) *F*
if *A*: *f* ∈ Θ[*F*](*g*) **and** *B*: *eventually* (λ*x*. *f x* ≠ *0*) *F*
for *f g* :: *'a* ⇒ *'b*
proof −
from *A* **obtain** *c1 c2* **where**
∀ *F x in F*. *norm* (*f x*) ≤ *c1* ∗ *norm* (*g x*)
∀ *F x in F*. *c2* ∗ *norm* (*g x*) ≤ *norm* (*f x*)
unfolding *bigtheta-def* **by** (*elim landau-o.bigE landau-omega.bigE IntE*)
with *B* **show** *?thesis* **by** *eventually-elim auto*
qed
with *assms* **show** *?thesis* **by** (*force simp*: *bigtheta-sym*)
qed

55.2 Landau symbols and limits

lemma *bigoI-tendsto-norm*:

fixes *f g*
assumes $((\lambda x.\ norm\ (f\ x\ /\ g\ x))\longrightarrow c)\ F$
assumes *eventually* $(\lambda x.\ g\ x\neq 0)\ F$
shows $f\in O[F](g)$
proof (*rule bigoI*)
from *assms* **have** *eventually* $(\lambda x.\ dist\ (norm\ (f\ x\ /\ g\ x))\ c<1)\ F$
using *tendstoD* **by** *force*
thus *eventually* $(\lambda x.\ (norm\ (f\ x))\leq (norm\ c+1)*(norm\ (g\ x)))\ F$
unfolding *dist-real-def* **using** *assms*(*2*)
proof *eventually-elim*
case (*elim x*)
have $(norm\ (f\ x))-norm\ c*(norm\ (g\ x))\leq norm\ ((norm\ (f\ x))-c*(norm\ (g\ x)))$
unfolding *norm-mult* [*symmetric*] **using** *norm-triangle-ineq2*[*of norm* $(f\ x)$ $c*norm\ (g\ x)$]
by (*simp add*: *norm-mult abs-mult*)
also from *elim* **have** $\ldots = norm\ (norm\ (g\ x))*norm\ (norm\ (f\ x\ /\ g\ x)-c)$
unfolding *norm-mult* [*symmetric*] **by** (*simp add*: *algebra-simps norm-divide*)
also from *elim* **have** $norm\ (norm\ (f\ x\ /\ g\ x)-c)\leq 1$ **by** *simp*
hence $norm\ (norm\ (g\ x))*norm\ (norm\ (f\ x\ /\ g\ x)-c)\leq norm\ (norm\ (g\ x))*1$
by (*rule mult-left-mono*) *simp-all*
finally show *?case* **by** (*simp add*: *algebra-simps*)
qed
qed

lemma *bigoI-tendsto*:
assumes $((\lambda x.\ f\ x\ /\ g\ x)\longrightarrow c)\ F$
assumes *eventually* $(\lambda x.\ g\ x\neq 0)\ F$
shows $f\in O[F](g)$
using *assms* **by** (*rule bigoI-tendsto-norm*[*OF tendsto-norm*])

lemma *bigomegaI-tendsto-norm*:
assumes *c-not-0*: $(c::real)\neq 0$
assumes *lim*: $((\lambda x.\ norm\ (f\ x\ /\ g\ x))\longrightarrow c)\ F$
shows $f\in \Omega[F](g)$
proof (*cases* $F=bot$)
case *False*
show *?thesis*
proof (*rule landau-omega.bigI*)
from *lim* **have** $c\geq 0$ **by** (*rule tendsto-lowerbound*) (*insert False*, *simp-all*)
with *c-not-0* **have** $c>0$ **by** *simp*
with *c-not-0* **show** $c/2>0$ **by** *simp*
from *lim* **have** *ev*: $\bigwedge\varepsilon.\ \varepsilon>0\Longrightarrow eventually\ (\lambda x.\ norm\ (norm\ (f\ x\ /\ g\ x)-c)<\varepsilon)\ F$
by (*subst* (*asm*) *tendsto-iff*) (*simp add*: *dist-real-def*)
from *ev*[*OF* ‹$c/2>0$›] **show** *eventually* $(\lambda x.\ (norm\ (f\ x))\geq c/2*(norm\ (g\ x)))\ F$
proof (*eventually-elim*)

```
    fix x assume B: norm (norm (f x / g x) - c) < c / 2
    from B have g: g x ≠ 0 by auto
    from B have -c/2 < -norm (norm (f x / g x) - c) by simp
    also have ... ≤ norm (f x / g x) - c by simp
    finally show (norm (f x)) ≥ c/2 * (norm (g x)) using g
      by (simp add: field-simps norm-mult norm-divide)
   qed
  qed
qed simp

lemma bigomegaI-tendsto:
  assumes c-not-0:  (c::real) ≠ 0
  assumes lim:      ((λx. f x / g x) ⟶ c) F
  shows   f ∈ Ω[F](g)
 by (rule bigomegaI-tendsto-norm[OF - tendsto-norm, of c]) (insert assms, simp-all)

lemma smallomegaI-filterlim-at-top-norm:
  assumes lim: filterlim (λx. norm (f x / g x)) at-top F
  shows   f ∈ ω[F](g)
proof (rule landau-omega.smallI)
  fix c :: real assume c-pos: c > 0
  from lim have ev: eventually (λx. norm (f x / g x) ≥ c) F
    by (subst (asm) filterlim-at-top) simp
  thus eventually (λx. (norm (f x)) ≥ c * (norm (g x))) F
  proof eventually-elim
    fix x assume A: norm (f x / g x) ≥ c
    from A c-pos have g x ≠ 0 by auto
     with A show (norm (f x)) ≥ c * (norm (g x)) by (simp add: field-simps
norm-divide)
  qed
qed

lemma smallomegaI-filterlim-at-infinity:
  assumes lim: filterlim (λx. f x / g x) at-infinity F
  shows   f ∈ ω[F](g)
proof (rule smallomegaI-filterlim-at-top-norm)
  from lim show filterlim (λx. norm (f x / g x)) at-top F
    by (rule filterlim-at-infinity-imp-norm-at-top)
qed

lemma smallomegaD-filterlim-at-top-norm:
  assumes f ∈ ω[F](g)
  assumes eventually (λx. g x ≠ 0) F
  shows   LIM x F. norm (f x / g x) :> at-top
proof (subst filterlim-at-top-gt, clarify)
  fix c :: real assume c: c > 0
  from landau-omega.smallD[OF assms(1) this] assms(2)
    show eventually (λx. norm (f x / g x) ≥ c) F
    by eventually-elim (simp add: field-simps norm-divide)
```

qed

lemma *smallomegaD-filterlim-at-infinity*:
assumes $f \in \omega[F](g)$
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
shows *LIM x F. f x / g x :> at-infinity*
using *assms* **by** (*intro filterlim-norm-at-top-imp-at-infinity smallomegaD-filterlim-at-top-norm*)

lemma *smallomega-1-conv-filterlim*: $f \in \omega[F](\lambda\text{-}.\ 1) \longleftrightarrow$ *filterlim f at-infinity F*
by (*auto intro*: *smallomegaI-filterlim-at-infinity dest*: *smallomegaD-filterlim-at-infinity*)

lemma *smalloI-tendsto*:
assumes *lim*: $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 0)\ F$
assumes *eventually* $(\lambda x.\ g\ x \neq 0)\ F$
shows $f \in o[F](g)$
proof (*rule landau-o.smallI*)
fix *c* :: *real* **assume** *c-pos*: $c > 0$
from *c-pos* **and** *lim* **have** *ev*: *eventually* $(\lambda x.\ norm\ (f\ x\ /\ g\ x) < c)\ F$
by (*subst* (*asm*) *tendsto-iff*) (*simp add*: *dist-real-def*)
with *assms*(*2*) **show** *eventually* $(\lambda x.\ (norm\ (f\ x)) \leq c * (norm\ (g\ x)))\ F$
by *eventually-elim* (*simp add*: *field-simps norm-divide*)
qed

lemma *smalloD-tendsto*:
assumes $f \in o[F](g)$
shows $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 0)\ F$
unfolding *tendsto-iff*
proof *clarify*
fix *e* :: *real* **assume** *e*: $e > 0$
hence $e/2 > 0$ **by** *simp*
from *landau-o.smallD*[*OF assms this*] **show** *eventually* $(\lambda x.\ dist\ (f\ x\ /\ g\ x)\ 0 <$
$e)\ F$
proof *eventually-elim*
fix *x* **assume** $(norm\ (f\ x)) \leq e/2 * (norm\ (g\ x))$
with *e* **have** *dist* $(f\ x\ /\ g\ x)\ 0 \leq e/2$
by (*cases g x = 0*) (*simp-all add*: *dist-real-def norm-divide field-simps*)
also from *e* **have** $... < e$ **by** *simp*
finally show *dist* $(f\ x\ /\ g\ x)\ 0 < e$ **by** *simp*
qed
qed

lemma *bigthetaI-tendsto-norm*:
assumes *c-not-0*: $(c::real) \neq 0$
assumes *lim*: $((\lambda x.\ norm\ (f\ x\ /\ g\ x)) \longrightarrow c)\ F$
shows $f \in \Theta[F](g)$
proof (*rule bigthetaI*)
from *c-not-0* **have** $|c| > 0$ **by** *simp*
with *lim* **have** *eventually* $(\lambda x.\ norm\ (norm\ (f\ x\ /\ g\ x) - c) < |c|)\ F$
by (*subst* (*asm*) *tendsto-iff*) (*simp add*: *dist-real-def*)

hence *g*: *eventually* (*λx. g x ≠ 0*) *F* **by** *eventually-elim* (*auto simp add*: *field-simps*)

from *lim g* **show** *f* ∈ *O*[*F*](*g*) **by** (*rule bigoI-tendsto-norm*)
from *c-not-0* **and** *lim* **show** *f* ∈ Ω[*F*](*g*) **by** (*rule bigomegaI-tendsto-norm*)
qed

lemma *bigthetaI-tendsto*:
assumes *c-not-0*: (*c*::*real*) ≠ *0*
assumes *lim*: ((*λx. f x / g x*) ⟶ *c*) *F*
shows *f* ∈ Θ[*F*](*g*)
using *assms* **by** (*intro bigthetaI-tendsto-norm*[*OF - tendsto-norm, of c*]) *simp-all*

lemma *tendsto-add-smallo*:
assumes (*f1* ⟶ *a*) *F*
assumes *f2* ∈ *o*[*F*](*f1*)
shows ((*λx. f1 x + f2 x*) ⟶ *a*) *F*
proof (*subst filterlim-cong*[*OF refl refl*])
from *landau-o.smallD*[*OF assms*(*2*) *zero-less-one*]
have *eventually* (*λx. norm* (*f2 x*) ≤ *norm* (*f1 x*)) *F* **by** *simp*
thus *eventually* (*λx. f1 x + f2 x = f1 x ∗* (*1 + f2 x / f1 x*)) *F*
by *eventually-elim* (*auto simp*: *field-simps*)
next
from *assms*(*1*) **show** ((*λx. f1 x ∗* (*1 + f2 x / f1 x*)) ⟶ *a*) *F*
by (*force intro*: *tendsto-eq-intros smalloD-tendsto*[*OF assms*(*2*)])
qed

lemma *tendsto-diff-smallo*:
shows (*f1* ⟶ *a*) *F* ⟹ *f2* ∈ *o*[*F*](*f1*) ⟹ ((*λx. f1 x − f2 x*) ⟶ *a*) *F*
using *tendsto-add-smallo*[*of f1 a F λx. −f2 x*] **by** *simp*

lemma *tendsto-add-smallo-iff*:
assumes *f2* ∈ *o*[*F*](*f1*)
shows (*f1* ⟶ *a*) *F* ⟷ ((*λx. f1 x + f2 x*) ⟶ *a*) *F*
proof
assume ((*λx. f1 x + f2 x*) ⟶ *a*) *F*
hence ((*λx. f1 x + f2 x − f2 x*) ⟶ *a*) *F*
by (*rule tendsto-diff-smallo*) (*simp add*: *landau-o.small.plus-absorb2 assms*)
thus (*f1* ⟶ *a*) *F* **by** *simp*
qed (*rule tendsto-add-smallo*[*OF - assms*])

lemma *tendsto-diff-smallo-iff*:
shows *f2* ∈ *o*[*F*](*f1*) ⟹ (*f1* ⟶ *a*) *F* ⟷ ((*λx. f1 x − f2 x*) ⟶ *a*) *F*
using *tendsto-add-smallo-iff*[*of λx. −f2 x F f1 a*] **by** *simp*

lemma *tendsto-divide-smallo*:
assumes ((*λx. f1 x / g1 x*) ⟶ *a*) *F*
assumes *f2* ∈ *o*[*F*](*f1*) *g2* ∈ *o*[*F*](*g1*)
assumes *eventually* (*λx. g1 x ≠ 0*) *F*
shows ((*λx.* (*f1 x + f2 x*) / (*g1 x + g2 x*)) ⟶ *a*) *F* (**is** (*?f* ⟶ -) -)

proof (*subst tendsto-cong*)
let *?f'* = λx. (*f1 x* / *g1 x*) ∗ (*1* + *f2 x* / *f1 x*) / (*1* + *g2 x* / *g1 x*)

have (*?f'* ⟶ *a* ∗ (*1* + *0*) / (*1* + *0*)) *F*
by (*rule tendsto-mult tendsto-divide tendsto-add assms tendsto-const*
smalloD-tendsto[*OF assms*(*2*)] *smalloD-tendsto*[*OF assms*(*3*)])+ *simp-all*
thus (*?f'* ⟶ *a*) *F* **by** *simp*

have (*1*/*2*::*real*) > *0* **by** *simp*
from *landau-o.smallD*[*OF assms*(*2*) *this*] *landau-o.smallD*[*OF assms*(*3*) *this*]
have *eventually* (λx. *norm* (*f2 x*) ≤ *norm* (*f1 x*)/*2*) *F*
eventually (λx. *norm* (*g2 x*) ≤ *norm* (*g1 x*)/*2*) *F* **by** *simp-all*
with *assms*(*4*) **show** *eventually* (λx. *?f x* = *?f' x*) *F*
proof *eventually-elim*
fix *x* **assume** *A*: *norm* (*f2 x*) ≤ *norm* (*f1 x*)/*2* **and**
B: *norm* (*g2 x*) ≤ *norm* (*g1 x*)/*2* **and** *C*: *g1 x* ≠ *0*
show *?f x* = *?f' x*
proof (*cases f1 x* = *0*)
assume *D*: *f1 x* ≠ *0*
from *D* **have** *f1 x* + *f2 x* = *f1 x* ∗ (*1* + *f2 x*/*f1 x*) **by** (*simp add*: *field-simps*)
moreover from *C* **have** *g1 x* + *g2 x* = *g1 x* ∗ (*1* + *g2 x*/*g1 x*) **by** (*simp add*: *field-simps*)
ultimately have *?f x* = (*f1 x* ∗ (*1* + *f2 x*/*f1 x*)) / (*g1 x* ∗ (*1* + *g2 x*/*g1 x*))
by (*simp only*:)
also have ... = *?f' x* **by** *simp*
finally show *?thesis* **.**
qed (*insert A*, *simp*)
qed
qed

lemma *bigo-powr*:
fixes *f* :: *'a* ⇒ *real*
assumes *f* ∈ *O*[*F*](*g*) *p* ≥ *0*
shows (λx. |*f x*| *powr p*) ∈ *O*[*F*](λx. |*g x*| *powr p*)
proof−
from *assms*(*1*) **obtain** *c* **where** *c*: *c* > *0* **and** ∗: ∀ *F* *x in F*. *norm* (*f x*) ≤ *c* ∗ *norm* (*g x*)
by (*elim landau-o.bigE landau-omega.bigE IntE*)
from *assms*(*2*) ∗ **have** *eventually* (λx. (*norm* (*f x*)) *powr p* ≤ (*c* ∗ *norm* (*g x*)) *powr p*) *F*
by (*auto elim*!: *eventually-mono intro*!: *powr-mono2*)
with *c* **show** (λx. |*f x*| *powr p*) ∈ *O*[*F*](λx. |*g x*| *powr p*)
by (*intro bigoI*[*of* - *c powr p*]) (*simp-all add*: *powr-mult*)
qed

lemma *smallo-powr*:
fixes *f* :: *'a* ⇒ *real*
assumes *f* ∈ *o*[*F*](*g*) *p* > *0*

shows $(\lambda x.\ |f\ x|\ powr\ p) \in o[F](\lambda x.\ |g\ x|\ powr\ p)$
proof (*rule landau-o.smallI*)
fix c :: *real* **assume** c: $c > 0$
hence $c\ powr\ (1/p) > 0$ **by** *simp*
from *landau-o.smallD*[*OF assms*(*1*) *this*]
show *eventually* $(\lambda x.\ norm\ (|f\ x|\ powr\ p) \leq c * norm\ (|g\ x|\ powr\ p))\ F$
proof *eventually-elim*
fix x **assume** $(norm\ (f\ x)) \leq c\ powr\ (1\ /\ p) * (norm\ (g\ x))$
with *assms*(*2*) **have** $(norm\ (f\ x))\ powr\ p \leq (c\ powr\ (1\ /\ p) * (norm\ (g\ x)))$ *powr p*
by (*intro powr-mono2*) *simp-all*
also from *assms*(*2*) c **have** $... = c * (norm\ (g\ x))\ powr\ p$
by (*simp add*: *field-simps powr-mult powr-powr*)
finally show $norm\ (|f\ x|\ powr\ p) \leq c * norm\ (|g\ x|\ powr\ p)$ **by** *simp*
qed
qed

lemma *smallo-powr-nonneg*:
fixes f :: $'a \Rightarrow real$
assumes $f \in o[F](g)$ $p > 0$ *eventually* $(\lambda x.\ f\ x \geq 0)\ F$ *eventually* $(\lambda x.\ g\ x \geq 0)$ *F*
shows $(\lambda x.\ f\ x\ powr\ p) \in o[F](\lambda x.\ g\ x\ powr\ p)$
proof −
from *assms*(*3*) **have** $(\lambda x.\ f\ x\ powr\ p) \in \Theta[F](\lambda x.\ |f\ x|\ powr\ p)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
also have $(\lambda x.\ |f\ x|\ powr\ p) \in o[F](\lambda x.\ |g\ x|\ powr\ p)$ **by** (*intro smallo-powr*) *fact+*
also from *assms*(*4*) **have** $(\lambda x.\ |g\ x|\ powr\ p) \in \Theta[F](\lambda x.\ g\ x\ powr\ p)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
finally show *?thesis* **.**
qed

lemma *bigtheta-powr*:
fixes f :: $'a \Rightarrow real$
shows $f \in \Theta[F](g) \Longrightarrow (\lambda x.\ |f\ x|\ powr\ p) \in \Theta[F](\lambda x.\ |g\ x|\ powr\ p)$
apply (*cases* $p < 0$)
apply (*subst bigtheta-inverse*[*symmetric*], *subst* (*1 2*) *powr-minus*[*symmetric*])
unfolding *bigtheta-def* **apply** (*auto simp*: *bigomega-iff-bigo intro*!: *bigo-powr*)
done

lemma *bigo-powr-nonneg*:
fixes f :: $'a \Rightarrow real$
assumes $f \in O[F](g)$ $p \geq 0$ *eventually* $(\lambda x.\ f\ x \geq 0)\ F$ *eventually* $(\lambda x.\ g\ x \geq 0)$ *F*
shows $(\lambda x.\ f\ x\ powr\ p) \in O[F](\lambda x.\ g\ x\ powr\ p)$
proof −
from *assms*(*3*) **have** $(\lambda x.\ f\ x\ powr\ p) \in \Theta[F](\lambda x.\ |f\ x|\ powr\ p)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
also have $(\lambda x.\ |f\ x|\ powr\ p) \in O[F](\lambda x.\ |g\ x|\ powr\ p)$ **by** (*intro bigo-powr*) *fact+*

also from *assms*(*4*) **have** $(\lambda x.\ |g\ x|\ powr\ p) \in \Theta[F](\lambda x.\ g\ x\ powr\ p)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
finally show *?thesis* **.**
qed

lemma *zero-in-smallo* [*simp*]: $(\lambda\text{-}.\ 0) \in o[F](f)$
by (*intro landau-o.smallI*) *simp-all*

lemma *zero-in-bigo* [*simp*]: $(\lambda\text{-}.\ 0) \in O[F](f)$
by (*intro landau-o.bigI*[*of 1*]) *simp-all*

lemma *in-bigomega-zero* [*simp*]: $f \in \Omega[F](\lambda x.\ 0)$
by (*rule landau-omega.bigI*[*of 1*]) *simp-all*

lemma *in-smallomega-zero* [*simp*]: $f \in \omega[F](\lambda x.\ 0)$
by (*simp add*: *smallomega-iff-smallo*)

lemma *in-smallo-zero-iff* [*simp*]: $f \in o[F](\lambda\text{-}.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
proof
assume $f \in o[F](\lambda\text{-}.\ 0)$
from *landau-o.smallD*[*OF this*, *of 1*] **show** *eventually* $(\lambda x.\ f\ x = 0)\ F$ **by** *simp*
next
assume *eventually* $(\lambda x.\ f\ x = 0)\ F$
hence $\forall\, c{>}0.$ *eventually* $(\lambda x.\ (norm\ (f\ x)) \leq c * |0|)\ F$ **by** *simp*
thus $f \in o[F](\lambda\text{-}.\ 0)$ **unfolding** *smallo-def* **by** *simp*
qed

lemma *in-bigo-zero-iff* [*simp*]: $f \in O[F](\lambda\text{-}.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
proof
assume $f \in O[F](\lambda\text{-}.\ 0)$
thus *eventually* $(\lambda x.\ f\ x = 0)\ F$ **by** (*elim landau-o.bigE*) *simp*
next
assume *eventually* $(\lambda x.\ f\ x = 0)\ F$
hence *eventually* $(\lambda x.\ (norm\ (f\ x)) \leq 1 * |0|)\ F$ **by** *simp*
thus $f \in O[F](\lambda\text{-}.\ 0)$ **by** (*intro landau-o.bigI*[*of 1*]) *simp-all*
qed

lemma *zero-in-smallomega-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \omega[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
by (*simp add*: *smallomega-iff-smallo*)

lemma *zero-in-bigomega-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \Omega[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
by (*simp add*: *bigomega-iff-bigo*)

lemma *zero-in-bigtheta-iff* [*simp*]: $(\lambda\text{-}.\ 0) \in \Theta[F](f) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
unfolding *bigtheta-def* **by** *simp*

lemma *in-bigtheta-zero-iff* [*simp*]: $f \in \Theta[F](\lambda x.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)$ *F*
unfolding *bigtheta-def* **by** *simp*

lemma *cmult-in-bigo-iff* [*simp*]: $(\lambda x.\ c * f\ x) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
and *cmult-in-bigo-iff'* [*simp*]: $(\lambda x.\ f\ x * c) \in O[F](g) \longleftrightarrow c = 0 \vee f \in O[F](g)$
and *cmult-in-smallo-iff* [*simp*]: $(\lambda x.\ c * f\ x) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
and *cmult-in-smallo-iff'* [*simp*]: $(\lambda x.\ f\ x * c) \in o[F](g) \longleftrightarrow c = 0 \vee f \in o[F](g)$
by (*cases* $c = 0$, *simp*, *simp*)+

lemma *bigo-const* [*simp*]: $(\lambda\text{-}.\ c) \in O[F](\lambda\text{-}.\ 1)$ **by** (*rule bigoI*[*of - norm c*]) *simp*

lemma *bigo-const-iff* [*simp*]: $(\lambda\text{-}.\ c1) \in O[F](\lambda\text{-}.\ c2) \longleftrightarrow F = bot \vee c1 = 0 \vee c2 \neq 0$
by (*cases* $c1 = 0$; *cases* $c2 = 0$)
(*auto simp*: *bigo-def eventually-False intro*: *exI*[*of - 1*] *exI*[*of - norm c1 / norm c2*])

lemma *bigomega-const-iff* [*simp*]: $(\lambda\text{-}.\ c1) \in \Omega[F](\lambda\text{-}.\ c2) \longleftrightarrow F = bot \vee c1 \neq 0 \vee c2 = 0$
by (*cases* $c1 = 0$; *cases* $c2 = 0$)
(*auto simp*: *bigomega-def eventually-False mult-le-0-iff*
intro: *exI*[*of - 1*] *exI*[*of - norm c1 / norm c2*])

lemma *smallo-real-nat-transfer*:
$(f :: real \Rightarrow real) \in o(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in o(\lambda x.\ g\ (real\ x))$
by (*rule landau-o.small.compose*[*OF - filterlim-real-sequentially*])

lemma *bigo-real-nat-transfer*:
$(f :: real \Rightarrow real) \in O(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in O(\lambda x.\ g\ (real\ x))$
by (*rule landau-o.big.compose*[*OF - filterlim-real-sequentially*])

lemma *smallomega-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \omega(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \omega(\lambda x.\ g\ (real\ x))$
by (*rule landau-omega.small.compose*[*OF - filterlim-real-sequentially*])

lemma *bigomega-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \Omega(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \Omega(\lambda x.\ g\ (real\ x))$
by (*rule landau-omega.big.compose*[*OF - filterlim-real-sequentially*])

lemma *bigtheta-real-nat-transfer*:
$(f :: real \Rightarrow real) \in \Theta(g) \Longrightarrow (\lambda x{::}nat.\ f\ (real\ x)) \in \Theta(\lambda x.\ g\ (real\ x))$
unfolding *bigtheta-def* **using** *bigo-real-nat-transfer bigomega-real-nat-transfer*
by *blast*

lemmas *landau-real-nat-transfer* [*intro*] =

bigo-real-nat-transfer smallo-real-nat-transfer bigomega-real-nat-transfer smallomega-real-nat-transfer bigtheta-real-nat-transfer

lemma *landau-symbol-if-at-top-eq* [*simp*]:
assumes *landau-symbol L L' Lr*
shows *L at-top* (λx::'a::*linordered-semidom*. *if x = a then f x else g x*) = *L at-top* (*g*)
apply (*rule landau-symbol.cong*[*OF assms*])
by (*auto simp add*: *frequently-def eventually-at-top-linorder dest*!: *spec* [**where** *x*= *a + 1*])

lemmas *landau-symbols-if-at-top-eq* [*simp*] = *landau-symbols*[*THEN landau-symbol-if-at-top-eq*]

lemma *sum-in-smallo*:
assumes *f* ∈ *o*[*F*](*h*) *g* ∈ *o*[*F*](*h*)
shows (λx. *f x + g x*) ∈ *o*[*F*](*h*) (λx. *f x − g x*) ∈ *o*[*F*](*h*)
proof −
have (λx. *f x + g x*) ∈ *o*[*F*](*h*) **if** *f* ∈ *o*[*F*](*h*) *g* ∈ *o*[*F*](*h*) **for** *f g*
proof (*rule landau-o.smallI*)
fix *c* :: *real* **assume** *c > 0*
hence *c*/*2 > 0* **by** *simp*
from *that*[*THEN landau-o.smallD*[*OF - this*]]
show *eventually* (λx. *norm* (*f x + g x*) ≤ *c* ∗ (*norm* (*h x*))) *F*
by *eventually-elim* (*auto intro*: *order.trans*[*OF norm-triangle-ineq*])
qed
from *this*[*of f g*] *this*[*of f* λx. −*g x*] *assms*
show (λx. *f x + g x*) ∈ *o*[*F*](*h*) (λx. *f x − g x*) ∈ *o*[*F*](*h*) **by** *simp-all*
qed

lemma *big-sum-in-smallo*:
assumes ⋀x. *x* ∈ *A* ⟹ *f x* ∈ *o*[*F*](*g*)
shows (λx. *sum* (λy. *f y x*) *A*) ∈ *o*[*F*](*g*)
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*: *sum-in-smallo*)

lemma *sum-in-bigo*:
assumes *f* ∈ *O*[*F*](*h*) *g* ∈ *O*[*F*](*h*)
shows (λx. *f x + g x*) ∈ *O*[*F*](*h*) (λx. *f x − g x*) ∈ *O*[*F*](*h*)
proof −
have (λx. *f x + g x*) ∈ *O*[*F*](*h*) **if** *f* ∈ *O*[*F*](*h*) *g* ∈ *O*[*F*](*h*) **for** *f g*
proof −
from *that* **obtain** *c1 c2* **where** ∗: *c1 > 0 c2 > 0*
and ∗∗: ∀ *F x in F*. *norm* (*f x*) ≤ *c1* ∗ *norm* (*h x*)
∀ *F x in F*. *norm* (*g x*) ≤ *c2* ∗ *norm* (*h x*)
by (*elim landau-o.bigE*)
from ∗∗ **have** *eventually* (λx. *norm* (*f x + g x*) ≤ (*c1 + c2*) ∗ (*norm* (*h x*)))
F

by *eventually-elim* (*auto simp*: *algebra-simps intro*: *order.trans*[*OF norm-triangle-ineq*])
then show *?thesis* **by** (*rule bigoI*)
qed
from *assms this*[*of f g*] *this*[*of f* $\lambda x.\ -\ g\ x$]
show $(\lambda x.\ f\ x + g\ x) \in O[F](h)$ $(\lambda x.\ f\ x - g\ x) \in O[F](h)$ **by** *simp-all*
qed

lemma *big-sum-in-bigo*:
assumes $\bigwedge x.\ x \in A \Longrightarrow f\ x \in O[F](g)$
shows $(\lambda x.\ sum\ (\lambda y.\ f\ y\ x)\ A) \in O[F](g)$
using *assms* **by** (*induction A rule*: *infinite-finite-induct*) (*auto intro*: *sum-in-bigo*)

lemma *smallo-multiples*:
assumes *f*: $f \in o(real)$ **and** $k>0$
shows $(\lambda n.\ f\ (k * n)) \in o(real)$
proof (*clarsimp simp*: *smallo-def*)
fix *c*::*real*
assume $c>0$
then have $c/k > 0$
by (*simp add*: *assms*)
with *assms* **have** $\forall_F\ n$ *in sequentially*. $|f\ n| \leq c\ /\ real\ k * n$
by (*force simp*: *smallo-def del*: *divide-const-simps*)
then obtain *N* **where** $\bigwedge n.\ n \geq N \Longrightarrow |f\ n| \leq c/k * n$
by (*meson eventually-at-top-linorder*)
then have $\bigwedge m.\ (k*m) \geq N \Longrightarrow |f\ (k*m)| \leq c/k * (k*m)$
by *blast*
with ‹$k>0$› **have** $\forall_F\ m$ *in sequentially*. $|f\ (k*m)| \leq c/k * (k*m)$
by (*smt* (*verit*, *del-insts*) *One-nat-def Suc-leI eventually-at-top-linorderI mult-1-left mult-le-mono*)
then show $\forall_F\ n$ *in sequentially*. $|f\ (k * n)| \leq c * n$
by *eventually-elim* (*use* ‹$k>0$› **in** *auto*)
qed

lemma *maxmin-in-smallo*:
assumes $f \in o[F](h)$ $g \in o[F](h)$
shows $(\lambda k.\ max\ (f\ k)\ (g\ k)) \in o[F](h)$ $(\lambda k.\ min\ (f\ k)\ (g\ k)) \in o[F](h)$
proof −
have $\forall_F\ x$ *in F*. $norm\ (max\ (f\ x)\ (g\ x)) \leq c * norm(h\ x) \wedge norm\ (min\ (f\ x)\ (g\ x)) \leq c * norm(h\ x)$
if $c>0$ **for** *c*::*real*
proof −
from *assms smallo-def that*
have $\forall_F\ x$ *in F*. $norm\ (f\ x) \leq c * norm(h\ x)$ $\forall_F\ x$ *in F*. $norm(g\ x) \leq c * norm(h\ x)$
by (*auto simp*: *smallo-def*)
then show *?thesis*
by (*smt* (*verit*) *eventually-elim2 max-def min-def*)
qed
with *assms* **show** $(\lambda x.\ max\ (f\ x)\ (g\ x)) \in o[F](h)$ $(\lambda x.\ min\ (f\ x)\ (g\ x)) \in o[F](h)$

by (*smt* (*verit*) *eventually-elim2 landau-o.smallI*)+
qed

lemma *le-imp-bigo-real*:
assumes $c \geq 0$ *eventually* $(\lambda x.\ f\ x \leq c * (g\ x :: real))\ F$ *eventually* $(\lambda x.\ 0 \leq f\ x)\ F$
shows $f \in O[F](g)$
proof −
have *eventually* $(\lambda x.\ norm\ (f\ x) \leq c * norm\ (g\ x))\ F$
using *assms*(*2*,*3*)
proof *eventually-elim*
case (*elim x*)
have $norm\ (f\ x) \leq c * g\ x$ **using** *elim* **by** *simp*
also have $\ldots \leq c * norm\ (g\ x)$ **by** (*intro mult-left-mono assms*) *auto*
finally show *?case* **.**
qed
thus *?thesis* **by** (*intro bigoI*[*of* - *c*]) *auto*
qed

context *landau-symbol*
begin

lemma *mult-cancel-left*:
assumes $f1 \in \Theta[F](g1)$ **and** *eventually* $(\lambda x.\ g1\ x \neq 0)\ F$
notes [*trans*] = *bigtheta-trans1 bigtheta-trans2*
shows $(\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x) \longleftrightarrow f2 \in L\ F\ (g2)$
proof
assume *A*: $(\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x)$
from *assms* **have** *nz*: *eventually* $(\lambda x.\ f1\ x \neq 0)\ F$ **by** (*simp add*: *eventually-nonzero-bigtheta*)
hence $f2 \in \Theta[F](\lambda x.\ f1\ x * f2\ x\ /\ f1\ x)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
also from *A assms nz* **have** $(\lambda x.\ f1\ x * f2\ x\ /\ f1\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x\ /\ f1\ x)$
by (*intro divide-right*) *simp-all*
also from *assms nz* **have** $(\lambda x.\ g1\ x * g2\ x\ /\ f1\ x) \in \Theta[F](\lambda x.\ g1\ x * g2\ x\ /\ g1\ x)$
by (*intro landau-theta.mult landau-theta.divide*) (*simp-all add*: *bigtheta-sym*)
also from *assms* **have** $(\lambda x.\ g1\ x * g2\ x\ /\ g1\ x) \in \Theta[F](g2)$
by (*intro bigthetaI-cong*) (*auto elim*!: *eventually-mono*)
finally show $f2 \in L\ F\ (g2)$ **.**
next
assume $f2 \in L\ F\ (g2)$
hence $(\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ f1\ x * g2\ x)$ **by** (*rule mult-left*)
also have $(\lambda x.\ f1\ x * g2\ x) \in \Theta[F](\lambda x.\ g1\ x * g2\ x)$
by (*intro landau-theta.mult-right assms*)
finally show $(\lambda x.\ f1\ x * f2\ x) \in L\ F\ (\lambda x.\ g1\ x * g2\ x)$ **.**
qed

lemma *mult-cancel-right*:
assumes *f2* ∈ Θ[*F*](*g2*) **and** *eventually* (λ*x*. *g2 x* ≠ *0*) *F*
shows (λ*x*. *f1 x* ∗ *f2 x*) ∈ *L F* (λ*x*. *g1 x* ∗ *g2 x*) ⟷ *f1* ∈ *L F* (*g1*)
by (*subst* (*1 2*) *mult.commute*) (*rule mult-cancel-left*[*OF assms*])

lemma *divide-cancel-right*:
assumes *f2* ∈ Θ[*F*](*g2*) **and** *eventually* (λ*x*. *g2 x* ≠ *0*) *F*
shows (λ*x*. *f1 x* / *f2 x*) ∈ *L F* (λ*x*. *g1 x* / *g2 x*) ⟷ *f1* ∈ *L F* (*g1*)
by (*subst* (*1 2*) *divide-inverse*, *intro mult-cancel-right bigtheta-inverse*) (*simp-all add*: *assms*)

lemma *divide-cancel-left*:
assumes *f1* ∈ Θ[*F*](*g1*) **and** *eventually* (λ*x*. *g1 x* ≠ *0*) *F*
shows (λ*x*. *f1 x* / *f2 x*) ∈ *L F* (λ*x*. *g1 x* / *g2 x*) ⟷
(λ*x*. *inverse* (*f2 x*)) ∈ *L F* (λ*x*. *inverse* (*g2 x*))
by (*simp only*: *divide-inverse mult-cancel-left*[*OF assms*])

end

lemma *powr-smallo-iff*:
assumes *filterlim g at-top F F* ≠ *bot*
shows (λ*x*. *g x powr p* :: *real*) ∈ *o*[*F*](λ*x*. *g x powr q*) ⟷ *p* < *q*
proof−
from *assms* **have** *eventually* (λ*x*. *g x* ≥ *1*) *F* **by** (*force simp*: *filterlim-at-top*)
hence *A*: *eventually* (λ*x*. *g x* ≠ *0*) *F* **by** *eventually-elim simp*
have *B*: (λ*x*. *g x powr q*) ∈ *O*[*F*](λ*x*. *g x powr p*) ⟹ (λ*x*. *g x powr p*) ∉ *o*[*F*](λ*x*. *g x powr q*)
proof
assume (λ*x*. *g x powr q*) ∈ *O*[*F*](λ*x*. *g x powr p*) (λ*x*. *g x powr p*) ∈ *o*[*F*](λ*x*. *g x powr q*)
from *landau-o.big-small-asymmetric*[*OF this*] **have** *eventually* (λ*x*. *g x* = *0*) *F* **by** *simp*
with *A* **have** *eventually* (λ-::′*a*. *False*) *F* **by** *eventually-elim simp*
thus *False* **by** (*simp add*: *eventually-False assms*)
qed
show *?thesis*
proof (*cases p q rule*: *linorder-cases*)
assume *p* < *q*
hence (λ*x*. *g x powr p*) ∈ *o*[*F*](λ*x*. *g x powr q*) **using** *assms A*
by (*auto intro*!: *smalloI-tendsto tendsto-neg-powr simp flip*: *powr-diff*)
with ‹*p* < *q*› **show** *?thesis* **by** *auto*
next
assume *p* = *q*
hence (λ*x*. *g x powr q*) ∈ *O*[*F*](λ*x*. *g x powr p*) **by** (*auto intro*!: *bigthetaD1*)
with *B* ‹*p* = *q*› **show** *?thesis* **by** *auto*
next
assume *p* > *q*
hence (λ*x*. *g x powr q*) ∈ *O*[*F*](λ*x*. *g x powr p*) **using** *assms A*

by (*auto intro*!: *smalloI-tendsto tendsto-neg-powr landau-o.small-imp-big simp flip*: *powr-diff*)
with *B* ‹*p* > *q*› **show** *?thesis* **by** *auto*
qed
qed

lemma *powr-bigo-iff*:
assumes *filterlim g at-top F F* ≠ *bot*
shows (λ*x*. *g x powr p* :: *real*) ∈ *O*[*F*](λ*x*. *g x powr q*) ⟷ *p* ≤ *q*
proof−
from *assms* **have** *eventually* (λ*x*. *g x* ≥ *1*) *F* **by** (*force simp*: *filterlim-at-top*)
hence *A*: *eventually* (λ*x*. *g x* ≠ *0*) *F* **by** *eventually-elim simp*
have *B*: (λ*x*. *g x powr q*) ∈ *o*[*F*](λ*x*. *g x powr p*) ⟹ (λ*x*. *g x powr p*) ∉ *O*[*F*](λ*x*. *g x powr q*)
proof
assume (λ*x*. *g x powr q*) ∈ *o*[*F*](λ*x*. *g x powr p*) (λ*x*. *g x powr p*) ∈ *O*[*F*](λ*x*. *g x powr q*)
from *landau-o.small-big-asymmetric*[*OF this*] **have** *eventually* (λ*x*. *g x* = *0*) *F* **by** *simp*
with *A* **have** *eventually* (λ-::'*a*. *False*) *F* **by** *eventually-elim simp*
thus *False* **by** (*simp add*: *eventually-False assms*)
qed
show *?thesis*
proof (*cases p q rule*: *linorder-cases*)
assume *p* < *q*
hence (λ*x*. *g x powr p*) ∈ *o*[*F*](λ*x*. *g x powr q*) **using** *assms A*
by (*auto intro*!: *smalloI-tendsto tendsto-neg-powr simp flip*: *powr-diff*)
with ‹*p* < *q*› **show** *?thesis* **by** (*auto intro*: *landau-o.small-imp-big*)
next
assume *p* = *q*
hence (λ*x*. *g x powr q*) ∈ *O*[*F*](λ*x*. *g x powr p*) **by** (*auto intro*!: *bigthetaD1*)
with *B* ‹*p* = *q*› **show** *?thesis* **by** *auto*
next
assume *p* > *q*
hence (λ*x*. *g x powr q*) ∈ *o*[*F*](λ*x*. *g x powr p*) **using** *assms A*
by (*auto intro*!: *smalloI-tendsto tendsto-neg-powr simp flip*: *powr-diff*)
with *B* ‹*p* > *q*› **show** *?thesis* **by** (*auto intro*: *landau-o.small-imp-big*)
qed
qed

lemma *powr-bigtheta-iff*:
assumes *filterlim g at-top F F* ≠ *bot*
shows (λ*x*. *g x powr p* :: *real*) ∈ Θ[*F*](λ*x*. *g x powr q*) ⟷ *p* = *q*
using *assms* **unfolding** *bigtheta-def* **by** (*auto simp*: *bigomega-iff-bigo powr-bigo-iff*)

55.3 Flatness of real functions

Given two real-valued functions f and g, we say that f is flatter than g if any power of $f(x)$ is asymptotically dominated by any positive power of $g(x)$.

This is a useful notion since, given two products of powers of functions sorted by flatness, we can compare them asymptotically by simply comparing the exponent lists lexicographically.

A simple sufficient criterion for flatness it that $\ln f(x) \in o(\ln g(x))$, which we show now.

lemma *ln-smallo-imp-flat*:
fixes *f g* :: *real* ⇒ *real*
assumes *lim-f*: *filterlim f at-top at-top*
assumes *lim-g*: *filterlim g at-top at-top*
assumes *ln-o-ln*: (λ*x*. *ln* (*f x*)) ∈ *o*(λ*x*. *ln* (*g x*))
assumes *q*: *q* > *0*
shows (λ*x*. *f x powr p*) ∈ *o*(λ*x*. *g x powr q*)
proof (*rule smalloI-tendsto*)
from *lim-f* **have** *eventually* (λ*x*. *f x* > *0*) *at-top*
by (*simp add*: *filterlim-at-top-dense*)
hence *f-nz*: *eventually* (λ*x*. *f x* ≠ *0*) *at-top* **by** *eventually-elim simp*

from *lim-g* **have** *g-gt-1*: *eventually* (λ*x*. *g x* > *1*) *at-top*
by (*simp add*: *filterlim-at-top-dense*)
hence *g-nz*: *eventually* (λ*x*. *g x* ≠ *0*) *at-top* **by** *eventually-elim simp*
thus *eventually* (λ*x*. *g x powr q* ≠ *0*) *at-top*
by *eventually-elim simp*

have *eq*: *eventually* (λ*x*. *q* ∗ (*p*/*q* ∗ (*ln* (*f x*) / *ln* (*g x*)) − *1*) ∗ *ln* (*g x*) =
p ∗ *ln* (*f x*) − *q* ∗ *ln* (*g x*)) *at-top*
using *g-gt-1* **by** *eventually-elim* (*insert q*, *simp-all add*: *field-simps*)
have *filterlim* (λ*x*. *q* ∗ (*p*/*q* ∗ (*ln* (*f x*) / *ln* (*g x*)) − *1*) ∗ *ln* (*g x*)) *at-bot at-top*
by (*insert q*)
(*rule filterlim-tendsto-neg-mult-at-bot tendsto-mult*
tendsto-const tendsto-diff smalloD-tendsto[*OF ln-o-ln*] *lim-g*
filterlim-compose[*OF ln-at-top*] | *simp*)+
hence *filterlim* (λ*x*. *p* ∗ *ln* (*f x*) − *q* ∗ *ln* (*g x*)) *at-bot at-top*
by (*subst* (*asm*) *filterlim-cong*[*OF refl refl eq*])
hence ∗: ((λ*x*. *exp* (*p* ∗ *ln* (*f x*) − *q* ∗ *ln* (*g x*))) ⟶ *0*) *at-top*
by (*rule filterlim-compose*[*OF exp-at-bot*])
have *eq*: *eventually* (λ*x*. *exp* (*p* ∗ *ln* (*f x*) − *q* ∗ *ln* (*g x*)) = *f x powr p* / *g x powr q*) *at-top*
using *f-nz g-nz* **by** *eventually-elim* (*simp add*: *powr-def exp-diff*)
show ((λ*x*. *f x powr p* / *g x powr q*) ⟶ *0*) *at-top*
using ∗ **by** (*subst* (*asm*) *filterlim-cong*[*OF refl refl eq*])
qed

lemma *ln-smallo-imp-flat′*:
fixes *f g* :: *real* ⇒ *real*
assumes *lim-f*: *filterlim f at-top at-top*
assumes *lim-g*: *filterlim g at-top at-top*
assumes *ln-o-ln*: (λ*x*. *ln* (*f x*)) ∈ *o*(λ*x*. *ln* (*g x*))
assumes *q*: *q* < *0*

shows $(\lambda x.\ g\ x\ powr\ q) \in o(\lambda x.\ f\ x\ powr\ p)$
proof −
from *lim-f lim-g* **have** *eventually* $(\lambda x.\ f\ x > 0)$ *at-top eventually* $(\lambda x.\ g\ x > 0)$ *at-top*
by (*simp-all add*: *filterlim-at-top-dense*)
hence *eventually* $(\lambda x.\ f\ x \neq 0)$ *at-top eventually* $(\lambda x.\ g\ x \neq 0)$ *at-top*
by (*auto elim*: *eventually-mono*)
moreover from *assms* **have** $(\lambda x.\ f\ x\ powr\ -p) \in o(\lambda x.\ g\ x\ powr\ -q)$
by (*intro ln-smallo-imp-flat assms*) *simp-all*
ultimately show *?thesis* **unfolding** *powr-minus*
by (*simp add*: *landau-o.small.inverse-cancel*)
qed

55.4 Asymptotic Equivalence

named-theorems *asymp-equiv-intros*
named-theorems *asymp-equiv-simps*

definition *asymp-equiv* :: $('a \Rightarrow ('b :: real\text{-}normed\text{-}field)) \Rightarrow 'a\ filter \Rightarrow ('a \Rightarrow 'b) \Rightarrow bool$
(‹(‹*open-block notation*=‹*mixfix asymp-equiv*››- ∼[-] -)› [*51*, *10*, *51*] *50*)
where $f \sim[F]\ g \longleftrightarrow ((\lambda x.\ if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$

abbreviation (*input*) *asymp-equiv-at-top* **where**
$asymp\text{-}equiv\text{-}at\text{-}top\ f\ g \equiv f \sim[at\text{-}top]\ g$

bundle *asymp-equiv-syntax*
begin
notation *asymp-equiv-at-top* (**infix** ‹∼› *50*)
end

lemma *asymp-equivI*: $((\lambda x.\ if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x\ /\ g\ x) \longrightarrow 1)\ F \Longrightarrow f \sim[F]\ g$
by (*simp add*: *asymp-equiv-def*)

lemma *asymp-equivD*: $f \sim[F]\ g \Longrightarrow ((\lambda x.\ if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
by (*simp add*: *asymp-equiv-def*)

lemma *asymp-equiv-filtermap-iff*:
$f \sim[filtermap\ h\ F]\ g \longleftrightarrow (\lambda x.\ f\ (h\ x)) \sim[F]\ (\lambda x.\ g\ (h\ x))$
by (*simp add*: *asymp-equiv-def filterlim-filtermap*)

lemma *asymp-equiv-refl* [*simp*, *asymp-equiv-intros*]: $f \sim[F]\ f$
proof (*intro asymp-equivI*)
have *eventually* $(\lambda x.\ 1 = (if\ f\ x = 0 \wedge f\ x = 0\ then\ 1\ else\ f\ x\ /\ f\ x))\ F$
by (*intro always-eventually*) *simp*
moreover have $((\lambda\text{-}.\ 1) \longrightarrow 1)\ F$ **by** *simp*
ultimately show $((\lambda x.\ if\ f\ x = 0 \wedge f\ x = 0\ then\ 1\ else\ f\ x\ /\ f\ x) \longrightarrow 1)\ F$

by (*simp add*: *tendsto-eventually*)
qed

lemma *asymp-equiv-symI*:
assumes $f \sim[F] g$
shows $g \sim[F] f$
using *tendsto-inverse*[*OF asymp-equivD*[*OF assms*]]
by (*auto intro*!: *asymp-equivI simp*: *if-distrib conj-commute cong*: *if-cong*)

lemma *asymp-equiv-sym*: $f \sim[F] g \longleftrightarrow g \sim[F] f$
by (*blast intro*: *asymp-equiv-symI*)

lemma *asymp-equivI'*:
assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
shows $f \sim[F] g$
proof (*cases* $F = bot$)
case *False*
have *eventually* $(\lambda x.\ f\ x \neq 0)\ F$
proof (*rule ccontr*)
assume $\neg$*eventually* $(\lambda x.\ f\ x \neq 0)\ F$
hence *frequently* $(\lambda x.\ f\ x = 0)\ F$ **by** (*simp add*: *frequently-def*)
hence *frequently* $(\lambda x.\ f\ x\ /\ g\ x = 0)\ F$ **by** (*auto elim*!: *frequently-elim1*)
from *limit-frequently-eq*[*OF False this assms*] **show** *False* **by** *simp-all*
qed
hence *eventually* $(\lambda x.\ f\ x\ /\ g\ x = (if\ f\ x = 0 \wedge g\ x = 0\ then\ 1\ else\ f\ x\ /\ g\ x))\ F$
by *eventually-elim simp*
with *assms* **show** $f \sim[F] g$ **unfolding** *asymp-equiv-def*
by (*rule Lim-transform-eventually*)
qed (*simp-all add*: *asymp-equiv-def*)

lemma *tendsto-imp-asymp-equiv-const*:
assumes $(f \longrightarrow c)\ F\ c \neq 0$
shows $f \sim[F] (\lambda\text{-}.\ c)$
by (*rule asymp-equivI' tendsto-eq-intros assms refl*)+ (*use assms* **in** *auto*)

lemma *asymp-equiv-cong*:
assumes *eventually* $(\lambda x.\ f1\ x = f2\ x)\ F$ *eventually* $(\lambda x.\ g1\ x = g2\ x)\ F$
shows $f1 \sim[F] g1 \longleftrightarrow f2 \sim[F] g2$
unfolding *asymp-equiv-def*
proof (*rule tendsto-cong*, *goal-cases*)
case *1*
from *assms* **show** *?case* **by** *eventually-elim simp*
qed

lemma *asymp-equiv-eventually-zeros*:
fixes $f\ g :: 'a \Rightarrow 'b :: real\text{-}normed\text{-}field$
assumes $f \sim[F] g$
shows *eventually* $(\lambda x.\ f\ x = 0 \longleftrightarrow g\ x = 0)\ F$
proof −

let *?h* = λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*
have *eventually* (λ*x*. *x* ≠ *0*) (*nhds* (*1*::*'b*))
by (*rule t1-space-nhds*) *auto*
hence *eventually* (λ*x*. *x* ≠ *0*) (*filtermap ?h F*)
using *assms* **unfolding** *asymp-equiv-def filterlim-def*
by (*rule filter-leD* [*rotated*])
hence *eventually* (λ*x*. *?h x* ≠ *0*) *F* **by** (*simp add*: *eventually-filtermap*)
thus *?thesis* **by** *eventually-elim* (*auto split*: *if-splits*)
qed

lemma *asymp-equiv-transfer*:
assumes *f1* ∼[*F*] *g1 eventually* (λ*x*. *f1 x* = *f2 x*) *F eventually* (λ*x*. *g1 x* = *g2 x*) *F*
shows *f2* ∼[*F*] *g2*
using *assms*(*1*) *asymp-equiv-cong*[*OF assms*(*2*,*3*)] **by** *simp*

lemma *asymp-equiv-transfer-trans* [*trans*]:
assumes (λ*x*. *f x* (*h1 x*)) ∼[*F*] (λ*x*. *g x* (*h1 x*))
assumes *eventually* (λ*x*. *h1 x* = *h2 x*) *F*
shows (λ*x*. *f x* (*h2 x*)) ∼[*F*] (λ*x*. *g x* (*h2 x*))
by (*rule asymp-equiv-transfer*[*OF assms*(*1*)]) (*insert assms*(*2*), *auto elim*!: *eventually-mono*)

lemma *asymp-equiv-trans* [*trans*]:
fixes *f g h*
assumes *f* ∼[*F*] *g g* ∼[*F*] *h*
shows *f* ∼[*F*] *h*
proof −
let *?T* = λ*f g x*. *if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*
from *tendsto-mult*[*OF assms*[*THEN asymp-equivD*]]
have ((λ*x*. *?T f g x* ∗ *?T g h x*) ⟶ *1*) *F* **by** *simp*
moreover from *assms*[*THEN asymp-equiv-eventually-zeros*]
have *eventually* (λ*x*. *?T f g x* ∗ *?T g h x* = *?T f h x*) *F* **by** *eventually-elim simp*
ultimately show *?thesis* **unfolding** *asymp-equiv-def* **by** (*rule Lim-transform-eventually*)
qed

lemma *asymp-equiv-trans-lift1* [*trans*]:
assumes *a* ∼[*F*] *f b b* ∼[*F*] *c* ⋀*c d*. *c* ∼[*F*] *d* ⟹ *f c* ∼[*F*] *f d*
shows *a* ∼[*F*] *f c*
using *assms* **by** (*blast intro*: *asymp-equiv-trans*)

lemma *asymp-equiv-trans-lift2* [*trans*]:
assumes *f a* ∼[*F*] *b a* ∼[*F*] *c* ⋀*c d*. *c* ∼[*F*] *d* ⟹ *f c* ∼[*F*] *f d*
shows *f c* ∼[*F*] *b*
using *asymp-equiv-symI*[*OF assms*(*3*)[*OF assms*(*2*)]] *assms*(*1*)
by (*blast intro*: *asymp-equiv-trans*)

lemma *asymp-equivD-const*:
assumes *f* ∼[*F*] (λ-. *c*)

```
  shows   (f ⟶ c) F
proof (cases c = 0)
  case False
  with tendsto-mult-right[OF asymp-equivD[OF assms], of c] show ?thesis by simp
next
  case True
  with asymp-equiv-eventually-zeros[OF assms] show ?thesis
    by (simp add: tendsto-eventually)
qed

lemma asymp-equiv-refl-ev:
  assumes eventually (λx. f x = g x) F
  shows   f ∼[F] g
  by (intro asymp-equivI tendsto-eventually)
     (insert assms, auto elim!: eventually-mono)

lemma asymp-equiv-nhds-iff: f ∼[nhds (z :: 'a :: t1-space)] g ⟷ f ∼[at z] g ∧ f
z = g z
  by (auto simp: asymp-equiv-def tendsto-nhds-iff)

lemma asymp-equiv-sandwich:
  fixes f g h :: 'a ⇒ 'b :: {real-normed-field, order-topology, linordered-field}
  assumes eventually (λx. f x ≥ 0) F
  assumes eventually (λx. f x ≤ g x) F
  assumes eventually (λx. g x ≤ h x) F
  assumes f ∼[F] h
  shows   g ∼[F] f g ∼[F] h
proof −
  show g ∼[F] f
  proof (rule asymp-equivI, rule tendsto-sandwich)
    from assms(1−3) asymp-equiv-eventually-zeros[OF assms(4)]
      show eventually (λn. (if h n = 0 ∧ f n = 0 then 1 else h n / f n) ≥
                           (if g n = 0 ∧ f n = 0 then 1 else g n / f n)) F
        by eventually-elim (auto intro!: divide-right-mono)
    from assms(1−3) asymp-equiv-eventually-zeros[OF assms(4)]
      show eventually (λn. 1 ≤
                           (if g n = 0 ∧ f n = 0 then 1 else g n / f n)) F
        by eventually-elim (auto intro!: divide-right-mono)
  qed (insert asymp-equiv-symI[OF assms(4)], simp-all add: asymp-equiv-def)
  also note ‹f ∼[F] h›
  finally show g ∼[F] h .
qed

lemma asymp-equiv-imp-eventually-same-sign:
  fixes f g :: real ⇒ real
  assumes f ∼[F] g
  shows   eventually (λx. sgn (f x) = sgn (g x)) F
proof −
  from assms have ((λx. sgn (if f x = 0 ∧ g x = 0 then 1 else f x / g x)) ⟶
```

sgn 1) *F*
unfolding *asymp-equiv-def* **by** (*rule tendsto-sgn*) *simp-all*
from *order-tendstoD*(*1*)[*OF this*, *of 1/2*]
have *eventually* (λ*x*. *sgn* (*if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*) > *1/2*) *F*
by *simp*
thus *eventually* (λ*x*. *sgn* (*f x*) = *sgn* (*g x*)) *F*
proof *eventually-elim*
case (*elim x*)
thus *?case*
by (*cases f x 0* :: *real rule*: *linorder-cases*;
cases g x 0 :: *real rule*: *linorder-cases*) *simp-all*
qed
qed

lemma
fixes *f g* :: - ⇒ *real*
assumes *f* ∼[*F*] *g*
shows *asymp-equiv-eventually-same-sign*: *eventually* (λ*x*. *sgn* (*f x*) = *sgn* (*g x*)) *F* (**is** *?th1*)
and *asymp-equiv-eventually-neg-iff*: *eventually* (λ*x*. *f x* < *0* ⟷ *g x* < *0*) *F* (**is** *?th2*)
and *asymp-equiv-eventually-pos-iff*: *eventually* (λ*x*. *f x* > *0* ⟷ *g x* > *0*) *F* (**is** *?th3*)
proof −
from *assms* **have** *filterlim* (λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*) (*nhds 1*) *F*
by (*rule asymp-equivD*)
from *order-tendstoD*(*1*)[*OF this zero-less-one*]
show *?th1 ?th2 ?th3*
by (*eventually-elim*; *force simp*: *sgn-if field-split-simps split*: *if-splits*)+
qed

lemma *asymp-equiv-tendsto-transfer*:
assumes *f* ∼[*F*] *g* **and** (*f* ⟶ *c*) *F*
shows (*g* ⟶ *c*) *F*
proof −
let *?h* = λ*x*. (*if g x* = *0* ∧ *f x* = *0 then 1 else g x* / *f x*) ∗ *f x*
from *assms*(*1*) **have** *g* ∼[*F*] *f* **by** (*rule asymp-equiv-symI*)
hence *filterlim* (λ*x*. *if g x* = *0* ∧ *f x* = *0 then 1 else g x* / *f x*) (*nhds 1*) *F*
by (*rule asymp-equivD*)
from *tendsto-mult*[*OF this assms*(*2*)] **have** (*?h* ⟶ *c*) *F* **by** *simp*
moreover
have *eventually* (λ*x*. *?h x* = *g x*) *F*
using *asymp-equiv-eventually-zeros*[*OF assms*(*1*)] **by** *eventually-elim simp*
ultimately show *?thesis*
by (*rule Lim-transform-eventually*)
qed

lemma *tendsto-asymp-equiv-cong*:

assumes $f \sim[F]\ g$
shows $(f \longrightarrow c)\ F \longleftrightarrow (g \longrightarrow c)\ F$
proof −
have $(f \longrightarrow c * 1)\ F$ **if** *fg*: $f \sim[F]\ g$ **and** $(g \longrightarrow c)\ F$ **for** $f\ g :: 'a \Rightarrow 'b$
proof −
from *that* **have** ∗: $((\lambda x.\ g\ x * (\mathit{if}\ f\ x = 0 \wedge g\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ f\ x\ /\ g\ x)) \longrightarrow c * 1)\ F$
by (*intro tendsto-intros asymp-equivD*)
have *eventually* $(\lambda x.\ g\ x * (\mathit{if}\ f\ x = 0 \wedge g\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ f\ x\ /\ g\ x) = f\ x)\ F$
using *asymp-equiv-eventually-zeros*[*OF fg*] **by** *eventually-elim simp*
with ∗ **show** *?thesis* **by** (*rule Lim-transform-eventually*)
qed
from *this*[*of f g*] *this*[*of g f*] *assms* **show** *?thesis* **by** (*auto simp*: *asymp-equiv-sym*)
qed

lemma *smallo-imp-eventually-sgn*:
fixes $f\ g :: \mathit{real} \Rightarrow \mathit{real}$
assumes $g \in o(f)$
shows *eventually* $(\lambda x.\ \mathit{sgn}\ (f\ x + g\ x) = \mathit{sgn}\ (f\ x))$ *at-top*
proof −
have $0 < (1/2 :: \mathit{real})$ **by** *simp*
from *landau-o.smallD*[*OF assms*, *OF this*]
have *eventually* $(\lambda x.\ |g\ x| \leq 1/2 * |f\ x|)$ *at-top* **by** *simp*
thus *?thesis*
proof *eventually-elim*
case (*elim x*)
thus *?case*
by (*cases f x 0*::*real rule*: *linorder-cases*;
cases f x + g x 0::*real rule*: *linorder-cases*) *simp-all*
qed
qed

context
begin

private lemma *asymp-equiv-add-rightI*:
assumes $f \sim[F]\ g\ h \in o[F](g)$
shows $(\lambda x.\ f\ x + h\ x) \sim[F]\ g$
proof −
let $?T = \lambda f\ g\ x.\ \mathit{if}\ f\ x = 0 \wedge g\ x = 0\ \mathit{then}\ 1\ \mathit{else}\ f\ x\ /\ g\ x$
from *landau-o.smallD*[*OF assms*(*2*) *zero-less-one*]
have *ev*: *eventually* $(\lambda x.\ g\ x = 0 \longrightarrow h\ x = 0)\ F$ **by** *eventually-elim auto*
have $(\lambda x.\ f\ x + h\ x) \sim[F]\ g \longleftrightarrow ((\lambda x.\ ?T\ f\ g\ x + h\ x\ /\ g\ x) \longrightarrow 1)\ F$
unfolding *asymp-equiv-def* **using** *ev*
by (*intro tendsto-cong*) (*auto elim*!: *eventually-mono simp*: *field-split-simps*)
also have $\ldots \longleftrightarrow ((\lambda x.\ ?T\ f\ g\ x + h\ x\ /\ g\ x) \longrightarrow 1 + 0)\ F$ **by** *simp*
also have $\ldots$ **by** (*intro tendsto-intros asymp-equivD assms smalloD-tendsto*)
finally show $(\lambda x.\ f\ x + h\ x) \sim[F]\ g$ **.**

qed

lemma *asymp-equiv-add-right* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $(\lambda x.\ f\ x + h\ x) \sim[F]\ g \longleftrightarrow f \sim[F]\ g$
proof
assume $(\lambda x.\ f\ x + h\ x) \sim[F]\ g$
from *asymp-equiv-add-rightI*[*OF this*, *of* $\lambda x.\ -h\ x$] *assms* **show** $f \sim[F]\ g$
by *simp*
qed (*simp-all add*: *asymp-equiv-add-rightI assms*)

end

lemma *asymp-equiv-add-left* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $(\lambda x.\ h\ x + f\ x) \sim[F]\ g \longleftrightarrow f \sim[F]\ g$
using *asymp-equiv-add-right*[*OF assms*] **by** (*simp add*: *add.commute*)

lemma *asymp-equiv-add-right'* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $g \sim[F]\ (\lambda x.\ f\ x + h\ x) \longleftrightarrow g \sim[F]\ f$
using *asymp-equiv-add-right*[*OF assms*] **by** (*simp add*: *asymp-equiv-sym*)

lemma *asymp-equiv-add-left'* [*asymp-equiv-simps*]:
assumes $h \in o[F](g)$
shows $g \sim[F]\ (\lambda x.\ h\ x + f\ x) \longleftrightarrow g \sim[F]\ f$
using *asymp-equiv-add-left*[*OF assms*] **by** (*simp add*: *asymp-equiv-sym*)

lemma *smallo-imp-asymp-equiv*:
assumes $(\lambda x.\ f\ x - g\ x) \in o[F](g)$
shows $f \sim[F]\ g$
proof −
from *assms* **have** $(\lambda x.\ f\ x - g\ x + g\ x) \sim[F]\ g$
by (*subst asymp-equiv-add-left*) *simp-all*
thus *?thesis* **by** *simp*
qed

lemma *asymp-equiv-uminus* [*asymp-equiv-intros*]:
$f \sim[F]\ g \Longrightarrow (\lambda x.\ -f\ x) \sim[F]\ (\lambda x.\ -g\ x)$
by (*simp add*: *asymp-equiv-def cong*: *if-cong*)

lemma *asymp-equiv-uminus-iff* [*asymp-equiv-simps*]:
$(\lambda x.\ -f\ x) \sim[F]\ g \longleftrightarrow f \sim[F]\ (\lambda x.\ -g\ x)$
by (*simp add*: *asymp-equiv-def cong*: *if-cong*)

lemma *asymp-equiv-mult* [*asymp-equiv-intros*]:
fixes *f1 f2 g1 g2* :: $'a \Rightarrow 'b$ :: *real-normed-field*
assumes *f1* $\sim[F]$ *g1 f2* $\sim[F]$ *g2*
shows $(\lambda x.\ f1\ x * f2\ x) \sim[F]\ (\lambda x.\ g1\ x * g2\ x)$

proof −
let *?T* = λ*f g x*. *if f x* = *0* ∧ *g x* = *0 then 1 else f x* / *g x*
let *?S* = λ*x*. (*if f1 x* = *0* ∧ *g1 x* = *0 then 1* − *?T f2 g2 x*
else if f2 x = *0* ∧ *g2 x* = *0 then 1* − *?T f1 g1 x else 0*)
let *?S′* = λ*x*. *?T* (λ*x*. *f1 x* ∗ *f2 x*) (λ*x*. *g1 x* ∗ *g2 x*) *x* − *?T f1 g1 x* ∗ *?T f2 g2 x*
have *A*: ((λ*x*. *1* − *?T f g x*) ⟶ *0*) *F* **if** *f* ∼[*F*] *g* **for** *f g* :: *'a* ⇒ *'b*
by (*rule tendsto-eq-intros refl asymp-equivD*[*OF that*])+ *simp-all*

from *assms* **have** ∗: ((λ*x*. *?T f1 g1 x* ∗ *?T f2 g2 x*) ⟶ *1* ∗ *1*) *F*
by (*intro tendsto-mult asymp-equivD*)
{
have (*?S* ⟶ *0*) *F*
by (*intro filterlim-If assms*[*THEN A*, *THEN tendsto-mono*[*rotated*]])
(*auto intro*: *le-infI1 le-infI2*)
moreover have *eventually* (λ*x*. *?S x* = *?S′ x*) *F*
using *assms*[*THEN asymp-equiv-eventually-zeros*] **by** *eventually-elim auto*
ultimately have (*?S′* ⟶ *0*) *F* **by** (*rule Lim-transform-eventually*)
}
with ∗ **have** (*?T* (λ*x*. *f1 x* ∗ *f2 x*) (λ*x*. *g1 x* ∗ *g2 x*) ⟶ *1* ∗ *1*) *F*
by (*rule Lim-transform*)
then show *?thesis* **by** (*simp add*: *asymp-equiv-def*)
qed

lemma *asymp-equiv-power* [*asymp-equiv-intros*]:
f ∼[*F*] *g* ⟹ (λ*x*. *f x* ^ *n*) ∼[*F*] (λ*x*. *g x* ^ *n*)
by (*induction n*) (*simp-all add*: *asymp-equiv-mult*)

lemma *asymp-equiv-inverse* [*asymp-equiv-intros*]:
assumes *f* ∼[*F*] *g*
shows (λ*x*. *inverse* (*f x*)) ∼[*F*] (λ*x*. *inverse* (*g x*))
proof −
from *tendsto-inverse*[*OF asymp-equivD*[*OF assms*]]
have ((λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else g x* / *f x*) ⟶ *1*) *F*
by (*simp add*: *if-distrib cong*: *if-cong*)
also have (λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else g x* / *f x*) =
(λ*x*. *if f x* = *0* ∧ *g x* = *0 then 1 else inverse* (*f x*) / *inverse* (*g x*))
by (*intro ext*) (*simp add*: *field-simps*)
finally show *?thesis* **by** (*simp add*: *asymp-equiv-def*)
qed

lemma *asymp-equiv-inverse-iff* [*asymp-equiv-simps*]:
(λ*x*. *inverse* (*f x*)) ∼[*F*] (λ*x*. *inverse* (*g x*)) ⟷ *f* ∼[*F*] *g*
proof
assume (λ*x*. *inverse* (*f x*)) ∼[*F*] (λ*x*. *inverse* (*g x*))
hence (λ*x*. *inverse* (*inverse* (*f x*))) ∼[*F*] (λ*x*. *inverse* (*inverse* (*g x*))) (**is** *?P*)
by (*rule asymp-equiv-inverse*)
also have *?P* ⟷ *f* ∼[*F*] *g* **by** (*intro asymp-equiv-cong*) *simp-all*
finally show *f* ∼[*F*] *g* **.**
qed (*simp-all add*: *asymp-equiv-inverse*)

lemma *asymp-equiv-divide* [*asymp-equiv-intros*]:
assumes *f1* $\sim[F]$ *g1 f2* $\sim[F]$ *g2*
shows $(\lambda x.\ f1\ x\ /\ f2\ x) \sim[F] (\lambda x.\ g1\ x\ /\ g2\ x)$
using *asymp-equiv-mult*[*OF assms*(*1*) *asymp-equiv-inverse*[*OF assms*(*2*)]] **by** (*simp add*: *field-simps*)

lemma *asymp-equivD-strong*:
assumes *f* $\sim[F]$ *g eventually* $(\lambda x.\ f\ x \neq 0 \vee g\ x \neq 0)\ F$
shows $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
proof −
from *assms*(*1*) **have** $((\lambda x.\ \textit{if}\ f\ x = 0 \wedge g\ x = 0\ \textit{then}\ 1\ \textit{else}\ f\ x\ /\ g\ x) \longrightarrow 1)\ F$
by (*rule asymp-equivD*)
also have *?this* $\longleftrightarrow$ *?thesis*
by (*intro filterlim-cong eventually-mono*[*OF assms*(*2*)]) *auto*
finally show *?thesis* **.**
qed

lemma *asymp-equiv-compose* [*asymp-equiv-intros*]:
assumes *f* $\sim[G]$ *g filterlim h G F*
shows $f \circ h \sim[F] g \circ h$
proof −
let *?T* $= \lambda f\ g\ x.\ \textit{if}\ f\ x = 0 \wedge g\ x = 0\ \textit{then}\ 1\ \textit{else}\ f\ x\ /\ g\ x$
have $f \circ h \sim[F] g \circ h \longleftrightarrow ((?T\ f\ g \circ h) \longrightarrow 1)\ F$
by (*simp add*: *asymp-equiv-def o-def*)
also have $\ldots \longleftrightarrow (?T\ f\ g \longrightarrow 1)$ (*filtermap h F*)
by (*rule tendsto-compose-filtermap*)
also have ...
by (*rule tendsto-mono*[*of - G*]) (*insert assms*, *simp-all add*: *asymp-equiv-def filterlim-def*)
finally show *?thesis* **.**
qed

lemma *asymp-equiv-compose′*:
assumes *f* $\sim[G]$ *g filterlim h G F*
shows $(\lambda x.\ f\ (h\ x)) \sim[F] (\lambda x.\ g\ (h\ x))$
using *asymp-equiv-compose*[*OF assms*] **by** (*simp add*: *o-def*)

lemma *asymp-equiv-powr-real* [*asymp-equiv-intros*]:
fixes *f g* :: *'a* $\Rightarrow$ *real*
assumes *f* $\sim[F]$ *g eventually* $(\lambda x.\ f\ x \geq 0)\ F$ *eventually* $(\lambda x.\ g\ x \geq 0)\ F$
shows $(\lambda x.\ f\ x\ \textit{powr}\ y) \sim[F] (\lambda x.\ g\ x\ \textit{powr}\ y)$
proof −
let *?T* $= \lambda f\ g\ x.\ \textit{if}\ f\ x = 0 \wedge g\ x = 0\ \textit{then}\ 1\ \textit{else}\ f\ x\ /\ g\ x$
have $((\lambda x.\ ?T\ f\ g\ x\ \textit{powr}\ y) \longrightarrow 1\ \textit{powr}\ y)\ F$
by (*intro tendsto-intros asymp-equivD*[*OF assms*(*1*)]) *simp-all*
hence $((\lambda x.\ ?T\ f\ g\ x\ \textit{powr}\ y) \longrightarrow 1)\ F$ **by** *simp*
moreover have *eventually* $(\lambda x.\ ?T\ f\ g\ x\ \textit{powr}\ y = ?T\ (\lambda x.\ f\ x\ \textit{powr}\ y)\ (\lambda x.\ g\ x\ \textit{powr}\ y)\ x)\ F$

using *asymp-equiv-eventually-zeros*[*OF assms*(*1*)] *assms*(*2*,*3*)
by *eventually-elim* (*auto simp*: *powr-divide*)
ultimately show *?thesis* **unfolding** *asymp-equiv-def* **by** (*rule Lim-transform-eventually*)
qed

lemma *asymp-equiv-norm* [*asymp-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow 'b$ :: *real-normed-field*
assumes $f \sim[F]\ g$
shows $(\lambda x.\ norm\ (f\ x)) \sim[F]\ (\lambda x.\ norm\ (g\ x))$
using *tendsto-norm*[*OF asymp-equivD*[*OF assms*]]
by (*simp add*: *if-distrib asymp-equiv-def norm-divide cong*: *if-cong*)

lemma *asymp-equiv-abs-real* [*asymp-equiv-intros*]:
fixes $f\ g :: 'a \Rightarrow real$
assumes $f \sim[F]\ g$
shows $(\lambda x.\ |f\ x|) \sim[F]\ (\lambda x.\ |g\ x|)$
using *tendsto-rabs*[*OF asymp-equivD*[*OF assms*]]
by (*simp add*: *if-distrib asymp-equiv-def cong*: *if-cong*)

lemma *asymp-equiv-imp-eventually-le*:
assumes $f \sim[F]\ g\ c > 1$
shows $eventually\ (\lambda x.\ norm\ (f\ x) \leq c * norm\ (g\ x))\ F$
proof −
from *order-tendstoD*(*2*)[*OF asymp-equivD*[*OF asymp-equiv-norm*[*OF assms*(*1*)]] *assms*(*2*)]
asymp-equiv-eventually-zeros[*OF assms*(*1*)]
show *?thesis* **by** *eventually-elim* (*auto split*: *if-splits simp*: *field-simps*)
qed

lemma *asymp-equiv-imp-eventually-ge*:
assumes $f \sim[F]\ g\ c < 1$
shows $eventually\ (\lambda x.\ norm\ (f\ x) \geq c * norm\ (g\ x))\ F$
proof −
from *order-tendstoD*(*1*)[*OF asymp-equivD*[*OF asymp-equiv-norm*[*OF assms*(*1*)]] *assms*(*2*)]
asymp-equiv-eventually-zeros[*OF assms*(*1*)]
show *?thesis* **by** *eventually-elim* (*auto split*: *if-splits simp*: *field-simps*)
qed

lemma *asymp-equiv-imp-bigo*:
assumes $f \sim[F]\ g$
shows $f \in O[F](g)$
proof (*rule bigoI*)
have $(3/2{::}real) > 1$ **by** *simp*
from *asymp-equiv-imp-eventually-le*[*OF assms this*]
show $eventually\ (\lambda x.\ norm\ (f\ x) \leq 3/2 * norm\ (g\ x))\ F$
by *eventually-elim simp*
qed

lemma *asymp-equiv-imp-bigomega*:
 $f \sim[F]\ g \Longrightarrow f \in \Omega[F](g)$
 using *asymp-equiv-imp-bigo*[*of g F f*] **by** (*simp add*: *asymp-equiv-sym bigomega-iff-bigo*)

lemma *asymp-equiv-imp-bigtheta*:
 $f \sim[F]\ g \Longrightarrow f \in \Theta[F](g)$
 by (*intro bigthetaI asymp-equiv-imp-bigo asymp-equiv-imp-bigomega*)

lemma *asymp-equiv-at-infinity-transfer*:
 assumes $f \sim[F]\ g$ *filterlim f at-infinity F*
 shows *filterlim g at-infinity F*
proof −
 from *assms*(*1*) **have** $g \in \Theta[F](f)$ **by** (*rule asymp-equiv-imp-bigtheta*[*OF asymp-equiv-symI*])
 also from *assms* **have** $f \in \omega[F](\lambda\text{-}.\ 1)$ **by** (*simp add*: *smallomega-1-conv-filterlim*)
 finally show *?thesis* **by** (*simp add*: *smallomega-1-conv-filterlim*)
qed

lemma *asymp-equiv-at-top-transfer*:
 fixes *f g* :: - ⇒ *real*
 assumes $f \sim[F]\ g$ *filterlim f at-top F*
 shows *filterlim g at-top F*
proof (*rule filterlim-at-infinity-imp-filterlim-at-top*)
 show *filterlim g at-infinity F*
 by (*rule asymp-equiv-at-infinity-transfer*[*OF assms*(*1*) *filterlim-mono*[*OF assms*(*2*)]])
 (*auto simp*: *at-top-le-at-infinity*)
 from *assms*(*2*) **have** *eventually* $(\lambda x.\ f\ x > 0)\ F$
 using *filterlim-at-top-dense* **by** *blast*
 with *asymp-equiv-eventually-pos-iff*[*OF assms*(*1*)] **show** *eventually* $(\lambda x.\ g\ x >$
0) *F*
 by *eventually-elim blast*
qed

lemma *asymp-equiv-at-bot-transfer*:
 fixes *f g* :: - ⇒ *real*
 assumes $f \sim[F]\ g$ *filterlim f at-bot F*
 shows *filterlim g at-bot F*
 unfolding *filterlim-uminus-at-bot*
 by (*rule asymp-equiv-at-top-transfer*[*of* $\lambda x.\ -f\ x\ F\ \lambda x.\ -g\ x$])
 (*insert assms*, *auto simp*: *filterlim-uminus-at-bot asymp-equiv-uminus*)

lemma *asymp-equivI′-const*:
 assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow c)\ F\ c \neq 0$
 shows $f \sim[F]\ (\lambda x.\ c * g\ x)$
 using *tendsto-mult*[*OF assms*(*1*) *tendsto-const*[*of inverse c*]] *assms*(*2*)
 by (*intro asymp-equivI′*) (*simp add*: *field-simps*)

lemma *asymp-equivI′-inverse-const*:
 assumes $((\lambda x.\ f\ x\ /\ g\ x) \longrightarrow inverse\ c)\ F\ c \neq 0$
 shows $(\lambda x.\ c * f\ x) \sim[F]\ g$

using *tendsto-mult*[*OF assms*(*1*) *tendsto-const*[*of c*]] *assms*(*2*)
by (*intro asymp-equivI'*) (*simp add*: *field-simps*)

lemma *filterlim-at-bot-imp-at-infinity*: *filterlim f at-bot F* $\Longrightarrow$ *filterlim f at-infinity F*
for *f* :: - $\Rightarrow$ *real* **using** *at-bot-le-at-infinity filterlim-mono* **by** *blast*

lemma *asymp-equiv-imp-diff-smallo*:
assumes $f \sim[F] g$
shows $(\lambda x.\ f\ x - g\ x) \in o[F](g)$
proof (*rule landau-o.smallI*)
fix *c* :: *real* **assume** $c > 0$
hence *c*: $min\ c\ 1 > 0$ **by** *simp*
let *?h* = $\lambda x.$ *if f x* = *0* $\wedge$ *g x* = *0 then 1 else f x* / *g x*
from *assms* **have** $((\lambda x.\ ?h\ x - 1) \longrightarrow 1 - 1)\ F$
by (*intro tendsto-diff asymp-equivD tendsto-const*)
from *tendstoD*[*OF this c*] **show** *eventually* $(\lambda x.\ norm\ (f\ x - g\ x) \leq c * norm\ (g\ x))\ F$
proof *eventually-elim*
case (*elim x*)
from *elim* **have** $norm\ (f\ x - g\ x) \leq norm\ (f\ x\ /\ g\ x - 1) * norm\ (g\ x)$
by (*subst norm-mult* [*symmetric*]) (*auto split*: *if-splits simp add*: *algebra-simps*)
also have $norm\ (f\ x\ /\ g\ x - 1) * norm\ (g\ x) \leq c * norm\ (g\ x)$ **using** *elim*
by (*auto split*: *if-splits simp*: *mult-right-mono*)
finally show *?case* **.**
qed
qed

lemma *asymp-equiv-altdef*:
$f \sim[F] g \longleftrightarrow (\lambda x.\ f\ x - g\ x) \in o[F](g)$
by (*rule iffI*[*OF asymp-equiv-imp-diff-smallo smallo-imp-asymp-equiv*])

lemma *asymp-equiv-0-left-iff* [*simp*]: $(\lambda\text{-}.\ 0) \sim[F] f \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
and *asymp-equiv-0-right-iff* [*simp*]: $f \sim[F] (\lambda\text{-}.\ 0) \longleftrightarrow$ *eventually* $(\lambda x.\ f\ x = 0)\ F$
by (*simp-all add*: *asymp-equiv-altdef landau-o.small-refl-iff*)

lemma *asymp-equiv-sandwich-real*:
fixes *f g l u* :: *'a* $\Rightarrow$ *real*
assumes $l \sim[F] g$ $u \sim[F] g$ *eventually* $(\lambda x.\ f\ x \in \{l\ x..u\ x\})\ F$
shows $f \sim[F] g$
unfolding *asymp-equiv-altdef*
proof (*rule landau-o.smallI*)
fix *c* :: *real* **assume** *c*: $c > 0$
have *eventually* $(\lambda x.\ norm\ (f\ x - g\ x) \leq max\ (norm\ (l\ x - g\ x))\ (norm\ (u\ x - g\ x)))\ F$
using *assms*(*3*) **by** *eventually-elim auto*
moreover have *eventually* $(\lambda x.\ norm\ (l\ x - g\ x) \leq c * norm\ (g\ x))\ F$

eventually (λ*x*. *norm* (*u x* − *g x*) ≤ *c* ∗ *norm* (*g x*)) *F*
using *assms*(*1*,*2*) **by** (*auto simp*: *asymp-equiv-altdef dest*: *landau-o.smallD*[*OF* - *c*])
hence *eventually* (λ*x*. *max* (*norm* (*l x* − *g x*)) (*norm* (*u x* − *g x*)) ≤ *c* ∗ *norm* (*g x*)) *F*
by *eventually-elim simp*
ultimately show *eventually* (λ*x*. *norm* (*f x* − *g x*) ≤ *c* ∗ *norm* (*g x*)) *F*
by *eventually-elim* (*rule order.trans*)
qed

lemma *asymp-equiv-sandwich-real'*:
fixes *f g l u* :: *'a* ⇒ *real*
assumes *f* ∼[*F*] *l f* ∼[*F*] *u eventually* (λ*x*. *g x* ∈ {*l x*..*u x*}) *F*
shows *f* ∼[*F*] *g*
using *asymp-equiv-sandwich-real*[*of l F f u g*] *assms* **by** (*simp add*: *asymp-equiv-sym*)

lemma *asymp-equiv-sandwich-real''*:
fixes *f g l u* :: *'a* ⇒ *real*
assumes *l1* ∼[*F*] *u1 u1* ∼[*F*] *l2 l2* ∼[*F*] *u2*
eventually (λ*x*. *f x* ∈ {*l1 x*..*u1 x*}) *F eventually* (λ*x*. *g x* ∈ {*l2 x*..*u2 x*}) *F*
shows *f* ∼[*F*] *g*
by (*meson assms asymp-equiv-sandwich-real asymp-equiv-sandwich-real' asymp-equiv-trans*)

end

56 Values extended by a bottom element

theory *Lattice-Constructions*
imports *Main*
begin

datatype *'a bot* = *Value 'a* | *Bot*

instantiation *bot* :: (*preorder*) *preorder*
begin

definition *less-eq-bot* **where**
x ≤ *y* ⟷ (*case x of Bot* ⇒ *True* | *Value x* ⇒ (*case y of Bot* ⇒ *False* | *Value y* ⇒ *x* ≤ *y*))

definition *less-bot* **where**
x < *y* ⟷ (*case y of Bot* ⇒ *False* | *Value y* ⇒ (*case x of Bot* ⇒ *True* | *Value x* ⇒ *x* < *y*))

lemma *less-eq-bot-Bot* [*simp*]: *Bot* ≤ *x*
by (*simp add*: *less-eq-bot-def*)

lemma *less-eq-bot-Bot-code* [*code*]: *Bot* ≤ *x* ⟷ *True*
by *simp*

lemma *less-eq-bot-Bot-is-Bot*: $x \leq Bot \Longrightarrow x = Bot$
by (*cases x*) (*simp-all add*: *less-eq-bot-def*)

lemma *less-eq-bot-Value-Bot* [*simp*, *code*]: $Value\ x \leq Bot \longleftrightarrow False$
by (*simp add*: *less-eq-bot-def*)

lemma *less-eq-bot-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
by (*simp add*: *less-eq-bot-def*)

lemma *less-bot-Bot* [*simp*, *code*]: $x < Bot \longleftrightarrow False$
by (*simp add*: *less-bot-def*)

lemma *less-bot-Bot-is-Value*: $Bot < x \Longrightarrow \exists z.\ x = Value\ z$
by (*cases x*) (*simp-all add*: *less-bot-def*)

lemma *less-bot-Bot-Value* [*simp*]: $Bot < Value\ x$
by (*simp add*: *less-bot-def*)

lemma *less-bot-Bot-Value-code* [*code*]: $Bot < Value\ x \longleftrightarrow True$
by *simp*

lemma *less-bot-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
by (*simp add*: *less-bot-def*)

instance
by *standard*
(*auto simp add*: *less-eq-bot-def less-bot-def less-le-not-le elim*: *order-trans split*: *bot.splits*)

end

instance *bot* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-bot-def less-bot-def split*: *bot.splits*)

instance *bot* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *less-eq-bot-def less-bot-def split*: *bot.splits*)

instantiation *bot* :: (*order*) *bot*
begin
definition *bot* = *Bot*
instance ..
end

instantiation *bot* :: (*top*) *top*
begin
definition *top* = *Value top*
instance ..
end

instantiation *bot* :: (*semilattice-inf*) *semilattice-inf*
begin

definition *inf-bot*
where
inf x y =
(*case x of*
Bot ⇒ *Bot*
| *Value v* ⇒
(*case y of*
Bot ⇒ *Bot*
| *Value v'* ⇒ *Value* (*inf v v'*)))

instance
by *standard* (*auto simp add*: *inf-bot-def less-eq-bot-def split*: *bot.splits*)

end

instantiation *bot* :: (*semilattice-sup*) *semilattice-sup*
begin

definition *sup-bot*
where
sup x y =
(*case x of*
Bot ⇒ *y*
| *Value v* ⇒
(*case y of*
Bot ⇒ *x*
| *Value v'* ⇒ *Value* (*sup v v'*)))

instance
by *standard* (*auto simp add*: *sup-bot-def less-eq-bot-def split*: *bot.splits*)

end

instance *bot* :: (*lattice*) *bounded-lattice-bot*
by *intro-classes* (*simp add*: *bot-bot-def*)

56.1 Values extended by a top element

datatype *'a top* = *Value 'a* | *Top*

instantiation *top* :: (*preorder*) *preorder*
begin

definition *less-eq-top* **where**
x ≤ *y* ⟷ (*case y of Top* ⇒ *True* | *Value y* ⇒ (*case x of Top* ⇒ *False* | *Value x*

$\Rightarrow x \leq y))$

definition *less-top* **where**
$x < y \longleftrightarrow (case\ x\ of\ Top \Rightarrow False \mid Value\ x \Rightarrow (case\ y\ of\ Top \Rightarrow True \mid Value\ y \Rightarrow x < y))$

lemma *less-eq-top-Top* [*simp*]: $x \leq Top$
by (*simp add*: *less-eq-top-def*)

lemma *less-eq-top-Top-code* [*code*]: $x \leq Top \longleftrightarrow True$
by *simp*

lemma *less-eq-top-is-Top*: $Top \leq x \Longrightarrow x = Top$
by (*cases x*) (*simp-all add*: *less-eq-top-def*)

lemma *less-eq-top-Top-Value* [*simp*, *code*]: $Top \leq Value\ x \longleftrightarrow False$
by (*simp add*: *less-eq-top-def*)

lemma *less-eq-top-Value-Value* [*simp*, *code*]: $Value\ x \leq Value\ y \longleftrightarrow x \leq y$
by (*simp add*: *less-eq-top-def*)

lemma *less-top-Top* [*simp*, *code*]: $Top < x \longleftrightarrow False$
by (*simp add*: *less-top-def*)

lemma *less-top-Top-is-Value*: $x < Top \Longrightarrow \exists z.\ x = Value\ z$
by (*cases x*) (*simp-all add*: *less-top-def*)

lemma *less-top-Value-Top* [*simp*]: $Value\ x < Top$
by (*simp add*: *less-top-def*)

lemma *less-top-Value-Top-code* [*code*]: $Value\ x < Top \longleftrightarrow True$
by *simp*

lemma *less-top-Value* [*simp*, *code*]: $Value\ x < Value\ y \longleftrightarrow x < y$
by (*simp add*: *less-top-def*)

instance
by *standard*
(*auto simp add*: *less-eq-top-def less-top-def less-le-not-le elim*: *order-trans split*: *top.splits*)

end

instance *top* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-top-def less-top-def split*: *top.splits*)

instance *top* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *less-eq-top-def less-top-def split*: *top.splits*)

```
instantiation top :: (order) top
begin
  definition top = Top
  instance ..
end

instantiation top :: (bot) bot
begin
  definition bot = Value bot
  instance ..
end

instantiation top :: (semilattice-inf) semilattice-inf
begin

definition inf-top
where
  inf x y =
    (case x of
      Top ⇒ y
    | Value v ⇒
        (case y of
          Top ⇒ x
        | Value v' ⇒ Value (inf v v')))

instance
  by standard (auto simp add: inf-top-def less-eq-top-def split: top.splits)

end

instantiation top :: (semilattice-sup) semilattice-sup
begin

definition sup-top
where
  sup x y =
    (case x of
      Top ⇒ Top
    | Value v ⇒
        (case y of
          Top ⇒ Top
        | Value v' ⇒ Value (sup v v')))

instance
  by standard (auto simp add: sup-top-def less-eq-top-def split: top.splits)

end

instance top :: (lattice) bounded-lattice-top
```

by *standard* (*simp add*: *top-top-def*)

56.2 Values extended by a top and a bottom element

datatype *'a flat-complete-lattice* = *Value 'a* | *Bot* | *Top*

instantiation *flat-complete-lattice* :: (*type*) *order*
begin

definition *less-eq-flat-complete-lattice*
where
x ≤ *y* ≡
(*case x of*
Bot ⇒ *True*
| *Value v1* ⇒
(*case y of*
Bot ⇒ *False*
| *Value v2* ⇒ *v1* = *v2*
| *Top* ⇒ *True*)
| *Top* ⇒ *y* = *Top*)

definition *less-flat-complete-lattice*
where
x < *y* =
(*case x of*
Bot ⇒ *y* ≠ *Bot*
| *Value v1* ⇒ *y* = *Top*
| *Top* ⇒ *False*)

lemma [*simp*]: *Bot* ≤ *y*
unfolding *less-eq-flat-complete-lattice-def* **by** *auto*

lemma [*simp*]: *y* ≤ *Top*
unfolding *less-eq-flat-complete-lattice-def* **by** (*auto split*: *flat-complete-lattice.splits*)

lemma *greater-than-two-values*:
assumes *a* ≠ *b Value a* ≤ *z Value b* ≤ *z*
shows *z* = *Top*
using *assms*
by (*cases z*) (*auto simp add*: *less-eq-flat-complete-lattice-def*)

lemma *lesser-than-two-values*:
assumes *a* ≠ *b z* ≤ *Value a z* ≤ *Value b*
shows *z* = *Bot*
using *assms*
by (*cases z*) (*auto simp add*: *less-eq-flat-complete-lattice-def*)

instance
by *standard*

```
  (auto simp add: less-eq-flat-complete-lattice-def less-flat-complete-lattice-def
    split: flat-complete-lattice.splits)

end

instantiation flat-complete-lattice :: (type) bot
begin
  definition bot = Bot
  instance ..
end

instantiation flat-complete-lattice :: (type) top
begin
  definition top = Top
  instance ..
end

instantiation flat-complete-lattice :: (type) lattice
begin

definition inf-flat-complete-lattice
where
  inf x y =
    (case x of
      Bot ⇒ Bot
    | Value v1 ⇒
        (case y of
          Bot ⇒ Bot
        | Value v2 ⇒ if v1 = v2 then x else Bot
        | Top ⇒ x)
    | Top ⇒ y)

definition sup-flat-complete-lattice
where
  sup x y =
    (case x of
      Bot ⇒ y
    | Value v1 ⇒
        (case y of
          Bot ⇒ x
        | Value v2 ⇒ if v1 = v2 then x else Top
        | Top ⇒ Top)
    | Top ⇒ Top)

instance
  by standard
    (auto simp add: inf-flat-complete-lattice-def sup-flat-complete-lattice-def
      less-eq-flat-complete-lattice-def split: flat-complete-lattice.splits)
```

end

instantiation *flat-complete-lattice* :: (*type*) *complete-lattice*
begin

definition *Sup-flat-complete-lattice*
where
Sup A =
(*if A* = {} ∨ *A* = {*Bot*} *then Bot*
else if ∃ *v*. *A* − {*Bot*} = {*Value v*} *then Value* (*THE v*. *A* − {*Bot*} = {*Value v*})
else Top)

definition *Inf-flat-complete-lattice*
where
Inf A =
(*if A* = {} ∨ *A* = {*Top*} *then Top*
else if ∃ *v*. *A* − {*Top*} = {*Value v*} *then Value* (*THE v*. *A* − {*Top*} = {*Value v*})
else Bot)

instance
proof
fix *x* :: *'a flat-complete-lattice*
fix *A*
assume *x* ∈ *A*
{
fix *v*
assume *A* − {*Top*} = {*Value v*}
then have (*THE v*. *A* − {*Top*} = {*Value v*}) = *v*
by (*auto intro*!: *the1-equality*)
moreover
from ‹*x* ∈ *A*› ‹*A* − {*Top*} = {*Value v*}› **have** *x* = *Top* ∨ *x* = *Value v*
by *auto*
ultimately have *Value* (*THE v*. *A* − {*Top*} = {*Value v*}) ≤ *x*
by *auto*
}
with ‹*x* ∈ *A*› **show** *Inf A* ≤ *x*
unfolding *Inf-flat-complete-lattice-def*
by *fastforce*
next
fix *z* :: *'a flat-complete-lattice*
fix *A*
show *z* ≤ *Inf A* **if** *z*: ⋀*x*. *x* ∈ *A* ⟹ *z* ≤ *x*
proof −
consider *A* = {} ∨ *A* = {*Top*}
| *A* ≠ {} *A* ≠ {*Top*} ∃ *v*. *A* − {*Top*} = {*Value v*}
| *A* ≠ {} *A* ≠ {*Top*} ¬ (∃ *v*. *A* − {*Top*} = {*Value v*})
by *blast*

```
    then show ?thesis
    proof cases
      case 1
      then have Inf A = Top
        unfolding Inf-flat-complete-lattice-def by auto
      then show ?thesis by simp
    next
      case 2
      then obtain v where v1: A − {Top} = {Value v}
        by auto
      then have v2: (THE v. A − {Top} = {Value v}) = v
        by (auto intro!: the1-equality)
      from 2 v2 have Inf: Inf A = Value v
        unfolding Inf-flat-complete-lattice-def by simp
      from v1 have Value v ∈ A by blast
      then have z ≤ Value v by (rule z)
      with Inf show ?thesis by simp
    next
      case 3
      then have Inf: Inf A = Bot
        unfolding Inf-flat-complete-lattice-def by auto
      have z ≤ Bot
      proof (cases A − {Top} = {Bot})
        case True
        then have Bot ∈ A by blast
        then show ?thesis by (rule z)
      next
        case False
        from 3 obtain a1 where a1: a1 ∈ A − {Top}
          by auto
        from 3 False a1 obtain a2 where a2 ∈ A − {Top} ∧ a1 ≠ a2
          by (cases a1) auto
        with a1 z[of a1] z[of a2] show ?thesis
          apply (cases a1)
          apply auto
          apply (cases a2)
          apply auto
          apply (auto dest!: lesser-than-two-values)
          done
      qed
      with Inf show ?thesis by simp
    qed
  qed
next
  fix x :: 'a flat-complete-lattice
  fix A
  assume x ∈ A
  {
    fix v
```

```
    assume A − {Bot} = {Value v}
    then have (THE v. A − {Bot} = {Value v}) = v
      by (auto intro!: the1-equality)
    moreover
    from ‹x ∈ A› ‹A − {Bot} = {Value v}› have x = Bot ∨ x = Value v
      by auto
    ultimately have x ≤ Value (THE v. A − {Bot} = {Value v})
      by auto
  }
  with ‹x ∈ A› show x ≤ Sup A
    unfolding Sup-flat-complete-lattice-def
    by fastforce
next
  fix z :: 'a flat-complete-lattice
  fix A
  show Sup A ≤ z if z: ⋀x. x ∈ A ⟹ x ≤ z
  proof −
    consider A = {} ∨ A = {Bot}
      | A ≠ {} A ≠ {Bot} ∃ v. A − {Bot} = {Value v}
      | A ≠ {} A ≠ {Bot} ¬ (∃ v. A − {Bot} = {Value v})
      by blast
    then show ?thesis
    proof cases
      case 1
      then have Sup A = Bot
        unfolding Sup-flat-complete-lattice-def by auto
      then show ?thesis by simp
    next
      case 2
      then obtain v where v1: A − {Bot} = {Value v}
        by auto
      then have v2: (THE v. A − {Bot} = {Value v}) = v
        by (auto intro!: the1-equality)
      from 2 v2 have Sup: Sup A = Value v
        unfolding Sup-flat-complete-lattice-def by simp
      from v1 have Value v ∈ A by blast
      then have Value v ≤ z by (rule z)
      with Sup show ?thesis by simp
    next
      case 3
      then have Sup: Sup A = Top
        unfolding Sup-flat-complete-lattice-def by auto
      have Top ≤ z
      proof (cases A − {Bot} = {Top})
        case True
        then have Top ∈ A by blast
        then show ?thesis by (rule z)
      next
        case False
```

```
      from 3 obtain a1 where a1: a1 ∈ A − {Bot}
        by auto
      from 3 False a1 obtain a2 where a2 ∈ A − {Bot} ∧ a1 ≠ a2
        by (cases a1) auto
      with a1 z[of a1] z[of a2] show ?thesis
        apply (cases a1)
        apply auto
        apply (cases a2)
        apply (auto dest!: greater-than-two-values)
        done
    qed
    with Sup show ?thesis by simp
  qed
 qed
next
 show Inf {} = (top :: 'a flat-complete-lattice)
  by (simp add: Inf-flat-complete-lattice-def top-flat-complete-lattice-def)
 show Sup {} = (bot :: 'a flat-complete-lattice)
  by (simp add: Sup-flat-complete-lattice-def bot-flat-complete-lattice-def)
qed

end

end
```

57 Infinite Streams

```
theory Stream
 imports Nat-Bijection
begin

codatatype (sset: 'a) stream =
 SCons (shd: 'a) (stl: 'a stream) (infixr ‹##› 65)
for
 map: smap
 rel: stream-all2

context
begin

— for code generation only
qualified definition smember :: 'a ⇒ 'a stream ⇒ bool where
 [code-abbrev]: smember x s ⟷ x ∈ sset s

lemma smember-code[code, simp]: smember x (y ## s) = (if x = y then True else
smember x s)
 unfolding smember-def by auto

end
```

lemmas *smap-simps*[*simp*] = *stream.map-sel*
lemmas *shd-sset* = *stream.set-sel*(*1*)
lemmas *stl-sset* = *stream.set-sel*(*2*)

theorem *sset-induct*[*consumes 1*, *case-names shd stl*, *induct set*: *sset*]:
  assumes $y \in sset\ s$ **and** $\bigwedge s.\ P\ (shd\ s)\ s$ **and** $\bigwedge s\ y.\ [\![y \in sset\ (stl\ s);\ P\ y\ (stl\ s)]\!] \Longrightarrow P\ y\ s$
  shows $P\ y\ s$
using *assms* **by** *induct* (*metis stream.sel*(*1*), *auto*)

lemma *smap-ctr*: $smap\ f\ s = x\ \#\#\ s' \longleftrightarrow f\ (shd\ s) = x \wedge smap\ f\ (stl\ s) = s'$
  by (*cases s*) *simp*

57.1 prepend list to stream

primrec *shift* :: *'a list* ⇒ *'a stream* ⇒ *'a stream* (**infixr** ‹@−› *65*) **where**
  shift [] *s* = *s*
| *shift* (*x* # *xs*) *s* = *x* ## *shift xs s*

lemma *smap-shift*[*simp*]: *smap f* (*xs* @− *s*) = *map f xs* @− *smap f s*
  by (*induct xs*) *auto*

lemma *shift-append*[*simp*]: (*xs* @ *ys*) @− *s* = *xs* @− *ys* @− *s*
  by (*induct xs*) *auto*

lemma *shift-simps*[*simp*]:
   shd (*xs* @− *s*) = (*if xs* = [] *then shd s else hd xs*)
   stl (*xs* @− *s*) = (*if xs* = [] *then stl s else tl xs* @− *s*)
  by (*induct xs*) *auto*

lemma *sset-shift*[*simp*]: *sset* (*xs* @− *s*) = *set xs* ∪ *sset s*
  by (*induct xs*) *auto*

lemma *shift-left-inj*[*simp*]: *xs* @− *s1* = *xs* @− *s2* ⟷ *s1* = *s2*
  by (*induct xs*) *auto*

57.2 set of streams with elements in some fixed set

context
  notes [[*inductive-internals*]]
begin

coinductive-set
  streams :: *'a set* ⇒ *'a stream set*
  for *A* :: *'a set*
where
  Stream[*intro*!, *simp*, *no-atp*]: $[\![a \in A;\ s \in streams\ A]\!] \Longrightarrow a\ \#\#\ s \in streams\ A$

end

lemma *in-streams*: *stl s* ∈ *streams S* ⟹ *shd s* ∈ *S* ⟹ *s* ∈ *streams S*
by (*cases s*) *auto*

lemma *streamsE*: *s* ∈ *streams A* ⟹ (*shd s* ∈ *A* ⟹ *stl s* ∈ *streams A* ⟹ *P*) ⟹ *P*
by (*erule streams.cases*) *simp-all*

lemma *Stream-image*: *x* ## *y* ∈ ((##) *x'*) ‘ *Y* ⟷ *x* = *x'* ∧ *y* ∈ *Y*
by *auto*

lemma *shift-streams*: ⟦*w* ∈ *lists A*; *s* ∈ *streams A*⟧ ⟹ *w* @− *s* ∈ *streams A*
by (*induct w*) *auto*

lemma *streams-Stream*: *x* ## *s* ∈ *streams A* ⟷ *x* ∈ *A* ∧ *s* ∈ *streams A*
by (*auto elim*: *streams.cases*)

lemma *streams-stl*: *s* ∈ *streams A* ⟹ *stl s* ∈ *streams A*
by (*cases s*) (*auto simp*: *streams-Stream*)

lemma *streams-shd*: *s* ∈ *streams A* ⟹ *shd s* ∈ *A*
by (*cases s*) (*auto simp*: *streams-Stream*)

lemma *sset-streams*:
assumes *sset s* ⊆ *A*
shows *s* ∈ *streams A*
using *assms* **proof** (*coinduction arbitrary*: *s*)
case *streams* **then show** *?case* **by** (*cases s*) *simp*
qed

lemma *streams-sset*:
assumes *s* ∈ *streams A*
shows *sset s* ⊆ *A*
proof
fix *x* **assume** *x* ∈ *sset s* **from** *this* ‹*s* ∈ *streams A*› **show** *x* ∈ *A*
by (*induct s*) (*auto intro*: *streams-shd streams-stl*)
qed

lemma *streams-iff-sset*: *s* ∈ *streams A* ⟷ *sset s* ⊆ *A*
by (*metis sset-streams streams-sset*)

lemma *streams-mono*: *s* ∈ *streams A* ⟹ *A* ⊆ *B* ⟹ *s* ∈ *streams B*
unfolding *streams-iff-sset* **by** *auto*

lemma *streams-mono2*: *S* ⊆ *T* ⟹ *streams S* ⊆ *streams T*
by (*auto intro*: *streams-mono*)

lemma *smap-streams*: *s* ∈ *streams A* ⟹ (⋀*x*. *x* ∈ *A* ⟹ *f x* ∈ *B*) ⟹ *smap f s* ∈ *streams B*

unfolding *streams-iff-sset stream.set-map* **by** *auto*

lemma *streams-empty*: *streams* {} = {}
by (*auto elim*: *streams.cases*)

lemma *streams-UNIV*[*simp*]: *streams UNIV* = *UNIV*
by (*auto simp*: *streams-iff-sset*)

57.3 nth, take, drop for streams

primrec *snth* :: *'a stream* ⇒ *nat* ⇒ *'a* (**infixl** ‹!!› *100*) **where**
s !! *0* = *shd s*
| *s* !! *Suc n* = *stl s* !! *n*

lemma *snth-Stream*: (*x* ## *s*) !! *Suc i* = *s* !! *i*
by *simp*

lemma *snth-smap*[*simp*]: *smap f s* !! *n* = *f* (*s* !! *n*)
by (*induct n arbitrary*: *s*) *auto*

lemma *shift-snth-less*[*simp*]: *p* < *length xs* ⟹ (*xs* @− *s*) !! *p* = *xs* ! *p*
by (*induct p arbitrary*: *xs*) (*auto simp*: *hd-conv-nth nth-tl*)

lemma *shift-snth-ge*[*simp*]: *p* ≥ *length xs* ⟹ (*xs* @− *s*) !! *p* = *s* !! (*p* − *length xs*)
by (*induct p arbitrary*: *xs*) (*auto simp*: *Suc-diff-eq-diff-pred*)

lemma *shift-snth*: (*xs* @− *s*) !! *n* = (*if n* < *length xs then xs* ! *n else s* !! (*n* − *length xs*))
by *auto*

lemma *snth-sset*[*simp*]: *s* !! *n* ∈ *sset s*
by (*induct n arbitrary*: *s*) (*auto intro*: *shd-sset stl-sset*)

lemma *sset-range*: *sset s* = *range* (*snth s*)
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* ∈ *sset s*
thus *x* ∈ *range* (*snth s*)
proof (*induct s*)
case (*stl s x*)
then obtain *n* **where** *x* = *stl s* !! *n* **by** *auto*
thus *?case* **by** (*auto intro*: *range-eqI*[*of - - Suc n*])
qed (*auto intro*: *range-eqI*[*of - - 0*])
qed *auto*

lemma *streams-iff-snth*: *s* ∈ *streams X* ⟷ (∀ *n*. *s* !! *n* ∈ *X*)
by (*force simp*: *streams-iff-sset sset-range*)

lemma *snth-in*: *s* ∈ *streams X* ⟹ *s* !! *n* ∈ *X*
by (*simp add*: *streams-iff-snth*)

primrec *stake* :: *nat* ⇒ *'a stream* ⇒ *'a list* **where**
  stake 0 s = []
| *stake* (*Suc n*) *s* = *shd s* # *stake n* (*stl s*)

lemma *length-stake*[*simp*]: *length* (*stake n s*) = *n*
  by (*induct n arbitrary*: *s*) *auto*

lemma *stake-smap*[*simp*]: *stake n* (*smap f s*) = *map f* (*stake n s*)
  by (*induct n arbitrary*: *s*) *auto*

lemma *take-stake*: *take n* (*stake m s*) = *stake* (*min n m*) *s*
proof (*induct m arbitrary*: *s n*)
  case (*Suc m*) **thus** *?case* **by** (*cases n*) *auto*
qed *simp*

primrec *sdrop* :: *nat* ⇒ *'a stream* ⇒ *'a stream* **where**
  sdrop 0 s = *s*
| *sdrop* (*Suc n*) *s* = *sdrop n* (*stl s*)

lemma *sdrop-simps*[*simp*]:
  shd (*sdrop n s*) = *s* !! *n stl* (*sdrop n s*) = *sdrop* (*Suc n*) *s*
  by (*induct n arbitrary*: *s*) *auto*

lemma *sdrop-smap*[*simp*]: *sdrop n* (*smap f s*) = *smap f* (*sdrop n s*)
  by (*induct n arbitrary*: *s*) *auto*

lemma *sdrop-stl*: *sdrop n* (*stl s*) = *stl* (*sdrop n s*)
  by (*induct n*) *auto*

lemma *drop-stake*: *drop n* (*stake m s*) = *stake* (*m* − *n*) (*sdrop n s*)
proof (*induct m arbitrary*: *s n*)
  case (*Suc m*) **thus** *?case* **by** (*cases n*) *auto*
qed *simp*

lemma *stake-sdrop*: *stake n s* @− *sdrop n s* = *s*
  by (*induct n arbitrary*: *s*) *auto*

lemma *id-stake-snth-sdrop*:
  s = *stake i s* @− *s* !! *i* ## *sdrop* (*Suc i*) *s*
  by (*subst stake-sdrop*[*symmetric*, *of - i*]) (*metis sdrop-simps stream.collapse*)

lemma *smap-alt*: *smap f s* = *s'* ⟷ (∀ *n*. *f* (*s* !! *n*) = *s'* !! *n*) (**is** *?L* = *?R*)
proof
  assume *?R*
  then have ⋀*n*. *smap f* (*sdrop n s*) = *sdrop n s'*
    by *coinduction* (*auto intro*: *exI*[*of - 0*] *simp del*: *sdrop.simps*(*2*))
  then show *?L* **using** *sdrop.simps*(*1*) **by** *metis*
qed *auto*

lemma *stake-invert-Nil*[*iff*]: *stake n s* = [] ⟷ *n* = *0*
by (*induct n*) *auto*

lemma *sdrop-shift*: *sdrop i* (*w* @− *s*) = *drop i w* @− *sdrop* (*i* − *length w*) *s*
by (*induct i arbitrary*: *w s*) (*auto simp*: *drop-tl drop-Suc neq-Nil-conv*)

lemma *stake-shift*: *stake i* (*w* @− *s*) = *take i w* @ *stake* (*i* − *length w*) *s*
by (*induct i arbitrary*: *w s*) (*auto simp*: *neq-Nil-conv*)

lemma *stake-add*[*simp*]: *stake m s* @ *stake n* (*sdrop m s*) = *stake* (*m* + *n*) *s*
by (*induct m arbitrary*: *s*) *auto*

lemma *sdrop-add*[*simp*]: *sdrop n* (*sdrop m s*) = *sdrop* (*m* + *n*) *s*
by (*induct m arbitrary*: *s*) *auto*

lemma *sdrop-snth*: *sdrop n s* !! *m* = *s* !! (*n* + *m*)
by (*induct n arbitrary*: *m s*) *auto*

partial-function (*tailrec*) *sdrop-while* :: ('*a* ⇒ *bool*) ⇒ '*a stream* ⇒ '*a stream*
where
sdrop-while P s = (*if P* (*shd s*) *then sdrop-while P* (*stl s*) *else s*)

lemma *sdrop-while-SCons*[*code*]:
sdrop-while P (*a* ## *s*) = (*if P a then sdrop-while P s else a* ## *s*)
by (*subst sdrop-while.simps*) *simp*

lemma *sdrop-while-sdrop-LEAST*:
assumes ∃ *n*. *P* (*s* !! *n*)
shows *sdrop-while* (*Not* ∘ *P*) *s* = *sdrop* (*LEAST n*. *P* (*s* !! *n*)) *s*
proof −
from *assms* **obtain** *m* **where** *P* (*s* !! *m*) ⋀*n*. *P* (*s* !! *n*) ⟹ *m* ≤ *n*
and ∗: (*LEAST n*. *P* (*s* !! *n*)) = *m* **by** *atomize-elim* (*auto intro*: *LeastI Least-le*)
thus *?thesis* **unfolding** ∗
proof (*induct m arbitrary*: *s*)
case (*Suc m*)
hence *sdrop-while* (*Not* ∘ *P*) (*stl s*) = *sdrop m* (*stl s*)
by (*metis* (*full-types*) *not-less-eq-eq snth.simps*(*2*))
moreover from *Suc*(*3*) **have** ¬ (*P* (*s* !! *0*)) **by** *blast*
ultimately show *?case* **by** (*subst sdrop-while.simps*) *simp*
qed (*metis comp-apply sdrop.simps*(*1*) *sdrop-while.simps snth.simps*(*1*))
qed

primcorec *sfilter* **where**
shd (*sfilter P s*) = *shd* (*sdrop-while* (*Not* ∘ *P*) *s*)
| *stl* (*sfilter P s*) = *sfilter P* (*stl* (*sdrop-while* (*Not* ∘ *P*) *s*))

lemma *sfilter-Stream*: *sfilter P* (*x* ## *s*) = (*if P x then x* ## *sfilter P s else sfilter P s*)

proof (*cases P x*)
case *True* **thus** *?thesis* **by** (*subst sfilter.ctr*) (*simp add*: *sdrop-while-SCons*)
next
case *False* **thus** *?thesis* **by** (*subst* (*1 2*) *sfilter.ctr*) (*simp add*: *sdrop-while-SCons*)
qed

57.4 unary predicates lifted to streams

definition *stream-all P s* = (∀ *p*. *P* (*s* !! *p*))

lemma *stream-all-iff*[*iff*]: *stream-all P s* ⟷ *Ball* (*sset s*) *P*
unfolding *stream-all-def sset-range* **by** *auto*

lemma *stream-all-shift*[*simp*]: *stream-all P* (*xs* @− *s*) = (*list-all P xs* ∧ *stream-all P s*)
unfolding *stream-all-iff list-all-iff* **by** *auto*

lemma *stream-all-Stream*: *stream-all P* (*x* ## *X*) ⟷ *P x* ∧ *stream-all P X*
by *simp*

57.5 recurring stream out of a list

primcorec *cycle* :: *'a list* ⇒ *'a stream* **where**
shd (*cycle xs*) = *hd xs*
| *stl* (*cycle xs*) = *cycle* (*tl xs* @ [*hd xs*])

lemma *cycle-decomp*: *u* ≠ [] ⟹ *cycle u* = *u* @− *cycle u*
proof (*coinduction arbitrary*: *u*)
case *Eq-stream* **then show** *?case* **using** *stream.collapse*[*of cycle u*]
by (*auto intro*!: *exI*[*of* - *tl u* @ [*hd u*]])
qed

lemma *cycle-Cons*[*code*]: *cycle* (*x* # *xs*) = *x* ## *cycle* (*xs* @ [*x*])
by (*subst cycle.ctr*) *simp*

lemma *cycle-rotated*: ⟦*v* ≠ []; *cycle u* = *v* @− *s*⟧ ⟹ *cycle* (*tl u* @ [*hd u*]) = *tl v* @− *s*
by (*auto dest*: *arg-cong*[*of* - - *stl*])

lemma *stake-append*: *stake n* (*u* @− *s*) = *take* (*min* (*length u*) *n*) *u* @ *stake* (*n* − *length u*) *s*
proof (*induct n arbitrary*: *u*)
case (*Suc n*) **thus** *?case* **by** (*cases u*) *auto*
qed *auto*

lemma *stake-cycle-le*[*simp*]:
assumes *u* ≠ [] *n* < *length u*
shows *stake n* (*cycle u*) = *take n u*
using *min-absorb2*[*OF less-imp-le-nat*[*OF assms*(*2*)]]
by (*subst cycle-decomp*[*OF assms*(*1*)], *subst stake-append*) *auto*

lemma *stake-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow$ *stake* (*length u*) (*cycle u*) = *u*
by (*subst cycle-decomp*) (*auto simp*: *stake-shift*)

lemma *sdrop-cycle-eq*[*simp*]: $u \neq [] \Longrightarrow$ *sdrop* (*length u*) (*cycle u*) = *cycle u*
by (*subst cycle-decomp*) (*auto simp*: *sdrop-shift*)

lemma *stake-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n \bmod length\ u = 0]\!] \Longrightarrow$
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*)
by (*induct n div length u arbitrary*: *n u*)
(*auto simp*: *stake-add* [*symmetric*] *mod-eq-0-iff-dvd elim*!: *dvdE*)

lemma *sdrop-cycle-eq-mod-0*[*simp*]: $[\![u \neq []; n \bmod length\ u = 0]\!] \Longrightarrow$
sdrop n (*cycle u*) = *cycle u*
by (*induct n div length u arbitrary*: *n u*)
(*auto simp*: *sdrop-add* [*symmetric*] *mod-eq-0-iff-dvd elim*!: *dvdE*)

lemma *stake-cycle*: $u \neq [] \Longrightarrow$
stake n (*cycle u*) = *concat* (*replicate* (*n div length u*) *u*) @ *take* (*n mod length u*) *u*
by (*subst div-mult-mod-eq*[*of n length u*, *symmetric*], *unfold stake-add*[*symmetric*]) *auto*

lemma *sdrop-cycle*: $u \neq [] \Longrightarrow$ *sdrop n* (*cycle u*) = *cycle* (*rotate* (*n mod length u*) *u*)
by (*induct n arbitrary*: *u*) (*auto simp*: *rotate1-rotate-swap rotate1-hd-tl rotate-conv-mod*[*symmetric*])

lemma *sset-cycle*[*simp*]:
assumes *xs* $\neq$ []
shows *sset* (*cycle xs*) = *set xs*
proof (*intro set-eqI iffI*)
fix *x*
assume *x* $\in$ *sset* (*cycle xs*)
then show *x* $\in$ *set xs* **using** *assms*
by (*induction cycle xs arbitrary*: *xs rule*: *sset-induct*) (*fastforce simp*: *neq-Nil-conv*)+
qed (*metis assms UnI1 cycle-decomp sset-shift*)

57.6 iterated application of a function

primcorec *siterate* **where**
shd (*siterate f x*) = *x*
| *stl* (*siterate f x*) = *siterate f* (*f x*)

lemma *stake-Suc*: *stake* (*Suc n*) *s* = *stake n s* @ [*s* !! *n*]
by (*induct n arbitrary*: *s*) *auto*

lemma *snth-siterate*[*simp*]: *siterate f x* !! *n* = (*f*^^*n*) *x*
by (*induct n arbitrary*: *x*) (*auto simp*: *funpow-swap1*)

lemma *sdrop-siterate*[*simp*]: *sdrop n* (*siterate f x*) = *siterate f* ((*f*^^*n*) *x*)
by (*induct n arbitrary*: *x*) (*auto simp*: *funpow-swap1*)

lemma *stake-siterate*[*simp*]: *stake n* (*siterate f x*) = *map* (λ*n*. (*f*^^*n*) *x*) [*0* ..< *n*]
by (*induct n arbitrary*: *x*) (*auto simp del*: *stake.simps*(*2*) *simp*: *stake-Suc*)

lemma *sset-siterate*: *sset* (*siterate f x*) = {(*f*^^*n*) *x* | *n*. *True*}
by (*auto simp*: *sset-range*)

lemma *smap-siterate*: *smap f* (*siterate f x*) = *siterate f* (*f x*)
by (*coinduction arbitrary*: *x*) *auto*

57.7 stream repeating a single element

abbreviation *sconst* ≡ *siterate id*

lemma *shift-replicate-sconst*[*simp*]: *replicate n x* @− *sconst x* = *sconst x*
by (*subst* (*3*) *stake-sdrop*[*symmetric*]) (*simp add*: *map-replicate-trivial*)

lemma *sset-sconst*[*simp*]: *sset* (*sconst x*) = {*x*}
by (*simp add*: *sset-siterate*)

lemma *sconst-alt*: *s* = *sconst x* ⟷ *sset s* = {*x*}
proof
assume *sset s* = {*x*}
then show *s* = *sconst x*
proof (*coinduction arbitrary*: *s*)
case *Eq-stream*
then have *shd s* = *x sset* (*stl s*) ⊆ {*x*} **by** (*cases s*; *auto*)+
then have *sset* (*stl s*) = {*x*} **by** (*cases stl s*) *auto*
with ‹*shd s* = *x*› **show** *?case* **by** *auto*
qed
qed *simp*

lemma *sconst-cycle*: *sconst x* = *cycle* [*x*]
by *coinduction auto*

lemma *smap-sconst*: *smap f* (*sconst x*) = *sconst* (*f x*)
by *coinduction auto*

lemma *sconst-streams*: *x* ∈ *A* ⟹ *sconst x* ∈ *streams A*
by (*simp add*: *streams-iff-sset*)

lemma *streams-empty-iff*: *streams S* = {} ⟷ *S* = {}
proof *safe*
fix *x* **assume** *x* ∈ *S streams S* = {}
then have *sconst x* ∈ *streams S*
by (*intro sconst-streams*)
then show *x* ∈ {}

unfolding ‹*streams S = {}*› **by** *simp*
qed (*auto simp*: *streams-empty*)

57.8 stream of natural numbers

abbreviation *fromN* ≡ *siterate Suc*

abbreviation *nats* ≡ *fromN 0*

lemma *sset-fromN*[*simp*]: *sset* (*fromN n*) = {*n* ..}
by (*auto simp add*: *sset-siterate le-iff-add*)

lemma *stream-smap-fromN*: *s* = *smap* (λ*j*. *let i* = *j* − *n in s* !! *i*) (*fromN n*)
by (*coinduction arbitrary*: *s n*)
(*force simp*: *neq-Nil-conv Let-def Suc-diff-Suc simp flip*: *snth.simps*(*2*)
intro: *stream.map-cong split*: *if-splits*)

lemma *stream-smap-nats*: *s* = *smap* (*snth s*) *nats*
using *stream-smap-fromN*[**where** *n* = *0*] **by** *simp*

57.9 flatten a stream of lists

primcorec *flat* **where**
shd (*flat ws*) = *hd* (*shd ws*)
| *stl* (*flat ws*) = *flat* (*if tl* (*shd ws*) = [] *then stl ws else tl* (*shd ws*) ## *stl ws*)

lemma *flat-Cons*[*simp*, *code*]: *flat* ((*x* # *xs*) ## *ws*) = *x* ## *flat* (*if xs* = [] *then ws else xs* ## *ws*)
by (*subst flat.ctr*) *simp*

lemma *flat-Stream*[*simp*]: *xs* ≠ [] ⟹ *flat* (*xs* ## *ws*) = *xs* @− *flat ws*
by (*induct xs*) *auto*

lemma *flat-unfold*: *shd ws* ≠ [] ⟹ *flat ws* = *shd ws* @− *flat* (*stl ws*)
by (*cases ws*) *auto*

lemma *flat-snth*: ∀ *xs* ∈ *sset s*. *xs* ≠ [] ⟹ *flat s* !! *n* = (*if n* < *length* (*shd s*) *then*
shd s ! *n else flat* (*stl s*) !! (*n* − *length* (*shd s*)))
by (*metis flat-unfold not-less shd-sset shift-snth-ge shift-snth-less*)

lemma *sset-flat*[*simp*]: ∀ *xs* ∈ *sset s*. *xs* ≠ [] ⟹
sset (*flat s*) = (⋃ *xs* ∈ *sset s*. *set xs*) (**is** *?P* ⟹ *?L* = *?R*)
proof *safe*
fix *x* **assume** *?P x* ∈ *?L*
then obtain *m* **where** *x* = *flat s* !! *m* **by** (*metis image-iff sset-range*)
with ‹*?P*› **obtain** *n m'* **where** *x* = *s* !! *n* ! *m' m'* < *length* (*s* !! *n*)
proof (*atomize-elim*, *induct m arbitrary*: *s rule*: *less-induct*)
case (*less y*)
thus *?case*
proof (*cases y* < *length* (*shd s*))

case *True* **thus** *?thesis* **by** (*metis flat-snth less(2,3) snth.simps(1)*)
next
case *False*
hence $x = flat\ (stl\ s)\ !!\ (y - length\ (shd\ s))$ **by** (*metis less(2,3) flat-snth*)
moreover have $y - length\ (shd\ s) < y$
proof −
from *less(2)* **have** $*$: $length\ (shd\ s) > 0$ **by** (*cases s*) *simp-all*
with *False* **have** $y > 0$ **by** (*cases y*) *simp-all*
with $*$ **show** *?thesis* **by** *simp*
qed
moreover have $\forall xs \in sset\ (stl\ s).\ xs \neq []$ **using** *less(2)* **by** (*cases s*) *auto*
ultimately have $\exists n\ m'.\ x = stl\ s\ !!\ n\ !\ m' \wedge m' < length\ (stl\ s\ !!\ n)$ **by** (*intro less(1)*) *auto*
thus *?thesis* **by** (*metis snth.simps(2)*)
qed
qed
thus $x \in ?R$ **by** (*auto simp*: *sset-range dest*!: *nth-mem*)
next
fix *x xs*
assume $xs \in sset\ s$ $?P\ x \in set\ xs$
thus $x \in ?L$
by (*induct rule*: *sset-induct*)
(*metis UnI1 flat-unfold shift.simps(1) sset-shift*,
metis UnI2 flat-unfold shd-sset stl-sset sset-shift)
qed

57.10 merge a stream of streams

definition *smerge* :: *'a stream stream* ⇒ *'a stream* **where**
smerge ss = *flat* (*smap* (λ*n*. *map* (λ*s*. *s* !! *n*) (*stake* (*Suc n*) *ss*) @ *stake n* (*ss* !! *n*)) *nats*)

lemma *stake-nth*[*simp*]: $m < n \Longrightarrow stake\ n\ s\ !\ m = s\ !!\ m$
by (*induct n arbitrary*: *s m*) (*auto simp*: *nth-Cons'*, *metis Suc-pred snth.simps(2)*)

lemma *snth-sset-smerge*: *ss* !! *n* !! *m* ∈ *sset* (*smerge ss*)
proof (*cases n* ≤ *m*)
case *False* **thus** *?thesis* **unfolding** *smerge-def*
by (*subst sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth simp del*: *stake.simps*
intro!: *exI*[*of - n, OF disjI2*] *exI*[*of - m, OF mp*])
next
case *True* **thus** *?thesis* **unfolding** *smerge-def*
by (*subst sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth image-iff simp del*: *stake.simps snth.simps*
intro!: *exI*[*of - m, OF disjI1*] *bexI*[*of - ss* !! *n*] *exI*[*of - n, OF mp*])
qed

lemma *sset-smerge*: *sset* (*smerge ss*) = $\bigcup$(*sset* ' (*sset ss*))
proof *safe*
fix *x* **assume** *x* ∈ *sset* (*smerge ss*)
thus *x* ∈ $\bigcup$(*sset* ' (*sset ss*))
unfolding *smerge-def* **by** (*subst* (*asm*) *sset-flat*)
(*auto simp*: *stream.set-map in-set-conv-nth sset-range simp del*: *stake.simps*, *fast*+)
next
fix *s x* **assume** *s* ∈ *sset ss x* ∈ *sset s*
thus *x* ∈ *sset* (*smerge ss*) **using** *snth-sset-smerge* **by** (*auto simp*: *sset-range*)
qed

57.11 product of two streams

definition *sproduct* :: 'a *stream* ⇒ 'b *stream* ⇒ ('a × 'b) *stream* **where**
sproduct s1 s2 = *smerge* (*smap* (λx. *smap* (*Pair x*) *s2*) *s1*)

lemma *sset-sproduct*: *sset* (*sproduct s1 s2*) = *sset s1* × *sset s2*
unfolding *sproduct-def sset-smerge* **by** (*auto simp*: *stream.set-map*)

57.12 interleave two streams

primcorec *sinterleave* **where**
shd (*sinterleave s1 s2*) = *shd s1*
| *stl* (*sinterleave s1 s2*) = *sinterleave s2* (*stl s1*)

lemma *sinterleave-code*[*code*]:
sinterleave (*x* ## *s1*) *s2* = *x* ## *sinterleave s2 s1*
by (*subst sinterleave.ctr*) *simp*

lemma *sinterleave-snth*[*simp*]:
even n ⟹ *sinterleave s1 s2* !! *n* = *s1* !! (*n div 2*)
odd n ⟹ *sinterleave s1 s2* !! *n* = *s2* !! (*n div 2*)
by (*induct n arbitrary*: *s1 s2*) *simp-all*

lemma *sset-sinterleave*: *sset* (*sinterleave s1 s2*) = *sset s1* ∪ *sset s2*
proof (*intro equalityI subsetI*)
fix *x* **assume** *x* ∈ *sset* (*sinterleave s1 s2*)
then obtain *n* **where** *x* = *sinterleave s1 s2* !! *n* **unfolding** *sset-range* **by** *blast*
thus *x* ∈ *sset s1* ∪ *sset s2* **by** (*cases even n*) *auto*
next
fix *x* **assume** *x* ∈ *sset s1* ∪ *sset s2*
thus *x* ∈ *sset* (*sinterleave s1 s2*)
proof
assume *x* ∈ *sset s1*
then obtain *n* **where** *x* = *s1* !! *n* **unfolding** *sset-range* **by** *blast*
hence *sinterleave s1 s2* !! (2 ∗ *n*) = *x* **by** *simp*
thus *?thesis* **unfolding** *sset-range* **by** *blast*
next
assume *x* ∈ *sset s2*

then obtain *n* **where** *x = s2 !! n* **unfolding** *sset-range* **by** *blast*
hence *sinterleave s1 s2 !! (2 * n + 1) = x* **by** *simp*
thus *?thesis* **unfolding** *sset-range* **by** *blast*
qed
qed

57.13 zip

primcorec *szip* **where**
shd (szip s1 s2) = (shd s1, shd s2)
| *stl (szip s1 s2) = szip (stl s1) (stl s2)*

lemma *szip-unfold*[*code*]: *szip (a ## s1) (b ## s2) = (a, b) ## (szip s1 s2)*
by (*subst szip.ctr*) *simp*

lemma *snth-szip*[*simp*]: *szip s1 s2 !! n = (s1 !! n, s2 !! n)*
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *stake-szip*[*simp*]:
stake n (szip s1 s2) = zip (stake n s1) (stake n s2)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *sdrop-szip*[*simp*]: *sdrop n (szip s1 s2) = szip (sdrop n s1) (sdrop n s2)*
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *smap-szip-fst*:
smap (λx. f (fst x)) (szip s1 s2) = smap f s1
by (*coinduction arbitrary*: *s1 s2*) *auto*

lemma *smap-szip-snd*:
smap (λx. g (snd x)) (szip s1 s2) = smap g s2
by (*coinduction arbitrary*: *s1 s2*) *auto*

57.14 zip via function

primcorec *smap2* **where**
shd (smap2 f s1 s2) = f (shd s1) (shd s2)
| *stl (smap2 f s1 s2) = smap2 f (stl s1) (stl s2)*

lemma *smap2-unfold*[*code*]:
smap2 f (a ## s1) (b ## s2) = f a b ## (smap2 f s1 s2)
by (*subst smap2.ctr*) *simp*

lemma *smap2-szip*:
smap2 f s1 s2 = smap (case-prod f) (szip s1 s2)
by (*coinduction arbitrary*: *s1 s2*) *auto*

lemma *smap-smap2*[*simp*]:
smap f (smap2 g s1 s2) = smap2 (λx y. f (g x y)) s1 s2
unfolding *smap2-szip stream.map-comp o-def split-def* **..**

lemma *smap2-alt*:
(*smap2 f s1 s2* = *s*) = (∀ *n*. *f* (*s1* !! *n*) (*s2* !! *n*) = *s* !! *n*)
unfolding *smap2-szip smap-alt* **by** *auto*

lemma *snth-smap2*[*simp*]:
smap2 f s1 s2 !! *n* = *f* (*s1* !! *n*) (*s2* !! *n*)
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *stake-smap2*[*simp*]:
stake n (*smap2 f s1 s2*) = *map* (*case-prod f*) (*zip* (*stake n s1*) (*stake n s2*))
by (*induct n arbitrary*: *s1 s2*) *auto*

lemma *sdrop-smap2*[*simp*]:
sdrop n (*smap2 f s1 s2*) = *smap2 f* (*sdrop n s1*) (*sdrop n s2*)
by (*induct n arbitrary*: *s1 s2*) *auto*

end

58 List prefixes, suffixes, and homeomorphic embedding

theory *Sublist*
imports *Main*
begin

58.1 Prefix order on lists

definition *prefix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *prefix xs ys* ⟷ (∃ *zs*. *ys* = *xs* @ *zs*)

definition *strict-prefix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *strict-prefix xs ys* ⟷ *prefix xs ys* ∧ *xs* ≠ *ys*

global-interpretation *prefix-order*: *ordering prefix strict-prefix*
by *standard* (*auto simp add*: *prefix-def strict-prefix-def*)

interpretation *prefix-order*: *order prefix strict-prefix*
by *standard* (*auto simp*: *prefix-def strict-prefix-def*)

global-interpretation *prefix-bot*: *ordering-top* ‹λ*xs ys*. *prefix ys xs*› ‹λ*xs ys*. *strict-prefix ys xs*› ‹[]›
by *standard* (*simp add*: *prefix-def*)

interpretation *prefix-bot*: *order-bot Nil prefix strict-prefix*
by *standard* (*simp add*: *prefix-def*)

lemma *prefixI* [*intro?*]: *ys* = *xs* @ *zs* ⟹ *prefix xs ys*

unfolding *prefix-def* **by** *blast*

lemma *prefixE* [*elim?*]:
assumes *prefix xs ys*
obtains *zs* **where** *ys* = *xs* @ *zs*
using *assms* **unfolding** *prefix-def* **by** *blast*

lemma *strict-prefixI'* [*intro?*]: *ys* = *xs* @ *z* # *zs* ⟹ *strict-prefix xs ys*
unfolding *strict-prefix-def prefix-def* **by** *blast*

lemma *strict-prefixE'* [*elim?*]:
assumes *strict-prefix xs ys*
obtains *z zs* **where** *ys* = *xs* @ *z* # *zs*
proof −
from ‹*strict-prefix xs ys*› **obtain** *us* **where** *ys* = *xs* @ *us* **and** *xs* ≠ *ys*
unfolding *strict-prefix-def prefix-def* **by** *blast*
with *that* **show** *?thesis* **by** (*auto simp add*: *neq-Nil-conv*)
qed

lemma *strict-prefixI* [*intro?*]: *prefix xs ys* ⟹ *xs* ≠ *ys* ⟹ *strict-prefix xs ys*
by(*fact prefix-order.le-neq-trans*)

lemma *strict-prefixE* [*elim?*]:
fixes *xs ys* :: *'a list*
assumes *strict-prefix xs ys*
obtains *prefix xs ys* **and** *xs* ≠ *ys*
using *assms* **unfolding** *strict-prefix-def* **by** *blast*

58.2 Basic properties of prefixes

theorem *Nil-prefix* [*simp*]: *prefix* [] *xs*
by (*fact prefix-bot.bot-least*)

theorem *prefix-Nil* [*simp*]: (*prefix xs* []) = (*xs* = [])
by (*fact prefix-bot.bot-unique*)

lemma *prefix-snoc* [*simp*]: *prefix xs* (*ys* @ [*y*]) ⟷ *xs* = *ys* @ [*y*] ∨ *prefix xs ys*
proof
assume *prefix xs* (*ys* @ [*y*])
then obtain *zs* **where** *zs*: *ys* @ [*y*] = *xs* @ *zs* **..**
show *xs* = *ys* @ [*y*] ∨ *prefix xs ys*
by (*metis append-Nil2 butlast-append butlast-snoc prefixI zs*)
next
assume *xs* = *ys* @ [*y*] ∨ *prefix xs ys*
then show *prefix xs* (*ys* @ [*y*])
using *prefix-def prefix-order.order-trans* **by** *blast*
qed

lemma *Cons-prefix-Cons* [*simp*]: *prefix* (*x* # *xs*) (*y* # *ys*) = (*x* = *y* ∧ *prefix xs ys*)
by (*auto simp add*: *prefix-def*)

lemma *prefix-code* [*code*]:
prefix [] *xs* ⟷ *True*
prefix (*x* # *xs*) [] ⟷ *False*
prefix (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *prefix xs ys*
by *simp-all*

lemma *same-prefix-prefix* [*simp*]: *prefix* (*xs* @ *ys*) (*xs* @ *zs*) = *prefix ys zs*
by (*induct xs*) *simp-all*

lemma *same-prefix-nil* [*simp*]: *prefix* (*xs* @ *ys*) *xs* = (*ys* = [])
by (*simp add*: *prefix-def*)

lemma *prefix-prefix* [*simp*]: *prefix xs ys* ⟹ *prefix xs* (*ys* @ *zs*)
unfolding *prefix-def* **by** *fastforce*

lemma *append-prefixD*: *prefix* (*xs* @ *ys*) *zs* ⟹ *prefix xs zs*
by (*auto simp add*: *prefix-def*)

theorem *prefix-Cons*: *prefix xs* (*y* # *ys*) = (*xs* = [] ∨ (∃ *zs*. *xs* = *y* # *zs* ∧ *prefix zs ys*))
by (*cases xs*) (*auto simp add*: *prefix-def*)

theorem *prefix-append*:
prefix xs (*ys* @ *zs*) = (*prefix xs ys* ∨ (∃ *us*. *xs* = *ys* @ *us* ∧ *prefix us zs*))
proof (*induct zs rule*: *rev-induct*)
case *Nil*
then show *?case* **by** *force*
next
case (*snoc x xs*)
then show *?case*
by (*metis append.assoc*[*of ys xs* [*x*]] *prefix-snoc*)
qed

lemma *append-one-prefix*:
prefix xs ys ⟹ *length xs* < *length ys* ⟹ *prefix* (*xs* @ [*ys* ! *length xs*]) *ys*
proof (*unfold prefix-def*)
assume *a1*: ∃ *zs*. *ys* = *xs* @ *zs*
then obtain *sk* :: *'a list* **where** *sk*: *ys* = *xs* @ *sk* **by** *fastforce*
assume *a2*: *length xs* < *length ys*
have *f1*: ⋀*v*. ([]::*'a list*) @ *v* = *v* **using** *append-Nil2* **by** *simp*
have [] ≠ *sk* **using** *a1 a2 sk less-not-refl* **by** *force*
hence ∃ *v*. *xs* @ *hd sk* # *v* = *ys* **using** *sk* **by** (*metis hd-Cons-tl*)
thus ∃ *zs*. *ys* = (*xs* @ [*ys* ! *length xs*]) @ *zs* **using** *f1* **by** *fastforce*
qed

theorem *prefix-length-le*: *prefix xs ys* ⟹ *length xs* ≤ *length ys*

by (*auto simp add*: *prefix-def*)

lemma *prefix-same-cases*:
prefix (*xs*$_1$::'*a list*) *ys* $\Longrightarrow$ *prefix xs*$_2$ *ys* $\Longrightarrow$ *prefix xs*$_1$ *xs*$_2$ $\vee$ *prefix xs*$_2$ *xs*$_1$
unfolding *prefix-def* **by** (*force simp*: *append-eq-append-conv2*)

lemma *prefix-length-prefix*:
prefix ps xs $\Longrightarrow$ *prefix qs xs* $\Longrightarrow$ *length ps* $\leq$ *length qs* $\Longrightarrow$ *prefix ps qs*
by (*auto simp*: *prefix-def*) (*metis append-Nil2 append-eq-append-conv-if*)

lemma *set-mono-prefix*: *prefix xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by (*auto simp add*: *prefix-def*)

lemma *take-is-prefix*: *prefix* (*take n xs*) *xs*
unfolding *prefix-def* **by** (*metis append-take-drop-id*)

lemma *takeWhile-is-prefix*: *prefix* (*takeWhile P xs*) *xs*
unfolding *prefix-def* **by** (*metis takeWhile-dropWhile-id*)

lemma *prefixeq-butlast*: *prefix* (*butlast xs*) *xs*
by (*simp add*: *butlast-conv-take take-is-prefix*)

lemma *prefix-map-rightE*:
assumes *prefix xs* (*map f ys*)
shows $\exists$ *xs'*. *prefix xs' ys* $\wedge$ *xs* = *map f xs'*
proof −
define *n* **where** *n* = *length xs*
have *xs* = *take n* (*map f ys*)
using *assms* **by** (*auto simp*: *prefix-def n-def*)
thus *?thesis*
by (*intro exI*[*of* - *take n ys*]) (*auto simp*: *take-map take-is-prefix*)
qed

lemma *map-mono-prefix*: *prefix xs ys* $\Longrightarrow$ *prefix* (*map f xs*) (*map f ys*)
by (*auto simp*: *prefix-def*)

lemma *filter-mono-prefix*: *prefix xs ys* $\Longrightarrow$ *prefix* (*filter P xs*) (*filter P ys*)
by (*auto simp*: *prefix-def*)

lemma *sorted-antimono-prefix*: *prefix xs ys* $\Longrightarrow$ *sorted ys* $\Longrightarrow$ *sorted xs*
by (*metis sorted-append prefix-def*)

lemma *prefix-length-less*: *strict-prefix xs ys* $\Longrightarrow$ *length xs* < *length ys*
by (*auto simp*: *strict-prefix-def prefix-def*)

lemma *prefix-snocD*: *prefix* (*xs*@[*x*]) *ys* $\Longrightarrow$ *strict-prefix xs ys*
by (*simp add*: *strict-prefixI' prefix-order.dual-order.strict-trans1*)

lemma *strict-prefix-simps* [*simp*, *code*]:

strict-prefix xs [] ⟷ *False*
strict-prefix [] (*x* # *xs*) ⟷ *True*
strict-prefix (*x* # *xs*) (*y* # *ys*) ⟷ *x* = *y* ∧ *strict-prefix xs ys*
by (*simp-all add*: *strict-prefix-def cong*: *conj-cong*)

lemma *take-strict-prefix*: *strict-prefix xs ys* ⟹ *strict-prefix* (*take n xs*) *ys*
proof (*induct n arbitrary*: *xs ys*)
case *0*
then show *?case* **by** (*cases ys*) *simp-all*
next
case (*Suc n*)
then show *?case* **by** (*metis prefix-order.less-trans strict-prefixI take-is-prefix*)
qed

lemma *prefix-takeWhile*:
assumes *prefix xs ys*
shows *prefix* (*takeWhile P xs*) (*takeWhile P ys*)
proof −
from *assms* **obtain** *zs* **where** *ys*: *ys* = *xs* @ *zs*
by (*auto simp*: *prefix-def*)
have *prefix* (*takeWhile P xs*) (*takeWhile P* (*xs* @ *zs*))
by (*induction xs*) *auto*
thus *?thesis* **by** (*simp add*: *ys*)
qed

lemma *prefix-dropWhile*:
assumes *prefix xs ys*
shows *prefix* (*dropWhile P xs*) (*dropWhile P ys*)
proof −
from *assms* **obtain** *zs* **where** *ys*: *ys* = *xs* @ *zs*
by (*auto simp*: *prefix-def*)
have *prefix* (*dropWhile P xs*) (*dropWhile P* (*xs* @ *zs*))
by (*induction xs*) *auto*
thus *?thesis* **by** (*simp add*: *ys*)
qed

lemma *prefix-remdups-adj*:
assumes *prefix xs ys*
shows *prefix* (*remdups-adj xs*) (*remdups-adj ys*)
using *assms*
proof (*induction length xs arbitrary*: *xs ys rule*: *less-induct*)
case (*less xs*)
show *?case*
proof (*cases xs*)
case [*simp*]: (*Cons x xs'*)
then obtain *y ys'* **where** [*simp*]: *ys* = *y* # *ys'*
using ‹*prefix xs ys*› **by** (*cases ys*) *auto*
from *less* **show** *?thesis*
by (*auto simp*: *remdups-adj-Cons' less-Suc-eq-le length-dropWhile-le*

```
            intro!: less prefix-dropWhile)
  qed auto
qed

lemma not-prefix-cases:
  assumes pfx: ¬ prefix ps ls
  obtains
    (c1) ps ≠ [] and ls = []
  | (c2) a as x xs where ps = a#as and ls = x#xs and x = a and ¬ prefix as xs
  | (c3) a as x xs where ps = a#as and ls = x#xs and x ≠ a
proof (cases ps)
  case Nil
  then show ?thesis using pfx by simp
next
  case (Cons a as)
  note c = ‹ps = a#as›
  show ?thesis
  proof (cases ls)
    case Nil then show ?thesis by (metis append-Nil2 pfx c1 same-prefix-nil)
  next
    case (Cons x xs)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ prefix as xs using pfx c Cons True by simp
      with c Cons True show ?thesis by (rule c2)
    next
      case False
      with c Cons show ?thesis by (rule c3)
    qed
  qed
qed

lemma not-prefix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ prefix ps ls
    and base: ⋀x xs. P (x#xs) []
    and r1: ⋀x xs y ys. x ≠ y ⟹ P (x#xs) (y#ys)
    and r2: ⋀x xs y ys. ⟦ x = y; ¬ prefix xs ys; P xs ys ⟧ ⟹ P (x#xs) (y#ys)
  shows P ps ls using np
proof (induct ls arbitrary: ps)
  case Nil
  then show ?case
    by (auto simp: neq-Nil-conv elim!: not-prefix-cases intro!: base)
next
  case (Cons y ys)
  then have npfx: ¬ prefix ps (y # ys) by simp
  then obtain x xs where pv: ps = x # xs
    by (rule not-prefix-cases) auto
  show ?case by (metis Cons.hyps Cons-prefix-Cons npfx pv r1 r2)
```

qed

58.3 Prefixes

primrec *prefixes* **where**
prefixes [] = [[]] |
prefixes (*x*#*xs*) = [] # *map* ((#) *x*) (*prefixes xs*)

lemma *in-set-prefixes*[*simp*]: *xs* ∈ *set* (*prefixes ys*) ⟷ *prefix xs ys*
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** (*cases ys*) *auto*
next
case (*Cons a xs*)
then show *?case* **by** (*cases ys*) *auto*
qed

lemma *length-prefixes*[*simp*]: *length* (*prefixes xs*) = *length xs*+*1*
by (*induction xs*) *auto*

lemma *distinct-prefixes* [*intro*]: *distinct* (*prefixes xs*)
by (*induction xs*) (*auto simp*: *distinct-map*)

lemma *prefixes-snoc* [*simp*]: *prefixes* (*xs*@[*x*]) = *prefixes xs* @ [*xs*@[*x*]]
by (*induction xs*) *auto*

lemma *prefixes-not-Nil* [*simp*]: *prefixes xs* ≠ []
by (*cases xs*) *auto*

lemma *hd-prefixes* [*simp*]: *hd* (*prefixes xs*) = []
by (*cases xs*) *simp-all*

lemma *last-prefixes* [*simp*]: *last* (*prefixes xs*) = *xs*
by (*induction xs*) (*simp-all add*: *last-map*)

lemma *prefixes-append*:
prefixes (*xs* @ *ys*) = *prefixes xs* @ *map* (λ*ys*′. *xs* @ *ys*′) (*tl* (*prefixes ys*))
proof (*induction xs*)
case *Nil*
thus *?case* **by** (*cases ys*) *auto*
qed *simp-all*

lemma *prefixes-eq-snoc*:
prefixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *zs*@[*z*] ∧ *xs* = *prefixes zs*)) ∧ *x* = *ys*
by (*cases ys rule*: *rev-cases*) *auto*

lemma *prefixes-tailrec* [*code*]:
prefixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, *rev* (*x*#*acc1*)#*acc2*))

([],[[]]) *xs*))
proof −
have *foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, *rev* (*x*#*acc1*)#*acc2*)) (*ys*, *rev ys* # *zs*) *xs* =
(*rev xs* @ *ys*, *rev* (*map* (λ*as*. *rev ys* @ *as*) (*prefixes xs*)) @ *zs*) **for** *ys zs*
proof (*induction xs arbitrary*: *ys zs*)
case (*Cons x xs ys zs*)
from *Cons.IH*[*of x* # *ys rev ys* # *zs*]
show *?case* **by** (*simp add*: *o-def*)
qed *simp-all*
from *this* [*of* [] []] **show** *?thesis* **by** *simp*
qed

lemma *set-prefixes-eq*: *set* (*prefixes xs*) = {*ys*. *prefix ys xs*}
by *auto*

lemma *card-set-prefixes* [*simp*]: *card* (*set* (*prefixes xs*)) = *Suc* (*length xs*)
by (*subst distinct-card*) *auto*

lemma *set-prefixes-append*:
set (*prefixes* (*xs* @ *ys*)) = *set* (*prefixes xs*) ∪ {*xs* @ *ys*′ |*ys*′. *ys*′ ∈ *set* (*prefixes ys*)}
by (*subst prefixes-append*, *cases ys*) *auto*

58.4 Longest Common Prefix

definition *Longest-common-prefix* :: ′*a list set* ⇒ ′*a list* **where**
Longest-common-prefix L = (*ARG-MAX length ps*. ∀ *xs* ∈ *L*. *prefix ps xs*)

lemma *Longest-common-prefix-ex*: *L* ≠ {} ⟹
∃ *ps*. (∀ *xs* ∈ *L*. *prefix ps xs*) ∧ (∀ *qs*. (∀ *xs* ∈ *L*. *prefix qs xs*) ⟶ *size qs* ≤ *size ps*)
(**is** - ⟹ ∃ *ps*. *?P L ps*)
proof(*induction LEAST n*. ∃ *xs* ∈*L*. *n* = *length xs arbitrary*: *L*)
case *0*
have [] ∈ *L* **using** *0.hyps LeastI*[*of* λ*n*. ∃ *xs*∈*L*. *n* = *length xs*] ‹*L* ≠ {}›
by *auto*
hence *?P L* [] **by**(*auto*)
thus *?case* **..**
next
case (*Suc n*)
let *?EX* = λ*n*. ∃ *xs*∈*L*. *n* = *length xs*
obtain *x xs* **where** *xxs*: *x*#*xs* ∈ *L size xs* = *n* **using** *Suc.prems Suc.hyps*(*2*)
by(*metis LeastI-ex*[*of ?EX*] *Suc-length-conv ex-in-conv*)
hence [] ∉ *L* **using** *Suc.hyps*(*2*) **by** *auto*
show *?case*
proof (*cases* ∀ *xs* ∈ *L*. ∃ *ys*. *xs* = *x*#*ys*)
case *True*
let *?L* = {*ys*. *x*#*ys* ∈ *L*}

```
    have 1: (LEAST n. ∃ xs ∈ ?L. n = length xs) = n
      using xxs Suc.prems Suc.hyps(2) Least-le[of ?EX]
      by − (rule Least-equality, fastforce+)
    have 2: ?L ≠ {} using ‹x # xs ∈ L› by auto
    from Suc.hyps(1)[OF 1[symmetric] 2] obtain ps where IH: ?P ?L ps ..
    have length qs ≤ Suc (length ps)
      if ∀ qs. (∀ xa. x # xa ∈ L ⟶ prefix qs xa) ⟶ length qs ≤ length ps
      and ∀ xs∈L. prefix qs xs for qs
    proof −
      from that have length (tl qs) ≤ length ps
        by (metis Cons-prefix-Cons hd-Cons-tl list.sel(2) Nil-prefix)
      thus ?thesis by auto
    qed
    hence ?P L (x#ps) using True IH by auto
    thus ?thesis ..
  next
    case False
    then obtain y ys where yys: x≠y y#ys ∈ L using ‹[] ∉ L›
      by (auto) (metis list.exhaust)
    have ∀ qs. (∀ xs∈L. prefix qs xs) ⟶ qs = [] using yys ‹x#xs ∈ L›
      by auto (metis Cons-prefix-Cons prefix-Cons)
    hence ?P L [] by auto
    thus ?thesis ..
  qed
qed

lemma Longest-common-prefix-unique:
  ‹∃! ps. (∀ xs ∈ L. prefix ps xs) ∧ (∀ qs. (∀ xs ∈ L. prefix qs xs) ⟶ length qs ≤
length ps)›
  if ‹L ≠ {}›
  apply (intro ex-ex1I[OF Longest-common-prefix-ex [OF that]])
  by (meson that all-not-in-conv prefix-length-prefix prefix-order.dual-order.eq-iff)

lemma Longest-common-prefix-eq:
  ⟦ L ≠ {};  ∀ xs ∈ L. prefix ps xs;
    ∀ qs. (∀ xs ∈ L. prefix qs xs) ⟶ size qs ≤ size ps ⟧
  ⟹ Longest-common-prefix L = ps
  unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
  by(rule some1-equality[OF Longest-common-prefix-unique]) auto

lemma Longest-common-prefix-prefix:
  xs ∈ L ⟹ prefix (Longest-common-prefix L) xs
  unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
  by(rule someI2-ex[OF Longest-common-prefix-ex]) auto

lemma Longest-common-prefix-longest:
  L ≠ {} ⟹ ∀ xs∈L. prefix ps xs ⟹ length ps ≤ length(Longest-common-prefix
L)
  unfolding Longest-common-prefix-def arg-max-def is-arg-max-linorder
```

by(*rule someI2-ex*[*OF Longest-common-prefix-ex*]) *auto*

lemma *Longest-common-prefix-max-prefix*:
L ≠ {} ⟹ ∀ *xs*∈*L*. *prefix ps xs* ⟹ *prefix ps* (*Longest-common-prefix L*)
by(*metis Longest-common-prefix-prefix Longest-common-prefix-longest prefix-length-prefix ex-in-conv*)

lemma *Longest-common-prefix-Nil*: [] ∈ *L* ⟹ *Longest-common-prefix L* = []
using *Longest-common-prefix-prefix prefix-Nil* **by** *blast*

lemma *Longest-common-prefix-image-Cons*:
assumes *L* ≠ {}
shows *Longest-common-prefix* ((#) *x* ‘ *L*) = *x* # *Longest-common-prefix L*
proof (*intro Longest-common-prefix-eq strip*)
show ⋀*qs*. ∀ *xs*∈(#) *x* ‘ *L*. *prefix qs xs* ⟹
length qs ≤ *length* (*x* # *Longest-common-prefix L*)
by (*metis assms Longest-common-prefix-longest*[*of L*] *Cons-prefix-Cons Suc-le-mono hd-Cons-tl*
image-eqI length-Cons prefix-bot.bot-least prefix-length-le)
qed (*auto simp add*: *assms Longest-common-prefix-prefix*)

lemma *Longest-common-prefix-eq-Cons*: **assumes** *L* ≠ {} [] ∉ *L* ∀ *xs*∈*L*. *hd xs* = *x*
shows *Longest-common-prefix L* = *x* # *Longest-common-prefix* {*ys*. *x*#*ys* ∈ *L*}
proof −
have *L* = (#) *x* ‘ {*ys*. *x*#*ys* ∈ *L*} **using** *assms*(*2*,*3*)
by (*auto simp*: *image-def*)(*metis hd-Cons-tl*)
thus *?thesis*
by (*metis Longest-common-prefix-image-Cons image-is-empty assms*(*1*))
qed

lemma *Longest-common-prefix-eq-Nil*:
⟦*x*#*ys* ∈ *L*; *y*#*zs* ∈ *L*; *x* ≠ *y* ⟧ ⟹ *Longest-common-prefix L* = []
by (*metis Longest-common-prefix-prefix list.inject prefix-Cons*)

fun *longest-common-prefix* :: *'a list* ⇒ *'a list* ⇒ *'a list* **where**
longest-common-prefix (*x*#*xs*) (*y*#*ys*) =
(*if x*=*y then x* # *longest-common-prefix xs ys else* []) |
longest-common-prefix - - = []

lemma *longest-common-prefix-prefix1*:
prefix (*longest-common-prefix xs ys*) *xs*
by(*induction xs ys rule*: *longest-common-prefix.induct*) *auto*

lemma *longest-common-prefix-prefix2*:
prefix (*longest-common-prefix xs ys*) *ys*
by(*induction xs ys rule*: *longest-common-prefix.induct*) *auto*

lemma *longest-common-prefix-max-prefix*:

⟦ *prefix ps xs*; *prefix ps ys* ⟧
⟹ *prefix ps* (*longest-common-prefix xs ys*)
by(*induction xs ys arbitrary*: *ps rule*: *longest-common-prefix.induct*)
(*auto simp*: *prefix-Cons*)

58.5 Parallel lists

definition *parallel* :: *'a list* ⇒ *'a list* ⇒ *bool* (**infixl** ‹∥› *50*)
where (*xs* ∥ *ys*) = (¬ *prefix xs ys* ∧ ¬ *prefix ys xs*)

lemma *parallelI* [*intro*]: ¬ *prefix xs ys* ⟹ ¬ *prefix ys xs* ⟹ *xs* ∥ *ys*
unfolding *parallel-def* **by** *blast*

lemma *parallelE* [*elim*]:
assumes *xs* ∥ *ys*
obtains ¬ *prefix xs ys* ∧ ¬ *prefix ys xs*
using *assms* **unfolding** *parallel-def* **by** *blast*

theorem *prefix-cases*:
obtains *prefix xs ys* | *strict-prefix ys xs* | *xs* ∥ *ys*
unfolding *parallel-def strict-prefix-def* **by** *blast*

lemma *parallel-cancel*: *a*#*xs* ∥ *a*#*ys* ⟹ *xs* ∥ *ys*
by (*simp add*: *parallel-def*)

theorem *parallel-decomp*:
xs ∥ *ys* ⟹ ∃ *as b bs c cs*. *b* ≠ *c* ∧ *xs* = *as* @ *b* # *bs* ∧ *ys* = *as* @ *c* # *cs*
proof (*induct rule*: *list-induct2'*, *blast*, *force*, *force*)
case (*4 x xs y ys*)
then show *?case*
proof (*cases x* ≠ *y*, *blast*)
assume ¬ *x* ≠ *y* **hence** *x* = *y* **by** *blast*
then show *?thesis*
using *4.hyps*[*OF parallel-cancel*[*OF 4.prems*[*folded* ‹*x* = *y*›]]]
by (*meson Cons-eq-appendI*)
qed
qed

lemma *parallel-append*: *a* ∥ *b* ⟹ *a* @ *c* ∥ *b* @ *d*
by (*meson parallelE parallelI prefixI prefix-order.trans prefix-same-cases*)

lemma *parallel-appendI*: *xs* ∥ *ys* ⟹ *x* = *xs* @ *xs'* ⟹ *y* = *ys* @ *ys'* ⟹ *x* ∥ *y*
by (*simp add*: *parallel-append*)

lemma *parallel-commute*: *a* ∥ *b* ⟷ *b* ∥ *a*
unfolding *parallel-def* **by** *auto*

58.6 Suffix order on lists

definition *suffix* :: *'a list* ⇒ *'a list* ⇒ *bool*

where *suffix xs ys* = (∃ *zs*. *ys* = *zs* @ *xs*)

definition *strict-suffix* :: *'a list* ⇒ *'a list* ⇒ *bool*
where *strict-suffix xs ys* ⟷ *suffix xs ys* ∧ *xs* ≠ *ys*

global-interpretation *suffix-order*: *ordering suffix strict-suffix*
by *standard* (*auto simp*: *suffix-def strict-suffix-def*)

interpretation *suffix-order*: *order suffix strict-suffix*
by *standard* (*auto simp*: *suffix-def strict-suffix-def*)

global-interpretation *suffix-bot*: *ordering-top* ‹λ*xs ys*. *suffix ys xs*› ‹λ*xs ys*. *strict-suffix ys xs*› ‹[]›
by *standard* (*simp add*: *suffix-def*)

interpretation *suffix-bot*: *order-bot Nil suffix strict-suffix*
by *standard* (*simp add*: *suffix-def*)

lemma *suffixI* [*intro?*]: *ys* = *zs* @ *xs* ⟹ *suffix xs ys*
unfolding *suffix-def* **by** *blast*

lemma *suffixE* [*elim?*]:
assumes *suffix xs ys*
obtains *zs* **where** *ys* = *zs* @ *xs*
using *assms* **unfolding** *suffix-def* **by** *blast*

lemma *suffix-tl* [*simp*]: *suffix* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *suffix-def*)

lemma *strict-suffix-tl* [*simp*]: *xs* ≠ [] ⟹ *strict-suffix* (*tl xs*) *xs*
by (*induct xs*) (*auto simp*: *strict-suffix-def suffix-def*)

lemma *Nil-suffix* [*simp*]: *suffix* [] *xs*
by (*simp add*: *suffix-def*)

lemma *suffix-Nil* [*simp*]: (*suffix xs* []) = (*xs* = [])
by (*auto simp add*: *suffix-def*)

lemma *suffix-ConsI*: *suffix xs ys* ⟹ *suffix xs* (*y* # *ys*)
by (*auto simp add*: *suffix-def*)

lemma *suffix-ConsD*: *suffix* (*x* # *xs*) *ys* ⟹ *suffix xs ys*
by (*auto simp add*: *suffix-def*)

lemma *suffix-appendI*: *suffix xs ys* ⟹ *suffix xs* (*zs* @ *ys*)
by (*auto simp add*: *suffix-def*)

lemma *suffix-appendD*: *suffix* (*zs* @ *xs*) *ys* ⟹ *suffix xs ys*
by (*auto simp add*: *suffix-def*)

lemma *strict-suffix-set-subset*: *strict-suffix xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by (*auto simp*: *strict-suffix-def suffix-def*)

lemma *set-mono-suffix*: *suffix xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by (*auto simp*: *suffix-def*)

lemma *sorted-antimono-suffix*: *suffix xs ys* $\Longrightarrow$ *sorted ys* $\Longrightarrow$ *sorted xs*
by (*metis sorted-append suffix-def*)

lemma *suffix-ConsD2*: *suffix* (*x* # *xs*) (*y* # *ys*) $\Longrightarrow$ *suffix xs ys*
proof −
assume *suffix* (*x* # *xs*) (*y* # *ys*)
then obtain *zs* **where** *y* # *ys* = *zs* @ *x* # *xs* **..**
then show *?thesis*
by (*induct zs*) (*auto intro*!: *suffix-appendI suffix-ConsI*)
qed

lemma *suffix-to-prefix* [*code*]: *suffix xs ys* $\longleftrightarrow$ *prefix* (*rev xs*) (*rev ys*)
proof
assume *suffix xs ys*
then obtain *zs* **where** *ys* = *zs* @ *xs* **..**
then have *rev ys* = *rev xs* @ *rev zs* **by** *simp*
then show *prefix* (*rev xs*) (*rev ys*) **..**
next
assume *prefix* (*rev xs*) (*rev ys*)
then obtain *zs* **where** *rev ys* = *rev xs* @ *zs* **..**
then have *rev* (*rev ys*) = *rev zs* @ *rev* (*rev xs*) **by** *simp*
then have *ys* = *rev zs* @ *xs* **by** *simp*
then show *suffix xs ys* **..**
qed

lemma *strict-suffix-to-prefix* [*code*]: *strict-suffix xs ys* $\longleftrightarrow$ *strict-prefix* (*rev xs*) (*rev ys*)
by (*auto simp*: *suffix-to-prefix strict-suffix-def strict-prefix-def*)

lemma *distinct-suffix*: *distinct ys* $\Longrightarrow$ *suffix xs ys* $\Longrightarrow$ *distinct xs*
by (*clarsimp elim*!: *suffixE*)

lemma *map-mono-suffix*: *suffix xs ys* $\Longrightarrow$ *suffix* (*map f xs*) (*map f ys*)
by (*auto elim*!: *suffixE intro*: *suffixI*)

lemma *map-mono-strict-suffix*: *strict-suffix xs ys* $\Longrightarrow$ *strict-suffix* (*map f xs*) (*map f ys*)
by (*auto simp*: *strict-suffix-def suffix-def*)

lemma *filter-mono-suffix*: *suffix xs ys* $\Longrightarrow$ *suffix* (*filter P xs*) (*filter P ys*)
by (*auto simp*: *suffix-def*)

lemma *suffix-drop*: *suffix* (*drop n as*) *as*
unfolding *suffix-def* **by** (*metis append-take-drop-id*)

lemma *suffix-dropWhile*: *suffix* (*dropWhile P xs*) *xs*
unfolding *suffix-def* **by** (*metis takeWhile-dropWhile-id*)

lemma *suffix-take*: *suffix xs ys* $\Longrightarrow$ *ys* = *take* (*length ys* − *length xs*) *ys* @ *xs*
by (*auto elim*!: *suffixE*)

lemma *strict-suffix-reflclp-conv*: *strict-suffix*$^{==}$ = *suffix*
by (*intro ext*) (*auto simp*: *suffix-def strict-suffix-def*)

lemma *suffix-lists*: *suffix xs ys* $\Longrightarrow$ *ys* ∈ *lists A* $\Longrightarrow$ *xs* ∈ *lists A*
unfolding *suffix-def* **by** *auto*

lemma *suffix-snoc* [*simp*]: *suffix xs* (*ys* @ [*y*]) $\longleftrightarrow$ *xs* = [] ∨ (∃ *zs*. *xs* = *zs* @ [*y*] ∧ *suffix zs ys*)
by (*cases xs rule*: *rev-cases*) (*auto simp*: *suffix-def*)

lemma *snoc-suffix-snoc* [*simp*]: *suffix* (*xs* @ [*x*]) (*ys* @ [*y*]) = (*x* = *y* ∧ *suffix xs ys*)
by (*auto simp add*: *suffix-def*)

lemma *same-suffix-suffix* [*simp*]: *suffix* (*ys* @ *xs*) (*zs* @ *xs*) = *suffix ys zs*
by (*simp add*: *suffix-to-prefix*)

lemma *same-suffix-nil* [*simp*]: *suffix* (*ys* @ *xs*) *xs* = (*ys* = [])
by (*simp add*: *suffix-to-prefix*)

theorem *suffix-Cons*: *suffix xs* (*y* # *ys*) $\longleftrightarrow$ *xs* = *y* # *ys* ∨ *suffix xs ys*
unfolding *suffix-def* **by** (*auto simp*: *Cons-eq-append-conv*)

theorem *suffix-append*:
suffix xs (*ys* @ *zs*) $\longleftrightarrow$ *suffix xs zs* ∨ (∃ *xs*′. *xs* = *xs*′ @ *zs* ∧ *suffix xs*′ *ys*)
by (*auto simp*: *suffix-def append-eq-append-conv2*)

theorem *suffix-length-le*: *suffix xs ys* $\Longrightarrow$ *length xs* ≤ *length ys*
by (*auto simp add*: *suffix-def*)

lemma *suffix-same-cases*:
suffix (*xs*$_1$::′*a list*) *ys* $\Longrightarrow$ *suffix xs*$_2$ *ys* $\Longrightarrow$ *suffix xs*$_1$ *xs*$_2$ ∨ *suffix xs*$_2$ *xs*$_1$
unfolding *suffix-def* **by** (*force simp*: *append-eq-append-conv2*)

lemma *suffix-length-suffix*:
suffix ps xs $\Longrightarrow$ *suffix qs xs* $\Longrightarrow$ *length ps* ≤ *length qs* $\Longrightarrow$ *suffix ps qs*
by (*auto simp*: *suffix-to-prefix intro*: *prefix-length-prefix*)

lemma *suffix-length-less*: *strict-suffix xs ys* $\Longrightarrow$ *length xs* < *length ys*
by (*auto simp*: *strict-suffix-def suffix-def*)

lemma *suffix-ConsD'*: *suffix* (*x*#*xs*) *ys* ⟹ *strict-suffix xs ys*
by (*auto simp*: *strict-suffix-def suffix-def*)

lemma *drop-strict-suffix*: *strict-suffix xs ys* ⟹ *strict-suffix* (*drop n xs*) *ys*
proof (*induct n arbitrary*: *xs ys*)
case *0*
then show *?case* **by** (*cases ys*) *simp-all*
next
case (*Suc n*)
then show *?case*
by (*cases xs*) (*auto intro*: *Suc dest*: *suffix-ConsD' suffix-order.less-imp-le*)
qed

lemma *suffix-map-rightE*:
assumes *suffix xs* (*map f ys*)
shows ∃ *xs'*. *suffix xs' ys* ∧ *xs* = *map f xs'*
proof −
from *assms* **obtain** *xs'* **where** *xs'*: *map f ys* = *xs'* @ *xs*
by (*auto simp*: *suffix-def*)
define *n* **where** *n* = *length xs'*
have *xs* = *drop n* (*map f ys*)
by (*simp add*: *xs' n-def*)
thus *?thesis*
by (*intro exI*[*of* - *drop n ys*]) (*auto simp*: *drop-map suffix-drop*)
qed

lemma *suffix-remdups-adj*: *suffix xs ys* ⟹ *suffix* (*remdups-adj xs*) (*remdups-adj ys*)
using *prefix-remdups-adj*[*of rev xs rev ys*]
by (*simp add*: *suffix-to-prefix*)

lemma *not-suffix-cases*:
assumes *pfx*: ¬ *suffix ps ls*
obtains
(*c1*) *ps* ≠ [] **and** *ls* = []
| (*c2*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* = *a* **and** ¬ *suffix as xs*
| (*c3*) *a as x xs* **where** *ps* = *as*@[*a*] **and** *ls* = *xs*@[*x*] **and** *x* ≠ *a*
proof (*cases ps rule*: *rev-cases*)
case *Nil*
then show *?thesis* **using** *pfx* **by** *simp*
next
case (*snoc as a*)
note *c* = ‹*ps* = *as*@[*a*]›
show *?thesis*
proof (*cases ls rule*: *rev-cases*)
case *Nil* **then show** *?thesis* **by** (*metis append-Nil2 pfx c1 same-suffix-nil*)
next

```
    case (snoc xs x)
    show ?thesis
    proof (cases x = a)
      case True
      have ¬ suffix as xs using pfx c snoc True by simp
      with c snoc True show ?thesis by (rule c2)
    next
      case False
      with c snoc show ?thesis by (rule c3)
    qed
  qed
qed

lemma not-suffix-induct [consumes 1, case-names Nil Neq Eq]:
  assumes np: ¬ suffix ps ls
    and base: ⋀x xs. P (xs@[x]) []
    and r1: ⋀x xs y ys. x ≠ y ⟹ P (xs@[x]) (ys@[y])
    and r2: ⋀x xs y ys. ⟦ x = y; ¬ suffix xs ys; P xs ys ⟧ ⟹ P (xs@[x]) (ys@[y])
  shows P ps ls using np
proof (induct ls arbitrary: ps rule: rev-induct)
  case Nil
  then show ?case by (cases ps rule: rev-cases) (auto intro: base)
next
  case (snoc y ys ps)
  then have npfx: ¬ suffix ps (ys @ [y]) by simp
  then obtain x xs where pv: ps = xs @ [x]
    by (rule not-suffix-cases) auto
  show ?case by (metis snoc.hyps snoc-suffix-snoc npfx pv r1 r2)
qed


lemma parallelD1: x ∥ y ⟹ ¬ prefix x y
  by blast

lemma parallelD2: x ∥ y ⟹ ¬ prefix y x
  by blast

lemma parallel-Nil1 [simp]: ¬ x ∥ []
  unfolding parallel-def by simp

lemma parallel-Nil2 [simp]: ¬ [] ∥ x
  unfolding parallel-def by simp

lemma Cons-parallelI1: a ≠ b ⟹ a # as ∥ b # bs
  by auto

lemma Cons-parallelI2: ⟦ a = b; as ∥ bs ⟧ ⟹ a # as ∥ b # bs
  by (metis Cons-prefix-Cons parallelE parallelI)
```

lemma *not-equal-is-parallel*:
  assumes *neq*: $xs \neq ys$
    and *len*: *length xs* = *length ys*
  shows $xs \parallel ys$
  using *len neq*
proof (*induct rule*: *list-induct2*)
  case *Nil*
  then show *?case* **by** *simp*
next
  case (*Cons a as b bs*)
  have *ih*: $as \neq bs \Longrightarrow as \parallel bs$ **by** *fact*
  show *?case*
  proof (*cases a* = *b*)
    case *True*
    then have $as \neq bs$ **using** *Cons* **by** *simp*
    then show *?thesis* **by** (*rule Cons-parallelI2* [*OF True ih*])
  next
    case *False*
    then show *?thesis* **by** (*rule Cons-parallelI1*)
  qed
qed

58.7 Suffixes

primrec *suffixes* **where**
  suffixes [] = [[]]
| *suffixes* (*x*#*xs*) = *suffixes xs* @ [*x* # *xs*]

lemma *in-set-suffixes* [*simp*]: *xs* ∈ *set* (*suffixes ys*) ⟷ *suffix xs ys*
  by (*induction ys*) (*auto simp*: *suffix-def Cons-eq-append-conv*)

lemma *distinct-suffixes* [*intro*]: *distinct* (*suffixes xs*)
  by (*induction xs*) (*auto simp*: *suffix-def*)

lemma *length-suffixes* [*simp*]: *length* (*suffixes xs*) = *Suc* (*length xs*)
  by (*induction xs*) *auto*

lemma *suffixes-snoc* [*simp*]: *suffixes* (*xs* @ [*x*]) = [] # *map* (λ*ys*. *ys* @ [*x*]) (*suffixes xs*)
  by (*induction xs*) *auto*

lemma *suffixes-not-Nil* [*simp*]: *suffixes xs* ≠ []
  by (*cases xs*) *auto*

lemma *hd-suffixes* [*simp*]: *hd* (*suffixes xs*) = []
  by (*induction xs*) *simp-all*

lemma *last-suffixes* [*simp*]: *last* (*suffixes xs*) = *xs*
  by (*cases xs*) *simp-all*

lemma *suffixes-append*:
suffixes (*xs* @ *ys*) = *suffixes ys* @ *map* (λ*xs'*. *xs'* @ *ys*) (*tl* (*suffixes xs*))
proof (*induction ys rule*: *rev-induct*)
case *Nil*
thus *?case* **by** (*cases xs rule*: *rev-cases*) *auto*
next
case (*snoc y ys*)
show *?case*
by (*simp only*: *append.assoc* [*symmetric*] *suffixes-snoc snoc.IH*) *simp*
qed

lemma *suffixes-eq-snoc*:
suffixes ys = *xs* @ [*x*] ⟷
(*ys* = [] ∧ *xs* = [] ∨ (∃ *z zs*. *ys* = *z*#*zs* ∧ *xs* = *suffixes zs*)) ∧ *x* = *ys*
by (*cases ys*) *auto*

lemma *suffixes-tailrec* [*code*]:
suffixes xs = *rev* (*snd* (*foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, (*x*#*acc1*)#*acc2*)) ([],[[]])
(*rev xs*)))
proof −
have *foldl* (λ(*acc1*, *acc2*) *x*. (*x*#*acc1*, (*x*#*acc1*)#*acc2*)) (*ys*, *ys* # *zs*) (*rev xs*)
=
(*xs* @ *ys*, *rev* (*map* (λ*as*. *as* @ *ys*) (*suffixes xs*)) @ *zs*) **for** *ys zs*
proof (*induction xs arbitrary*: *ys zs*)
case (*Cons x xs ys zs*)
from *Cons.IH*[*of ys zs*]
show *?case* **by** (*simp add*: *o-def case-prod-unfold*)
qed *simp-all*
from *this* [*of* [] []] **show** *?thesis* **by** *simp*
qed

lemma *set-suffixes-eq*: *set* (*suffixes xs*) = {*ys*. *suffix ys xs*}
by *auto*

lemma *card-set-suffixes* [*simp*]: *card* (*set* (*suffixes xs*)) = *Suc* (*length xs*)
by (*subst distinct-card*) *auto*

lemma *set-suffixes-append*:
set (*suffixes* (*xs* @ *ys*)) = *set* (*suffixes ys*) ∪ {*xs'* @ *ys* |*xs'*. *xs'* ∈ *set* (*suffixes xs*)}
by (*subst suffixes-append*, *cases xs rule*: *rev-cases*) *auto*

lemma *suffixes-conv-prefixes*: *suffixes xs* = *map rev* (*prefixes* (*rev xs*))
by (*induction xs*) *auto*

lemma *prefixes-conv-suffixes*: *prefixes xs* = *map rev* (*suffixes* (*rev xs*))
by (*induction xs*) *auto*

lemma *prefixes-rev*: *prefixes* (*rev xs*) = *map rev* (*suffixes xs*)
by (*induction xs*) *auto*

lemma *suffixes-rev*: *suffixes* (*rev xs*) = *map rev* (*prefixes xs*)
by (*induction xs*) *auto*

58.8 Homeomorphic embedding on lists

inductive *list-emb* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a list* ⇒ '*a list* ⇒ *bool*
for *P* :: ('*a* ⇒ '*a* ⇒ *bool*)
where
list-emb-Nil [*intro*, *simp*]: *list-emb P* [] *ys*
| *list-emb-Cons* [*intro*] : *list-emb P xs ys* ⟹ *list-emb P xs* (*y*#*ys*)
| *list-emb-Cons2* [*intro*]: *P x y* ⟹ *list-emb P xs ys* ⟹ *list-emb P* (*x*#*xs*) (*y*#*ys*)

lemma *list-emb-mono*:
assumes ⋀*x y*. *P x y* ⟶ *Q x y*
shows *list-emb P xs ys* ⟶ *list-emb Q xs ys*
proof
assume *list-emb P xs ys*
then show *list-emb Q xs ys* **by** (*induct*) (*auto simp*: *assms*)
qed

lemma *list-emb-Nil2* [*simp*]:
assumes *list-emb P xs* [] **shows** *xs* = []
using *assms* **by** (*cases rule*: *list-emb.cases*) *auto*

lemma *list-emb-refl*:
assumes ⋀*x*. *x* ∈ *set xs* ⟹ *P x x*
shows *list-emb P xs xs*
using *assms* **by** (*induct xs*) *auto*

lemma *list-emb-Cons-Nil* [*simp*]: *list-emb P* (*x*#*xs*) [] = *False*
proof
show *False* **if** *list-emb P* (*x*#*xs*) []
using *list-emb-Nil2* [*OF that*] **by** *simp*
show *list-emb P* (*x*#*xs*) [] **if** *False*
using *that* **..**
qed

lemma *list-emb-append2* [*intro*]: *list-emb P xs ys* ⟹ *list-emb P xs* (*zs* @ *ys*)
by (*induct zs*) *auto*

lemma *list-emb-prefix* [*intro*]:
assumes *list-emb P xs ys* **shows** *list-emb P xs* (*ys* @ *zs*)
using *assms*
by (*induct arbitrary*: *zs*) *auto*

lemma *list-emb-ConsD*:
assumes *list-emb P* (*x*#*xs*) *ys*
shows ∃ *us v vs*. *ys* = *us* @ *v* # *vs* ∧ *P x v* ∧ *list-emb P xs vs*
using *assms*
proof (*induct x* ≡ *x* # *xs ys arbitrary*: *x xs*)
case *list-emb-Cons*
then show *?case* **by** (*metis append-Cons*)
next
case (*list-emb-Cons2 x y xs ys*)
then show *?case* **by** *blast*
qed

lemma *list-emb-appendD*:
assumes *list-emb P* (*xs* @ *ys*) *zs*
shows ∃ *us vs*. *zs* = *us* @ *vs* ∧ *list-emb P xs us* ∧ *list-emb P ys vs*
using *assms*
proof (*induction xs arbitrary*: *ys zs*)
case *Nil* **then show** *?case* **by** *auto*
next
case (*Cons x xs*)
then obtain *us v vs* **where**
zs: *zs* = *us* @ *v* # *vs* **and** *p*: *P x v* **and** *lh*: *list-emb P* (*xs* @ *ys*) *vs*
by (*auto dest*: *list-emb-ConsD*)
obtain sk_0 :: *'a list* ⇒ *'a list* ⇒ *'a list* **and** sk_1 :: *'a list* ⇒ *'a list* ⇒ *'a list*
where
sk: ∀ x_0 x_1. ¬ *list-emb P* (*xs* @ x_0) x_1 ∨ sk_0 x_0 x_1 @ sk_1 x_0 x_1 = x_1 ∧ *list-emb P xs* (sk_0 x_0 x_1) ∧ *list-emb P* x_0 (sk_1 x_0 x_1)
using *Cons*(*1*) **by** (*metis* (*no-types*))
hence ∀ x_2. *list-emb P* (*x* # *xs*) (x_2 @ *v* # sk_0 *ys vs*) **using** *p lh* **by** *auto*
thus *?case* **using** *lh zs sk* **by** (*metis* (*no-types*) *append-Cons append-assoc*)
qed

lemma *list-emb-strict-suffix*:
assumes *list-emb P xs ys* **and** *strict-suffix ys zs*
shows *list-emb P xs zs*
using *assms*(*2*) **and** *list-emb-append2* [*OF assms*(*1*)] **by** (*auto simp*: *strict-suffix-def suffix-def*)

lemma *list-emb-suffix*:
assumes *list-emb P xs ys* **and** *suffix ys zs*
shows *list-emb P xs zs*
using *assms* **and** *list-emb-strict-suffix*
unfolding *strict-suffix-reflclp-conv*[*symmetric*] **by** *auto*

lemma *list-emb-length*: *list-emb P xs ys* ⟹ *length xs* ≤ *length ys*
by (*induct rule*: *list-emb.induct*) *auto*

lemma *list-emb-trans*:
assumes ⋀*x y z*. ⟦*x* ∈ *set xs*; *y* ∈ *set ys*; *z* ∈ *set zs*; *P x y*; *P y z*⟧ ⟹ *P x z*

```
  shows ⟦list-emb P xs ys; list-emb P ys zs⟧ ⟹ list-emb P xs zs
proof −
  assume list-emb P xs ys and list-emb P ys zs
  then show list-emb P xs zs using assms
  proof (induction arbitrary: zs)
    case list-emb-Nil show ?case by blast
  next
    case (list-emb-Cons xs ys y)
    from list-emb-ConsD [OF ‹list-emb P (y#ys) zs›] obtain us v vs
      where zs: zs = us @ v # vs and P== y v and list-emb P ys vs by blast
    then have list-emb P ys (v#vs) by blast
    then have list-emb P ys zs unfolding zs by (rule list-emb-append2)
   from list-emb-Cons.IH [OF this] and list-emb-Cons.prems show ?case by auto
  next
    case (list-emb-Cons2 x y xs ys)
    from list-emb-ConsD [OF ‹list-emb P (y#ys) zs›] obtain us v vs
      where zs: zs = us @ v # vs and P y v and list-emb P ys vs by blast
    with list-emb-Cons2 have list-emb P xs vs by auto
    moreover have P x v
    proof −
      from zs have v ∈ set zs by auto
      moreover have x ∈ set (x#xs) and y ∈ set (y#ys) by simp-all
      ultimately show ?thesis
        using ‹P x y› and ‹P y v› and list-emb-Cons2
        by blast
    qed
    ultimately have list-emb P (x#xs) (v#vs) by blast
    then show ?case unfolding zs by (rule list-emb-append2)
  qed
qed

lemma list-emb-set:
  assumes list-emb P xs ys and x ∈ set xs
  obtains y where y ∈ set ys and P x y
  using assms by (induct) auto

lemma list-emb-Cons-iff1 [simp]:
  assumes P x y
  shows   list-emb P (x#xs) (y#ys) ⟷ list-emb P xs ys
  using assms by (subst list-emb.simps) (auto dest: list-emb-ConsD)

lemma list-emb-Cons-iff2 [simp]:
  assumes ¬P x y
  shows   list-emb P (x#xs) (y#ys) ⟷ list-emb P (x#xs) ys
  using assms by (subst list-emb.simps) auto

lemma list-emb-code [code]:
  list-emb P [] ys ⟷ True
  list-emb P (x#xs) [] ⟷ False
```

list-emb P (x#xs) (y#ys) ⟷ (if P x y then list-emb P xs ys else list-emb P (x#xs) ys)
by *simp-all*

58.9 Subsequences (special case of homeomorphic embedding)

abbreviation *subseq* :: *'a list ⇒ 'a list ⇒ bool*
where *subseq xs ys ≡ list-emb (=) xs ys*

definition *strict-subseq* **where** *strict-subseq xs ys ⟷ xs ≠ ys ∧ subseq xs ys*

lemma *subseq-Cons2*: *subseq xs ys ⟹ subseq (x#xs) (x#ys)* **by** *auto*

lemma *subseq-same-length*:
assumes *subseq xs ys* **and** *length xs = length ys* **shows** *xs = ys*
using *assms* **by** (*induct*) (*auto dest*: *list-emb-length*)

lemma *not-subseq-length* [*simp*]: *length ys < length xs ⟹ ¬ subseq xs ys*
by (*metis list-emb-length linorder-not-less*)

lemma *subseq-Cons'*: *subseq (x#xs) ys ⟹ subseq xs ys*
by (*induct xs*, *simp*, *blast dest*: *list-emb-ConsD*)

lemma *subseq-Cons2'*:
assumes *subseq (x#xs) (x#ys)* **shows** *subseq xs ys*
using *assms* **by** (*cases*) (*rule subseq-Cons'*)

lemma *subseq-Cons2-neq*:
assumes *subseq (x#xs) (y#ys)*
shows *x ≠ y ⟹ subseq (x#xs) ys*
using *assms* **by** (*cases*) *auto*

lemma *subseq-Cons2-iff* [*simp*]:
subseq (x#xs) (y#ys) = (if x = y then subseq xs ys else subseq (x#xs) ys)
by *simp*

lemma *subseq-append'*: *subseq (zs @ xs) (zs @ ys) ⟷ subseq xs ys*
by (*induct zs*) *simp-all*

global-interpretation *subseq-order*: *ordering subseq strict-subseq*
proof
show ‹*subseq xs xs*› **for** *xs* :: ‹*'a list*›
using *refl* **by** (*rule list-emb-refl*)
show ‹*subseq xs zs*› **if** ‹*subseq xs ys*› **and** ‹*subseq ys zs*›
for *xs ys zs* :: ‹*'a list*›
using *trans* [*OF refl*] *that* **by** (*rule list-emb-trans*) *simp*
show ‹*xs = ys*› **if** ‹*subseq xs ys*› **and** ‹*subseq ys xs*›
for *xs ys* :: ‹*'a list*›

using *that* **proof** *induction*
case *list-emb-Nil*
from *list-emb-Nil2* [*OF this*] **show** *?case* **by** *simp*
next
case *list-emb-Cons2*
then show *?case* **by** *simp*
next
case *list-emb-Cons*
hence *False* **using** *subseq-Cons'* **by** *fastforce*
then show *?case* **..**
qed
show ‹*strict-subseq xs ys* ⟷ *subseq xs ys* ∧ *xs* ≠ *ys*›
for *xs ys* :: ‹*'a list*›
by (*auto simp*: *strict-subseq-def*)
qed

interpretation *subseq-order*: *order subseq strict-subseq*
by (*rule ordering-orderI*) *standard*

lemma *in-set-subseqs* [*simp*]: *xs* ∈ *set* (*subseqs ys*) ⟷ *subseq xs ys*
proof
assume *xs* ∈ *set* (*subseqs ys*)
thus *subseq xs ys*
by (*induction ys arbitrary*: *xs*) (*auto simp*: *Let-def*)
next
have [*simp*]: [] ∈ *set* (*subseqs ys*) **for** *ys* :: *'a list*
by (*induction ys*) (*auto simp*: *Let-def*)
assume *subseq xs ys*
thus *xs* ∈ *set* (*subseqs ys*)
by (*induction xs ys rule*: *list-emb.induct*) (*auto simp*: *Let-def*)
qed

lemma *set-subseqs-eq*: *set* (*subseqs ys*) = {*xs*. *subseq xs ys*}
by *auto*

lemma *subseq-append-le-same-iff*: *subseq* (*xs* @ *ys*) *ys* ⟷ *xs* = []
by (*auto dest*: *list-emb-length*)

lemma *subseq-singleton-left*: *subseq* [*x*] *ys* ⟷ *x* ∈ *set ys*
by (*fastforce dest*: *list-emb-ConsD split-list-last*)

lemma *list-emb-append-mono*:
⟦ *list-emb P xs xs'*; *list-emb P ys ys'* ⟧ ⟹ *list-emb P* (*xs*@*ys*) (*xs'*@*ys'*)
by (*induct rule*: *list-emb.induct*) *auto*

lemma *prefix-imp-subseq* [*intro*]: *prefix xs ys* ⟹ *subseq xs ys*
by (*auto simp*: *prefix-def*)

lemma *suffix-imp-subseq* [*intro*]: *suffix xs ys* ⟹ *subseq xs ys*

by (*auto simp*: *suffix-def*)

a subsequence of a sorted list

lemma *sorted-subset-imp-subseq*:
fixes *xs* :: *'a*::*order list*
assumes *set xs* ⊆ *set ys sorted-wrt* (<) *xs sorted-wrt* (≤) *ys*
shows *subseq xs ys*
using *assms*
proof (*induction xs arbitrary*: *ys*)
case *Nil*
then show *?case*
by *auto*
next
case (*Cons x xs*)
then have *x* ∈ *set ys*
by *auto*
then obtain *us vs* **where** §: *ys* = *us* @ [*x*] @ *vs*
by (*metis append.left-neutral append-eq-Cons-conv split-list*)
moreover
have *set xs* ⊆ *set vs*
using *Cons.prems* **by** (*fastforce simp*: § *sorted-wrt-append*)
with *Cons* **have** *subseq xs vs*
by (*metis* § *sorted-wrt.simps*(*2*) *sorted-wrt-append*)
ultimately show *?case*
by *auto*
qed

58.10 Appending elements

lemma *subseq-append* [*simp*]:
subseq (*xs* @ *zs*) (*ys* @ *zs*) ⟷ *subseq xs ys* (**is** *?l* = *?r*)
proof
have *xs'* = *xs* @ *zs* ∧ *ys'* = *ys* @ *zs* ⟶ *subseq xs ys*
if *subseq xs' ys'* **for** *xs' ys' xs ys zs* :: *'a list*
using *that*
proof (*induct arbitrary*: *xs ys zs*)
case *list-emb-Nil*
show *?case* **by** *simp*
next
case (*list-emb-Cons xs' ys' x*)
have *?case* **if** *ys* = []
using *list-emb-Cons*(*1*) *that* **by** *auto*
moreover
have *?case* **if** *ys* = *x*#*us* **for** *us*
using *list-emb-Cons*(*2*) *that* **by** (*simp add*: *list-emb.list-emb-Cons*)
ultimately show *?case*
by (*auto simp*: *Cons-eq-append-conv*)
next
case (*list-emb-Cons2 x y xs' ys'*)
have *?case* **if** *xs* = []

```
    using list-emb-Cons2(1) that by auto
  moreover
  have ?case if xs = x#us ys = x#vs for us vs
    using list-emb-Cons2 that by auto
  moreover
  have ?case  if xs = x#us ys = [] for us
    using list-emb-Cons2(2) that by bestsimp
  ultimately show ?case
    using ‹x = y› by (auto simp: Cons-eq-append-conv)
 qed
 then show ?l ⟹ ?r by blast
 show ?r ⟹ ?l by (metis list-emb-append-mono subseq-order.order-refl)
qed

lemma subseq-append-iff:
 subseq xs (ys @ zs) ⟷ (∃ xs1 xs2. xs = xs1 @ xs2 ∧ subseq xs1 ys ∧ subseq xs2
zs)
 (is ?lhs = ?rhs)
proof
 assume ?lhs thus ?rhs
 proof (induction xs ys @ zs arbitrary: ys zs rule: list-emb.induct)
   case (list-emb-Cons xs ws y ys zs)
  from list-emb-Cons(2)[of tl ys zs] and list-emb-Cons(2)[of [] tl zs] and list-emb-Cons(1,3)
   show ?case by (cases ys) auto
 next
   case (list-emb-Cons2 x y xs ws ys zs)
   from list-emb-Cons2(3)[of tl ys zs] and list-emb-Cons2(3)[of [] tl zs]
     and list-emb-Cons2(1,2,4)
   show ?case by (cases ys) (auto simp: Cons-eq-append-conv)
 qed auto
qed (auto intro: list-emb-append-mono)

lemma subseq-appendE [case-names append]:
 assumes subseq xs (ys @ zs)
 obtains xs1 xs2 where xs = xs1 @ xs2 subseq xs1 ys subseq xs2 zs
 using assms by (subst (asm) subseq-append-iff) auto

lemma subseq-drop-many: subseq xs ys ⟹ subseq xs (zs @ ys)
 by (induct zs) auto

lemma subseq-rev-drop-many: subseq xs ys ⟹ subseq xs (ys @ zs)
 by (metis append-Nil2 list-emb-Nil list-emb-append-mono)
```

58.11 Relation to standard list operations

```
lemma subseq-map:
 assumes subseq xs ys shows subseq (map f xs) (map f ys)
 using assms by (induct) auto
```

lemma *subseq-filter-left* [*simp*]: *subseq* (*filter P xs*) *xs*
by (*induct xs*) *auto*

lemma *subseq-filter* [*simp*]:
assumes *subseq xs ys* **shows** *subseq* (*filter P xs*) (*filter P ys*)
using *assms* **by** *induct auto*

lemma *subseq-conv-nths*: *subseq xs ys* ⟷ (∃ *N*. *xs* = *nths ys N*)
(**is** *?L* = *?R*)
proof
show *?R* **if** *?L* **using** *that*
proof (*induct*)
case *list-emb-Nil*
show *?case* **by** (*metis nths-empty*)
next
case (*list-emb-Cons xs ys x*)
then obtain *N* **where** *xs* = *nths ys N* **by** *blast*
then have *xs* = *nths* (*x*#*ys*) (*Suc* ‘ *N*)
by (*clarsimp simp add*: *nths-Cons inj-image-mem-iff*)
then show *?case* **by** *blast*
next
case (*list-emb-Cons2 x y xs ys*)
then obtain *N* **where** *xs* = *nths ys N* **by** *blast*
then have *x*#*xs* = *nths* (*x*#*ys*) (*insert 0* (*Suc* ‘ *N*))
by (*clarsimp simp add*: *nths-Cons inj-image-mem-iff*)
moreover from *list-emb-Cons2* **have** *x* = *y* **by** *simp*
ultimately show *?case* **by** *blast*
qed
show *?L* **if** *?R*
proof −
from *that* **obtain** *N* **where** *xs* = *nths ys N* **..**
moreover have *subseq* (*nths ys N*) *ys*
proof (*induct ys arbitrary*: *N*)
case *Nil*
show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*auto simp*: *nths-Cons*)
qed
ultimately show *?thesis* **by** *simp*
qed
qed

58.12 Contiguous sublists

58.12.1 *sublist*

definition *sublist* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
sublist xs ys = (∃ *ps ss*. *ys* = *ps* @ *xs* @ *ss*)

definition *strict-sublist* :: *'a list* ⇒ *'a list* ⇒ *bool* **where**
strict-sublist xs ys ⟷ *sublist xs ys* ∧ *xs* ≠ *ys*

interpretation *sublist-order*: *order sublist strict-sublist*
proof
fix *xs ys zs* :: *'a list*
assume *sublist xs ys sublist ys zs*
then obtain *xs1 xs2 ys1 ys2* **where** *ys* = *xs1* @ *xs* @ *xs2 zs* = *ys1* @ *ys* @ *ys2*
by (*auto simp*: *sublist-def*)
hence *zs* = (*ys1* @ *xs1*) @ *xs* @ (*xs2* @ *ys2*) **by** *simp*
thus *sublist xs zs* **unfolding** *sublist-def* **by** *blast*
next
fix *xs ys* :: *'a list*
show *xs* = *ys* **if** *sublist xs ys sublist ys xs*
proof −
from *that* **obtain** *as bs cs ds* **where** *xs*: *xs* = *as* @ *ys* @ *bs* **and** *ys*: *ys* = *cs* @ *xs* @ *ds*
by (*auto simp*: *sublist-def*)
have *xs* = *as* @ *cs* @ *xs* @ *ds* @ *bs* **by** (*subst xs*, *subst ys*) *auto*
also have *length* ... = *length as* + *length cs* + *length xs* + *length bs* + *length ds*
by *simp*
finally have *as* = [] *bs* = [] **by** *simp-all*
with *xs* **show** *?thesis* **by** *simp*
qed
thus *strict-sublist xs ys* ⟷ (*sublist xs ys* ∧ ¬ *sublist ys xs*)
by (*auto simp*: *strict-sublist-def*)
qed (*auto simp*: *strict-sublist-def sublist-def intro*: *exI*[*of* - []])

lemma *sublist-Nil-left* [*simp*, *intro*]: *sublist* [] *ys*
by (*auto simp*: *sublist-def*)

lemma *sublist-Cons-Nil* [*simp*]: ¬*sublist* (*x*#*xs*) []
by (*auto simp*: *sublist-def*)

lemma *sublist-Nil-right* [*simp*]: *sublist xs* [] ⟷ *xs* = []
by (*cases xs*) *auto*

lemma *sublist-appendI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs* @ *ss*)
by (*auto simp*: *sublist-def*)

lemma *sublist-append-leftI* [*simp*, *intro*]: *sublist xs* (*ps* @ *xs*)
by (*auto simp*: *sublist-def intro*: *exI*[*of* - []])

lemma *sublist-append-rightI* [*simp*, *intro*]: *sublist xs* (*xs* @ *ss*)
by (*metis append-eq-append-conv2 sublist-appendI*)

lemma *sublist-altdef*: *sublist xs ys* ⟷ (∃ *ys'*. *prefix ys' ys* ∧ *suffix xs ys'*)
by (*metis append-assoc prefix-def sublist-def suffix-def*)

lemma *sublist-altdef'*: *sublist xs ys* $\longleftrightarrow$ ($\exists$ *ys'*. *suffix ys' ys* $\wedge$ *prefix xs ys'*)
by (*metis prefixE prefixI sublist-appendI sublist-def suffixE suffixI*)

lemma *sublist-Cons-right*: *sublist xs* (*y # ys*) $\longleftrightarrow$ *prefix xs* (*y # ys*) $\vee$ *sublist xs ys*
by (*auto simp*: *sublist-def prefix-def Cons-eq-append-conv*)

lemma *sublist-code* [*code*]:
sublist [] *ys* $\longleftrightarrow$ *True*
sublist (*x # xs*) [] $\longleftrightarrow$ *False*
sublist (*x # xs*) (*y # ys*) $\longleftrightarrow$ *prefix* (*x # xs*) (*y # ys*) $\vee$ *sublist* (*x # xs*) *ys*
by (*simp-all add*: *sublist-Cons-right*)

lemma *sublist-append*:
sublist xs (*ys* @ *zs*) $\longleftrightarrow$
sublist xs ys $\vee$ *sublist xs zs* $\vee$ ($\exists$ *xs1 xs2*. *xs* = *xs1* @ *xs2* $\wedge$ *suffix xs1 ys* $\wedge$ *prefix xs2 zs*)
by (*auto simp*: *sublist-altdef prefix-append suffix-append*)

lemma *map-mono-sublist*:
assumes *sublist xs ys*
shows *sublist* (*map f xs*) (*map f ys*)
proof −
from *assms* **obtain** *xs1 xs2* **where** *ys*: *ys* = *xs1* @ *xs* @ *xs2*
by (*auto simp*: *sublist-def*)
have *map f ys* = *map f xs1* @ *map f xs* @ *map f xs2*
by (*auto simp*: *ys*)
thus *?thesis*
by (*auto simp*: *sublist-def*)
qed

lemma *sublist-length-le*: *sublist xs ys* $\Longrightarrow$ *length xs* $\leq$ *length ys*
by (*auto simp add*: *sublist-def*)

lemma *set-mono-sublist*: *sublist xs ys* $\Longrightarrow$ *set xs* $\subseteq$ *set ys*
by (*auto simp add*: *sublist-def*)

lemma *prefix-imp-sublist* [*simp*, *intro*]: *prefix xs ys* $\Longrightarrow$ *sublist xs ys*
by (*auto simp*: *sublist-def prefix-def intro*: *exI*[*of* - []])

lemma *suffix-imp-sublist* [*simp*, *intro*]: *suffix xs ys* $\Longrightarrow$ *sublist xs ys*
by (*auto simp*: *sublist-def suffix-def intro*: *exI*[*of* - []])

lemma *sublist-take* [*simp*, *intro*]: *sublist* (*take n xs*) *xs*
by (*rule prefix-imp-sublist*[*OF take-is-prefix*])

lemma *sublist-takeWhile* [*simp*, *intro*]: *sublist* (*takeWhile P xs*) *xs*
by (*rule prefix-imp-sublist*[*OF takeWhile-is-prefix*])

lemma *sublist-drop* [*simp*, *intro*]: *sublist* (*drop n xs*) *xs*
by (*rule suffix-imp-sublist*[*OF suffix-drop*])

lemma *sublist-dropWhile* [*simp*, *intro*]: *sublist* (*dropWhile P xs*) *xs*
by (*rule suffix-imp-sublist*[*OF suffix-dropWhile*])

lemma *sublist-tl* [*simp*, *intro*]: *sublist* (*tl xs*) *xs*
by (*rule suffix-imp-sublist*) (*simp-all add*: *suffix-drop*)

lemma *sublist-butlast* [*simp*, *intro*]: *sublist* (*butlast xs*) *xs*
by (*rule prefix-imp-sublist*) (*simp-all add*: *prefixeq-butlast*)

lemma *sublist-rev* [*simp*]: *sublist* (*rev xs*) (*rev ys*) = *sublist xs ys*
proof
assume *sublist* (*rev xs*) (*rev ys*)
then obtain *as bs* **where** *rev ys* = *as* @ *rev xs* @ *bs*
by (*auto simp*: *sublist-def*)
also have *rev* ... = *rev bs* @ *xs* @ *rev as* **by** *simp*
finally show *sublist xs ys* **by** *simp*
next
assume *sublist xs ys*
then obtain *as bs* **where** *ys* = *as* @ *xs* @ *bs*
by (*auto simp*: *sublist-def*)
also have *rev* ... = *rev bs* @ *rev xs* @ *rev as* **by** *simp*
finally show *sublist* (*rev xs*) (*rev ys*) **by** *simp*
qed

lemma *sublist-rev-left*: *sublist* (*rev xs*) *ys* = *sublist xs* (*rev ys*)
by (*subst sublist-rev* [*symmetric*]) (*simp only*: *rev-rev-ident*)

lemma *sublist-rev-right*: *sublist xs* (*rev ys*) = *sublist* (*rev xs*) *ys*
by (*subst sublist-rev* [*symmetric*]) (*simp only*: *rev-rev-ident*)

lemma *snoc-sublist-snoc*:
sublist (*xs* @ [*x*]) (*ys* @ [*y*]) ⟷
(*x* = *y* ∧ *suffix xs ys* ∨ *sublist* (*xs* @ [*x*]) *ys*)
by (*subst* (*1 2*) *sublist-rev* [*symmetric*])
(*simp del*: *sublist-rev add*: *sublist-Cons-right suffix-to-prefix*)

lemma *sublist-snoc*:
sublist xs (*ys* @ [*y*]) ⟷ *suffix xs* (*ys* @ [*y*]) ∨ *sublist xs ys*
by (*subst* (*1 2*) *sublist-rev* [*symmetric*])
(*simp del*: *sublist-rev add*: *sublist-Cons-right suffix-to-prefix*)

lemma *sublist-imp-subseq* [*intro*]: *sublist xs ys* ⟹ *subseq xs ys*
by (*auto simp*: *sublist-def*)

lemma *sublist-map-rightE*:

assumes *sublist xs* (*map f ys*)
shows ∃ *xs'*. *sublist xs' ys* ∧ *xs* = *map f xs'*
proof −
note *takedrop* = *sublist-take sublist-drop*
define *n* **where** *n* = (*length ys* − *length xs*)
from *assms* **obtain** *xs1 xs2* **where** *xs12*: *map f ys* = *xs1* @ *xs* @ *xs2*
by (*auto simp*: *sublist-def*)
define *n* **where** *n* = *length xs1*
have *xs* = *take* (*length xs*) (*drop n* (*map f ys*))
by (*simp add*: *xs12 n-def*)
thus *?thesis*
by (*intro exI*[*of - take* (*length xs*) (*drop n ys*)])
(*auto simp*: *take-map drop-map intro*!: *takedrop*[*THEN sublist-order.order.trans*])
qed

lemma *sublist-remdups-adj*:
assumes *sublist xs ys*
shows *sublist* (*remdups-adj xs*) (*remdups-adj ys*)
proof −
from *assms* **obtain** *xs1 xs2* **where** *ys*: *ys* = *xs1* @ *xs* @ *xs2*
by (*auto simp*: *sublist-def*)
have *suffix* (*remdups-adj* (*xs* @ *xs2*)) (*remdups-adj* (*xs1* @ *xs* @ *xs2*))
by (*rule suffix-remdups-adj*, *rule suffix-appendI*) *auto*
then obtain *zs1* **where** *zs1*: *remdups-adj* (*xs1* @ *xs* @ *xs2*) = *zs1* @ *remdups-adj* (*xs* @ *xs2*)
by (*auto simp*: *suffix-def*)
have *prefix* (*remdups-adj xs*) (*remdups-adj* (*xs* @ *xs2*))
by (*intro prefix-remdups-adj*) *auto*
then obtain *zs2* **where** *zs2*: *remdups-adj* (*xs* @ *xs2*) = *remdups-adj xs* @ *zs2*
by (*auto simp*: *prefix-def*)
show *?thesis*
by (*simp add*: *ys zs1 zs2*)
qed

58.12.2 *sublists*

primrec *sublists* :: *'a list* ⇒ *'a list list* **where**
sublists [] = [[]]
| *sublists* (*x* # *xs*) = *sublists xs* @ *map* ((#) *x*) (*prefixes xs*)

lemma *in-set-sublists* [*simp*]: *xs* ∈ *set* (*sublists ys*) ⟷ *sublist xs ys*
by (*induction ys arbitrary*: *xs*) (*auto simp*: *sublist-Cons-right prefix-Cons*)

lemma *set-sublists-eq*: *set* (*sublists xs*) = {*ys*. *sublist ys xs*}
by *auto*

lemma *length-sublists* [*simp*]: *length* (*sublists xs*) = *Suc* (*length xs* ∗ *Suc* (*length xs*) *div 2*)
by (*induction xs*) *simp-all*

58.13 Parametricity

context includes *lifting-syntax*
begin

private lemma *prefix-primrec*:
prefix = *rec-list* (λ*xs*. *True*) (λ*x xs xsa ys*.
case *ys of* [] ⇒ *False* | *y* # *ys* ⇒ *x* = *y* ∧ *xsa ys*)
proof (*intro ext*, *goal-cases*)
case (*1 xs ys*)
show *?case* **by** (*induction xs arbitrary*: *ys*) (*auto simp*: *prefix-Cons split*: *list.splits*)
qed

private lemma *sublist-primrec*:
sublist = (λ*xs ys*. *rec-list* (λ*xs*. *xs* = []) (λ*y ys ysa xs*. *prefix xs* (*y* # *ys*) ∨ *ysa xs*) *ys xs*)
proof (*intro ext*, *goal-cases*)
case (*1 xs ys*)
show *?case* **by** (*induction ys*) (*auto simp*: *sublist-Cons-right*)
qed

private lemma *list-emb-primrec*:
list-emb = (λ*uu l' l*. *rec-list* (λ*P xs*. *List.null xs*) (λ*y ys ysa P xs*. *case xs of* [] ⇒ *True*
| *x* # *xs* ⇒ *if P x y then ysa P xs else ysa P* (*x* # *xs*)) *l uu l'*)
proof (*intro ext*, *goal-cases*)
case (*1 P xs ys*)
show *?case*
by (*induction ys arbitrary*: *xs*)
(*auto simp*: *list-emb-code split*: *list.splits*)
qed

lemma *prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *prefix prefix*
unfolding *prefix-primrec* **by** *transfer-prover*

lemma *suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *suffix suffix*
unfolding *suffix-to-prefix* [*abs-def*] **by** *transfer-prover*

lemma *sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *sublist sublist*
unfolding *sublist-primrec* **by** *transfer-prover*

lemma *parallel-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *parallel parallel*

unfolding *parallel-def* **by** *transfer-prover*

lemma *list-emb-transfer* [*transfer-rule*]:
((*A* ===> *A* ===> (=)) ===> *list-all2 A* ===> *list-all2 A* ===> (=)) *list-emb list-emb*
unfolding *list-emb-primrec* **by** *transfer-prover*

lemma *strict-prefix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-prefix strict-prefix*
unfolding *strict-prefix-def* **by** *transfer-prover*

lemma *strict-suffix-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-suffix strict-suffix*
unfolding *strict-suffix-def* **by** *transfer-prover*

lemma *strict-subseq-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-subseq strict-subseq*
unfolding *strict-subseq-def* **by** *transfer-prover*

lemma *strict-sublist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2 A* ===> (=)) *strict-sublist strict-sublist*
unfolding *strict-sublist-def* **by** *transfer-prover*

lemma *prefixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *prefixes prefixes*
unfolding *prefixes-def* **by** *transfer-prover*

lemma *suffixes-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *suffixes suffixes*
unfolding *suffixes-def* **by** *transfer-prover*

lemma *sublists-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> *list-all2* (*list-all2 A*)) *sublists sublists*
unfolding *sublists-def* **by** *transfer-prover*

end

end

59 Linear Temporal Logic on Streams

theory *Linear-Temporal-Logic-on-Streams*
imports *Stream Sublist Extended-Nat Infinite-Set*
begin

60 Preliminaries

lemma *shift-prefix*:
assumes xl @− $xs = yl$ @− ys **and** $length\ xl \leq length\ yl$
shows $prefix\ xl\ yl$
using *assms* **proof**(*induct xl arbitrary*: *yl xs ys*)
case (*Cons x xl yl xs ys*)
thus *?case* **by** (*cases yl*) *auto*
qed *auto*

lemma *shift-prefix-cases*:
assumes xl @− $xs = yl$ @− ys
shows $prefix\ xl\ yl \vee prefix\ yl\ xl$
using *shift-prefix*[*OF assms*]
by (*cases length xl* ≤ *length yl*) (*metis*, *metis assms nat-le-linear shift-prefix*)

61 Linear temporal logic

Propositional connectives:

abbreviation (*input*) *IMPL* (**infix** ‹*impl*› *60*)
where $\varphi\ impl\ \psi \equiv \lambda\ xs.\ \varphi\ xs \longrightarrow \psi\ xs$

abbreviation (*input*) *OR* (**infix** ‹*or*› *60*)
where $\varphi\ or\ \psi \equiv \lambda\ xs.\ \varphi\ xs \vee \psi\ xs$

abbreviation (*input*) *AND* (**infix** ‹*aand*› *60*)
where $\varphi\ aand\ \psi \equiv \lambda\ xs.\ \varphi\ xs \wedge \psi\ xs$

abbreviation (*input*) *not* **where** $not\ \varphi \equiv \lambda\ xs.\ \neg\ \varphi\ xs$

abbreviation (*input*) $true \equiv \lambda\ xs.\ True$

abbreviation (*input*) $false \equiv \lambda\ xs.\ False$

lemma *impl-not-or*: $\varphi\ impl\ \psi = (not\ \varphi)\ or\ \psi$
by *blast*

lemma *not-or*: $not\ (\varphi\ or\ \psi) = (not\ \varphi)\ aand\ (not\ \psi)$
by *blast*

lemma *not-aand*: $not\ (\varphi\ aand\ \psi) = (not\ \varphi)\ or\ (not\ \psi)$
by *blast*

lemma *non-not*[*simp*]: *not* (*not* φ) = φ **by** *simp*

Temporal (LTL) connectives:

fun *holds* **where** *holds P xs* $\longleftrightarrow$ *P* (*shd xs*)
fun *nxt* **where** *nxt* φ *xs* = φ (*stl xs*)

definition *HLD s* = *holds* ($\lambda x.\ x \in s$)

abbreviation *HLD-nxt* (**infixr** ‹·› *65*) **where**
s · *P* ≡ *HLD s aand nxt P*

context
notes [[*inductive-internals*]]
begin

inductive *ev* **for** φ **where**
base: φ *xs* $\Longrightarrow$ *ev* φ *xs*
|
step: *ev* φ (*stl xs*) $\Longrightarrow$ *ev* φ *xs*

coinductive *alw* **for** φ **where**
alw: ⟦φ *xs*; *alw* φ (*stl xs*)⟧ $\Longrightarrow$ *alw* φ *xs*

— weak until:
coinductive *UNTIL* (**infix** ‹*until*› *60*) **for** φ ψ **where**
base: ψ *xs* $\Longrightarrow$ (φ *until* ψ) *xs*
|
step: ⟦φ *xs*; (φ *until* ψ) (*stl xs*)⟧ $\Longrightarrow$ (φ *until* ψ) *xs*

end

lemma *holds-mono*:
assumes *holds*: *holds P xs* **and** *0*: $\bigwedge$ *x*. *P x* $\Longrightarrow$ *Q x*
shows *holds Q xs*
using *assms* **by** *auto*

lemma *holds-aand*:
(*holds P aand holds Q*) *steps* $\longleftrightarrow$ *holds* (λ *step*. *P step* $\wedge$ *Q step*) *steps* **by** *auto*

lemma *HLD-iff*: *HLD s* ω $\longleftrightarrow$ *shd* ω $\in$ *s*
by (*simp add*: *HLD-def*)

lemma *HLD-Stream*[*simp*]: *HLD X* (*x* ## ω) $\longleftrightarrow$ *x* $\in$ *X*
by (*simp add*: *HLD-iff*)

lemma *nxt-mono*:
assumes *nxt*: *nxt* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *nxt* ψ *xs*

using *assms* **by** *auto*

declare *ev.intros*[*intro*]
declare *alw.cases*[*elim*]

lemma *ev-induct-strong*[*consumes 1*, *case-names base step*]:
ev φ *x* $\Longrightarrow$ ($\bigwedge$*xs*. φ *xs* $\Longrightarrow$ *P xs*) $\Longrightarrow$ ($\bigwedge$*xs*. *ev* φ (*stl xs*) $\Longrightarrow$ $\neg$ φ *xs* $\Longrightarrow$ *P* (*stl xs*) $\Longrightarrow$ *P xs*) $\Longrightarrow$ *P x*
by (*induct rule*: *ev.induct*) *auto*

lemma *alw-coinduct*[*consumes 1*, *case-names alw stl*]:
X x $\Longrightarrow$ ($\bigwedge$*x*. *X x* $\Longrightarrow$ φ *x*) $\Longrightarrow$ ($\bigwedge$*x*. *X x* $\Longrightarrow$ $\neg$ *alw* φ (*stl x*) $\Longrightarrow$ *X* (*stl x*)) $\Longrightarrow$ *alw* φ *x*
using *alw.coinduct*[*of X x* φ] **by** *auto*

lemma *ev-mono*:
assumes *ev*: *ev* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *ev* ψ *xs*
using *ev* **by** *induct* (*auto simp*: *0*)

lemma *alw-mono*:
assumes *alw*: *alw* φ *xs* **and** *0*: $\bigwedge$ *xs*. φ *xs* $\Longrightarrow$ ψ *xs*
shows *alw* ψ *xs*
using *alw* **by** *coinduct* (*auto simp*: *0*)

lemma *until-monoL*:
assumes *until*: (φ*1 until* ψ) *xs* **and** *0*: $\bigwedge$ *xs*. φ*1 xs* $\Longrightarrow$ φ*2 xs*
shows (φ*2 until* ψ) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-monoR*:
assumes *until*: (φ *until* ψ*1*) *xs* **and** *0*: $\bigwedge$ *xs*. ψ*1 xs* $\Longrightarrow$ ψ*2 xs*
shows (φ *until* ψ*2*) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-mono*:
assumes *until*: (φ*1 until* ψ*1*) *xs* **and**
0: $\bigwedge$ *xs*. φ*1 xs* $\Longrightarrow$ φ*2 xs* $\bigwedge$ *xs*. ψ*1 xs* $\Longrightarrow$ ψ*2 xs*
shows (φ*2 until* ψ*2*) *xs*
using *until* **by** *coinduct* (*auto elim*: *UNTIL.cases simp*: *0*)

lemma *until-false*: φ *until false* = *alw* φ
proof−
{fix *xs* **assume** (φ *until false*) *xs* **hence** *alw* φ *xs*
by *coinduct* (*auto elim*: *UNTIL.cases*)
}
moreover
{fix *xs* **assume** *alw* φ *xs* **hence** (φ *until false*) *xs*
by *coinduct auto*

```
  }
  ultimately show ?thesis by blast
qed

lemma ev-nxt: ev φ = (φ or nxt (ev φ))
by (rule ext) (metis ev.simps nxt.simps)

lemma alw-nxt: alw φ = (φ aand nxt (alw φ))
by (rule ext) (metis alw.simps nxt.simps)

lemma ev-ev[simp]: ev (ev φ) = ev φ
proof−
  {fix xs
   assume ev (ev φ) xs hence ev φ xs
   by induct auto
  }
  thus ?thesis by auto
qed

lemma alw-alw[simp]: alw (alw φ) = alw φ
proof−
  {fix xs
   assume alw φ xs hence alw (alw φ) xs
   by coinduct auto
  }
  thus ?thesis by auto
qed

lemma ev-shift:
assumes ev φ xs
shows ev φ (xl @− xs)
using assms by (induct xl) auto

lemma ev-imp-shift:
assumes ev φ xs  shows ∃ xl xs2. xs = xl @− xs2 ∧ φ xs2
using assms by induct (metis shift.simps(1), metis shift.simps(2) stream.collapse)+

lemma alw-ev-shift: alw φ xs1 ⟹ ev (alw φ) (xl @− xs1)
by (auto intro: ev-shift)

lemma alw-shift:
assumes alw φ (xl @− xs)
shows alw φ xs
using assms by (induct xl) auto

lemma ev-ex-nxt:
assumes ev φ xs
shows ∃ n. (nxt ^^ n) φ xs
using assms proof induct
```

case (*base xs*) **thus** *?case* **by** (*intro exI*[*of - 0*]) *auto*
next
case (*step xs*)
then obtain *n* **where** (*nxt* ^^ *n*) φ (*stl xs*) **by** *blast*
thus *?case* **by** (*intro exI*[*of - Suc n*]) (*metis funpow.simps*(*2*) *nxt.simps o-def*)
qed

lemma *alw-sdrop*:
assumes *alw* φ *xs* **shows** *alw* φ (*sdrop n xs*)
by (*metis alw-shift assms stake-sdrop*)

lemma *nxt-sdrop*: (*nxt* ^^ *n*) φ *xs* $\longleftrightarrow$ φ (*sdrop n xs*)
by (*induct n arbitrary*: *xs*) *auto*

definition *wait* φ *xs* $\equiv$ *LEAST n*. (*nxt* ^^ *n*) φ *xs*

lemma *nxt-wait*:
assumes *ev* φ *xs* **shows** (*nxt* ^^ (*wait* φ *xs*)) φ *xs*
unfolding *wait-def* **using** *ev-ex-nxt*[*OF assms*] **by**(*rule LeastI-ex*)

lemma *nxt-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: (*nxt* ^^ *n*) φ *xs* **shows** *wait* φ *xs* $\leq$ *n*
unfolding *wait-def* **using** *ev-ex-nxt*[*OF ev*] **by** (*metis Least-le nxt*)

lemma *sdrop-wait*:
assumes *ev* φ *xs* **shows** φ (*sdrop* (*wait* φ *xs*) *xs*)
using *nxt-wait*[*OF assms*] **unfolding** *nxt-sdrop* **.**

lemma *sdrop-wait-least*:
assumes *ev*: *ev* φ *xs* **and** *nxt*: φ (*sdrop n xs*) **shows** *wait* φ *xs* $\leq$ *n*
using *assms nxt-wait-least* **unfolding** *nxt-sdrop* **by** *auto*

lemma *nxt-ev*: (*nxt* ^^ *n*) φ *xs* $\Longrightarrow$ *ev* φ *xs*
by (*induct n arbitrary*: *xs*) *auto*

lemma *not-ev*: *not* (*ev* φ) = *alw* (*not* φ)
proof(*rule ext*, *safe*)
fix *xs* **assume** *not* (*ev* φ) *xs* **thus** *alw* (*not* φ) *xs*
by (*coinduct*) *auto*
next
fix *xs* **assume** *ev* φ *xs* **and** *alw* (*not* φ) *xs* **thus** *False*
by (*induct*) *auto*
qed

lemma *not-alw*: *not* (*alw* φ) = *ev* (*not* φ)
proof−
have *not* (*alw* φ) = *not* (*alw* (*not* (*not* φ))) **by** *simp*
also have ... = *ev* (*not* φ) **unfolding** *not-ev*[*symmetric*] **by** *simp*
finally show *?thesis* **.**

qed

lemma *not-ev-not*[*simp*]: *not* (*ev* (*not* φ)) = *alw* φ
unfolding *not-ev* **by** *simp*

lemma *not-alw-not*[*simp*]: *not* (*alw* (*not* φ)) = *ev* φ
unfolding *not-alw* **by** *simp*

lemma *alw-ev-sdrop*:
assumes *alw* (*ev* φ) (*sdrop m xs*)
shows *alw* (*ev* φ) *xs*
using *assms*
by *coinduct* (*metis alw-nxt ev-shift funpow-swap1 nxt.simps nxt-sdrop stake-sdrop*)

lemma *ev-alw-imp-alw-ev*:
assumes *ev* (*alw* φ) *xs* **shows** *alw* (*ev* φ) *xs*
using *assms* **by** *induct* (*metis* (*full-types*) *alw-mono ev.base*, *metis alw alw-nxt ev.step*)

lemma *alw-aand*: *alw* (φ *aand* ψ) = *alw* φ *aand alw* ψ
proof−
  {fix *xs* **assume** *alw* (φ *aand* ψ) *xs* **hence** (*alw* φ *aand alw* ψ) *xs*
   by (*auto elim*: *alw-mono*)
  }
  moreover
  {fix *xs* **assume** (*alw* φ *aand alw* ψ) *xs* **hence** *alw* (φ *aand* ψ) *xs*
   by *coinduct auto*
  }
  ultimately show *?thesis* **by** *blast*
qed

lemma *ev-or*: *ev* (φ *or* ψ) = *ev* φ *or ev* ψ
proof−
  {fix *xs* **assume** (*ev* φ *or ev* ψ) *xs* **hence** *ev* (φ *or* ψ) *xs*
   by (*auto elim*: *ev-mono*)
  }
  moreover
  {fix *xs* **assume** *ev* (φ *or* ψ) *xs* **hence** (*ev* φ *or ev* ψ) *xs*
   by *induct auto*
  }
  ultimately show *?thesis* **by** *blast*
qed

lemma *ev-alw-aand*:
assumes φ: *ev* (*alw* φ) *xs* **and** ψ: *ev* (*alw* ψ) *xs*
shows *ev* (*alw* (φ *aand* ψ)) *xs*
proof−
  obtain *xl xs1* **where** *xs1*: *xs* = *xl* @− *xs1* **and** $\varphi\varphi$: *alw* φ *xs1*
  using φ **by** (*metis ev-imp-shift*)

moreover obtain *yl ys1* **where** *xs2*: *xs* = *yl* @− *ys1* **and** *ψψ*: *alw ψ ys1*
using *ψ* **by** (*metis ev-imp-shift*)
ultimately have *0*: *xl* @− *xs1* = *yl* @− *ys1* **by** *auto*
hence *prefix xl yl* ∨ *prefix yl xl* **using** *shift-prefix-cases* **by** *auto*
thus *?thesis* **proof**
assume *prefix xl yl*
then obtain *yl1* **where** *yl*: *yl* = *xl* @ *yl1* **by** (*elim prefixE*)
have *xs1′*: *xs1* = *yl1* @− *ys1* **using** *0* **unfolding** *yl* **by** *simp*
have *alw φ ys1* **using** *φφ* **unfolding** *xs1′* **by** (*metis alw-shift*)
hence *alw* (*φ aand ψ*) *ys1* **using** *ψψ* **unfolding** *alw-aand* **by** *auto*
thus *?thesis* **unfolding** *xs2* **by** (*auto intro*: *alw-ev-shift*)
next
assume *prefix yl xl*
then obtain *xl1* **where** *xl*: *xl* = *yl* @ *xl1* **by** (*elim prefixE*)
have *ys1′*: *ys1* = *xl1* @− *xs1* **using** *0* **unfolding** *xl* **by** *simp*
have *alw ψ xs1* **using** *ψψ* **unfolding** *ys1′* **by** (*metis alw-shift*)
hence *alw* (*φ aand ψ*) *xs1* **using** *φφ* **unfolding** *alw-aand* **by** *auto*
thus *?thesis* **unfolding** *xs1* **by** (*auto intro*: *alw-ev-shift*)
qed
qed

lemma *ev-alw-alw-impl*:
assumes *ev* (*alw φ*) *xs* **and** *alw* (*alw φ impl ev ψ*) *xs*
shows *ev ψ xs*
using *assms* **by** *induct auto*

lemma *ev-alw-stl*[*simp*]: *ev* (*alw φ*) (*stl x*) ⟷ *ev* (*alw φ*) *x*
by (*metis* (*full-types*) *alw-nxt ev-nxt nxt.simps*)

lemma *alw-alw-impl-ev*:
alw (*alw φ impl ev ψ*) = (*ev* (*alw φ*) *impl alw* (*ev ψ*)) (**is** *?A* = *?B*)
proof−
{fix *xs* **assume** *?A xs* ∧ *ev* (*alw φ*) *xs* **hence** *alw* (*ev ψ*) *xs*
by *coinduct* (*auto elim*: *ev-alw-alw-impl*)
}
moreover
{fix *xs* **assume** *?B xs* **hence** *?A xs*
by *coinduct auto*
}
ultimately show *?thesis* **by** *blast*
qed

lemma *ev-alw-impl*:
assumes *ev φ xs* **and** *alw* (*φ impl ψ*) *xs* **shows** *ev ψ xs*
using *assms* **by** *induct auto*

lemma *ev-alw-impl-ev*:
assumes *ev φ xs* **and** *alw* (*φ impl ev ψ*) *xs* **shows** *ev ψ xs*
using *ev-alw-impl*[*OF assms*] **by** *simp*

lemma *alw-mp*:
assumes *alw* φ *xs* **and** *alw* (φ *impl* ψ) *xs*
shows *alw* ψ *xs*
proof−
 {assume *alw* φ *xs* $\wedge$ *alw* (φ *impl* ψ) *xs* **hence** *?thesis*
  by *coinduct auto*
 }
 thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *all-imp-alw*:
assumes $\bigwedge$ *xs*. φ *xs* **shows** *alw* φ *xs*
proof−
 {assume $\forall$ *xs*. φ *xs*
  hence *?thesis* **by** *coinduct auto*
 }
 thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *alw-impl-ev-alw*:
assumes *alw* (φ *impl ev* ψ) *xs*
shows *alw* (*ev* φ *impl ev* ψ) *xs*
using *assms* **by** *coinduct* (*auto dest*: *ev-alw-impl*)

lemma *ev-holds-sset*:
ev (*holds P*) *xs* $\longleftrightarrow$ ($\exists$ *x* $\in$ *sset xs*. *P x*) (**is** *?L* $\longleftrightarrow$ *?R*)
proof *safe*
 assume *?L* **thus** *?R* **by** *induct* (*metis holds.simps stream.set-sel*(*1*), *metis stl-sset*)
next
 fix *x* **assume** *x* $\in$ *sset xs P x*
 thus *?L* **by** (*induct rule*: *sset-induct*) (*simp-all add*: *ev.base ev.step*)
qed

LTL as a program logic:

lemma *alw-invar*:
assumes φ *xs* **and** *alw* (φ *impl nxt* φ) *xs*
shows *alw* φ *xs*
proof−
 {assume φ *xs* $\wedge$ *alw* (φ *impl nxt* φ) *xs* **hence** *?thesis*
  by *coinduct auto*
 }
 thus *?thesis* **using** *assms* **by** *auto*
qed

lemma *variance*:
assumes *1*: φ *xs* **and** *2*: *alw* (φ *impl* (ψ *or nxt* φ)) *xs*
shows (*alw* φ *or ev* ψ) *xs*
proof−

{assume ¬ *ev* ψ *xs* **hence** *alw* (*not* ψ) *xs* **unfolding** *not-ev*[*symmetric*] **.**
moreover have *alw* (*not* ψ *impl* (φ *impl nxt* φ)) *xs*
using *2* **by** *coinduct auto*
ultimately have *alw* (φ *impl nxt* φ) *xs* **by**(*auto dest*: *alw-mp*)
with *1* **have** *alw* φ *xs* **by**(*rule alw-invar*)
}
thus *?thesis* **by** *blast*
qed

lemma *ev-alw-imp-nxt*:
assumes *e*: *ev* φ *xs* **and** *a*: *alw* (φ *impl* (*nxt* φ)) *xs*
shows *ev* (*alw* φ) *xs*
proof−
obtain *xl xs1* **where** *xs*: *xs* = *xl* @− *xs1* **and** φ: φ *xs1*
using *e* **by** (*metis ev-imp-shift*)
have φ *xs1* ∧ *alw* (φ *impl* (*nxt* φ)) *xs1* **using** *a* φ **unfolding** *xs* **by** (*metis alw-shift*)
hence *alw* φ *xs1* **by**(*coinduct xs1 rule*: *alw.coinduct*) *auto*
thus *?thesis* **unfolding** *xs* **by** (*auto intro*: *alw-ev-shift*)
qed

inductive *ev-at* :: ('*a stream* ⇒ *bool*) ⇒ *nat* ⇒ '*a stream* ⇒ *bool* **for** *P* :: '*a stream* ⇒ *bool* **where**
base: *P* ω ⟹ *ev-at P 0* ω
| *step*:¬ *P* ω ⟹ *ev-at P n* (*stl* ω) ⟹ *ev-at P* (*Suc n*) ω

inductive-simps *ev-at-0*[*simp*]: *ev-at P 0* ω
inductive-simps *ev-at-Suc*[*simp*]: *ev-at P* (*Suc n*) ω

lemma *ev-at-imp-snth*: *ev-at P n* ω ⟹ *P* (*sdrop n* ω)
by (*induction n arbitrary*: ω) *auto*

lemma *ev-at-HLD-imp-snth*: *ev-at* (*HLD X*) *n* ω ⟹ ω !! *n* ∈ *X*
by (*auto dest*!: *ev-at-imp-snth simp*: *HLD-iff*)

lemma *ev-at-HLD-single-imp-snth*: *ev-at* (*HLD* {*x*}) *n* ω ⟹ ω !! *n* = *x*
by (*drule ev-at-HLD-imp-snth*) *simp*

lemma *ev-at-unique*: *ev-at P n* ω ⟹ *ev-at P m* ω ⟹ *n* = *m*
proof (*induction arbitrary*: *m rule*: *ev-at.induct*)
case (*base* ω) **then show** *?case*
by (*simp add*: *ev-at.simps*[*of* - - ω])
next
case (*step* ω *n*) **from** *step.prems step.hyps step.IH*[*of m* − *1*] **show** *?case*
by (*auto simp add*: *ev-at.simps*[*of* - - ω])
qed

lemma *ev-iff-ev-at*: *ev P* ω ⟷ (∃ *n*. *ev-at P n* ω)

proof
assume *ev P ω* **then show** ∃ *n*. *ev-at P n ω*
by (*induction rule*: *ev-induct-strong*) (*auto intro*: *ev-at.intros*)
next
assume ∃ *n*. *ev-at P n ω*
then obtain *n* **where** *ev-at P n ω*
by *auto*
then show *ev P ω*
by *induction auto*
qed

lemma *ev-at-shift*: *ev-at* (*HLD X*) *i* (*stake* (*Suc i*) *ω* @− *ω'* :: *'s stream*) ⟷ *ev-at* (*HLD X*) *i ω*
by (*induction i arbitrary*: *ω*) (*auto simp*: *HLD-iff*)

lemma *ev-iff-ev-at-unique*: *ev P ω* ⟷ (∃!*n*. *ev-at P n ω*)
by (*auto intro*: *ev-at-unique simp*: *ev-iff-ev-at*)

lemma *alw-HLD-iff-streams*: *alw* (*HLD X*) *ω* ⟷ *ω* ∈ *streams X*
proof
assume *alw* (*HLD X*) *ω* **then show** *ω* ∈ *streams X*
proof (*coinduction arbitrary*: *ω*)
case (*streams ω*) **then show** *?case* **by** (*cases ω*) *auto*
qed
next
assume *ω* ∈ *streams X* **then show** *alw* (*HLD X*) *ω*
proof (*coinduction arbitrary*: *ω*)
case (*alw ω*) **then show** *?case* **by** (*cases ω*) *auto*
qed
qed

lemma *not-HLD*: *not* (*HLD X*) = *HLD* (− *X*)
by (*auto simp*: *HLD-iff*)

lemma *not-alw-iff*: ¬ (*alw P ω*) ⟷ *ev* (*not P*) *ω*
using *not-alw*[*of P*] **by** (*simp add*: *fun-eq-iff*)

lemma *not-ev-iff*: ¬ (*ev P ω*) ⟷ *alw* (*not P*) *ω*
using *not-alw-iff*[*of not P ω*, *symmetric*] **by** *simp*

lemma *ev-Stream*: *ev P* (*x* ## *s*) ⟷ *P* (*x* ## *s*) ∨ *ev P s*
by (*auto elim*: *ev.cases*)

lemma *alw-ev-imp-ev-alw*:
assumes *alw* (*ev P*) *ω* **shows** *ev* (*P aand alw* (*ev P*)) *ω*
proof −
have *ev P ω* **using** *assms* **by** *auto*
from *this assms* **show** *?thesis*
by *induct auto*

qed

lemma *ev-False*: *ev* (λ*x*. *False*) ω ⟷ *False*
proof
  assume *ev* (λ*x*. *False*) ω **then show** *False*
    by *induct auto*
qed *auto*

lemma *alw-False*: *alw* (λ*x*. *False*) ω ⟷ *False*
  by *auto*

lemma *ev-iff-sdrop*: *ev P* ω ⟷ (∃ *m*. *P* (*sdrop m* ω))
proof *safe*
  assume *ev P* ω **then show** ∃ *m*. *P* (*sdrop m* ω)
    by (*induct rule*: *ev-induct-strong*) (*auto intro*: *exI*[*of* - *0*] *exI*[*of* - *Suc n* **for** *n*])
next
  fix *m* **assume** *P* (*sdrop m* ω) **then show** *ev P* ω
    by (*induct m arbitrary*: ω) *auto*
qed

lemma *alw-iff-sdrop*: *alw P* ω ⟷ (∀ *m*. *P* (*sdrop m* ω))
proof *safe*
  fix *m* **assume** *alw P* ω **then show** *P* (*sdrop m* ω)
    by (*induct m arbitrary*: ω) *auto*
next
  assume ∀ *m*. *P* (*sdrop m* ω) **then show** *alw P* ω
    by (*coinduction arbitrary*: ω) (*auto elim*: *allE*[*of* - *0*] *allE*[*of* - *Suc n* **for** *n*])
qed

lemma *infinite-iff-alw-ev*: *infinite* {*m*. *P* (*sdrop m* ω)} ⟷ *alw* (*ev P*) ω
  unfolding *infinite-nat-iff-unbounded-le alw-iff-sdrop ev-iff-sdrop*
  by *simp* (*metis le-Suc-ex le-add1*)

lemma *alw-inv*:
  assumes *stl*: ⋀*s*. *f* (*stl s*) = *stl* (*f s*)
  shows *alw P* (*f s*) ⟷ *alw* (λ*x*. *P* (*f x*)) *s*
proof
  assume *alw P* (*f s*) **then show** *alw* (λ*x*. *P* (*f x*)) *s*
    by (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
      (*auto simp*: *stl*)
next
  assume *alw* (λ*x*. *P* (*f x*)) *s* **then show** *alw P* (*f s*)
    by (*coinduction arbitrary*: *s rule*: *alw-coinduct*) (*auto simp flip*: *stl*)
qed

lemma *ev-inv*:
  assumes *stl*: ⋀*s*. *f* (*stl s*) = *stl* (*f s*)
  shows *ev P* (*f s*) ⟷ *ev* (λ*x*. *P* (*f x*)) *s*
proof

assume *ev P* (*f s*) **then show** *ev* (λ*x*. *P* (*f x*)) *s*
by (*induction f s arbitrary*: *s*) (*auto simp*: *stl*)
next
assume *ev* (λ*x*. *P* (*f x*)) *s* **then show** *ev P* (*f s*)
by *induction* (*auto simp flip*: *stl*)
qed

lemma *alw-smap*: *alw P* (*smap f s*) ⟷ *alw* (λ*x*. *P* (*smap f x*)) *s*
by (*rule alw-inv*) *simp*

lemma *ev-smap*: *ev P* (*smap f s*) ⟷ *ev* (λ*x*. *P* (*smap f x*)) *s*
by (*rule ev-inv*) *simp*

lemma *alw-cong*:
assumes *P*: *alw P* ω **and** *eq*: ⋀ω. *P* ω ⟹ *Q1* ω ⟷ *Q2* ω
shows *alw Q1* ω ⟷ *alw Q2* ω
proof −
from *eq* **have** (*alw P aand Q1*) = (*alw P aand Q2*) **by** *auto*
then have *alw* (*alw P aand Q1*) ω = *alw* (*alw P aand Q2*) ω **by** *auto*
with *P* **show** *alw Q1* ω ⟷ *alw Q2* ω
by (*simp add*: *alw-aand*)
qed

lemma *ev-cong*:
assumes *P*: *alw P* ω **and** *eq*: ⋀ω. *P* ω ⟹ *Q1* ω ⟷ *Q2* ω
shows *ev Q1* ω ⟷ *ev Q2* ω
proof −
from *P* **have** *alw* (λ*xs*. *Q1 xs* ⟶ *Q2 xs*) ω **by** (*rule alw-mono*) (*simp add*: *eq*)
moreover from *P* **have** *alw* (λ*xs*. *Q2 xs* ⟶ *Q1 xs*) ω **by** (*rule alw-mono*) (*simp add*: *eq*)
moreover note *ev-alw-impl*[*of Q1* ω *Q2*] *ev-alw-impl*[*of Q2* ω *Q1*]
ultimately show *ev Q1* ω ⟷ *ev Q2* ω
by *auto*
qed

lemma *alwD*: *alw P x* ⟹ *P x*
by *auto*

lemma *alw-alwD*: *alw P* ω ⟹ *alw* (*alw P*) ω
by *simp*

lemma *alw-ev-stl*: *alw* (*ev P*) (*stl* ω) ⟷ *alw* (*ev P*) ω
by (*auto intro*: *alw.intros*)

lemma *holds-Stream*: *holds P* (*x* ## *s*) ⟷ *P x*
by *simp*

lemma *holds-eq1*[*simp*]: *holds* ((=) *x*) = *HLD* {*x*}
by *rule* (*auto simp*: *HLD-iff*)

lemma *holds-eq2*[*simp*]: *holds* (λ*y*. *y* = *x*) = *HLD* {*x*}
by *rule* (*auto simp*: *HLD-iff*)

lemma *not-holds-eq*[*simp*]: *holds* (− (=) *x*) = *not* (*HLD* {*x*})
by *rule* (*auto simp*: *HLD-iff*)

Strong until

context
notes [[*inductive-internals*]]
begin

inductive *suntil* (**infix** ‹*suntil*› *60*) **for** φ ψ **where**
base: ψ ω ⟹ (φ *suntil* ψ) ω
| *step*: φ ω ⟹ (φ *suntil* ψ) (*stl* ω) ⟹ (φ *suntil* ψ) ω

inductive-simps *suntil-Stream*: (φ *suntil* ψ) (*x* ## *s*)

end

lemma *suntil-induct-strong*[*consumes 1*, *case-names base step*]:
(φ *suntil* ψ) *x* ⟹
(⋀ω. ψ ω ⟹ *P* ω) ⟹
(⋀ω. φ ω ⟹ ¬ ψ ω ⟹ (φ *suntil* ψ) (*stl* ω) ⟹ *P* (*stl* ω) ⟹ *P* ω) ⟹ *P x*
using *suntil.induct*[*of* φ ψ *x P*] **by** *blast*

lemma *ev-suntil*: (φ *suntil* ψ) ω ⟹ *ev* ψ ω
by (*induct rule*: *suntil.induct*) *auto*

lemma *suntil-inv*:
assumes *stl*: ⋀*s*. *f* (*stl s*) = *stl* (*f s*)
shows (*P suntil Q*) (*f s*) ⟷ ((λ*x*. *P* (*f x*)) *suntil* (λ*x*. *Q* (*f x*))) *s*
proof
assume (*P suntil Q*) (*f s*) **then show** ((λ*x*. *P* (*f x*)) *suntil* (λ*x*. *Q* (*f x*))) *s*
by (*induction f s arbitrary*: *s*) (*auto simp*: *stl intro*: *suntil.intros*)
next
assume ((λ*x*. *P* (*f x*)) *suntil* (λ*x*. *Q* (*f x*))) *s* **then show** (*P suntil Q*) (*f s*)
by *induction* (*auto simp flip*: *stl intro*: *suntil.intros*)
qed

lemma *suntil-smap*: (*P suntil Q*) (*smap f s*) ⟷ ((λ*x*. *P* (*smap f x*)) *suntil* (λ*x*. *Q* (*smap f x*))) *s*
by (*rule suntil-inv*) *simp*

lemma *hld-smap*: *HLD x* (*smap f s*) = *holds* (λ*y*. *f y* ∈ *x*) *s*
by (*simp add*: *HLD-def*)

lemma *suntil-mono*:
assumes *eq*: ⋀ω. *P* ω ⟹ *Q1* ω ⟹ *Q2* ω ⋀ω. *P* ω ⟹ *R1* ω ⟹ *R2* ω

assumes $*$: $(Q1\ suntil\ R1)\ \omega\ alw\ P\ \omega$ **shows** $(Q2\ suntil\ R2)\ \omega$
using $*$ **by** *induct* (*auto intro*: *eq suntil.intros*)

lemma *suntil-cong*:
$alw\ P\ \omega \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow Q1\ \omega \longleftrightarrow Q2\ \omega) \Longrightarrow (\bigwedge\omega.\ P\ \omega \Longrightarrow R1\ \omega \longleftrightarrow R2\ \omega) \Longrightarrow$
$(Q1\ suntil\ R1)\ \omega \longleftrightarrow (Q2\ suntil\ R2)\ \omega$
using *suntil-mono*[*of P Q1 Q2 R1 R2* ω] *suntil-mono*[*of P Q2 Q1 R2 R1* ω] **by** *auto*

lemma *ev-suntil-iff*: $ev\ (P\ suntil\ Q)\ \omega \longleftrightarrow ev\ Q\ \omega$
proof
assume $ev\ (P\ suntil\ Q)\ \omega$ **then show** $ev\ Q\ \omega$
by *induct* (*auto dest*: *ev-suntil*)
next
assume $ev\ Q\ \omega$ **then show** $ev\ (P\ suntil\ Q)\ \omega$
by *induct* (*auto intro*: *suntil.intros*)
qed

lemma *true-suntil*: $((\lambda\text{-}.\ True)\ suntil\ P) = ev\ P$
by (*simp add*: *suntil-def ev-def*)

lemma *suntil-lfp*: $(\varphi\ suntil\ \psi) = lfp\ (\lambda P\ s.\ \psi\ s \vee (\varphi\ s \wedge P\ (stl\ s)))$
by (*simp add*: *suntil-def*)

lemma *sfilter-P*[*simp*]: $P\ (shd\ s) \Longrightarrow sfilter\ P\ s = shd\ s\ \#\#\ sfilter\ P\ (stl\ s)$
using *sfilter-Stream*[*of P shd s stl s*] **by** *simp*

lemma *sfilter-not-P*[*simp*]: $\neg\ P\ (shd\ s) \Longrightarrow sfilter\ P\ s = sfilter\ P\ (stl\ s)$
using *sfilter-Stream*[*of P shd s stl s*] **by** *simp*

lemma *sfilter-eq*:
assumes $ev\ (holds\ P)\ s$
shows $sfilter\ P\ s = x\ \#\#\ s' \longleftrightarrow$
$P\ x \wedge (not\ (holds\ P)\ suntil\ (HLD\ \{x\}\ aand\ nxt\ (\lambda s.\ sfilter\ P\ s = s')))\ s$
using *assms*
by (*induct rule*: *ev-induct-strong*)
(*auto simp add*: *HLD-iff intro*: *suntil.intros elim*: *suntil.cases*)

lemma *sfilter-streams*:
$alw\ (ev\ (holds\ P))\ \omega \Longrightarrow \omega \in streams\ A \Longrightarrow sfilter\ P\ \omega \in streams\ \{x{\in}A.\ P\ x\}$
proof (*coinduction arbitrary*: ω)
case (*streams* ω)
then have $ev\ (holds\ P)\ \omega$ **by** *blast*
from *this streams* **show** *?case*
by (*induct rule*: *ev-induct-strong*) (*auto elim*: *streamsE*)
qed

lemma *alw-sfilter*:

assumes ∗: *alw* (*ev* (*holds P*)) *s*
shows *alw Q* (*sfilter P s*) ⟷ *alw* (λ*x*. *Q* (*sfilter P x*)) *s*
proof
assume *alw Q* (*sfilter P s*) **with** ∗ **show** *alw* (λ*x*. *Q* (*sfilter P x*)) *s*
proof (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
case (*stl s*)
then have *ev* (*holds P*) *s*
by *blast*
from *this stl* **show** *?case*
by (*induct rule*: *ev-induct-strong*) *auto*
qed *auto*
next
assume *alw* (λ*x*. *Q* (*sfilter P x*)) *s* **with** ∗ **show** *alw Q* (*sfilter P s*)
proof (*coinduction arbitrary*: *s rule*: *alw-coinduct*)
case (*stl s*)
then have *ev* (*holds P*) *s*
by *blast*
from *this stl* **show** *?case*
by (*induct rule*: *ev-induct-strong*) *auto*
qed *auto*
qed

lemma *ev-sfilter*:
assumes ∗: *alw* (*ev* (*holds P*)) *s*
shows *ev Q* (*sfilter P s*) ⟷ *ev* (λ*x*. *Q* (*sfilter P x*)) *s*
proof
assume *ev Q* (*sfilter P s*) **from** *this* ∗ **show** *ev* (λ*x*. *Q* (*sfilter P x*)) *s*
proof (*induction sfilter P s arbitrary*: *s rule*: *ev-induct-strong*)
case (*step s*)
then have *ev* (*holds P*) *s*
by *blast*
from *this step* **show** *?case*
by (*induct rule*: *ev-induct-strong*) *auto*
qed *auto*
next
assume *ev* (λ*x*. *Q* (*sfilter P x*)) *s* **then show** *ev Q* (*sfilter P s*)
proof (*induction rule*: *ev-induct-strong*)
case (*step s*) **then show** *?case*
by (*cases P* (*shd s*)) *auto*
qed *auto*
qed

lemma *holds-sfilter*:
assumes *ev* (*holds Q*) *s* **shows** *holds P* (*sfilter Q s*) ⟷ (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s*
proof
assume *holds P* (*sfilter Q s*) **with** *assms* **show** (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s*
by (*induct rule*: *ev-induct-strong*) (*auto intro*: *suntil.intros*)

next
assume (*not* (*holds Q*) *suntil* (*holds* (*Q aand P*))) *s* **then show** *holds P* (*sfilter Q s*)
by *induct auto*
qed

lemma *suntil-aand-nxt*:
(*φ suntil* (*φ aand nxt ψ*)) *ω* ⟷ (*φ aand nxt* (*φ suntil ψ*)) *ω*
proof
assume (*φ suntil* (*φ aand nxt ψ*)) *ω* **then show** (*φ aand nxt* (*φ suntil ψ*)) *ω*
by *induction* (*auto intro*: *suntil.intros*)
next
assume (*φ aand nxt* (*φ suntil ψ*)) *ω*
then have (*φ suntil ψ*) (*stl ω*) *φ ω*
by *auto*
then show (*φ suntil* (*φ aand nxt ψ*)) *ω*
by (*induction stl ω arbitrary*: *ω*)
(*auto elim*: *suntil.cases intro*: *suntil.intros*)
qed

lemma *alw-sconst*: *alw P* (*sconst x*) ⟷ *P* (*sconst x*)
proof
assume *P* (*sconst x*) **then show** *alw P* (*sconst x*)
by *coinduction auto*
qed *auto*

lemma *ev-sconst*: *ev P* (*sconst x*) ⟷ *P* (*sconst x*)
proof
assume *ev P* (*sconst x*) **then show** *P* (*sconst x*)
by (*induction sconst x*) *auto*
qed *auto*

lemma *suntil-sconst*: (*φ suntil ψ*) (*sconst x*) ⟷ *ψ* (*sconst x*)
proof
assume (*φ suntil ψ*) (*sconst x*) **then show** *ψ* (*sconst x*)
by (*induction sconst x*) *auto*
qed (*auto intro*: *suntil.intros*)

lemma *hld-smap'*: *HLD x* (*smap f s*) = *HLD* (*f* −‘ *x*) *s*
by (*simp add*: *HLD-def*)

lemma *pigeonhole-stream*:
assumes *alw* (*HLD s*) *ω*
assumes *finite s*
shows ∃ *x*∈*s*. *alw* (*ev* (*HLD* {*x*})) *ω*
proof −
have ∀ *i*∈*UNIV*. ∃ *x*∈*s*. *ω* !! *i* = *x*
using ‹*alw* (*HLD s*) *ω*› **by** (*simp add*: *alw-iff-sdrop HLD-iff*)
from *pigeonhole-infinite-rel*[*OF infinite-UNIV-nat* ‹*finite s*› *this*]

show *?thesis*
by (*simp add*: *HLD-iff flip*: *infinite-iff-alw-ev*)
qed

lemma *ev-eq-suntil*: *ev P ω ⟷ (not P suntil P) ω*
proof
assume *ev P ω* **then show** *((λxs. ¬ P xs) suntil P) ω*
by (*induction rule*: *ev-induct-strong*) (*auto intro*: *suntil.intros*)
qed (*auto simp*: *ev-suntil*)

62 Weak vs. strong until (contributed by Michael Foster, University of Sheffield)

lemma *suntil-implies-until*: *(φ suntil ψ) ω ⟹ (φ until ψ) ω*
by (*induct rule*: *suntil-induct-strong*) (*auto intro*: *UNTIL.intros*)

lemma *alw-implies-until*: *alw φ ω ⟹ (φ until ψ) ω*
unfolding *until-false*[*symmetric*] **by** (*auto elim*: *until-mono*)

lemma *until-ev-suntil*: *(φ until ψ) ω ⟹ ev ψ ω ⟹ (φ suntil ψ) ω*
proof (*rotate-tac*, *induction rule*: *ev.induct*)
case (*base xs*)
then show *?case*
by (*simp add*: *suntil.base*)
next
case (*step xs*)
then show *?case*
by (*metis UNTIL.cases suntil.base suntil.step*)
qed

lemma *suntil-as-until*: *(φ suntil ψ) ω = ((φ until ψ) ω ∧ ev ψ ω)*
using *ev-suntil suntil-implies-until until-ev-suntil* **by** *blast*

lemma *until-not-relesased-now*: *(φ until ψ) ω ⟹ ¬ ψ ω ⟹ φ ω*
using *UNTIL.cases* **by** *auto*

lemma *until-must-release-ev*: *(φ until ψ) ω ⟹ ev (not φ) ω ⟹ ev ψ ω*
proof (*rotate-tac*, *induction rule*: *ev.induct*)
case (*base xs*)
then show *?case*
using *until-not-relesased-now* **by** *blast*
next
case (*step xs*)
then show *?case*
using *UNTIL.cases* **by** *blast*
qed

lemma *until-as-suntil*: *(φ until ψ) ω = ((φ suntil ψ) or (alw φ)) ω*

using *alw-implies-until not-alw-iff suntil-implies-until until-ev-suntil until-must-release-ev*
by *blast*

lemma *alw-holds*: *alw* (*holds P*) (*h##t*) = (*P h* ∧ *alw* (*holds P*) *t*)
by (*metis alw.simps holds-Stream stream.sel*(*2*))

lemma *alw-holds2*: *alw* (*holds P*) *ss* = (*P* (*shd ss*) ∧ *alw* (*holds P*) (*stl ss*))
by (*meson alw.simps holds.elims*(*2*) *holds.elims*(*3*))

lemma *alw-eq-sconst*: (*alw* (*HLD* {*h*}) *t*) = (*t* = *sconst h*)
unfolding *sconst-alt alw-HLD-iff-streams streams-iff-sset*
using *stream.set-sel*(*1*) **by** *force*

lemma *sdrop-if-suntil*: (*p suntil q*) *ω* ⟹ ∃*j*. *q* (*sdrop j ω*) ∧ (∀*k* < *j*. *p* (*sdrop k ω*))
proof(*induction rule*: *suntil.induct*)
case (*base ω*)
then show *?case*
by *force*
next
case (*step ω*)
then obtain *j* **where** *q* (*sdrop j* (*stl ω*)) ∀*k*<*j*. *p* (*sdrop k* (*stl ω*)) **by** *blast*
with *step*(*1*,*2*) **show** *?case*
using *ev-at-imp-snth less-Suc-eq-0-disj* **by** (*auto intro*!: *exI*[**where** *x*=*j*+*1*])
qed

lemma *not-suntil*: (¬ (*p suntil q*) *ω*) = (¬ (*p until q*) *ω* ∨ *alw* (*not q*) *ω*)
by (*simp add*: *suntil-as-until alw-iff-sdrop ev-iff-sdrop*)

lemma *sdrop-until*: *q* (*sdrop j ω*) ⟹ ∀*k*<*j*. *p* (*sdrop k ω*) ⟹ (*p until q*) *ω*
proof(*induct j arbitrary*: *ω*)
case *0*
then show *?case*
by (*simp add*: *UNTIL.base*)
next
case (*Suc j*)
then show *?case*
by (*metis Suc-mono UNTIL.simps sdrop.simps*(*1*) *sdrop.simps*(*2*) *zero-less-Suc*)
qed

lemma *sdrop-suntil*: *q* (*sdrop j ω*) ⟹ (∀*k* < *j*. *p* (*sdrop k ω*)) ⟹ (*p suntil q*) *ω*
by (*metis ev-iff-sdrop sdrop-until suntil-as-until*)

lemma *suntil-iff-sdrop*: (*p suntil q*) *ω* = (∃*j*. *q* (*sdrop j ω*) ∧ (∀*k* < *j*. *p* (*sdrop k ω*)))
using *sdrop-if-suntil sdrop-suntil* **by** *blast*

end

63 Lists as vectors

theory *ListVector*
imports *Main*
begin

A vector-space like structure of lists and arithmetic operations on them. Is only a vector space if restricted to lists of the same length.

Multiplication with a scalar:

abbreviation *scale* :: ('*a*::*times*) ⇒ '*a list* ⇒ '*a list* (**infix** ‹$*_s$› *70*)
where $x *_s xs \equiv map\ ((*)\ x)\ xs$

lemma *scale1*[*simp*]: (*1*::'*a*::*monoid-mult*) $*_s$ *xs* = *xs*
by (*induct xs*) *simp-all*

63.1 + and −

fun *zipwith0* :: ('*a*::*zero* ⇒ '*b*::*zero* ⇒ '*c*) ⇒ '*a list* ⇒ '*b list* ⇒ '*c list*
where
zipwith0 f [] [] = [] |
zipwith0 f (*x*#*xs*) (*y*#*ys*) = *f x y* # *zipwith0 f xs ys* |
zipwith0 f (*x*#*xs*) [] = *f x 0* # *zipwith0 f xs* [] |
zipwith0 f [] (*y*#*ys*) = *f 0 y* # *zipwith0 f* [] *ys*

instantiation *list* :: ({*zero*, *plus*}) *plus*
begin

definition
list-add-def: (+) = *zipwith0* (+)

instance ..

end

instantiation *list* :: ({*zero*, *uminus*}) *uminus*
begin

definition
list-uminus-def: *uminus* = *map uminus*

instance ..

end

instantiation *list* :: ({*zero*,*minus*}) *minus*
begin

definition
list-diff-def: (−) = *zipwith0* (−)

instance ..

end

lemma *zipwith0-Nil*[*simp*]: *zipwith0 f* [] *ys* = *map* (*f 0*) *ys*
by(*induct ys*) *simp-all*

lemma *list-add-Nil*[*simp*]: [] + *xs* = (*xs*::*'a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Nil2*[*simp*]: *xs* + [] = (*xs*::*'a*::*monoid-add list*)
by (*induct xs*) (*auto simp*:*list-add-def*)

lemma *list-add-Cons*[*simp*]: (*x*#*xs*) + (*y*#*ys*) = (*x*+*y*)#(*xs*+*ys*)
by(*auto simp*:*list-add-def*)

lemma *list-diff-Nil*[*simp*]: [] − *xs* = −(*xs*::*'a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def list-uminus-def*)

lemma *list-diff-Nil2*[*simp*]: *xs* − [] = (*xs*::*'a*::*group-add list*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-diff-Cons-Cons*[*simp*]: (*x*#*xs*) − (*y*#*ys*) = (*x*−*y*)#(*xs*−*ys*)
by (*induct xs*) (*auto simp*:*list-diff-def*)

lemma *list-uminus-Cons*[*simp*]: −(*x*#*xs*) = (−*x*)#(−*xs*)
by (*induct xs*) (*auto simp*:*list-uminus-def*)

lemma *self-list-diff*:
xs − *xs* = *replicate* (*length*(*xs*::*'a*::*group-add list*)) *0*
by(*induct xs*) *simp-all*

lemma *list-add-assoc*:
fixes *xs* :: *'a*::*monoid-add list*
shows (*xs*+*ys*)+*zs* = *xs*+(*ys*+*zs*)
proof (*induct xs arbitrary*: *ys zs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs ys zs*)
show *?case*
by (*cases ys*; *cases zs*; *simp add*: *add.assoc Cons*)
qed

63.2 Inner product

definition *iprod* :: *'a*::*ring list* ⇒ *'a list* ⇒ *'a* (‹(‹*open-block notation*=‹*mixfix iprod*››⟨-,-⟩)›)

where $\langle xs,ys\rangle = (\sum (x,y) \leftarrow zip\ xs\ ys.\ x*y)$

lemma *iprod-Nil*[*simp*]: $\langle[],ys\rangle = 0$
by(*simp add*: *iprod-def*)

lemma *iprod-Nil2*[*simp*]: $\langle xs,[]\rangle = 0$
by(*simp add*: *iprod-def*)

lemma *iprod-Cons*[*simp*]: $\langle x\#xs,y\#ys\rangle = x*y + \langle xs,ys\rangle$
by(*simp add*: *iprod-def*)

lemma *iprod0-if-coeffs0*: $\forall\, c{\in}set\ cs.\ c = 0 \Longrightarrow \langle cs,xs\rangle = 0$
proof (*induct cs arbitrary*: *xs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a cs xs*)
then show *?case*
by (*cases xs*; *fastforce*)
qed

lemma *iprod-uminus*[*simp*]: $\langle -xs,ys\rangle = -\langle xs,ys\rangle$
by(*simp add*: *iprod-def uminus-sum-list-map o-def split-def map-zip-map list-uminus-def*)

lemma *iprod-left-add-distrib*: $\langle xs + ys,zs\rangle = \langle xs,zs\rangle + \langle ys,zs\rangle$
proof (*induct xs arbitrary*: *ys zs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs ys zs*)
show *?case*
by (*cases ys*; *cases zs*; *simp add*: *distrib-right Cons*)
qed

lemma *iprod-left-diff-distrib*: $\langle xs - ys,\ zs\rangle = \langle xs,zs\rangle - \langle ys,zs\rangle$
proof (*induct xs arbitrary*: *ys zs*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons a xs ys zs*)
show *?case*
by (*cases ys*; *cases zs*; *simp add*: *left-diff-distrib Cons*)
qed

lemma *iprod-assoc*: $\langle x *_s xs,\ ys\rangle = x * \langle xs,ys\rangle$
proof (*induct xs arbitrary*: *ys*)
case *Nil*
then show *?case* **by** *simp*
next

case (*Cons a xs ys*)
show *?case*
by (*cases ys*; *simp add*: *distrib-left mult.assoc Cons*)
qed

end

64 Definitions of Least Upper Bounds and Greatest Lower Bounds

theory *Lub-Glb*
imports *Complex-Main*
begin

Thanks to suggestions by James Margetson

definition *setle* :: *'a set* ⇒ *'a*::*ord* ⇒ *bool* (**infixl** ‹∗<=› *70*)
where *S* ∗<= *x* = (∀ *y*∈*S*. *y* ≤ *x*)

definition *setge* :: *'a*::*ord* ⇒ *'a set* ⇒ *bool* (**infixl** ‹<=∗› *70*)
where *x* <=∗ *S* = (∀ *y*∈*S*. *x* ≤ *y*)

64.1 Rules for the Relations ∗<= and <=∗

lemma *setleI*: ∀ *y*∈*S*. *y* ≤ *x* ⟹ *S* ∗<= *x*
by (*simp add*: *setle-def*)

lemma *setleD*: *S* ∗<= *x* ⟹ *y*∈*S* ⟹ *y* ≤ *x*
by (*simp add*: *setle-def*)

lemma *setgeI*: ∀ *y*∈*S*. *x* ≤ *y* ⟹ *x* <=∗ *S*
by (*simp add*: *setge-def*)

lemma *setgeD*: *x* <=∗ *S* ⟹ *y*∈*S* ⟹ *x* ≤ *y*
by (*simp add*: *setge-def*)

definition *leastP* :: (*'a* ⇒ *bool*) ⇒ *'a*::*ord* ⇒ *bool*
where *leastP P x* = (*P x* ∧ *x* <=∗ *Collect P*)

definition *isUb* :: *'a set* ⇒ *'a set* ⇒ *'a*::*ord* ⇒ *bool*
where *isUb R S x* = (*S* ∗<= *x* ∧ *x* ∈ *R*)

definition *isLub* :: *'a set* ⇒ *'a set* ⇒ *'a*::*ord* ⇒ *bool*
where *isLub R S x* = *leastP* (*isUb R S*) *x*

definition *ubs* :: *'a set* ⇒ *'a*::*ord set* ⇒ *'a set*
where *ubs R S* = *Collect* (*isUb R S*)

64.2 Rules about the Operators *leastP*, *ub* and *lub*

lemma *leastPD1*: $leastP\ P\ x \Longrightarrow P\ x$
by (*simp add*: *leastP-def*)

lemma *leastPD2*: $leastP\ P\ x \Longrightarrow x <=* Collect\ P$
by (*simp add*: *leastP-def*)

lemma *leastPD3*: $leastP\ P\ x \Longrightarrow y \in Collect\ P \Longrightarrow x \leq y$
by (*blast dest*!: *leastPD2 setgeD*)

lemma *isLubD1*: $isLub\ R\ S\ x \Longrightarrow S\ *<=\ x$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLubD1a*: $isLub\ R\ S\ x \Longrightarrow x \in R$
by (*simp add*: *isLub-def isUb-def leastP-def*)

lemma *isLub-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ x$
unfolding *isUb-def* **by** (*blast dest*: *isLubD1 isLubD1a*)

lemma *isLubD2*: $isLub\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
by (*blast dest*!: *isLubD1 setleD*)

lemma *isLubD3*: $isLub\ R\ S\ x \Longrightarrow leastP\ (isUb\ R\ S)\ x$
by (*simp add*: *isLub-def*)

lemma *isLubI1*: $leastP(isUb\ R\ S)\ x \Longrightarrow isLub\ R\ S\ x$
by (*simp add*: *isLub-def*)

lemma *isLubI2*: $isUb\ R\ S\ x \Longrightarrow x <=* Collect\ (isUb\ R\ S) \Longrightarrow isLub\ R\ S\ x$
by (*simp add*: *isLub-def leastP-def*)

lemma *isUbD*: $isUb\ R\ S\ x \Longrightarrow y \in S \Longrightarrow y \leq x$
by (*simp add*: *isUb-def setle-def*)

lemma *isUbD2*: $isUb\ R\ S\ x \Longrightarrow S\ *<=\ x$
by (*simp add*: *isUb-def*)

lemma *isUbD2a*: $isUb\ R\ S\ x \Longrightarrow x \in R$
by (*simp add*: *isUb-def*)

lemma *isUbI*: $S\ *<=\ x \Longrightarrow x \in R \Longrightarrow isUb\ R\ S\ x$
by (*simp add*: *isUb-def*)

lemma *isLub-le-isUb*: $isLub\ R\ S\ x \Longrightarrow isUb\ R\ S\ y \Longrightarrow x \leq y$
unfolding *isLub-def* **by** (*blast intro*!: *leastPD3*)

lemma *isLub-ubs*: $isLub\ R\ S\ x \Longrightarrow x <=* ubs\ R\ S$
unfolding *ubs-def isLub-def* **by** (*rule leastPD2*)

lemma *isLub-unique*: [| *isLub R S x*; *isLub R S y* |] ==> *x* = (*y*::'*a*::*linorder*)
apply (*frule isLub-isUb*)
apply (*frule-tac x* = *y* **in** *isLub-isUb*)
apply (*blast intro*!: *order-antisym dest*!: *isLub-le-isUb*)
done

lemma *isUb-UNIV-I*: (⋀*y*. *y* ∈ *S* ⟹ *y* ≤ *u*) ⟹ *isUb UNIV S u*
by (*simp add*: *isUbI setleI*)

definition *greatestP* :: ('*a* ⇒ *bool*) ⇒ '*a*::*ord* ⇒ *bool*
where *greatestP P x* = (*P x* ∧ *Collect P* *<= *x*)

definition *isLb* :: '*a set* ⇒ '*a set* ⇒ '*a*::*ord* ⇒ *bool*
where *isLb R S x* = (*x* <=* *S* ∧ *x* ∈ *R*)

definition *isGlb* :: '*a set* ⇒ '*a set* ⇒ '*a*::*ord* ⇒ *bool*
where *isGlb R S x* = *greatestP* (*isLb R S*) *x*

definition *lbs* :: '*a set* ⇒ '*a*::*ord set* ⇒ '*a set*
where *lbs R S* = *Collect* (*isLb R S*)

64.3 Rules about the Operators *greatestP*, *isLb* and *isGlb*

lemma *greatestPD1*: *greatestP P x* ⟹ *P x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD2*: *greatestP P x* ⟹ *Collect P* *<= *x*
by (*simp add*: *greatestP-def*)

lemma *greatestPD3*: *greatestP P x* ⟹ *y* ∈ *Collect P* ⟹ *x* ≥ *y*
by (*blast dest*!: *greatestPD2 setleD*)

lemma *isGlbD1*: *isGlb R S x* ⟹ *x* <=* *S*
by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlbD1a*: *isGlb R S x* ⟹ *x* ∈ *R*
by (*simp add*: *isGlb-def isLb-def greatestP-def*)

lemma *isGlb-isLb*: *isGlb R S x* ⟹ *isLb R S x*
unfolding *isLb-def* **by** (*blast dest*: *isGlbD1 isGlbD1a*)

lemma *isGlbD2*: *isGlb R S x* ⟹ *y* ∈ *S* ⟹ *y* ≥ *x*
by (*blast dest*!: *isGlbD1 setgeD*)

lemma *isGlbD3*: *isGlb R S x* ⟹ *greatestP* (*isLb R S*) *x*
by (*simp add*: *isGlb-def*)

lemma *isGlbI1*: *greatestP* (*isLb R S*) *x* ⟹ *isGlb R S x*

by (*simp add*: *isGlb-def*)

lemma *isGlbI2*: *isLb R S x* $\Longrightarrow$ *Collect* (*isLb R S*) *∗<= x* $\Longrightarrow$ *isGlb R S x*
by (*simp add*: *isGlb-def greatestP-def*)

lemma *isLbD*: *isLb R S x* $\Longrightarrow$ $y \in S \Longrightarrow y \geq x$
by (*simp add*: *isLb-def setge-def*)

lemma *isLbD2*: *isLb R S x* $\Longrightarrow$ *x <=∗ S*
by (*simp add*: *isLb-def*)

lemma *isLbD2a*: *isLb R S x* $\Longrightarrow$ $x \in R$
by (*simp add*: *isLb-def*)

lemma *isLbI*: *x <=∗ S* $\Longrightarrow$ $x \in R \Longrightarrow$ *isLb R S x*
by (*simp add*: *isLb-def*)

lemma *isGlb-le-isLb*: *isGlb R S x* $\Longrightarrow$ *isLb R S y* $\Longrightarrow$ $x \geq y$
unfolding *isGlb-def* **by** (*blast intro*!: *greatestPD3*)

lemma *isGlb-ubs*: *isGlb R S x* $\Longrightarrow$ *lbs R S ∗<= x*
unfolding *lbs-def isGlb-def* **by** (*rule greatestPD2*)

lemma *isGlb-unique*: [| *isGlb R S x*; *isGlb R S y* |] ==> $x = (y::'a::linorder)$
apply (*frule isGlb-isLb*)
apply (*frule-tac x = y* **in** *isGlb-isLb*)
apply (*blast intro*!: *order-antisym dest*!: *isGlb-le-isLb*)
done

lemma *bdd-above-setle*: *bdd-above A* $\longleftrightarrow$ $(\exists\, a.\ A$ *∗<= a*$)$
by (*auto simp*: *bdd-above-def setle-def*)

lemma *bdd-below-setge*: *bdd-below A* $\longleftrightarrow$ $(\exists\, a.\ a$ *<=∗ A*$)$
by (*auto simp*: *bdd-below-def setge-def*)

lemma *isLub-cSup*:
$(S::'a$ :: *conditionally-complete-lattice set*$) \neq \{\} \Longrightarrow (\exists\, b.\ S$ *∗<= b*$) \Longrightarrow$ *isLub UNIV S* (*Sup S*)
by (*auto simp add*: *isLub-def setle-def leastP-def isUb-def*
intro!: *setgeI cSup-upper cSup-least*)

lemma *isGlb-cInf*:
$(S::'a$ :: *conditionally-complete-lattice set*$) \neq \{\} \Longrightarrow (\exists\, b.\ b$ *<=∗ S*$) \Longrightarrow$ *isGlb UNIV S* (*Inf S*)
by (*auto simp add*: *isGlb-def setge-def greatestP-def isLb-def*
intro!: *setleI cInf-lower cInf-greatest*)

lemma *cSup-le*: $(S::'a::$*conditionally-complete-lattice set*$) \neq \{\} \Longrightarrow S$ *∗<= b* $\Longrightarrow$ *Sup S* $\leq b$

by (*metis cSup-least setle-def*)

lemma *cInf-ge*: (*S*::'*a* :: *conditionally-complete-lattice set*) ≠ {} ⟹ *b* <=* *S* ⟹ *Inf S* ≥ *b*
by (*metis cInf-greatest setge-def*)

lemma *cSup-bounds*:
fixes *S* :: '*a* :: *conditionally-complete-lattice set*
shows *S* ≠ {} ⟹ *a* <=* *S* ⟹ *S* *<= *b* ⟹ *a* ≤ *Sup S* ∧ *Sup S* ≤ *b*
using *cSup-least*[*of S b*] *cSup-upper2*[*of - S a*]
by (*auto simp*: *bdd-above-setle setge-def setle-def*)

lemma *cSup-unique*: (*S*::'*a* :: {*conditionally-complete-linorder*, *no-bot*} *set*) *<= *b* ⟹ (∀ *b'*<*b*. ∃ *x*∈*S*. *b'* < *x*) ⟹ *Sup S* = *b*
by (*rule cSup-eq*) (*auto simp*: *not-le*[*symmetric*] *setle-def*)

lemma *cInf-unique*: *b* <=* (*S*::'*a* :: {*conditionally-complete-linorder*, *no-top*} *set*) ⟹ (∀ *b'*>*b*. ∃ *x*∈*S*. *b'* > *x*) ⟹ *Inf S* = *b*
by (*rule cInf-eq*) (*auto simp*: *not-le*[*symmetric*] *setge-def*)

Use completeness of reals (supremum property) to show that any bounded sequence has a least upper bound

lemma *reals-complete*: ∃ *X*. *X* ∈ *S* ⟹ ∃ *Y*. *isUb* (*UNIV*::*real set*) *S Y* ⟹ ∃ *t*. *isLub* (*UNIV* :: *real set*) *S t*
by (*intro exI*[*of - Sup S*] *isLub-cSup*) (*auto simp*: *setle-def isUb-def intro*!: *cSup-upper*)

lemma *Bseq-isUb*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isUb* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
by (*auto intro*: *isUbI setleI simp add*: *Bseq-def abs-le-iff*)

lemma *Bseq-isLub*: ⋀*X* :: *nat* ⇒ *real*. *Bseq X* ⟹ ∃ *U*. *isLub* (*UNIV*::*real set*) {*x*. ∃ *n*. *X n* = *x*} *U*
by (*blast intro*: *reals-complete Bseq-isUb*)

lemma *isLub-mono-imp-LIMSEQ*:
fixes *X* :: *nat* ⇒ *real*
assumes *u*: *isLub UNIV* {*x*. ∃ *n*. *X n* = *x*} *u*
assumes *X*: ∀ *m n*. *m* ≤ *n* ⟶ *X m* ≤ *X n*
shows *X* ⟶⟶ *u*
proof −
have *X* ⟶⟶ (*SUP i*. *X i*)
using *u*[*THEN isLubD1*] *X*
by (*intro LIMSEQ-incseq-SUP*) (*auto simp*: *incseq-def image-def eq-commute bdd-above-setle*)
also have (*SUP i*. *X i*) = *u*
using *isLub-cSup*[*of range X*] *u*[*THEN isLubD1*]
by (*intro isLub-unique*[*OF - u*]) (*auto simp add*: *image-def eq-commute*)
finally show *?thesis* .
qed

lemmas *real-isGlb-unique* = *isGlb-unique*[**where** *'a=real*]

lemma *real-le-inf-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ b <=* s \Longrightarrow Inf\ s \leq Inf$ (*t*::*real set*)
by (*rule cInf-superset-mono*) (*auto simp*: *bdd-below-setge*)

lemma *real-ge-sup-subset*: $t \neq \{\} \Longrightarrow t \subseteq s \Longrightarrow \exists\, b.\ s *<= b \Longrightarrow Sup\ s \geq Sup$ (*t*::*real set*)
by (*rule cSup-subset-mono*) (*auto simp*: *bdd-above-setle*)

end

65 An abstract view on maps for code generation.

theory *Mapping*
imports *Main AList*
begin

65.1 Parametricity transfer rules

lemma *map-of-foldr*: *map-of xs* = *foldr* ($\lambda(k,\ v)\ m.\ m(k \mapsto v)$) *xs Map.empty*
using *map-add-map-of-foldr* [*of Map.empty*] **by** *auto*

context includes *lifting-syntax*
begin

lemma *empty-parametric*: (*A* ===> *rel-option B*) *Map.empty Map.empty*
by *transfer-prover*

lemma *lookup-parametric*: ((*A* ===> *B*) ===> *A* ===> *B*) ($\lambda m\ k.\ m\ k$) (λm $k.\ m\ k$)
by *transfer-prover*

lemma *update-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> *B* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option* B)
($\lambda k\ v\ m.\ m(k \mapsto v)$) ($\lambda k\ v\ m.\ m(k \mapsto v)$)
by *transfer-prover*

lemma *delete-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
($\lambda k\ m.\ m(k := None)$) ($\lambda k\ m.\ m(k := None)$)
by *transfer-prover*

lemma *is-none-parametric* [*transfer-rule*]:
(*rel-option A* ===> *HOL.eq*) *Option.is-none Option.is-none*

by (*auto simp add*: *Option.is-none-def rel-fun-def rel-option-iff split*: *option.split*)

lemma *dom-parametric*:
assumes [*transfer-rule*]: *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set A*) *dom dom*
unfolding *dom-def* [*abs-def*] *Option.is-none-def* [*symmetric*] **by** *transfer-prover*

lemma *graph-parametric*:
assumes *bi-total A*
shows ((*A* ===> *rel-option B*) ===> *rel-set* (*rel-prod A B*)) *Map.graph Map.graph*
proof
fix *f g* **assume** (*A* ===> *rel-option B*) *f g*
with *assms*[*unfolded bi-total-def*] **show** *rel-set* (*rel-prod A B*) (*Map.graph f*) (*Map.graph g*)
unfolding *graph-def rel-set-def rel-fun-def*
by *auto* (*metis option-rel-Some1 option-rel-Some2*)+
qed

lemma *map-of-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *R1* ===> *rel-option R2*) *map-of map-of*
unfolding *map-of-def* **by** *transfer-prover*

lemma *map-entry-parametric* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique A*
shows (*A* ===> (*B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> *A* ===> *rel-option B*)
(λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*)))) (λ*k f m*. (*case m k of None* ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))))
by *transfer-prover*

lemma *tabulate-parametric*:
assumes [*transfer-rule*]: *bi-unique A*
shows (*list-all2 A* ===> (*A* ===> *B*) ===> *A* ===> *rel-option B*)
(λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*))) (λ*ks f*. (*map-of* (*map* (λ*k*. (*k*, *f k*)) *ks*)))
by *transfer-prover*

lemma *bulkload-parametric*:
(*list-all2 A* ===> *HOL.eq* ===> *rel-option A*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
(λ*xs k*. *if k* < *length xs then Some* (*xs* ! *k*) *else None*)
proof
fix *xs ys*
assume *list-all2 A xs ys*
then show
(*HOL.eq* ===> *rel-option A*)

(λ*k*. *if k < length xs then Some* (*xs* ! *k*) *else None*)
(λ*k*. *if k < length ys then Some* (*ys* ! *k*) *else None*)
by *induct* (*auto simp add*: *list-all2-lengthD list-all2-nthD rel-funI*)
qed

lemma *map-parametric*:
((*A* ===> *B*) ===> (*C* ===> *D*) ===> (*B* ===> *rel-option C*) ===> *A* ===> *rel-option D*)
(λ*f g m*. (*map-option g* ∘ *m* ∘ *f*)) (λ*f g m*. (*map-option g* ∘ *m* ∘ *f*))
by *transfer-prover*

lemma *combine-with-key-parametric*:
((*A* ===> *B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options* (*f x*) (*m1 x*) (*m2 x*))
unfolding *combine-options-def* **by** *transfer-prover*

lemma *combine-parametric*:
((*B* ===> *B* ===> *B*) ===> (*A* ===> *rel-option B*) ===> (*A* ===> *rel-option B*) ===>
(*A* ===> *rel-option B*)) (λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
(λ*f m1 m2 x*. *combine-options f* (*m1 x*) (*m2 x*))
unfolding *combine-options-def* **by** *transfer-prover*

end

65.2 Type definition and primitive operations

typedef ('*a*, '*b*) *mapping* = *UNIV* :: ('*a* ⇀ '*b*) *set*
morphisms *rep Mapping* **..**

setup-lifting *type-definition-mapping*

lift-definition *empty* :: ('*a*, '*b*) *mapping*
is *Map.empty* **parametric** *empty-parametric* **.**

lift-definition *lookup* :: ('*a*, '*b*) *mapping* ⇒ '*a* ⇒ '*b option*
is λ*m k*. *m k* **parametric** *lookup-parametric* **.**

definition *lookup-default d m k* = (*case Mapping.lookup m k of None* ⇒ *d* | *Some v* ⇒ *v*)

lift-definition *update* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k v m*. *m*(*k* ↦ *v*) **parametric** *update-parametric* **.**

lift-definition *delete* :: '*a* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is λ*k m*. *m*(*k* := *None*) **parametric** *delete-parametric* **.**

lift-definition *filter* :: $('a \Rightarrow 'b \Rightarrow bool) \Rightarrow ('a, 'b)\ mapping \Rightarrow ('a, 'b)\ mapping$
is $\lambda P\ m\ k.\ case\ m\ k\ of\ None \Rightarrow None \mid Some\ v \Rightarrow if\ P\ k\ v\ then\ Some\ v\ else\ None$
.

lift-definition *keys* :: $('a, 'b)\ mapping \Rightarrow 'a\ set$
is *dom* **parametric** *dom-parametric* .

lift-definition *entries* :: $('a, 'b)\ mapping \Rightarrow ('a \times 'b)\ set$
is *Map.graph* **parametric** *graph-parametric* .

lift-definition *tabulate* :: $'a\ list \Rightarrow ('a \Rightarrow 'b) \Rightarrow ('a, 'b)\ mapping$
is $\lambda ks\ f.\ (map\text{-}of\ (List.map\ (\lambda k.\ (k, f\ k))\ ks))$ **parametric** *tabulate-parametric* .

lift-definition *bulkload* :: $'a\ list \Rightarrow (nat, 'a)\ mapping$
is $\lambda xs\ k.\ if\ k < length\ xs\ then\ Some\ (xs\ !\ k)\ else\ None$ **parametric** *bulkload-parametric* .

lift-definition *map* :: $('c \Rightarrow 'a) \Rightarrow ('b \Rightarrow 'd) \Rightarrow ('a, 'b)\ mapping \Rightarrow ('c, 'd)\ mapping$
is $\lambda f\ g\ m.\ (map\text{-}option\ g \circ m \circ f)$ **parametric** *map-parametric* .

lift-definition *map-values* :: $('c \Rightarrow 'a \Rightarrow 'b) \Rightarrow ('c, 'a)\ mapping \Rightarrow ('c, 'b)\ mapping$
is $\lambda f\ m\ x.\ map\text{-}option\ (f\ x)\ (m\ x)$.

lift-definition *combine-with-key* ::
$('a \Rightarrow 'b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a,'b)\ mapping \Rightarrow ('a,'b)\ mapping \Rightarrow ('a,'b)\ mapping$
is $\lambda f\ m1\ m2\ x.\ combine\text{-}options\ (f\ x)\ (m1\ x)\ (m2\ x)$ **parametric** *combine-with-key-parametric*
.

lift-definition *combine* ::
$('b \Rightarrow 'b \Rightarrow 'b) \Rightarrow ('a,'b)\ mapping \Rightarrow ('a,'b)\ mapping \Rightarrow ('a,'b)\ mapping$
is $\lambda f\ m1\ m2\ x.\ combine\text{-}options\ f\ (m1\ x)\ (m2\ x)$ **parametric** *combine-parametric*
.

definition $All\text{-}mapping\ m\ P \longleftrightarrow$
$(\forall x.\ case\ Mapping.lookup\ m\ x\ of\ None \Rightarrow True \mid Some\ y \Rightarrow P\ x\ y)$

declare [[*code drop*: *map*]]

65.3 Functorial structure

functor *map*: *map*
by (*transfer*, *auto simp add*: *fun-eq-iff option.map-comp option.map-id*)+

65.4 Derived operations

definition *ordered-keys* :: $('a{::}linorder, 'b)\ mapping \Rightarrow 'a\ list$
where $ordered\text{-}keys\ m = (if\ finite\ (keys\ m)\ then\ sorted\text{-}list\text{-}of\text{-}set\ (keys\ m)\ else\ [])$

definition *ordered-entries* :: $('a{::}linorder, 'b)\ mapping \Rightarrow ('a \times 'b)\ list$

where *ordered-entries m* = (*if finite* (*entries m*) *then sorted-key-list-of-set fst* (*entries m*)
else [])

definition *fold* :: ('*a*::*linorder* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *mapping* ⇒ '*c* ⇒ '*c*
where *fold f m a* = *List.fold* (*case-prod f*) (*ordered-entries m*) *a*

definition *is-empty* :: ('*a*, '*b*) *mapping* ⇒ *bool*
where *is-empty m* ⟷ *keys m* = {}

definition *size* :: ('*a*, '*b*) *mapping* ⇒ *nat*
where *size m* = (*if finite* (*keys m*) *then card* (*keys m*) *else 0*)

definition *replace* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *replace k v m* = (*if k* ∈ *keys m then update k v m else m*)

definition *default* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *default k v m* = (*if k* ∈ *keys m then m else update k v m*)

Manual derivation of transfer rule is non-trivial

lift-definition *map-entry* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
is
λk f m.
(*case m k of*
None ⇒ *m*
| *Some v* ⇒ *m* (*k* ↦ (*f v*))) **parametric** *map-entry-parametric* **.**

lemma *map-entry-code* [*code*]:
map-entry k f m =
(*case lookup m k of*
None ⇒ *m*
| *Some v* ⇒ *update k* (*f v*) *m*)
by *transfer rule*

definition *map-default* :: '*a* ⇒ '*b* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *mapping* ⇒ ('*a*, '*b*) *mapping*
where *map-default k v f m* = *map-entry k f* (*default k v m*)

definition *of-alist* :: ('*k* × '*v*) *list* ⇒ ('*k*, '*v*) *mapping*
where *of-alist xs* = *foldr* (*λ*(*k*, *v*) *m*. *update k v m*) *xs empty*

instantiation *mapping* :: (*type*, *type*) *equal*
begin

definition *HOL.equal m1 m2* ⟷ (∀ *k*. *lookup m1 k* = *lookup m2 k*)

instance
proof
show ⋀*x y*::('*a*, '*b*) *mapping*. *equal-class.equal x y* = (*x* = *y*)

unfolding *equal-mapping-def*
by *transfer auto*
qed

end

context includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-total A*
and [*transfer-rule*]: *bi-unique B*
shows (*pcr-mapping A B* ===> *pcr-mapping A B* ===> (=)) *HOL.eq HOL.equal*
unfolding *equal* **by** *transfer-prover*

lemma *of-alist-transfer* [*transfer-rule*]:
assumes [*transfer-rule*]: *bi-unique R1*
shows (*list-all2* (*rel-prod R1 R2*) ===> *pcr-mapping R1 R2*) *map-of of-alist*
unfolding *of-alist-def* [*abs-def*] *map-of-foldr* [*abs-def*] **by** *transfer-prover*

end

65.5 Properties

lemma *mapping-eqI*: $(\bigwedge x.\ lookup\ m\ x = lookup\ m'\ x) \Longrightarrow m = m'$
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *mapping-eqI'*:
assumes $\bigwedge x.\ x \in Mapping.keys\ m \Longrightarrow Mapping.lookup\text{-}default\ d\ m\ x = Mapping.lookup\text{-}default\ d\ m'\ x$
and *Mapping.keys m* = *Mapping.keys m'*
shows *m* = *m'*
proof (*intro mapping-eqI*)
show *Mapping.lookup m x* = *Mapping.lookup m' x* **for** *x*
proof (*cases Mapping.lookup m x*)
case *None*
then have $x \notin Mapping.keys\ m$
by *transfer* (*simp add*: *dom-def*)
then have $x \notin Mapping.keys\ m'$
by (*simp add*: *assms*)
then have *Mapping.lookup m' x* = *None*
by *transfer* (*simp add*: *dom-def*)
with *None* **show** *?thesis*
by *simp*
next
case (*Some y*)
then have *A*: $x \in Mapping.keys\ m$
by *transfer* (*simp add*: *dom-def*)
then have $x \in Mapping.keys\ m'$

```
    by (simp add: assms)
  then have ∃ y'. Mapping.lookup m' x = Some y'
    by transfer (simp add: dom-def)
  with Some assms(1)[OF A] show ?thesis
    by (auto simp add: lookup-default-def)
 qed
qed

lemma lookup-update[simp]: lookup (update k v m) k = Some v
 by transfer simp

lemma lookup-update-neq[simp]: k ≠ k' ⟹ lookup (update k v m) k' = lookup m
k'
 by transfer simp

lemma lookup-update': lookup (update k v m) k' = (if k = k' then Some v else
lookup m k')
 by transfer simp

lemma lookup-empty[simp]: lookup empty k = None
 by transfer simp

lemma lookup-delete[simp]: lookup (delete k m) k = None
 by transfer simp

lemma lookup-delete-neq[simp]: k ≠ k' ⟹ lookup (delete k m) k' = lookup m k'
 by transfer simp

lemma lookup-filter:
 lookup (filter P m) k =
  (case lookup m k of
    None ⇒ None
  | Some v ⇒ if P k v then Some v else None)
 by transfer simp-all

lemma lookup-map-values: lookup (map-values f m) k = map-option (f k) (lookup
m k)
 by transfer simp-all

lemma lookup-default-empty: lookup-default d empty k = d
 by (simp add: lookup-default-def lookup-empty)

lemma lookup-default-update: lookup-default d (update k v m) k = v
 by (simp add: lookup-default-def)

lemma lookup-default-update-neq:
 k ≠ k' ⟹ lookup-default d (update k v m) k' = lookup-default d m k'
 by (simp add: lookup-default-def)
```

lemma *lookup-default-update'*:
lookup-default d (update k v m) k' = (if k = k' then v else lookup-default d m k')
by (*auto simp*: *lookup-default-update lookup-default-update-neq*)

lemma *lookup-default-filter*:
lookup-default d (filter P m) k =
(if P k (lookup-default d m k) then lookup-default d m k else d)
by (*simp add*: *lookup-default-def lookup-filter split*: *option.splits*)

lemma *lookup-default-map-values*:
lookup-default (f k d) (map-values f m) k = f k (lookup-default d m k)
by (*simp add*: *lookup-default-def lookup-map-values split*: *option.splits*)

lemma *lookup-combine-with-key*:
Mapping.lookup (combine-with-key f m1 m2) x =
combine-options (f x) (Mapping.lookup m1 x) (Mapping.lookup m2 x)
by *transfer* (*auto split*: *option.splits*)

lemma *combine-altdef*: *combine f m1 m2 = combine-with-key (λ-. f) m1 m2*
by *transfer'* (*rule refl*)

lemma *lookup-combine*:
Mapping.lookup (combine f m1 m2) x =
combine-options f (Mapping.lookup m1 x) (Mapping.lookup m2 x)
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-default-neutral-combine-with-key*:
assumes $\bigwedge x.$ *f k d x = x* $\bigwedge x.$ *f k x d = x*
shows *Mapping.lookup-default d (combine-with-key f m1 m2) k =*
f k (Mapping.lookup-default d m1 k) (Mapping.lookup-default d m2 k)
by (*auto simp*: *lookup-default-def lookup-combine-with-key assms split*: *option.splits*)

lemma *lookup-default-neutral-combine*:
assumes $\bigwedge x.$ *f d x = x* $\bigwedge x.$ *f x d = x*
shows *Mapping.lookup-default d (combine f m1 m2) x =*
f (Mapping.lookup-default d m1 x) (Mapping.lookup-default d m2 x)
by (*auto simp*: *lookup-default-def lookup-combine assms split*: *option.splits*)

lemma *lookup-map-entry*: *lookup (map-entry x f m) x = map-option f (lookup m x)*
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-map-entry-neq*: $x \neq y \Longrightarrow$ *lookup (map-entry x f m) y = lookup m y*
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-map-entry'*:
lookup (map-entry x f m) y =
(if x = y then map-option f (lookup m y) else lookup m y)

by *transfer* (*auto split*: *option.splits*)

lemma *lookup-default*: *lookup* (*default x d m*) *x* = *Some* (*lookup-default d m x*)
unfolding *lookup-default-def default-def*
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-default-neq*: *x* ≠ *y* ⟹ *lookup* (*default x d m*) *y* = *lookup m y*
unfolding *lookup-default-def default-def*
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-default'*:
lookup (*default x d m*) *y* =
(*if x* = *y then Some* (*lookup-default d m x*) *else lookup m y*)
unfolding *lookup-default-def default-def*
by *transfer* (*auto split*: *option.splits*)

lemma *lookup-map-default*: *lookup* (*map-default x d f m*) *x* = *Some* (*f* (*lookup-default d m x*))
unfolding *lookup-default-def default-def*
by (*simp add*: *map-default-def lookup-map-entry lookup-default lookup-default-def*)

lemma *lookup-map-default-neq*: *x* ≠ *y* ⟹ *lookup* (*map-default x d f m*) *y* = *lookup m y*
unfolding *lookup-default-def default-def*
by (*simp add*: *map-default-def lookup-map-entry-neq lookup-default-neq*)

lemma *lookup-map-default'*:
lookup (*map-default x d f m*) *y* =
(*if x* = *y then Some* (*f* (*lookup-default d m x*)) *else lookup m y*)
unfolding *lookup-default-def default-def*
by (*simp add*: *map-default-def lookup-map-entry' lookup-default' lookup-default-def*)

lemma *lookup-tabulate*:
assumes *distinct xs*
shows *Mapping.lookup* (*Mapping.tabulate xs f*) *x* = (*if x* ∈ *set xs then Some* (*f x*) *else None*)
using *assms* **by** *transfer* (*auto simp*: *map-of-eq-None-iff o-def dest*!: *map-of-SomeD*)

lemma *lookup-of-alist*: *lookup* (*of-alist xs*) *k* = *map-of xs k*
by *transfer simp-all*

lemma *keys-is-none-rep* [*code-unfold*]: *k* ∈ *keys m* ⟷ ¬ (*Option.is-none* (*lookup m k*))
by *transfer* (*auto simp add*: *Option.is-none-def*)

lemma *update-update*:
update k v (*update k w m*) = *update k v m*
k ≠ *l* ⟹ *update k v* (*update l w m*) = *update l w* (*update k v m*)
by (*transfer*; *simp add*: *fun-upd-twist*)+

lemma *update-delete* [*simp*]: *update k v* (*delete k m*) = *update k v m*
by *transfer simp*

lemma *delete-update*:
delete k (*update k v m*) = *delete k m*
k ≠ *l* ⟹ *delete k* (*update l v m*) = *update l v* (*delete k m*)
by (*transfer*; *simp add*: *fun-upd-twist*)+

lemma *delete-empty* [*simp*]: *delete k empty* = *empty*
by *transfer simp*

lemma *Mapping-delete-if-notin-keys*[*simp*]:
k ∉ *keys m* ⟹ *delete k m* = *m*
by *transfer simp*

lemma *replace-update*:
k ∉ *keys m* ⟹ *replace k v m* = *m*
k ∈ *keys m* ⟹ *replace k v m* = *update k v m*
by (*transfer*; *auto simp add*: *replace-def fun-upd-twist*)+

lemma *map-values-update*: *map-values f* (*update k v m*) = *update k* (*f k v*) (*map-values f m*)
by *transfer* (*simp-all add*: *fun-eq-iff*)

lemma *size-mono*: *finite* (*keys m'*) ⟹ *keys m* ⊆ *keys m'* ⟹ *size m* ≤ *size m'*
unfolding *size-def* **by** (*auto intro*: *card-mono*)

lemma *size-empty* [*simp*]: *size empty* = *0*
unfolding *size-def* **by** *transfer simp*

lemma *size-update*:
finite (*keys m*) ⟹ *size* (*update k v m*) =
(*if k* ∈ *keys m then size m else Suc* (*size m*))
unfolding *size-def* **by** *transfer* (*auto simp add*: *insert-dom*)

lemma *size-delete*: *size* (*delete k m*) = (*if k* ∈ *keys m then size m* − *1 else size m*)
unfolding *size-def* **by** *transfer simp*

lemma *size-tabulate* [*simp*]: *size* (*tabulate ks f*) = *length* (*remdups ks*)
unfolding *size-def* **by** *transfer* (*auto simp add*: *map-of-map-restrict card-set comp-def*)

lemma *keys-filter*: *keys* (*filter P m*) ⊆ *keys m*
by *transfer* (*auto split*: *option.splits*)

lemma *size-filter*: *finite* (*keys m*) ⟹ *size* (*filter P m*) ≤ *size m*
by (*intro size-mono keys-filter*)

lemma *bulkload-tabulate*: *bulkload xs* = *tabulate* [*0*..<*length xs*] (*nth xs*)
by *transfer* (*auto simp add*: *map-of-map-restrict*)

lemma *is-empty-empty* [*simp*]: *is-empty empty*
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-update* [*simp*]: ¬ *is-empty* (*update k v m*)
unfolding *is-empty-def* **by** *transfer simp*

lemma *is-empty-delete*: *is-empty* (*delete k m*) ⟷ *is-empty m* ∨ *keys m* = {*k*}
unfolding *is-empty-def* **by** *transfer* (*auto simp del*: *dom-eq-empty-conv*)

lemma *is-empty-replace* [*simp*]: *is-empty* (*replace k v m*) ⟷ *is-empty m*
unfolding *is-empty-def replace-def* **by** *transfer auto*

lemma *is-empty-default* [*simp*]: ¬ *is-empty* (*default k v m*)
unfolding *is-empty-def default-def* **by** *transfer auto*

lemma *is-empty-map-entry* [*simp*]: *is-empty* (*map-entry k f m*) ⟷ *is-empty m*
unfolding *is-empty-def* **by** *transfer* (*auto split*: *option.split*)

lemma *is-empty-map-values* [*simp*]: *is-empty* (*map-values f m*) ⟷ *is-empty m*
unfolding *is-empty-def* **by** *transfer* (*auto simp*: *fun-eq-iff*)

lemma *is-empty-map-default* [*simp*]: ¬ *is-empty* (*map-default k v f m*)
by (*simp add*: *map-default-def*)

lemma *keys-dom-lookup*: *keys m* = *dom* (*Mapping.lookup m*)
by *transfer rule*

lemma *keys-empty* [*simp*]: *keys empty* = {}
by *transfer* (*fact dom-empty*)

lemma *in-keysD*: *k* ∈ *keys m* ⟹ ∃ *v*. *lookup m k* = *Some v*
by *transfer* (*fact domD*)

lemma *keys-update* [*simp*]: *keys* (*update k v m*) = *insert k* (*keys m*)
by *transfer simp*

lemma *keys-delete* [*simp*]: *keys* (*delete k m*) = *keys m* − {*k*}
by *transfer simp*

lemma *keys-replace* [*simp*]: *keys* (*replace k v m*) = *keys m*
unfolding *replace-def* **by** *transfer* (*simp add*: *insert-absorb*)

lemma *keys-default* [*simp*]: *keys* (*default k v m*) = *insert k* (*keys m*)
unfolding *default-def* **by** *transfer* (*simp add*: *insert-absorb*)

lemma *keys-map-entry* [*simp*]: *keys* (*map-entry k f m*) = *keys m*

by *transfer* (*auto split*: *option.split*)

lemma *keys-map-default* [*simp*]: *keys* (*map-default k v f m*) = *insert k* (*keys m*)
by (*simp add*: *map-default-def*)

lemma *keys-map-values* [*simp*]: *keys* (*map-values f m*) = *keys m*
by *transfer* (*simp-all add*: *dom-def*)

lemma *keys-combine-with-key* [*simp*]:
Mapping.keys (*combine-with-key f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
by *transfer* (*auto simp*: *dom-def combine-options-def split*: *option.splits*)

lemma *keys-combine* [*simp*]: *Mapping.keys* (*combine f m1 m2*) = *Mapping.keys m1* ∪ *Mapping.keys m2*
by (*simp add*: *combine-altdef*)

lemma *keys-tabulate* [*simp*]: *keys* (*tabulate ks f*) = *set ks*
by *transfer* (*simp add*: *map-of-map-restrict o-def*)

lemma *keys-of-alist* [*simp*]: *keys* (*of-alist xs*) = *set* (*List.map fst xs*)
by *transfer* (*simp-all add*: *dom-map-of-conv-image-fst*)

lemma *keys-bulkload* [*simp*]: *keys* (*bulkload xs*) = {*0..<length xs*}
by (*simp add*: *bulkload-tabulate*)

lemma *finite-keys-update*[*simp*]:
finite (*keys* (*update k v m*)) = *finite* (*keys m*)
by *transfer simp*

lemma *set-ordered-keys*[*simp*]:
finite (*Mapping.keys m*) ⟹ *set* (*Mapping.ordered-keys m*) = *Mapping.keys m*
unfolding *ordered-keys-def* **by** *transfer auto*

lemma *distinct-ordered-keys* [*simp*]: *distinct* (*ordered-keys m*)
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-infinite* [*simp*]: ¬ *finite* (*keys m*) ⟹ *ordered-keys m* = []
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-empty* [*simp*]: *ordered-keys empty* = []
by (*simp add*: *ordered-keys-def*)

lemma *sorted-ordered-keys*[*simp*]: *sorted* (*ordered-keys m*)
unfolding *ordered-keys-def* **by** *simp*

lemma *ordered-keys-update* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*update k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹

ordered-keys (*update k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *ordered-keys-def*)
(*auto simp only*: *sorted-list-of-set-insert-remove*[*symmetric*] *insert-absorb*)

lemma *ordered-keys-delete* [*simp*]: *ordered-keys* (*delete k m*) = *remove1 k* (*ordered-keys m*)
proof (*cases finite* (*keys m*))
case *False*
then show *?thesis* **by** *simp*
next
case *fin*: *True*
show *?thesis*
proof (*cases k* ∈ *keys m*)
case *False*
with *fin* **have** *k* ∉ *set* (*sorted-list-of-set* (*keys m*))
by *simp*
with *False* **show** *?thesis*
by (*simp add*: *ordered-keys-def remove1-idem*)
next
case *True*
with *fin* **show** *?thesis*
by (*simp add*: *ordered-keys-def sorted-list-of-set-remove*)
qed
qed

lemma *ordered-keys-replace* [*simp*]: *ordered-keys* (*replace k v m*) = *ordered-keys m*
by (*simp add*: *replace-def*)

lemma *ordered-keys-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*default k v m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*default k v m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *default-def*)

lemma *ordered-keys-map-entry* [*simp*]: *ordered-keys* (*map-entry k f m*) = *ordered-keys m*
by (*simp add*: *ordered-keys-def*)

lemma *ordered-keys-map-default* [*simp*]:
k ∈ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *ordered-keys m*
finite (*keys m*) ⟹ *k* ∉ *keys m* ⟹ *ordered-keys* (*map-default k v f m*) = *insort k* (*ordered-keys m*)
by (*simp-all add*: *map-default-def*)

lemma *ordered-keys-tabulate* [*simp*]: *ordered-keys* (*tabulate ks f*) = *sort* (*remdups ks*)
by (*simp add*: *ordered-keys-def sorted-list-of-set-sort-remdups*)

lemma *ordered-keys-bulkload* [*simp*]: *ordered-keys* (*bulkload ks*) = [*0*..<*length ks*]

by (*simp add*: *ordered-keys-def*)

lemma *tabulate-fold*: *tabulate xs f* = *List.fold* (λ*k m*. *update k* (*f k*) *m*) *xs empty*
proof *transfer*
fix *f* :: '*a* ⇒ '*b* **and** *xs*
have *map-of* (*List.map* (λ*k*. (*k*, *f k*)) *xs*) = *foldr* (λ*k m*. *m*(*k* ↦ *f k*)) *xs Map.empty*
by (*simp add*: *foldr-map comp-def map-of-foldr*)
also have *foldr* (λ*k m*. *m*(*k* ↦ *f k*)) *xs* = *List.fold* (λ*k m*. *m*(*k* ↦ *f k*)) *xs*
by (*rule foldr-fold*) (*simp add*: *fun-eq-iff*)
ultimately show *map-of* (*List.map* (λ*k*. (*k*, *f k*)) *xs*) = *List.fold* (λ*k m*. *m*(*k* ↦ *f k*)) *xs Map.empty*
by *simp*
qed

lemma *All-mapping-mono*:
(⋀*k v*. *k* ∈ *keys m* ⟹ *P k v* ⟹ *Q k v*) ⟹ *All-mapping m P* ⟹ *All-mapping m Q*
unfolding *All-mapping-def* **by** *transfer* (*auto simp*: *All-mapping-def dom-def split*: *option.splits*)

lemma *All-mapping-empty* [*simp*]: *All-mapping Mapping.empty P*
by (*auto simp*: *All-mapping-def lookup-empty*)

lemma *All-mapping-update-iff*:
All-mapping (*Mapping.update k v m*) *P* ⟷ *P k v* ∧ *All-mapping m* (λ*k*′ *v*′. *k* = *k*′ ∨ *P k*′ *v*′)
unfolding *All-mapping-def*
proof *safe*
assume ∀*x*. *case Mapping.lookup* (*Mapping.update k v m*) *x of None* ⇒ *True* | *Some y* ⇒ *P x y*
then have ∗: *case Mapping.lookup* (*Mapping.update k v m*) *x of None* ⇒ *True* | *Some y* ⇒ *P x y* **for** *x*
by *blast*
from ∗[*of k*] **show** *P k v*
by (*simp add*: *lookup-update*)
show *case Mapping.lookup m x of None* ⇒ *True* | *Some v*′ ⇒ *k* = *x* ∨ *P x v*′ **for** *x*
using ∗[*of x*] **by** (*auto simp add*: *lookup-update*′ *split*: *if-splits option.splits*)
next
assume *P k v*
assume ∀*x*. *case Mapping.lookup m x of None* ⇒ *True* | *Some v*′ ⇒ *k* = *x* ∨ *P x v*′
then have *A*: *case Mapping.lookup m x of None* ⇒ *True* | *Some v*′ ⇒ *k* = *x* ∨ *P x v*′ **for** *x*
by *blast*
show *case Mapping.lookup* (*Mapping.update k v m*) *x of None* ⇒ *True* | *Some xa* ⇒ *P x xa* **for** *x*
using ‹*P k v*› *A*[*of x*] **by** (*auto simp*: *lookup-update*′ *split*: *option.splits*)

qed

lemma *All-mapping-update*:
P k v ⟹ *All-mapping m* (λ*k' v'*. *k* = *k'* ∨ *P k' v'*) ⟹ *All-mapping* (*Mapping.update k v m*) *P*
by (*simp add*: *All-mapping-update-iff*)

lemma *All-mapping-filter-iff*: *All-mapping* (*filter P m*) *Q* ⟷ *All-mapping m* (λ*k v*. *P k v* ⟶ *Q k v*)
by (*auto simp*: *All-mapping-def lookup-filter split*: *option.splits*)

lemma *All-mapping-filter*: *All-mapping m Q* ⟹ *All-mapping* (*filter P m*) *Q*
by (*auto simp*: *All-mapping-filter-iff intro*: *All-mapping-mono*)

lemma *All-mapping-map-values*: *All-mapping* (*map-values f m*) *P* ⟷ *All-mapping m* (λ*k v*. *P k* (*f k v*))
by (*auto simp*: *All-mapping-def lookup-map-values split*: *option.splits*)

lemma *All-mapping-tabulate*: (∀ *x*∈*set xs*. *P x* (*f x*)) ⟹ *All-mapping* (*Mapping.tabulate xs f*) *P*
unfolding *All-mapping-def*
by *transfer* (*auto split*: *option.split dest*!: *map-of-SomeD*)

lemma *All-mapping-alist*:
(⋀*k v*. (*k*, *v*) ∈ *set xs* ⟹ *P k v*) ⟹ *All-mapping* (*Mapping.of-alist xs*) *P*
by (*auto simp*: *All-mapping-def lookup-of-alist dest*!: *map-of-SomeD split*: *option.splits*)

lemma *combine-empty* [*simp*]: *combine f Mapping.empty y* = *y combine f y Mapping.empty* = *y*
by (*transfer*; *force*)+

lemma (**in** *abel-semigroup*) *comm-monoid-set-combine*: *comm-monoid-set* (*combine f*) *Mapping.empty*
by *standard* (*transfer fixing*: *f*, *simp add*: *combine-options-ac*[*of f*] *ac-simps*)+

locale *combine-mapping-abel-semigroup* = *abel-semigroup*
begin

sublocale *combine*: *comm-monoid-set combine f Mapping.empty*
by (*rule comm-monoid-set-combine*)

lemma *fold-combine-code*:
combine.F g (*set xs*) = *foldr* (λ*x*. *combine f* (*g x*)) (*remdups xs*) *Mapping.empty*
proof −
have *combine.F g* (*set xs*) = *foldr* (λ*x*. *combine f* (*g x*)) *xs Mapping.empty*
if *distinct xs* **for** *xs*
using *that* **by** (*induction xs*) *simp-all*
from *this*[*of remdups xs*] **show** *?thesis* **by** *simp*

qed

lemma *keys-fold-combine*: *finite A* $\Longrightarrow$ *Mapping.keys* (*combine.F g A*) = ($\bigcup x{\in}A$. *Mapping.keys* (*g x*))
  by (*induct A rule*: *finite-induct*) *simp-all*

end

65.5.1 *entries*, *ordered-entries*, and *fold*

context *linorder*
begin

sublocale *folding-Map-graph*: *folding-insort-key* ($\leq$) ($<$) *Map.graph m fst* **for** *m*
  by *unfold-locales* (*fact inj-on-fst-graph*)

end

lemma *sorted-fst-list-of-set-insort-Map-graph*[*simp*]:
  assumes *finite* (*dom m*) *fst x* $\notin$ *dom m*
  shows *sorted-key-list-of-set fst* (*insert x* (*Map.graph m*))
      = *insort-key fst x* (*sorted-key-list-of-set fst* (*Map.graph m*))
proof(*cases x*)
  case (*Pair k v*)
  with ‹*fst x* $\notin$ *dom m*› **have** *Map.graph m* $\subseteq$ *Map.graph* (*m*($k \mapsto v$))
    by(*auto simp*: *graph-def*)
  moreover from *Pair* ‹*fst x* $\notin$ *dom m*› **have** (*k*, *v*) $\notin$ *Map.graph m*
    using *graph-domD* **by** *fastforce*
  ultimately show *?thesis*
   using *Pair assms folding-Map-graph.sorted-key-list-of-set-insert*[**where** *?m*=*m*($k \mapsto v$)]
    by *auto*
qed

lemma *sorted-fst-list-of-set-insort-insert-Map-graph*[*simp*]:
  assumes *finite* (*dom m*) *fst x* $\notin$ *dom m*
  shows *sorted-key-list-of-set fst* (*insert x* (*Map.graph m*))
      = *insort-insert-key fst x* (*sorted-key-list-of-set fst* (*Map.graph m*))
proof(*cases x*)
  case (*Pair k v*)
  with ‹*fst x* $\notin$ *dom m*› **have** *Map.graph m* $\subseteq$ *Map.graph* (*m*($k \mapsto v$))
    by(*auto simp*: *graph-def*)
  with *assms Pair* **show** *?thesis*
   unfolding *sorted-fst-list-of-set-insort-Map-graph*[*OF assms*] *insort-insert-key-def*
    using *folding-Map-graph.set-sorted-key-list-of-set in-graphD* **by** (*fastforce split*: *if-splits*)
qed

lemma *linorder-finite-Map-induct*[*consumes 1*, *case-names empty update*]:

fixes $m :: 'a::linorder \rightharpoonup 'b$
assumes *finite* (*dom m*)
assumes *P Map.empty*
assumes $\bigwedge k\ v\ m.\ [\![$ *finite* (*dom m*); $k \notin$ *dom m*; $(\bigwedge k'.\ k' \in$ *dom m* $\Longrightarrow k' \leq k)$; *P m* $]\!]$
$\Longrightarrow$ *P* $(m(k \mapsto v))$
shows *P m*
proof −
let *?key-list* = $\lambda m.$ *sorted-list-of-set* (*dom m*)
from *assms*(*1*,*2*) **show** *?thesis*
proof(*induction length* (*?key-list m*) *arbitrary*: *m*)
case *0*
then have *sorted-list-of-set* (*dom m*) = []
by *auto*
with ‹*finite* (*dom m*)› **have** *m* = *Map.empty*
by *auto*
with ‹*P Map.empty*› **show** *?case* **by** *simp*
next
case (*Suc n*)
then obtain *x xs* **where** *x-xs*: *sorted-list-of-set* (*dom m*) = *xs* @ [*x*]
by (*metis append-butlast-last-id length-greater-0-conv zero-less-Suc*)
have *sorted-list-of-set* (*dom* (*m*(*x* := *None*))) = *xs*
proof −
have *distinct* (*xs* @ [*x*])
by (*metis sorted-list-of-set.distinct-sorted-key-list-of-set x-xs*)
then have *remove1 x* (*xs* @ [*x*]) = *xs*
by (*simp add*: *remove1-append*)
with ‹*finite* (*dom m*)› *x-xs* **show** *?thesis*
by (*simp add*: *sorted-list-of-set-remove*)
qed
moreover have *k* ≤ *x* **if** *k* ∈ *dom* (*m*(*x* := *None*)) **for** *k*
proof −
from *x-xs* **have** *sorted* (*xs* @ [*x*])
by (*metis sorted-list-of-set.sorted-sorted-key-list-of-set*)
moreover from ‹*k* ∈ *dom* (*m*(*x* := *None*))› **have** *k* ∈ *set xs*
using ‹*finite* (*dom m*)› ‹*sorted-list-of-set* (*dom* (*m*(*x* := *None*))) = *xs*›
by *auto*
ultimately show *k* ≤ *x*
by (*simp add*: *sorted-append*)
qed
moreover from ‹*finite* (*dom m*)› **have** *finite* (*dom* (*m*(*x* := *None*))) *x* ∉ *dom* (*m*(*x* := *None*))
by *simp-all*
moreover have *P* (*m*(*x* := *None*))
using *Suc* ‹*sorted-list-of-set* (*dom* (*m*(*x* := *None*))) = *xs*› *x-xs* **by** *auto*
ultimately show *?case*
using *assms*(*3*)[**where** *?m*=*m*(*x* := *None*)] **by** (*metis fun-upd-triv fun-upd-upd not-Some-eq*)
qed

qed

lemma *delete-insert-fst*[*simp*]: *AList.delete k* (*insert-key fst* (*k*, *v*) *xs*) = *AList.delete k xs*
by (*induction xs*) *simp-all*

lemma *insert-fst-delete*: ⟦ *fst x* ≠ *k2*; *sorted* (*List.map fst xs*) ⟧
⟹ *insert-key fst x* (*AList.delete k2 xs*) = *AList.delete k2* (*insert-key fst x xs*)
by (*induction xs*) (*fastforce simp add*: *insert-is-Cons order-trans*)+

lemma *sorted-fst-list-of-set-Map-graph-fun-upd-None*[*simp*]:
sorted-key-list-of-set fst (*Map.graph* (*m*(*k* := *None*)))
= *AList.delete k* (*sorted-key-list-of-set fst* (*Map.graph m*))
proof(*cases finite* (*Map.graph m*))
assume *finite* (*Map.graph m*)
from *this*[*unfolded finite-graph-iff-finite-dom*] **show** *?thesis*
proof(*induction rule*: *finite-Map-induct*)
let *?list-of*=*sorted-key-list-of-set fst*
case (*update k2 v2 m*)
note [*simp*] = ‹*k2* ∉ *dom m*› ‹*finite* (*dom m*)›

have *right-eq*: *AList.delete k* (*?list-of* (*Map.graph* (*m*(*k2* ↦ *v2*))))
= *AList.delete k* (*insert-key fst* (*k2*, *v2*) (*?list-of* (*Map.graph m*)))
by *simp*

show *?case*
proof(*cases k* = *k2*)
case *True*
then have *?list-of* (*Map.graph* ((*m*(*k2* ↦ *v2*))(*k* := *None*)))
= *AList.delete k* (*insert-key fst* (*k2*, *v2*) (*?list-of* (*Map.graph m*)))
using *fst-graph-eq-dom update.IH* **by** *auto*
then show *?thesis*
using *right-eq* **by** *metis*
next
case *False*
then have *AList.delete k* (*insert-key fst* (*k2*, *v2*) (*?list-of* (*Map.graph m*)))
= *insert-key fst* (*k2*, *v2*) (*?list-of* (*Map.graph* (*m*(*k* := *None*))))
by (*auto simp add*: *insert-fst-delete update.IH*
folding-Map-graph.sorted-sorted-key-list-of-set[*OF subset-refl*])
also have ... = *?list-of* (*insert* (*k2*, *v2*) (*Map.graph* (*m*(*k* := *None*))))
by *auto*
also from *False* ‹*k2* ∉ *dom m*› **have** ... = *?list-of* (*Map.graph* ((*m*(*k2* ↦ *v2*))(*k* := *None*)))
by (*metis graph-map-upd domIff fun-upd-triv fun-upd-twist*)
finally show *?thesis* **using** *right-eq* **by** *metis*
qed
qed *simp*
qed *simp*

lemma *entries-empty*[*simp*]: *entries empty* = {}
by *transfer* (*fact graph-empty*)

lemma *entries-lookup*: *entries m* = *Map.graph* (*lookup m*)
by *transfer rule*

lemma *in-entriesI*: *lookup m k* = *Some v* ⟹ (*k*, *v*) ∈ *entries m*
by *transfer* (*fact in-graphI*)

lemma *in-entriesD*: (*k*, *v*) ∈ *entries m* ⟹ *lookup m k* = *Some v*
by *transfer* (*fact in-graphD*)

lemma *fst-image-entries-eq-keys*[*simp*]: *fst* ‘ *Mapping.entries m* = *Mapping.keys m*
by *transfer* (*fact fst-graph-eq-dom*)

lemma *finite-entries-iff-finite-keys*[*simp*]:
finite (*entries m*) = *finite* (*keys m*)
by *transfer* (*fact finite-graph-iff-finite-dom*)

lemma *entries-update*:
entries (*update k v m*) = *insert* (*k*, *v*) (*entries* (*delete k m*))
by *transfer* (*fact graph-map-upd*)

lemma *entries-delete*:
entries (*delete k m*) = {*e* ∈ *entries m*. *fst e* ≠ *k*}
by *transfer* (*fact graph-fun-upd-None*)

lemma *entries-of-alist*[*simp*]:
distinct (*List.map fst xs*) ⟹ *entries* (*of-alist xs*) = *set xs*
by *transfer* (*fact graph-map-of-if-distinct-dom*)

lemma *entries-keysD*:
x ∈ *entries m* ⟹ *fst x* ∈ *keys m*
by *transfer* (*fact graph-domD*)

lemma *set-ordered-entries*[*simp*]:
finite (*keys m*) ⟹ *set* (*ordered-entries m*) = *entries m*
unfolding *ordered-entries-def*
by *transfer* (*auto simp*: *folding-Map-graph.set-sorted-key-list-of-set*[*OF subset-refl*])

lemma *distinct-ordered-entries*[*simp*]: *distinct* (*List.map fst* (*ordered-entries m*))
unfolding *ordered-entries-def*
by *transfer* (*simp add*: *folding-Map-graph.distinct-sorted-key-list-of-set*[*OF subset-refl*])

lemma *sorted-ordered-entries*[*simp*]: *sorted* (*List.map fst* (*ordered-entries m*))
unfolding *ordered-entries-def*
by *transfer* (*auto intro*: *folding-Map-graph.sorted-sorted-key-list-of-set*)

lemma *ordered-entries-infinite*[*simp*]:
  ¬ *finite* (*Mapping.keys m*) ⟹ *ordered-entries m* = []
  by (*simp add*: *ordered-entries-def*)

lemma *ordered-entries-empty*[*simp*]: *ordered-entries empty* = []
  by (*simp add*: *ordered-entries-def*)

lemma *ordered-entries-update*[*simp*]:
  assumes *finite* (*keys m*)
  shows *ordered-entries* (*update k v m*)
   = *insort-insert-key fst* (*k*, *v*) (*AList.delete k* (*ordered-entries m*))
proof −
  let *?list-of*=*sorted-key-list-of-set fst* **and** *?insort*=*insort-insert-key fst*

  have ∗: *?list-of* (*insert* (*k*, *v*) (*Map.graph* (*m*(*k* := *None*))))
   = *?insort* (*k*, *v*) (*AList.delete k* (*?list-of* (*Map.graph m*))) **if** *finite* (*dom m*) **for** *m*
  proof −
   from ‹*finite* (*dom m*)› **have** *?list-of* (*insert* (*k*, *v*) (*Map.graph* (*m*(*k* := *None*))))
     = *?insort* (*k*, *v*) (*?list-of* (*Map.graph* (*m*(*k* := *None*))))
      by (*intro sorted-fst-list-of-set-insort-insert-Map-graph*) (*simp-all add*: *subset-insertI*)
    then show *?thesis* **by** *simp*
  qed
  from *assms* **show** *?thesis*
    unfolding *ordered-entries-def*
    by (*transfer fixing*: *k v*) (*use* ∗ **in** *auto*)
qed

lemma *ordered-entries-delete*[*simp*]:
  ordered-entries (*delete k m*) = *AList.delete k* (*ordered-entries m*)
  unfolding *ordered-entries-def* **by** *transfer auto*

lemma *map-fst-ordered-entries*[*simp*]:
  List.map fst (*ordered-entries m*) = *ordered-keys m*
proof(*cases finite* (*Mapping.keys m*))
  case *True*
 then have *set* (*List.map fst* (*Mapping.ordered-entries m*)) = *set* (*Mapping.ordered-keys m*)
    unfolding *ordered-entries-def ordered-keys-def*
     by (*transfer*) (*simp add*: *folding-Map-graph.set-sorted-key-list-of-set*[*OF subset-refl*] *fst-graph-eq-dom*)
 with *True* **show** *List.map fst* (*Mapping.ordered-entries m*) = *Mapping.ordered-keys m*
   by (*metis distinct-ordered-entries ordered-keys-def sorted-list-of-set.idem-if-sorted-distinct*
        sorted-list-of-set.set-sorted-key-list-of-set sorted-ordered-entries)
next

case *False*
then show *?thesis*
unfolding *ordered-entries-def ordered-keys-def* **by** *simp*
qed

lemma *fold-empty*[*simp*]: *fold f empty a = a*
unfolding *fold-def* **by** *simp*

lemma *insort-key-is-snoc-if-sorted-and-distinct*:
assumes *sorted* (*List.map f xs*) *f y* ∉ *f ' set xs* ∀ *x* ∈ *set xs*. *f x* ≤ *f y*
shows *insort-key f y xs = xs* @ [*y*]
using *assms* **by** (*induction xs*) (*auto dest*!: *insort-is-Cons*)

lemma *fold-update*:
assumes *finite* (*keys m*)
assumes *k* ∉ *keys m* ⋀*k'*. *k'* ∈ *keys m* ⟹ *k'* ≤ *k*
shows *fold f* (*update k v m*) *a = f k v* (*fold f m a*)
proof −
from *assms* **have** *k-notin-entries*: *k* ∉ *fst ' set* (*ordered-entries m*)
using *entries-keysD* **by** *fastforce*
with *assms* **have** *ordered-entries* (*update k v m*)
= insort-insert-key fst (*k*, *v*) (*ordered-entries m*)
by *simp*
also from *k-notin-entries* **have** *... = ordered-entries m* @ [(*k*, *v*)]
proof −
from *assms* **have** ∀ *x* ∈ *set* (*ordered-entries m*). *fst x* ≤ *fst* (*k*, *v*)
unfolding *ordered-entries-def*
by *transfer* (*fastforce simp*: *folding-Map-graph.set-sorted-key-list-of-set*[*OF order-refl*]
dest: *graph-domD*)
from *insort-key-is-snoc-if-sorted-and-distinct*[*OF - - this*] *k-notin-entries* ‹*finite* (*keys m*)›
show *?thesis*
using *sorted-ordered-keys*
unfolding *insort-insert-key-def* **by** *auto*
qed
finally show *?thesis* **unfolding** *fold-def* **by** *simp*
qed

lemma *linorder-finite-Mapping-induct*[*consumes 1*, *case-names empty update*]:
fixes *m* :: ('*a*::*linorder*, '*b*) *mapping*
assumes *finite* (*keys m*)
assumes *P empty*
assumes ⋀*k v m*.
⟦ *finite* (*keys m*); *k* ∉ *keys m*; (⋀*k'*. *k'* ∈ *keys m* ⟹ *k'* ≤ *k*); *P m* ⟧
⟹ *P* (*update k v m*)
shows *P m*
using *assms* **by** *transfer* (*simp add*: *linorder-finite-Map-induct*)

65.6 Code generator setup

hide-const (**open**) *empty is-empty rep lookup lookup-default filter update delete ordered-keys*
keys size replace default map-entry map-default tabulate bulkload map map-values combine of-alist
entries ordered-entries fold

end

66 Monad notation for arbitrary types

theory *Monad-Syntax*
imports *Main*
begin

We provide a convenient do-notation for monadic expressions well-known from Haskell. *Let* is printed specially in do-expressions.

consts
bind :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$ (**infixl** ‹≫=› *54*)

notation (*ASCII*)
bind (**infixl** ‹>>=› *54*)

abbreviation (*do-notation*)
bind-do :: $'a \Rightarrow ('b \Rightarrow 'c) \Rightarrow 'd$
where *bind-do* ≡ *bind*

notation (**output**)
bind-do (**infixl** ‹≫=› *54*)

notation (*ASCII* **output**)
bind-do (**infixl** ‹>>=› *54*)

nonterminal *do-binds* **and** *do-bind*
syntax
-do-block :: *do-binds* ⇒ $'a$
(‹(‹*open-block notation=‹mixfix do block››do* {//(*2* -)//})› [*12*] *62*)
-do-bind :: [*pttrn*, $'a$] ⇒ *do-bind*
(‹(‹*indent=2 notation=‹infix do bind››*- ←/ -)› *13*)
-do-let :: [*pttrn*, $'a$] ⇒ *do-bind*
(‹(‹*indent=2 notation=‹infix do let››let* - =/ -)› [*1000*, *13*] *13*)
-do-then :: $'a$ ⇒ *do-bind* (‹-› [*14*] *13*)
-do-final :: $'a$ ⇒ *do-binds* (‹-›)
-do-cons :: [*do-bind*, *do-binds*] ⇒ *do-binds*
(‹(‹*open-block notation=‹infix do next››*-;//-)› [*13*, *12*] *12*)
-thenM :: [$'a$, $'b$] ⇒ $'c$ (**infixl** ‹≫› *54*)

syntax (*ASCII*)
-do-bind :: [*pttrn*, *'a*] ⇒ *do-bind*
(‹(‹*indent=2 notation=‹infix do bind›*›- <−/ -)› *13*)
-thenM :: [*'a*, *'b*] ⇒ *'c* (**infixl** ‹>>› *54*)

syntax-consts
-do-block -do-cons -do-bind -do-then ⇌ *bind* **and**
-do-let ⇌ *Let*

translations
-do-block (*-do-cons* (*-do-then t*) (*-do-final e*))
⇌ *CONST bind-do t* (λ*-. e*)
-do-block (*-do-cons* (*-do-bind p t*) (*-do-final e*))
⇌ *CONST bind-do t* (λ*p. e*)
-do-block (*-do-cons* (*-do-let p t*) *bs*)
⇌ *let p = t in -do-block bs*
-do-block (*-do-cons b* (*-do-cons c cs*))
⇌ *-do-block* (*-do-cons b* (*-do-final* (*-do-block* (*-do-cons c cs*))))
-do-cons (*-do-let p t*) (*-do-final s*)
⇌ *-do-final* (*let p = t in s*)
-do-block (*-do-final e*) ⇀ *e*
(*m* ≫ *n*) ⇀ (*m* ≫= (λ*-. n*))

adhoc-overloading
bind ⇌ *Set.bind Predicate.bind Option.bind List.bind*

end

67 Less common functions on lists

theory *More-List*
imports *Main*
begin

definition *strip-while* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *'a list*
where
strip-while P = *rev* ∘ *dropWhile P* ∘ *rev*

lemma *strip-while-rev* [*simp*]:
strip-while P (*rev xs*) = *rev* (*dropWhile P xs*)
by (*simp add*: *strip-while-def*)

lemma *strip-while-Nil* [*simp*]:
strip-while P [] = []
by (*simp add*: *strip-while-def*)

lemma *strip-while-append* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*xs* @ [*x*]) = *xs* @ [*x*]

by (*simp add*: *strip-while-def*)

lemma *strip-while-append-rec* [*simp*]:
P x ⟹ *strip-while P* (*xs* @ [*x*]) = *strip-while P xs*
by (*simp add*: *strip-while-def*)

lemma *strip-while-Cons* [*simp*]:
¬ *P x* ⟹ *strip-while P* (*x* # *xs*) = *x* # *strip-while P xs*
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *strip-while-eq-Nil* [*simp*]:
strip-while P xs = [] ⟷ (∀ *x*∈*set xs*. *P x*)
by (*simp add*: *strip-while-def*)

lemma *strip-while-eq-Cons-rec*:
strip-while P (*x* # *xs*) = *x* # *strip-while P xs* ⟷ ¬ (*P x* ∧ (∀ *x*∈*set xs*. *P x*))
by (*induct xs rule*: *rev-induct*) (*simp-all add*: *strip-while-def*)

lemma *split-strip-while-append*:
fixes *xs* :: *'a list*
obtains *ys zs* :: *'a list*
where *strip-while P xs* = *ys* **and** ∀ *x*∈*set zs*. *P x* **and** *xs* = *ys* @ *zs*
proof (*rule that*)
show *strip-while P xs* = *strip-while P xs* **..**
show ∀ *x*∈*set* (*rev* (*takeWhile P* (*rev xs*))). *P x* **by** (*simp add*: *takeWhile-eq-all-conv* [*symmetric*])
have *rev xs* = *rev* (*strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*)))
by (*simp add*: *strip-while-def*)
then show *xs* = *strip-while P xs* @ *rev* (*takeWhile P* (*rev xs*))
by (*simp only*: *rev-is-rev-conv*)
qed

lemma *strip-while-snoc* [*simp*]:
strip-while P (*xs* @ [*x*]) = (*if P x then strip-while P xs else xs* @ [*x*])
by (*simp add*: *strip-while-def*)

lemma *strip-while-map*:
strip-while P (*map f xs*) = *map f* (*strip-while* (*P* ∘ *f*) *xs*)
by (*simp add*: *strip-while-def rev-map dropWhile-map*)

lemma *strip-while-dropWhile-commute*:
strip-while P (*dropWhile Q xs*) = *dropWhile Q* (*strip-while P xs*)
proof (*induct xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons x xs*)
show *?case*

```
  proof (cases ∀ y∈set xs. P y)
    case True
    with dropWhile-append2 [of rev xs] show ?thesis
      by (auto simp add: strip-while-def dest: set-dropWhileD)
  next
    case False
    then obtain y where y ∈ set xs and ¬ P y
      by blast
    with Cons dropWhile-append3 [of P y rev xs] show ?thesis
      by (simp add: strip-while-def)
  qed
qed

lemma dropWhile-strip-while-commute:
  dropWhile P (strip-while Q xs) = strip-while Q (dropWhile P xs)
  by (simp add: strip-while-dropWhile-commute)


definition no-leading :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  no-leading P xs ⟷ (xs ≠ [] ⟶ ¬ P (hd xs))

lemma no-leading-Nil [iff]:
  no-leading P []
  by (simp add: no-leading-def)

lemma no-leading-Cons [iff]:
  no-leading P (x # xs) ⟷ ¬ P x
  by (simp add: no-leading-def)

lemma no-leading-append [simp]:
  no-leading P (xs @ ys) ⟷ no-leading P xs ∧ (xs = [] ⟶ no-leading P ys)
  by (induct xs) simp-all

lemma no-leading-dropWhile [simp]:
  no-leading P (dropWhile P xs)
  by (induct xs) simp-all

lemma dropWhile-eq-obtain-leading:
  assumes dropWhile P xs = ys
  obtains zs where xs = zs @ ys and ⋀z. z ∈ set zs ⟹ P z and no-leading P
ys
proof −
  from assms have ∃ zs. xs = zs @ ys ∧ (∀ z ∈ set zs. P z) ∧ no-leading P ys
  proof (induct xs arbitrary: ys)
    case Nil then show ?case by simp
  next
    case (Cons x xs ys)
    show ?case proof (cases P x)
```

```
      case True with Cons.hyps [of ys] Cons.prems
      have ∃ zs. xs = zs @ ys ∧ (∀ a∈set zs. P a) ∧ no-leading P ys
        by simp
      then obtain zs where xs = zs @ ys and ⋀z. z ∈ set zs ⟹ P z
        and ∗: no-leading P ys
        by blast
      with True have x # xs = (x # zs) @ ys and ⋀z. z ∈ set (x # zs) ⟹ P z
        by auto
      with ∗ show ?thesis
        by blast    next
      case False
      with Cons show ?thesis by (cases ys) simp-all
    qed
  qed
  with that show thesis
    by blast
qed

lemma dropWhile-idem-iff:
  dropWhile P xs = xs ⟷ no-leading P xs
  by (cases xs) (auto elim: dropWhile-eq-obtain-leading)


abbreviation no-trailing :: ('a ⇒ bool) ⇒ 'a list ⇒ bool
where
  no-trailing P xs ≡ no-leading P (rev xs)

lemma no-trailing-unfold:
  no-trailing P xs ⟷ (xs ≠ [] ⟶ ¬ P (last xs))
  by (induct xs) simp-all

lemma no-trailing-Nil [iff]:
  no-trailing P []
  by simp

lemma no-trailing-Cons [simp]:
  no-trailing P (x # xs) ⟷ no-trailing P xs ∧ (xs = [] ⟶ ¬ P x)
  by simp

lemma no-trailing-append:
  no-trailing P (xs @ ys) ⟷ no-trailing P ys ∧ (ys = [] ⟶ no-trailing P xs)
  by (induct xs) simp-all

lemma no-trailing-append-Cons [simp]:
  no-trailing P (xs @ y # ys) ⟷ no-trailing P (y # ys)
  by simp

lemma no-trailing-strip-while [simp]:
  no-trailing P (strip-while P xs)
```

by (*induct xs rule*: *rev-induct*) *simp-all*

lemma *strip-while-idem* [*simp*]:
no-trailing P xs ⟹ *strip-while P xs* = *xs*
by (*cases xs rule*: *rev-cases*) *simp-all*

lemma *strip-while-eq-obtain-trailing*:
assumes *strip-while P xs* = *ys*
obtains *zs* **where** *xs* = *ys* @ *zs* **and** ⋀*z*. *z* ∈ *set zs* ⟹ *P z* **and** *no-trailing P ys*
proof −
from *assms* **have** *rev* (*rev* (*dropWhile P* (*rev xs*))) = *rev ys*
by (*simp add*: *strip-while-def*)
then have *dropWhile P* (*rev xs*) = *rev ys*
by *simp*
then obtain *zs* **where** *A*: *rev xs* = *zs* @ *rev ys* **and** *B*: ⋀*z*. *z* ∈ *set zs* ⟹ *P z*
and *C*: *no-trailing P ys*
using *dropWhile-eq-obtain-leading* **by** *blast*
from *A* **have** *rev* (*rev xs*) = *rev* (*zs* @ *rev ys*)
by *simp*
then have *xs* = *ys* @ *rev zs*
by *simp*
moreover from *B* **have** ⋀*z*. *z* ∈ *set* (*rev zs*) ⟹ *P z*
by *simp*
ultimately show *thesis* **using** *that C* **by** *blast*
qed

lemma *strip-while-idem-iff*:
strip-while P xs = *xs* ⟷ *no-trailing P xs*
proof −
define *ys* **where** *ys* = *rev xs*
moreover have *strip-while P* (*rev ys*) = *rev ys* ⟷ *no-trailing P* (*rev ys*)
by (*simp add*: *dropWhile-idem-iff*)
ultimately show *?thesis* **by** *simp*
qed

lemma *no-trailing-map*:
no-trailing P (*map f xs*) ⟷ *no-trailing* (*P* ∘ *f*) *xs*
by (*simp add*: *last-map no-trailing-unfold*)

lemma *no-trailing-drop* [*simp*]:
no-trailing P (*drop n xs*) **if** *no-trailing P xs*
proof −
from *that* **have** *no-trailing P* (*take n xs* @ *drop n xs*)
by *simp*
then show *?thesis*
by (*simp only*: *no-trailing-append*)
qed

lemma *no-trailing-upt* [*simp*]:
no-trailing P [*n*..<*m*] ⟷ (*n* < *m* ⟶ ¬ *P* (*m* − *1*))
by (*auto simp add*: *no-trailing-unfold*)

definition *nth-default* :: *'a* ⇒ *'a list* ⇒ *nat* ⇒ *'a*
where
nth-default dflt xs n = (*if n* < *length xs then xs* ! *n else dflt*)

lemma *nth-default-nth*:
n < *length xs* ⟹ *nth-default dflt xs n* = *xs* ! *n*
by (*simp add*: *nth-default-def*)

lemma *nth-default-beyond*:
length xs ≤ *n* ⟹ *nth-default dflt xs n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Nil* [*simp*]:
nth-default dflt [] *n* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-Cons*:
nth-default dflt (*x* # *xs*) *n* = (*case n of 0* ⇒ *x* | *Suc n'* ⇒ *nth-default dflt xs n'*)
by (*simp add*: *nth-default-def split*: *nat.split*)

lemma *nth-default-Cons-0* [*simp*]:
nth-default dflt (*x* # *xs*) *0* = *x*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-Cons-Suc* [*simp*]:
nth-default dflt (*x* # *xs*) (*Suc n*) = *nth-default dflt xs n*
by (*simp add*: *nth-default-Cons*)

lemma *nth-default-replicate-dflt* [*simp*]:
nth-default dflt (*replicate n dflt*) *m* = *dflt*
by (*simp add*: *nth-default-def*)

lemma *nth-default-append*:
nth-default dflt (*xs* @ *ys*) *n* =
(*if n* < *length xs then nth xs n else nth-default dflt ys* (*n* − *length xs*))
by (*auto simp add*: *nth-default-def nth-append*)

lemma *nth-default-append-trailing* [*simp*]:
nth-default dflt (*xs* @ *replicate n dflt*) = *nth-default dflt xs*
by (*simp add*: *fun-eq-iff nth-default-append*) (*simp add*: *nth-default-def*)

lemma *nth-default-snoc-default* [*simp*]:
nth-default dflt (*xs* @ [*dflt*]) = *nth-default dflt xs*
by (*auto simp add*: *nth-default-def fun-eq-iff nth-append*)

lemma *nth-default-eq-dflt-iff*:
nth-default dflt xs k = dflt ⟷ (*k < length xs* ⟶ *xs ! k = dflt*)
by (*simp add*: *nth-default-def*)

lemma *nth-default-take-eq*:
nth-default dflt (*take m xs*) *n* =
(*if n < m then nth-default dflt xs n else dflt*)
by (*simp add*: *nth-default-def*)

lemma *in-indexed-from-iff-nth-default-eq*:
x ≠ dflt ⟹ (*n*, *x*) ∈ *set* (*indexed-from 0 xs*) ⟷ *nth-default dflt xs n = x*
by (*auto simp add*: *nth-default-def in-set-conv-nth indexed-from-eq-zip*)

lemma *last-conv-nth-default*:
assumes *xs ≠* []
shows *last xs = nth-default dflt xs* (*length xs − 1*)
using *assms* **by** (*simp add*: *nth-default-def last-conv-nth*)

lemma *nth-default-map-eq*:
f dflt′ = dflt ⟹ *nth-default dflt* (*map f xs*) *n = f* (*nth-default dflt′ xs n*)
by (*simp add*: *nth-default-def*)

lemma *finite-nth-default-neq-default* [*simp*]:
finite {*k*. *nth-default dflt xs k ≠ dflt*}
by (*simp add*: *nth-default-def*)

lemma *sorted-list-of-set-nth-default*:
sorted-list-of-set {*k*. *nth-default dflt xs k ≠ dflt*} = *map fst* (*filter* (λ(-, *x*). *x ≠ dflt*) (*indexed-from 0 xs*))
by (*rule sorted-distinct-set-unique*) (*auto simp add*: *nth-default-def in-set-conv-nth sorted-filter distinct-map-filter indexed-from-eq-zip intro*: *rev-image-eqI*)

lemma *map-nth-default*:
map (*nth-default x xs*) [*0*..<*length xs*] = *xs*
proof −
have ∗: *map* (*nth-default x xs*) [*0*..<*length xs*] = *map* (*List.nth xs*) [*0*..<*length xs*]
by (*rule map-cong*) (*simp-all add*: *nth-default-nth*)
show *?thesis* **by** (*simp add*: ∗ *map-nth*)
qed

lemma *range-nth-default* [*simp*]:
range (*nth-default dflt xs*) = *insert dflt* (*set xs*)
by (*auto simp add*: *nth-default-def* [*abs-def*] *in-set-conv-nth*)

lemma *nth-strip-while*:
assumes *n < length* (*strip-while P xs*)
shows *strip-while P xs ! n = xs ! n*

proof −
have *length* (*dropWhile P* (*rev xs*)) + *length* (*takeWhile P* (*rev xs*)) = *length xs*
by (*subst add.commute*)
(*simp add*: *arg-cong* [**where** *f*=*length*, *OF takeWhile-dropWhile-id*, *unfolded length-append*])
then show *?thesis* **using** *assms*
by (*simp add*: *strip-while-def rev-nth dropWhile-nth*)
qed

lemma *length-strip-while-le*:
length (*strip-while P xs*) ≤ *length xs*
unfolding *strip-while-def o-def length-rev*
by (*subst* (*2*) *length-rev*[*symmetric*])
(*simp add*: *strip-while-def length-dropWhile-le del*: *length-rev*)

lemma *nth-default-strip-while-dflt* [*simp*]:
nth-default dflt (*strip-while* ((=) *dflt*) *xs*) = *nth-default dflt xs*
by (*induct xs rule*: *rev-induct*) *auto*

lemma *nth-default-eq-iff*:
nth-default dflt xs = *nth-default dflt ys*
⟷ *strip-while* (*HOL.eq dflt*) *xs* = *strip-while* (*HOL.eq dflt*) *ys* (**is** *?P* ⟷ *?Q*)
proof
let *?strip-while* = ‹*strip-while* (*HOL.eq dflt*)›
let *?xs* = *?strip-while xs*
let *?ys* = *?strip-while ys*
assume *?P*
then have *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
by *simp*
have *len*: *length ?xs* = *length ?ys*
proof (*rule ccontr*)
assume *neq*: ¬ *?thesis*
{ fix *xs ys* :: *'a list*
let *?xs* = *?strip-while xs*
let *?ys* = *?strip-while ys*
assume *eq*: *nth-default dflt ?xs* = *nth-default dflt ?ys*
assume *len*: *length ?xs* < *length ?ys*
then have *length ?ys* > *0* **by** *arith*
then have *?ys* ≠ [] **by** *simp*
with *last-conv-nth-default* [*of ?ys dflt*]
have *last ?ys* = *nth-default dflt ?ys* (*length ?ys* − *1*)
by *auto*
moreover from ‹*?ys* ≠ []› *no-trailing-strip-while* [*of HOL.eq dflt ys*]
have *last ?ys* ≠ *dflt* **by** (*simp add*: *no-trailing-unfold*)
ultimately have *nth-default dflt ?xs* (*length ?ys* − *1*) ≠ *dflt*
using *eq* **by** *simp*
moreover from *len* **have** *length ?ys* − *1* ≥ *length ?xs* **by** *simp*
ultimately have *False* **by** (*simp only*: *nth-default-beyond*) *simp*

```
    }
    from this [of xs ys] this [of ys xs] neq eq show False
      by (auto simp only: linorder-class.neq-iff)
  qed
  then show ?Q
  proof (rule nth-equalityI [rule-format])
    fix n
    assume n: n < length ?xs
    with len have n < length ?ys
      by simp
    with n have xs: nth-default dflt ?xs n = ?xs ! n
      and ys: nth-default dflt ?ys n = ?ys ! n
      by (simp-all only: nth-default-nth)
    with eq show ?xs ! n = ?ys ! n
      by simp
  qed
next
  assume ?Q
   then have nth-default dflt (strip-while (HOL.eq dflt) xs) = nth-default dflt
(strip-while (HOL.eq dflt) ys)
    by simp
  then show ?P
    by simp
qed

lemma nth-default-map2:
  ‹nth-default d (map2 f xs ys) n = f (nth-default d1 xs n) (nth-default d2 ys n)›
    if ‹length xs = length ys› and ‹f d1 d2 = d› for bs cs
using that proof (induction xs ys arbitrary: n rule: list-induct2)
  case Nil
  then show ?case
    by simp
next
  case (Cons x xs y ys)
  then show ?case
    by (cases n) simp-all
qed

end


theory Cancellation
imports Main
begin

named-theorems cancelation-simproc-pre ‹These theorems are here to normalise
the term. Special
  handling of constructors should be here. Remark that only the simproc @{term
NO-MATCH} is also
```

included.›

named-theorems *cancelation-simproc-post* ‹*These theorems are here to normalise the term, after the*
cancelation simproc. Normalisation of ‹*iterate-add*› *back to the normale representation*
should be put here.›

named-theorems *cancelation-simproc-eq-elim* ‹*These theorems are here to help deriving contradiction*
(*e.g.*, ‹*Suc - = 0*›).›

definition *iterate-add* :: ‹*nat* ⇒ *'a*::*cancel-comm-monoid-add* ⇒ *'a*› **where**
‹*iterate-add n a* = (((+) *a*) ^^ *n*) *0*›

lemma *iterate-add-simps*[*simp*]:
‹*iterate-add 0 a* = *0*›
‹*iterate-add* (*Suc n*) *a* = *a* + *iterate-add n a*›
unfolding *iterate-add-def* **by** *auto*

lemma *iterate-add-empty*[*simp*]: ‹*iterate-add n 0* = *0*›
unfolding *iterate-add-def* **by** (*induction n*) *auto*

lemma *iterate-add-distrib*[*simp*]: ‹*iterate-add* (*m*+*n*) *a* = *iterate-add m a* + *iterate-add n a*›
by (*induction n*) (*auto simp*: *ac-simps*)

lemma *iterate-add-Numeral1*: ‹*iterate-add n Numeral1* = *of-nat n*›
by (*induction n*) *auto*

lemma *iterate-add-1*: ‹*iterate-add n 1* = *of-nat n*›
using *iterate-add-Numeral1* **by** *auto*

lemma *iterate-add-eq-add-iff1*:
‹*i* ≤ *j* ⟹ (*iterate-add j u* + *m* = *iterate-add i u* + *n*) = (*iterate-add* (*j* − *i*) *u* + *m* = *n*)›
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-eq-add-iff2*:
‹*i* ≤ *j* ⟹ (*iterate-add i u* + *m* = *iterate-add j u* + *n*) = (*m* = *iterate-add* (*j* − *i*) *u* + *n*)›
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-less-iff1*:
j ≤ (*i*::*nat*) ⟹ (*iterate-add i* (*u*:: *'a* :: {*cancel-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*}) + *m* < *iterate-add j u* + *n*) = (*iterate-add* (*i*−*j*) *u* + *m* < *n*)
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-less-iff2*:

$i \leq (j::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\})$
$+\ m < iterate\text{-}add\ j\ u + n) = (m < iterate\text{-}add\ (j - i)\ u + n)$
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-less-eq-iff1*:
$j \leq (i::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\})$
$+\ m \leq iterate\text{-}add\ j\ u + n) = (iterate\text{-}add\ (i-j)\ u + m \leq n)$
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-less-eq-iff2*:
$i \leq (j::nat) \Longrightarrow (iterate\text{-}add\ i\ (u::'a::\{cancel\text{-}comm\text{-}monoid\text{-}add, ordered\text{-}ab\text{-}semigroup\text{-}add\text{-}imp\text{-}le\})$
$+\ m \leq iterate\text{-}add\ j\ u + n) = (m \leq iterate\text{-}add\ (j - i)\ u + n)$
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-add-eq1*:
$j \leq (i::nat) \Longrightarrow ((iterate\text{-}add\ i\ u + m) - (iterate\text{-}add\ j\ u + n)) = ((iterate\text{-}add$
$(i-j)\ u + m) - n)$
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

lemma *iterate-add-diff-add-eq2*:
$i \leq (j::nat) \Longrightarrow ((iterate\text{-}add\ i\ u + m) - (iterate\text{-}add\ j\ u + n)) = (m -$
$(iterate\text{-}add\ (j-i)\ u + n))$
by (*auto dest*!: *le-Suc-ex add-right-imp-eq simp*: *ab-semigroup-add-class.add-ac*(*1*))

Simproc Set-Up

ML-file ‹*Cancellation/cancel.ML*›
ML-file ‹*Cancellation/cancel-data.ML*›
ML-file ‹*Cancellation/cancel-simprocs.ML*›

end

68 (Finite) Multisets

theory *Multiset*
imports *Cancellation*
begin

68.1 The type of multisets

typedef *'a multiset* = ‹$\{f :: 'a \Rightarrow nat.\ finite\ \{x.\ f\ x > 0\}\}$›
morphisms *count Abs-multiset*
proof
show ‹$(\lambda x.\ 0::nat) \in \{f.\ finite\ \{x.\ f\ x > 0\}\}$›
by *simp*
qed

setup-lifting *type-definition-multiset*

lemma *count-Abs-multiset*:

‹*count* (*Abs-multiset f*) = *f*› **if** ‹*finite* {*x*. *f x* > *0*}›
by (*rule Abs-multiset-inverse*) (*simp add*: *that*)

lemma *multiset-eq-iff*: *M* = *N* ⟷ (∀ *a*. *count M a* = *count N a*)
by (*simp only*: *count-inject* [*symmetric*] *fun-eq-iff*)

lemma *multiset-eqI*: (⋀*x*. *count A x* = *count B x*) ⟹ *A* = *B*
using *multiset-eq-iff* **by** *auto*

Preservation of the representing set *multiset*.

lemma *diff-preserves-multiset*:
‹*finite* {*x*. *0* < *M x* − *N x*}› **if** ‹*finite* {*x*. *0* < *M x*}› **for** *M N* :: ‹*'a* ⇒ *nat*›
using *that* **by** (*rule rev-finite-subset*) *auto*

lemma *filter-preserves-multiset*:
‹*finite* {*x*. *0* < (*if P x then M x else 0*)}› **if** ‹*finite* {*x*. *0* < *M x*}› **for** *M N* ::
‹*'a* ⇒ *nat*›
using *that* **by** (*rule rev-finite-subset*) *auto*

lemmas *in-multiset* = *diff-preserves-multiset filter-preserves-multiset*

68.2 Representing multisets

Multiset enumeration

instantiation *multiset* :: (*type*) *cancel-comm-monoid-add*
begin

lift-definition *zero-multiset* :: ‹*'a multiset*›
is ‹λ*a*. *0*›
by *simp*

abbreviation *empty-mset* :: ‹*'a multiset*› (‹{#}›)
where ‹*empty-mset* ≡ *0*›

lift-definition *plus-multiset* :: ‹*'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*›
is ‹λ*M N a*. *M a* + *N a*›
by *simp*

lift-definition *minus-multiset* :: ‹*'a multiset* ⇒ *'a multiset* ⇒ *'a multiset*›
is ‹λ*M N a*. *M a* − *N a*›
by (*rule diff-preserves-multiset*)

instance
by (*standard*; *transfer*) (*simp-all add*: *fun-eq-iff*)

end

context
begin

qualified definition *is-empty* :: *'a multiset* ⇒ *bool* **where**
[*code-abbrev*]: *is-empty A* ⟷ *A* = {#}

end

lemma *add-mset-in-multiset*:
‹*finite* {*x*. 0 < (*if x* = *a then Suc* (*M x*) *else M x*)}›
if ‹*finite* {*x*. 0 < *M x*}›
using *that* **by** (*simp add*: *flip*: *insert-Collect*)

lift-definition *add-mset* :: *'a* ⇒ *'a multiset* ⇒ *'a multiset* **is**
λ*a M b*. *if b* = *a then Suc* (*M b*) *else M b*
by (*rule add-mset-in-multiset*)

syntax
-multiset :: *args* ⇒ *'a multiset* (‹(‹*indent=2 notation=‹mixfix multiset enumeration*››{#-#})›)
syntax-consts
-multiset ⇌ *add-mset*
translations
{#*x*, *xs*#} == *CONST add-mset x* {#*xs*#}
{#*x*#} == *CONST add-mset x* {#}

lemma *count-empty* [*simp*]: *count* {#} *a* = *0*
by (*simp add*: *zero-multiset.rep-eq*)

lemma *count-add-mset* [*simp*]:
count (*add-mset b A*) *a* = (*if b* = *a then Suc* (*count A a*) *else count A a*)
by (*simp add*: *add-mset.rep-eq*)

lemma *count-single*: *count* {#*b*#} *a* = (*if b* = *a then 1 else 0*)
by *simp*

lemma
add-mset-not-empty [*simp*]: ‹*add-mset a A* ≠ {#}› **and**
empty-not-add-mset [*simp*]: {#} ≠ *add-mset a A*
by (*auto simp*: *multiset-eq-iff*)

lemma *add-mset-add-mset-same-iff* [*simp*]:
add-mset a A = *add-mset a B* ⟷ *A* = *B*
by (*auto simp*: *multiset-eq-iff*)

lemma *add-mset-commute*:
add-mset x (*add-mset y M*) = *add-mset y* (*add-mset x M*)
by (*auto simp*: *multiset-eq-iff*)

68.3 Basic operations

68.3.1 Conversion to set and membership

definition *set-mset* :: ‹*'a multiset* ⇒ *'a set*›
where ‹*set-mset M* = {*x*. *count M x* > *0*}›

abbreviation *member-mset* :: ‹*'a* ⇒ *'a multiset* ⇒ *bool*›
where ‹*member-mset a M* ≡ *a* ∈ *set-mset M*›

notation
member-mset (‹*'*(∈#*'*)›) **and**
member-mset (‹(‹*notation*=‹*infix* ∈#››-/ ∈# -)› [*50*, *51*] *50*)

notation (*ASCII*)
member-mset (‹*'*(:#*'*)›) **and**
member-mset (‹(‹*notation*=‹*infix* :#››-/ :# -)› [*50*, *51*] *50*)

abbreviation *not-member-mset* :: ‹*'a* ⇒ *'a multiset* ⇒ *bool*›
where ‹*not-member-mset a M* ≡ *a* ∉ *set-mset M*›

notation
not-member-mset (‹*'*(∉#*'*)›) **and**
not-member-mset (‹(‹*notation*=‹*infix* ∉#››-/ ∉# -)› [*50*, *51*] *50*)

notation (*ASCII*)
not-member-mset (‹*'*(~:#*'*)›) **and**
not-member-mset (‹(‹*notation*=‹*infix* ~:#››-/ ~:# -)› [*50*, *51*] *50*)

context
begin

qualified abbreviation *Ball* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Ball M* ≡ *Set.Ball* (*set-mset M*)

qualified abbreviation *Bex* :: *'a multiset* ⇒ (*'a* ⇒ *bool*) ⇒ *bool*
where *Bex M* ≡ *Set.Bex* (*set-mset M*)

end

syntax
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool*
(‹(‹*indent*=*3 notation*=‹*binder* ∀››∀ -∈#-./ -)› [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool*
(‹(‹*indent*=*3 notation*=‹*binder* ∃››∃ -∈#-./ -)› [*0*, *0*, *10*] *10*)
syntax (*ASCII*)
-MBall :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool*
(‹(‹*indent*=*3 notation*=‹*binder* ∀››∀ -:#-./ -)› [*0*, *0*, *10*] *10*)
-MBex :: *pttrn* ⇒ *'a set* ⇒ *bool* ⇒ *bool*
(‹(‹*indent*=*3 notation*=‹*binder* ∃››∃ -:#-./ -)› [*0*, *0*, *10*] *10*)

syntax-consts
-MBall ⇌ *Multiset.Ball* **and**
-MBex ⇌ *Multiset.Bex*
translations
∀ *x*∈#*A*. *P* ⇌ *CONST Multiset.Ball A* (λ*x*. *P*)
∃ *x*∈#*A*. *P* ⇌ *CONST Multiset.Bex A* (λ*x*. *P*)

typed-print-translation ‹
[(***const-syntax*** ‹*Multiset.Ball*›, *Syntax-Trans.preserve-binder-abs2-tr'* ***syntax-const*** ‹*-MBall*›),
(***const-syntax*** ‹*Multiset.Bex*›, *Syntax-Trans.preserve-binder-abs2-tr'* ***syntax-const*** ‹*-MBex*›)]
› — to avoid eta-contraction of body

lemma *count-eq-zero-iff*:
count M x = *0* ⟷ *x* ∉# *M*
by (*auto simp add*: *set-mset-def*)

lemma *not-in-iff*:
x ∉# *M* ⟷ *count M x* = *0*
by (*auto simp add*: *count-eq-zero-iff*)

lemma *count-greater-zero-iff* [*simp*]:
count M x > *0* ⟷ *x* ∈# *M*
by (*auto simp add*: *set-mset-def*)

lemma *count-inI*:
assumes *count M x* = *0* ⟹ *False*
shows *x* ∈# *M*
proof (*rule ccontr*)
assume *x* ∉# *M*
with *assms* **show** *False* **by** (*simp add*: *not-in-iff*)
qed

lemma *in-countE*:
assumes *x* ∈# *M*
obtains *n* **where** *count M x* = *Suc n*
proof −
from *assms* **have** *count M x* > *0* **by** *simp*
then obtain *n* **where** *count M x* = *Suc n*
using *gr0-conv-Suc* **by** *blast*
with *that* **show** *thesis* **.**
qed

lemma *count-greater-eq-Suc-zero-iff* [*simp*]:
count M x ≥ *Suc 0* ⟷ *x* ∈# *M*
by (*simp add*: *Suc-le-eq*)

lemma *count-greater-eq-one-iff* [*simp*]:
count M x ≥ *1* ⟷ *x* ∈# *M*
by *simp*

lemma *set-mset-empty* [*simp*]:
set-mset {#} = {}
by (*simp add*: *set-mset-def*)

lemma *set-mset-single*:
set-mset {#*b*#} = {*b*}
by (*simp add*: *set-mset-def*)

lemma *set-mset-eq-empty-iff* [*simp*]:
set-mset M = {} ⟷ *M* = {#}
by (*auto simp add*: *multiset-eq-iff count-eq-zero-iff*)

lemma *finite-set-mset* [*iff*]:
finite (*set-mset M*)
using *count* [*of M*] **by** *simp*

lemma *set-mset-add-mset-insert* [*simp*]: ‹*set-mset* (*add-mset a A*) = *insert a* (*set-mset A*)›
by (*auto simp flip*: *count-greater-eq-Suc-zero-iff split*: *if-splits*)

lemma *multiset-nonemptyE* [*elim*]:
assumes *A* ≠ {#}
obtains *x* **where** *x* ∈# *A*
proof −
have ∃*x*. *x* ∈# *A* **by** (*rule ccontr*) (*insert assms*, *auto*)
with *that* **show** *?thesis* **by** *blast*
qed

lemma *count-gt-imp-in-mset*: *count M x* > *n* ⟹ *x* ∈# *M*
using *count-greater-zero-iff* **by** *fastforce*

68.3.2 Union

lemma *count-union* [*simp*]:
count (*M* + *N*) *a* = *count M a* + *count N a*
by (*simp add*: *plus-multiset.rep-eq*)

lemma *set-mset-union* [*simp*]:
set-mset (*M* + *N*) = *set-mset M* ∪ *set-mset N*
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-union*) *simp*

lemma *union-mset-add-mset-left* [*simp*]:
add-mset a A + *B* = *add-mset a* (*A* + *B*)
by (*auto simp*: *multiset-eq-iff*)

lemma *union-mset-add-mset-right* [*simp*]:
A + *add-mset a B* = *add-mset a* (*A* + *B*)
by (*auto simp*: *multiset-eq-iff*)

lemma *add-single-right-eq-add-mset*: ‹*A* + {#*a*#} = *add-mset a A*›
by *simp*

lemma *add-single-left-eq-add-mset*: ‹{#*a*#} + *A* = *add-mset a A*›
by *simp*

68.3.3 Difference

instance *multiset* :: (*type*) *comm-monoid-diff*
by *standard* (*transfer*; *simp add*: *fun-eq-iff*)

lemma *count-diff* [*simp*]:
count (*M* − *N*) *a* = *count M a* − *count N a*
by (*simp add*: *minus-multiset.rep-eq*)

lemma *add-mset-diff-bothsides*:
‹*add-mset a M* − *add-mset a A* = *M* − *A*›
by (*auto simp*: *multiset-eq-iff*)

lemma *in-diff-count*:
a ∈# *M* − *N* ⟷ *count N a* < *count M a*
by (*simp add*: *set-mset-def*)

lemma *count-in-diffI*:
assumes ⋀*n*. *count N x* = *n* + *count M x* ⟹ *False*
shows *x* ∈# *M* − *N*
proof (*rule ccontr*)
assume *x* ∉# *M* − *N*
then have *count N x* = (*count N x* − *count M x*) + *count M x*
by (*simp add*: *in-diff-count not-less*)
with *assms* **show** *False* **by** *auto*
qed

lemma *in-diff-countE*:
assumes *x* ∈# *M* − *N*
obtains *n* **where** *count M x* = *Suc n* + *count N x*
proof −
from *assms* **have** *count M x* − *count N x* > *0* **by** (*simp add*: *in-diff-count*)
then have *count M x* > *count N x* **by** *simp*
then obtain *n* **where** *count M x* = *Suc n* + *count N x*
using *less-iff-Suc-add* **by** *auto*
with *that* **show** *thesis* **.**
qed

lemma *in-diffD*:
assumes *a* ∈# *M* − *N*
shows *a* ∈# *M*

proof −
have *0* ≤ *count N a* **by** *simp*
also from *assms* **have** *count N a* < *count M a*
by (*simp add*: *in-diff-count*)
finally show *?thesis* **by** *simp*
qed

lemma *set-mset-diff*:
set-mset (*M* − *N*) = {*a*. *count N a* < *count M a*}
by (*simp add*: *set-mset-def*)

lemma *diff-empty* [*simp*]: *M* − {#} = *M* ∧ {#} − *M* = {#}
by *rule* (*fact Groups.diff-zero*, *fact Groups.zero-diff*)

lemma *diff-cancel*: *A* − *A* = {#}
by (*fact Groups.diff-cancel*)

lemma *diff-union-cancelR*: *M* + *N* − *N* = (*M*::'*a multiset*)
by (*fact add-diff-cancel-right'*)

lemma *diff-union-cancelL*: *N* + *M* − *N* = (*M*::'*a multiset*)
by (*fact add-diff-cancel-left'*)

lemma *diff-right-commute*:
fixes *M N Q* :: '*a multiset*
shows *M* − *N* − *Q* = *M* − *Q* − *N*
by (*fact diff-right-commute*)

lemma *diff-add*:
fixes *M N Q* :: '*a multiset*
shows *M* − (*N* + *Q*) = *M* − *N* − *Q*
by (*rule sym*) (*fact diff-diff-add*)

lemma *insert-DiffM* [*simp*]: *x* ∈# *M* ⟹ *add-mset x* (*M* − {#*x*#}) = *M*
by (*clarsimp simp*: *multiset-eq-iff*)

lemma *insert-DiffM2*: *x* ∈# *M* ⟹ (*M* − {#*x*#}) + {#*x*#} = *M*
by *simp*

lemma *diff-union-swap*: *a* ≠ *b* ⟹ *add-mset b* (*M* − {#*a*#}) = *add-mset b M* − {#*a*#}
by (*auto simp add*: *multiset-eq-iff*)

lemma *diff-add-mset-swap* [*simp*]: *b* ∉# *A* ⟹ *add-mset b M* − *A* = *add-mset b* (*M* − *A*)
by (*auto simp add*: *multiset-eq-iff simp*: *not-in-iff*)

lemma *diff-union-swap2* [*simp*]: *y* ∈# *M* ⟹ *add-mset x M* − {#*y*#} = *add-mset x* (*M* − {#*y*#})

by (*metis add-mset-diff-bothsides diff-union-swap diff-zero insert-DiffM*)

lemma *diff-diff-add-mset* [*simp*]: $(M::'a\ multiset) - N - P = M - (N + P)$
by (*rule diff-diff-add*)

lemma *diff-union-single-conv*:
$a \in\# J \Longrightarrow I + J - \{\#a\#\} = I + (J - \{\#a\#\})$
by (*simp add*: *multiset-eq-iff Suc-le-eq*)

lemma *mset-add* [*elim?*]:
assumes $a \in\# A$
obtains B **where** $A = add\text{-}mset\ a\ B$
proof −
from *assms* **have** $A = add\text{-}mset\ a\ (A - \{\#a\#\})$
by *simp*
with *that* **show** *thesis* **.**
qed

lemma *union-iff*:
$a \in\# A + B \longleftrightarrow a \in\# A \vee a \in\# B$
by *auto*

lemma *count-minus-inter-lt-count-minus-inter-iff*:
$count\ (M2 - M1)\ y < count\ (M1 - M2)\ y \longleftrightarrow y \in\# M1 - M2$
by (*meson count-greater-zero-iff gr-implies-not-zero in-diff-count leI order.strict-trans2 order-less-asym*)

lemma *minus-inter-eq-minus-inter-iff*:
$(M1 - M2) = (M2 - M1) \longleftrightarrow set\text{-}mset\ (M1 - M2) = set\text{-}mset\ (M2 - M1)$
by (*metis add.commute count-diff count-eq-zero-iff diff-add-zero in-diff-countE multiset-eq-iff*)

68.3.4 Min and Max

abbreviation *Min-mset* :: $'a$::*linorder multiset* ⇒ $'a$ **where**
Min-mset m ≡ *Min* (*set-mset m*)

abbreviation *Max-mset* :: $'a$::*linorder multiset* ⇒ $'a$ **where**
Max-mset m ≡ *Max* (*set-mset m*)

lemma
Min-in-mset: $M \neq \{\#\} \Longrightarrow Min\text{-}mset\ M \in\# M$ **and**
Max-in-mset: $M \neq \{\#\} \Longrightarrow Max\text{-}mset\ M \in\# M$
by *simp+*

68.3.5 Equality of multisets

lemma *single-eq-single* [*simp*]: $\{\#a\#\} = \{\#b\#\} \longleftrightarrow a = b$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-eq-empty* [*iff*]: $M + N = \{\#\} \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *empty-eq-union* [*iff*]: $\{\#\} = M + N \longleftrightarrow M = \{\#\} \wedge N = \{\#\}$
by (*auto simp add*: *multiset-eq-iff*)

lemma *multi-self-add-other-not-self* [*simp*]: $M = add\text{-}mset\ x\ M \longleftrightarrow False$
by (*auto simp add*: *multiset-eq-iff*)

lemma *add-mset-remove-trivial* [*simp*]: $‹add\text{-}mset\ x\ M - \{\#x\#\} = M›$
by (*auto simp*: *multiset-eq-iff*)

lemma *diff-single-trivial*: $\neg\ x \in\#\ M \Longrightarrow M - \{\#x\#\} = M$
by (*auto simp add*: *multiset-eq-iff not-in-iff*)

lemma *diff-single-eq-union*: $x \in\#\ M \Longrightarrow M - \{\#x\#\} = N \longleftrightarrow M = add\text{-}mset\ x\ N$
by *auto*

lemma *union-single-eq-diff*: $add\text{-}mset\ x\ M = N \Longrightarrow M = N - \{\#x\#\}$
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of* - *M*] **by** (*fact add-implies-diff*)

lemma *union-single-eq-member*: $add\text{-}mset\ x\ M = N \Longrightarrow x \in\#\ N$
by *auto*

lemma *add-mset-remove-trivial-If*:
$add\text{-}mset\ a\ (N - \{\#a\#\}) = (if\ a \in\#\ N\ then\ N\ else\ add\text{-}mset\ a\ N)$
by (*simp add*: *diff-single-trivial*)

lemma *add-mset-remove-trivial-eq*: $‹N = add\text{-}mset\ a\ (N - \{\#a\#\}) \longleftrightarrow a \in\#\ N›$
by (*auto simp*: *add-mset-remove-trivial-If*)

lemma *union-is-single*:
$M + N = \{\#a\#\} \longleftrightarrow M = \{\#a\#\} \wedge N = \{\#\} \vee M = \{\#\} \wedge N = \{\#a\#\}$
(**is** *?lhs* = *?rhs*)
proof
show *?rhs* $\Longrightarrow$ *?lhs* **by** *auto*
show *?lhs* $\Longrightarrow$ *?rhs*
by (*metis Multiset.diff-cancel add.commute*[*of* - *M*] *add-diff-cancel-left' diff-add-zero*
diff-single-trivial[*of a*] *insert-DiffM*[*of a*])
qed

lemma *single-is-union*: $\{\#a\#\} = M + N \longleftrightarrow \{\#a\#\} = M \wedge N = \{\#\} \vee M = \{\#\} \wedge \{\#a\#\} = N$
by (*auto simp add*: *eq-commute* [*of* $\{\#a\#\}\ M + N$] *union-is-single*)

lemma *add-eq-conv-diff*:
$add\text{-}mset\ a\ M = add\text{-}mset\ b\ N \longleftrightarrow M = N \wedge a = b \vee M = add\text{-}mset\ b\ (N -$

$\{\#a\#\}) \wedge N = \mathit{add\text{-}mset}\ a\ (M - \{\#b\#\})$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)

proof
show *?lhs* **if** *?rhs*
using *that*
by (*auto simp add*: *add-mset-commute*[*of a b*])
show *?rhs* **if** *?lhs*
proof (*cases* $a = b$)
case *True* **with** ‹*?lhs*› **show** *?thesis* **by** *simp*
next
case *False*
from ‹*?lhs*› **have** $a \in\# \mathit{add\text{-}mset}\ b\ N$ **by** (*rule union-single-eq-member*)
with *False* **have** $a \in\# N$ **by** *auto*
moreover from ‹*?lhs*› **have** $M = \mathit{add\text{-}mset}\ b\ N - \{\#a\#\}$ **by** (*rule union-single-eq-diff*)
moreover note *False*
ultimately show *?thesis* **by** (*auto simp add*: *diff-right-commute* [*of* - $\{\#a\#\}$])
qed
qed

lemma *add-mset-eq-single* [*iff*]: $\mathit{add\text{-}mset}\ b\ M = \{\#a\#\} \longleftrightarrow b = a \wedge M = \{\#\}$
by (*auto simp*: *add-eq-conv-diff*)

lemma *single-eq-add-mset* [*iff*]: $\{\#a\#\} = \mathit{add\text{-}mset}\ b\ M \longleftrightarrow b = a \wedge M = \{\#\}$
by (*auto simp*: *add-eq-conv-diff*)

lemma *insert-noteq-member*:
assumes *BC*: $\mathit{add\text{-}mset}\ b\ B = \mathit{add\text{-}mset}\ c\ C$
and *bnotc*: $b \neq c$
shows $c \in\# B$
proof −
have $c \in\# \mathit{add\text{-}mset}\ c\ C$ **by** *simp*
have *nc*: $\neg\ c \in\# \{\#b\#\}$ **using** *bnotc* **by** *simp*
then have $c \in\# \mathit{add\text{-}mset}\ b\ B$ **using** *BC* **by** *simp*
then show $c \in\# B$ **using** *nc* **by** *simp*
qed

lemma *add-eq-conv-ex*:
$(\mathit{add\text{-}mset}\ a\ M = \mathit{add\text{-}mset}\ b\ N) =$
$(M = N \wedge a = b \vee (\exists K.\ M = \mathit{add\text{-}mset}\ b\ K \wedge N = \mathit{add\text{-}mset}\ a\ K))$
by (*auto simp add*: *add-eq-conv-diff*)

lemma *multi-member-split*: $x \in\# M \Longrightarrow \exists A.\ M = \mathit{add\text{-}mset}\ x\ A$
by (*rule exI* [**where** $x = M - \{\#x\#\}$]) *simp*

lemma *multiset-add-sub-el-shuffle*:
assumes $c \in\# B$
and $b \neq c$
shows $\mathit{add\text{-}mset}\ b\ (B - \{\#c\#\}) = \mathit{add\text{-}mset}\ b\ B - \{\#c\#\}$

proof −
from ‹$c \in\# B$› **obtain** A **where** B: $B = add\text{-}mset\ c\ A$
by (*blast dest*: *multi-member-split*)
have $add\text{-}mset\ b\ A = add\text{-}mset\ c\ (add\text{-}mset\ b\ A) - \{\#c\#\}$ **by** *simp*
then have $add\text{-}mset\ b\ A = add\text{-}mset\ b\ (add\text{-}mset\ c\ A) - \{\#c\#\}$
by (*simp add*: ‹$b \neq c$›)
then show *?thesis* **using** B **by** *simp*
qed

lemma *add-mset-eq-singleton-iff*[*iff*]:
$add\text{-}mset\ x\ M = \{\#y\#\} \longleftrightarrow M = \{\#\} \wedge x = y$
by *auto*

68.3.6 Pointwise ordering induced by count

definition $subseteq\text{-}mset :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** ‹$\subseteq\#$› *50*)
where $A \subseteq\# B \longleftrightarrow (\forall a.\ count\ A\ a \leq count\ B\ a)$

definition $subset\text{-}mset :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** ‹$\subset\#$› *50*)
where $A \subset\# B \longleftrightarrow A \subseteq\# B \wedge A \neq B$

abbreviation (*input*) $supseteq\text{-}mset :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** ‹$\supseteq\#$› *50*)
where $supseteq\text{-}mset\ A\ B \equiv B \subseteq\# A$

abbreviation (*input*) $supset\text{-}mset :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool$ (**infix** ‹$\supset\#$› *50*)
where $supset\text{-}mset\ A\ B \equiv B \subset\# A$

notation (*input*)
$subseteq\text{-}mset$ (**infix** ‹$\leq\#$› *50*) **and**
$supseteq\text{-}mset$ (**infix** ‹$\geq\#$› *50*)

notation (*ASCII*)
$subseteq\text{-}mset$ (**infix** ‹$<=\#$› *50*) **and**
$subset\text{-}mset$ (**infix** ‹$<\#$› *50*) **and**
$supseteq\text{-}mset$ (**infix** ‹$>=\#$› *50*) **and**
$supset\text{-}mset$ (**infix** ‹$>\#$› *50*)

global-interpretation *subset-mset*: *ordering* ‹$(\subseteq\#)$› ‹$(\subset\#)$›
by *standard* (*auto simp add*: *subset-mset-def subseteq-mset-def multiset-eq-iff intro*: *order.trans order.antisym*)

interpretation *subset-mset*: *ordered-ab-semigroup-add-imp-le* ‹$(+)$› ‹$(-)$› ‹$(\subseteq\#)$› ‹$(\subset\#)$›
by *standard* (*auto simp add*: *subset-mset-def subseteq-mset-def multiset-eq-iff intro*: *order-trans antisym*)
— FIXME: avoid junk stemming from type class interpretation

interpretation *subset-mset*: *ordered-ab-semigroup-monoid-add-imp-le* (+) *0* (−) (⊆#) (⊂#)
by *standard*
— FIXME: avoid junk stemming from type class interpretation

lemma *mset-subset-eqI*:
(⋀*a*. *count A a* ≤ *count B a*) ⟹ *A* ⊆# *B*
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eq-count*:
A ⊆# *B* ⟹ *count A a* ≤ *count B a*
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eq-exists-conv*: (*A*::'*a multiset*) ⊆# *B* ⟷ (∃ *C*. *B* = *A* + *C*)
unfolding *subseteq-mset-def*
by (*metis add-diff-cancel-left' count-diff count-union le-Suc-ex le-add-same-cancel1 multiset-eq-iff zero-le*)

interpretation *subset-mset*: *ordered-cancel-comm-monoid-diff* (+) *0* (⊆#) (⊂#) (−)
by *standard* (*simp*, *fact mset-subset-eq-exists-conv*)
— FIXME: avoid junk stemming from type class interpretation

declare *subset-mset.add-diff-assoc*[*simp*] *subset-mset.add-diff-assoc2*[*simp*]

lemma *mset-subset-eq-mono-add-right-cancel*: (*A*::'*a multiset*) + *C* ⊆# *B* + *C* ⟷ *A* ⊆# *B*
by (*fact subset-mset.add-le-cancel-right*)

lemma *mset-subset-eq-mono-add-left-cancel*: *C* + (*A*::'*a multiset*) ⊆# *C* + *B* ⟷ *A* ⊆# *B*
by (*fact subset-mset.add-le-cancel-left*)

lemma *mset-subset-eq-mono-add*: (*A*::'*a multiset*) ⊆# *B* ⟹ *C* ⊆# *D* ⟹ *A* + *C* ⊆# *B* + *D*
by (*fact subset-mset.add-mono*)

lemma *mset-subset-eq-add-left*: (*A*::'*a multiset*) ⊆# *A* + *B*
by *simp*

lemma *mset-subset-eq-add-right*: *B* ⊆# (*A*::'*a multiset*) + *B*
by *simp*

lemma *single-subset-iff* [*simp*]:
{#*a*#} ⊆# *M* ⟷ *a* ∈# *M*
by (*auto simp add*: *subseteq-mset-def Suc-le-eq*)

lemma *mset-subset-eq-single*: *a* ∈# *B* ⟹ {#*a*#} ⊆# *B*
by *simp*

lemma *mset-subset-eq-add-mset-cancel*: ‹*add-mset a A* ⊆# *add-mset a B* ⟷ *A* ⊆# *B*›
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of* - *A*] *add-single-right-eq-add-mset*[*symmetric*, *of* - *B*]
by (*rule mset-subset-eq-mono-add-right-cancel*)

lemma *multiset-diff-union-assoc*:
fixes *A B C D* :: *'a multiset*
shows *C* ⊆# *B* ⟹ *A* + *B* − *C* = *A* + (*B* − *C*)
by (*fact subset-mset.diff-add-assoc*)

lemma *mset-subset-eq-multiset-union-diff-commute*:
fixes *A B C D* :: *'a multiset*
shows *B* ⊆# *A* ⟹ *A* − *B* + *C* = *A* + *C* − *B*
by (*fact subset-mset.add-diff-assoc2*)

lemma *diff-subset-eq-self*[*simp*]:
(*M*::*'a multiset*) − *N* ⊆# *M*
by (*simp add*: *subseteq-mset-def*)

lemma *mset-subset-eqD*:
assumes *A* ⊆# *B* **and** *x* ∈# *A*
shows *x* ∈# *B*
proof −
from ‹*x* ∈# *A*› **have** *count A x* > *0* **by** *simp*
also from ‹*A* ⊆# *B*› **have** *count A x* ≤ *count B x*
by (*simp add*: *subseteq-mset-def*)
finally show *?thesis* **by** *simp*
qed

lemma *mset-subsetD*:
A ⊂# *B* ⟹ *x* ∈# *A* ⟹ *x* ∈# *B*
by (*auto intro*: *mset-subset-eqD* [*of A*])

lemma *set-mset-mono*:
A ⊆# *B* ⟹ *set-mset A* ⊆ *set-mset B*
by (*metis mset-subset-eqD subsetI*)

lemma *mset-subset-eq-insertD*:
assumes *add-mset x A* ⊆# *B*
shows *x* ∈# *B* ∧ *A* ⊂# *B*
proof
show *x* ∈# *B*
using *assms* **by** (*simp add*: *mset-subset-eqD*)
have *A* ⊆# *add-mset x A*
by (*metis* (*no-types*) *add-single-right-eq-add-mset*[*symmetric*] *mset-subset-eq-add-left*)
then have *A* ⊂# *add-mset x A*
by (*meson multi-self-add-other-not-self subset-mset.le-imp-less-or-eq*)

then show $A \subset\# B$
using *assms subset-mset.strict-trans2* **by** *blast*
qed

lemma *mset-subset-insertD*:
$add\text{-}mset\ x\ A \subset\# B \Longrightarrow x \in\# B \wedge A \subset\# B$
by (*rule mset-subset-eq-insertD*) *simp*

lemma *mset-subset-of-empty*[*simp*]: $A \subset\# \{\#\} \longleftrightarrow False$
by (*simp only*: *subset-mset.not-less-zero*)

lemma *empty-subset-add-mset*[*simp*]: $\{\#\} \subset\# add\text{-}mset\ x\ M$
by (*auto intro*: *subset-mset.gr-zeroI*)

lemma *empty-le*: $\{\#\} \subseteq\# A$
by (*fact subset-mset.zero-le*)

lemma *insert-subset-eq-iff*:
$add\text{-}mset\ a\ A \subseteq\# B \longleftrightarrow a \in\# B \wedge A \subseteq\# B - \{\#a\#\}$
using *mset-subset-eq-insertD subset-mset.le-diff-conv2* **by** *fastforce*

lemma *insert-union-subset-iff*:
$add\text{-}mset\ a\ A \subset\# B \longleftrightarrow a \in\# B \wedge A \subset\# B - \{\#a\#\}$
by (*auto simp add*: *insert-subset-eq-iff subset-mset-def*)

lemma *subset-eq-diff-conv*:
$A - C \subseteq\# B \longleftrightarrow A \subseteq\# B + C$
by (*simp add*: *subseteq-mset-def le-diff-conv*)

lemma *multi-psub-of-add-self* [*simp*]: $A \subset\# add\text{-}mset\ x\ A$
by (*auto simp*: *subset-mset-def subseteq-mset-def*)

lemma *multi-psub-self*: $A \subset\# A = False$
by *simp*

lemma *mset-subset-add-mset* [*simp*]: $add\text{-}mset\ x\ N \subset\# add\text{-}mset\ x\ M \longleftrightarrow N \subset\# M$
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of* - *N*] *add-single-right-eq-add-mset*[*symmetric*, *of* - *M*]
by (*fact subset-mset.add-less-cancel-right*)

lemma *mset-subset-diff-self*: $c \in\# B \Longrightarrow B - \{\#c\#\} \subset\# B$
by (*auto simp*: *subset-mset-def elim*: *mset-add*)

lemma *Diff-eq-empty-iff-mset*: $A - B = \{\#\} \longleftrightarrow A \subseteq\# B$
by (*auto simp*: *multiset-eq-iff subseteq-mset-def*)

lemma *add-mset-subseteq-single-iff*[*iff*]: $add\text{-}mset\ a\ M \subseteq\# \{\#b\#\} \longleftrightarrow M = \{\#\} \wedge a = b$

proof
assume *A*: *add-mset a M* ⊆# {#*b*#}
then have ‹*a* = *b*›
by (*auto dest*: *mset-subset-eq-insertD*)
then show *M*={#} ∧ *a*=*b*
using *A* **by** (*simp add*: *mset-subset-eq-add-mset-cancel*)
qed *simp*

lemma *nonempty-subseteq-mset-eq-single*: *M* ≠ {#} ⟹ *M* ⊆# {#*x*#} ⟹ *M* = {#*x*#}
by (*cases M*) (*metis single-is-union subset-mset.less-eqE*)

lemma *nonempty-subseteq-mset-iff-single*: (*M* ≠ {#} ∧ *M* ⊆# {#*x*#} ∧ *P*) ⟷ *M* = {#*x*#} ∧ *P*
by (*cases M*) (*metis empty-not-add-mset nonempty-subseteq-mset-eq-single subset-mset.order-refl*)

68.3.7 Intersection and bounded union

definition *inter-mset* :: ‹'*a multiset* ⇒ '*a multiset* ⇒ '*a multiset*› (**infixl** ‹∩#› *70*)
where ‹*A* ∩# *B* = *A* − (*A* − *B*)›

lemma *count-inter-mset* [*simp*]:
‹*count* (*A* ∩# *B*) *x* = *min* (*count A x*) (*count B x*)›
by (*simp add*: *inter-mset-def*)

interpretation *subset-mset*: *semilattice-inf* ‹(∩#)› ‹(⊆#)› ‹(⊂#)›
by *standard* (*simp-all add*: *multiset-eq-iff subseteq-mset-def*)
— FIXME: avoid junk stemming from type class interpretation

definition *union-mset* :: ‹'*a multiset* ⇒ '*a multiset* ⇒ '*a multiset*› (**infixl** ‹∪#› *70*)
where ‹*A* ∪# *B* = *A* + (*B* − *A*)›

lemma *count-union-mset* [*simp*]:
‹*count* (*A* ∪# *B*) *x* = *max* (*count A x*) (*count B x*)›
by (*simp add*: *union-mset-def*)

global-interpretation *subset-mset*: *semilattice-neutr-order* ‹(∪#)› ‹{#}› ‹(⊇#)› ‹(⊃#)›
proof
show ⋀*a b*. (*b* ⊆# *a*) = (*a* = *a* ∪# *b*)
by (*simp add*: *Diff-eq-empty-iff-mset union-mset-def*)
show ⋀*a b*. (*b* ⊂# *a*) = (*a* = *a* ∪# *b* ∧ *a* ≠ *b*)
by (*metis Diff-eq-empty-iff-mset add-cancel-left-right subset-mset-def union-mset-def*)
qed (*auto simp*: *multiset-eqI union-mset-def*)

interpretation *subset-mset*: *semilattice-sup* ‹(∪#)› ‹(⊆#)› ‹(⊂#)›
proof −
  have [*simp*]: $m \leq n \Longrightarrow q \leq n \Longrightarrow m + (q - m) \leq n$ **for** *m n q* :: *nat*
    by *arith*
  show *class.semilattice-sup* (∪#) (⊆#) (⊂#)
    by *standard* (*auto simp add*: *union-mset-def subseteq-mset-def*)
qed — FIXME: avoid junk stemming from type class interpretation

interpretation *subset-mset*: *bounded-lattice-bot* (∩#) (⊆#) (⊂#)
  (∪#) {#}
  by *standard auto*
    — FIXME: avoid junk stemming from type class interpretation

68.3.8 Additional intersection facts

lemma *set-mset-inter* [*simp*]:
  set-mset (*A* ∩# *B*) = *set-mset A* ∩ *set-mset B*
  by (*simp only*: *set-mset-def*) *auto*

lemma *diff-intersect-left-idem* [*simp*]:
  M − *M* ∩# *N* = *M* − *N*
  by (*simp add*: *multiset-eq-iff min-def*)

lemma *diff-intersect-right-idem* [*simp*]:
  M − *N* ∩# *M* = *M* − *N*
  by (*simp add*: *multiset-eq-iff min-def*)

lemma *multiset-inter-single*[*simp*]: *a* ≠ *b* ⟹ {#*a*#} ∩# {#*b*#} = {#}
  by (*rule multiset-eqI*) *auto*

lemma *multiset-union-diff-commute*:
  assumes *B* ∩# *C* = {#}
  shows *A* + *B* − *C* = *A* − *C* + *B*
proof (*rule multiset-eqI*)
  fix *x*
  from *assms* **have** *min* (*count B x*) (*count C x*) = *0*
    by (*auto simp add*: *multiset-eq-iff*)
  then have *count B x* = *0* ∨ *count C x* = *0*
    unfolding *min-def* **by** (*auto split*: *if-splits*)
  then show *count* (*A* + *B* − *C*) *x* = *count* (*A* − *C* + *B*) *x*
    by *auto*
qed

lemma *disjunct-not-in*:
  A ∩# *B* = {#} ⟷ (∀ *a*. *a* ∉# *A* ∨ *a* ∉# *B*)
  by (*metis disjoint-iff set-mset-eq-empty-iff set-mset-inter*)

lemma *inter-mset-empty-distrib-right*: *A* ∩# (*B* + *C*) = {#} ⟷ *A* ∩# *B* =

$\{\#\} \wedge A \cap\# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *inter-mset-empty-distrib-left*: $(A + B) \cap\# C = \{\#\} \longleftrightarrow A \cap\# C = \{\#\} \wedge B \cap\# C = \{\#\}$
by (*meson disjunct-not-in union-iff*)

lemma *add-mset-inter-add-mset* [*simp*]:
add-mset a A $\cap\#$ *add-mset a B* = *add-mset a* (*A* $\cap\#$ *B*)
by (*rule multiset-eqI*) *simp*

lemma *add-mset-disjoint* [*simp*]:
add-mset a A $\cap\#$ *B* = $\{\#\} \longleftrightarrow a \notin\# B \wedge A \cap\# B = \{\#\}$
$\{\#\}$ = *add-mset a A* $\cap\#$ *B* $\longleftrightarrow a \notin\# B \wedge \{\#\} = A \cap\# B$
by (*auto simp*: *disjunct-not-in*)

lemma *disjoint-add-mset* [*simp*]:
B $\cap\#$ *add-mset a A* = $\{\#\} \longleftrightarrow a \notin\# B \wedge B \cap\# A = \{\#\}$
$\{\#\} = A \cap\#$ *add-mset b B* $\longleftrightarrow b \notin\# A \wedge \{\#\} = A \cap\# B$
by (*auto simp*: *disjunct-not-in*)

lemma *inter-add-left1*: $\neg x \in\# N \Longrightarrow$ (*add-mset x M*) $\cap\#$ *N* = *M* $\cap\#$ *N*
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-left2*: $x \in\# N \Longrightarrow$ (*add-mset x M*) $\cap\#$ *N* = *add-mset x* (*M* $\cap\#$ (*N* − {#*x*#}))
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *inter-add-right1*: $\neg x \in\# N \Longrightarrow N \cap\#$ (*add-mset x M*) = *N* $\cap\#$ *M*
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *inter-add-right2*: $x \in\# N \Longrightarrow N \cap\#$ (*add-mset x M*) = *add-mset x* ((*N* − {#*x*#}) $\cap\#$ *M*)
by (*auto simp add*: *multiset-eq-iff elim*: *mset-add*)

lemma *disjunct-set-mset-diff*:
assumes *M* $\cap\#$ *N* = $\{\#\}$
shows *set-mset* (*M* − *N*) = *set-mset M*
proof (*rule set-eqI*)
fix *a*
from *assms* **have** $a \notin\# M \vee a \notin\# N$
by (*simp add*: *disjunct-not-in*)
then show $a \in\# M - N \longleftrightarrow a \in\# M$
by (*auto dest*: *in-diffD*) (*simp add*: *in-diff-count not-in-iff*)
qed

lemma *at-most-one-mset-mset-diff*:
assumes $a \notin\# M - \{\#a\#\}$
shows *set-mset* (*M* − {#*a*#}) = *set-mset M* − {*a*}

using *assms* **by** (*auto simp add*: *not-in-iff in-diff-count set-eq-iff*)

lemma *more-than-one-mset-mset-diff*:
assumes $a \in\# M - \{\#a\#\}$
shows *set-mset* $(M - \{\#a\#\}) =$ *set-mset* M
proof (*rule set-eqI*)
fix b
have *Suc 0* $<$ *count* $M\ b \Longrightarrow$ *count* $M\ b > 0$ **by** *arith*
then show $b \in\# M - \{\#a\#\} \longleftrightarrow b \in\# M$
using *assms* **by** (*auto simp add*: *in-diff-count*)
qed

lemma *inter-iff*:
$a \in\# A \cap\# B \longleftrightarrow a \in\# A \wedge a \in\# B$
by *simp*

lemma *inter-union-distrib-left*:
$A \cap\# B + C = (A + C) \cap\# (B + C)$
by (*simp add*: *multiset-eq-iff min-add-distrib-left*)

lemma *inter-union-distrib-right*:
$C + A \cap\# B = (C + A) \cap\# (C + B)$
using *inter-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *inter-subset-eq-union*:
$A \cap\# B \subseteq\# A + B$
by (*auto simp add*: *subseteq-mset-def*)

68.3.9 Additional bounded union facts

lemma *set-mset-sup* [*simp*]:
‹*set-mset* $(A \cup\# B) =$ *set-mset* $A \cup$ *set-mset* B›
by (*simp only*: *set-mset-def*) (*auto simp add*: *less-max-iff-disj*)

lemma *sup-union-left1* [*simp*]: $\neg\ x \in\# N \Longrightarrow$ (*add-mset* $x\ M$) $\cup\# N =$ *add-mset* $x\ (M \cup\# N)$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-left2*: $x \in\# N \Longrightarrow$ (*add-mset* $x\ M$) $\cup\# N =$ *add-mset* $x\ (M \cup\# (N - \{\#x\#\}))$
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-right1* [*simp*]: $\neg\ x \in\# N \Longrightarrow N \cup\#$ (*add-mset* $x\ M$) = *add-mset* $x\ (N \cup\# M)$
by (*simp add*: *multiset-eq-iff not-in-iff*)

lemma *sup-union-right2*: $x \in\# N \Longrightarrow N \cup\#$ (*add-mset* $x\ M$) = *add-mset* $x\ ((N - \{\#x\#\}) \cup\# M)$
by (*simp add*: *multiset-eq-iff*)

lemma *sup-union-distrib-left*:
$A \cup\# B + C = (A + C) \cup\# (B + C)$
by (*simp add*: *multiset-eq-iff max-add-distrib-left*)

lemma *union-sup-distrib-right*:
$C + A \cup\# B = (C + A) \cup\# (C + B)$
using *sup-union-distrib-left* [*of A B C*] **by** (*simp add*: *ac-simps*)

lemma *union-diff-inter-eq-sup*:
$A + B - A \cap\# B = A \cup\# B$
by (*auto simp add*: *multiset-eq-iff*)

lemma *union-diff-sup-eq-inter*:
$A + B - A \cup\# B = A \cap\# B$
by (*auto simp add*: *multiset-eq-iff*)

lemma *add-mset-union*:
‹*add-mset a A* $\cup\#$ *add-mset a B* = *add-mset a* (*A* $\cup\#$ *B*)›
by (*auto simp*: *multiset-eq-iff max-def*)

68.4 Replicate and repeat operations

definition *replicate-mset* :: *nat* ⇒ *'a* ⇒ *'a multiset* **where**
replicate-mset n x = (*add-mset x* ^^ *n*) {#}

lemma *replicate-mset-0*[*simp*]: *replicate-mset 0 x* = {#}
unfolding *replicate-mset-def* **by** *simp*

lemma *replicate-mset-Suc* [*simp*]: *replicate-mset* (*Suc n*) *x* = *add-mset x* (*replicate-mset n x*)
unfolding *replicate-mset-def* **by** (*induct n*) (*auto intro*: *add.commute*)

lemma *count-replicate-mset*[*simp*]: *count* (*replicate-mset n x*) *y* = (*if y* = *x then n else 0*)
unfolding *replicate-mset-def* **by** (*induct n*) *auto*

lift-definition *repeat-mset* :: ‹*nat* ⇒ *'a multiset* ⇒ *'a multiset*›
is ‹λ*n M a*. *n* ∗ *M a*› **by** *simp*

lemma *count-repeat-mset* [*simp*]: *count* (*repeat-mset i A*) *a* = *i* ∗ *count A a*
by *transfer rule*

lemma *repeat-mset-0* [*simp*]:
‹*repeat-mset 0 M* = {#}›
by *transfer simp*

lemma *repeat-mset-Suc* [*simp*]:
‹*repeat-mset* (*Suc n*) *M* = *M* + *repeat-mset n M*›

by *transfer simp*

lemma *repeat-mset-right* [*simp*]: *repeat-mset a* (*repeat-mset b A*) = *repeat-mset* (*a* ∗ *b*) *A*
by (*auto simp*: *multiset-eq-iff left-diff-distrib′*)

lemma *left-diff-repeat-mset-distrib′*: ‹*repeat-mset* (*i* − *j*) *u* = *repeat-mset i u* − *repeat-mset j u*›
by (*auto simp*: *multiset-eq-iff left-diff-distrib′*)

lemma *left-add-mult-distrib-mset*:
repeat-mset i u + (*repeat-mset j u* + *k*) = *repeat-mset* (*i*+*j*) *u* + *k*
by (*auto simp*: *multiset-eq-iff add-mult-distrib*)

lemma *repeat-mset-distrib*:
repeat-mset (*m* + *n*) *A* = *repeat-mset m A* + *repeat-mset n A*
by (*auto simp*: *multiset-eq-iff Nat.add-mult-distrib*)

lemma *repeat-mset-distrib2*[*simp*]:
repeat-mset n (*A* + *B*) = *repeat-mset n A* + *repeat-mset n B*
by (*auto simp*: *multiset-eq-iff add-mult-distrib2*)

lemma *repeat-mset-replicate-mset*[*simp*]:
repeat-mset n {#*a*#} = *replicate-mset n a*
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-distrib-add-mset*[*simp*]:
repeat-mset n (*add-mset a A*) = *replicate-mset n a* + *repeat-mset n A*
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-empty*[*simp*]: *repeat-mset n* {#} = {#}
by *transfer simp*

lemma *set-mset-sum*: *finite A* ⟹ *set-mset* (∑ *x*∈*A*. *f x*) = (⋃ *x*∈*A*. *set-mset* (*f x*))
by (*induction A rule*: *finite-induct*) *auto*

68.4.1 Simprocs

lemma *repeat-mset-iterate-add*: ‹*repeat-mset n M* = *iterate-add n M*›
unfolding *iterate-add-def* **by** (*induction n*) *auto*

lemma *mset-subseteq-add-iff1*:
j ≤ (*i*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*repeat-mset* (*i*−*j*) *u* + *m* ⊆# *n*)
by (*auto simp add*: *subseteq-mset-def nat-le-add-iff1*)

lemma *mset-subseteq-add-iff2*:
i ≤ (*j*::*nat*) ⟹ (*repeat-mset i u* + *m* ⊆# *repeat-mset j u* + *n*) = (*m* ⊆#

repeat-mset $(j-i)$ $u + n$)
by (*auto simp add*: *subseteq-mset-def nat-le-add-iff2*)

lemma *mset-subset-add-iff1*:
$j \leq (i::nat) \Longrightarrow$ (*repeat-mset* i $u + m \subset\#$ *repeat-mset* j $u + n$) = (*repeat-mset* $(i-j)$ $u + m \subset\#$ n)
unfolding *subset-mset-def repeat-mset-iterate-add*
by (*simp add*: *iterate-add-eq-add-iff1 mset-subseteq-add-iff1*[*unfolded repeat-mset-iterate-add*])

lemma *mset-subset-add-iff2*:
$i \leq (j::nat) \Longrightarrow$ (*repeat-mset* i $u + m \subset\#$ *repeat-mset* j $u + n$) = ($m \subset\#$ *repeat-mset* $(j-i)$ $u + n$)
unfolding *subset-mset-def repeat-mset-iterate-add*
by (*simp add*: *iterate-add-eq-add-iff2 mset-subseteq-add-iff2*[*unfolded repeat-mset-iterate-add*])

ML-file ‹*multiset-simprocs.ML*›

lemma *add-mset-replicate-mset-safe*[*cancelation-simproc-pre*]: ‹*NO-MATCH* $\{\#\}$ $M \Longrightarrow$ *add-mset* a $M = \{\#a\#\} + M$›
by *simp*

declare *repeat-mset-iterate-add*[*cancelation-simproc-pre*]

declare *iterate-add-distrib*[*cancelation-simproc-pre*]
declare *repeat-mset-iterate-add*[*symmetric*, *cancelation-simproc-post*]

declare *add-mset-not-empty*[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]
empty-not-add-mset[*cancelation-simproc-eq-elim*]
add-mset-not-empty[*cancelation-simproc-eq-elim*]
subset-mset.le-zero-eq[*cancelation-simproc-eq-elim*]
le-zero-eq[*cancelation-simproc-eq-elim*]

simproc-setup *mseteq-cancel*
(($l::'a$ *multiset*) $+ m = n \mid (l::'a$ *multiset*) $= m + n \mid$
add-mset a $m = n \mid m =$ *add-mset* a $n \mid$
replicate-mset p $a = n \mid m =$ *replicate-mset* p $a \mid$
repeat-mset p $m = n \mid m =$ *repeat-mset* p m) =
‹*K Cancel-Simprocs.eq-cancel*›

simproc-setup *msetsubset-cancel*
(($l::'a$ *multiset*) $+ m \subset\# n \mid (l::'a$ *multiset*) $\subset\# m + n \mid$
add-mset a $m \subset\# n \mid m \subset\#$ *add-mset* a $n \mid$
replicate-mset p $r \subset\# n \mid m \subset\#$ *replicate-mset* p $r \mid$
repeat-mset p $m \subset\# n \mid m \subset\#$ *repeat-mset* p m) =
‹*K Multiset-Simprocs.subset-cancel-msets*›

simproc-setup *msetsubset-eq-cancel*

((*l*::'*a multiset*) + *m* ⊆# *n* | (*l*::'*a multiset*) ⊆# *m* + *n* |
add-mset a m ⊆# *n* | *m* ⊆# *add-mset a n* |
replicate-mset p r ⊆# *n* | *m* ⊆# *replicate-mset p r* |
repeat-mset p m ⊆# *n* | *m* ⊆# *repeat-mset p m*) =
‹*K Multiset-Simprocs.subseteq-cancel-msets*›

simproc-setup *msetdiff-cancel*
(((*l*::'*a multiset*) + *m*) − *n* | (*l*::'*a multiset*) − (*m* + *n*) |
add-mset a m − *n* | *m* − *add-mset a n* |
replicate-mset p r − *n* | *m* − *replicate-mset p r* |
repeat-mset p m − *n* | *m* − *repeat-mset p m*) =
‹*K Cancel-Simprocs.diff-cancel*›

68.4.2 Conditionally complete lattice

instantiation *multiset* :: (*type*) *Inf*
begin

lift-definition *Inf-multiset* :: '*a multiset set* ⇒ '*a multiset* **is**
λA i. if A = {} *then 0 else Inf* ((*λf. f i*) ‘ *A*)
proof −
fix *A* :: ('*a* ⇒ *nat*) *set*
assume ∗: ⋀*f*. *f* ∈ *A* ⟹ *finite* {*x*. *0* < *f x*}
show ‹*finite* {*i*. *0* < (*if A* = {} *then 0 else INF f*∈*A*. *f i*)}›
proof (*cases A* = {})
case *False*
then obtain *f* **where** *f* ∈ *A* **by** *blast*
hence {*i*. *Inf* ((*λf*. *f i*) ‘ *A*) > *0*} ⊆ {*i*. *f i* > *0*}
by (*auto intro*: *less-le-trans*[*OF* - *cInf-lower*])
moreover from ‹*f* ∈ *A*› ∗ **have** *finite* ... **by** *simp*
ultimately have *finite* {*i*. *Inf* ((*λf*. *f i*) ‘ *A*) > *0*} **by** (*rule finite-subset*)
with *False* **show** *?thesis* **by** *simp*
qed *simp-all*
qed

instance ..

end

lemma *Inf-multiset-empty*: *Inf* {} = {#}
by *transfer simp-all*

lemma *count-Inf-multiset-nonempty*: *A* ≠ {} ⟹ *count* (*Inf A*) *x* = *Inf* ((*λX*. *count X x*) ‘ *A*)
by *transfer simp-all*

instantiation *multiset* :: (*type*) *Sup*
begin

definition *Sup-multiset* :: *'a multiset set* ⇒ *'a multiset* **where**
Sup-multiset A = (*if A* ≠ {} ∧ *subset-mset.bdd-above A then*
Abs-multiset (λ*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*)) *else* {#})

lemma *Sup-multiset-empty*: *Sup* {} = {#}
by (*simp add*: *Sup-multiset-def*)

lemma *Sup-multiset-unbounded*: ¬ *subset-mset.bdd-above A* ⟹ *Sup A* = {#}
by (*simp add*: *Sup-multiset-def*)

instance ..

end

lemma *bdd-above-multiset-imp-bdd-above-count*:
assumes *subset-mset.bdd-above* (*A* :: *'a multiset set*)
shows *bdd-above* ((λ*X*. *count X x*) ‘ *A*)
proof −
from *assms* **obtain** *Y* **where** *Y*: ∀ *X*∈*A*. *X* ⊆# *Y*
by (*meson subset-mset.bdd-above.E*)
hence *count X x* ≤ *count Y x* **if** *X* ∈ *A* **for** *X*
using *that* **by** (*auto intro*: *mset-subset-eq-count*)
thus *?thesis* **by** (*intro bdd-aboveI*[*of* - *count Y x*]) *auto*
qed

lemma *bdd-above-multiset-imp-finite-support*:
assumes *A* ≠ {} *subset-mset.bdd-above* (*A* :: *'a multiset set*)
shows *finite* (⋃ *X*∈*A*. {*x*. *count X x* > *0*})
proof −
from *assms* **obtain** *Y* **where** *Y*: ∀ *X*∈*A*. *X* ⊆# *Y*
by (*meson subset-mset.bdd-above.E*)
hence *count X x* ≤ *count Y x* **if** *X* ∈ *A* **for** *X x*
using *that* **by** (*auto intro*: *mset-subset-eq-count*)
hence (⋃ *X*∈*A*. {*x*. *count X x* > *0*}) ⊆ {*x*. *count Y x* > *0*}
by *safe* (*erule less-le-trans*)
moreover have *finite* ... **by** *simp*
ultimately show *?thesis* **by** (*rule finite-subset*)
qed

lemma *Sup-multiset-in-multiset*:
‹*finite* {*i*. *0* < (*SUP M*∈*A*. *count M i*)}›
if ‹*A* ≠ {}› ‹*subset-mset.bdd-above A*›
proof −
have {*i*. *Sup* ((λ*X*. *count X i*) ‘ *A*) > *0*} ⊆ (⋃ *X*∈*A*. {*i*. *0* < *count X i*})
proof *safe*
fix *i* **assume** *pos*: (*SUP X*∈*A*. *count X i*) > *0*
show *i* ∈ (⋃ *X*∈*A*. {*i*. *0* < *count X i*})
proof (*rule ccontr*)

```
      assume i ∉ (⋃X∈A. {i. 0 < count X i})
      hence ∀X∈A. count X i ≤ 0 by (auto simp: count-eq-zero-iff)
      with that have (SUP X∈A. count X i) ≤ 0
        by (intro cSup-least bdd-above-multiset-imp-bdd-above-count) auto
      with pos show False by simp
    qed
  qed
  moreover from that have finite ...
    by (rule bdd-above-multiset-imp-finite-support)
  ultimately show finite {i. Sup ((λX. count X i) ` A) > 0}
    by (rule finite-subset)
qed

lemma count-Sup-multiset-nonempty:
  ‹count (Sup A) x = (SUP X∈A. count X x)›
  if ‹A ≠ {}› ‹subset-mset.bdd-above A›
 using that by (simp add: Sup-multiset-def Sup-multiset-in-multiset count-Abs-multiset)

interpretation subset-mset: conditionally-complete-lattice Inf Sup (∩#) (⊆#) (⊂#)
(∪#)
proof
  fix X :: 'a multiset and A
  assume X ∈ A
  show Inf A ⊆# X
   by (metis ‹X ∈ A› count-Inf-multiset-nonempty empty-iff image-eqI mset-subset-eqI
wellorder-Inf-le1)
next
  fix X :: 'a multiset and A
  assume nonempty: A ≠ {} and le: ⋀Y. Y ∈ A ⟹ X ⊆# Y
  show X ⊆# Inf A
  proof (rule mset-subset-eqI)
    fix x
    from nonempty have count X x ≤ (INF X∈A. count X x)
      by (intro cInf-greatest) (auto intro: mset-subset-eq-count le)
   also from nonempty have ... = count (Inf A) x by (simp add: count-Inf-multiset-nonempty)
    finally show count X x ≤ count (Inf A) x .
  qed
next
  fix X :: 'a multiset and A
  assume X: X ∈ A and bdd: subset-mset.bdd-above A
  show X ⊆# Sup A
  proof (rule mset-subset-eqI)
    fix x
    from X have A ≠ {} by auto
    have count X x ≤ (SUP X∈A. count X x)
      by (intro cSUP-upper X bdd-above-multiset-imp-bdd-above-count bdd)
    also from count-Sup-multiset-nonempty[OF ‹A ≠ {}› bdd]
      have (SUP X∈A. count X x) = count (Sup A) x by simp
    finally show count X x ≤ count (Sup A) x .
```

qed
next
fix $X :: 'a\ multiset$ **and** A
assume *nonempty*: $A \neq \{\}$ **and** *ge*: $\bigwedge Y.\ Y \in A \Longrightarrow Y \subseteq\# X$
from *ge* **have** *bdd*: *subset-mset.bdd-above A*
by *blast*
show $Sup\ A \subseteq\# X$
proof (*rule mset-subset-eqI*)
fix x
from *count-Sup-multiset-nonempty*[*OF* ‹$A \neq \{\}$› *bdd*]
have $count\ (Sup\ A)\ x = (SUP\ X{\in}A.\ count\ X\ x)$ **.**
also from *nonempty* **have** $\ldots \leq count\ X\ x$
by (*intro cSup-least*) (*auto intro*: *mset-subset-eq-count ge*)
finally show $count\ (Sup\ A)\ x \leq count\ X\ x$ **.**
qed
qed — FIXME: avoid junk stemming from type class interpretation

lemma *set-mset-Inf*:
assumes $A \neq \{\}$
shows $set\text{-}mset\ (Inf\ A) = (\bigcap X{\in}A.\ set\text{-}mset\ X)$
proof *safe*
fix $x\ X$ **assume** $x \in\# Inf\ A\ X \in A$
hence *nonempty*: $A \neq \{\}$ **by** (*auto simp*: *Inf-multiset-empty*)
from ‹$x \in\# Inf\ A$› **have** $\{\#x\#\} \subseteq\# Inf\ A$ **by** *auto*
also from ‹$X \in A$› **have** $\ldots \subseteq\# X$ **by** (*rule subset-mset.cInf-lower*) *simp-all*
finally show $x \in\# X$ **by** *simp*
next
fix x **assume** x: $x \in (\bigcap X{\in}A.\ set\text{-}mset\ X)$
hence $\{\#x\#\} \subseteq\# X$ **if** $X \in A$ **for** X **using** *that* **by** *auto*
from *assms* **and** *this* **have** $\{\#x\#\} \subseteq\# Inf\ A$ **by** (*rule subset-mset.cInf-greatest*)
thus $x \in\# Inf\ A$ **by** *simp*
qed

lemma *in-Inf-multiset-iff*:
assumes $A \neq \{\}$
shows $x \in\# Inf\ A \longleftrightarrow (\forall X{\in}A.\ x \in\# X)$
proof −
from *assms* **have** $set\text{-}mset\ (Inf\ A) = (\bigcap X{\in}A.\ set\text{-}mset\ X)$ **by** (*rule set-mset-Inf*)
also have $x \in \ldots \longleftrightarrow (\forall X{\in}A.\ x \in\# X)$ **by** *simp*
finally show *?thesis* **.**
qed

lemma *in-Inf-multisetD*: $x \in\# Inf\ A \Longrightarrow X \in A \Longrightarrow x \in\# X$
by (*subst* (*asm*) *in-Inf-multiset-iff*) *auto*

lemma *set-mset-Sup*:
assumes *subset-mset.bdd-above A*
shows $set\text{-}mset\ (Sup\ A) = (\bigcup X{\in}A.\ set\text{-}mset\ X)$
proof *safe*

fix x **assume** $x \in\# \mathit{Sup}\ A$
hence *nonempty*: $A \neq \{\}$ **by** (*auto simp*: *Sup-multiset-empty*)
show $x \in (\bigcup X{\in}A.\ \mathit{set\text{-}mset}\ X)$
proof (*rule ccontr*)
assume *x*: $x \notin (\bigcup X{\in}A.\ \mathit{set\text{-}mset}\ X)$
have $\mathit{count}\ X\ x \leq \mathit{count}\ (\mathit{Sup}\ A)\ x$ **if** $X \in A$ **for** $X\ x$
using *that* **by** (*intro mset-subset-eq-count subset-mset.cSup-upper assms*)
with x **have** $X \subseteq\# \mathit{Sup}\ A - \{\#x\#\}$ **if** $X \in A$ **for** X
using *that* **by** (*auto simp*: *subseteq-mset-def algebra-simps not-in-iff*)
hence $\mathit{Sup}\ A \subseteq\# \mathit{Sup}\ A - \{\#x\#\}$ **by** (*intro subset-mset.cSup-least nonempty*)
with ‹$x \in\# \mathit{Sup}\ A$› **show** *False*
using *mset-subset-diff-self* **by** *fastforce*
qed
next
fix $x\ X$ **assume** $x \in \mathit{set\text{-}mset}\ X\ X \in A$
hence $\{\#x\#\} \subseteq\# X$ **by** *auto*
also have $X \subseteq\# \mathit{Sup}\ A$ **by** (*intro subset-mset.cSup-upper* ‹$X \in A$› *assms*)
finally show $x \in \mathit{set\text{-}mset}\ (\mathit{Sup}\ A)$ **by** *simp*
qed

lemma *in-Sup-multiset-iff*:
assumes *subset-mset.bdd-above A*
shows $x \in\# \mathit{Sup}\ A \longleftrightarrow (\exists X{\in}A.\ x \in\# X)$
by (*simp add*: *assms set-mset-Sup*)

lemma *in-Sup-multisetD*:
assumes $x \in\# \mathit{Sup}\ A$
shows $\exists X{\in}A.\ x \in\# X$
using *Sup-multiset-unbounded assms in-Sup-multiset-iff* **by** *fastforce*

interpretation *subset-mset*: *distrib-lattice* $(\cap\#)$ $(\subseteq\#)$ $(\subset\#)$ $(\cup\#)$
proof
fix $A\ B\ C :: 'a\ \mathit{multiset}$
show $A \cup\# (B \cap\# C) = A \cup\# B \cap\# (A \cup\# C)$
by (*intro multiset-eqI*) *simp-all*
qed — FIXME: avoid junk stemming from type class interpretation

68.4.3 Filter (with comprehension syntax)

Multiset comprehension

lift-definition *filter-mset* :: $('a \Rightarrow \mathit{bool}) \Rightarrow 'a\ \mathit{multiset} \Rightarrow 'a\ \mathit{multiset}$
is $\lambda P\ M.\ \lambda x.\ \mathit{if}\ P\ x\ \mathit{then}\ M\ x\ \mathit{else}\ 0$
by (*rule filter-preserves-multiset*)

syntax (*ASCII*)
-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset*
(‹(‹*indent=1 notation=‹mixfix multiset comprehension›*›{#- :# -./ -#})›)
syntax
-MCollect :: *pttrn* ⇒ *'a multiset* ⇒ *bool* ⇒ *'a multiset*

(‹(‹*indent=1 notation=‹mixfix multiset comprehension›*›{#- ∈# -./ -#})›)
syntax-consts
-MCollect == *filter-mset*
translations
{#*x* ∈# *M*. *P*#} == *CONST filter-mset* (λ*x*. *P*) *M*

lemma *count-filter-mset* [*simp*]:
count (*filter-mset P M*) *a* = (*if P a then count M a else 0*)
by (*simp add*: *filter-mset.rep-eq*)

lemma *set-mset-filter* [*simp*]:
set-mset (*filter-mset P M*) = {*a* ∈ *set-mset M*. *P a*}
by (*simp only*: *set-eq-iff count-greater-zero-iff* [*symmetric*] *count-filter-mset*) *simp*

lemma *filter-empty-mset* [*simp*]: *filter-mset P* {#} = {#}
by (*rule multiset-eqI*) *simp*

lemma *filter-single-mset*: *filter-mset P* {#*x*#} = (*if P x then* {#*x*#} *else* {#})
by (*rule multiset-eqI*) *simp*

lemma *filter-union-mset* [*simp*]: *filter-mset P* (*M* + *N*) = *filter-mset P M* + *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-diff-mset* [*simp*]: *filter-mset P* (*M* − *N*) = *filter-mset P M* − *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-inter-mset* [*simp*]: *filter-mset P* (*M* ∩# *N*) = *filter-mset P M* ∩# *filter-mset P N*
by (*rule multiset-eqI*) *simp*

lemma *filter-sup-mset*[*simp*]: *filter-mset P* (*A* ∪# *B*) = *filter-mset P A* ∪# *filter-mset P B*
by (*rule multiset-eqI*) *simp*

lemma *filter-mset-add-mset* [*simp*]:
filter-mset P (*add-mset x A*) =
(*if P x then add-mset x* (*filter-mset P A*) *else filter-mset P A*)
by (*auto simp*: *multiset-eq-iff*)

lemma *multiset-filter-subset*[*simp*]: *filter-mset f M* ⊆# *M*
by (*simp add*: *mset-subset-eqI*)

lemma *filter-mset-mono-strong*:
assumes *A* ⊆# *B* ⋀*x*. *x* ∈# *A* ⟹ *P x* ⟹ *Q x*
shows *filter-mset P A* ⊆# *filter-mset Q B*
by (*rule mset-subset-eqI*) (*insert assms*, *auto simp*: *mset-subset-eq-count count-eq-zero-iff*)

lemma *multiset-filter-mono*:
assumes $A \subseteq\# B$
shows *filter-mset f A* $\subseteq\#$ *filter-mset f B*
using *filter-mset-mono-strong*[*OF* ‹$A \subseteq\# B$›] **.**

lemma *filter-mset-eq-conv*:
filter-mset P M $= N \longleftrightarrow N \subseteq\# M \wedge (\forall b{\in}\#N.\ P\ b) \wedge (\forall a{\in}\#M - N.\ \neg\ P\ a)$
(**is** *?P* $\longleftrightarrow$ *?Q*)
proof
assume *?P* **then show** *?Q* **by** *auto* (*simp add*: *multiset-eq-iff in-diff-count*)
next
assume *?Q*
then obtain *Q* **where** *M*: $M = N + Q$
by (*auto simp add*: *mset-subset-eq-exists-conv*)
then have *MN*: $M - N = Q$ **by** *simp*
show *?P*
proof (*rule multiset-eqI*)
fix *a*
from ‹*?Q*› *MN* **have** $*$: $\neg\ P\ a \Longrightarrow a \notin\# N\ P\ a \Longrightarrow a \notin\# Q$
by *auto*
show *count* (*filter-mset P M*) *a* = *count N a*
proof (*cases a* $\in\# M$)
case *True*
with $*$ **show** *?thesis*
by (*simp add*: *not-in-iff M*)
next
case *False* **then have** *count M a* = *0*
by (*simp add*: *not-in-iff*)
with *M* **show** *?thesis* **by** *simp*
qed
qed
qed

lemma *filter-mset-eq-mempty-iff*[*simp*]: *filter-mset P A* $= \{\#\} \longleftrightarrow (\forall x.\ x \in\# A \longrightarrow \neg\ P\ x)$
by (*auto simp*: *multiset-eq-iff count-eq-zero-iff*)

lemma *filter-filter-mset*: *filter-mset P* (*filter-mset Q M*) $= \{\#x \in\# M.\ Q\ x \wedge P\ x\#\}$
by (*auto simp*: *multiset-eq-iff*)

lemma
filter-mset-True[*simp*]: $\{\#y \in\# M.\ True\#\} = M$ **and**
filter-mset-False[*simp*]: $\{\#y \in\# M.\ False\#\} = \{\#\}$
by (*auto simp*: *multiset-eq-iff*)

lemma *filter-mset-cong0*:
assumes $\bigwedge x.\ x \in\# M \Longrightarrow f\ x \longleftrightarrow g\ x$

shows *filter-mset f M = filter-mset g M*
proof (*rule subset-mset.antisym*; *unfold subseteq-mset-def*; *rule allI*)
fix *x*
show *count* (*filter-mset f M*) *x* ≤ *count* (*filter-mset g M*) *x*
using *assms* **by** (*cases x* ∈# *M*) (*simp-all add*: *not-in-iff*)
next
fix *x*
show *count* (*filter-mset g M*) *x* ≤ *count* (*filter-mset f M*) *x*
using *assms* **by** (*cases x* ∈# *M*) (*simp-all add*: *not-in-iff*)
qed

lemma *filter-mset-cong*:
assumes *M = M'* **and** ⋀*x*. *x* ∈# *M'* ⟹ *f x* ⟷ *g x*
shows *filter-mset f M = filter-mset g M'*
unfolding ‹*M = M'*›
using *assms* **by** (*auto intro*: *filter-mset-cong0*)

lemma *filter-eq-replicate-mset*: {#*y* ∈# *D*. *y = x*#} = *replicate-mset* (*count D x*) *x*
by (*induct D*) (*simp add*: *multiset-eqI*)

68.4.4 Size

definition *wcount* **where** *wcount f M* = (λ*x*. *count M x* ∗ *Suc* (*f x*))

lemma *wcount-union*: *wcount f* (*M* + *N*) *a* = *wcount f M a* + *wcount f N a*
by (*auto simp*: *wcount-def add-mult-distrib*)

lemma *wcount-add-mset*:
wcount f (*add-mset x M*) *a* = (*if x = a then Suc* (*f a*) *else 0*) + *wcount f M a*
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of - M*] *wcount-union* **by** (*auto simp*: *wcount-def*)

definition *size-multiset* :: ('*a* ⇒ *nat*) ⇒ '*a multiset* ⇒ *nat* **where**
size-multiset f M = sum (*wcount f M*) (*set-mset M*)

lemmas *size-multiset-eq = size-multiset-def*[*unfolded wcount-def*]

instantiation *multiset* :: (*type*) *size*
begin

definition *size-multiset* **where**
size-multiset-overloaded-def: *size-multiset = Multiset.size-multiset* (λ-. *0*)
instance ..

end

lemmas *size-multiset-overloaded-eq* =
size-multiset-overloaded-def[*THEN fun-cong*, *unfolded size-multiset-eq*, *simplified*]

lemma *size-multiset-empty* [*simp*]: *size-multiset f* {#} = *0*
by (*simp add*: *size-multiset-def*)

lemma *size-empty* [*simp*]: *size* {#} = *0*
by (*simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-single* : *size-multiset f* {#*b*#} = *Suc* (*f b*)
by (*simp add*: *size-multiset-eq*)

lemma *size-single*: *size* {#*b*#} = *1*
by (*simp add*: *size-multiset-overloaded-def size-multiset-single*)

lemma *sum-wcount-Int*:
finite A ⟹ *sum* (*wcount f N*) (*A* ∩ *set-mset N*) = *sum* (*wcount f N*) *A*
by (*induct rule*: *finite-induct*)
(*simp-all add*: *Int-insert-left wcount-def count-eq-zero-iff*)

lemma *size-multiset-union* [*simp*]:
size-multiset f (*M* + *N*::'*a multiset*) = *size-multiset f M* + *size-multiset f N*
apply (*simp add*: *size-multiset-def sum-Un-nat sum.distrib sum-wcount-Int wcount-union*)
by (*metis add-implies-diff finite-set-mset inf.commute sum-wcount-Int*)

lemma *size-multiset-add-mset* [*simp*]:
size-multiset f (*add-mset a M*) = *Suc* (*f a*) + *size-multiset f M*
by (*metis add.commute add-single-right-eq-add-mset*[*symmetric*] *size-multiset-single size-multiset-union*)

lemma *size-add-mset* [*simp*]: *size* (*add-mset a A*) = *Suc* (*size A*)
by (*simp add*: *size-multiset-overloaded-def wcount-add-mset*)

lemma *size-union* [*simp*]: *size* (*M* + *N*::'*a multiset*) = *size M* + *size N*
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *size-multiset-eq-0-iff-empty* [*iff*]:
size-multiset f M = *0* ⟷ *M* = {#}
by (*auto simp add*: *size-multiset-eq count-eq-zero-iff*)

lemma *size-eq-0-iff-empty* [*iff*]: (*size M* = *0*) = (*M* = {#})
by (*auto simp add*: *size-multiset-overloaded-def*)

lemma *nonempty-has-size*: (*S* ≠ {#}) = (*0* < *size S*)
by (*metis gr0I gr-implies-not0 size-empty size-eq-0-iff-empty*)

lemma *size-eq-Suc-imp-elem*: *size M* = *Suc n* ⟹ ∃ *a*. *a* ∈# *M*
using *all-not-in-conv* **by** *fastforce*

lemma *size-eq-Suc-imp-eq-union*:
assumes *size M* = *Suc n*

shows $\exists a\ N.\ M = add\text{-}mset\ a\ N$
by (*metis assms insert-DiffM size-eq-Suc-imp-elem*)

lemma *size-mset-mono*:
fixes $A\ B :: 'a\ multiset$
assumes $A \subseteq\# B$
shows $size\ A \leq size\ B$
proof −
from *assms*[*unfolded mset-subset-eq-exists-conv*]
obtain C **where** B: $B = A + C$ **by** *auto*
show *?thesis* **unfolding** B **by** (*induct* C) *auto*
qed

lemma *size-filter-mset-lesseq*[*simp*]: $size\ (filter\text{-}mset\ f\ M) \leq size\ M$
by (*rule size-mset-mono*[*OF multiset-filter-subset*])

lemma *size-Diff-submset*:
$M \subseteq\# M' \Longrightarrow size\ (M' - M) = size\ M' - size(M::'a\ multiset)$
by (*metis add-diff-cancel-left' size-union mset-subset-eq-exists-conv*)

lemma *size-lt-imp-ex-count-lt*: $size\ M < size\ N \Longrightarrow \exists x \in\# N.\ count\ M\ x < count\ N\ x$
by (*metis count-eq-zero-iff leD not-le-imp-less not-less-zero size-mset-mono subseteq-mset-def*)

68.5 Induction and case splits

theorem *multiset-induct* [*case-names empty add*, *induct type*: *multiset*]:
assumes *empty*: $P\ \{\#\}$
assumes *add*: $\bigwedge x\ M.\ P\ M \Longrightarrow P\ (add\text{-}mset\ x\ M)$
shows $P\ M$
proof (*induct size M arbitrary*: M)
case *0* **thus** $P\ M$ **by** (*simp add*: *empty*)
next
case (*Suc k*)
obtain $N\ x$ **where** $M = add\text{-}mset\ x\ N$
using ‹*Suc k* = *size M*› [*symmetric*]
using *size-eq-Suc-imp-eq-union* **by** *fast*
with *Suc add* **show** $P\ M$ **by** *simp*
qed

lemma *multiset-induct-min*[*case-names empty add*]:
fixes $M :: 'a::linorder\ multiset$
assumes
empty: $P\ \{\#\}$ **and**
add: $\bigwedge x\ M.\ P\ M \Longrightarrow (\forall y \in\# M.\ y \geq x) \Longrightarrow P\ (add\text{-}mset\ x\ M)$
shows $P\ M$
proof (*induct size M arbitrary*: M)
case (*Suc k*)

note *ih* = *this*(*1*) **and** *Sk-eq-sz-M* = *this*(*2*)

let *?y* = *Min-mset M*
let *?N* = *M* − {#*?y*#}

have *M*: *M* = *add-mset ?y ?N*
by (*metis Min-in Sk-eq-sz-M finite-set-mset insert-DiffM lessI not-less-zero set-mset-eq-empty-iff size-empty*)
show *?case*
by (*subst M*, *rule add*, *rule ih*, *metis M Sk-eq-sz-M nat.inject size-add-mset*, *meson Min-le finite-set-mset in-diffD*)
qed (*simp add*: *empty*)

lemma *multiset-induct-max*[*case-names empty add*]:
fixes *M* :: *'a*::*linorder multiset*
assumes
empty: *P* {#} **and**
add: ⋀*x M*. *P M* ⟹ (∀ *y* ∈# *M*. *y* ≤ *x*) ⟹ *P* (*add-mset x M*)
shows *P M*
proof (*induct size M arbitrary*: *M*)
case (*Suc k*)
note *ih* = *this*(*1*) **and** *Sk-eq-sz-M* = *this*(*2*)

let *?y* = *Max-mset M*
let *?N* = *M* − {#*?y*#}

have *M*: *M* = *add-mset ?y ?N*
by (*metis Max-in Sk-eq-sz-M finite-set-mset insert-DiffM lessI not-less-zero set-mset-eq-empty-iff size-empty*)
show *?case*
by (*subst M*, *rule add*, *rule ih*, *metis M Sk-eq-sz-M nat.inject size-add-mset*, *meson Max-ge finite-set-mset in-diffD*)
qed (*simp add*: *empty*)

lemma *multi-nonempty-split*: *M* ≠ {#} ⟹ ∃ *A a*. *M* = *add-mset a A*
by (*induct M*) *auto*

lemma *multiset-cases* [*cases type*]:
obtains (*empty*) *M* = {#} | (*add*) *x N* **where** *M* = *add-mset x N*
by (*induct M*) *simp-all*

lemma *multi-drop-mem-not-eq*: *c* ∈# *B* ⟹ *B* − {#*c*#} ≠ *B*
by (*cases B* = {#}) (*auto dest*: *multi-member-split*)

lemma *union-filter-mset-complement*[*simp*]:
∀ *x*. *P x* = (¬ *Q x*) ⟹ *filter-mset P M* + *filter-mset Q M* = *M*
by (*subst multiset-eq-iff*) *auto*

lemma *multiset-partition*: *M* = {#*x* ∈# *M*. *P x*#} + {#*x* ∈# *M*. ¬ *P x*#}

by *simp*

lemma *mset-subset-size*: $A \subset\# B \Longrightarrow$ *size* $A <$ *size* B
proof (*induct A arbitrary*: B)
case *empty*
then show *?case*
using *nonempty-has-size* **by** *auto*
next
case (*add x A*)
have *add-mset x A* $\subseteq\# B$
by (*meson add.prems subset-mset-def*)
then show *?case*
using *add.prems subset-mset.less-eqE* **by** *fastforce*
qed

lemma *size-1-singleton-mset*: *size* $M = 1 \Longrightarrow \exists\, a.\ M = \{\#a\#\}$
by (*cases M*) *auto*

lemma *set-mset-subset-singletonD*:
assumes *set-mset* $A \subseteq \{x\}$
shows $A =$ *replicate-mset* (*size A*) x
using *assms* **by** (*induction A*) *auto*

lemma *count-conv-size-mset*: *count* $A\ x =$ *size* (*filter-mset* ($\lambda y.\ y = x$) A)
by (*induction A*) *auto*

lemma *size-conv-count-bool-mset*: *size* $A =$ *count* A *True* + *count* A *False*
by (*induction A*) *auto*

68.5.1 Strong induction and subset induction for multisets

Well-foundedness of strict subset relation

lemma *wf-subset-mset-rel*: *wf* $\{(M, N :: 'a\ multiset).\ M \subset\# N\}$
using *mset-subset-size wfp-def wfp-if-convertible-to-nat* **by** *blast*

lemma *wfp-subset-mset*[*simp*]: *wfp* ($\subset\#$)
by (*rule wf-subset-mset-rel*[*to-pred*])

lemma *full-multiset-induct* [*case-names less*]:
assumes *ih*: $\bigwedge B.\ \forall (A::'a\ multiset).\ A \subset\# B \longrightarrow P\ A \Longrightarrow P\ B$
shows $P\ B$
apply (*rule wf-subset-mset-rel* [*THEN wf-induct*])
apply (*rule ih*, *auto*)
done

lemma *multi-subset-induct* [*consumes 2*, *case-names empty add*]:
assumes $F \subseteq\# A$
and *empty*: $P\ \{\#\}$
and *insert*: $\bigwedge a\ F.\ a \in\# A \Longrightarrow P\ F \Longrightarrow P$ (*add-mset a F*)

```
  shows P F
proof −
  from ‹F ⊆# A›
  show ?thesis
  proof (induct F)
    show P {#} by fact
  next
    fix x F
    assume P: F ⊆# A ⟹ P F and i: add-mset x F ⊆# A
    show P (add-mset x F)
    proof (rule insert)
      from i show x ∈# A by (auto dest: mset-subset-eq-insertD)
      from i have F ⊆# A by (auto dest: mset-subset-eq-insertD)
      with P show P F .
    qed
  qed
qed
```

68.6 Least and greatest elements

```
context begin

qualified lemma
  assumes
    M ≠ {#} and
    transp-on (set-mset M) R and
    totalp-on (set-mset M) R
  shows
    bex-least-element: (∃ l ∈# M. ∀ x ∈# M. x ≠ l ⟶ R l x) and
    bex-greatest-element: (∃ g ∈# M. ∀ x ∈# M. x ≠ g ⟶ R x g)
  using assms
  by (auto intro: Finite-Set.bex-least-element Finite-Set.bex-greatest-element)

end
```

68.7 The fold combinator

```
definition fold-mset :: ('a ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ 'a multiset ⇒ 'b
where
  fold-mset f s M = Finite-Set.fold (λx. f x ^^ count M x) s (set-mset M)

lemma fold-mset-empty [simp]: fold-mset f s {#} = s
  by (simp add: fold-mset-def)

lemma fold-mset-single [simp]: fold-mset f s {#x#} = f x s
  by (simp add: fold-mset-def)

context comp-fun-commute
begin
```

lemma *fold-mset-add-mset* [*simp*]: *fold-mset f s* (*add-mset x M*) = *f x* (*fold-mset f s M*)
proof −
interpret *mset*: *comp-fun-commute* λ*y*. *f y* ^^ *count M y*
by (*fact comp-fun-commute-funpow*)
interpret *mset-union*: *comp-fun-commute* λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*
by (*fact comp-fun-commute-funpow*)
show *?thesis*
proof (*cases x* ∈ *set-mset M*)
case *False*
then have ∗: *count* (*add-mset x M*) *x* = *1*
by (*simp add*: *not-in-iff*)
from *False* **have** *Finite-Set.fold* (λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*) *s* (*set-mset M*) =
Finite-Set.fold (λ*y*. *f y* ^^ *count M y*) *s* (*set-mset M*)
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-on-funpow*)
with *False* ∗ **show** *?thesis*
by (*simp add*: *fold-mset-def del*: *count-add-mset*)
next
case *True*
define *N* **where** *N* = *set-mset M* − {*x*}
from *N-def True* **have** ∗: *set-mset M* = *insert x N x* ∉ *N finite N* **by** *auto*
then have *Finite-Set.fold* (λ*y*. *f y* ^^ *count* (*add-mset x M*) *y*) *s N* =
Finite-Set.fold (λ*y*. *f y* ^^ *count M y*) *s N*
by (*auto intro*!: *Finite-Set.fold-cong comp-fun-commute-on-funpow*)
with ∗ **show** *?thesis* **by** (*simp add*: *fold-mset-def del*: *count-add-mset*) *simp*
qed
qed

lemma *fold-mset-fun-left-comm*: *f x* (*fold-mset f s M*) = *fold-mset f* (*f x s*) *M*
by (*induct M*) (*simp-all add*: *fun-left-comm*)

lemma *fold-mset-union* [*simp*]: *fold-mset f s* (*M* + *N*) = *fold-mset f* (*fold-mset f s M*) *N*
by (*induct M*) (*simp-all add*: *fold-mset-fun-left-comm*)

lemma *fold-mset-fusion*:
assumes *comp-fun-commute g*
and ∗: ⋀*x y*. *h* (*g x y*) = *f x* (*h y*)
shows *h* (*fold-mset g w A*) = *fold-mset f* (*h w*) *A*
proof −
interpret *comp-fun-commute g* **by** (*fact assms*)
from ∗ **show** *?thesis* **by** (*induct A*) *auto*
qed

end

lemma *union-fold-mset-add-mset*: *A* + *B* = *fold-mset add-mset A B*
proof −

interpret *comp-fun-commute add-mset*
by *standard auto*
show *?thesis*
by (*induction B*) *auto*
qed

A note on code generation: When defining some function containing a subterm *fold-mset F*, code generation is not automatic. When interpreting locale *left-commutative* with *F*, the would be code thms for *fold-mset* become thms like *fold-mset F z* $\{\#\} = z$ where *F* is not a pattern but contains defined symbols, i.e. is not a code thm. Hence a separate constant with its own code thms needs to be introduced for *F*. See the image operator below.

68.8 Image

definition *image-mset* :: $('a \Rightarrow 'b) \Rightarrow$ *'a multiset* $\Rightarrow$ *'b multiset* **where**
image-mset f = *fold-mset* (*add-mset* $\circ$ *f*) $\{\#\}$

lemma *comp-fun-commute-mset-image*: *comp-fun-commute* (*add-mset* $\circ$ *f*)
by *unfold-locales* (*simp add*: *fun-eq-iff*)

lemma *image-mset-empty* [*simp*]: *image-mset f* $\{\#\} = \{\#\}$
by (*simp add*: *image-mset-def*)

lemma *image-mset-single*: *image-mset f* $\{\#x\#\} = \{\#f\ x\#\}$
by (*simp add*: *comp-fun-commute.fold-mset-add-mset comp-fun-commute-mset-image image-mset-def*)

lemma *image-mset-union* [*simp*]: *image-mset f* $(M + N)$ = *image-mset f M* + *image-mset f N*
proof −
interpret *comp-fun-commute add-mset* $\circ$ *f*
by (*fact comp-fun-commute-mset-image*)
show *?thesis* **by** (*induct N*) (*simp-all add*: *image-mset-def*)
qed

corollary *image-mset-add-mset* [*simp*]:
image-mset f (*add-mset a M*) = *add-mset* (*f a*) (*image-mset f M*)
unfolding *image-mset-union add-single-right-eq-add-mset*[*symmetric*, *of a M*] **by** (*simp add*: *image-mset-single*)

lemma *image-mset-sum*: *image-mset f* $(\sum x{\in}A.\ g\ x)$ = $(\sum x{\in}A.$ *image-mset f* (*g x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *set-image-mset* [*simp*]: *set-mset* (*image-mset f M*) = *image f* (*set-mset M*)
by (*induct M*) *simp-all*

lemma *size-image-mset* [*simp*]: *size* (*image-mset f M*) = *size M*

by (*induct M*) *simp-all*

lemma *image-mset-is-empty-iff* [*simp*]: *image-mset f M* = {#} ⟷ *M* = {#}
by (*cases M*) *auto*

lemma *image-mset-If*:
image-mset (λ*x*. *if P x then f x else g x*) *A* =
image-mset f (*filter-mset P A*) + *image-mset g* (*filter-mset* (λ*x*. ¬*P x*) *A*)
by (*induction A*) *auto*

lemma *filter-mset-image-mset*:
filter-mset P (*image-mset f A*) = *image-mset f* (*filter-mset* (λ*x*. *P* (*f x*)) *A*)
by (*induction A*) *auto*

lemma *image-mset-Diff*:
assumes *B* ⊆# *A*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
proof −
have *image-mset f* (*A* − *B* + *B*) = *image-mset f* (*A* − *B*) + *image-mset f B*
by *simp*
also from *assms* **have** *A* − *B* + *B* = *A*
by (*simp add*: *subset-mset.diff-add*)
finally show *?thesis* **by** *simp*
qed

lemma *minus-add-mset-if-not-in-lhs*[*simp*]: *x* ∉# *A* ⟹ *A* − *add-mset x B* = *A* − *B*
by (*metis diff-intersect-left-idem inter-add-right1*)

lemma *image-mset-diff-if-inj*:
fixes *f A B*
assumes *inj f*
shows *image-mset f* (*A* − *B*) = *image-mset f A* − *image-mset f B*
proof (*induction B*)
case *empty*
show *?case*
by *simp*
next
case (*add x B*)
show *?case*
proof (*cases x* ∈# *A* − *B*)
case *True*

have *image-mset f* (*A* − *add-mset x B*) =
add-mset (*f x*) (*image-mset f* (*A* − *add-mset x B*)) − {#*f x*#}
unfolding *add-mset-remove-trivial* **..**

also have ... = *image-mset f* (*add-mset x* (*A* − *add-mset x B*)) − {#*f x*#}
unfolding *image-mset-add-mset* **..**

```
    also have ... = image-mset f (add-mset x (A − B − {#x#})) − {#f x#}
      unfolding add-single-right-eq-add-mset diff-diff-add-mset ..

    also have ... = image-mset f (A − B) − {#f x#}
      unfolding insert-DiffM[OF ‹x ∈# A − B›] ..

    also have ... = image-mset f A − image-mset f B − {#f x#}
      unfolding add.IH ..

    also have ... = image-mset f A − image-mset f (add-mset x B)
     unfolding diff-diff-add-mset add-single-right-eq-add-mset image-mset-add-mset
..

    finally show ?thesis .
  next
    case False

    hence image-mset f (A − add-mset x B) = image-mset f (A − B)
      using diff-single-trivial by fastforce

    also have ... = image-mset f A − image-mset f B − {#f x#}
    proof −
      have f x ∉ f ‘ set-mset (A − B)
        using False[folded inj-image-mem-iff[OF ‹inj f›]] .

      hence f x ∉# image-mset f (A − B)
        unfolding set-image-mset .

      thus ?thesis
        unfolding add.IH[symmetric]
        by (metis diff-single-trivial)
    qed

    also have ... = image-mset f A − image-mset f (add-mset x B)
      by simp

    finally show ?thesis .
  qed
qed

lemma count-image-mset:
  ‹count (image-mset f A) x = (∑ y∈f −‘ {x} ∩ set-mset A. count A y)›
proof (induction A)
  case empty
  then show ?case by simp
next
  case (add x A)
  moreover have ∗: (if x = y then Suc n else n) = n + (if x = y then 1 else 0)
```

for *n y*
by *simp*
ultimately show *?case*
by (*auto simp*: *sum.distrib intro*!: *sum.mono-neutral-left*)
qed

lemma *count-image-mset'*:
‹*count* (*image-mset f X*) *y* = (∑ *x* | *x* ∈# *X* ∧ *y* = *f x*. *count X x*)›
by (*auto simp add*: *count-image-mset simp flip*: *singleton-conv2 simp add*: *Collect-conj-eq ac-simps*)

lemma *image-mset-subseteq-mono*: *A* ⊆# *B* ⟹ *image-mset f A* ⊆# *image-mset f B*
by (*metis image-mset-union subset-mset.le-iff-add*)

lemma *image-mset-subset-mono*: *M* ⊂# *N* ⟹ *image-mset f M* ⊂# *image-mset f N*
by (*metis* (*no-types*) *Diff-eq-empty-iff-mset image-mset-Diff image-mset-is-empty-iff image-mset-subseteq-mono subset-mset.less-le-not-le*)

syntax (*ASCII*)
-comprehension-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(‹(‹*notation*=‹*mixfix multiset comprehension*››{#-/. - :# -#})›)
syntax
-comprehension-mset :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *'a multiset*
(‹(‹*notation*=‹*mixfix multiset comprehension*››{#-/. - ∈# -#})›)
syntax-consts
-comprehension-mset ⇌ *image-mset*
translations
{#*e*. *x* ∈# *M*#} ⇌ *CONST image-mset* (λ*x*. *e*) *M*

syntax (*ASCII*)
-comprehension-mset' :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(‹(‹*notation*=‹*mixfix multiset comprehension*››{#-/ | - :# -./ -#})›)
syntax
-comprehension-mset' :: *'a* ⇒ *'b* ⇒ *'b multiset* ⇒ *bool* ⇒ *'a multiset*
(‹(‹*notation*=‹*mixfix multiset comprehension*››{#-/ | - ∈# -./ -#})›)
syntax-consts
-comprehension-mset' ⇌ *image-mset*
translations
{#*e* | *x*∈#*M*. *P*#} ↽ {#*e*. *x* ∈# {# *x*∈#*M*. *P*#}#}

This allows to write not just filters like {#*x* ∈# *M*. *x* < *c*#} but also images like {#*x* + *x*. *x* ∈# *M*#} and {#*x*+*x*|*x*∈#*M*. *x*<*c*#}, where the latter is currently displayed as {#*x* + *x*. *x* ∈# {#*x* ∈# *M*. *x* < *c*#}#}.

lemma *in-image-mset*: *y* ∈# {#*f x*. *x* ∈# *M*#} ⟷ *y* ∈ *f* ‘ *set-mset M*
by *simp*

functor *image-mset*: *image-mset*

proof −
fix *f g* **show** *image-mset f* ∘ *image-mset g* = *image-mset* (*f* ∘ *g*)
proof
fix *A*
show (*image-mset f* ∘ *image-mset g*) *A* = *image-mset* (*f* ∘ *g*) *A*
by (*induct A*) *simp-all*
qed
show *image-mset id* = *id*
proof
fix *A*
show *image-mset id A* = *id A*
by (*induct A*) *simp-all*
qed
qed

declare
image-mset.id [*simp*]
image-mset.identity [*simp*]

lemma *image-mset-id*[*simp*]: *image-mset id x* = *x*
unfolding *id-def* **by** *auto*

lemma *image-mset-cong*: (⋀*x*. *x* ∈# *M* ⟹ *f x* = *g x*) ⟹ {#*f x*. *x* ∈# *M*#} = {#*g x*. *x* ∈# *M*#}
by (*induct M*) *auto*

lemma *image-mset-cong-pair*:
(∀ *x y*. (*x*, *y*) ∈# *M* ⟶ *f x y* = *g x y*) ⟹ {#*f x y*. (*x*, *y*) ∈# *M*#} = {#*g x y*. (*x*, *y*) ∈# *M*#}
by (*metis image-mset-cong split-cong*)

lemma *image-mset-const-eq*:
{#*c*. *a* ∈# *M*#} = *replicate-mset* (*size M*) *c*
by (*induct M*) *simp-all*

lemma *image-mset-filter-mset-swap*:
image-mset f (*filter-mset* (λ*x*. *P* (*f x*)) *M*) = *filter-mset P* (*image-mset f M*)
by (*induction M rule*: *multiset-induct*) *simp-all*

lemma *image-mset-eq-plusD*:
image-mset f A = *B* + *C* ⟹ ∃ *B*′ *C*′. *A* = *B*′ + *C*′ ∧ *B* = *image-mset f B*′ ∧ *C* = *image-mset f C*′
proof (*induction A arbitrary*: *B C*)
case *empty*
thus *?case* **by** *simp*
next
case (*add x A*)
show *?case*
proof (*cases f x* ∈# *B*)

case *True*
with *add.prems* **have** *image-mset f A = (B − {#f x#}) + C*
by (*metis add-mset-remove-trivial image-mset-add-mset mset-subset-eq-single subset-mset.add-diff-assoc2*)
thus *?thesis*
using *add.IH*[*of B − {#f x#} C*] *add.prems* **by** *force*
next
case *False*
with *add.prems* **have** *image-mset f A = B + (C − {#f x#})*
by (*metis diff-single-eq-union diff-union-single-conv image-mset-add-mset union-iff union-single-eq-member*)
then show *?thesis*
using *add.IH*[*of B C − {#f x#}*] *add.prems* **by** *force*
qed
qed

lemma *image-mset-eq-image-mset-plusD*:
assumes *image-mset f A = image-mset f B + C* **and** *inj-f*: *inj-on f* (*set-mset A ∪ set-mset B*)
shows ∃ *C′*. *A = B + C′ ∧ C = image-mset f C′*
using *assms*
proof (*induction A arbitrary*: *B C*)
case *empty*
thus *?case* **by** *simp*
next
case (*add x A*)
show *?case*
proof (*cases x ∈# B*)
case *True*
have ∃ *M3′*. *A = B − {#x#} + M3′ ∧ C = image-mset f M3′*
proof (*rule add.IH*)
show *image-mset f A = image-mset f* (*B − {#x#}*) *+ C*
using *True add.prems*(*1*)
by (*smt* (*verit*) *add-mset-add-mset-same-iff image-mset-add-mset insert-DiffM union-mset-add-mset-left*)
next
show *inj-on f* (*set-mset A ∪ set-mset* (*B − {#x#}*))
using ‹*inj-on f* (*set-mset* (*add-mset x A*) *∪ set-mset B*)›
by (*rule inj-on-subset*) (*use in-diffD* **in** *fastforce*)
qed
with *True* **show** *?thesis*
by *auto*
next
case *False*
with *add.prems*(*2*) **have** *f x ∉# image-mset f B*
by *auto*
with *add.prems*(*1*) **have** *image-mset f A = image-mset f B + (C − {#f x#})*
by (*metis* (*no-types*, *lifting*) *diff-union-single-conv image-eqI image-mset-Diff*

image-mset-single mset-subset-eq-single set-image-mset union-iff union-single-eq-diff union-single-eq-member)
with *add.prems*(*2*) *add.IH* **have** $\exists M3'.\ A = B + M3' \wedge C - \{\#f\ x\#\} = image\text{-}mset\ f\ M3'$
by *auto*
then show *?thesis*
by (*metis add.prems*(*1*) *add-diff-cancel-left'* *image-mset-Diff mset-subset-eq-add-left union-mset-add-mset-right*)
qed
qed

lemma *image-mset-eq-plus-image-msetD*:
$image\text{-}mset\ f\ A = B + image\text{-}mset\ f\ C \Longrightarrow inj\text{-}on\ f\ (set\text{-}mset\ A \cup set\text{-}mset\ C) \Longrightarrow \exists B'.\ A = B' + C \wedge B = image\text{-}mset\ f\ B'$
unfolding *add.commute*[*of B*] *add.commute*[*of - C*]
by (*rule image-mset-eq-image-mset-plusD*; *assumption*)

68.9 Further conversions

primrec $mset :: 'a\ list \Rightarrow 'a\ multiset$ **where**
$mset\ [] = \{\#\}$ |
$mset\ (a\ \#\ x) = add\text{-}mset\ a\ (mset\ x)$

lemma *in-multiset-in-set*:
$x \in\# mset\ xs \longleftrightarrow x \in set\ xs$
by (*induct xs*) *simp-all*

lemma *count-mset*:
$count\ (mset\ xs)\ x = count\text{-}list\ xs\ x$
by (*induct xs*) *simp-all*

lemma *mset-zero-iff*[*simp*]: $(mset\ x = \{\#\}) = (x = [])$
by (*induct x*) *auto*

lemma *mset-zero-iff-right*[*simp*]: $(\{\#\} = mset\ x) = (x = [])$
by (*induct x*) *auto*

lemma *mset-replicate* [*simp*]: $mset\ (replicate\ n\ x) = replicate\text{-}mset\ n\ x$
by (*induction n*) *auto*

lemma *count-mset-gt-0*: $x \in set\ xs \Longrightarrow count\ (mset\ xs)\ x > 0$
by (*induction xs*) *auto*

lemma *count-mset-0-iff* [*simp*]: $count\ (mset\ xs)\ x = 0 \longleftrightarrow x \notin set\ xs$
by (*induction xs*) *auto*

lemma *mset-single-iff*[*iff*]: $mset\ xs = \{\#x\#\} \longleftrightarrow xs = [x]$
by (*cases xs*) *auto*

lemma *mset-single-iff-right*[*iff*]: $\{\#x\#\} = mset\ xs \longleftrightarrow xs = [x]$
by (*cases xs*) *auto*

lemma *set-mset-mset*[*simp*]: *set-mset* (*mset xs*) = *set xs*
by (*induct xs*) *auto*

lemma *set-mset-comp-mset* [*simp*]: *set-mset* ∘ *mset* = *set*
by (*simp add*: *fun-eq-iff*)

lemma *size-mset* [*simp*]: *size* (*mset xs*) = *length xs*
by (*induct xs*) *simp-all*

lemma *mset-append* [*simp*]: *mset* (*xs* @ *ys*) = *mset xs* + *mset ys*
by (*induct xs arbitrary*: *ys*) *auto*

lemma *mset-filter*[*simp*]: *mset* (*filter P xs*) = $\{\#x \in\# mset\ xs.\ P\ x\ \#\}$
by (*induct xs*) *simp-all*

lemma *mset-rev* [*simp*]:
mset (*rev xs*) = *mset xs*
by (*induct xs*) *simp-all*

lemma *surj-mset*: *surj mset*
unfolding *surj-def*
proof (*rule allI*)
fix *M*
show ∃ *xs*. *M* = *mset xs*
by (*induction M*) (*auto intro*: *exI*[*of* - - # -])
qed

lemma *distinct-count-atmost-1*:
distinct x = (∀ *a*. *count* (*mset x*) *a* = (*if a* ∈ *set x then 1 else 0*))
proof (*induct x*)
case *Nil* **then show** *?case* **by** *simp*
next
case (*Cons x xs*) **show** *?case* (**is** *?lhs* ⟷ *?rhs*)
proof
assume *?lhs* **then show** *?rhs* **using** *Cons* **by** *simp*
next
assume *?rhs* **then have** *x* ∉ *set xs*
by (*simp split*: *if-splits*)
moreover from ‹*?rhs*› **have** (∀ *a*. *count* (*mset xs*) *a* =
(*if a* ∈ *set xs then 1 else 0*))
by (*auto split*: *if-splits simp add*: *count-eq-zero-iff*)
ultimately show *?lhs* **using** *Cons* **by** *simp*
qed
qed

lemma *mset-eq-setD*:
assumes *mset xs = mset ys*
shows *set xs = set ys*
proof −
from *assms* **have** *set-mset* (*mset xs*) = *set-mset* (*mset ys*)
by *simp*
then show *?thesis* **by** *simp*
qed

lemma *set-eq-iff-mset-eq-distinct*:
‹*distinct x* ⟹ *distinct y* ⟹ *set x = set y* ⟷ *mset x = mset y*›
by (*auto simp*: *multiset-eq-iff distinct-count-atmost-1*)

lemma *set-eq-iff-mset-remdups-eq*:
‹*set x = set y* ⟷ *mset* (*remdups x*) = *mset* (*remdups y*)›
using *set-eq-iff-mset-eq-distinct* **by** *fastforce*

lemma *mset-eq-imp-distinct-iff*:
‹*distinct xs* ⟷ *distinct ys*› **if** ‹*mset xs = mset ys*›
using *that* **by** (*auto simp add*: *distinct-count-atmost-1 dest*: *mset-eq-setD*)

lemma *nth-mem-mset*: *i* < *length ls* ⟹ (*ls* ! *i*) ∈# *mset ls*
proof (*induct ls arbitrary*: *i*)
case *Nil*
then show *?case* **by** *simp*
next
case *Cons*
then show *?case* **by** (*cases i*) *auto*
qed

lemma *mset-remove1*[*simp*]: *mset* (*remove1 a xs*) = *mset xs* − {#*a*#}
by (*induct xs*) (*auto simp add*: *multiset-eq-iff*)

lemma *mset-eq-length*:
assumes *mset xs = mset ys*
shows *length xs = length ys*
using *assms* **by** (*metis size-mset*)

lemma *mset-eq-length-filter*:
assumes *mset xs = mset ys*
shows *count-list xs z = count-list ys z*
using *assms* **by** (*metis count-mset*)

lemma *fold-multiset-equiv*:
‹*List.fold f xs = List.fold f ys*›
if *f*: ‹⋀*x y*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *f x* ∘ *f y* = *f y* ∘ *f x*›
and ‹*mset xs = mset ys*›
using *f* ‹*mset xs = mset ys*› [*symmetric*] **proof** (*induction xs arbitrary*: *ys*)
case *Nil*

then show *?case* **by** *simp*
next
case (*Cons x xs*)
then have ∗: ‹*set ys* = *set* (*x* # *xs*)›
by (*blast dest*: *mset-eq-setD*)
have ‹⋀*x y*. *x* ∈ *set ys* ⟹ *y* ∈ *set ys* ⟹ *f x* ∘ *f y* = *f y* ∘ *f x*›
by (*rule Cons.prems*(*1*)) (*simp-all add*: ∗)
moreover from ∗ **have** ‹*x* ∈ *set ys*›
by *simp*
ultimately have ‹*List.fold f ys* = *List.fold f* (*remove1 x ys*) ∘ *f x*›
by (*fact fold-remove1-split*)
moreover from *Cons.prems* **have** ‹*List.fold f xs* = *List.fold f* (*remove1 x ys*)›
by (*auto intro*: *Cons.IH*)
ultimately show *?case*
by *simp*
qed

lemma *fold-permuted-eq*:
‹*List.fold* (⊙) *xs z* = *List.fold* (⊙) *ys z*›
if ‹*mset xs* = *mset ys*›
and ‹*P z*› **and** *P*: ‹⋀*x z*. *x* ∈ *set xs* ⟹ *P z* ⟹ *P* (*x* ⊙ *z*)›
and *f*: ‹⋀*x y z*. *x* ∈ *set xs* ⟹ *y* ∈ *set xs* ⟹ *P z* ⟹ *x* ⊙ (*y* ⊙ *z*) = *y* ⊙ (*x* ⊙ *z*)›
for *f* (**infixl** ‹⊙› *70*)
using ‹*P z*› *P f* ‹*mset xs* = *mset ys*› [*symmetric*] **proof** (*induction xs arbitrary*: *ys z*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons x xs*)
then have ∗: ‹*set ys* = *set* (*x* # *xs*)›
by (*blast dest*: *mset-eq-setD*)
have ‹*P z*›
by (*fact Cons.prems*(*1*))
moreover have ‹⋀*x z*. *x* ∈ *set ys* ⟹ *P z* ⟹ *P* (*x* ⊙ *z*)›
by (*rule Cons.prems*(*2*)) (*simp-all add*: ∗)
moreover have ‹⋀*x y z*. *x* ∈ *set ys* ⟹ *y* ∈ *set ys* ⟹ *P z* ⟹ *x* ⊙ (*y* ⊙ *z*) = *y* ⊙ (*x* ⊙ *z*)›
by (*rule Cons.prems*(*3*)) (*simp-all add*: ∗)
moreover from ∗ **have** ‹*x* ∈ *set ys*›
by *simp*
ultimately have ‹*fold* (⊙) *ys z* = *fold* (⊙) (*remove1 x ys*) (*x* ⊙ *z*)›
by (*induction ys arbitrary*: *z*) *auto*
moreover from *Cons.prems* **have** ‹*fold* (⊙) *xs* (*x* ⊙ *z*) = *fold* (⊙) (*remove1 x ys*) (*x* ⊙ *z*)›
by (*auto intro*: *Cons.IH*)
ultimately show *?case*
by *simp*
qed

lemma *mset-shuffles*: *zs* ∈ *shuffles xs ys* ⟹ *mset zs* = *mset xs* + *mset ys*
by (*induction xs ys arbitrary*: *zs rule*: *shuffles.induct*) *auto*

lemma *mset-insort* [*simp*]: *mset* (*insort x xs*) = *add-mset x* (*mset xs*)
by (*induct xs*) *simp-all*

lemma *mset-map*[*simp*]: *mset* (*map f xs*) = *image-mset f* (*mset xs*)
by (*induct xs*) *simp-all*

lemma *mset-removeAll-eq*:
‹*mset* (*removeAll x xs*) = *filter-mset* ((≠) *x*) (*mset xs*)›
by (*induction xs*) *auto*

lemma *singleton-set-mset-subset*: **fixes** *X Y* :: *'a list set*
assumes ∀ *xs* ∈ *X*. *set xs* ⊆ {*a*} *mset* ‘ *X* ⊆ *mset* ‘ *Y*
shows *X* ⊆ *Y*
proof
fix *xs* **assume** *xs* ∈ *X*
obtain *ys* **where** *ys*: *ys* ∈ *Y mset xs* = *mset ys*
using ‹*xs* ∈ *X*› *assms*(*2*) **by** *auto*
then show *xs* ∈ *Y* **using** ‹*xs* ∈ *X*› *assms*(*1*) *ys*
by (*metis singleton-iff mset-eq-setD replicate-eqI set-empty subset-singletonD size-mset*)
qed

lemma *singleton-set-mset-eq*: **fixes** *X Y* :: *'a list set*
assumes ∀ *xs* ∈ *X*. *set xs* ⊆ {*a*} *mset* ‘ *X* = *mset* ‘ *Y*
shows *X* = *Y*
proof −
have ∀ *ys* ∈ *Y*. *set ys* ⊆ {*a*}
by (*metis* (*mono-tags*, *lifting*) *assms image-iff mset-eq-setD*)
thus *?thesis*
by (*metis antisym assms singleton-set-mset-subset subset-refl*)
qed

global-interpretation *mset-set*: *folding add-mset* {#}
defines *mset-set* = *folding-on.F add-mset* {#}
by *standard* (*simp add*: *fun-eq-iff*)

lemma *sum-multiset-singleton* [*simp*]: *sum* (λ*n*. {#*n*#}) *A* = *mset-set A*
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *count-mset-set* [*simp*]:
finite A ⟹ *x* ∈ *A* ⟹ *count* (*mset-set A*) *x* = *1* (**is** *PROP ?P*)
¬ *finite A* ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?Q*)
x ∉ *A* ⟹ *count* (*mset-set A*) *x* = *0* (**is** *PROP ?R*)
proof −
have ∗: *count* (*mset-set A*) *x* = *0* **if** *x* ∉ *A* **for** *A*

proof (*cases finite A*)
case *False* **then show** *?thesis* **by** *simp*
next
case *True* **from** *True* ‹$x \notin A$› **show** *?thesis* **by** (*induct A*) *auto*
qed
then show *PROP ?P PROP ?Q PROP ?R*
by (*auto elim*!: *Set.set-insert*)
qed — TODO: maybe define *mset-set* also in terms of *Abs-multiset*

lemma *elem-mset-set*[*simp*, *intro*]: *finite A* $\Longrightarrow$ $x \in\# $ *mset-set A* $\longleftrightarrow$ $x \in A$
by (*induct A rule*: *finite-induct*) *simp-all*

lemma *mset-set-Union*:
finite A $\Longrightarrow$ *finite B* $\Longrightarrow$ $A \cap B = \{\}$ $\Longrightarrow$ *mset-set* $(A \cup B)$ = *mset-set A* + *mset-set B*
by (*induction A rule*: *finite-induct*) *auto*

lemma *filter-mset-mset-set* [*simp*]:
finite A $\Longrightarrow$ *filter-mset P* (*mset-set A*) = *mset-set* $\{x \in A.\ P\ x\}$
proof (*induction A rule*: *finite-induct*)
case (*insert x A*)
from *insert.hyps* **have** *filter-mset P* (*mset-set* (*insert x A*)) =
filter-mset P (*mset-set A*) + *mset-set* (*if P x then* $\{x\}$ *else* $\{\}$)
by *simp*
also have *filter-mset P* (*mset-set A*) = *mset-set* $\{x \in A.\ P\ x\}$
by (*rule insert.IH*)
also from *insert.hyps*
have ... + *mset-set* (*if P x then* $\{x\}$ *else* $\{\}$) =
mset-set $(\{x \in A.\ P\ x\} \cup$ (*if P x then* $\{x\}$ *else* $\{\}$)) (**is** - = *mset-set ?A*)
by (*intro mset-set-Union* [*symmetric*]) *simp-all*
also from *insert.hyps* **have** *?A* = $\{y \in$*insert x A*. *P y*$\}$ **by** *auto*
finally show *?case* **.**
qed *simp-all*

lemma *mset-set-Diff*:
assumes *finite A* $B \subseteq A$
shows *mset-set* $(A - B)$ = *mset-set A* − *mset-set B*
proof −
from *assms* **have** *mset-set* $((A - B) \cup B)$ = *mset-set* $(A - B)$ + *mset-set B*
by (*intro mset-set-Union*) (*auto dest*: *finite-subset*)
also from *assms* **have** $A - B \cup B = A$ **by** *blast*
finally show *?thesis* **by** *simp*
qed

lemma *mset-minus-list-mset*[*simp*]: *mset*(*minus-list-mset xs ys*) = *mset xs* − *mset ys*
by (*simp add*: *count-mset multiset-eq-iff*)

lemma *mset-set-set*: *distinct xs* $\Longrightarrow$ *mset-set* (*set xs*) = *mset xs*

by (*induction xs*) *simp-all*

lemma *count-mset-set'*: *count* (*mset-set A*) *x* = (*if finite A* ∧ *x* ∈ *A then 1 else 0*)
by *auto*

lemma *subset-imp-msubset-mset-set*:
assumes *A* ⊆ *B finite B*
shows *mset-set A* ⊆# *mset-set B*
proof (*rule mset-subset-eqI*)
fix *x* :: *'a*
from *assms* **have** *finite A* **by** (*rule finite-subset*)
with *assms* **show** *count* (*mset-set A*) *x* ≤ *count* (*mset-set B*) *x*
by (*cases x* ∈ *A*; *cases x* ∈ *B*) *auto*
qed

lemma *mset-set-set-mset-msubset*: *mset-set* (*set-mset A*) ⊆# *A*
proof (*rule mset-subset-eqI*)
fix *x* **show** *count* (*mset-set* (*set-mset A*)) *x* ≤ *count A x*
by (*cases x* ∈# *A*) *simp-all*
qed

lemma *mset-set-upto-eq-mset-upto*:
‹*mset-set* {..<*n*} = *mset* [*0*..<*n*]›
by (*induction n*) (*auto simp*: *ac-simps lessThan-Suc*)

context *linorder*
begin

definition *sorted-list-of-multiset* :: *'a multiset* ⇒ *'a list*
where
sorted-list-of-multiset M = *fold-mset insort* [] *M*

lemma *sorted-list-of-multiset-empty* [*simp*]:
sorted-list-of-multiset {#} = []
by (*simp add*: *sorted-list-of-multiset-def*)

lemma *sorted-list-of-multiset-singleton* [*simp*]:
sorted-list-of-multiset {#*x*#} = [*x*]
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)
qed

lemma *sorted-list-of-multiset-insert* [*simp*]:
sorted-list-of-multiset (*add-mset x M*) = *List.insort x* (*sorted-list-of-multiset M*)
proof −
interpret *comp-fun-commute insort* **by** (*fact comp-fun-commute-insort*)
show *?thesis* **by** (*simp add*: *sorted-list-of-multiset-def*)

qed

end

lemma *mset-sorted-list-of-multiset*[*simp*]: *mset* (*sorted-list-of-multiset M*) = *M*
by (*induct M*) *simp-all*

lemma *sorted-list-of-multiset-mset*[*simp*]: *sorted-list-of-multiset* (*mset xs*) = *sort xs*
by (*induct xs*) *simp-all*

lemma *finite-set-mset-mset-set*[*simp*]: *finite A* ⟹ *set-mset* (*mset-set A*) = *A*
by *auto*

lemma *mset-set-empty-iff*: *mset-set A* = {#} ⟷ *A* = {} ∨ *infinite A*
using *finite-set-mset-mset-set* **by** *fastforce*

lemma *infinite-set-mset-mset-set*: ¬ *finite A* ⟹ *set-mset* (*mset-set A*) = {}
by *simp*

lemma *set-sorted-list-of-multiset* [*simp*]:
set (*sorted-list-of-multiset M*) = *set-mset M*
by (*induct M*) (*simp-all add*: *set-insort-key*)

lemma *sorted-sorted-list-of-multiset* [*iff*]:
‹*sorted* (*sorted-list-of-multiset M*)›
by (*induction M*) (*simp-all add*: *sorted-insort*)

lemma *sorted-list-of-mset-set* [*simp*]:
sorted-list-of-multiset (*mset-set A*) = *sorted-list-of-set A*
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *mset-upt* [*simp*]: *mset* [*m*..<*n*] = *mset-set* {*m*..<*n*}
by (*metis distinct-upt mset-set-set set-upt*)

lemma *image-mset-map-of*:
distinct (*map fst xs*) ⟹ {#*the* (*map-of xs i*). *i* ∈# *mset* (*map fst xs*)#} = *mset* (*map snd xs*)
proof (*induction xs*)
case (*Cons x xs*)
have {#*the* (*map-of* (*x* # *xs*) *i*). *i* ∈# *mset* (*map fst* (*x* # *xs*))#} =
add-mset (*snd x*) {#*the* (*if i* = *fst x then Some* (*snd x*) *else map-of xs i*).
i ∈# *mset* (*map fst xs*)#} (**is** - = *add-mset* - *?A*) **by** *simp*
also from *Cons.prems* **have** *?A* = {#*the* (*map-of xs i*). *i* :# *mset* (*map fst xs*)#}
by (*cases x*, *intro image-mset-cong*) (*auto simp*: *in-multiset-in-set*)
also from *Cons.prems* **have** ... = *mset* (*map snd xs*) **by** (*intro Cons.IH*) *simp-all*
finally show *?case* **by** *simp*

qed *simp-all*

lemma *msubset-mset-set-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* $\subseteq\#$ *mset-set B* $\longleftrightarrow$ $A \subseteq B$
using *assms set-mset-mono subset-imp-msubset-mset-set* **by** *fastforce*

lemma *mset-set-eq-iff*[*simp*]:
assumes *finite A finite B*
shows *mset-set A* = *mset-set B* $\longleftrightarrow$ $A = B$
using *assms* **by** (*fastforce dest*: *finite-set-mset-mset-set*)

lemma *image-mset-mset-set*:
assumes *inj-on f A*
shows *image-mset f* (*mset-set A*) = *mset-set* (*f* ‘ *A*)
proof *cases*
assume *finite A*
from *this* ‹*inj-on f A*› **show** *?thesis*
by (*induct A*) *auto*
next
assume *infinite A*
from *this* ‹*inj-on f A*› **have** *infinite* (*f* ‘ *A*)
using *finite-imageD* **by** *blast*
from ‹*infinite A*› ‹*infinite* (*f* ‘ *A*)› **show** *?thesis* **by** *simp*
qed

68.10 More properties of the replicate, repeat, and image operations

lemma *in-replicate-mset*[*simp*]: $x \in\#$ *replicate-mset n y* $\longleftrightarrow$ $n > 0 \wedge x = y$
unfolding *replicate-mset-def* **by** (*induct n*) *auto*

lemma *set-mset-replicate-mset-subset*[*simp*]: *set-mset* (*replicate-mset n x*) = (*if n* = *0 then* {} *else* {*x*})
by *auto*

lemma *size-replicate-mset*[*simp*]: *size* (*replicate-mset n M*) = *n*
by (*induct n*, *simp-all*)

lemma *size-repeat-mset* [*simp*]: *size* (*repeat-mset n A*) = *n* ∗ *size A*
by (*induction n*) *auto*

lemma *size-multiset-sum* [*simp*]: *size* ($\sum x{\in}A.$ *f x* :: *'a multiset*) = ($\sum x{\in}A.$ *size* (*f x*))
by (*induction A rule*: *infinite-finite-induct*) *auto*

lemma *size-multiset-sum-list* [*simp*]: *size* ($\sum X{\leftarrow}Xs.$ *X* :: *'a multiset*) = ($\sum X{\leftarrow}Xs.$ *size X*)
by (*induction Xs*) *auto*

lemma *count-le-replicate-mset-subset-eq*: $n \leq$ *count* M $x \longleftrightarrow$ *replicate-mset* n x $\subseteq\#$ M
by (*auto simp add*: *mset-subset-eqI*) (*metis count-replicate-mset subseteq-mset-def*)

lemma *replicate-count-mset-eq-filter-eq*: *replicate* (*count* (*mset xs*) k) k = *filter* (*HOL.eq k*) *xs*
by (*induct xs*) *auto*

lemma *replicate-mset-eq-empty-iff* [*simp*]: *replicate-mset* n $a = \{\#\} \longleftrightarrow n = 0$
by (*induct n*) *simp-all*

lemma *replicate-mset-eq-iff*:
replicate-mset m a = *replicate-mset* n $b \longleftrightarrow m = 0 \wedge n = 0 \vee m = n \wedge a = b$
by (*auto simp add*: *multiset-eq-iff*)

lemma *repeat-mset-cancel1*: *repeat-mset* a A = *repeat-mset* a $B \longleftrightarrow A = B \vee a = 0$
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-cancel2*: *repeat-mset* a A = *repeat-mset* b $A \longleftrightarrow a = b \vee A = \{\#\}$
by (*auto simp*: *multiset-eq-iff*)

lemma *repeat-mset-eq-empty-iff*: *repeat-mset* n $A = \{\#\} \longleftrightarrow n = 0 \vee A = \{\#\}$
by (*cases n*) *auto*

lemma *image-replicate-mset* [*simp*]:
image-mset f (*replicate-mset* n a) = *replicate-mset* n (f a)
by (*induct n*) *simp-all*

lemma *replicate-mset-msubseteq-iff*:
replicate-mset m $a \subseteq\#$ *replicate-mset* n $b \longleftrightarrow m = 0 \vee a = b \wedge m \leq n$
by (*cases m*)
(*auto simp*: *insert-subset-eq-iff simp flip*: *count-le-replicate-mset-subset-eq*)

lemma *msubseteq-replicate-msetE*:
assumes $A \subseteq\#$ *replicate-mset* n a
obtains m **where** $m \leq n$ **and** A = *replicate-mset* m a
proof (*cases* $n = 0$)
case *True*
with *assms that* **show** *thesis*
by *simp*
next
case *False*
from *assms* **have** *set-mset* $A \subseteq$ *set-mset* (*replicate-mset* n a)
by (*rule set-mset-mono*)
with *False* **have** *set-mset* $A \subseteq \{a\}$
by *simp*

```
  then have ∃m. A = replicate-mset m a
  proof (induction A)
    case empty
    then show ?case
      by simp
  next
    case (add b A)
    then obtain m where A = replicate-mset m a
      by auto
    with add.prems show ?case
      by (auto intro: exI [of - Suc m])
  qed
  then obtain m where A: A = replicate-mset m a ..
  with assms have m ≤ n
    by (auto simp add: replicate-mset-msubseteq-iff)
  then show thesis using A ..
qed

lemma count-image-mset-lt-imp-lt-raw:
  assumes
    finite A and
    A = set-mset M ∪ set-mset N and
    count (image-mset f M) b < count (image-mset f N) b
  shows ∃x. f x = b ∧ count M x < count N x
  using assms
proof (induct A arbitrary: M N b rule: finite-induct)
  case (insert x F)
  note fin = this(1) and x-ni-f = this(2) and ih = this(3) and x-f-eq-m-n =
this(4) and
    cnt-b = this(5)

  let ?Ma = {#y ∈# M. y ≠ x#}
  let ?Mb = {#y ∈# M. y = x#}
  let ?Na = {#y ∈# N. y ≠ x#}
  let ?Nb = {#y ∈# N. y = x#}

  have m-part: M = ?Mb + ?Ma and n-part: N = ?Nb + ?Na
    using multiset-partition by blast+

  have f-eq-ma-na: F = set-mset ?Ma ∪ set-mset ?Na
    using x-f-eq-m-n x-ni-f by auto

  show ?case
  proof (cases count (image-mset f ?Ma) b < count (image-mset f ?Na) b)
    case cnt-ba: True
    obtain xa where f xa = b and count ?Ma xa < count ?Na xa
      using ih[OF f-eq-ma-na cnt-ba] by blast
    thus ?thesis
      by (metis count-filter-mset not-less0)
```

```
  next
    case cnt-ba: False
    have fx-eq-b: f x = b
      using cnt-b cnt-ba
      by (subst (asm) m-part, subst (asm) n-part,
          auto simp: filter-eq-replicate-mset split: if-splits)
    moreover have count M x < count N x
      using cnt-b cnt-ba
      by (subst (asm) m-part, subst (asm) n-part,
          auto simp: filter-eq-replicate-mset split: if-splits)
    ultimately show ?thesis
      by blast
  qed
qed auto

lemma count-image-mset-lt-imp-lt:
  assumes cnt-b: count (image-mset f M) b < count (image-mset f N) b
  shows ∃ x. f x = b ∧ count M x < count N x
  by (rule count-image-mset-lt-imp-lt-raw[of set-mset M ∪ set-mset N, OF - refl
cnt-b]) auto

lemma count-image-mset-le-imp-lt-raw:
  assumes
    finite A and
    A = set-mset M ∪ set-mset N and
    count (image-mset f M) (f a) + count N a < count (image-mset f N) (f a) +
count M a
  shows ∃ b. f b = f a ∧ count M b < count N b
  using assms
proof (induct A arbitrary: M N rule: finite-induct)
  case (insert x F)
  note fin = this(1) and x-ni-f = this(2) and ih = this(3) and x-f-eq-m-n =
this(4) and
    cnt-lt = this(5)

  let ?Ma = {#y ∈# M. y ≠ x#}
  let ?Mb = {#y ∈# M. y = x#}
  let ?Na = {#y ∈# N. y ≠ x#}
  let ?Nb = {#y ∈# N. y = x#}

  have m-part: M = ?Mb + ?Ma and n-part: N = ?Nb + ?Na
    using multiset-partition by blast+

  have f-eq-ma-na: F = set-mset ?Ma ∪ set-mset ?Na
    using x-f-eq-m-n x-ni-f by auto

  show ?case
  proof (cases f x = f a)
    case fx-ne-fa: False
```

```
    have cnt-fma-fa: count (image-mset f ?Ma) (f a) = count (image-mset f M) (f
a)
      using fx-ne-fa by (subst (2) m-part) (auto simp: filter-eq-replicate-mset)
    have cnt-fna-fa: count (image-mset f ?Na) (f a) = count (image-mset f N) (f
a)
      using fx-ne-fa by (subst (2) n-part) (auto simp: filter-eq-replicate-mset)
    have cnt-ma-a: count ?Ma a = count M a
      using fx-ne-fa by (subst (2) m-part) (auto simp: filter-eq-replicate-mset)
    have cnt-na-a: count ?Na a = count N a
      using fx-ne-fa by (subst (2) n-part) (auto simp: filter-eq-replicate-mset)

    obtain b where fb-eq-fa: f b = f a and cnt-b: count ?Ma b < count ?Na b
        using ih[OF f-eq-ma-na] cnt-lt unfolding cnt-fma-fa cnt-fna-fa cnt-ma-a
cnt-na-a by blast
    have fx-ne-fb: f x ≠ f b
      using fb-eq-fa fx-ne-fa by simp

    have cnt-ma-b: count ?Ma b = count M b
      using fx-ne-fb by (subst (2) m-part) auto
    have cnt-na-b: count ?Na b = count N b
      using fx-ne-fb by (subst (2) n-part) auto

    show ?thesis
      using fb-eq-fa cnt-b unfolding cnt-ma-b cnt-na-b by blast
  next
    case fx-eq-fa: True
    show ?thesis
    proof (cases x = a)
      case x-eq-a: True
      have count (image-mset f ?Ma) (f a) + count ?Na a
        < count (image-mset f ?Na) (f a) + count ?Ma a
        using cnt-lt x-eq-a by (subst (asm) (1 2) m-part, subst (asm) (1 2) n-part,
            auto simp: filter-eq-replicate-mset)
      thus ?thesis
        using ih[OF f-eq-ma-na] by (metis count-filter-mset nat-neq-iff)
    next
      case x-ne-a: False
      show ?thesis
      proof (cases count M x < count N x)
        case True
        thus ?thesis
          using fx-eq-fa by blast
      next
        case False
        hence cnt-x: count M x ≥ count N x
          by fastforce
        have count M x + count (image-mset f ?Ma) (f a) + count ?Na a
          < count N x + count (image-mset f ?Na) (f a) + count ?Ma a
```

```
        using cnt-lt x-ne-a fx-eq-fa by (subst (asm) (1 2) m-part, subst (asm) (1
2) n-part,
          auto simp: filter-eq-replicate-mset)
       hence count (image-mset f ?Ma) (f a) + count ?Na a
        < count (image-mset f ?Na) (f a) + count ?Ma a
        using cnt-x by linarith
       thus ?thesis
        using ih[OF f-eq-ma-na] by (metis count-filter-mset nat-neq-iff)
     qed
   qed
  qed
qed auto

lemma count-image-mset-le-imp-lt:
  assumes
    count (image-mset f M) (f a) ≤ count (image-mset f N) (f a) and
    count M a > count N a
  shows ∃ b. f b = f a ∧ count M b < count N b
   using assms by (auto intro: count-image-mset-le-imp-lt-raw[of set-mset M ∪
set-mset N])

lemma size-filter-unsat-elem:
  assumes x ∈# M and ¬ P x
  shows size {#x ∈# M. P x#} < size M
proof −
  have size (filter-mset P M) ≠ size M
    using assms
     by (metis dual-order.strict-iff-order filter-mset-eq-conv mset-subset-size sub-
set-mset.nless-le)
  then show ?thesis
    by (meson leD nat-neq-iff size-filter-mset-lesseq)
qed

lemma size-filter-ne-elem: x ∈# M ⟹ size {#y ∈# M. y ≠ x#} < size M
  by (simp add: size-filter-unsat-elem[of x M λy. y ≠ x])

lemma size-eq-ex-count-lt:
  assumes size M = size N and M ≠ N
  shows ∃ x. count M x < count N x
proof −
  from ‹M ≠ N› obtain x where count M x ≠ count N x
    using count-inject by blast
  then consider (lt) count M x < count N x | (gt) count M x > count N x
    by linarith
  then show ?thesis
  proof cases
    case lt
    then show ?thesis ..
  next
```

case *gt*
from ‹*size M = size N*› **have** *size {#y ∈# M. y = x#} + size {#y ∈# M. y ≠ x#} =*
size {#y ∈# N. y = x#} + size {#y ∈# N. y ≠ x#}
using *multiset-partition* **by** (*metis size-union*)
with *gt* **have** ∗: *size {#y ∈# M. y ≠ x#} < size {#y ∈# N. y ≠ x#}*
by (*simp add*: *filter-eq-replicate-mset*)
then obtain *y* **where** *count {#y ∈# M. y ≠ x#} y < count {#y ∈# N. y ≠ x#} y*
using *size-lt-imp-ex-count-lt*[*OF* ∗] **by** *blast*
then have *count M y < count N y*
by (*metis count-filter-mset less-asym*)
then show *?thesis* **..**
qed
qed

68.11 Big operators

locale *comm-monoid-mset = comm-monoid*
begin

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*)

context
begin

definition *F* :: *'a multiset ⇒ 'a*
where *eq-fold*: *F M = fold-mset f* **1** *M*

lemma *empty* [*simp*]: *F {#}* = **1**
by (*simp add*: *eq-fold*)

lemma *singleton* [*simp*]: *F {#x#} = x*
proof −
interpret *comp-fun-commute*
by *standard* (*simp add*: *fun-eq-iff left-commute*)
show *?thesis* **by** (*simp add*: *eq-fold*)
qed

lemma *union* [*simp*]: *F (M + N) = F M ∗ F N*
proof −
interpret *comp-fun-commute f*
by *standard* (*simp add*: *fun-eq-iff left-commute*)
show *?thesis*
by (*induct N*) (*simp-all add*: *left-commute eq-fold*)

qed

lemma *add-mset* [*simp*]: *F* (*add-mset x N*) = *x* ∗ *F N*
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of x N*] *union* **by** (*simp add*: *ac-simps*)

lemma *insert* [*simp*]:
shows *F* (*image-mset g* (*add-mset x A*)) = *g x* ∗ *F* (*image-mset g A*)
by (*simp add*: *eq-fold*)

lemma *remove*:
assumes *x* ∈# *A*
shows *F A* = *x* ∗ *F* (*A* − {#*x*#})
using *multi-member-split*[*OF assms*] **by** *auto*

lemma *neutral*:
∀ *x*∈#*A*. *x* = **1** ⟹ *F A* = **1**
by (*induct A*) *simp-all*

lemma *neutral-const* [*simp*]:
F (*image-mset* (λ-. **1**) *A*) = **1**
by (*simp add*: *neutral*)

private lemma *F-image-mset-product*:
F {#*g x j* ∗ *F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#} =
F (*image-mset* (*g x*) *B*) ∗ *F* {#*F* {#*g i j*. *i* ∈# *A*#}. *j* ∈# *B*#}
by (*induction B*) (*simp-all add*: *left-commute semigroup.assoc semigroup-axioms*)

lemma *swap*:
F (*image-mset* (λ*i*. *F* (*image-mset* (*g i*) *B*)) *A*) =
F (*image-mset* (λ*j*. *F* (*image-mset* (λ*i*. *g i j*) *A*)) *B*)
apply (*induction A*, *simp*)
apply (*induction B*, *auto simp add*: *F-image-mset-product ac-simps*)
done

lemma *distrib*: *F* (*image-mset* (λ*x*. *g x* ∗ *h x*) *A*) = *F* (*image-mset g A*) ∗ *F* (*image-mset h A*)
by (*induction A*) (*auto simp*: *ac-simps*)

lemma *union-disjoint*:
A ∩# *B* = {#} ⟹ *F* (*image-mset g* (*A* ∪# *B*)) = *F* (*image-mset g A*) ∗ *F* (*image-mset g B*)
by (*induction A*) (*auto simp*: *ac-simps*)

end
end

lemma *comp-fun-commute-plus-mset*[*simp*]: *comp-fun-commute* ((+) :: *'a multiset* ⇒ - ⇒ -)

by *standard* (*simp add*: *add-ac comp-def*)

declare *comp-fun-commute.fold-mset-add-mset*[*OF comp-fun-commute-plus-mset*, *simp*]

lemma *in-mset-fold-plus-iff*[*iff*]: $x \in\# \mathit{fold\text{-}mset}\ (+)\ M\ NN \longleftrightarrow x \in\# M \vee (\exists N.\ N \in\# NN \wedge x \in\# N)$
by (*induct NN*) *auto*

context *comm-monoid-add*
begin

sublocale *sum-mset*: *comm-monoid-mset plus 0*
defines *sum-mset* = *sum-mset.F* **..**

lemma *sum-unfold-sum-mset*:
$\mathit{sum}\ f\ A = \mathit{sum\text{-}mset}\ (\mathit{image\text{-}mset}\ f\ (\mathit{mset\text{-}set}\ A))$
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

end

notation *sum-mset* ($\langle\sum_{\#}\rangle$)

syntax (*ASCII*)
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add*
($\langle$($\langle$*indent=3 notation=*$\langle$*binder SUM*$\rangle\rangle$*SUM -:#-. -*)$\rangle$ [*0*, *51*, *10*] *10*)
syntax
-sum-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-add*
($\langle$($\langle$*indent=3 notation=*$\langle$*binder* $\sum\rangle\rangle\sum$ *-∈#-. -*)$\rangle$ [*0*, *51*, *10*] *10*)
syntax-consts
-sum-mset-image ⇌ *sum-mset*
translations
$\sum i \in\# A.\ b \rightleftharpoons \mathit{CONST\ sum\text{-}mset}\ (\mathit{CONST\ image\text{-}mset}\ (\lambda i.\ b)\ A)$

context *comm-monoid-add*
begin

lemma *sum-mset-sum-list*:
$\mathit{sum\text{-}mset}\ (\mathit{mset}\ xs) = \mathit{sum\text{-}list}\ xs$
by (*induction xs*) *auto*

lemma *sum-mset-sum*: $\mathit{sum\text{-}mset}\ (\sum x{\in}A.\ f\ x) = (\sum x{\in}A.\ \mathit{sum\text{-}mset}\ (f\ x))$
by (*induction A rule*: *infinite-finite-induct*) *auto*

end

context *canonically-ordered-monoid-add*
begin

lemma *sum-mset-0-iff* [*simp*]:
sum-mset M = 0 $\longleftrightarrow$ $(\forall x \in$ *set-mset M*. *x = 0*$)$
by (*induction M*) *auto*

end

context *ordered-comm-monoid-add*
begin

lemma *sum-mset-mono*:
sum-mset (*image-mset f K*) $\leq$ *sum-mset* (*image-mset g K*)
if $\bigwedge i$. *i* $\in\#$ *K* $\Longrightarrow$ *f i* $\leq$ *g i*
using *that* **by** (*induction K*) (*simp-all add*: *add-mono*)

end

context *cancel-comm-monoid-add*
begin

lemma *sum-mset-diff*:
sum-mset (*M* − *N*) = *sum-mset M* − *sum-mset N* **if** *N* $\subseteq\#$ *M* **for** *M N* :: *'a multiset*
using *that* **by** (*auto simp add*: *subset-mset.le-iff-add*)

end

context *ab-group-add*
begin

lemma *sum-mset-negf*: $(\sum x{\in}\#M.\ -f\ x :: 'a) = -(\sum x{\in}\#M.\ f\ x)$
proof −
have $(\sum x{\in}\#M.\ -f\ x) + (\sum x{\in}\#M.\ f\ x) = 0$
by (*subst sum-mset.distrib* [*symmetric*]) *auto*
thus *?thesis*
using *local.eq-neg-iff-add-eq-0* **by** *blast*
qed

lemma *sum-mset-subtractf*: $(\sum x{\in}\#M.\ f\ x - g\ x) = (\sum x{\in}\#M.\ f\ x) - (\sum x{\in}\#M.\ g\ x)$
using *sum-mset.distrib*[*of f* $\lambda x.\ -g\ x$ *M*] **by** (*simp add*: *sum-mset-negf*)

end

context *semiring-0*
begin

lemma *sum-mset-distrib-left*:
c ∗ $(\sum x \in\#$ *M*. *f x*$) = (\sum x \in\#$ *M*. *c* ∗ *f*(*x*)$)$
by (*induction M*) (*simp-all add*: *algebra-simps*)

lemma *sum-mset-distrib-right*:
(∑ x ∈# M. f x) * c = (∑ x ∈# M. f x * c)
by (*induction M*) (*simp-all add*: *algebra-simps*)

end

lemma *sum-mset-product*:
fixes f :: 'a::{*comm-monoid-add*,*times*} ⇒ 'b::*semiring-0*
shows (∑ i ∈# A. f i) * (∑ i ∈# B. g i) = (∑ i∈#A. ∑ j∈#B. f i * g j)
by (*subst sum-mset.swap*) (*simp add*: *sum-mset-distrib-left sum-mset-distrib-right*)

context *semiring-1*
begin

lemma *sum-mset-replicate-mset* [*simp*]:
sum-mset (*replicate-mset n a*) = *of-nat n* * *a*
by (*induction n*) (*simp-all add*: *algebra-simps*)

lemma *sum-mset-delta*:
sum-mset (*image-mset* (λx. *if x* = *y then c else 0*) *A*) = *c* * *of-nat* (*count A y*)
by (*induction A*) (*simp-all add*: *algebra-simps*)

lemma *sum-mset-delta'*:
sum-mset (*image-mset* (λx. *if y* = *x then c else 0*) *A*) = *c* * *of-nat* (*count A y*)
by (*induction A*) (*simp-all add*: *algebra-simps*)

end

lemma *of-nat-sum-mset* [*simp*]:
of-nat (*sum-mset A*) = *sum-mset* (*image-mset of-nat A*)
by (*induction A*) *auto*

lemma *size-eq-sum-mset*:
size M = (∑ a∈#M. *1*)
using *image-mset-const-eq* [*of 1*::*nat M*] **by** *simp*

lemma *size-mset-set* [*simp*]:
size (*mset-set A*) = *card A*
by (*simp only*: *size-eq-sum-mset card-eq-sum sum-unfold-sum-mset*)

lemma *sum-mset-constant* [*simp*]:
fixes *y* :: 'b::*semiring-1*
shows ‹(∑ x∈#A. *y*) = *of-nat* (*size A*) * *y*›
by (*induction A*) (*auto simp*: *algebra-simps*)

lemma *set-mset-Union-mset*[*simp*]: *set-mset* (∑ # *MM*) = (⋃ *M* ∈ *set-mset MM*. *set-mset M*)
by (*induct MM*) *auto*

lemma *in-Union-mset-iff*[*iff*]: $x \in\# \sum_\# MM \longleftrightarrow (\exists M.\ M \in\# MM \wedge x \in\# M)$
by (*induct MM*) *auto*

lemma *count-sum*:
$count\ (sum\ f\ A)\ x = sum\ (\lambda a.\ count\ (f\ a)\ x)\ A$
by (*induct A rule*: *infinite-finite-induct*) *simp-all*

lemma *sum-eq-empty-iff*:
assumes *finite A*
shows $sum\ f\ A = \{\#\} \longleftrightarrow (\forall a{\in}A.\ f\ a = \{\#\})$
using *assms* **by** *induct simp-all*

lemma *mset-concat*: $mset\ (concat\ xss) = (\sum xs{\leftarrow}xss.\ mset\ xs)$
by (*induction xss*) *auto*

lemma *set-mset-sum-list* [*simp*]: $set\text{-}mset\ (sum\text{-}list\ xs) = (\bigcup x{\in}set\ xs.\ set\text{-}mset\ x)$
by (*induction xs*) *auto*

lemma *filter-mset-sum-list*: $filter\text{-}mset\ P\ (sum\text{-}list\ xs) = sum\text{-}list\ (map\ (filter\text{-}mset\ P)\ xs)$
by (*induction xs*) *simp-all*

lemma *sum-mset-singleton-mset* [*simp*]: $(\sum x{\in}\#A.\ \{\#f\ x\#\}) = image\text{-}mset\ f\ A$
by (*induction A*) *auto*

lemma *sum-list-singleton-mset* [*simp*]: $(\sum x{\leftarrow}xs.\ \{\#f\ x\#\}) = image\text{-}mset\ f\ (mset\ xs)$
by (*induction xs*) *auto*

lemma *Union-mset-empty-conv*[*simp*]: $\sum_\# M = \{\#\} \longleftrightarrow (\forall i{\in}\#M.\ i = \{\#\})$
by (*induction M*) *auto*

lemma *Union-image-single-mset*[*simp*]: $\sum_\# (image\text{-}mset\ (\lambda x.\ \{\#x\#\})\ m) = m$
by(*induction m*) *auto*

lemma *size-mset-sum-mset-conv* [*simp*]: $size\ (\sum_\# A :: 'a\ multiset) = (\sum X{\in}\#A.\ size\ X)$
by (*induction A*) *auto*

lemma *sum-mset-image-mset-mono-strong*:
assumes $A \subseteq\# B$ **and** *f-subeq-g*: $\bigwedge x.\ x \in\# A \Longrightarrow f\ x \subseteq\# g\ x$
shows $(\sum x{\in}\#A.\ f\ x) \subseteq\# (\sum x{\in}\#B.\ g\ x)$
proof −
define B' **where**
$B' = B - A$

have $B = A + B'$
using B'-*def assms*(*1*) **by** *fastforce*

have $\sum_{\#}$ (*image-mset f A*) $\subseteq\#$ $\sum_{\#}$ (*image-mset g A*)
using *f-subeq-g* **by** (*induction A*) (*auto intro*!: *subset-mset.add-mono*)
also have ... $\subseteq\#$ $\sum_{\#}$ (*image-mset g A*) + $\sum_{\#}$ (*image-mset g B'*)
by *simp*
also have ... = $\sum_{\#}$ (*image-mset g A* + *image-mset g B'*)
by *simp*
also have ... = $\sum_{\#}$ (*image-mset g* (*A* + *B'*))
by *simp*
also have ... = $\sum_{\#}$ (*image-mset g B*)
unfolding ‹*B* = *A* + *B'*› **..**
finally show *?thesis* **.**
qed

context *comm-monoid-mult*
begin

sublocale *prod-mset*: *comm-monoid-mset times 1*
defines *prod-mset* = *prod-mset.F* **..**

lemma *prod-mset-empty*:
prod-mset {#} = *1*
by (*fact prod-mset.empty*)

lemma *prod-mset-singleton*:
prod-mset {#*x*#} = *x*
by (*fact prod-mset.singleton*)

lemma *prod-mset-Un*:
prod-mset (*A* + *B*) = *prod-mset A* ∗ *prod-mset B*
by (*fact prod-mset.union*)

lemma *prod-mset-prod-list*:
prod-mset (*mset xs*) = *prod-list xs*
by (*induct xs*) *auto*

lemma *prod-mset-replicate-mset* [*simp*]:
prod-mset (*replicate-mset n a*) = *a* ^ *n*
by (*induct n*) *simp-all*

lemma *prod-unfold-prod-mset*:
prod f A = *prod-mset* (*image-mset f* (*mset-set A*))
by (*cases finite A*) (*induct A rule*: *finite-induct*, *simp-all*)

lemma *prod-mset-multiplicity*:
prod-mset M = *prod* (λ*x*. *x* ^ *count M x*) (*set-mset M*)
by (*simp add*: *fold-mset-def prod.eq-fold prod-mset.eq-fold funpow-times-power comp-def*)

lemma *prod-mset-delta*: *prod-mset* (*image-mset* (λ*x*. *if x = y then c else 1*) *A*) = *c* ^ *count A y*
by (*induction A*) *simp-all*

lemma *prod-mset-delta'*: *prod-mset* (*image-mset* (λ*x*. *if y = x then c else 1*) *A*) = *c* ^ *count A y*
by (*induction A*) *simp-all*

lemma *prod-mset-subset-imp-dvd*:
assumes *A* ⊆# *B*
shows *prod-mset A dvd prod-mset B*
proof −
from *assms* **have** *B* = (*B* − *A*) + *A* **by** (*simp add*: *subset-mset.diff-add*)
also have *prod-mset* ... = *prod-mset* (*B* − *A*) ∗ *prod-mset A* **by** *simp*
also have *prod-mset A dvd* ... **by** *simp*
finally show *?thesis* **.**
qed

lemma *dvd-prod-mset*:
assumes *x* ∈# *A*
shows *x dvd prod-mset A*
using *assms prod-mset-subset-imp-dvd* [*of* {#*x*#} *A*] **by** *simp*

end

notation *prod-mset* (‹∏ #›)

syntax (*ASCII*)
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult*
(‹(‹*indent=3 notation=‹binder PROD››PROD -:#-. -*)› [*0*, *51*, *10*] *10*)
syntax
-prod-mset-image :: *pttrn* ⇒ *'b set* ⇒ *'a* ⇒ *'a*::*comm-monoid-mult*
(‹(‹*indent=3 notation=‹binder* ∏ ››∏ *-∈#-. -*)› [*0*, *51*, *10*] *10*)
syntax-consts
-prod-mset-image ⇌ *prod-mset*
translations
∏ *i* ∈# *A*. *b* ⇌ *CONST prod-mset* (*CONST image-mset* (λ*i*. *b*) *A*)

lemma *prod-mset-constant* [*simp*]: (∏ *-∈#A*. *c*) = *c* ^ *size A*
by (*simp add*: *image-mset-const-eq*)

lemma (**in** *semidom*) *prod-mset-zero-iff* [*iff*]:
prod-mset A = *0* ⟷ *0* ∈# *A*
by (*induct A*) *auto*

lemma (**in** *semidom-divide*) *prod-mset-diff*:
assumes *B* ⊆# *A* **and** *0* ∉# *B*
shows *prod-mset* (*A* − *B*) = *prod-mset A div prod-mset B*
proof −

```
  from assms obtain C where A = B + C
    by (metis subset-mset.add-diff-inverse)
  with assms show ?thesis by simp
qed

lemma (in semidom-divide) prod-mset-minus:
  assumes a ∈# A and a ≠ 0
  shows prod-mset (A - {#a#}) = prod-mset A div a
  using assms prod-mset-diff [of {#a#} A] by auto

lemma (in normalization-semidom) normalize-prod-mset-normalize:
  normalize (prod-mset (image-mset normalize A)) = normalize (prod-mset A)
proof (induction A)
  case (add x A)
  have normalize (prod-mset (image-mset normalize (add-mset x A))) =
        normalize (x * normalize (prod-mset (image-mset normalize A)))
    by simp
  also note add.IH
  finally show ?case by simp
qed auto

lemma (in algebraic-semidom) is-unit-prod-mset-iff:
  is-unit (prod-mset A) ⟷ (∀ x ∈# A. is-unit x)
  by (induct A) (auto simp: is-unit-mult-iff)

lemma (in normalization-semidom-multiplicative) normalize-prod-mset:
  normalize (prod-mset A) = prod-mset (image-mset normalize A)
  by (induct A) (simp-all add: normalize-mult)

lemma (in normalization-semidom-multiplicative) normalized-prod-msetI:
  assumes ⋀a. a ∈# A ⟹ normalize a = a
  shows normalize (prod-mset A) = prod-mset A
proof -
  from assms have image-mset normalize A = A
    by (induct A) simp-all
  then show ?thesis by (simp add: normalize-prod-mset)
qed

lemma image-prod-mset-multiplicity:
  prod-mset (image-mset f M) = prod (λx. f x ^ count M x) (set-mset M)
proof (induction M)
  case (add x M)
  show ?case
  proof (cases x ∈ set-mset M)
    case True
    have (∏ y∈set-mset (add-mset x M). f y ^ count (add-mset x M) y) =
          (∏ y∈set-mset M. (if y = x then f x else 1) * f y ^ count M y)
      using True add by (intro prod.cong) auto
    also have ... = f x * (∏ y∈set-mset M. f y ^ count M y)
```

```
    using True by (subst prod.distrib) auto
  also note add.IH [symmetric]
  finally show ?thesis using True by simp
 next
  case False
  hence (∏ y∈set-mset (add-mset x M). f y ^ count (add-mset x M) y) =
          f x * (∏ y∈set-mset M. f y ^ count (add-mset x M) y)
    by (auto simp: not-in-iff)
  also have (∏ y∈set-mset M. f y ^ count (add-mset x M) y) =
            (∏ y∈set-mset M. f y ^ count M y)
    using False by (intro prod.cong) auto
  also note add.IH [symmetric]
  finally show ?thesis by simp
 qed
qed auto
```

68.12 Multiset as order-ignorant lists

```
context linorder
begin

lemma mset-insort [simp]:
  mset (insort-key k x xs) = add-mset x (mset xs)
  by (induct xs) simp-all

lemma mset-sort [simp]:
  mset (sort-key k xs) = mset xs
  by (induct xs) simp-all
```

This lemma shows which properties suffice to show that a function f with $f\ xs = ys$ behaves like sort.

```
lemma properties-for-sort-key:
  assumes mset ys = mset xs
    and ⋀k. k ∈ set ys ⟹ filter (λx. f k = f x) ys = filter (λx. f k = f x) xs
    and sorted (map f ys)
  shows sort-key f xs = ys
  using assms
proof (induct xs arbitrary: ys)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  from Cons.prems(2) have
    ∀k ∈ set ys. filter (λx. f k = f x) (remove1 x ys) = filter (λx. f k = f x) xs
    by (simp add: filter-remove1)
  with Cons.prems have sort-key f xs = remove1 x ys
    by (auto intro!: Cons.hyps simp add: sorted-map-remove1)
  moreover from Cons.prems have x ∈# mset ys
    by auto
  then have x ∈ set ys
```

```
    by simp
  ultimately show ?case using Cons.prems by (simp add: insert-key-remove1)
qed

lemma properties-for-sort:
  assumes multiset: mset ys = mset xs
    and sorted ys
  shows sort xs = ys
proof (rule properties-for-sort-key)
  from multiset show mset ys = mset xs .
  from ‹sorted ys› show sorted (map (λx. x) ys) by simp
  from multiset have length (filter (λy. k = y) ys) = length (filter (λx. k = x)
xs) for k
    by (metis mset-filter size-mset)
  then have replicate (length (filter (λy. k = y) ys)) k =
    replicate (length (filter (λx. k = x) xs)) k for k
    by simp
  then show k ∈ set ys ⟹ filter (λy. k = y) ys = filter (λx. k = x) xs for k
    by (simp add: replicate-length-filter)
qed

lemma sort-key-inj-key-eq:
  assumes mset-equal: mset xs = mset ys
    and inj-on f (set xs)
    and sorted (map f ys)
  shows sort-key f xs = ys
proof (rule properties-for-sort-key)
  from mset-equal
  show mset ys = mset xs by simp
  from ‹sorted (map f ys)›
  show sorted (map f ys) .
  show [x←ys . f k = f x] = [x←xs . f k = f x] if k ∈ set ys for k
  proof −
    from mset-equal
    have set-equal: set xs = set ys by (rule mset-eq-setD)
    with that have insert k (set ys) = set ys by auto
    with ‹inj-on f (set xs)› have inj: inj-on f (insert k (set ys))
      by (simp add: set-equal)
    from inj have [x←ys . f k = f x] = filter (HOL.eq k) ys
      by (auto intro!: inj-on-filter-key-eq)
    also have ... = replicate (count (mset ys) k) k
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = replicate (count (mset xs) k) k
      using mset-equal by simp
    also have ... = filter (HOL.eq k) xs
      by (simp add: replicate-count-mset-eq-filter-eq)
    also have ... = [x←xs . f k = f x]
     using inj by (auto intro!: inj-on-filter-key-eq [symmetric] simp add: set-equal)
    finally show ?thesis .
```

```
  qed
qed

lemma sort-key-eq-sort-key:
  assumes mset xs = mset ys
    and inj-on f (set xs)
  shows sort-key f xs = sort-key f ys
  by (rule sort-key-inj-key-eq) (simp-all add: assms)

lemma sort-key-by-quicksort:
  sort-key f xs = sort-key f [x←xs. f x < f (xs ! (length xs div 2))]
    @ [x←xs. f x = f (xs ! (length xs div 2))]
    @ sort-key f [x←xs. f x > f (xs ! (length xs div 2))] (is sort-key f ?lhs = ?rhs)
proof (rule properties-for-sort-key)
  show mset ?rhs = mset ?lhs
    by (rule multiset-eqI) auto
  show sorted (map f ?rhs)
    by (auto simp add: sorted-append intro: sorted-map-same)
next
  fix l
  assume l ∈ set ?rhs
  let ?pivot = f (xs ! (length xs div 2))
  have *: ⋀x. f l = f x ⟷ f x = f l by auto
  have [x ← sort-key f xs . f x = f l] = [x ← xs. f x = f l]
   unfolding filter-sort by (rule properties-for-sort-key) (auto intro: sorted-map-same)
  with * have **: [x ← sort-key f xs . f l = f x] = [x ← xs. f l = f x] by simp
  have ⋀x P. P (f x) ?pivot ∧ f l = f x ⟷ P (f l) ?pivot ∧ f l = f x by auto
  then have ⋀P. [x ← sort-key f xs . P (f x) ?pivot ∧ f l = f x] =
    [x ← sort-key f xs. P (f l) ?pivot ∧ f l = f x] by simp
  note *** = this [of (<)] this [of (>)] this [of (=)]
  show [x ← ?rhs. f l = f x] = [x ← ?lhs. f l = f x]
  proof (cases f l ?pivot rule: linorder-cases)
    case less
    then have f l ≠ ?pivot and ¬ f l > ?pivot by auto
    with less show ?thesis
      by (simp add: filter-sort [symmetric] ** ***)
  next
    case equal then show ?thesis
      by (simp add: * less-le)
  next
    case greater
    then have f l ≠ ?pivot and ¬ f l < ?pivot by auto
    with greater show ?thesis
      by (simp add: filter-sort [symmetric] ** ***)
  qed
qed

lemma sort-by-quicksort:
  sort xs = sort [x←xs. x < xs ! (length xs div 2)]
```

```
    @ [x←xs. x = xs ! (length xs div 2)]
    @ sort [x←xs. x > xs ! (length xs div 2)] (is sort ?lhs = ?rhs)
  using sort-key-by-quicksort [of λx. x, symmetric] by simp

lemma sort-append:
  assumes ⋀x y. x ∈ set xs ⟹ y ∈ set ys ⟹ x ≤ y
  shows   sort (xs @ ys) = sort xs @ sort ys
  using assms by (intro properties-for-sort) (auto simp: sorted-append)

lemma sort-append-replicate-left:
  (⋀y. y ∈ set xs ⟹ x ≤ y) ⟹ sort (replicate n x @ xs) = replicate n x @ sort
xs
  by (subst sort-append) auto

lemma sort-append-replicate-right:
  (⋀y. y ∈ set xs ⟹ x ≥ y) ⟹ sort (xs @ replicate n x) = sort xs @ replicate n
x
  by (subst sort-append) auto
```

A stable parameterized quicksort

```
definition part :: ('b ⇒ 'a) ⇒ 'a ⇒ 'b list ⇒ 'b list × 'b list × 'b list where
  part f pivot xs = ([x ← xs. f x < pivot], [x ← xs. f x = pivot], [x ← xs. pivot <
f x])

lemma part-code [code]:
  part f pivot [] = ([], [], [])
  part f pivot (x # xs) = (let (lts, eqs, gts) = part f pivot xs; x' = f x in
    if x' < pivot then (x # lts, eqs, gts)
    else if x' > pivot then (lts, eqs, x # gts)
    else (lts, x # eqs, gts))
  by (auto simp add: part-def Let-def split-def)

lemma sort-key-by-quicksort-code [code]:
  sort-key f xs =
    (case xs of
      [] ⇒ []
    | [x] ⇒ xs
    | [x, y] ⇒ (if f x ≤ f y then xs else [y, x])
    | - ⇒
        let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
        in sort-key f lts @ eqs @ sort-key f gts)
proof (cases xs)
  case Nil then show ?thesis by simp
next
  case (Cons - ys) note hyps = Cons show ?thesis
  proof (cases ys)
    case Nil with hyps show ?thesis by simp
  next
    case (Cons - zs) note hyps = hyps Cons show ?thesis
```

```
    proof (cases zs)
      case Nil with hyps show ?thesis by auto
    next
      case Cons
      from sort-key-by-quicksort [of f xs]
      have sort-key f xs = (let (lts, eqs, gts) = part f (f (xs ! (length xs div 2))) xs
        in sort-key f lts @ eqs @ sort-key f gts)
      by (simp only: split-def Let-def part-def fst-conv snd-conv)
      with hyps Cons show ?thesis by (simp only: list.cases)
    qed
  qed
qed

end

hide-const (open) part

lemma sort-sorted-list-of-multiset-eq [simp]:
  ‹sort (sorted-list-of-multiset M) = sorted-list-of-multiset M› for M :: ‹'a::linorder
multiset›
  by (rule properties-for-sort) simp-all

lemma mset-remdups-subset-eq: mset (remdups xs) ⊆# mset xs
  by (induct xs) (auto intro: subset-mset.order-trans)

lemma mset-update:
  i < length ls ⟹ mset (ls[i := v]) = add-mset v (mset ls − {#ls ! i#})
proof (induct ls arbitrary: i)
  case Nil then show ?case by simp
next
  case (Cons x xs)
  show ?case
  proof (cases i)
    case 0 then show ?thesis by simp
  next
    case (Suc i')
    with Cons show ?thesis
      by (cases ‹x = xs ! i'›) auto
  qed
qed

lemma mset-swap:
  i < length ls ⟹ j < length ls ⟹
    mset (ls[j := ls ! i, i := ls ! j]) = mset ls
  by (cases i = j) (simp-all add: mset-update nth-mem-mset)

lemma mset-eq-finite:
  ‹finite {ys. mset ys = mset xs}›
proof −
```

have ‹{*ys*. *mset ys* = *mset xs*} ⊆ {*ys*. *set ys* ⊆ *set xs* ∧ *length ys* ≤ *length xs*}›
by (*auto simp add*: *dest*: *mset-eq-setD mset-eq-length*)
moreover have ‹*finite* {*ys*. *set ys* ⊆ *set xs* ∧ *length ys* ≤ *length xs*}›
using *finite-lists-length-le* **by** *blast*
ultimately show *?thesis*
by (*rule finite-subset*)
qed

68.13 The multiset order

definition *mult1* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult1 r = {(*N*, *M*). ∃ *a M0 K*. *M* = *add-mset a M0* ∧ *N* = *M0* + *K* ∧
(∀ *b*. *b* ∈# *K* ⟶ (*b*, *a*) ∈ *r*)}

definition *mult* :: ('*a* × '*a*) *set* ⇒ ('*a multiset* × '*a multiset*) *set* **where**
mult r = (*mult1 r*)$^+$

definition *multp* :: ('*a* ⇒ '*a* ⇒ *bool*) ⇒ '*a multiset* ⇒ '*a multiset* ⇒ *bool* **where**
multp r M N ⟷ (*M*, *N*) ∈ *mult* {(*x*, *y*). *r x y*}

declare *multp-def*[*pred-set-conv*]

lemma *mult1I*:
assumes *M* = *add-mset a M0* **and** *N* = *M0* + *K* **and** ⋀*b*. *b* ∈# *K* ⟹ (*b*, *a*) ∈ *r*
shows (*N*, *M*) ∈ *mult1 r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *mult1E*:
assumes (*N*, *M*) ∈ *mult1 r*
obtains *a M0 K* **where** *M* = *add-mset a M0 N* = *M0* + *K* ⋀*b*. *b* ∈# *K* ⟹ (*b*, *a*) ∈ *r*
using *assms* **unfolding** *mult1-def* **by** *blast*

lemma *mono-mult1*:
assumes *r* ⊆ *r*′ **shows** *mult1 r* ⊆ *mult1 r*′
unfolding *mult1-def* **using** *assms* **by** *blast*

lemma *mono-mult*:
assumes *r* ⊆ *r*′ **shows** *mult r* ⊆ *mult r*′
unfolding *mult-def* **using** *mono-mult1*[*OF assms*] *trancl-mono* **by** *blast*

lemma *mono-multp*[*mono*]: *r* ≤ *r*′ ⟹ *multp r* ≤ *multp r*′
unfolding *le-fun-def le-bool-def*
proof (*intro allI impI*)
fix *M N* :: '*a multiset*
assume ∀ *x xa*. *r x xa* ⟶ *r*′ *x xa*
hence {(*x*, *y*). *r x y*} ⊆ {(*x*, *y*). *r*′ *x y*}
by *blast*

```
  thus multp r M N ⟹ multp r' M N
    unfolding multp-def
    by (fact mono-mult[THEN subsetD, rotated])
qed

lemma not-less-empty [iff]: (M, {#}) ∉ mult1 r
  by (simp add: mult1-def)
```

68.13.1 Well-foundedness

```
lemma less-add:
  assumes mult1: (N, add-mset a M0) ∈ mult1 r
  shows
    (∃ M. (M, M0) ∈ mult1 r ∧ N = add-mset a M) ∨
     (∃ K. (∀ b. b ∈# K ⟶ (b, a) ∈ r) ∧ N = M0 + K)
proof −
  let ?r = λK a. ∀ b. b ∈# K ⟶ (b, a) ∈ r
  let ?R = λN M. ∃ a M0 K. M = add-mset a M0 ∧ N = M0 + K ∧ ?r K a
  obtain a' M0' K where M0: add-mset a M0 = add-mset a' M0'
    and N: N = M0' + K
    and r: ?r K a'
    using mult1 unfolding mult1-def by auto
  show ?thesis (is ?case1 ∨ ?case2)
  proof −
    from M0 consider M0 = M0' a = a'
      | K' where M0 = add-mset a' K' M0' = add-mset a K'
      by atomize-elim (simp only: add-eq-conv-ex)
    then show ?thesis
    proof cases
      case 1
      with N r have ?r K a ∧ N = M0 + K by simp
      then have ?case2 ..
      then show ?thesis ..
    next
      case 2
      from N 2(2) have n: N = add-mset a (K' + K) by simp
      with r 2(1) have ?R (K' + K) M0 by blast
      with n have ?case1 by (simp add: mult1-def)
      then show ?thesis ..
    qed
  qed
qed

lemma all-accessible:
  assumes wf r
  shows ∀ M. M ∈ Wellfounded.acc (mult1 r)
proof
  let ?R = mult1 r
  let ?W = Wellfounded.acc ?R
```

```
{
  fix M M0 a
  assume M0: M0 ∈ ?W
    and wf-hyp: ⋀b. (b, a) ∈ r ⟹ (∀ M ∈ ?W. add-mset b M ∈ ?W)
    and acc-hyp: ∀ M. (M, M0) ∈ ?R ⟶ add-mset a M ∈ ?W
  have add-mset a M0 ∈ ?W
  proof (rule accI [of add-mset a M0])
    fix N
    assume (N, add-mset a M0) ∈ ?R
    then consider M where (M, M0) ∈ ?R N = add-mset a M
      | K where ∀ b. b ∈# K ⟶ (b, a) ∈ r N = M0 + K
      by atomize-elim (rule less-add)
    then show N ∈ ?W
    proof cases
      case 1
      from acc-hyp have (M, M0) ∈ ?R ⟶ add-mset a M ∈ ?W ..
      from this and ‹(M, M0) ∈ ?R› have add-mset a M ∈ ?W ..
      then show N ∈ ?W by (simp only: ‹N = add-mset a M›)
    next
      case 2
      from this(1) have M0 + K ∈ ?W
      proof (induct K)
        case empty
        from M0 show M0 + {#} ∈ ?W by simp
      next
        case (add x K)
        from add.prems have (x, a) ∈ r by simp
        with wf-hyp have ∀ M ∈ ?W. add-mset x M ∈ ?W by blast
        moreover from add have M0 + K ∈ ?W by simp
        ultimately have add-mset x (M0 + K) ∈ ?W ..
        then show M0 + (add-mset x K) ∈ ?W by simp
      qed
      then show N ∈ ?W by (simp only: 2(2))
    qed
  qed
} note tedious-reasoning = this

show M ∈ ?W for M
proof (induct M)
  show {#} ∈ ?W
  proof (rule accI)
    fix b assume (b, {#}) ∈ ?R
    with not-less-empty show b ∈ ?W by contradiction
  qed

  fix M a assume M ∈ ?W
  from ‹wf r› have ∀ M ∈ ?W. add-mset a M ∈ ?W
  proof induct
    fix a
```

assume *r*: $\bigwedge b.\ (b,\ a) \in r \Longrightarrow (\forall M \in ?W.\ add\text{-}mset\ b\ M \in ?W)$
show $\forall M \in ?W.\ add\text{-}mset\ a\ M \in ?W$
proof
fix *M* **assume** $M \in ?W$
then show *add-mset a M* $\in ?W$
by (*rule acc-induct*) (*rule tedious-reasoning* [*OF* - *r*])
qed
qed
from *this* **and** ‹$M \in ?W$› **show** *add-mset a M* $\in ?W$ **..**
qed
qed

lemma *wf-mult1*: *wf r* $\Longrightarrow$ *wf* (*mult1 r*)
by (*rule acc-wfI*) (*rule all-accessible*)

lemma *wf-mult*: *wf r* $\Longrightarrow$ *wf* (*mult r*)
unfolding *mult-def* **by** (*rule wf-trancl*) (*rule wf-mult1*)

lemma *wfp-multp*: *wfp r* $\Longrightarrow$ *wfp* (*multp r*)
unfolding *multp-def wfp-def*
by (*simp add*: *wf-mult*)

68.13.2 Closure-free presentation

One direction.

lemma *mult-implies-one-step*:
assumes
trans: *trans r* **and**
MN: $(M,\ N) \in mult\ r$
shows $\exists I\ J\ K.\ N = I + J \wedge M = I + K \wedge J \neq \{\#\} \wedge (\forall k \in set\text{-}mset\ K.\ \exists j \in set\text{-}mset\ J.\ (k,\ j) \in r)$
using *MN* **unfolding** *mult-def mult1-def*
proof (*induction rule*: *converse-trancl-induct*)
case (*base y*)
then show *?case* **by** *force*
next
case (*step y z*) **note** *yz* = *this*(*1*) **and** *zN* = *this*(*2*) **and** *N-decomp* = *this*(*3*)
obtain *I J K* **where**
N: $N = I + J\ z = I + K\ J \neq \{\#\}\ \forall k \in \#K.\ \exists j \in \#J.\ (k,\ j) \in r$
using *N-decomp* **by** *blast*
obtain *a M0 K'* **where**
z: *z* = *add-mset a M0* **and** *y*: *y* = *M0* + *K'* **and** *K*: $\forall b.\ b \in \#\ K' \longrightarrow (b,\ a) \in r$
using *yz* **by** *blast*
show *?case*
proof (*cases a* $\in \#$ *K*)
case *True*
moreover have $\exists j \in \#J.\ (k,\ j) \in r$ **if** $k \in \#\ K'$ **for** *k*
using *K N trans True* **by** (*meson that transE*)

```
    ultimately show ?thesis
      by (rule-tac x = I in exI, rule-tac x = J in exI, rule-tac x = (K - {#a#})
+ K' in exI)
         (use z y N in ‹auto simp del: subset-mset.add-diff-assoc2 dest: in-diffD›)
  next
    case False
    then have a ∈# I by (metis N(2) union-iff union-single-eq-member z)
    moreover have M0 = I + K - {#a#}
      using N(2) z by force
    ultimately show ?thesis
      by (rule-tac x = I - {#a#} in exI, rule-tac x = add-mset a J in exI,
          rule-tac x = K + K' in exI)
        (use z y N False K in ‹auto simp: add.assoc›)
  qed
qed

lemma multp-implies-one-step:
  transp R ⟹ multp R M N ⟹ ∃ I J K. N = I + J ∧ M = I + K ∧ J ≠ {#}
∧ (∀ k∈#K. ∃ x∈#J. R k x)
  by (rule mult-implies-one-step[to-pred])

lemma one-step-implies-mult:
  assumes
    J ≠ {#} and
    ∀ k ∈ set-mset K. ∃ j ∈ set-mset J. (k, j) ∈ r
  shows (I + K, I + J) ∈ mult r
  using assms
proof (induction size J arbitrary: I J K)
  case 0
  then show ?case by auto
next
  case (Suc n) note IH = this(1) and size-J = this(2)[THEN sym]
  obtain J' a where J: J = add-mset a J'
    using size-J by (blast dest: size-eq-Suc-imp-eq-union)
  show ?case
  proof (cases J' = {#})
    case True
    then show ?thesis
      using J Suc by (fastforce simp add: mult-def mult1-def)
  next
    case [simp]: False
    have K: K = {#x ∈# K. (x, a) ∈ r#} + {#x ∈# K. (x, a) ∉ r#}
      by simp
    have (I + K, (I + {# x ∈# K. (x, a) ∈ r #}) + J') ∈ mult r
      using IH[of J' {# x ∈# K. (x, a) ∉ r#} I + {# x ∈# K. (x, a) ∈ r#}]
        J Suc.prems K size-J by (auto simp: ac-simps)
    moreover have (I + {#x ∈# K. (x, a) ∈ r#} + J', I + J) ∈ mult r
      by (fastforce simp: J mult1-def mult-def)
    ultimately show ?thesis
```

unfolding *mult-def* **by** *simp*
qed
qed

lemma *one-step-implies-multp*:
$J \neq \{\#\} \Longrightarrow \forall\, k{\in}\#K.\ \exists\, j{\in}\#J.\ R\ k\ j \Longrightarrow$ *multp* $R\ (I + K)\ (I + J)$
by (*rule one-step-implies-mult*[*of* - - {$(x,\ y).\ r\ x\ y$} **for** r, *folded multp-def*, *simplified*])

lemma *subset-implies-mult*:
assumes *sub*: $A \subset\# B$
shows $(A,\ B) \in$ *mult* r
proof −
have *ApBmA*: $A + (B - A) = B$
using *sub* **by** *simp*
have *BmA*: $B - A \neq \{\#\}$
using *sub* **by** (*simp add*: *Diff-eq-empty-iff-mset subset-mset.less-le-not-le*)
thus *?thesis*
by (*rule one-step-implies-mult*[*of* $B - A$ {#} - A, *unfolded ApBmA*, *simplified*])
qed

lemma *subset-implies-multp*: $A \subset\# B \Longrightarrow$ *multp* $r\ A\ B$
by (*rule subset-implies-mult*[*of* - - {$(x,\ y).\ r\ x\ y$} **for** r, *folded multp-def*])

lemma *multp-repeat-mset-repeat-msetI*:
assumes *transp* R **and** *multp* $R\ A\ B$ **and** $n \neq 0$
shows *multp* R (*repeat-mset* $n\ A$) (*repeat-mset* $n\ \ B$)
proof −
from ‹*transp* R› ‹*multp* $R\ A\ B$› **obtain** $I\ J\ K$ **where**
$B = I + J$ **and** $A = I + K$ **and** $J \neq \{\#\}$ **and** $\forall\, k \in\# K.\ \exists\, x \in\# J.\ R\ k\ x$
by (*auto dest*: *multp-implies-one-step*)

have *repeat-n-A-eq*: *repeat-mset* $n\ A =$ *repeat-mset* $n\ I +$ *repeat-mset* $n\ K$
using ‹$A = I + K$› **by** *simp*

have *repeat-n-B-eq*: *repeat-mset* $n\ B =$ *repeat-mset* $n\ I +$ *repeat-mset* $n\ J$
using ‹$B = I + J$› **by** *simp*

show *?thesis*
unfolding *repeat-n-A-eq repeat-n-B-eq*
proof (*rule one-step-implies-multp*)
from ‹$n \neq 0$› **show** *repeat-mset* $n\ J \neq \{\#\}$
using ‹$J \neq \{\#\}$›
by (*simp add*: *repeat-mset-eq-empty-iff*)
next
show $\forall\, k \in\#$ *repeat-mset* $n\ K.\ \exists\, j \in\#$ *repeat-mset* $n\ J.\ R\ k\ j$
using ‹$\forall\, k \in\# K.\ \exists\, x \in\# J.\ R\ k\ x$›
by (*metis count-greater-zero-iff nat-0-less-mult-iff repeat-mset.rep-eq*)
qed

qed

68.13.3 Monotonicity

lemma *multp-mono-strong*:
assumes *multp R M1 M2* **and** *transp R* **and**
S-if-R: $\bigwedge x\ y.\ x \in \mathit{set\text{-}mset}\ M1 \Longrightarrow y \in \mathit{set\text{-}mset}\ M2 \Longrightarrow R\ x\ y \Longrightarrow S\ x\ y$
shows *multp S M1 M2*
proof −
obtain *I J K* **where** $M2 = I + J$ **and** $M1 = I + K$ **and** $J \neq \{\#\}$ **and** $\forall k{\in}\#K.\ \exists x{\in}\#J.\ R\ k\ x$
using *multp-implies-one-step*[*OF* ‹*transp R*› ‹*multp R M1 M2*›] **by** *auto*
show *?thesis*
unfolding ‹$M2 = I + J$› ‹$M1 = I + K$›
proof (*rule one-step-implies-multp*[*OF* ‹$J \neq \{\#\}$›])
show $\forall k{\in}\#K.\ \exists j{\in}\#J.\ S\ k\ j$
using *S-if-R*
by (*metis* ‹$M1 = I + K$› ‹$M2 = I + J$› ‹$\forall k{\in}\#K.\ \exists x{\in}\#J.\ R\ k\ x$› *union-iff*)
qed
qed

lemma *mult-mono-strong*:
assumes $(M1,\ M2) \in \mathit{mult}\ r$ **and** *trans r* **and**
S-if-R: $\bigwedge x\ y.\ x \in \mathit{set\text{-}mset}\ M1 \Longrightarrow y \in \mathit{set\text{-}mset}\ M2 \Longrightarrow (x,\ y) \in r \Longrightarrow (x,\ y) \in s$
shows $(M1,\ M2) \in \mathit{mult}\ s$
using *assms multp-mono-strong*[*of* $\lambda x\ y.\ (x,\ y) \in r$ *M1 M2* $\lambda x\ y.\ (x,\ y) \in s$, *unfolded multp-def transp-trans-eq*, *simplified*]
by *blast*

lemma *monotone-on-multp-multp-image-mset*:
assumes *monotone-on A orda ordb f* **and** *transp orda*
shows *monotone-on* $\{M.\ \mathit{set\text{-}mset}\ M \subseteq A\}$ (*multp orda*) (*multp ordb*) (*image-mset f*)
proof (*rule monotone-onI*)
fix *M1 M2*
assume
M1-in: $M1 \in \{M.\ \mathit{set\text{-}mset}\ M \subseteq A\}$ **and**
M2-in: $M2 \in \{M.\ \mathit{set\text{-}mset}\ M \subseteq A\}$ **and**
M1-lt-M2: *multp orda M1 M2*

from *multp-implies-one-step*[*OF* ‹*transp orda*› *M1-lt-M2*] **obtain** *I J K* **where**
M2-eq: $M2 = I + J$ **and**
M1-eq: $M1 = I + K$ **and**
J-neq-mempty: $J \neq \{\#\}$ **and**
ball-K-less: $\forall k{\in}\#K.\ \exists x{\in}\#J.\ \mathit{orda}\ k\ x$
by *metis*

have *multp ordb* (*image-mset f I* + *image-mset f K*) (*image-mset f I* + *im-*

```
age-mset f J)
  proof (intro one-step-implies-multp ballI)
    show image-mset f J ≠ {#}
      using J-neq-mempty by simp
  next
    fix k' assume k'∈#image-mset f K
    then obtain k where k' = f k and k-in: k ∈# K
      by auto
    then obtain j where j-in: j∈#J and orda k j
      using ball-K-less by auto

    have ordb (f k) (f j)
    proof (rule ‹monotone-on A orda ordb f›[THEN monotone-onD, OF - - ‹orda
k j›])
      show k ∈ A
        using M1-eq M1-in k-in by auto
    next
      show j ∈ A
        using M2-eq M2-in j-in by auto
    qed
    thus ∃j∈#image-mset f J. ordb k' j
      using ‹j ∈# J› ‹k' = f k› by auto
  qed
  thus multp ordb (image-mset f M1) (image-mset f M2)
    by (simp add: M1-eq M2-eq)
qed

lemma monotone-multp-multp-image-mset:
  assumes monotone orda ordb f and transp orda
  shows monotone (multp orda) (multp ordb) (image-mset f)
  by (rule monotone-on-multp-multp-image-mset[OF assms, simplified])

lemma multp-image-mset-image-msetI:
  assumes multp (λx y. R (f x) (f y)) M1 M2 and transp R
  shows multp R (image-mset f M1) (image-mset f M2)
proof −
  from ‹transp R› have transp (λx y. R (f x) (f y))
    by (auto intro: transpI dest: transpD)
  with ‹multp (λx y. R (f x) (f y)) M1 M2› obtain I J K where
    M2 = I + J and M1 = I + K and J ≠ {#} and ∀k∈#K. ∃x∈#J. R (f k)
(f x)
    using multp-implies-one-step by blast

  have multp R (image-mset f I + image-mset f K) (image-mset f I + image-mset
f J)
  proof (rule one-step-implies-multp)
    show image-mset f J ≠ {#}
      by (simp add: ‹J ≠ {#}›)
  next
```

```
    show ∀ k∈#image-mset f K. ∃ j∈#image-mset f J. R k j
      by (simp add: ‹∀ k∈#K. ∃ x∈#J. R (f k) (f x)›)
  qed
  thus ?thesis
    by (simp add: ‹M1 = I + K› ‹M2 = I + J›)
qed

lemma multp-image-mset-image-msetD:
  assumes
    multp R (image-mset f A) (image-mset f B) and
    transp R and
    inj-on-f: inj-on f (set-mset A ∪ set-mset B)
  shows multp (λx y. R (f x) (f y)) A B
proof −
  from assms(1,2) obtain I J K where
    f-B-eq: image-mset f B = I + J and
    f-A-eq: image-mset f A = I + K and
    J-neq-mempty: J ≠ {#} and
    ball-K-less: ∀ k∈#K. ∃ x∈#J. R k x
    by (auto dest: multp-implies-one-step)

  from f-B-eq obtain I' J' where
   B-def: B = I' + J' and I-def: I = image-mset f I' and J-def: J = image-mset
f J'
    using image-mset-eq-plusD by blast

  from inj-on-f have inj-on-f': inj-on f (set-mset A ∪ set-mset I')
    by (rule inj-on-subset) (auto simp add: B-def)

  from f-A-eq obtain K' where
    A-def: A = I' + K' and K-def: K = image-mset f K'
    by (auto simp: I-def dest: image-mset-eq-image-mset-plusD[OF - inj-on-f'])

  show ?thesis
    unfolding A-def B-def
  proof (intro one-step-implies-multp ballI)
    from J-neq-mempty show J' ≠ {#}
      by (simp add: J-def)
  next
    fix k assume k ∈# K'
    with ball-K-less obtain j' where j' ∈# J and R (f k) j'
      using K-def by auto
    moreover then obtain j where j ∈# J' and f j = j'
      using J-def by auto
    ultimately show ∃ j∈#J'. R (f k) (f j)
      by blast
  qed
qed
```

68.13.4 The multiset extension is cancellative for multiset union

lemma *mult-cancel*:
assumes *trans s* **and** *irrefl-on* (*set-mset Z*) *s*
shows (*X* + *Z*, *Y* + *Z*) ∈ *mult s* ⟷ (*X*, *Y*) ∈ *mult s* (**is** *?L* ⟷ *?R*)
proof
assume *?L* **thus** *?R*
using ‹*irrefl-on* (*set-mset Z*) *s*›
proof (*induct Z*)
case (*add z Z*)
obtain *X′ Y′ Z′* **where** ∗: *add-mset z X* + *Z* = *Z′* + *X′ add-mset z Y* + *Z* = *Z′* + *Y′ Y′* ≠ {#}
∀ *x* ∈ *set-mset X′*. ∃ *y* ∈ *set-mset Y′*. (*x*, *y*) ∈ *s*
using *mult-implies-one-step*[*OF* ‹*trans s*› *add*(*2*)] **by** *auto*
consider *Z2* **where** *Z′* = *add-mset z Z2* | *X2 Y2* **where** *X′* = *add-mset z X2* *Y′* = *add-mset z Y2*
using ∗(*1*,*2*) **by** (*metis add-mset-remove-trivial-If insert-iff set-mset-add-mset-insert union-iff*)
thus *?case*
proof (*cases*)
case *1* **thus** *?thesis*
using ∗ *one-step-implies-mult*[*of Y′ X′ s Z2*] *add*(*3*)
by (*auto simp*: *add.commute*[*of* - {#-#}] *add.assoc intro*: *add*(*1*) *elim*: *irrefl-on-subset*)
next
case *2* **then obtain** *y* **where** *y* ∈ *set-mset Y2* (*z*, *y*) ∈ *s*
using ∗(*4*) ‹*irrefl-on* (*set-mset* (*add-mset z Z*)) *s*›
by (*auto simp*: *irrefl-on-def*)
moreover from *this transD*[*OF* ‹*trans s*› - *this*(*2*)]
have *x′* ∈ *set-mset X2* ⟹ ∃ *y* ∈ *set-mset Y2*. (*x′*, *y*) ∈ *s* **for** *x′*
using *2* ∗(*4*)[*rule-format*, *of x′*] **by** *auto*
ultimately show *?thesis*
using ∗ *one-step-implies-mult*[*of Y2 X2 s Z′*] *2 add*(*3*)
by (*force simp*: *add.commute*[*of* {#-#}] *add.assoc*[*symmetric*] *intro*: *add*(*1*) *elim*: *irrefl-on-subset*)
qed
qed *auto*
next
assume *?R* **then obtain** *I J K*
where *Y* = *I* + *J X* = *I* + *K J* ≠ {#} ∀ *k* ∈ *set-mset K*. ∃ *j* ∈ *set-mset J*. (*k*, *j*) ∈ *s*
using *mult-implies-one-step*[*OF* ‹*trans s*›] **by** *blast*
thus *?L* **using** *one-step-implies-mult*[*of J K s I* + *Z*] **by** (*auto simp*: *ac-simps*)
qed

lemma *multp-cancel*:
transp R ⟹ *irreflp-on* (*set-mset Z*) *R* ⟹ *multp R* (*X* + *Z*) (*Y* + *Z*) ⟷ *multp R X Y*
by (*rule mult-cancel*[*to-pred*])

lemma *mult-cancel-add-mset*:
trans r $\Longrightarrow$ *irefl-on* $\{z\}$ *r* $\Longrightarrow$
$((add\text{-}mset\ z\ X,\ add\text{-}mset\ z\ Y) \in mult\ r) = ((X,\ Y) \in mult\ r)$
by (*rule mult-cancel*[*of* - $\{\#\text{-}\#\}$, *simplified*])

lemma *multp-cancel-add-mset*:
transp R $\Longrightarrow$ *irreflp-on* $\{z\}$ *R* $\Longrightarrow$ *multp R* (*add-mset z X*) (*add-mset z Y*) = *multp R X Y*
by (*rule mult-cancel-add-mset*[*to-pred*, *folded bot-set-def*])

lemma *mult-cancel-max0*:
assumes *trans s* **and** *irefl-on* (*set-mset X* $\cap$ *set-mset Y*) *s*
shows $(X,\ Y) \in mult\ s \longleftrightarrow (X - X \cap\# Y,\ Y - X \cap\# Y) \in mult\ s$ (**is** *?L* $\longleftrightarrow$ *?R*)
proof −
have $(X - X \cap\# Y + X \cap\# Y,\ Y - X \cap\# Y + X \cap\# Y) \in mult\ s \longleftrightarrow (X - X \cap\# Y,\ Y - X \cap\# Y) \in mult\ s$
proof (*rule mult-cancel*)
from *assms* **show** *trans s*
by *simp*
next
from *assms* **show** *irefl-on* (*set-mset* $(X \cap\# Y)$) *s*
by *simp*
qed
moreover have $X - X \cap\# Y + X \cap\# Y = X\ Y - X \cap\# Y + X \cap\# Y = Y$
by (*auto simp flip*: *count-inject*)
ultimately show *?thesis*
by *simp*
qed

lemma *mult-cancel-max*:
trans r $\Longrightarrow$ *irefl-on* (*set-mset X* $\cap$ *set-mset Y*) *r* $\Longrightarrow$
$(X,\ Y) \in mult\ r \longleftrightarrow (X - Y,\ Y - X) \in mult\ r$
by (*rule mult-cancel-max0*[*simplified*])

lemma *multp-cancel-max*:
transp R $\Longrightarrow$ *irreflp-on* (*set-mset X* $\cap$ *set-mset Y*) *R* $\Longrightarrow$ *multp R X Y* $\longleftrightarrow$ *multp R* $(X - Y)$ $(Y - X)$
by (*rule mult-cancel-max*[*to-pred*])

68.13.5 Strict partial-order properties

lemma *mult1-lessE*:
assumes $(N,\ M) \in mult1\ \{(a,\ b).\ r\ a\ b\}$ **and** *asymp r*
obtains *a M0 K* **where** $M = add\text{-}mset\ a\ M0\ N = M0 + K$
$a \notin\# K \bigwedge b.\ b \in\# K \Longrightarrow r\ b\ a$
proof −
from *assms* **obtain** *a M0 K* **where** $M = add\text{-}mset\ a\ M0\ N = M0 + K$ **and**
$*$: $b \in\# K \Longrightarrow r\ b\ a$ **for** *b* **by** (*blast elim*: *mult1E*)

```
  moreover from ∗ [of a] have a ∉# K
    using ‹asymp r› by (meson asympD)
  ultimately show thesis by (auto intro: that)
qed

lemma trans-on-mult:
  assumes trans-on A r and ⋀M. M ∈ B ⟹ set-mset M ⊆ A
  shows trans-on B (mult r)
  using assms by (metis mult-def subset-UNIV trans-on-subset trans-trancl)

lemma trans-mult: trans r ⟹ trans (mult r)
  using trans-on-mult[of UNIV r UNIV, simplified] .

lemma transp-on-multp:
  assumes transp-on A r and ⋀M. M ∈ B ⟹ set-mset M ⊆ A
  shows transp-on B (multp r)
  by (metis mult-def multp-def transD trans-trancl transp-onI)

lemma transp-multp: transp r ⟹ transp (multp r)
  using transp-on-multp[of UNIV r UNIV, simplified] .

lemma irrefl-mult:
  assumes trans r irrefl r
  shows irrefl (mult r)
proof (intro irreflI notI)
  fix M
  assume (M, M) ∈ mult r
  then obtain I J K where M = I + J and M = I + K
    and J ≠ {#} and (∀ k∈set-mset K. ∃ j∈set-mset J. (k, j) ∈ r)
    using mult-implies-one-step[OF ‹trans r›] by blast
  then have ∗: K ≠ {#} and ∗∗: ∀ k∈set-mset K. ∃ j∈set-mset K. (k, j) ∈ r by
auto
  have finite (set-mset K) by simp
  hence set-mset K = {}
    using ∗∗
  proof (induction rule: finite-induct)
    case empty
    thus ?case by simp
  next
    case (insert x F)
    have False
      using ‹irrefl r›[unfolded irrefl-def, rule-format]
      using ‹trans r›[THEN transD]
      by (metis equals0D insert.IH insert.prems insertE insertI1 insertI2)
    thus ?case ..
  qed
  with ∗ show False by simp
qed
```

lemma *irreflp-multp*: *transp R* $\Longrightarrow$ *irreflp R* $\Longrightarrow$ *irreflp* (*multp R*)
by (*rule irrefl-mult*[*of* {(*x*, *y*). *r x y*} **for** *r*,
folded transp-trans-eq irreflp-irrefl-eq, *simplified*, *folded multp-def*])

instantiation *multiset* :: (*preorder*) *order* **begin**

definition *less-multiset* :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool*
where *M* $<$ *N* $\longleftrightarrow$ *multp* (*<*) *M N*

definition *less-eq-multiset* :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool*
where *less-eq-multiset M N* $\longleftrightarrow$ *M* $<$ *N* $\vee$ *M* $=$ *N*

instance
proof *intro-classes*
fix *M N* :: *'a multiset*
show (*M* $<$ *N*) $=$ (*M* $\leq$ *N* $\wedge$ $\neg$ *N* $\leq$ *M*)
unfolding *less-eq-multiset-def less-multiset-def*
by (*metis irreflp-def irreflp-on-less irreflp-multp transpE transp-on-less transp-multp*)
next
fix *M* :: *'a multiset*
show *M* $\leq$ *M*
unfolding *less-eq-multiset-def*
by *simp*
next
fix *M1 M2 M3* :: *'a multiset*
show *M1* $\leq$ *M2* $\Longrightarrow$ *M2* $\leq$ *M3* $\Longrightarrow$ *M1* $\leq$ *M3*
unfolding *less-eq-multiset-def less-multiset-def*
using *transp-multp*[*OF transp-on-less*, *THEN transpD*]
by *blast*
next
fix *M N* :: *'a multiset*
show *M* $\leq$ *N* $\Longrightarrow$ *N* $\leq$ *M* $\Longrightarrow$ *M* $=$ *N*
unfolding *less-eq-multiset-def less-multiset-def*
using *transp-multp*[*OF transp-on-less*, *THEN transpD*]
using *irreflp-multp*[*OF transp-on-less irreflp-on-less*, *unfolded irreflp-def*, *rule-format*]
by *blast*
qed

end

lemma *mset-le-irrefl* [*elim*!]:
fixes *M* :: *'a*::*preorder multiset*
shows *M* $<$ *M* $\Longrightarrow$ *R*
by *simp*

lemma *wfp-less-multiset*[*simp*]:
assumes *wf*: *wfp* ((*<*) :: (*'a* :: *preorder*) $\Rightarrow$ *'a* $\Rightarrow$ *bool*)
shows *wfp* ((*<*) :: *'a multiset* $\Rightarrow$ *'a multiset* $\Rightarrow$ *bool*)
unfolding *less-multiset-def*

using *wfp-multp*[*OF wf*] **.**

68.13.6 Strict total-order properties

lemma *total-on-mult*:
assumes *total-on A r* **and** *trans r* **and** $\bigwedge M.\ M \in B \Longrightarrow$ *set-mset M* $\subseteq A$
shows *total-on B* (*mult r*)
proof (*rule total-onI*)
fix *M1 M2* **assume** *M1* $\in B$ **and** *M2* $\in B$ **and** *M1* $\neq$ *M2*
let *?I* = *M1* $\cap\#$ *M2*
show (*M1*, *M2*) $\in$ *mult r* $\vee$ (*M2*, *M1*) $\in$ *mult r*
proof (*cases M1* $-$ *?I* = {#} $\vee$ *M2* $-$ *?I* = {#})
case *True*
with ‹*M1* $\neq$ *M2*› **show** *?thesis*
by (*metis Diff-eq-empty-iff-mset diff-intersect-left-idem diff-intersect-right-idem subset-implies-mult subset-mset.less-le*)
next
case *False*
from *assms*(*1*) **have** *total-on* (*set-mset* (*M1* $-$ *?I*)) *r*
by (*meson* ‹*M1* $\in B$› *assms*(*3*) *diff-subset-eq-self set-mset-mono total-on-subset*)
with *False* **obtain** *greatest1* **where**
greatest1-in: *greatest1* $\in\#$ *M1* $-$ *?I* **and**
greatest1-greatest: $\forall x \in\#$ *M1* $-$ *?I*. *greatest1* $\neq x \longrightarrow$ (*x*, *greatest1*) $\in r$
using *Multiset.bex-greatest-element*[*to-set*, *of M1* $-$ *?I r*]
by (*metis* ‹*trans r*› *transD trans-onI*)

from *assms*(*1*) **have** *total-on* (*set-mset* (*M2* $-$ *?I*)) *r*
by (*meson* ‹*M2* $\in B$› *assms*(*3*) *diff-subset-eq-self set-mset-mono total-on-subset*)
with *False* **obtain** *greatest2* **where**
greatest2-in: *greatest2* $\in\#$ *M2* $-$ *?I* **and**
greatest2-greatest: $\forall x \in\#$ *M2* $-$ *?I*. *greatest2* $\neq x \longrightarrow$ (*x*, *greatest2*) $\in r$
using *Multiset.bex-greatest-element*[*to-set*, *of M2* $-$ *?I r*]
by (*metis* ‹*trans r*› *top-greatest trans-on-subset*)

have *greatest1* $\neq$ *greatest2*
using *greatest1-in* ‹*greatest2* $\in\#$ *M2* $-$ *?I*›
by (*metis diff-intersect-left-idem diff-intersect-right-idem dual-order.eq-iff in-diff-count*
in-diff-countE le-add-same-cancel2 less-irrefl zero-le)
hence (*greatest1*, *greatest2*) $\in r \vee$ (*greatest2*, *greatest1*) $\in r$
using ‹*total-on A r*›[*unfolded total-on-def*, *rule-format*, *of greatest1 greatest2*]
‹*M1* $\in B$› ‹*M2* $\in B$› *greatest1-in greatest2-in assms*(*3*)
by (*meson in-diffD in-mono*)
thus *?thesis*
proof (*elim disjE*)
assume (*greatest1*, *greatest2*) $\in r$
have (*?I* + (*M1* $-$ *?I*), *?I* + (*M2* $-$ *?I*)) $\in$ *mult r*
proof (*rule one-step-implies-mult*[*of M2* $-$ *?I M1* $-$ *?I r ?I*])
show *M2* $-$ *?I* $\neq$ {#}

```
      using False by force
    next
      show ∀k∈#M1 − ?I. ∃j∈#M2 − ?I. (k, j) ∈ r
        using ‹(greatest1, greatest2) ∈ r› greatest2-in greatest1-greatest
        by (metis ‹trans r› transD)
    qed
    hence (M1, M2) ∈ mult r
      by (metis subset-mset.add-diff-inverse subset-mset.inf.cobounded1
          subset-mset.inf.cobounded2)
    thus (M1, M2) ∈ mult r ∨ (M2, M1) ∈ mult r ..
  next
    assume (greatest2, greatest1) ∈ r
    have (?I + (M2 − ?I), ?I + (M1 − ?I)) ∈ mult r
    proof (rule one-step-implies-mult[of M1 − ?I M2 − ?I r ?I])
      show M1 − M1 ∩# M2 ≠ {#}
        using False by force
    next
      show ∀k∈#M2 − ?I. ∃j∈#M1 − ?I. (k, j) ∈ r
        using ‹(greatest2, greatest1) ∈ r› greatest1-in greatest2-greatest
        by (metis assms(2) transD)
    qed
    hence (M2, M1) ∈ mult r
      by (metis subset-mset.add-diff-inverse subset-mset.inf.cobounded1
          subset-mset.inf.cobounded2)
    thus (M1, M2) ∈ mult r ∨ (M2, M1) ∈ mult r ..
  qed
 qed
qed

lemma total-mult: total r ⟹ trans r ⟹ total (mult r)
 by (rule total-on-mult[of UNIV r UNIV, simplified])

lemma totalp-on-multp:
 totalp-on A R ⟹ transp R ⟹ (⋀M. M ∈ B ⟹ set-mset M ⊆ A) ⟹ totalp-on
B (multp R)
 using total-on-mult[of A {(x,y). R x y} B, to-pred]
 by (simp add: multp-def total-on-def totalp-on-def)

lemma totalp-multp: totalp R ⟹ transp R ⟹ totalp (multp R)
 by (rule totalp-on-multp[of UNIV R UNIV, simplified])
```

68.14 Quasi-executable version of the multiset extension

Predicate variants of *mult* and the reflexive closure of *mult*, which are executable whenever the given predicate P is. Together with the standard code equations for $(\cap\#)$ and $(-)$ this should yield quadratic (with respect to calls to P) implementations of *multp-code* and *multeqp-code*.

```
definition multp-code :: ('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool
where
```

multp-code P N M =
(let Z = M ∩# N; X = M − Z in
X ≠ {#} ∧ (let Y = N − Z in (∀ y ∈ set-mset Y. ∃ x ∈ set-mset X. P y x)))

definition *multeqp-code* :: *('a ⇒ 'a ⇒ bool) ⇒ 'a multiset ⇒ 'a multiset ⇒ bool*
where
multeqp-code P N M =
(let Z = M ∩# N; X = M − Z; Y = N − Z in
(∀ y ∈ set-mset Y. ∃ x ∈ set-mset X. P y x))

lemma *multp-code-iff-mult*:
assumes *irrefl-on (set-mset N ∩ set-mset M) R* **and** *trans R* **and**
[simp]: *⋀x y. P x y ⟷ (x, y) ∈ R*
shows *multp-code P N M ⟷ (N, M) ∈ mult R* (**is** *?L ⟷ ?R*)
proof −
have ∗: *M ∩# N + (N − M ∩# N) = N M ∩# N + (M − M ∩# N) = M*
(M − M ∩# N) ∩# (N − M ∩# N) = {#} **by** (*auto simp flip*: *count-inject*)
show *?thesis*
proof
assume *?L* **thus** *?R*
using *one-step-implies-mult*[*of M − M ∩# N N − M ∩# N R M ∩# N*] ∗
by (*auto simp*: *multp-code-def Let-def*)
next
have [*dest*!]: *I = {#}* **if** *(I + J) ∩# (I + K) = {#}* **for** *I J K*
using *that* **by** (*metis inter-union-distrib-right union-eq-empty*)
assume *?R* **thus** *?L*
using *mult-cancel-max*
using *mult-implies-one-step*[*OF assms*(*2*), *of N − M ∩# N M − M ∩# N*]
mult-cancel-max[*OF assms*(*2*,*1*)] ∗ **by** (*auto simp*: *multp-code-def*)
qed
qed

lemma *multp-code-iff-multp*:
irreflp-on (set-mset M ∩ set-mset N) R ⟹ transp R ⟹ multp-code R M N
⟷ multp R M N
using *multp-code-iff-mult*[*simplified*, *to-pred*, *of M N R R*] **by** *simp*

lemma *multp-code-eq-multp*:
assumes *irreflp R* **and** *transp R*
shows *multp-code R = multp R*
proof (*intro ext*)
fix *M N*
show *multp-code R M N = multp R M N*
proof (*rule multp-code-iff-multp*)
from *assms* **show** *irreflp-on (set-mset M ∩ set-mset N) R*
by (*auto intro*: *irreflp-on-subset*)
next
from *assms* **show** *transp R*
by *simp*

qed
qed

lemma *multeqp-code-iff-reflcl-mult*:
assumes *irrefl-on* (*set-mset N* ∩ *set-mset M*) *R* **and** *trans R* **and** ⋀*x y*. *P x y* ⟷ (*x*, *y*) ∈ *R*
shows *multeqp-code P N M* ⟷ (*N*, *M*) ∈ (*mult R*)=
proof −
have ∃ *y*. *count M y* < *count N y* **if** *N* ≠ *M M* − *M* ∩# *N* = {#}
proof −
from *that* **obtain** *y* **where** *count N y* ≠ *count M y*
by (*auto simp flip*: *count-inject*)
then show *?thesis*
using ‹*M* − *M* ∩# *N* = {#}›
by (*auto simp flip*: *count-inject dest*!: *le-neq-implies-less fun-cong*[*of* - - *y*])
qed
then have *multeqp-code P N M* ⟷ *multp-code P N M* ∨ *N* = *M*
by (*auto simp*: *multeqp-code-def multp-code-def Let-def in-diff-count*)
thus *?thesis*
using *multp-code-iff-mult*[*OF assms*] **by** *simp*
qed

lemma *multeqp-code-iff-reflclp-multp*:
irreflp-on (*set-mset M* ∩ *set-mset N*) *R* ⟹ *transp R* ⟹ *multeqp-code R M N* ⟷ (*multp R*)== *M N*
using *multeqp-code-iff-reflcl-mult*[*simplified*, *to-pred*, *of M N R R*] **by** *simp*

lemma *multeqp-code-eq-reflclp-multp*:
assumes *irreflp R* **and** *transp R*
shows *multeqp-code R* = (*multp R*)==
proof (*intro ext*)
fix *M N*
show *multeqp-code R M N* ⟷ (*multp R*)== *M N*
proof (*rule multeqp-code-iff-reflclp-multp*)
from *assms* **show** *irreflp-on* (*set-mset M* ∩ *set-mset N*) *R*
by (*auto intro*: *irreflp-on-subset*)
next
from *assms* **show** *transp R*
by *simp*
qed
qed

68.14.1 Monotonicity of multiset union

lemma *mult1-union*: (*B*, *D*) ∈ *mult1 r* ⟹ (*C* + *B*, *C* + *D*) ∈ *mult1 r*
by (*force simp*: *mult1-def*)

lemma *union-le-mono2*: *B* < *D* ⟹ *C* + *B* < *C* + (*D*::'*a*::*preorder multiset*)
unfolding *less-multiset-def multp-def mult-def*

by (*induction rule*: *trancl-induct*; *blast intro*: *mult1-union trancl-trans*)

lemma *union-le-mono1*: $B < D \Longrightarrow B + C < D + (C::'a::preorder\ multiset)$
by (*metis add.commute union-le-mono2*)

lemma *union-less-mono*:
fixes *A B C D* :: $'a::preorder\ multiset$
shows $A < C \Longrightarrow B < D \Longrightarrow A + B < C + D$
by (*blast intro*!: *union-le-mono1 union-le-mono2 less-trans*)

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-add*
begin
instance
by *standard* (*auto simp add*: *less-eq-multiset-def intro*: *union-le-mono2*)
end

68.14.2 Termination proofs with multiset orders

lemma *multi-member-skip*: $x \in\# XS \Longrightarrow x \in\# \{\#\ y\ \#\} + XS$
and *multi-member-this*: $x \in\# \{\#\ x\ \#\} + XS$
and *multi-member-last*: $x \in\# \{\#\ x\ \#\}$
by *auto*

definition *ms-strict* = *mult pair-less*
definition *ms-weak* = *ms-strict* $\cup$ *Id*

lemma *ms-reduction-pair*: *reduction-pair* (*ms-strict*, *ms-weak*)
unfolding *reduction-pair-def ms-strict-def ms-weak-def pair-less-def*
by (*auto intro*: *wf-mult1 wf-trancl simp*: *mult-def*)

lemma *smsI*:
(*set-mset A*, *set-mset B*) $\in$ *max-strict* $\Longrightarrow$ (*Z* + *A*, *Z* + *B*) $\in$ *ms-strict*
unfolding *ms-strict-def*
by (*rule one-step-implies-mult*) (*auto simp add*: *max-strict-def pair-less-def elim*!:*max-ext.cases*)

lemma *wmsI*:
(*set-mset A*, *set-mset B*) $\in$ *max-strict* $\vee$ $A = \{\#\} \wedge B = \{\#\}$
$\Longrightarrow$ (*Z* + *A*, *Z* + *B*) $\in$ *ms-weak*
unfolding *ms-weak-def ms-strict-def*
by (*auto simp add*: *pair-less-def max-strict-def elim*!:*max-ext.cases intro*: *one-step-implies-mult*)

inductive *pw-leq*
where
pw-leq-empty: *pw-leq* {#} {#}
| *pw-leq-step*: ⟦(*x*,*y*) ∈ *pair-leq*; *pw-leq X Y* ⟧ ⟹ *pw-leq* ({#*x*#} + *X*) ({#*y*#}
+ *Y*)

lemma *pw-leq-lstep*:
(*x*, *y*) $\in$ *pair-leq* $\Longrightarrow$ *pw-leq* {#*x*#} {#*y*#}

by (*drule pw-leq-step*) (*rule pw-leq-empty*, *simp*)

lemma *pw-leq-split*:
assumes *pw-leq X Y*
shows $\exists A\ B\ Z.\ X = A + Z \wedge Y = B + Z \wedge ((\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
using *assms*
proof *induct*
case *pw-leq-empty* **thus** *?case* **by** *auto*
next
case (*pw-leq-step x y X Y*)
then obtain *A B Z* **where**
[*simp*]: $X = A + Z\ Y = B + Z$
and *1*[*simp*]: $(\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\})$
by *auto*
from *pw-leq-step* **consider** $x = y \mid (x, y) \in \textit{pair-less}$
unfolding *pair-leq-def* **by** *auto*
thus *?case*
proof *cases*
case [*simp*]: *1*
have $\{\#x\#\} + X = A + (\{\#y\#\}+Z) \wedge \{\#y\#\} + Y = B + (\{\#y\#\}+Z) \wedge$
$((\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \vee (B = \{\#\} \wedge A = \{\#\}))$
by *auto*
thus *?thesis* **by** *blast*
next
case *2*
let $?A' = \{\#x\#\} + A$ **and** $?B' = \{\#y\#\} + B$
have $\{\#x\#\} + X = ?A' + Z$
$\{\#y\#\} + Y = ?B' + Z$
by *auto*
moreover have
$(\textit{set-mset}\ ?A', \textit{set-mset}\ ?B') \in \textit{max-strict}$
using *1 2* **unfolding** *max-strict-def*
by (*auto elim*!: *max-ext.cases*)
ultimately show *?thesis* **by** *blast*
qed
qed

lemma
assumes *pwleq*: *pw-leq Z Z'*
shows *ms-strictI*: $(\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B) \in \textit{ms-strict}$
and *ms-weakI1*: $(\textit{set-mset}\ A, \textit{set-mset}\ B) \in \textit{max-strict} \Longrightarrow (Z + A, Z' + B) \in \textit{ms-weak}$
and *ms-weakI2*: $(Z + \{\#\}, Z' + \{\#\}) \in \textit{ms-weak}$
proof −
from *pw-leq-split*[*OF pwleq*]
obtain *A' B' Z''*
where [*simp*]: $Z = A' + Z''\ Z' = B' + Z''$

```
    and mx-or-empty: (set-mset A', set-mset B') ∈ max-strict ∨ (A' = {#} ∧ B'
= {#})
    by blast
  {
    assume max: (set-mset A, set-mset B) ∈ max-strict
    from mx-or-empty
    have (Z'' + (A + A'), Z'' + (B + B')) ∈ ms-strict
    proof
      assume max': (set-mset A', set-mset B') ∈ max-strict
      with max have (set-mset (A + A'), set-mset (B + B')) ∈ max-strict
        by (auto simp: max-strict-def intro: max-ext-additive)
      thus ?thesis by (rule smsI)
    next
      assume [simp]: A' = {#} ∧ B' = {#}
      show ?thesis by (rule smsI) (auto intro: max)
    qed
    thus (Z + A, Z' + B) ∈ ms-strict by (simp add: ac-simps)
    thus (Z + A, Z' + B) ∈ ms-weak by (simp add: ms-weak-def)
  }
  from mx-or-empty
  have (Z'' + A', Z'' + B') ∈ ms-weak by (rule wmsI)
  thus (Z + {#}, Z' + {#}) ∈ ms-weak by (simp add: ac-simps)
qed

lemma empty-neutral: {#} + x = x x + {#} = x
and nonempty-plus: {# x #} + rs ≠ {#}
and nonempty-single: {# x #} ≠ {#}
by auto

setup ‹
  let
    fun msetT T = Type ‹multiset T›;

    fun mk-mset T [] = instantiate ‹'a = T in term ‹{#}››
      | mk-mset T [x] = instantiate ‹'a = T and x in term ‹{#x#}››
      | mk-mset T (x :: xs) = Const ‹plus ‹msetT T› for ‹mk-mset T [x]› ‹mk-mset
T xs››

    fun mset-member-tac ctxt m i =
      if m <= 0 then
        resolve-tac ctxt @{thms multi-member-this} i ORELSE
        resolve-tac ctxt @{thms multi-member-last} i
      else
        resolve-tac ctxt @{thms multi-member-skip} i THEN mset-member-tac ctxt
(m - 1) i

    fun mset-nonempty-tac ctxt =
      resolve-tac ctxt @{thms nonempty-plus} ORELSE'
      resolve-tac ctxt @{thms nonempty-single}
```

```
  fun regroup-munion-conv ctxt =
   Function-Lib.regroup-conv ctxt const-abbrev ‹empty-mset› const-name ‹plus›
    (map (fn t => t RS eq-reflection) (@{thms ac-simps} @ @{thms empty-neutral}))

  fun unfold-pwleq-tac ctxt i =
    (resolve-tac ctxt @{thms pw-leq-step} i THEN (fn st => unfold-pwleq-tac ctxt
(i + 1) st))
      ORELSE (resolve-tac ctxt @{thms pw-leq-lstep} i)
      ORELSE (resolve-tac ctxt @{thms pw-leq-empty} i)

  val set-mset-simps = [@{thm set-mset-empty}, @{thm set-mset-single}, @{thm
set-mset-union},
                        @{thm Un-insert-left}, @{thm Un-empty-left}]
 in
  ScnpReconstruct.multiset-setup (ScnpReconstruct.Multiset
  {
   msetT=msetT, mk-mset=mk-mset, mset-regroup-conv=regroup-munion-conv,
   mset-member-tac=mset-member-tac, mset-nonempty-tac=mset-nonempty-tac,
   mset-pwleq-tac=unfold-pwleq-tac, set-of-simps=set-mset-simps,
    smsI'= @{thm ms-strictI}, wmsI2''= @{thm ms-weakI2}, wmsI1= @{thm
ms-weakI1},
   reduction-pair = @{thm ms-reduction-pair}
  })
 end
›
```

68.15 Legacy theorem bindings

lemmas *multi-count-eq* = *multiset-eq-iff* [*symmetric*]

lemma *union-commute*: $M + N = N + (M::'a\ multiset)$
  by (*fact add.commute*)

lemma *union-assoc*: $(M + N) + K = M + (N + (K::'a\ multiset))$
  by (*fact add.assoc*)

lemma *union-lcomm*: $M + (N + K) = N + (M + (K::'a\ multiset))$
  by (*fact add.left-commute*)

lemmas *union-ac* = *union-assoc union-commute union-lcomm add-mset-commute*

lemma *union-right-cancel*: $M + K = N + K \longleftrightarrow M = (N::'a\ multiset)$
  by (*fact add-right-cancel*)

lemma *union-left-cancel*: $K + M = K + N \longleftrightarrow M = (N::'a\ multiset)$
  by (*fact add-left-cancel*)

lemma *multi-union-self-other-eq*: $(A::'a\ multiset) + X = A + Y \Longrightarrow X = Y$

by (*fact add-left-imp-eq*)

lemma *mset-subset-trans*: $(M::'a\ multiset) \subset\# K \Longrightarrow K \subset\# N \Longrightarrow M \subset\# N$
by (*fact subset-mset.less-trans*)

lemma *multiset-inter-commute*: $A \cap\# B = B \cap\# A$
by (*fact subset-mset.inf.commute*)

lemma *multiset-inter-assoc*: $A \cap\# (B \cap\# C) = A \cap\# B \cap\# C$
by (*fact subset-mset.inf.assoc* [*symmetric*])

lemma *multiset-inter-left-commute*: $A \cap\# (B \cap\# C) = B \cap\# (A \cap\# C)$
by (*fact subset-mset.inf.left-commute*)

lemmas *multiset-inter-ac* =
multiset-inter-commute
multiset-inter-assoc
multiset-inter-left-commute

lemma *mset-le-not-refl*: $\neg M < (M::'a::preorder\ multiset)$
by (*fact less-irrefl*)

lemma *mset-le-trans*: $K < M \Longrightarrow M < N \Longrightarrow K < (N::'a::preorder\ multiset)$
by (*fact less-trans*)

lemma *mset-le-not-sym*: $M < N \Longrightarrow \neg N < (M::'a::preorder\ multiset)$
by (*fact less-not-sym*)

lemma *mset-le-asym*: $M < N \Longrightarrow (\neg P \Longrightarrow N < (M::'a::preorder\ multiset)) \Longrightarrow P$
by (*fact less-asym*)

declaration ‹
let
fun multiset-postproc - maybe-name all-values (*T as Type* (-, [*elem-T*])) (*Const* - $ *t'*) =
let
val (*maybe-opt*, *ps*) =
Nitpick-Model.dest-plain-fun t'
||> (~~)
||> *map* (*apsnd* (*snd o HOLogic.dest-number*))
fun elems-for t =
(*case AList.lookup* (=) *ps t of*
SOME n => *replicate n t*
| *NONE* => [*Const* (*maybe-name*, *elem-T* --> *elem-T*) $ *t*])
in
(*case maps elems-for* (*all-values elem-T*) @
(*if maybe-opt then* [*Const* (*Nitpick-Model.unrep-mixfix* (), *elem-T*)]
else []) *of*

```
          [] => Const ‹Groups.zero T›
        | ts => foldl1 (fn (s, t) => Const ‹add-mset elem-T for s t›) ts)
      end
  | multiset-postproc - - - - t = t
 in Nitpick-Model.register-term-postprocessor typ ‹'a multiset› multiset-postproc
end
›
```

68.16 Naive implementation using lists

code-datatype *mset*

lemma [*code*]: {#} = *mset* []
by *simp*

lemma [*code*]: *add-mset x* (*mset xs*) = *mset* (*x* # *xs*)
by *simp*

lemma [*code*]: *Multiset.is-empty* (*mset xs*) ⟷ *List.null xs*
by (*simp add*: *Multiset.is-empty-def*)

lemma *union-code* [*code*]: *mset xs* + *mset ys* = *mset* (*xs* @ *ys*)
by *simp*

lemma [*code*]: *image-mset f* (*mset xs*) = *mset* (*map f xs*)
by *simp*

lemma [*code*]: *filter-mset f* (*mset xs*) = *mset* (*filter f xs*)
by *simp*

lemma [*code*]: *mset xs* − *mset ys* = *mset* (*minus-list-mset xs ys*)
by *simp*

lemma [*code*]:
mset xs ∩# *mset ys* =
mset (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, []))
proof −
have ⋀*zs*. *mset* (*snd* (*fold* (λ*x* (*ys*, *zs*).
if x ∈ *set ys then* (*remove1 x ys*, *x* # *zs*) *else* (*ys*, *zs*)) *xs* (*ys*, *zs*))) =
(*mset xs* ∩# *mset ys*) + *mset zs*
by (*induct xs arbitrary*: *ys*)
(*auto simp add*: *inter-add-right1 inter-add-right2 ac-simps*)
then show *?thesis* **by** *simp*
qed

lemma [*code*]:
mset xs ∪# *mset ys* =
mset (*case-prod append* (*fold* (λ*x* (*ys*, *zs*). (*remove1 x ys*, *x* # *zs*)) *xs* (*ys*, [])))

```
proof −
  have ⋀zs. mset (case-prod append (fold (λx (ys, zs). (remove1 x ys, x # zs)) xs
(ys, zs))) =
      (mset xs ∪# mset ys) + mset zs
    by (induct xs arbitrary: ys) (simp-all add: multiset-eq-iff)
  then show ?thesis by simp
qed

declare in-multiset-in-set [code-unfold]

lemma [code]: count (mset xs) x = fold (λy. if x = y then Suc else id) xs 0
proof −
  have ⋀n. fold (λy. if x = y then Suc else id) xs n = count (mset xs) x + n
    by (induct xs) simp-all
  then show ?thesis by simp
qed

declare set-mset-mset [code]

declare sorted-list-of-multiset-mset [code]

lemma [code]: — not very efficient, but representation-ignorant!
  mset-set A = mset (sorted-list-of-set A)
  by (metis mset-sorted-list-of-multiset sorted-list-of-mset-set)

declare size-mset [code]

fun subset-eq-mset-impl :: 'a list ⇒ 'a list ⇒ bool option where
  subset-eq-mset-impl [] ys = Some (ys ≠ [])
| subset-eq-mset-impl (Cons x xs) ys = (case List.extract ((=) x) ys of
    None ⇒ None
  | Some (ys1,-,ys2) ⇒ subset-eq-mset-impl xs (ys1 @ ys2))

lemma subset-eq-mset-impl: (subset-eq-mset-impl xs ys = None ⟷ ¬ mset xs
⊆# mset ys) ∧
  (subset-eq-mset-impl xs ys = Some True ⟷ mset xs ⊂# mset ys) ∧
  (subset-eq-mset-impl xs ys = Some False ⟶ mset xs = mset ys)
proof (induct xs arbitrary: ys)
  case (Nil ys)
  show ?case by (auto simp: subset-mset.zero-less-iff-neq-zero)
next
  case (Cons x xs ys)
  show ?case
  proof (cases List.extract ((=) x) ys)
    case None
    hence x: x ∉ set ys by (simp add: extract-None-iff)
    have nle: False if mset (x # xs) ⊆# mset ys
      using set-mset-mono[OF that] x by simp
    moreover
```

```
    have False if mset (x # xs) ⊂# mset ys
    proof −
      from that have mset (x # xs) ⊆# mset ys by auto
      from nle[OF this] show ?thesis .
    qed
    ultimately show ?thesis using None by auto
  next
    case (Some res)
    obtain ys1 y ys2 where res: res = (ys1,y,ys2) by (cases res, auto)
    note Some = Some[unfolded res]
    from extract-SomeE[OF Some] have ys = ys1 @ x # ys2 by simp
    hence id: mset ys = add-mset x (mset (ys1 @ ys2))
      by auto
    show ?thesis unfolding subset-eq-mset-impl.simps
      by (simp add: Some id Cons)
  qed
qed

lemma [code]: mset xs ⊆# mset ys ⟷ subset-eq-mset-impl xs ys ≠ None
  by (simp add: subset-eq-mset-impl)

lemma [code]: mset xs ⊂# mset ys ⟷ subset-eq-mset-impl xs ys = Some True
  using subset-eq-mset-impl by blast

instantiation multiset :: (equal) equal
begin

definition
  HOL.equal A (B :: 'a multiset) ⟷ A = B

lemma [code]: HOL.equal (mset xs) (mset ys) ⟷ subset-eq-mset-impl xs ys =
Some False
  unfolding equal-multiset-def
  using subset-eq-mset-impl[of xs ys] by (cases subset-eq-mset-impl xs ys, auto)

instance
  by standard (simp add: equal-multiset-def)

end

declare sum-mset-sum-list [code]

lemma [code]: prod-mset (mset xs) = fold times xs 1
proof −
  have ⋀x. fold times xs x = prod-mset (mset xs) * x
    by (induct xs) (simp-all add: ac-simps)
  then show ?thesis by simp
qed
```

Exercise for the casual reader: add implementations for $(\leq)$ and $(<)$

(multiset order).

Quickcheck generators

context
includes *term-syntax*
begin

definition
msetify :: *'a::typerep list* × (*unit* ⇒ *Code-Evaluation.term*)
⇒ *'a multiset* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *msetify xs* = *Code-Evaluation.valtermify mset* {·} *xs*

end

instantiation *multiset* :: (*random*) *random*
begin

context
includes *state-combinator-syntax*
begin

definition
Quickcheck-Random.random i = *Quickcheck-Random.random i* ∘→ (λ*xs*. *Pair* (*msetify xs*))

instance ..

end

end

instantiation *multiset* :: (*full-exhaustive*) *full-exhaustive*
begin

definition *full-exhaustive-multiset* :: (*'a multiset* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
full-exhaustive-multiset f i = *Quickcheck-Exhaustive.full-exhaustive* (λ*xs*. *f* (*msetify xs*)) *i*

instance ..

end

hide-const (**open**) *msetify*

68.17 BNF setup

definition *rel-mset* **where**
rel-mset R X Y ⟷ (∃ *xs ys*. *mset xs* = *X* ∧ *mset ys* = *Y* ∧ *list-all2 R xs ys*)

lemma *mset-zip-take-Cons-drop-twice*:
assumes *length xs = length ys j ≤ length xs*
shows *mset (zip (take j xs @ x # drop j xs) (take j ys @ y # drop j ys)) =*
add-mset (x,y) (mset (zip xs ys))
using *assms*
proof (*induct xs ys arbitrary*: *x y j rule*: *list-induct2*)
case *Nil*
thus *?case*
by *simp*
next
case (*Cons x xs y ys*)
thus *?case*
proof (*cases j = 0*)
case *True*
thus *?thesis*
by *simp*
next
case *False*
then obtain *k* **where** *k*: *j = Suc k*
by (*cases j*) *simp*
hence *k ≤ length xs*
using *Cons.prems* **by** *auto*
hence *mset (zip (take k xs @ x # drop k xs) (take k ys @ y # drop k ys)) =*
add-mset (x,y) (mset (zip xs ys))
by (*rule Cons.hyps*(*2*))
thus *?thesis*
unfolding *k* **by** *auto*
qed
qed

lemma *ex-mset-zip-left*:
assumes *length xs = length ys mset xs′ = mset xs*
shows ∃ *ys′*. *length ys′ = length xs′ ∧ mset (zip xs′ ys′) = mset (zip xs ys)*
using *assms*
proof (*induct xs ys arbitrary*: *xs′ rule*: *list-induct2*)
case *Nil*
thus *?case*
by *auto*
next
case (*Cons x xs y ys xs′*)
obtain *j* **where** *j-len*: *j < length xs′* **and** *nth-j*: *xs′ ! j = x*
by (*metis Cons.prems in-set-conv-nth list.set-intros*(*1*) *mset-eq-setD*)

define *xsa* **where** *xsa = take j xs′ @ drop (Suc j) xs′*
have *mset xs′ = {#x#} + mset xsa*
unfolding *xsa-def* **using** *j-len nth-j*
by (*metis Cons-nth-drop-Suc union-mset-add-mset-right add-mset-remove-trivial*
add-diff-cancel-left′

append-take-drop-id mset.simps(2) mset-append)
hence *ms-x*: *mset xsa = mset xs*
by (*simp add*: *Cons.prems*)
then obtain *ysa* **where**
len-a: *length ysa = length xsa* **and** *ms-a*: *mset* (*zip xsa ysa*) = *mset* (*zip xs ys*)
using *Cons.hyps*(*2*) **by** *blast*

define *ys′* **where** *ys′ = take j ysa @ y # drop j ysa*
have *xs′*: *xs′ = take j xsa @ x # drop j xsa*
using *ms-x j-len nth-j Cons.prems xsa-def*
by (*metis append-eq-append-conv append-take-drop-id diff-Suc-Suc Cons-nth-drop-Suc length-Cons*
length-drop size-mset)
have *j-len′*: *j* ≤ *length xsa*
using *j-len xs′ xsa-def*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append less-eq-Suc-le not-less*)
have *length ys′ = length xs′*
unfolding *ys′-def* **using** *Cons.prems len-a ms-x*
by (*metis add-Suc-right append-take-drop-id length-Cons length-append mset-eq-length*)
moreover have *mset* (*zip xs′ ys′*) = *mset* (*zip* (*x # xs*) (*y # ys*))
unfolding *xs′ ys′-def*
by (*rule trans*[*OF mset-zip-take-Cons-drop-twice*])
(*auto simp*: *len-a ms-a j-len′*)
ultimately show *?case*
by *blast*
qed

lemma *list-all2-reorder-left-invariance*:
assumes *rel*: *list-all2 R xs ys* **and** *ms-x*: *mset xs′ = mset xs*
shows ∃ *ys′*. *list-all2 R xs′ ys′* ∧ *mset ys′ = mset ys*
proof −
have *len*: *length xs = length ys*
using *rel list-all2-conv-all-nth* **by** *auto*
obtain *ys′* **where**
len′: *length xs′ = length ys′* **and** *ms-xy*: *mset* (*zip xs′ ys′*) = *mset* (*zip xs ys*)
using *len ms-x* **by** (*metis ex-mset-zip-left*)
have *list-all2 R xs′ ys′*
using *assms*(*1*) *len′ ms-xy* **unfolding** *list-all2-iff* **by** (*blast dest*: *mset-eq-setD*)
moreover have *mset ys′ = mset ys*
using *len len′ ms-xy map-snd-zip mset-map* **by** *metis*
ultimately show *?thesis*
by *blast*
qed

lemma *ex-mset*: ∃ *xs*. *mset xs = X*
by (*induct X*) (*simp*, *metis mset.simps*(*2*))

inductive *pred-mset* :: ('*a* ⇒ *bool*) ⇒ '*a multiset* ⇒ *bool*

where
pred-mset P {#}
| ⟦*P a*; *pred-mset P M*⟧ ⟹ *pred-mset P* (*add-mset a M*)

lemma *pred-mset-iff*: — TODO: alias for *Multiset.Ball*
‹*pred-mset P M* ⟷ *Multiset.Ball M P*› (**is** ‹*?P* ⟷ *?Q*›)
proof
assume *?P*
then show *?Q* **by** *induction simp-all*
next
assume *?Q*
then show *?P*
by (*induction M*) (*auto intro*: *pred-mset.intros*)
qed

bnf *'a multiset*
map: *image-mset*
sets: *set-mset*
bd: *natLeq*
wits: {#}
rel: *rel-mset*
pred: *pred-mset*
proof −
show *image-mset* (*g* ∘ *f*) = *image-mset g* ∘ *image-mset f* **for** *f g*
unfolding *comp-def* **by** (*rule ext*) (*simp add*: *comp-def image-mset.compositionality*)
show (⋀*z*. *z* ∈ *set-mset X* ⟹ *f z* = *g z*) ⟹ *image-mset f X* = *image-mset g X* **for** *f g X*
by (*induct X*) *simp-all*
show *card-order natLeq*
by (*rule natLeq-card-order*)
show *BNF-Cardinal-Arithmetic.cinfinite natLeq*
by (*rule natLeq-cinfinite*)
show *regularCard natLeq*
by (*rule regularCard-natLeq*)
show *ordLess2* (*card-of* (*set-mset X*)) *natLeq* **for** *X*
by *transfer*
(*auto simp*: *finite-iff-ordLess-natLeq*[*symmetric*])
show *rel-mset R OO rel-mset S* ≤ *rel-mset* (*R OO S*) **for** *R S*
unfolding *rel-mset-def*[*abs-def*] *OO-def*
by (*rule predicate2I*)
(*smt* (*verit*, *ccfv-SIG*) *list-all2-reorder-left-invariance list-all2-trans*)
show *rel-mset R* =
(λ*x y*. ∃*z*. *set-mset z* ⊆ {(*x*, *y*). *R x y*} ∧
image-mset fst z = *x* ∧ *image-mset snd z* = *y*) **for** *R*
unfolding *rel-mset-def*[*abs-def*]
by (*intro ext*)
(*metis* (*lifting*) *ex-mset list.in-rel*[*of R*] *mem-Collect-eq set-mset-mset*
mset-map[*of snd*] *mset-map*[*of fst*])
show *pred-mset P* = (λ*x*. *Ball* (*set-mset x*) *P*) **for** *P*

by (*simp add*: *fun-eq-iff pred-mset-iff*)
qed *auto*

inductive *rel-mset'* :: ‹('*a* ⇒ '*b* ⇒ *bool*) ⇒ '*a multiset* ⇒ '*b multiset* ⇒ *bool*›
where
Zero[*intro*]: *rel-mset' R* {#} {#}
| *Plus*[*intro*]: ⟦*R a b*; *rel-mset' R M N*⟧ ⟹ *rel-mset' R* (*add-mset a M*) (*add-mset b N*)

lemma *rel-mset-Zero*: *rel-mset R* {#} {#}
unfolding *rel-mset-def Grp-def* **by** *auto*

declare *multiset.count*[*simp*]
declare *count-Abs-multiset*[*simp*]
declare *multiset.count-inverse*[*simp*]

lemma *rel-mset-Plus*:
assumes *ab*: *R a b*
and *MN*: *rel-mset R M N*
shows *rel-mset R* (*add-mset a M*) (*add-mset b N*)
proof −
have ∃*ya*. *add-mset a* (*image-mset fst y*) = *image-mset fst ya* ∧
add-mset b (*image-mset snd y*) = *image-mset snd ya* ∧
set-mset ya ⊆ {(*x*, *y*). *R x y*}
if *R a b* **and** *set-mset y* ⊆ {(*x*, *y*). *R x y*} **for** *y*
using *that* **by** (*intro exI*[*of* - *add-mset* (*a*,*b*) *y*]) *auto*
thus *?thesis*
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *blast*
qed

lemma *rel-mset'-imp-rel-mset*: *rel-mset' R M N* ⟹ *rel-mset R M N*
by (*induct rule*: *rel-mset'.induct*) (*auto simp*: *rel-mset-Zero rel-mset-Plus*)

lemma *rel-mset-size*: *rel-mset R M N* ⟹ *size M* = *size N*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*

lemma *rel-mset-Zero-iff* [*simp*]:
shows *rel-mset rel* {#} *Y* ⟷ *Y* = {#} **and** *rel-mset rel X* {#} ⟷ *X* = {#}
by (*auto simp add*: *rel-mset-Zero dest*: *rel-mset-size*)

lemma *multiset-induct2*[*case-names empty addL addR*]:
assumes *empty*: *P* {#} {#}
and *addL*: ⋀*a M N*. *P M N* ⟹ *P* (*add-mset a M*) *N*
and *addR*: ⋀*a M N*. *P M N* ⟹ *P M* (*add-mset a N*)
shows *P M N*
by (*induct N rule*: *multiset-induct*; *induct M rule*: *multiset-induct*) (*auto simp*: *assms*)

lemma *multiset-induct2-size*[*consumes 1*, *case-names empty add*]:
assumes *c*: *size M* = *size N*
and *empty*: *P* {#} {#}
and *add*: ⋀*a b M N*. *P M N* ⟹ *P* (*add-mset a M*) (*add-mset b N*)
shows *P M N*
using *c*
proof (*induct M arbitrary*: *N rule*: *measure-induct-rule*[*of size*])
case (*less M*)
show *?case*
proof(*cases M* = {#})
case *True* **hence** *N* = {#} **using** *less.prems* **by** *auto*
thus *?thesis* **using** *True empty* **by** *auto*
next
case *False* **then obtain** *M1 a* **where** *M*: *M* = *add-mset a M1* **by** (*metis multi-nonempty-split*)
have *N* ≠ {#} **using** *False less.prems* **by** *auto*
then obtain *N1 b* **where** *N*: *N* = *add-mset b N1* **by** (*metis multi-nonempty-split*)
have *size M1* = *size N1* **using** *less.prems* **unfolding** *M N* **by** *auto*
thus *?thesis* **using** *M N less.hyps add* **by** *auto*
qed
qed

lemma *msed-map-invL*:
assumes *image-mset f* (*add-mset a M*) = *N*
shows ∃*N1*. *N* = *add-mset* (*f a*) *N1* ∧ *image-mset f M* = *N1*
proof −
have *f a* ∈# *N*
using *assms multiset.set-map*[*of f add-mset a M*] **by** *auto*
then obtain *N1* **where** *N*: *N* = *add-mset* (*f a*) *N1* **using** *multi-member-split* **by** *metis*
have *image-mset f M* = *N1* **using** *assms* **unfolding** *N* **by** *simp*
thus *?thesis* **using** *N* **by** *blast*
qed

lemma *msed-map-invR*:
assumes *image-mset f M* = *add-mset b N*
shows ∃*M1 a*. *M* = *add-mset a M1* ∧ *f a* = *b* ∧ *image-mset f M1* = *N*
proof −
obtain *a* **where** *a*: *a* ∈# *M* **and** *fa*: *f a* = *b*
using *multiset.set-map*[*of f M*] **unfolding** *assms*
by (*metis image-iff union-single-eq-member*)
then obtain *M1* **where** *M*: *M* = *add-mset a M1* **using** *multi-member-split* **by** *metis*
have *image-mset f M1* = *N* **using** *assms* **unfolding** *M fa*[*symmetric*] **by** *simp*
thus *?thesis* **using** *M fa* **by** *blast*
qed

lemma *msed-rel-invL*:
assumes *rel-mset R* (*add-mset a M*) *N*

shows $\exists N1\ b.\ N = \mathit{add\text{-}mset}\ b\ N1 \wedge R\ a\ b \wedge \mathit{rel\text{-}mset}\ R\ M\ N1$
proof −
obtain K **where** KM: $\mathit{image\text{-}mset}\ \mathit{fst}\ K = \mathit{add\text{-}mset}\ a\ M$
and KN: $\mathit{image\text{-}mset}\ \mathit{snd}\ K = N$ **and** sK: $\mathit{set\text{-}mset}\ K \subseteq \{(a,\ b).\ R\ a\ b\}$
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain $K1\ ab$ **where** K: $K = \mathit{add\text{-}mset}\ ab\ K1$ **and** a: $\mathit{fst}\ ab = a$
and $K1M$: $\mathit{image\text{-}mset}\ \mathit{fst}\ K1 = M$ **using** $\mathit{msed\text{-}map\text{-}invR}[OF\ KM]$ **by** *auto*
obtain $N1$ **where** N: $N = \mathit{add\text{-}mset}\ (\mathit{snd}\ ab)\ N1$ **and** $K1N1$: $\mathit{image\text{-}mset}\ \mathit{snd}\ K1 = N1$
using $\mathit{msed\text{-}map\text{-}invL}[OF\ KN[\mathit{unfolded}\ K]]$ **by** *auto*
have Rab: $R\ a\ (\mathit{snd}\ ab)$ **using** $sK\ a$ **unfolding** K **by** *auto*
have $\mathit{rel\text{-}mset}\ R\ M\ N1$ **using** $sK\ K1M\ K1N1$
unfolding K *multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** $N\ Rab$ **by** *auto*
qed

lemma $\mathit{msed\text{-}rel\text{-}invR}$:
assumes $\mathit{rel\text{-}mset}\ R\ M\ (\mathit{add\text{-}mset}\ b\ N)$
shows $\exists M1\ a.\ M = \mathit{add\text{-}mset}\ a\ M1 \wedge R\ a\ b \wedge \mathit{rel\text{-}mset}\ R\ M1\ N$
proof −
obtain K **where** KN: $\mathit{image\text{-}mset}\ \mathit{snd}\ K = \mathit{add\text{-}mset}\ b\ N$
and KM: $\mathit{image\text{-}mset}\ \mathit{fst}\ K = M$ **and** sK: $\mathit{set\text{-}mset}\ K \subseteq \{(a,\ b).\ R\ a\ b\}$
using *assms*
unfolding *multiset.rel-compp-Grp Grp-def* **by** *auto*
obtain $K1\ ab$ **where** K: $K = \mathit{add\text{-}mset}\ ab\ K1$ **and** b: $\mathit{snd}\ ab = b$
and $K1N$: $\mathit{image\text{-}mset}\ \mathit{snd}\ K1 = N$ **using** $\mathit{msed\text{-}map\text{-}invR}[OF\ KN]$ **by** *auto*
obtain $M1$ **where** M: $M = \mathit{add\text{-}mset}\ (\mathit{fst}\ ab)\ M1$ **and** $K1M1$: $\mathit{image\text{-}mset}\ \mathit{fst}\ K1 = M1$
using $\mathit{msed\text{-}map\text{-}invL}[OF\ KM[\mathit{unfolded}\ K]]$ **by** *auto*
have Rab: $R\ (\mathit{fst}\ ab)\ b$ **using** $sK\ b$ **unfolding** K **by** *auto*
have $\mathit{rel\text{-}mset}\ R\ M1\ N$ **using** $sK\ K1N\ K1M1$
unfolding K *multiset.rel-compp-Grp Grp-def* **by** *auto*
thus *?thesis* **using** $M\ Rab$ **by** *auto*
qed

lemma $\mathit{rel\text{-}mset\text{-}imp\text{-}rel\text{-}mset'}$:
assumes $\mathit{rel\text{-}mset}\ R\ M\ N$
shows $\mathit{rel\text{-}mset'}\ R\ M\ N$
using *assms* **proof**(*induct M arbitrary*: *N rule*: *measure-induct-rule*[*of size*])
case (*less M*)
have *c*: *size M* = *size N* **using** *rel-mset-size*[*OF less.prems*] **.**
show *?case*
proof(*cases M* = {#})
case *True* **hence** $N = \{\#\}$ **using** *c* **by** *simp*
thus *?thesis* **using** *True rel-mset'.Zero* **by** *auto*
next
case *False* **then obtain** *M1 a* **where** *M*: *M* = *add-mset a M1* **by** (*metis multi-nonempty-split*)

obtain *N1 b* **where** *N*: $N = \mathit{add\text{-}mset}\ b\ N1$ **and** *R*: $R\ a\ b$ **and** *ms*: *rel-mset R M1 N1*
using *msed-rel-invL*[*OF less.prems*[*unfolded M*]] **by** *auto*
have *rel-mset′ R M1 N1* **using** *less.hyps*[*of M1 N1*] *ms* **unfolding** *M* **by** *simp*
thus *?thesis* **using** *rel-mset′.Plus*[*of R a b*, *OF R*] **unfolding** *M N* **by** *simp*
qed
qed

lemma *rel-mset-rel-mset′*: *rel-mset R M N* = *rel-mset′ R M N*
using *rel-mset-imp-rel-mset′ rel-mset′-imp-rel-mset* **by** *auto*

The main end product for *rel-mset*: inductive characterization:

lemmas *rel-mset-induct*[*case-names empty add*, *induct pred*: *rel-mset*] =
rel-mset′.induct[*unfolded rel-mset-rel-mset′*[*symmetric*]]

68.18 Size setup

lemma *size-multiset-o-map*: *size-multiset g* ∘ *image-mset f* = *size-multiset* (*g* ∘ *f*)
apply (*rule ext*)
subgoal for *x* **by** (*induct x*) *auto*
done

setup ‹
BNF-LFP-Size.register-size-global ***type-name*** ‹*multiset*› ***const-name*** ‹*size-multiset*›
@{*thm size-multiset-overloaded-def*}
@{*thms size-multiset-empty size-multiset-single size-multiset-union size-empty size-single*
size-union}
@{*thms size-multiset-o-map*}
›

68.19 Lemmas about Size

lemma *size-mset-SucE*: *size A* = *Suc n* ⟹ (⋀*a B*. *A* = {#*a*#} + *B* ⟹ *size B* = *n* ⟹ *P*) ⟹ *P*
by (*cases A*) (*auto simp add*: *ac-simps*)

lemma *size-Suc-Diff1*: *x* ∈# *M* ⟹ *Suc* (*size* (*M* − {#*x*#})) = *size M*
using *arg-cong*[*OF insert-DiffM*, *of - - size*] **by** *simp*

lemma *size-Diff-singleton*: *x* ∈# *M* ⟹ *size* (*M* − {#*x*#}) = *size M* − *1*
by (*simp flip*: *size-Suc-Diff1*)

lemma *size-Diff-singleton-if*: *size* (*A* − {#*x*#}) = (*if x* ∈# *A then size A* − *1 else size A*)
by (*simp add*: *diff-single-trivial size-Diff-singleton*)

lemma *size-Un-Int*: *size A* + *size B* = *size* (*A* ∪# *B*) + *size* (*A* ∩# *B*)
by (*metis inter-subset-eq-union size-union subset-mset.diff-add union-diff-inter-eq-sup*)

lemma *size-Un-disjoint*: $A \cap\# B = \{\#\} \Longrightarrow size\ (A \cup\# B) = size\ A + size\ B$
using *size-Un-Int*[*of A B*] **by** *simp*

lemma *size-Diff-subset-Int*: $size\ (M - M') = size\ M - size\ (M \cap\# M')$
by (*metis diff-intersect-left-idem size-Diff-submset subset-mset.inf-le1*)

lemma *diff-size-le-size-Diff*: $size\ (M :: 'multiset) - size\ M' \leq size\ (M - M')$
by (*simp add*: *diff-le-mono2 size-Diff-subset-Int size-mset-mono*)

lemma *size-Diff1-less*: $x \in\# M \Longrightarrow size\ (M - \{\#x\#\}) < size\ M$
by (*rule Suc-less-SucD*) (*simp add*: *size-Suc-Diff1*)

lemma *size-Diff2-less*: $x \in\# M \Longrightarrow y \in\# M \Longrightarrow size\ (M - \{\#x\#\} - \{\#y\#\}) < size\ M$
by (*metis less-imp-diff-less size-Diff1-less size-Diff-subset-Int*)

lemma *size-Diff1-le*: $size\ (M - \{\#x\#\}) \leq size\ M$
by (*cases* $x \in\# M$) (*simp-all add*: *size-Diff1-less less-imp-le diff-single-trivial*)

lemma *size-psubset*: $M \subseteq\# M' \Longrightarrow size\ M < size\ M' \Longrightarrow M \subset\# M'$
using *less-irrefl subset-mset-def* **by** *blast*

lifting-update *multiset.lifting*
lifting-forget *multiset.lifting*

hide-const (**open**) *wcount*

68.20 The set of multisets of a given size

The following operator gives the set of all multisets consisting of n elements drawn from the set A. In other words: all the different ways to put n unlabelled balls into the labelled bins A.

definition *multisets-of-size* :: *'a set* ⇒ *nat* ⇒ *'a multiset set* **where**
multisets-of-size A n = {*X*. *set-mset X* ⊆ *A* ∧ *size X* = *n*}

lemma
assumes $X \in$ *multisets-of-size A n*
shows *multisets-of-size-subset*: *set-mset X* ⊆ *A*
and *multisets-of-size-size*: *size X* = *n*
using *assms* **by** (*auto simp*: *multisets-of-size-def*)

lemma *multisets-of-size-mono*:
assumes $A \subseteq B$
shows *multisets-of-size A n* ⊆ *multisets-of-size B n*
unfolding *multisets-of-size-def*
by (*intro Collect-mono*) (*use assms* **in** *auto*)

lemma *multisets-of-size-0* [*simp*]: *multisets-of-size A 0* = {{#}}
proof (*intro equalityI subsetI*)

```
  fix h :: 'a multiset assume h ∈ {{#}}
  thus h ∈ multisets-of-size A 0
    by (auto simp: multisets-of-size-def)
qed (auto simp: multisets-of-size-def fun-eq-iff)

lemma multisets-of-size-empty [simp]: n > 0 ⟹ multisets-of-size {} n = {}
  by (auto simp: multisets-of-size-def fun-eq-iff)

lemma count-le-size: count X x ≤ size X
  by (induction X) auto

lemma bij-betw-multisets-of-size-insert:
  assumes a ∉ A
  shows   bij-betw (λ(m,X). X + replicate-mset m a)
          (SIGMA m:{0..n}. multisets-of-size A (n − m)) (multisets-of-size (insert
a A) n)
proof −
  define B where B = (SIGMA m:{0..n}. multisets-of-size A (n − m))
  define C where C = (multisets-of-size (insert a A) n)
  define f where f = (λ(m,X). X + replicate-mset m a)
  define g where g = (λX. (count X a, filter-mset (λx. x ≠ a) X))
  note defs = B-def C-def f-def g-def

  have ∗: size (filter-mset (λx. x ≠ a) X) = size X − count X a for X
  proof −
    have size X = size (filter-mset (λx. x ≠ a) X) + count X a
      by (induction X) auto
    thus ?thesis
      by linarith
  qed

  have 1: f mX ∈ C if mX ∈ B for mX
    using that by (auto simp: multisets-of-size-def defs split: if-splits)

  have 2: g X ∈ B if X ∈ C for X
    using that by (auto simp: multisets-of-size-def count-le-size ∗ defs)

  have 3: g (f mX) = mX if mX ∈ B for mX
    using that assms
    by (auto simp: multisets-of-size-def multiset-eq-iff defs simp flip: not-in-iff)

  have 4: f (g X) = X if X ∈ C for X
    using that
    by (auto simp: multisets-of-size-def multiset-eq-iff defs simp flip: not-in-iff)

  have f ‘ B = C
    using 1 2 4 unfolding set-eq-iff image-iff by metis
  moreover have inj-on f B
    using 3 unfolding inj-on-def by metis
```

ultimately show *?thesis*
unfolding *bij-betw-def defs* **by** *metis*
qed

lemma *multisets-of-size-insert*:
assumes $a \notin A$
shows *multisets-of-size* (*insert a A*) *n* =
($\bigcup m{\leq}n$. (λX. *X* + *replicate-mset m a*) ‘ *multisets-of-size A* (*n* − *m*))
proof −
have *multisets-of-size* (*insert a A*) *n* =
(λ(*m*,*X*). *X* + *replicate-mset m a*) ‘ (*SIGMA m*:{*0*..*n*}. *multisets-of-size A* (*n* − *m*))
using *bij-betw-multisets-of-size-insert*[*OF assms*, *of n*] **unfolding** *bij-betw-def* **by** *simp*
also have ... = ($\bigcup m{\leq}n$. (λX. *X* + *replicate-mset m a*) ‘ *multisets-of-size A* (*n* − *m*))
unfolding *Sigma-def image-UN atLeast0AtMost image-insert image-empty prod.case*
UNION-singleton-eq-range image-image **by** (*rule refl*)
finally show *?thesis* **.**
qed

primrec *multisets-of-size-list* :: *'a list* ⇒ *nat* ⇒ *'a list list* **where**
multisets-of-size-list [] *n* = (*if n* = *0 then* [[]] *else* [])
| *multisets-of-size-list* (*x* # *xs*) *n* =
[*replicate m x* @ *ys* . *m* ← [*0*..<*n*+*1*], *ys* ← *multisets-of-size-list xs* (*n* − *m*)]

lemma *multisets-of-size-list-correct*:
assumes *distinct xs*
shows *mset* ‘ *set* (*multisets-of-size-list xs n*) = *multisets-of-size* (*set xs*) *n*
using *assms*
proof (*induction xs arbitrary*: *n*)
case *Nil*
thus *?case*
by (*cases n* = *0*) *auto*
next
case (*Cons x xs n*)
have *IH*: *multisets-of-size* (*set xs*) *n* = *mset* ‘ *set* (*multisets-of-size-list xs n*) **for** *n*
by (*rule Cons.IH* [*symmetric*]) (*use Cons.prems* **in** *auto*)
from *Cons.prems* **have** $x \notin$ *set xs*
by *auto*
thus *?case*
by (*simp add*: *multisets-of-size-insert image-UN atLeastLessThanSuc-atLeastAtMost image-image*
add-ac atLeast0AtMost IH del: *upt-Suc*)
qed

lemma *multisets-of-size-code* [*code*]:

multisets-of-size (*set xs*) *n* = *set* (*map mset* (*multisets-of-size-list* (*remdups xs*) *n*))
using *multisets-of-size-list-correct*[*of remdups xs*] **by** *simp*

lemma *finite-multisets-of-size* [*intro*]:
assumes *finite A*
shows *finite* (*multisets-of-size A n*)
using *assms*
proof (*induction arbitrary*: *n rule*: *finite-induct*)
case *empty*
thus *?case*
by (*cases n* = *0*) *auto*
next
case (*insert x A n*)
have *finite* (*SIGMA m*:{*0*..*n*}. *multisets-of-size A* (*n* − *m*))
by (*auto intro*: *insert.IH*)
also have *?this* ⟷ *finite* (*multisets-of-size* (*insert x A*) *n*)
by (*rule bij-betw-finite*, *rule bij-betw-multisets-of-size-insert*) *fact*
finally show *?case* **.**
qed

lemma *card-multisets-of-size*:
assumes *finite A*
shows *card* (*multisets-of-size A n*) = (*card A* + *n* − *1*) *choose n*
using *assms*
proof (*induction A arbitrary*: *n rule*: *finite-induct*)
case *empty*
thus *?case*
by (*cases n* = *0*) *auto*
next
case (*insert a A n*)
have *card* (*multisets-of-size* (*insert a A*) *n*) = *card* (*SIGMA m*:{*0*..*n*}. *multisets-of-size A* (*n* − *m*))
using *bij-betw-same-card*[*OF bij-betw-multisets-of-size-insert*[*of a A*], *of n*] *insert.hyps*
by *simp*
also have ... = (∑ *m*=*0*..*n*. *card* (*multisets-of-size A* (*n* − *m*)))
by (*intro card-SigmaI*) (*use insert.hyps* **in** *auto*)
also have ... = (∑ *m*=*0*..*n*. (*card A* + (*n* − *m*) − *1*) *choose* (*n* − *m*))
by (*intro sum.cong insert.IH refl*)
also have ... = (∑ *m*=*0*..*n*. (*card A* + *m* − *1*) *choose m*)
by (*intro sum.reindex-bij-witness*[*of* - λ*m*. *n* − *m* λ*m*. *n* − *m*]) *auto*
also have ... = (*card A* + *n*) *choose n*
proof (*cases card A* = *0*)
case *True*
have (∑ *m*=*0*..*n*. (*card A* + *m* − *1*) *choose m*) = (∑ *m*∈{*0*}. (*m*−*1*) *choose m*)
by (*intro sum.mono-neutral-cong-right*) (*use True* **in** *auto*)
also have ... = *1*

by *simp*
also have ... = (*card A* + *n*) *choose n*
using *True* **by** *simp*
finally show *?thesis* **.**
next
case *False*
have ($\sum$ *m*=*0*..*n*. (*card A* + *m* − *1*) *choose m*) = ($\sum$ *m*≤*n*. ((*card A* − *1*) + *m*) *choose m*)
by (*intro sum.cong*) (*use False* **in** ‹*auto simp*: *algebra-simps*›)
also have ... = ($\sum$ *m*≤*n*. ((*card A* − *1*) + *m*) *choose* (*card A* − *1*))
by (*subst binomial-symmetric*) *auto*
also have ... = (*card A* + *n*) *choose n*
using *choose-rising-sum*(*2*)[*of card A* − *1 n*] *False* **by** *simp*
finally show *?thesis* **.**
qed
finally show *?case*
using *insert.hyps* **by** *simp*
qed

end

69 More Theorems about the Multiset Order

theory *Multiset-Order*
imports *Multiset*
begin

69.1 Alternative Characterizations

69.1.1 The Dershowitz–Manna Ordering

definition *multp*$_{DM}$ **where**
multp$_{DM}$ *r M N* ⟷
(∃ *X Y*. *X* ≠ {#} ∧ *X* ⊆# *N* ∧ *M* = (*N* − *X*) + *Y* ∧ (∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *r k a*)))

lemma *multp*$_{DM}$*-imp-multp*:
multp$_{DM}$ *r M N* ⟹ *multp r M N*
proof −
assume *multp*$_{DM}$ *r M N*
then obtain *X Y* **where**
X ≠ {#} **and** *X* ⊆# *N* **and** *M* = *N* − *X* + *Y* **and** ∀ *k*. *k* ∈# *Y* ⟶ (∃ *a*. *a* ∈# *X* ∧ *r k a*)
unfolding *multp*$_{DM}$*-def* **by** *blast*
then have *multp r* (*N* − *X* + *Y*) (*N* − *X* + *X*)
by (*intro one-step-implies-multp*) (*auto simp*: *Bex-def trans-def*)
with ‹*M* = *N* − *X* + *Y*› ‹*X* ⊆# *N*› **show** *multp r M N*
by (*metis subset-mset.diff-add*)
qed

69.1.2 The Huet–Oppen Ordering

definition *multp$_{HO}$* **where**
multp$_{HO}$ r M N $\longleftrightarrow$ *M* $\neq$ *N* $\wedge$ ($\forall$ *y. count N y* $<$ *count M y* $\longrightarrow$ ($\exists$ *x. r y x* $\wedge$ *count M x* $<$ *count N x*))

lemma *multp-imp-multp$_{HO}$*:
assumes *asymp r* **and** *transp r*
shows *multp r M N* $\Longrightarrow$ *multp$_{HO}$ r M N*
unfolding *multp-def mult-def*
proof (*induction rule*: *trancl-induct*)
case (*base P*)
then show *?case*
using ‹*asymp r*›
by (*auto elim*!: *mult1-lessE simp*: *count-eq-zero-iff multp$_{HO}$-def split*: *if-splits dest*!: *Suc-lessD*)
next
case (*step N P*)
from *step*(*3*) **have** *M* $\neq$ *N* **and**
$**$: $\bigwedge$*y. count N y* $<$ *count M y* $\Longrightarrow$ ($\exists$ *x. r y x* $\wedge$ *count M x* $<$ *count N x*)
by (*simp-all add*: *multp$_{HO}$-def*)
from *step*(*2*) **obtain** *M0 a K* **where**
$*$: *P* = *add-mset a M0 N* = *M0* + *K a* $\notin\#$ *K* $\bigwedge$*b. b* $\in\#$ *K* $\Longrightarrow$ *r b a*
using ‹*asymp r*› **by** (*auto elim*: *mult1-lessE*)
from ‹*M* $\neq$ *N*› $**$ $*$(*1*,*2*,*3*) **have** *M* $\neq$ *P*
using $*$(*4*) ‹*asymp r*›
by (*metis asympD add-cancel-right-right add-diff-cancel-left' add-single-right-eq-add-mset*[*symmetric*] *count-inI*
count-union diff-diff-add-mset diff-single-trivial in-diff-count multi-member-last)
moreover
have *count-a*: $\exists$ *z. r a z* $\wedge$ *count M z* $<$ *count P z* **if** *count P a* $\leq$ *count M a*
proof −
from ‹*a* $\notin\#$ *K*› **and** *that* **have** *count N a* $<$ *count M a*
unfolding $*$(*1*,*2*) **by** (*auto simp add*: *not-in-iff*)
with $**$ **obtain** *z* **where** *z*: *r a z count M z* $<$ *count N z*
by *blast*
with $*$ **have** *count N z* $\leq$ *count P z*
using ‹*asymp r*›
by (*metis add-diff-cancel-left' add-single-right-eq-add-mset*[*symmetric*] *asympD diff-diff-add-mset*
diff-single-trivial in-diff-count not-le-imp-less)
with *z* **show** *?thesis* **by** *auto*
qed
have $\exists$ *x. r y x* $\wedge$ *count M x* $<$ *count P x* **if** *count-y*: *count P y* $<$ *count M y* **for** *y*
proof (*cases y* = *a*)
case *True*
with *count-y count-a* **show** *?thesis* **by** *auto*
next
case *False*

```
    show ?thesis
    proof (cases y ∈# K)
      case True
      with *(4) have r y a by simp
      then show ?thesis
       by (cases count P a ≤ count M a) (auto dest: count-a intro: ‹transp r›[THEN
transpD])
    next
      case False
      with ‹y ≠ a› have count P y = count N y unfolding *(1,2)
        by (simp add: not-in-iff)
      with count-y ** obtain z where z: r y z count M z < count N z by auto
      show ?thesis
      proof (cases z ∈# K)
        case True
        with *(4) have r z a by simp
        with z(1) show ?thesis
            by (cases count P a ≤ count M a) (auto dest!: count-a intro: ‹transp
r›[THEN transpD])
      next
        case False
        with ‹a ∉# K› have count N z ≤ count P z unfolding *
          by (auto simp add: not-in-iff)
        with z show ?thesis by auto
      qed
    qed
  qed
  ultimately show ?case unfolding multp_HO-def by blast
qed

lemma multp_HO-imp-multp_DM: multp_HO r M N ⟹ multp_DM r M N
unfolding multp_DM-def
proof (intro iffI exI conjI)
  assume multp_HO r M N
  then obtain z where z: count M z < count N z
    unfolding multp_HO-def by (auto simp: multiset-eq-iff nat-neq-iff)
  define X where X = N − M
  define Y where Y = M − N
  from z show X ≠ {#} unfolding X-def by (auto simp: multiset-eq-iff not-less-eq-eq
Suc-le-eq)
  from z show X ⊆# N unfolding X-def by auto
  show M = (N − X) + Y unfolding X-def Y-def multiset-eq-iff count-union
count-diff by force
  show ∀ k. k ∈# Y ⟶ (∃ a. a ∈# X ∧ r k a)
  proof (intro allI impI)
    fix k
    assume k ∈# Y
    then have count N k < count M k unfolding Y-def
      by (auto simp add: in-diff-count)
```

with ‹$multp_{HO}$ r M N› **obtain** a **where** r k a **and** $count$ M a < $count$ N a
unfolding $multp_{HO}$-def **by** $blast$
then show ∃ a. a ∈# X ∧ r k a **unfolding** X-def
by ($auto$ $simp$ add: in-$diff$-$count$)
qed
qed

lemma $multp$-eq-$multp_{DM}$: $asymp$ r ⟹ $transp$ r ⟹ $multp$ r = $multp_{DM}$ r
using $multp_{DM}$-imp-$multp$ $multp$-imp-$multp_{HO}$[$THEN$ $multp_{HO}$-imp-$multp_{DM}$]
by $blast$

lemma $multp$-eq-$multp_{HO}$: $asymp$ r ⟹ $transp$ r ⟹ $multp$ r = $multp_{HO}$ r
using $multp_{HO}$-imp-$multp_{DM}$[$THEN$ $multp_{DM}$-imp-$multp$] $multp$-imp-$multp_{HO}$
by $blast$

lemma $multp_{DM}$-$plus$-$plus I$[$simp$]:
assumes $multp_{DM}$ R $M1$ $M2$
shows $multp_{DM}$ R (M + $M1$) (M + $M2$)
proof −
from $assms$ **obtain** X Y **where**
X ≠ {#} **and** X ⊆# $M2$ **and** $M1$ = $M2$ − X + Y **and** ∀ k. k ∈# Y ⟶ (∃ a. a ∈# X ∧ R k a)
unfolding $multp_{DM}$-def **by** $auto$

show $multp_{DM}$ R (M + $M1$) (M + $M2$)
unfolding $multp_{DM}$-def
proof ($intro$ exI $conjI$)
show X ≠ {#}
using ‹X ≠ {#}› **by** $simp$
next
show X ⊆# M + $M2$
using ‹X ⊆# $M2$›
by ($simp$ add: $subset$-$mset.add$-$increasing$)
next
show M + $M1$ = M + $M2$ − X + Y
using ‹X ⊆# $M2$› ‹$M1$ = $M2$ − X + Y›
by ($metis$ $multiset$-$diff$-$union$-$assoc$ $union$-$assoc$)
next
show ∀ k. k ∈# Y ⟶ (∃ a. a ∈# X ∧ R k a)
using ‹∀ k. k ∈# Y ⟶ (∃ a. a ∈# X ∧ R k a)› **by** $simp$
qed
qed

lemma $multp_{HO}$-$plus$-$plus$[$simp$]: $multp_{HO}$ R (M + $M1$) (M + $M2$) ⟷ $multp_{HO}$ R $M1$ $M2$
unfolding $multp_{HO}$-def **by** $simp$

lemma $strict$-$subset$-$implies$-$multp_{DM}$: A ⊂# B ⟹ $multp_{DM}$ r A B
unfolding $multp_{DM}$-def

by (*metis add.right-neutral add-diff-cancel-right′ empty-iff mset-subset-eq-add-right set-mset-empty subset-mset.lessE*)

lemma *strict-subset-implies-multp$_{HO}$*: $A \subset\# B \Longrightarrow$ *multp$_{HO}$ r A B*
unfolding *multp$_{HO}$-def*
by (*simp add*: *leD mset-subset-eq-count*)

lemma *multp$_{HO}$-implies-one-step-strong*:
assumes *multp$_{HO}$ R A B*
defines $J \equiv B - A$ **and** $K \equiv A - B$
shows $J \neq \{\#\}$ **and** $\forall k \in\# K.\ \exists x \in\# J.\ R\ k\ x$
proof −
show $J \neq \{\#\}$
using ‹*multp$_{HO}$ R A B*›
by (*metis Diff-eq-empty-iff-mset J-def add.right-neutral multp$_{DM}$-def multp$_{HO}$-imp-multp$_{DM}$ multp$_{HO}$-plus-plus subset-mset.add-diff-inverse subset-mset.le-zero-eq*)

show $\forall k{\in\#}K.\ \exists x{\in\#}J.\ R\ k\ x$
using ‹*multp$_{HO}$ R A B*›
by (*metis J-def K-def in-diff-count multp$_{HO}$-def*)
qed

lemma *multp$_{HO}$-minus-inter-minus-inter-iff*:
fixes *M1 M2* :: - *multiset*
shows *multp$_{HO}$ R* (*M1* − *M2*) (*M2* − *M1*) ⟷ *multp$_{HO}$ R M1 M2*
by (*metis diff-intersect-left-idem multiset-inter-commute multp$_{HO}$-plus-plus subset-mset.add-diff-inverse subset-mset.inf.cobounded1*)

lemma *multp$_{HO}$-iff-set-mset-less$_{HO}$-set-mset*:
multp$_{HO}$ R M1 M2 ⟷ (*set-mset* (*M1* − *M2*) ≠ *set-mset* (*M2* − *M1*) ∧
($\forall y \in\# M1 - M2.\ (\exists x \in\# M2 - M1.\ R\ y\ x)$))
unfolding *multp$_{HO}$-minus-inter-minus-inter-iff*[*of R M1 M2*, *symmetric*]
unfolding *multp$_{HO}$-def*
unfolding *count-minus-inter-lt-count-minus-inter-iff*
unfolding *minus-inter-eq-minus-inter-iff*
by *auto*

69.1.3 Monotonicity

lemma *multp$_{DM}$-mono-strong*:
multp$_{DM}$ R M1 M2 ⟹ ($\bigwedge x\ y.\ x \in\# M1 \Longrightarrow y \in\# M2 \Longrightarrow R\ x\ y \Longrightarrow S\ x\ y$)
⟹ *multp$_{DM}$ S M1 M2*
unfolding *multp$_{DM}$-def*
by (*metis add-diff-cancel-left′ in-diffD subset-mset.diff-add*)

lemma *multp$_{HO}$-mono-strong*:
multp$_{HO}$ R M1 M2 ⟹ ($\bigwedge x\ y.\ x \in\# M1 \Longrightarrow y \in\# M2 \Longrightarrow R\ x\ y \Longrightarrow S\ x\ y$)
⟹ *multp$_{HO}$ S M1 M2*
unfolding *multp$_{HO}$-def*

by (*metis count-inI less-zeroE*)

69.1.4 Properties of Orders

Asymmetry The following lemma is a negative result stating that asymmetry of an arbitrary binary relation cannot be simply lifted to $multp_{HO}$. It suffices to have four distinct values to build a counterexample.

lemma *asymp-not-liftable-to-multp$_{HO}$*:
fixes *a b c d* :: *'a*
assumes *distinct* [*a*, *b*, *c*, *d*]
shows ¬ (∀(*R* :: *'a* ⇒ *'a* ⇒ *bool*). *asymp R* ⟶ *asymp* (*multp$_{HO}$ R*))
proof −
define *R* :: *'a* ⇒ *'a* ⇒ *bool* **where**
R = (λ*x y*. *x* = *a* ∧ *y* = *c* ∨ *x* = *b* ∧ *y* = *d* ∨ *x* = *c* ∧ *y* = *b* ∨ *x* = *d* ∧ *y* = *a*)

from *assms*(*1*) **have** {#*a*, *b*#} ≠ {#*c*, *d*#}
by (*metis add-single-right-eq-add-mset*[*symmetric*] *distinct.simps*(*2*) *list.set*(*1*) *list.simps*(*15*) *multi-member-this*
set-mset-add-mset-insert set-mset-single)

from *assms*(*1*) **have** *asymp R*
by (*auto simp*: *R-def intro*: *asymp-onI*)
moreover have ¬ *asymp* (*multp$_{HO}$ R*)
unfolding *asymp-on-def Set.ball-simps not-all not-imp not-not*
proof (*intro exI conjI*)
show *multp$_{HO}$ R* {#*a*, *b*#} {#*c*, *d*#}
unfolding *multp$_{HO}$-def*
using ‹{#*a*, *b*#} ≠ {#*c*, *d*#}› *R-def assms* **by** *auto*
next
show *multp$_{HO}$ R* {#*c*, *d*#} {#*a*, *b*#}
unfolding *multp$_{HO}$-def*
using ‹{#*a*, *b*#} ≠ {#*c*, *d*#}› *R-def assms* **by** *auto*
qed
ultimately show *?thesis*
unfolding *not-all not-imp* **by** *auto*
qed

However, if the binary relation is both asymmetric and transitive, then $multp_{HO}$ is also asymmetric.

lemma *asymp-on-multp$_{HO}$*:
assumes *asymp-on A R* **and** *transp-on A R* **and**
B-sub-A: ⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*
shows *asymp-on B* (*multp$_{HO}$ R*)
proof (*rule asymp-onI*)
fix *M1 M2* :: *'a multiset*
assume *M1* ∈ *B M2* ∈ *B multp$_{HO}$ R M1 M2*

from ‹*transp-on A R*› *B-sub-A* **have** *tran*: *transp-on* (*set-mset* (*M1* − *M2*)) *R*
using ‹*M1* ∈ *B*›

by (*meson in-diffD subset-eq transp-on-subset*)

from ‹*asymp-on A R*› *B-sub-A* **have** *asym*: *asymp-on* (*set-mset* (*M1* − *M2*)) *R*
using ‹*M1* ∈ *B*›
by (*meson in-diffD subset-eq asymp-on-subset*)

show ¬ $multp_{HO}$ *R M2 M1*
proof (*cases M1* − *M2* = {#})
case *True*
then show *?thesis*
using $multp_{HO}$*-implies-one-step-strong*(*1*) **by** *metis*
next
case *False*
hence ∃ *m*∈#*M1* − *M2*. ∀ *x*∈#*M1* − *M2*. *x* ≠ *m* ⟶ ¬ *R m x*
using *Finite-Set.bex-max-element*[*of set-mset* (*M1* − *M2*) *R*, *OF finite-set-mset asym tran*]
by *simp*
with ‹*transp-on A R*› *B-sub-A* **have** ∃ *y*∈#*M2* − *M1*. ∀ *x*∈#*M1* − *M2*. ¬ *R y x*
using ‹$multp_{HO}$ *R M1 M2*›[*THEN* $multp_{HO}$*-implies-one-step-strong*(*2*)]
using *asym*[*THEN irreflp-on-if-asymp-on*, *THEN irreflp-onD*]
by (*metis* ‹*M1* ∈ *B*› ‹*M2* ∈ *B*› *in-diffD subsetD transp-onD*)
thus *?thesis*
unfolding $multp_{HO}$*-iff-set-mset-less*$_{HO}$*-set-mset* **by** *simp*
qed
qed

lemma $asymp\text{-}multp_{HO}$:
assumes *asymp R* **and** *transp R*
shows *asymp* ($multp_{HO}$ *R*)
using *assms* $asymp\text{-}on\text{-}multp_{HO}$[*of UNIV*, *simplified*] **by** *metis*

Irreflexivity **lemma** $irreflp\text{-}on\text{-}multp_{HO}$[*simp*]: *irreflp-on B* ($multp_{HO}$ *R*)
by (*simp add*: *irreflp-onI* $multp_{HO}$*-def*)

Transitivity **lemma** $transp\text{-}on\text{-}multp_{HO}$:
assumes *asymp-on A R* **and** *transp-on A R* **and** *B-sub-A*: ⋀*M*. *M* ∈ *B* ⟹ *set-mset M* ⊆ *A*
shows *transp-on B* ($multp_{HO}$ *R*)
proof (*rule transp-onI*)
from *assms* **have** *asymp-on B* ($multp_{HO}$ *R*)
using $asymp\text{-}on\text{-}multp_{HO}$ **by** *metis*

fix *M1 M2 M3*
assume *hyps*: *M1* ∈ *B M2* ∈ *B M3* ∈ *B* $multp_{HO}$ *R M1 M2* $multp_{HO}$ *R M2 M3*

from *assms* **have**
[*intro*]: *asymp-on* (*set-mset M1* ∪ *set-mset M2*) *R transp-on* (*set-mset M1* ∪ *set-mset M2*) *R*

using ‹*M1* ∈ *B*› ‹*M2* ∈ *B*›
by (*simp-all add*: *asymp-on-subset transp-on-subset*)

from *assms* **have** *transp-on* (*set-mset M1*) *R*
by (*meson transp-on-subset hyps*(*1*))

from ‹$multp_{HO}$ *R M1 M2*› **have**
M1 ≠ *M2* **and**
∀ *y*. *count M2 y* < *count M1 y* ⟶ (∃ *x*. *R y x* ∧ *count M1 x* < *count M2 x*)
unfolding $multp_{HO}$*-def* **by** *simp-all*

from ‹$multp_{HO}$ *R M2 M3*› **have**
M2 ≠ *M3* **and**
∀ *y*. *count M3 y* < *count M2 y* ⟶ (∃ *x*. *R y x* ∧ *count M2 x* < *count M3 x*)
unfolding $multp_{HO}$*-def* **by** *simp-all*

show $multp_{HO}$ *R M1 M3*
proof (*rule ccontr*)
let *?P* = λ*x*. *count M3 x* < *count M1 x* ∧ (∀ *y*. *R x y* ⟶ *count M1 y* ≥ *count M3 y*)

assume ¬ $multp_{HO}$ *R M1 M3*
hence *M1* = *M3* ∨ (∃ *x*. *?P x*)
unfolding $multp_{HO}$*-def* **by** *force*
thus *False*
proof (*elim disjE*)
assume *M1* = *M3*
thus *False*
using ‹*asymp-on B* ($multp_{HO}$ *R*)›[*THEN asymp-onD*]
using ‹*M2* ∈ *B*› ‹*M3* ∈ *B*› ‹$multp_{HO}$ *R M1 M2*› ‹$multp_{HO}$ *R M2 M3*›
by *metis*
next
assume ∃ *x*. *?P x*
hence ∃ *x* ∈# *M1* + *M2*. *?P x*
by (*auto simp*: *count-inI*)
have ∃ *y* ∈# *M1* + *M2*. *?P y* ∧ (∀ *z* ∈# *M1* + *M2*. *R y z* ⟶ ¬ *?P z*)
proof (*rule Finite-Set.bex-max-element-with-property*)
show ∃ *x* ∈# *M1* + *M2*. *?P x*
using ‹∃ *x*. *?P x*›
by (*auto simp*: *count-inI*)
qed *auto*
then obtain *x* **where**
x ∈# *M1* + *M2* **and**
count M3 x < *count M1 x* **and**
∀ *y*. *R x y* ⟶ *count M1 y* ≥ *count M3 y* **and**
∀ *y* ∈# *M1* + *M2*. *R x y* ⟶ *count M3 y* < *count M1 y* ⟶ (∃ *z*. *R y z* ∧ *count M1 z* < *count M3 z*)
by *force*

let *?Q* = $\lambda x'$. $R^{==}$ *x x'* $\wedge$ *count M3 x'* < *count M2 x'*
show *False*
proof (*cases* $\exists$ *x'*. *?Q x'*)
case *True*
have $\exists$ *y* $\in\#$ *M1 + M2*. *?Q y* $\wedge$ ($\forall$ *z* $\in\#$ *M1 + M2*. *R y z* $\longrightarrow$ $\neg$ *?Q z*)
proof (*rule Finite-Set.bex-max-element-with-property*)
show $\exists$ *x* $\in\#$ *M1 + M2*. *?Q x*
using ‹$\exists$ *x*. *?Q x*›
by (*auto simp*: *count-inI*)
qed *auto*
then obtain *x'* **where**
x' $\in\#$ *M1 + M2* **and**
$R^{==}$ *x x'* **and**
count M3 x' < *count M2 x'* **and**
maximality-x': $\forall$ *z* $\in\#$ *M1 + M2*. *R x' z* $\longrightarrow$ $\neg$ ($R^{==}$ *x z*) $\vee$ *count M3 z* $\geq$ *count M2 z*
by (*auto simp*: *linorder-not-less*)
with ‹$multp_{HO}$ *R M2 M3*› **obtain** *y'* **where**
R x' y' **and** *count M2 y'* < *count M3 y'*
unfolding $multp_{HO}$*-def* **by** *auto*
hence *count M2 y'* < *count M1 y'*
by (*smt* (*verit*) ‹$R^{==}$ *x x'*› ‹$\forall$ *y*. *R x y* $\longrightarrow$ *count M3 y* $\leq$ *count M1 y*›
‹*count M3 x* < *count M1 x*› ‹*count M3 x'* < *count M2 x'*› *assms*(*2*) *count-inI*
dual-order.strict-trans1 hyps(*1*) *hyps*(*2*) *hyps*(*3*) *less-nat-zero-code B-sub-A subsetD*
sup2E transp-onD)
with ‹$multp_{HO}$ *R M1 M2*› **obtain** *y''* **where**
R y' y'' **and** *count M1 y''* < *count M2 y''*
unfolding $multp_{HO}$*-def* **by** *auto*
hence *count M3 y''* < *count M2 y''*
by (*smt* (*verit*, *del-insts*) ‹*R x' y'*› ‹$R^{==}$ *x x'*› ‹$\forall$ *y*. *R x y* $\longrightarrow$ *count M3 y* $\leq$ *count M1 y*›
‹*count M2 y'* < *count M3 y'*› ‹*count M3 x* < *count M1 x*› ‹*count M3 x'* < *count M2 x'*›
assms(*2*) *count-greater-zero-iff dual-order.strict-trans1 hyps*(*1*) *hyps*(*2*) *hyps*(*3*)
less-nat-zero-code linorder-not-less B-sub-A subset-iff sup2E transp-onD)

moreover have *count M2 y''* $\leq$ *count M3 y''*
proof −
have *y''* $\in\#$ *M1 + M2*
by (*metis* ‹*count M1 y''* < *count M2 y''*› *count-inI not-less-iff-gr-or-eq union-iff*)

moreover have *R x' y''*
by (*metis* ‹*R x' y'*› ‹*R y' y''*› ‹*count M2 y'* < *count M1 y'*›
‹*transp-on* (*set-mset M1* $\cup$ *set-mset M2*) *R*› ‹*x'* $\in\#$ *M1 + M2*›
calculation count-inI

nat-neq-iff set-mset-union transp-onD union-iff)

moreover have $R^{==}\ x\ y''$
using ‹$R^{==}\ x\ x'$›
by (*metis* (*mono-tags*, *opaque-lifting*) ‹*transp-on* (*set-mset M1* ∪ *set-mset M2*) *R*›
‹$x \in\# M1 + M2$› ‹$x' \in\# M1 + M2$› *calculation*(*1*) *calculation*(*2*) *set-mset-union sup2I1*
transp-onD transp-on-reflclp)

ultimately show *?thesis*
using *maximality-x'*[*rule-format*, *of* y''] **by** *metis*
qed

ultimately show *?thesis*
by *linarith*
next
case *False*
hence $\bigwedge x'.\ R^{==}\ x\ x' \Longrightarrow count\ M2\ x' \leq count\ M3\ x'$
by *auto*
hence $count\ M2\ x \leq count\ M3\ x$
by *simp*
hence $count\ M2\ x < count\ M1\ x$
using ‹$count\ M3\ x < count\ M1\ x$› **by** *linarith*
with ‹$multp_{HO}\ R\ M1\ M2$› **obtain** *y* **where**
R x y **and** $count\ M1\ y < count\ M2\ y$
unfolding $multp_{HO}$*-def* **by** *auto*
hence $count\ M3\ y < count\ M2\ y$
using ‹$\forall y.\ R\ x\ y \longrightarrow count\ M3\ y \leq count\ M1\ y$› *dual-order.strict-trans2*
by *metis*
then show *?thesis*
using *False* ‹*R x y*› **by** *auto*
qed
qed
qed
qed

lemma $transp\text{-}multp_{HO}$:
assumes *asymp R* **and** *transp R*
shows *transp* ($multp_{HO}\ R$)
using *assms* $transp\text{-}on\text{-}multp_{HO}$[*of UNIV*, *simplified*] **by** *metis*

Totality **lemma** $totalp\text{-}on\text{-}multp_{DM}$:
$totalp\text{-}on\ A\ R \Longrightarrow (\bigwedge M.\ M \in B \Longrightarrow set\text{-}mset\ M \subseteq A) \Longrightarrow totalp\text{-}on\ B\ (multp_{DM}\ R)$
by (*smt* (*verit*, *ccfv-SIG*) *count-inI in-mono* $multp_{HO}$*-def* $multp_{HO}$*-imp-*$multp_{DM}$ *not-less-iff-gr-or-eq*
totalp-onD totalp-onI)

lemma *totalp-multp$_{DM}$*: *totalp R* $\Longrightarrow$ *totalp* (*multp$_{DM}$ R*)
by (*rule totalp-on-multp$_{DM}$*[*of UNIV R UNIV*, *simplified*])

lemma *totalp-on-multp$_{HO}$*:
totalp-on A R $\Longrightarrow$ ($\bigwedge$*M*. *M* $\in$ *B* $\Longrightarrow$ *set-mset M* $\subseteq$ *A*) $\Longrightarrow$ *totalp-on B* (*multp$_{HO}$ R*)
by (*smt* (*verit*, *ccfv-SIG*) *count-inI in-mono multp$_{HO}$-def not-less-iff-gr-or-eq totalp-onD*
totalp-onI)

lemma *totalp-multp$_{HO}$*: *totalp R* $\Longrightarrow$ *totalp* (*multp$_{HO}$ R*)
by (*rule totalp-on-multp$_{HO}$*[*of UNIV R UNIV*, *simplified*])

Type Classes **context** *preorder*
begin

lemma *order-mult*: *class.order*
(λ*M N*. (*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*} $\vee$ *M* = *N*)
(λ*M N*. (*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*})
(**is** *class.order ?le ?less*)
proof −
have *irrefl*: $\bigwedge$*M* :: *'a multiset*. ¬ *?less M M*
proof
fix *M* :: *'a multiset*
have *trans* {(*x'*::*'a*, *x*). *x'* < *x*}
by (*rule transI*) (*blast intro*: *less-trans*)
moreover
assume (*M*, *M*) $\in$ *mult* {(*x*, *y*). *x* < *y*}
ultimately have ∃ *I J K*. *M* = *I* + *J* ∧ *M* = *I* + *K*
∧ *J* ≠ {#} ∧ (∀ *k*∈*set-mset K*. ∃ *j*∈*set-mset J*. (*k*, *j*) ∈ {(*x*, *y*). *x* < *y*})
by (*rule mult-implies-one-step*)
then obtain *I J K* **where** *M* = *I* + *J* **and** *M* = *I* + *K*
and *J* ≠ {#} **and** (∀ *k*∈*set-mset K*. ∃ *j*∈*set-mset J*. (*k*, *j*) ∈ {(*x*, *y*). *x* < *y*})
by *blast*
then have *aux1*: *K* ≠ {#} **and** *aux2*: ∀ *k*∈*set-mset K*. ∃ *j*∈*set-mset K*. *k* < *j*
by *auto*
have *finite* (*set-mset K*) **by** *simp*
moreover note *aux2*
ultimately have *set-mset K* = {}
by (*induct rule*: *finite-induct*)
(*simp*, *metis* (*mono-tags*) *insert-absorb insert-iff insert-not-empty less-irrefl less-trans*)
with *aux1* **show** *False* **by** *simp*
qed
have *trans*: $\bigwedge$*K M N* :: *'a multiset*. *?less K M* $\Longrightarrow$ *?less M N* $\Longrightarrow$ *?less K N*
unfolding *mult-def* **by** (*blast intro*: *trancl-trans*)
show *class.order ?le ?less*
by *standard* (*auto simp add*: *less-eq-multiset-def irrefl dest*: *trans*)
qed

The Dershowitz–Manna ordering:

definition *less-multiset$_{DM}$* **where**
less-multiset$_{DM}$ M N $\longleftrightarrow$
($\exists$ *X Y*. *X* $\neq$ {#} $\wedge$ *X* $\subseteq$# *N* $\wedge$ *M* = (*N* − *X*) + *Y* $\wedge$ ($\forall$ *k*. *k* $\in$# *Y* $\longrightarrow$ ($\exists$ *a*. *a* $\in$# *X* $\wedge$ *k* < *a*)))

The Huet–Oppen ordering:

definition *less-multiset$_{HO}$* **where**
less-multiset$_{HO}$ M N $\longleftrightarrow$ *M* $\neq$ *N* $\wedge$ ($\forall$ *y*. *count N y* < *count M y* $\longrightarrow$ ($\exists$ *x*. *y* < *x* $\wedge$ *count M x* < *count N x*))

lemma *mult-imp-less-multiset$_{HO}$*:
(*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*} $\Longrightarrow$ *less-multiset$_{HO}$ M N*
unfolding *multp-def*[*of* (<), *symmetric*]
using *multp-imp-multp$_{HO}$*[*of* (<)]
by (*simp add*: *less-multiset$_{HO}$-def multp$_{HO}$-def*)

lemma *less-multiset$_{DM}$-imp-mult*:
less-multiset$_{DM}$ M N $\Longrightarrow$ (*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*}
unfolding *multp-def*[*of* (<), *symmetric*]
by (*rule multp$_{DM}$-imp-multp*[*of* (<) *M N*]) (*simp add*: *less-multiset$_{DM}$-def multp$_{DM}$-def*)

lemma *less-multiset$_{HO}$-imp-less-multiset$_{DM}$*: *less-multiset$_{HO}$ M N* $\Longrightarrow$ *less-multiset$_{DM}$ M N*
unfolding *less-multiset$_{DM}$-def less-multiset$_{HO}$-def*
unfolding *multp$_{DM}$-def*[*symmetric*] *multp$_{HO}$-def*[*symmetric*]
by (*rule multp$_{HO}$-imp-multp$_{DM}$*)

lemma *mult-less-multiset$_{DM}$*: (*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*} $\longleftrightarrow$ *less-multiset$_{DM}$ M N*
unfolding *multp-def*[*of* (<), *symmetric*]
using *multp-eq-multp$_{DM}$*[*of* (<), *simplified*]
by (*simp add*: *multp$_{DM}$-def less-multiset$_{DM}$-def*)

lemma *mult-less-multiset$_{HO}$*: (*M*, *N*) $\in$ *mult* {(*x*, *y*). *x* < *y*} $\longleftrightarrow$ *less-multiset$_{HO}$ M N*
unfolding *multp-def*[*of* (<), *symmetric*]
using *multp-eq-multp$_{HO}$*[*of* (<), *simplified*]
by (*simp add*: *multp$_{HO}$-def less-multiset$_{HO}$-def*)

lemmas *mult$_{DM}$* = *mult-less-multiset$_{DM}$*[*unfolded less-multiset$_{DM}$-def*]
lemmas *mult$_{HO}$* = *mult-less-multiset$_{HO}$*[*unfolded less-multiset$_{HO}$-def*]

end

lemma *less-multiset-less-multiset$_{HO}$*: *M* < *N* $\longleftrightarrow$ *less-multiset$_{HO}$ M N*
unfolding *less-multiset-def multp-def mult$_{HO}$ less-multiset$_{HO}$-def* **..**

lemma *less-multiset$_{DM}$*:

$M < N \longleftrightarrow (\exists X\ Y.\ X \neq \{\#\} \wedge X \subseteq\# N \wedge M = N - X + Y \wedge (\forall k.\ k \in\# Y \longrightarrow (\exists a.\ a \in\# X \wedge k < a)))$
by (*rule* $mult_{DM}$[*folded multp-def less-multiset-def*])

lemma $less\text{-}multiset_{HO}$:
$M < N \longleftrightarrow M \neq N \wedge (\forall y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists x{>}y.\ count\ M\ x < count\ N\ x))$
by (*rule* $mult_{HO}$[*folded multp-def less-multiset-def*])

lemma *subset-eq-imp-le-multiset*:
shows $M \subseteq\# N \Longrightarrow M \leq N$
unfolding *less-eq-multiset-def* $less\text{-}multiset_{HO}$
by (*simp add*: *less-le-not-le subseteq-mset-def*)

lemma *le-multiset-right-total*: $M < add\text{-}mset\ x\ M$
unfolding *less-eq-multiset-def* $less\text{-}multiset_{HO}$ **by** *simp*

lemma *less-eq-multiset-empty-left*[*simp*]: $\{\#\} \leq M$
by (*simp add*: *subset-eq-imp-le-multiset*)

lemma *ex-gt-imp-less-multiset*: $(\exists y.\ y \in\# N \wedge (\forall x.\ x \in\# M \longrightarrow x < y)) \Longrightarrow M < N$
unfolding $less\text{-}multiset_{HO}$
by (*metis count-eq-zero-iff count-greater-zero-iff less-le-not-le*)

lemma *less-eq-multiset-empty-right*[*simp*]: $M \neq \{\#\} \Longrightarrow \neg\ M \leq \{\#\}$
by (*metis less-eq-multiset-empty-left antisym*)

lemma *le-multiset-empty-left*[*simp*]: $M \neq \{\#\} \Longrightarrow \{\#\} < M$
by (*simp add*: $less\text{-}multiset_{HO}$)

lemma *le-multiset-empty-right*[*simp*]: $\neg\ M < \{\#\}$
using *subset-mset.le-zero-eq less-multiset-def multp-def* $less\text{-}multiset_{DM}$ **by** *blast*

lemma *union-le-diff-plus*: $P \subseteq\# M \Longrightarrow N < P \Longrightarrow M - P + N < M$
by (*drule subset-mset.diff-add*[*symmetric*]) (*metis union-le-mono2*)

instantiation *multiset* :: (*preorder*) *ordered-ab-semigroup-monoid-add-imp-le*
begin

lemma $less\text{-}eq\text{-}multiset_{HO}$:
$M \leq N \longleftrightarrow (\forall y.\ count\ N\ y < count\ M\ y \longrightarrow (\exists x.\ y < x \wedge count\ M\ x < count\ N\ x))$
by (*auto simp*: *less-eq-multiset-def* $less\text{-}multiset_{HO}$)

instance by *standard* (*auto simp*: *less-eq-multiset*$_{HO}$)

lemma
fixes *M N* :: *'a multiset*
shows *less-eq-multiset-plus-left*: $N \leq (M + N)$
and *less-eq-multiset-plus-right*: $M \leq (M + N)$
by *simp-all*

lemma
fixes *M N* :: *'a multiset*
shows *le-multiset-plus-left-nonempty*: $M \neq \{\#\} \Longrightarrow N < M + N$
and *le-multiset-plus-right-nonempty*: $N \neq \{\#\} \Longrightarrow M < M + N$
by *simp-all*

end

lemma *all-lt-Max-imp-lt-mset*: $N \neq \{\#\} \Longrightarrow (\forall\, a \in\# M.\ a < Max\ (set\text{-}mset\ N))$
$\Longrightarrow M < N$
by (*meson Max-in*[*OF finite-set-mset*] *ex-gt-imp-less-multiset set-mset-eq-empty-iff*)

lemma *lt-imp-ex-count-lt*: $M < N \Longrightarrow \exists\, y.\ count\ M\ y < count\ N\ y$
by (*meson less-eq-multiset*$_{HO}$ *less-le-not-le*)

lemma *subset-imp-less-mset*: $A \subset\# B \Longrightarrow A < B$
by (*simp add*: *order.not-eq-order-implies-strict subset-eq-imp-le-multiset*)

lemma *image-mset-strict-mono*:
assumes *mono-f*: $\forall\, x \in set\text{-}mset\ M.\ \forall\, y \in set\text{-}mset\ N.\ x < y \longrightarrow f\ x < f\ y$
and *less*: $M < N$
shows *image-mset f M* < *image-mset f N*
proof −
obtain *Y X* **where**
y-nemp: $Y \neq \{\#\}$ **and** *y-sub-N*: $Y \subseteq\# N$ **and** *M-eq*: $M = N - Y + X$ **and**
ex-y: $\forall\, x.\ x \in\# X \longrightarrow (\exists\, y.\ y \in\# Y \wedge x < y)$
using *less*[*unfolded less-multiset*$_{DM}$] **by** *blast*

have *x-sub-M*: $X \subseteq\# M$
using *M-eq* **by** *simp*

let *?fY* = *image-mset f Y*
let *?fX* = *image-mset f X*

show *?thesis*
unfolding *less-multiset*$_{DM}$
proof (*intro exI conjI*)
show *image-mset f M* = *image-mset f N* − *?fY* + *?fX*
using *M-eq*[*THEN arg-cong*, *of image-mset f*] *y-sub-N*
by (*metis image-mset-Diff image-mset-union*)
next

```
    obtain y where y: ∀x. x ∈# X ⟶ y x ∈# Y ∧ x < y x
      using ex-y by metis

    show ∀fx. fx ∈# ?fX ⟶ (∃fy. fy ∈# ?fY ∧ fx < fy)
    proof (intro allI impI)
      fix fx
      assume fx ∈# ?fX
      then obtain x where fx: fx = f x and x-in: x ∈# X
        by auto
      hence y-in: y x ∈# Y and y-gt: x < y x
        using y[rule-format, OF x-in] by blast+
      hence f (y x) ∈# ?fY ∧ f x < f (y x)
        using mono-f y-sub-N x-sub-M x-in
        by (metis image-eqI in-image-mset mset-subset-eqD)
      thus ∃fy. fy ∈# ?fY ∧ fx < fy
        unfolding fx by auto
    qed
  qed (auto simp: y-nemp y-sub-N image-mset-subseteq-mono)
qed

lemma image-mset-mono:
  assumes mono-f: ∀x ∈ set-mset M. ∀y ∈ set-mset N. x < y ⟶ f x < f y
    and less: M ≤ N
  shows image-mset f M ≤ image-mset f N
 by (metis eq-iff image-mset-strict-mono less less-imp-le mono-f order.not-eq-order-implies-strict)

lemma mset-lt-single-right-iff[simp]: M < {#y#} ⟷ (∀x ∈# M. x < y) for y
:: 'a::linorder
proof (rule iffI)
  assume M-lt-y: M < {#y#}
  show ∀x ∈# M. x < y
  proof
    fix x
    assume x-in: x ∈# M
    hence M: M - {#x#} + {#x#} = M
      by (meson insert-DiffM2)
    hence ¬ {#x#} < {#y#} ⟹ x < y
      using x-in M-lt-y
       by (metis diff-single-eq-union le-multiset-empty-left less-add-same-cancel2
mset-le-trans)
    also have ¬ {#y#} < M
      using M-lt-y mset-le-not-sym by blast
    ultimately show x < y
       by (metis (no-types) Max-ge all-lt-Max-imp-lt-mset empty-iff finite-set-mset
insertE
        less-le-trans linorder-less-linear mset-le-not-sym set-mset-add-mset-insert
        set-mset-eq-empty-iff x-in)
  qed
next
```

assume *y-max*: $\forall x \in\# M.\ x < y$
show $M < \{\#y\#\}$
by (*rule all-lt-Max-imp-lt-mset*) (*auto intro*!: *y-max*)
qed

lemma *mset-le-single-right-iff*[*simp*]:
$M \leq \{\#y\#\} \longleftrightarrow M = \{\#y\#\} \vee (\forall x \in\# M.\ x < y)$ **for** $y :: 'a$::*linorder*
by (*meson less-eq-multiset-def mset-lt-single-right-iff*)

69.1.5 Simplifications

lemma $multp_{HO}$-*repeat-mset-repeat-mset*[*simp*]:
assumes $n \neq 0$
shows $multp_{HO}\ R$ (*repeat-mset n A*) (*repeat-mset n B*) $\longleftrightarrow$ $multp_{HO}\ R\ A\ B$
proof (*rule iffI*)
assume *hyp*: $multp_{HO}\ R$ (*repeat-mset n A*) (*repeat-mset n B*)
hence
1: *repeat-mset n A* $\neq$ *repeat-mset n B* **and**
2: $\forall y.\ n * count\ B\ y < n * count\ A\ y \longrightarrow (\exists x.\ R\ y\ x \wedge n * count\ A\ x < n * count\ B\ x)$
by (*simp-all add*: $multp_{HO}$-*def*)

from *1* ‹$n \neq 0$› **have** $A \neq B$
by *auto*

moreover from *2* ‹$n \neq 0$› **have** $\forall y.\ count\ B\ y < count\ A\ y \longrightarrow (\exists x.\ R\ y\ x \wedge count\ A\ x < count\ B\ x)$
by *auto*

ultimately show $multp_{HO}\ R\ A\ B$
by (*simp add*: $multp_{HO}$-*def*)
next
assume $multp_{HO}\ R\ A\ B$
hence *1*: $A \neq B$ **and** *2*: $\forall y.\ count\ B\ y < count\ A\ y \longrightarrow (\exists x.\ R\ y\ x \wedge count\ A\ x < count\ B\ x)$
by (*simp-all add*: $multp_{HO}$-*def*)

from *1* **have** *repeat-mset n A* $\neq$ *repeat-mset n B*
by (*simp add*: *assms repeat-mset-cancel1*)

moreover from *2* **have** $\forall y.\ n * count\ B\ y < n * count\ A\ y \longrightarrow$
$(\exists x.\ R\ y\ x \wedge n * count\ A\ x < n * count\ B\ x)$
by *auto*

ultimately show $multp_{HO}\ R$ (*repeat-mset n A*) (*repeat-mset n B*)
by (*simp add*: $multp_{HO}$-*def*)
qed

lemma $multp_{HO}$-*double-double*[*simp*]: $multp_{HO}\ R\ (A + A)\ (B + B) \longleftrightarrow multp_{HO}$

R A B
using *multp$_{HO}$-repeat-mset-repeat-mset*[*of 2*]
by (*simp add*: *numeral-Bit0*)

69.2 Simprocs

lemma *mset-le-add-iff1*:
$j \leq (i::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \leq repeat\text{-}mset\ j\ u + n) = (repeat\text{-}mset\ (i-j)\ u + m \leq n)$
proof −
assume $j \leq i$
then have $j + (i - j) = i$
using *le-add-diff-inverse* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) *add-le-cancel-left left-add-mult-distrib-mset*)
qed

lemma *mset-le-add-iff2*:
$i \leq (j::nat) \Longrightarrow (repeat\text{-}mset\ i\ u + m \leq repeat\text{-}mset\ j\ u + n) = (m \leq repeat\text{-}mset\ (j-i)\ u + n)$
proof −
assume $i \leq j$
then have $i + (j - i) = j$
using *le-add-diff-inverse* **by** *blast*
then show *?thesis*
by (*metis* (*no-types*) *add-le-cancel-left left-add-mult-distrib-mset*)
qed

simproc-setup *msetless-cancel*
$((l::'a::preorder\ multiset) + m < n \mid (l::'a\ multiset) < m + n \mid$
add-mset a m $<$ *n* $\mid$ *m* $<$ *add-mset a n* $\mid$
replicate-mset p a $<$ *n* $\mid$ *m* $<$ *replicate-mset p a* $\mid$
repeat-mset p m $<$ *n* $\mid$ *m* $<$ *repeat-mset p n*) =
‹*K Cancel-Simprocs.less-cancel*›

simproc-setup *msetle-cancel*
$((l::'a::preorder\ multiset) + m \leq n \mid (l::'a\ multiset) \leq m + n \mid$
add-mset a m $\leq$ *n* $\mid$ *m* $\leq$ *add-mset a n* $\mid$
replicate-mset p a $\leq$ *n* $\mid$ *m* $\leq$ *replicate-mset p a* $\mid$
repeat-mset p m $\leq$ *n* $\mid$ *m* $\leq$ *repeat-mset p n*) =
‹*K Cancel-Simprocs.less-eq-cancel*›

69.3 Additional facts and instantiations

lemma *ex-gt-count-imp-le-multiset*:
$(\forall y :: 'a :: order.\ y \in\# M + N \longrightarrow y \leq x) \Longrightarrow count\ M\ x < count\ N\ x \Longrightarrow M < N$
unfolding *less-multiset$_{HO}$*
by (*metis count-greater-zero-iff le-imp-less-or-eq less-imp-not-less not-gr-zero union-iff*)

lemma *mset-lt-single-iff*[*iff*]: $\{\#x\#\} < \{\#y\#\} \longleftrightarrow x < y$
unfolding *less-multiset*$_{HO}$ **by** *simp*

lemma *mset-le-single-iff*[*iff*]: $\{\#x\#\} \leq \{\#y\#\} \longleftrightarrow x \leq y$ **for** $x\ y :: 'a$::*order*
unfolding *less-eq-multiset*$_{HO}$ **by** *force*

instance *multiset* :: (*linorder*) *linordered-cancel-ab-semigroup-add*
by *standard* (*metis less-eq-multiset*$_{HO}$ *not-less-iff-gr-or-eq*)

lemma *less-eq-multiset-total*: $\neg\ M \leq N \Longrightarrow N \leq M$ **for** $M\ N :: 'a$ :: *linorder multiset*
by *simp*

instantiation *multiset* :: (*wellorder*) *wellorder*
begin

lemma *wf-less-multiset*: *wf* $\{(M :: 'a\ multiset, N).\ M < N\}$
unfolding *less-multiset-def multp-def* **by** (*auto intro*: *wf-mult wf*)

instance
proof *intro-classes*
fix $P :: 'a\ multiset \Rightarrow bool$ **and** $a :: 'a\ multiset$
have *wfp* $((<) :: 'a \Rightarrow 'a \Rightarrow bool)$
using *wfp-on-less* **.**
hence *wfp* $((<) :: 'a\ multiset \Rightarrow 'a\ multiset \Rightarrow bool)$
unfolding *less-multiset-def* **by** (*rule wfp-multp*)
thus $(\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x) \Longrightarrow P\ a$
unfolding *wfp-on-def*[*of UNIV*, *simplified*] **by** *metis*
qed

end

instantiation *multiset* :: (*preorder*) *order-bot*
begin

definition *bot-multiset* :: $'a\ multiset$ **where** *bot-multiset* $= \{\#\}$

instance by *standard* (*simp add*: *bot-multiset-def*)

end

instance *multiset* :: (*preorder*) *no-top*
proof *standard*
fix $x :: 'a\ multiset$
obtain $a :: 'a$ **where** *True* **by** *simp*
have $x < x + (x + \{\#a\#\})$
by *simp*
then show $\exists\, y.\ x < y$
by *blast*

qed

instance *multiset* :: (*preorder*) *ordered-cancel-comm-monoid-add*
by *standard*

instantiation *multiset* :: (*linorder*) *distrib-lattice*
begin

definition *inf-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **where**
inf-multiset A B = (*if A* < *B then A else B*)

definition *sup-multiset* :: *'a multiset* ⇒ *'a multiset* ⇒ *'a multiset* **where**
sup-multiset A B = (*if B* > *A then B else A*)

instance
by *intro-classes* (*auto simp*: *inf-multiset-def sup-multiset-def*)

end

lemma *add-mset-lt-left-lt*: *a* < *b* ⟹ *add-mset a A* < *add-mset b A*
by *fastforce*

lemma *add-mset-le-left-le*: *a* ≤ *b* ⟹ *add-mset a A* ≤ *add-mset b A* **for** *a* :: *'a* :: *linorder*
by *fastforce*

lemma *add-mset-lt-right-lt*: *A* < *B* ⟹ *add-mset a A* < *add-mset a B*
by *fastforce*

lemma *add-mset-le-right-le*: *A* ≤ *B* ⟹ *add-mset a A* ≤ *add-mset a B*
by *fastforce*

lemma *add-mset-lt-lt-lt*:
assumes *a-lt-b*: *a* < *b* **and** *A-le-B*: *A* < *B*
shows *add-mset a A* < *add-mset b B*
by (*rule less-trans*[*OF add-mset-lt-left-lt*[*OF a-lt-b*] *add-mset-lt-right-lt*[*OF A-le-B*]])

lemma *add-mset-lt-lt-le*: *a* < *b* ⟹ *A* ≤ *B* ⟹ *add-mset a A* < *add-mset b B*
using *add-mset-lt-lt-lt le-neq-trans* **by** *fastforce*

lemma *add-mset-lt-le-lt*: *a* ≤ *b* ⟹ *A* < *B* ⟹ *add-mset a A* < *add-mset b B* **for** *a* :: *'a* :: *linorder*
using *add-mset-lt-lt-lt* **by** (*metis add-mset-lt-right-lt le-less*)

lemma *add-mset-le-le-le*:
fixes *a* :: *'a* :: *linorder*
assumes *a-le-b*: *a* ≤ *b* **and** *A-le-B*: *A* ≤ *B*
shows *add-mset a A* ≤ *add-mset b B*
by (*rule order.trans*[*OF add-mset-le-left-le*[*OF a-le-b*] *add-mset-le-right-le*[*OF*

A-le-B]])

lemma *Max-lt-imp-lt-mset*:
assumes *n-nemp*: $N \neq \{\#\}$ **and** *max*: *Max-mset M* < *Max-mset N* (**is** *?max-M* < *?max-N*)
shows $M < N$
proof (*cases* $M = \{\#\}$)
case *m-nemp*: *False*

have *max-n-in-n*: *?max-N* $\in\#$ N
using *n-nemp* **by** *simp*
have *max-n-nin-m*: *?max-N* $\notin\#$ M
using *max Max-ge leD* **by** *auto*

have $M \neq N$
using *max* **by** *auto*
moreover
have $\exists x > y.\ count\ M\ x < count\ N\ x$ **if** $count\ N\ y < count\ M\ y$ **for** y
proof −
from *that* **have** $y \in\#\ M$
by (*simp add*: *count-inI*)
then have *?max-M* $\geq y$
by *simp*
then have *?max-N* $> y$
using *max* **by** *auto*
then show *?thesis*
using *max-n-nin-m max-n-in-n count-inI* **by** *force*
qed
ultimately show *?thesis*
unfolding $less\text{-}multiset_{HO}$ **by** *blast*
qed (*auto simp*: *n-nemp*)

end

70 Fixed Length Lists

theory *NList*
imports *Main*
begin

definition *nlists* :: *nat* ⇒ *'a set* ⇒ *'a list set*
where $nlists\ n\ A = \{xs.\ size\ xs = n \wedge set\ xs \subseteq A\}$

lemma *nlistsI*: $[\![\ size\ xs = n;\ set\ xs \subseteq A\ ]\!] \Longrightarrow xs \in nlists\ n\ A$
by (*simp add*: *nlists-def*)

These [simp] attributes are double-edged. Many proofs in Jinja rely on it but they can degrade performance.

lemma *nlistsE-length* [*simp*]: $xs \in nlists\ n\ A \Longrightarrow size\ xs = n$

by (*simp add*: *nlists-def*)

lemma *in-nlists-UNIV*: $xs \in nlists\ k\ UNIV \longleftrightarrow length\ xs = k$
unfolding *nlists-def* **by**(*auto*)

lemma *less-lengthI*: $[\![\ xs \in nlists\ n\ A;\ p < n\ ]\!] \Longrightarrow p < size\ xs$
by (*simp*)

lemma *nlistsE-set*[*simp*]: $xs \in nlists\ n\ A \Longrightarrow set\ xs \subseteq A$
unfolding *nlists-def* **by** (*simp*)

lemma *nlists-mono*:
assumes $A \subseteq B$ **shows** $nlists\ n\ A \subseteq nlists\ n\ B$
proof
  fix *xs* **assume** $xs \in nlists\ n\ A$
  then obtain *size*: $size\ xs = n$ **and** *inA*: $set\ xs \subseteq A$ **by** (*simp*)
  with *assms* **have** $set\ xs \subseteq B$ **by** *simp*
  with *size* **show** $xs \in nlists\ n\ B$ **by**(*clarsimp intro*!: *nlistsI*)
qed

lemma *nlists-singleton*: $nlists\ n\ \{a\} = \{replicate\ n\ a\}$
unfolding *nlists-def* **by**(*auto simp*: *replicate-length-same dest*!: *subset-singletonD*)

lemma *nlists-n-0* [*simp*]: $nlists\ 0\ A = \{[]\}$
unfolding *nlists-def* **by** (*auto*)

lemma *in-nlists-Suc-iff*: $(xs \in nlists\ (Suc\ n)\ A) = (\exists\, y{\in}A.\ \exists\, ys \in nlists\ n\ A.\ xs = y\#ys)$
unfolding *nlists-def* **by** (*cases xs*) *auto*

lemma *Cons-in-nlists-Suc* [*iff*]: $(x\#xs \in nlists\ (Suc\ n)\ A) \longleftrightarrow (x{\in}A \wedge xs \in nlists\ n\ A)$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-Suc*: $nlists\ (Suc\ n)\ A = (\bigcup a{\in}A.\ (\#)\ a\ `\ nlists\ n\ A)$
by(*auto simp*: *set-eq-iff image-iff in-nlists-Suc-iff*)

lemma *nlists-not-empty*: $A{\neq}\{\} \Longrightarrow \exists\, xs.\ xs \in nlists\ n\ A$
by (*induct n*) (*auto simp*: *in-nlists-Suc-iff*)

lemma *nlistsE-nth-in*: $[\![\ xs \in nlists\ n\ A;\ i < n\ ]\!] \Longrightarrow xs!i \in A$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-Cons-Suc* [*elim*!]:
  $l\#xs \in nlists\ n\ A \Longrightarrow (\bigwedge n'.\ n = Suc\ n' \Longrightarrow l \in A \Longrightarrow xs \in nlists\ n'\ A \Longrightarrow P) \Longrightarrow P$
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-appendE* [*elim*!]:

a@b ∈ nlists n A ⟹ (⋀n1 n2. n=n1+n2 ⟹ a ∈ nlists n1 A ⟹ b ∈ nlists n2 A ⟹ P) ⟹ P
proof −
have *⋀n. a@b ∈ nlists n A ⟹ ∃ n1 n2. n=n1+n2 ∧ a ∈ nlists n1 A ∧ b ∈ nlists n2 A*
(**is** *⋀n. ?list a n ⟹ ∃ n1 n2. ?P a n n1 n2*)
proof (*induct a*)
fix *n* **assume** *?list [] n*
hence *?P [] n 0 n* **by** *simp*
thus *∃ n1 n2. ?P [] n n1 n2* **by** *fast*
next
fix *n l ls*
assume *?list (l#ls) n*
then obtain *n′* **where** *n*: *n = Suc n′ l ∈ A* **and** *n′*: *ls@b ∈ nlists n′ A* **by** *fastforce*
assume *⋀n. ls @ b ∈ nlists n A ⟹ ∃ n1 n2. n = n1 + n2 ∧ ls ∈ nlists n1 A ∧ b ∈ nlists n2 A*
from *this* **and** *n′* **have** *∃ n1 n2. n′ = n1 + n2 ∧ ls ∈ nlists n1 A ∧ b ∈ nlists n2 A* **.**
then obtain *n1 n2* **where** *n′ = n1 + n2 ls ∈ nlists n1 A b ∈ nlists n2 A* **by** *fast*
with *n* **have** *?P (l#ls) n (n1+1) n2* **by** *simp*
thus *∃ n1 n2. ?P (l#ls) n n1 n2* **by** *fastforce*
qed
moreover assume *a@b ∈ nlists n A ⋀n1 n2. n=n1+n2 ⟹ a ∈ nlists n1 A ⟹ b ∈ nlists n2 A ⟹ P*
ultimately show *?thesis* **by** *blast*
qed

lemma *nlists-update-in-list* [*simp*, *intro*!]:
⟦ xs ∈ nlists n A; x∈A ⟧ ⟹ xs[i := x] ∈ nlists n A
by (*metis length-list-update nlistsE-length nlistsE-set nlistsI set-update-subsetI*)

lemma *nlists-appendI* [*intro?*]:
⟦ a ∈ nlists n A; b ∈ nlists m A ⟧ ⟹ a @ b ∈ nlists (n+m) A
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-append*:
xs @ ys ∈ nlists k A ⟷
k = length(xs @ ys) ∧ xs ∈ nlists (length xs) A ∧ ys ∈ nlists (length ys) A
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-map* [*simp*]: *(map f xs ∈ nlists (size xs) A) = (f ‘ set xs ⊆ A)*
unfolding *nlists-def* **by** (*auto*)

lemma *nlists-replicateI* [*intro*]: *x ∈ A ⟹ replicate n x ∈ nlists n A*
by (*induct n*) *auto*

Link to an executable version on lists in List.

lemma *nlists-set*[*code*]: *nlists n* (*set xs*) = *set*(*List.n-lists n xs*)
by (*metis nlists-def set-n-lists*)

end

71 Non-negative, non-positive integers and reals

theory *Nonpos-Ints*
imports *Complex-Main*
begin

71.1 Non-positive integers

The set of non-positive integers on a ring. (in analogy to the set of non-negative integers $\mathbb{N}$) This is useful e.g. for the Gamma function.

definition *nonpos-Ints* (‹$\mathbb{Z}_{\leq 0}$›) **where** $\mathbb{Z}_{\leq 0}$ = {*of-int n* |*n*. *n* ≤ *0*}

lemma *zero-in-nonpos-Ints* [*simp*,*intro*]: *0* ∈ $\mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro*!: *exI*[*of* - *0*::*int*])

lemma *neg-one-in-nonpos-Ints* [*simp*,*intro*]: −*1* ∈ $\mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro*!: *exI*[*of* - −*1*::*int*])

lemma *neg-numeral-in-nonpos-Ints* [*simp*,*intro*]: −*numeral n* ∈ $\mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** (*auto intro*!: *exI*[*of* - −*numeral n*::*int*])

lemma *one-notin-nonpos-Ints* [*simp*]: (*1* :: *'a* :: *ring-char-0*) ∉ $\mathbb{Z}_{\leq 0}$
by (*auto simp*: *nonpos-Ints-def*)

lemma *numeral-notin-nonpos-Ints* [*simp*]: (*numeral n* :: *'a* :: *ring-char-0*) ∉ $\mathbb{Z}_{\leq 0}$
by (*auto simp*: *nonpos-Ints-def*)

lemma *minus-of-nat-in-nonpos-Ints* [*simp*, *intro*]: − *of-nat n* ∈ $\mathbb{Z}_{\leq 0}$
proof −
have − *of-nat n* = *of-int* (−*int n*) **by** *simp*
also have −*int n* ≤ *0* **by** *simp*
hence *of-int* (−*int n*) ∈ $\mathbb{Z}_{\leq 0}$ **unfolding** *nonpos-Ints-def* **by** *blast*
finally show *?thesis* **.**
qed

lemma *of-nat-in-nonpos-Ints-iff*: (*of-nat n* :: *'a* :: {*ring-1*,*ring-char-0*}) ∈ $\mathbb{Z}_{\leq 0}$ ⟷ *n* = *0*
proof
assume (*of-nat n* :: *'a*) ∈ $\mathbb{Z}_{\leq 0}$
then obtain *m* **where** *of-nat n* = (*of-int m* :: *'a*) *m* ≤ *0* **by** (*auto simp*: *nonpos-Ints-def*)
hence (*of-int m* :: *'a*) = *of-nat n* **by** *simp*
also have ... = *of-int* (*int n*) **by** *simp*

finally have $m = int\ n$ **by** (*subst* (*asm*) *of-int-eq-iff*)
with ‹$m \leq 0$› **show** $n = 0$ **by** *auto*
qed *simp*

lemma *nonpos-Ints-of-int*: $n \leq 0 \Longrightarrow of\text{-}int\ n \in \mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def* **by** *blast*

lemma *nonpos-IntsI*:
$x \in \mathbb{Z} \Longrightarrow x \leq 0 \Longrightarrow (x :: 'a :: linordered\text{-}idom) \in \mathbb{Z}_{\leq 0}$
unfolding *nonpos-Ints-def Ints-def* **by** *auto*

lemma *nonpos-Ints-subset-Ints*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{Z}$
unfolding *nonpos-Ints-def Ints-def* **by** *blast*

lemma *nonpos-Ints-nonpos* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \leq (0 :: 'a :: linordered\text{-}idom)$
unfolding *nonpos-Ints-def* **by** *auto*

lemma *nonpos-Ints-Int* [*dest*]: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow x \in \mathbb{Z}$
unfolding *nonpos-Ints-def Ints-def* **by** *blast*

lemma *nonpos-Ints-cases*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = of\text{-}int\ n\ n \leq 0$
using *assms* **unfolding** *nonpos-Ints-def* **by** (*auto elim*!: *Ints-cases*)

lemma *nonpos-Ints-cases'*:
assumes $x \in \mathbb{Z}_{\leq 0}$
obtains n **where** $x = -of\text{-}nat\ n$
proof −
from *assms* **obtain** m **where** $x = of\text{-}int\ m$ **and** m: $m \leq 0$ **by** (*auto elim*!: *nonpos-Ints-cases*)
hence $x = -\ of\text{-}int\ (-m)$ **by** *auto*
also from m **have** $(of\text{-}int\ (-m) :: 'a) = of\text{-}nat\ (nat\ (-m))$ **by** *simp-all*
finally show *?thesis* **by** (*rule that*)
qed

lemma *of-real-in-nonpos-Ints-iff*: $(of\text{-}real\ x :: 'a :: real\text{-}algebra\text{-}1) \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
proof
assume $of\text{-}real\ x \in (\mathbb{Z}_{\leq 0} :: 'a\ set)$
then obtain n **where** $(of\text{-}real\ x :: 'a) = of\text{-}int\ n\ n \leq 0$ **by** (*erule nonpos-Ints-cases*)
note ‹$of\text{-}real\ x = of\text{-}int\ n$›
also have $of\text{-}int\ n = of\text{-}real\ (of\text{-}int\ n)$ **by** (*rule of-real-of-int-eq* [*symmetric*])
finally have $x = of\text{-}int\ n$ **by** (*subst* (*asm*) *of-real-eq-iff*)
with ‹$n \leq 0$› **show** $x \in \mathbb{Z}_{\leq 0}$ **by** (*simp add*: *nonpos-Ints-of-int*)
qed (*auto elim*!: *nonpos-Ints-cases intro*!: *nonpos-Ints-of-int*)

lemma *nonpos-Ints-altdef*: $\mathbb{Z}_{\leq 0} = \{n \in \mathbb{Z}.\ (n :: 'a :: linordered\text{-}idom) \leq 0\}$
by (*auto intro*!: *nonpos-IntsI elim*!: *nonpos-Ints-cases*)

lemma *uminus-in-Nats-iff*: $-x \in \mathbb{N} \longleftrightarrow x \in \mathbb{Z}_{\leq 0}$
proof
assume $-x \in \mathbb{N}$
then obtain n **where** $n \geq 0$ $-x = \textit{of-int}\ n$ **by** (*auto simp*: *Nats-altdef1*)
hence $-n \leq 0$ $x = \textit{of-int}\ (-n)$ **by** (*simp-all add*: *eq-commute minus-equation-iff*[*of* x])
thus $x \in \mathbb{Z}_{\leq 0}$ **unfolding** *nonpos-Ints-def* **by** *blast*
next
assume $x \in \mathbb{Z}_{\leq 0}$
then obtain n **where** $n \leq 0$ $x = \textit{of-int}\ n$ **by** (*auto simp*: *nonpos-Ints-def*)
hence $-n \geq 0$ $-x = \textit{of-int}\ (-n)$ **by** (*simp-all add*: *eq-commute minus-equation-iff*[*of* x])
thus $-x \in \mathbb{N}$ **unfolding** *Nats-altdef1* **by** *blast*
qed

lemma *uminus-in-nonpos-Ints-iff*: $-x \in \mathbb{Z}_{\leq 0} \longleftrightarrow x \in \mathbb{N}$
using *uminus-in-Nats-iff*[*of* $-x$] **by** *simp*

lemma *nonpos-Ints-mult*: $x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{N}$
using *Nats-mult*[*of* $-x$ $-y$] **by** (*simp add*: *uminus-in-Nats-iff*)

lemma *Nats-mult-nonpos-Ints*: $x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
using *Nats-mult*[*of* x $-y$] **by** (*simp add*: *uminus-in-Nats-iff*)

lemma *nonpos-Ints-mult-Nats*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x * y \in \mathbb{Z}_{\leq 0}$
using *Nats-mult*[*of* $-x$ y] **by** (*simp add*: *uminus-in-Nats-iff*)

lemma *nonpos-Ints-add*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x + y \in \mathbb{Z}_{\leq 0}$
using *Nats-add*[*of* $-x$ $-y$] *uminus-in-Nats-iff*[*of* $y+x$, *simplified minus-add*]
by (*simp add*: *uminus-in-Nats-iff add.commute*)

lemma *nonpos-Ints-diff-Nats*:
$x \in \mathbb{Z}_{\leq 0} \Longrightarrow y \in \mathbb{N} \Longrightarrow x - y \in \mathbb{Z}_{\leq 0}$
using *Nats-add*[*of* $-x$ y] *uminus-in-Nats-iff*[*of* $x-y$, *simplified minus-add*]
by (*simp add*: *uminus-in-Nats-iff add.commute*)

lemma *Nats-diff-nonpos-Ints*:
$x \in \mathbb{N} \Longrightarrow y \in \mathbb{Z}_{\leq 0} \Longrightarrow x - y \in \mathbb{N}$
using *Nats-add*[*of* x $-y$] **by** (*simp add*: *uminus-in-Nats-iff add.commute*)

lemma *plus-of-nat-eq-0-imp*: $z + \textit{of-nat}\ n = 0 \Longrightarrow z \in \mathbb{Z}_{\leq 0}$
proof −
assume $z + \textit{of-nat}\ n = 0$
hence *A*: $z = -\ \textit{of-nat}\ n$ **by** (*simp add*: *eq-neg-iff-add-eq-0*)
show $z \in \mathbb{Z}_{\leq 0}$ **by** (*subst A*) *simp*
qed

71.2 Non-negative reals

definition *nonneg-Reals* :: *'a::real-algebra-1 set* (‹$\mathbb{R}_{\geq 0}$›)
where $\mathbb{R}_{\geq 0} = \{$*of-real r* | *r*. $r \geq 0\}$

lemma *nonneg-Reals-of-real-iff* [*simp*]: *of-real r* $\in \mathbb{R}_{\geq 0} \longleftrightarrow r \geq 0$
by (*force simp add*: *nonneg-Reals-def*)

lemma *nonneg-Reals-subset-Reals*: $\mathbb{R}_{\geq 0} \subseteq \mathbb{R}$
unfolding *nonneg-Reals-def Reals-def* **by** *blast*

lemma *nonneg-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\geq 0} \Longrightarrow x \in \mathbb{R}$
unfolding *nonneg-Reals-def Reals-def* **by** *blast*

lemma *nonneg-Reals-of-nat-I* [*simp*]: *of-nat n* $\in \mathbb{R}_{\geq 0}$
by (*metis nonneg-Reals-of-real-iff of-nat-0-le-iff of-real-of-nat-eq*)

lemma *nonneg-Reals-cases*:
assumes $x \in \mathbb{R}_{\geq 0}$
obtains *r* **where** $x =$ *of-real r* $r \geq 0$
using *assms* **unfolding** *nonneg-Reals-def* **by** (*auto elim*!: *Reals-cases*)

lemma *nonneg-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\geq 0}$
unfolding *nonneg-Reals-def* **by** *auto*

lemma *nonneg-Reals-one-I* [*simp*]: $1 \in \mathbb{R}_{\geq 0}$
by (*metis* (*mono-tags*, *lifting*) *nonneg-Reals-of-nat-I of-nat-1*)

lemma *nonneg-Reals-minus-one-I* [*simp*]: $-1 \notin \mathbb{R}_{\geq 0}$
by (*metis nonneg-Reals-of-real-iff le-minus-one-simps*(*3*) *of-real-1 of-real-def real-vector.scale-minus-left*)

lemma *nonneg-Reals-numeral-I* [*simp*]: *numeral w* $\in \mathbb{R}_{\geq 0}$
by (*metis* (*no-types*) *nonneg-Reals-of-nat-I of-nat-numeral*)

lemma *nonneg-Reals-minus-numeral-I* [*simp*]: $-$ *numeral w* $\notin \mathbb{R}_{\geq 0}$
using *nonneg-Reals-of-real-iff not-zero-le-neg-numeral* **by** *fastforce*

lemma *nonneg-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\geq 0}$
apply (*simp add*: *nonneg-Reals-def*)
apply *clarify*
apply (*rename-tac r s*)
apply (*rule-tac x*=*r*+*s* **in** *exI*, *auto*)
done

lemma *nonneg-Reals-mult-I* [*simp*]: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\geq 0}$
unfolding *nonneg-Reals-def* **by** (*auto simp*: *of-real-def*)

lemma *nonneg-Reals-inverse-I* [*simp*]:
fixes *a* :: *'a::real-div-algebra*
shows $a \in \mathbb{R}_{\geq 0} \Longrightarrow$ *inverse a* $\in \mathbb{R}_{\geq 0}$

by (*simp add*: *nonneg-Reals-def image-iff*) (*metis inverse-nonnegative-iff-nonnegative of-real-inverse*)

lemma *nonneg-Reals-divide-I* [*simp*]:
fixes a :: *'a::real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\geq 0}$
by (*simp add*: *divide-inverse*)

lemma *nonneg-Reals-pow-I* [*simp*]: $a \in \mathbb{R}_{\geq 0} \Longrightarrow a\hat{}n \in \mathbb{R}_{\geq 0}$
by (*induction n*) *auto*

lemma *complex-nonneg-Reals-iff*: $z \in \mathbb{R}_{\geq 0} \longleftrightarrow Re\ z \geq 0 \wedge Im\ z = 0$
by (*auto simp*: *nonneg-Reals-def*) (*metis complex-of-real-def complex-surj*)

lemma *ii-not-nonneg-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\geq 0}$
by (*simp add*: *complex-nonneg-Reals-iff*)

71.3 Non-positive reals

definition *nonpos-Reals* :: *'a::real-algebra-1 set* ($\langle\mathbb{R}_{\leq 0}\rangle$)
where $\mathbb{R}_{\leq 0} = \{of\text{-}real\ r \mid r.\ r \leq 0\}$

lemma *nonpos-Reals-of-real-iff* [*simp*]: *of-real* $r \in \mathbb{R}_{\leq 0} \longleftrightarrow r \leq 0$
by (*force simp add*: *nonpos-Reals-def*)

lemma *nonpos-Reals-subset-Reals*: $\mathbb{R}_{\leq 0} \subseteq \mathbb{R}$
unfolding *nonpos-Reals-def Reals-def* **by** *blast*

lemma *nonpos-Ints-subset-nonpos-Reals*: $\mathbb{Z}_{\leq 0} \subseteq \mathbb{R}_{\leq 0}$
by (*metis nonpos-Ints-cases nonpos-Ints-nonpos nonpos-Ints-of-int nonpos-Reals-of-real-iff of-real-of-int-eq subsetI*)

lemma *nonpos-Reals-of-nat-iff* [*simp*]: *of-nat* $n \in \mathbb{R}_{\leq 0} \longleftrightarrow n{=}0$
by (*metis nonpos-Reals-of-real-iff of-nat-le-0-iff of-real-of-nat-eq*)

lemma *nonpos-Reals-Real* [*dest*]: $x \in \mathbb{R}_{\leq 0} \Longrightarrow x \in \mathbb{R}$
unfolding *nonpos-Reals-def Reals-def* **by** *blast*

lemma *nonpos-Reals-cases*:
assumes $x \in \mathbb{R}_{\leq 0}$
obtains r **where** $x = of\text{-}real\ r\ r \leq 0$
using *assms* **unfolding** *nonpos-Reals-def* **by** (*auto elim*!: *Reals-cases*)

lemma *uminus-nonneg-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\geq 0} \longleftrightarrow x \in \mathbb{R}_{\leq 0}$
apply (*auto simp*: *nonpos-Reals-def nonneg-Reals-def*)
apply (*metis nonpos-Reals-of-real-iff minus-minus neg-le-0-iff-le of-real-minus*)
done

lemma *uminus-nonpos-Reals-iff* [*simp*]: $-x \in \mathbb{R}_{\leq 0} \longleftrightarrow x \in \mathbb{R}_{\geq 0}$

by (*metis* (*no-types*) *minus-minus uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-zero-I* [*simp*]: $0 \in \mathbb{R}_{\leq 0}$
unfolding *nonpos-Reals-def* **by** *force*

lemma *nonpos-Reals-one-I* [*simp*]: $1 \notin \mathbb{R}_{\leq 0}$
using *nonneg-Reals-minus-one-I uminus-nonneg-Reals-iff* **by** *blast*

lemma *nonpos-Reals-numeral-I* [*simp*]: *numeral* $w \notin \mathbb{R}_{\leq 0}$
using *nonneg-Reals-minus-numeral-I uminus-nonneg-Reals-iff* **by** *blast*

lemma *nonpos-Reals-add-I* [*simp*]: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a + b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-add-I add-uminus-conv-diff minus-diff-eq minus-minus uminus-nonpos-Reals-iff*)

lemma *nonpos-Reals-mult-I1*: $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-mult-I mult-minus-right uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-mult-I2*: $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a * b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-mult-I mult-minus-left uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-mult-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a * \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
apply (*auto intro*: *nonpos-Reals-mult-I2*)
apply (*auto simp*: *nonpos-Reals-def*)
apply (*rule-tac x=r/n* **in** *exI*)
apply (*auto simp*: *field-split-simps*)
done

lemma *nonpos-Reals-inverse-I*:
fixes *a* :: *'a*::*real-div-algebra*
shows $a \in \mathbb{R}_{\leq 0} \Longrightarrow \mathit{inverse}\ a \in \mathbb{R}_{\leq 0}$
using *nonneg-Reals-inverse-I uminus-nonneg-Reals-iff* **by** *fastforce*

lemma *nonpos-Reals-divide-I1*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\geq 0};\ b \in \mathbb{R}_{\leq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
by (*simp add*: *nonpos-Reals-inverse-I nonpos-Reals-mult-I1 divide-inverse*)

lemma *nonpos-Reals-divide-I2*:
fixes *a* :: *'a*::*real-div-algebra*
shows $[\![a \in \mathbb{R}_{\leq 0};\ b \in \mathbb{R}_{\geq 0}]\!] \Longrightarrow a\ /\ b \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-divide-I minus-divide-left uminus-nonneg-Reals-iff*)

lemma *nonpos-Reals-divide-of-nat-iff*:
fixes *a*:: *'a* :: *real-div-algebra* **shows** $a\ /\ \mathit{of\text{-}nat}\ n \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0} \vee n{=}0$
apply (*auto intro*: *nonpos-Reals-divide-I2*)
apply (*auto simp*: *nonpos-Reals-def*)
apply (*rule-tac x=r*n* **in** *exI*)

apply (*auto simp*: *field-split-simps mult-le-0-iff*)
done

lemma *nonpos-Reals-inverse-iff* [*simp*]:
fixes $a :: 'a$::*real-div-algebra*
shows *inverse* $a \in \mathbb{R}_{\leq 0} \longleftrightarrow a \in \mathbb{R}_{\leq 0}$
using *nonpos-Reals-inverse-I* **by** *fastforce*

lemma *nonpos-Reals-pow-I*: $[\![a \in \mathbb{R}_{\leq 0}; \textit{odd } n]\!] \Longrightarrow a \hat{} n \in \mathbb{R}_{\leq 0}$
by (*metis nonneg-Reals-pow-I power-minus-odd uminus-nonneg-Reals-iff*)

lemma *complex-nonpos-Reals-iff*: $z \in \mathbb{R}_{\leq 0} \longleftrightarrow Re\ z \leq 0 \wedge Im\ z = 0$
using *complex-is-Real-iff* **by** (*force simp add*: *nonpos-Reals-def*)

lemma *ii-not-nonpos-Reals* [*iff*]: $\mathrm{i} \notin \mathbb{R}_{\leq 0}$
by (*simp add*: *complex-nonpos-Reals-iff*)

lemma *plus-one-in-nonpos-Ints-imp*: $z + 1 \in \mathbb{Z}_{\leq 0} \Longrightarrow z \in \mathbb{Z}_{\leq 0}$
using *nonpos-Ints-diff-Nats*[*of z+1 1*] **by** *simp-all*

lemma *of-int-in-nonpos-Ints-iff*:
(*of-int* $n :: 'a ::$ *ring-char-0*) $\in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq 0$
by (*auto simp*: *nonpos-Ints-def*)

lemma *one-plus-of-int-in-nonpos-Ints-iff*:
(*1 + of-int* $n :: 'a ::$ *ring-char-0*) $\in \mathbb{Z}_{\leq 0} \longleftrightarrow n \leq -1$
proof −
have *1 + of-int* $n =$ (*of-int* $(n + 1) :: 'a$) **by** *simp*
also have $\ldots \in \mathbb{Z}_{\leq 0} \longleftrightarrow n + 1 \leq 0$ **by** (*subst of-int-in-nonpos-Ints-iff*) *simp-all*
also have $\ldots \longleftrightarrow n \leq -1$ **by** *presburger*
finally show *?thesis* **.**
qed

lemma *one-minus-of-nat-in-nonpos-Ints-iff*:
(*1 − of-nat* $n :: 'a ::$ *ring-char-0*) $\in \mathbb{Z}_{\leq 0} \longleftrightarrow n > 0$
proof −
have (*1 − of-nat* $n :: 'a$) $=$ *of-int* (*1 − int* n) **by** *simp*
also have $\ldots \in \mathbb{Z}_{\leq 0} \longleftrightarrow n > 0$ **by** (*subst of-int-in-nonpos-Ints-iff*) *presburger*
finally show *?thesis* **.**
qed

lemma *fraction-not-in-Nats*:
assumes $\neg n$ *dvd m n* $\neq 0$
shows *of-int m / of-int n* $\notin$ ($\mathbb{N} :: 'a ::$ {*division-ring*,*ring-char-0*} *set*)
proof
assume *of-int m / of-int n* $\in$ ($\mathbb{N} :: 'a$ *set*)
also note *Nats-subset-Ints*
finally have *of-int m / of-int n* $\in$ ($\mathbb{Z} :: 'a$ *set*) **.**
moreover have *of-int m / of-int n* $\notin$ ($\mathbb{Z} :: 'a$ *set*)

using *assms* **by** (*intro fraction-not-in-Ints*)
ultimately show *False* **by** *contradiction*
qed

lemma *not-in-Ints-imp-not-in-nonpos-Ints*: $z \notin \mathbb{Z} \Longrightarrow z \notin \mathbb{Z}_{\leq 0}$
by (*auto simp*: *Ints-def nonpos-Ints-def*)

lemma *double-in-nonpos-Ints-imp*:
assumes $2 * (z :: 'a :: \textit{field-char-0}) \in \mathbb{Z}_{\leq 0}$
shows $z \in \mathbb{Z}_{\leq 0} \vee z + 1/2 \in \mathbb{Z}_{\leq 0}$
proof−
from *assms* **obtain** *k* **where** *k*: $2 * z = -$ *of-nat k* **by** (*elim nonpos-Ints-cases'*)
thus *?thesis* **by** (*cases even k*) (*auto elim*!: *evenE oddE simp*: *field-simps*)
qed

lemma *fraction-numeral-Ints-iff* [*simp*]:
$\textit{numeral } a \mathbin{/} \textit{numeral } b \in (\mathbb{Z} :: 'a :: \{\textit{division-ring}, \textit{ring-char-0}\}\ \textit{set})$
$\longleftrightarrow (\textit{numeral } b :: \textit{int})\ \textit{dvd numeral } a$ (**is** *?L*=*?R*)
proof
show *?L* $\Longrightarrow$ *?R*
by (*metis fraction-not-in-Ints of-int-numeral zero-neq-numeral*)
assume *?R*
then obtain *k*::*int* **where** $\textit{numeral } a = \textit{numeral } b * (\textit{of-int } k :: 'a)$
unfolding *dvd-def* **by** (*metis of-int-mult of-int-numeral*)
then show *?L*
by (*metis Ints-of-int divide-eq-eq mult.commute of-int-mult of-int-numeral*)
qed

lemma *fraction-numeral-Ints-iff1* [*simp*]:
$1 \mathbin{/} \textit{numeral } b \in (\mathbb{Z} :: 'a :: \{\textit{division-ring}, \textit{ring-char-0}\}\ \textit{set})$
$\longleftrightarrow b = \textit{Num.One}$ (**is** *?L*=*?R*)
using *fraction-numeral-Ints-iff* [*of Num.One*, **where** *'a*=*'a*] **by** *simp*

lemma *fraction-numeral-Nats-iff* [*simp*]:
$\textit{numeral } a \mathbin{/} \textit{numeral } b \in (\mathbb{N} :: 'a :: \{\textit{division-ring}, \textit{ring-char-0}\}\ \textit{set})$
$\longleftrightarrow (\textit{numeral } b :: \textit{int})\ \textit{dvd numeral } a$ (**is** *?L*=*?R*)
proof
show *?L* $\Longrightarrow$ *?R*
using *Nats-subset-Ints fraction-numeral-Ints-iff* **by** *blast*
assume *?R*
then obtain *k*::*nat* **where** $\textit{numeral } a = \textit{numeral } b * (\textit{of-nat } k :: 'a)$
unfolding *dvd-def*
by (*metis dvd-def int-dvd-int-iff of-nat-mult of-nat-numeral*)
then show *?L*
by (*metis mult-of-nat-commute nonzero-divide-eq-eq of-nat-in-Nats*
zero-neq-numeral)
qed

lemma *fraction-numeral-Nats-iff1* [*simp*]:

1 / numeral b ∈ (ℕ :: *'a* :: {*division-ring*, *ring-char-0*} *set*)
⟷ *b* = *Num.One* (**is** *?L=?R*)
using *fraction-numeral-Nats-iff* [*of Num.One*, **where** *'a='a*] **by** *simp*

end

72 Numeral Syntax for Types

theory *Numeral-Type*
imports *Cardinality*
begin

72.1 Numeral Types

typedef *num0* = *UNIV* :: *nat set* **..**
typedef *num1* = *UNIV* :: *unit set* **..**

typedef *'a bit0* = {*0* ..< *2* * *int CARD*(*'a*::*finite*)}
proof
show *0* ∈ {*0* ..< *2* * *int CARD*(*'a*)}
by *simp*
qed

typedef *'a bit1* = {*0* ..< *1* + *2* * *int CARD*(*'a*::*finite*)}
proof
show *0* ∈ {*0* ..< *1* + *2* * *int CARD*(*'a*)}
by *simp*
qed

lemma *card-num0* [*simp*]: *CARD* (*num0*) = *0*
unfolding *type-definition.card* [*OF type-definition-num0*]
by *simp*

lemma *infinite-num0*: ¬ *finite* (*UNIV* :: *num0 set*)
using *card-num0*[*unfolded card-eq-0-iff*]
by *simp*

lemma *card-num1* [*simp*]: *CARD*(*num1*) = *1*
unfolding *type-definition.card* [*OF type-definition-num1*]
by (*simp only*: *card-UNIV-unit*)

lemma *card-bit0* [*simp*]: *CARD*(*'a bit0*) = *2* * *CARD*(*'a*::*finite*)
unfolding *type-definition.card* [*OF type-definition-bit0*]
by *simp*

lemma *card-bit1* [*simp*]: *CARD*(*'a bit1*) = *Suc* (*2* * *CARD*(*'a*::*finite*))
unfolding *type-definition.card* [*OF type-definition-bit1*]
by *simp*

72.2 *num1*

instance *num1* :: *finite*
proof
  show *finite* (*UNIV*::*num1 set*)
    using *type-definition.Abs-image* [*OF type-definition-num1*]
    by (*metis finite-class.finite-UNIV finite-imageI*)
qed

instantiation *num1* :: *CARD-1*
begin

instance
proof
  show *CARD*(*num1*) = *1* **by** *auto*
qed

end

lemma *num1-eq-iff*: (*x*::*num1*) = (*y*::*num1*) ⟷ *True*
  by (*induct x*, *induct y*) *simp*

instantiation *num1* :: {*comm-ring*,*comm-monoid-mult*,*numeral*}
begin

instance
  by *standard* (*simp-all add*: *num1-eq-iff*)

end

lemma *num1-eqI*:
  fixes *a*::*num1* **shows** *a* = *b*
by(*simp add*: *num1-eq-iff*)

lemma *num1-eq1* [*simp*]:
  fixes *a*::*num1* **shows** *a* = *1*
  by (*rule num1-eqI*)

lemma *forall-1*[*simp*]: (∀ *i*::*num1*. *P i*) ⟷ *P 1*
  by (*metis* (*full-types*) *num1-eq-iff*)

lemma *ex-1*[*simp*]: (∃ *x*::*num1*. *P x*) ⟷ *P 1*
  by *auto* (*metis* (*full-types*) *num1-eq-iff*)

instantiation *num1* :: *linorder* **begin**
definition *a* < *b* ⟷ *Rep-num1 a* < *Rep-num1 b*
definition *a* ≤ *b* ⟷ *Rep-num1 a* ≤ *Rep-num1 b*
instance
  by *intro-classes* (*auto simp*: *less-eq-num1-def less-num1-def intro*: *num1-eqI*)
end

instance *num1* :: *wellorder*
by *intro-classes* (*auto simp*: *less-eq-num1-def less-num1-def*)

instance *bit0* :: (*finite*) *card2*
proof
show *finite* (*UNIV*::'*a bit0 set*)
by (*simp flip*: *type-definition.Abs-image* [*OF type-definition-bit0*])
show *2* ≤ *CARD*('*a bit0*)
by *simp*
qed

instance *bit1* :: (*finite*) *card2*
proof
show *finite* (*UNIV*::'*a bit1 set*)
by (*simp flip*: *type-definition.Abs-image* [*OF type-definition-bit1*])
show *2* ≤ *CARD*('*a bit1*)
by *simp*
qed

72.3 Locales for for modular arithmetic subtypes

locale *mod-type* =
fixes *n* :: *int*
and *Rep* :: '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
assumes *type*: *type-definition Rep Abs* {*0*..<*n*}
and *size1*: *1* < *n*
and *zero-def*: *0* = *Abs 0*
and *one-def*: *1* = *Abs 1*
and *add-def*: *x* + *y* = *Abs* ((*Rep x* + *Rep y*) *mod n*)
and *mult-def*: *x* * *y* = *Abs* ((*Rep x* * *Rep y*) *mod n*)
and *diff-def*: *x* − *y* = *Abs* ((*Rep x* − *Rep y*) *mod n*)
and *minus-def*: − *x* = *Abs* ((− *Rep x*) *mod n*)
begin

lemma *size0*: *0* < *n*
using *size1* **by** *simp*

lemmas *definitions* =
zero-def one-def add-def mult-def minus-def diff-def

lemma *Rep-less-n*: *Rep x* < *n*
by (*rule type-definition.Rep* [*OF type*, *simplified*, *THEN conjunct2*])

lemma *Rep-le-n*: *Rep x* ≤ *n*
by (*rule Rep-less-n* [*THEN order-less-imp-le*])

lemma *Rep-inject-sym*: $x = y \longleftrightarrow Rep\ x = Rep\ y$
by (*rule type-definition.Rep-inject* [*OF type*, *symmetric*])

lemma *Rep-inverse*: $Abs\ (Rep\ x) = x$
by (*rule type-definition.Rep-inverse* [*OF type*])

lemma *Abs-inverse*: $m \in \{0..<n\} \Longrightarrow Rep\ (Abs\ m) = m$
by (*rule type-definition.Abs-inverse* [*OF type*])

lemma *Rep-Abs-mod*: $Rep\ (Abs\ (m\ mod\ n)) = m\ mod\ n$
using *size0* **by** (*simp add*: *Abs-inverse*)

lemma *Rep-Abs-0*: $Rep\ (Abs\ 0) = 0$
by (*simp add*: *Abs-inverse size0*)

lemma *Rep-0*: $Rep\ 0 = 0$
by (*simp add*: *zero-def Rep-Abs-0*)

lemma *Rep-Abs-1*: $Rep\ (Abs\ 1) = 1$
by (*simp add*: *Abs-inverse size1*)

lemma *Rep-1*: $Rep\ 1 = 1$
by (*simp add*: *one-def Rep-Abs-1*)

lemma *Rep-mod*: $Rep\ x\ mod\ n = Rep\ x$
using *type-definition.Abs-cases* [*OF type*]
by (*metis Abs-inverse atLeastLessThan-iff mod-pos-pos-trivial*)

lemmas *Rep-simps* =
Rep-inject-sym Rep-inverse Rep-Abs-mod Rep-mod Rep-Abs-0 Rep-Abs-1

lemma *comm-ring-1*: *OFCLASS*('*a*, *comm-ring-1-class*)
by *intro-classes* (*auto simp*: *Rep-simps mod-simps field-simps definitions*)

end

locale *mod-ring* = *mod-type n Rep Abs*
for n :: *int*
and *Rep* :: '*a*::{*comm-ring-1*} ⇒ *int*
and *Abs* :: *int* ⇒ '*a*::{*comm-ring-1*}
begin

lemma *of-nat-eq*: *of-nat k* = *Abs* (*int k mod n*)
by (*induct k*) (*simp-all add*: *zero-def Rep-simps add-def one-def mod-simps ac-simps*)

lemma *of-int-eq*: *of-int z* = *Abs* (*z mod n*)
by (*cases z rule*: *int-diff-cases*) (*simp add*: *Rep-simps of-nat-eq diff-def mod-simps*)

lemma *Rep-numeral*: *Rep* (*numeral w*) = *numeral w mod n*
by (*metis Rep-Abs-mod of-int-eq of-int-numeral*)

lemma *iszero-numeral*:
iszero (*numeral w*::'*a*) ⟷ *numeral w mod n* = *0*
by (*simp add*: *Rep-inject-sym Rep-numeral Rep-0 iszero-def*)

lemma *cases*:
assumes *1*: ⋀*z*. ⟦(*x*::'*a*) = *of-int z*; *0* ≤ *z*; *z* < *n*⟧ ⟹ *P*
shows *P*
by (*metis Rep-inverse Rep-less-n Rep-mod assms of-int-eq pos-mod-sign size0*)

lemma *induct*:
(⋀*z*. ⟦*0* ≤ *z*; *z* < *n*⟧ ⟹ *P* (*of-int z*)) ⟹ *P* (*x*::'*a*)
by (*cases x rule*: *cases*) *simp*

lemma *UNIV-eq*: (*UNIV* :: '*a set*) = *Abs* ' {*0*..<*n*}
using *type type-definition.Abs-image* **by** *blast*

lemma *CARD-eq*: *CARD*('*a*) = *nat n*
by (*metis card-atLeastZeroLessThan-int type type-definition.card*)

lemma *CHAR-eq* [*simp*]: *CHAR*('*a*) = *CARD*('*a*)
proof (*rule CHAR-eqI*)
show *of-nat* (*CARD*('*a*)) = (*0* :: '*a*)
by (*simp add*: *CARD-eq of-nat-eq zero-def*)
next
fix *n* **assume** *of-nat n* = (*0* :: '*a*)
thus *CARD*('*a*) *dvd n*
by (*metis CARD-eq Rep-0 Rep-Abs-mod Rep-le-n mod-0-imp-dvd nat-dvd-iff of-nat-eq*)
qed

end

72.4 Ring class instances

Unfortunately *ring-1* instance is not possible for *num1*, since 0 and 1 are not distinct.

instantiation
bit0 **and** *bit1* :: (*finite*) {*zero*,*one*,*plus*,*times*,*uminus*,*minus*}
begin

definition *Abs-bit0'* :: *int* ⇒ '*a bit0* **where**
Abs-bit0' x = *Abs-bit0* (*x mod int CARD*('*a bit0*))

definition *Abs-bit1'* :: *int* ⇒ '*a bit1* **where**
Abs-bit1' x = *Abs-bit1* (*x mod int CARD*('*a bit1*))

definition *0 = Abs-bit0 0*
definition *1 = Abs-bit0 1*
definition *x + y = Abs-bit0' (Rep-bit0 x + Rep-bit0 y)*
definition *x * y = Abs-bit0' (Rep-bit0 x * Rep-bit0 y)*
definition *x - y = Abs-bit0' (Rep-bit0 x - Rep-bit0 y)*
definition *- x = Abs-bit0' (- Rep-bit0 x)*

definition *0 = Abs-bit1 0*
definition *1 = Abs-bit1 1*
definition *x + y = Abs-bit1' (Rep-bit1 x + Rep-bit1 y)*
definition *x * y = Abs-bit1' (Rep-bit1 x * Rep-bit1 y)*
definition *x - y = Abs-bit1' (Rep-bit1 x - Rep-bit1 y)*
definition *- x = Abs-bit1' (- Rep-bit1 x)*

instance ..

end

interpretation *bit0*:
  mod-type int CARD('a::finite bit0)
    Rep-bit0 :: 'a::finite bit0 ⇒ int
    Abs-bit0 :: int ⇒ 'a::finite bit0
apply (*rule mod-type.intro*)
apply (*simp add*: *type-definition-bit0*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit0-def*)
apply (*rule one-bit0-def*)
apply (*rule plus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule times-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule minus-bit0-def* [*unfolded Abs-bit0'-def*])
apply (*rule uminus-bit0-def* [*unfolded Abs-bit0'-def*])
done

interpretation *bit1*:
  mod-type int CARD('a::finite bit1)
    Rep-bit1 :: 'a::finite bit1 ⇒ int
    Abs-bit1 :: int ⇒ 'a::finite bit1
  apply (*rule mod-type.intro*)
apply (*simp add*: *type-definition-bit1*)
apply (*rule one-less-int-card*)
apply (*rule zero-bit1-def*)
apply (*rule one-bit1-def*)
apply (*rule plus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule times-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule minus-bit1-def* [*unfolded Abs-bit1'-def*])
apply (*rule uminus-bit1-def* [*unfolded Abs-bit1'-def*])
done

instance *bit0 :: (finite) comm-ring-1*

by (*rule bit0.comm-ring-1*)

instance *bit1* :: (*finite*) *comm-ring-1*
by (*rule bit1.comm-ring-1*)

interpretation *bit0*:
mod-ring int CARD('a::finite bit0)
Rep-bit0 :: *'a::finite bit0* ⇒ *int*
Abs-bit0 :: *int* ⇒ *'a::finite bit0*
..

interpretation *bit1*:
mod-ring int CARD('a::finite bit1)
Rep-bit1 :: *'a::finite bit1* ⇒ *int*
Abs-bit1 :: *int* ⇒ *'a::finite bit1*
..

Set up cases, induction, and arithmetic

lemmas *bit0-cases* [*case-names of-int*, *cases type*: *bit0*] = *bit0.cases*
lemmas *bit1-cases* [*case-names of-int*, *cases type*: *bit1*] = *bit1.cases*

lemmas *bit0-induct* [*case-names of-int*, *induct type*: *bit0*] = *bit0.induct*
lemmas *bit1-induct* [*case-names of-int*, *induct type*: *bit1*] = *bit1.induct*

lemmas *bit0-iszero-numeral* [*simp*] = *bit0.iszero-numeral*
lemmas *bit1-iszero-numeral* [*simp*] = *bit1.iszero-numeral*

lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit0*] **for** *dummy* :: *'a::finite*
lemmas [*simp*] = *eq-numeral-iff-iszero* [**where** *'a='a bit1*] **for** *dummy* :: *'a::finite*

72.5 Order instances

instantiation *bit0* **and** *bit1* :: (*finite*) *linorder* **begin**
definition $a < b \longleftrightarrow$ *Rep-bit0 a* < *Rep-bit0 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit0 a* ≤ *Rep-bit0 b*
definition $a < b \longleftrightarrow$ *Rep-bit1 a* < *Rep-bit1 b*
definition $a \leq b \longleftrightarrow$ *Rep-bit1 a* ≤ *Rep-bit1 b*

instance
by(*intro-classes*)
(*auto simp add*: *less-eq-bit0-def less-bit0-def less-eq-bit1-def less-bit1-def Rep-bit0-inject Rep-bit1-inject*)
end

instance *bit0* **and** *bit1* :: (*finite*) *wellorder*
proof −
have *wf* {(*x* :: *'a bit0*, *y*). *x* < *y*}
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS('a bit0*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*

next
have *wf* {(*x* :: *'a bit1*, *y*). *x* < *y*}
by(*auto simp add*: *trancl-def tranclp-less intro*!: *finite-acyclic-wf acyclicI*)
thus *OFCLASS*(*'a bit1*, *wellorder-class*)
by(*rule wf-wellorderI*) *intro-classes*
qed

72.6 Code setup and type classes for code generation

Code setup for *num0* and *num1*

definition *Num0* :: *num0* **where** *Num0* = *Abs-num0 0*
code-datatype *Num0*

instantiation *num0* :: *equal* **begin**
definition *equal-num0* :: *num0* ⇒ *num0* ⇒ *bool*
where *equal-num0* = (=)
instance by *intro-classes* (*simp add*: *equal-num0-def*)
end

lemma *equal-num0-code* [*code*]:
equal-class.equal Num0 Num0 = *True*
by(*rule equal-refl*)

code-datatype *1* :: *num1*

instantiation *num1* :: *equal* **begin**
definition *equal-num1* :: *num1* ⇒ *num1* ⇒ *bool*
where *equal-num1* = (=)
instance by *intro-classes* (*simp add*: *equal-num1-def*)
end

lemma *equal-num1-code* [*code*]:
equal-class.equal (*1* :: *num1*) *1* = *True*
by(*rule equal-refl*)

instantiation *num1* :: *enum* **begin**
definition *enum-class.enum* = [*1* :: *num1*]
definition *enum-class.enum-all P* = *P* (*1* :: *num1*)
definition *enum-class.enum-ex P* = *P* (*1* :: *num1*)
instance
by *intro-classes*
(*auto simp add*: *enum-num1-def enum-all-num1-def enum-ex-num1-def num1-eq-iff Ball-def*)
end

instantiation *num0* **and** *num1* :: *card-UNIV* **begin**
definition *finite-UNIV* = *Phantom*(*num0*) *False*
definition *card-UNIV* = *Phantom*(*num0*) *0*
definition *finite-UNIV* = *Phantom*(*num1*) *True*

definition *card-UNIV* = *Phantom*(*num1*) *1*
instance
by *intro-classes*
(*simp-all add*: *finite-UNIV-num0-def card-UNIV-num0-def infinite-num0 finite-UNIV-num1-def card-UNIV-num1-def*)
end

Code setup for *'a bit0* and *'a bit1*

declare
bit0.Rep-inverse[*code abstype*]
bit0.Rep-0[*code abstract*]
bit0.Rep-1[*code abstract*]

lemma *Abs-bit0'-code* [*code abstract*]:
Rep-bit0 (*Abs-bit0'* *x* :: *'a* :: *finite bit0*) = *x mod int* (*CARD*(*'a bit0*))
by(*auto simp add*: *Abs-bit0'-def intro*!: *Abs-bit0-inverse*)

lemma *inj-on-Abs-bit0*:
inj-on (*Abs-bit0* :: *int* ⇒ *'a bit0*) {*0..<2* * *int CARD*(*'a* :: *finite*)}
by(*auto intro*: *inj-onI simp add*: *Abs-bit0-inject*)

declare
bit1.Rep-inverse[*code abstype*]
bit1.Rep-0[*code abstract*]
bit1.Rep-1[*code abstract*]

lemma *Abs-bit1'-code* [*code abstract*]:
Rep-bit1 (*Abs-bit1'* *x* :: *'a* :: *finite bit1*) = *x mod int* (*CARD*(*'a bit1*))
by(*auto simp add*: *Abs-bit1'-def intro*!: *Abs-bit1-inverse*)

lemma *inj-on-Abs-bit1*:
inj-on (*Abs-bit1* :: *int* ⇒ *'a bit1*) {*0..<1* + *2* * *int CARD*(*'a* :: *finite*)}
by(*auto intro*: *inj-onI simp add*: *Abs-bit1-inject*)

instantiation *bit0* **and** *bit1* :: (*finite*) *equal* **begin**

definition *equal-class.equal x y* ⟷ *Rep-bit0 x* = *Rep-bit0 y*
definition *equal-class.equal x y* ⟷ *Rep-bit1 x* = *Rep-bit1 y*

instance
by *intro-classes* (*simp-all add*: *equal-bit0-def equal-bit1-def Rep-bit0-inject Rep-bit1-inject*)

end

instantiation *bit0* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit0 list*) = *map* (*Abs-bit0'* ∘ *int*) (*upt 0* (*CARD*(*'a bit0*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit0* ∈ *set enum-class.enum*. *P b*)

instance proof
show *distinct* (*enum-class.enum* :: *'a bit0 list*)
by (*simp add*: *enum-bit0-def distinct-map inj-on-def Abs-bit0'-def Abs-bit0-inject*)

let *?Abs* = *Abs-bit0* :: - ⇒ *'a bit0*
interpret *type-definition Rep-bit0 ?Abs* {*0..<2 * int CARD('a)*}
by (*fact type-definition-bit0*)
have *UNIV* = *?Abs* ' {*0..<2 * int CARD('a)*}
by (*simp add*: *Abs-image*)
also have ... = *?Abs* ' (*int* ' {*0..<2 * CARD('a)*})
by (*simp add*: *image-int-atLeastLessThan*)
also have ... = (*?Abs* ∘ *int*) ' {*0..<2 * CARD('a)*}
by (*simp add*: *image-image cong*: *image-cong*)
also have ... = *set enum-class.enum*
by (*simp add*: *enum-bit0-def Abs-bit0'-def cong*: *image-cong-simp*)
finally show *univ-eq*: (*UNIV* :: *'a bit0 set*) = *set enum-class.enum* **.**

fix *P* :: *'a bit0* ⇒ *bool*
show *enum-class.enum-all P* = *Ball UNIV P*
and *enum-class.enum-ex P* = *Bex UNIV P*
by(*simp-all add*: *enum-all-bit0-def enum-ex-bit0-def univ-eq*)
qed

end

instantiation *bit1* :: (*finite*) *enum* **begin**
definition (*enum-class.enum* :: *'a bit1 list*) = *map* (*Abs-bit1'* ∘ *int*) (*upt 0* (*CARD('a bit1*)))
definition *enum-class.enum-all P* = (∀ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)
definition *enum-class.enum-ex P* = (∃ *b* :: *'a bit1* ∈ *set enum-class.enum*. *P b*)

instance
proof(*intro-classes*)
show *distinct* (*enum-class.enum* :: *'a bit1 list*)
by(*simp only*: *Abs-bit1'-def zmod-int*[*symmetric*] *enum-bit1-def distinct-map Suc-eq-plus1 card-bit1 o-apply inj-on-def*)
(*clarsimp simp add*: *Abs-bit1-inject*)

let *?Abs* = *Abs-bit1* :: - ⇒ *'a bit1*
interpret *type-definition Rep-bit1 ?Abs* {*0..<1 + 2 * int CARD('a)*}
by (*fact type-definition-bit1*)
have *UNIV* = *?Abs* ' {*0..<1 + 2 * int CARD('a)*}
by (*simp add*: *Abs-image*)
also have ... = *?Abs* ' (*int* ' {*0..<1 + 2 * CARD('a)*})
by (*simp add*: *image-int-atLeastLessThan*)
also have ... = (*?Abs* ∘ *int*) ' {*0..<1 + 2 * CARD('a)*}
by (*simp add*: *image-image cong*: *image-cong*)
finally show *univ-eq*: (*UNIV* :: *'a bit1 set*) = *set enum-class.enum*

by (*simp only*: *enum-bit1-def set-map set-upt*) (*simp add*: *Abs-bit1'-def cong*: *image-cong-simp*)

fix *P* :: *'a bit1* ⇒ *bool*
show *enum-class.enum-all P* = *Ball UNIV P*
and *enum-class.enum-ex P* = *Bex UNIV P*
by(*simp-all add*: *enum-all-bit1-def enum-ex-bit1-def univ-eq*)
qed

end

instantiation *bit0* **and** *bit1* :: (*finite*) *finite-UNIV* **begin**
definition *finite-UNIV* = *Phantom*('*a bit0*) *True*
definition *finite-UNIV* = *Phantom*('*a bit1*) *True*
instance by *intro-classes* (*simp-all add*: *finite-UNIV-bit0-def finite-UNIV-bit1-def*)
end

instantiation *bit0* **and** *bit1* :: ({*finite*,*card-UNIV*}) *card-UNIV* **begin**
definition *card-UNIV* = *Phantom*('*a bit0*) (*2* ∗ *of-phantom* (*card-UNIV* :: '*a card-UNIV*))
definition *card-UNIV* = *Phantom*('*a bit1*) (*1* + *2* ∗ *of-phantom* (*card-UNIV* :: '*a card-UNIV*))
instance by *intro-classes* (*simp-all add*: *card-UNIV-bit0-def card-UNIV-bit1-def card-UNIV*)
end

72.7 Syntax

syntax
-NumeralType :: *num-token* => *type* (‹(‹*open-block notation*=‹*type-literal number*››-)›)
-NumeralType0 :: *type* (‹(‹*open-block notation*=‹*type-literal number*›› *0*)›)
-NumeralType1 :: *type* (‹(‹*open-block notation*=‹*type-literal number*›› *1*)›)

translations
(*type*) *1* == (*type*) *num1*
(*type*) *0* == (*type*) *num0*

parse-translation ‹
let
fun mk-bintype n =
let
fun mk-bit 0 = *Syntax.const* ***type-syntax***‹*bit0*›
| *mk-bit 1* = *Syntax.const* ***type-syntax***‹*bit1*›;
fun bin-of n =
if n = *1 then Syntax.const* ***type-syntax***‹*num1*›
else if n = *0 then Syntax.const* ***type-syntax***‹*num0*›
else if n = ~*1 then raise TERM* (*negative type numeral*, [])
else

```
        let val (q, r) = Integer.div-mod n 2;
        in mk-bit r $ bin-of q end;
    in bin-of n end;

  fun numeral-tr [Free (str, -)] = mk-bintype (the (Int.fromString str))
    | numeral-tr ts = raise TERM (numeral-tr, ts);

 in [(syntax-const ‹-NumeralType›, K numeral-tr)] end
›

print-translation ‹
 let
  fun int-of [] = 0
    | int-of (b :: bs) = b + 2 * int-of bs;

  fun bin-of (Const (type-syntax ‹num0›, -)) = []
    | bin-of (Const (type-syntax ‹num1›, -)) = [1]
    | bin-of (Const (type-syntax ‹bit0›, -) $ bs) = 0 :: bin-of bs
    | bin-of (Const (type-syntax ‹bit1›, -) $ bs) = 1 :: bin-of bs
    | bin-of t = raise TERM (bin-of, [t]);

  fun bit-tr' b [t] =
      let
        val rev-digs = b :: bin-of t handle TERM - => raise Match
        val i = int-of rev-digs;
        val num = string-of-int (abs i);
        in
        Syntax.const syntax-const ‹-NumeralType› $ Syntax.free num
        end
    | bit-tr' b - = raise Match;
 in
  [(type-syntax ‹bit0›, K (bit-tr' 0)),
   (type-syntax ‹bit1›, K (bit-tr' 1))]
 end
›
```

72.8 Examples

lemma *CARD(0) = 0* **by** *simp*
lemma *CARD(17) = 17* **by** *simp*
lemma *CHAR(23) = 23* **by** *simp*
lemma *8 * 11 ^ 3 - 6 = (2::5)* **by** *simp*

end

73 ω-words

theory *Omega-Words-Fun*

imports *Infinite-Set*
begin

Note: This theory is based on Stefan Merz's work.

Automata recognize languages, which are sets of words. For the theory of ω-automata, we are mostly interested in ω-words, but it is sometimes useful to reason about finite words, too. We are modeling finite words as lists; this lets us benefit from the existing library. Other formalizations could be investigated, such as representing words as functions whose domains are initial intervals of the natural numbers.

73.1 Type declaration and elementary operations

We represent ω-words as functions from the natural numbers to the alphabet type. Other possible formalizations include a coinductive definition or a uniform encoding of finite and infinite words, as studied by Müller et al.

type-synonym
$'a\ word = nat \Rightarrow 'a$

We can prefix a finite word to an ω-word, and a way to obtain an ω-word from a finite, non-empty word is by ω-iteration.

definition
$conc :: ['a\ list, 'a\ word] \Rightarrow 'a\ word$ (**infixr** ‹⌢› *65*)
where $w \frown x == \lambda n.\ if\ n < length\ w\ then\ w!n\ else\ x\ (n - length\ w)$

definition
$iter :: 'a\ list \Rightarrow 'a\ word$ (‹(‹*notation*=‹*postfix* ω››-^ω)› [*1000*])
where $iter\ w == if\ w = []\ then\ undefined\ else\ (\lambda n.\ w!(n\ mod\ (length\ w)))$

lemma *conc-empty*[*simp*]: $[] \frown w = w$
unfolding *conc-def* **by** *auto*

lemma *conc-fst*[*simp*]: $n < length\ w \Longrightarrow (w \frown x)\ n = w!n$
by (*simp add*: *conc-def*)

lemma *conc-snd*[*simp*]: $\neg(n < length\ w) \Longrightarrow (w \frown x)\ n = x\ (n - length\ w)$
by (*simp add*: *conc-def*)

lemma *iter-nth* [*simp*]: $0 < length\ w \Longrightarrow w^\omega\ n = w!(n\ mod\ (length\ w))$
by (*simp add*: *iter-def*)

lemma *conc-conc*[*simp*]: $u \frown v \frown w = (u\ @\ v) \frown w$ (**is** *?lhs = ?rhs*)
proof
fix *n*
have *u*: $n < length\ u \Longrightarrow ?lhs\ n = ?rhs\ n$
by (*simp add*: *conc-def nth-append*)
have *v*: $[\![\ \neg(n < length\ u);\ n < length\ u + length\ v\ ]\!] \Longrightarrow ?lhs\ n = ?rhs\ n$

```
    by (simp add: conc-def nth-append, arith)
  have w: ¬(n < length u + length v) ⟹ ?lhs n = ?rhs n
    by (simp add: conc-def nth-append, arith)
  from u v w show ?lhs n = ?rhs n by blast
qed

lemma range-conc[simp]: range (w₁ ⌢ w₂) = set w₁ ∪ range w₂
proof (intro equalityI subsetI)
  fix a
  assume a ∈ range (w₁ ⌢ w₂)
  then obtain i where 1: a = (w₁ ⌢ w₂) i by auto
  then show a ∈ set w₁ ∪ range w₂
    unfolding 1 by (cases i < length w₁) simp-all
next
  fix a
  assume a: a ∈ set w₁ ∪ range w₂
  then show a ∈ range (w₁ ⌢ w₂)
  proof
    assume a ∈ set w₁
    then obtain i where 1: i < length w₁ a = w₁ ! i
      using in-set-conv-nth by metis
    show ?thesis
    proof
      show a = (w₁ ⌢ w₂) i using 1 by auto
      show i ∈ UNIV by rule
    qed
  next
    assume a ∈ range w₂
    then obtain i where 1: a = w₂ i by auto
    show ?thesis
    proof
      show a = (w₁ ⌢ w₂) (length w₁ + i) using 1 by simp
      show length w₁ + i ∈ UNIV by rule
    qed
  qed
qed


lemma iter-unroll: 0 < length w ⟹ w^ω = w ⌢ w^ω
  by (simp add: fun-eq-iff mod-if)
```

73.2 Subsequence, Prefix, and Suffix

```
definition suffix :: [nat, 'a word] ⇒ 'a word
  where suffix k x ≡ λn. x (k+n)

definition subsequence :: 'a word ⇒ nat ⇒ nat ⇒ 'a list
    (‹(‹open-block notation=‹mixfix subsequence››- [- → -])› 900)
  where subsequence w i j ≡ map w [i..<j]
```

abbreviation *prefix* :: *nat* ⇒ *'a word* ⇒ *'a list*
where *prefix n w* ≡ *subsequence w 0 n*

lemma *suffix-nth* [*simp*]: (*suffix k x*) *n* = *x* (*k*+*n*)
by (*simp add*: *suffix-def*)

lemma *suffix-0* [*simp*]: *suffix 0 x* = *x*
by (*simp add*: *suffix-def*)

lemma *suffix-suffix* [*simp*]: *suffix m* (*suffix k x*) = *suffix* (*k*+*m*) *x*
by (*rule ext*) (*simp add*: *suffix-def add.assoc*)

lemma *subsequence-append*: *prefix* (*i* + *j*) *w* = *prefix i w* @ (*w* [*i* → *i* + *j*])
unfolding *map-append*[*symmetric*] *upt-add-eq-append*[*OF le0*] *subsequence-def* **..**

lemma *subsequence-drop*[*simp*]: *drop i* (*w* [*j* → *k*]) = *w* [*j* + *i* → *k*]
by (*simp add*: *subsequence-def drop-map*)

lemma *subsequence-empty*[*simp*]: *w* [*i* → *j*] = [] ⟷ *j* ≤ *i*
by (*auto simp add*: *subsequence-def*)

lemma *subsequence-length*[*simp*]: *length* (*subsequence w i j*) = *j* − *i*
by (*simp add*: *subsequence-def*)

lemma *subsequence-nth*[*simp*]: *k* < *j* − *i* ⟹ (*w* [*i* → *j*]) ! *k* = *w* (*i* + *k*)
unfolding *subsequence-def*
by *auto*

lemma *subseq-to-zero*[*simp*]: *w*[*i*→*0*] = []
by *simp*

lemma *subseq-to-smaller*[*simp*]: *i*≥*j* ⟹ *w*[*i*→*j*] = []
by *simp*

lemma *subseq-to-Suc*[*simp*]: *i*≤*j* ⟹ *w* [*i* → *Suc j*] = *w* [*i* → *j*] @ [*w j*]
by (*auto simp*: *subsequence-def*)

lemma *subsequence-singleton*[*simp*]: *w* [*i* → *Suc i*] = [*w i*]
by (*auto simp*: *subsequence-def*)

lemma *subsequence-prefix-suffix*: *prefix* (*j* − *i*) (*suffix i w*) = *w* [*i* → *j*]
proof (*cases i* ≤ *j*)
case *True*
have *w* [*i* → *j*] = *map w* (*map* (λ*n*. *n* + *i*) [*0*..<*j* − *i*])
unfolding *map-add-upt subsequence-def*
using *le-add-diff-inverse2*[*OF True*] **by** *force*
also

```
  have ... = map (λn. w (n + i)) [0..<j − i]
    unfolding map-map comp-def by blast
  finally
  show ?thesis
    unfolding subsequence-def suffix-def add.commute[of i] by simp
next
  case False
  then show ?thesis
    by (simp add: subsequence-def)
qed

lemma prefix-suffix: x = prefix n x ⌢ (suffix n x)
  by (rule ext) (simp add: subsequence-def conc-def)

declare prefix-suffix[symmetric, simp]


lemma word-split: obtains v₁ v₂ where v = v₁ ⌢ v₂ length v₁ = k
proof
  show v = prefix k v ⌢ suffix k v
    by (rule prefix-suffix)
  show length (prefix k v) = k
    by simp
qed


lemma set-subsequence[simp]: set (w[i→j]) = w‘{i..<j}
  unfolding subsequence-def by auto

lemma subsequence-take[simp]: take i (w [j → k]) = w [j → min (j + i) k]
  by (simp add: subsequence-def take-map min-def)

lemma subsequence-shift[simp]: (suffix i w) [j → k] = w [i + j → i + k]
  by (metis add-diff-cancel-left subsequence-prefix-suffix suffix-suffix)

lemma suffix-subseq-join[simp]: i ≤ j ⟹ v [i → j] ⌢ suffix j v = suffix i v
  by (metis (no-types, lifting) Nat.add-0-right le-add-diff-inverse prefix-suffix
    subsequence-shift suffix-suffix)

lemma prefix-conc-fst[simp]:
  assumes j ≤ length w
  shows prefix j (w ⌢ w′) = take j w
proof −
  have ∀ i < j. (prefix j (w ⌢ w′)) ! i = (take j w) ! i
    using assms by (simp add: conc-fst subsequence-def)
  thus ?thesis
    by (simp add: assms list-eq-iff-nth-eq min.absorb2)
qed
```

lemma *prefix-conc-snd*[*simp*]:
assumes $n \geq$ *length* u
shows *prefix* n $(u \frown v) = u$ @ *prefix* $(n -$ *length* $u)$ v
proof (*intro nth-equalityI*)
show *length* (*prefix* n $(u \frown v)) =$ *length* $(u$ @ *prefix* $(n -$ *length* $u)$ $v)$
using *assms* **by** *simp*
fix i
assume $i <$ *length* (*prefix* n $(u \frown v))$
then show *prefix* n $(u \frown v)$! $i =$ $(u$ @ *prefix* $(n -$ *length* $u)$ $v)$! i
by (*cases* $i <$ *length* u) (*auto simp*: *nth-append*)
qed

lemma *prefix-conc-length*[*simp*]: *prefix* (*length* w) $(w \frown w') = w$
by *simp*

lemma *suffix-conc-fst*[*simp*]:
assumes $n \leq$ *length* u
shows *suffix* n $(u \frown v) =$ *drop* n $u \frown v$
proof
show *suffix* n $(u \frown v)$ $i =$ (*drop* n $u \frown v)$ i **for** i
using *assms* **by** (*cases* $n + i <$ *length* u) (*auto simp*: *algebra-simps*)
qed

lemma *suffix-conc-snd*[*simp*]:
assumes $n \geq$ *length* u
shows *suffix* n $(u \frown v) =$ *suffix* $(n -$ *length* $u)$ v
proof
show *suffix* n $(u \frown v)$ $i =$ *suffix* $(n -$ *length* $u)$ v i **for** i
using *assms* **by** *simp*
qed

lemma *suffix-conc-length*[*simp*]: *suffix* (*length* w) $(w \frown w') = w'$
unfolding *conc-def* **by** *force*

lemma *concat-eq*[*iff*]:
assumes *length* $v_1 =$ *length* v_2
shows $v_1 \frown u_1 = v_2 \frown u_2 \longleftrightarrow v_1 = v_2 \wedge u_1 = u_2$
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then have *1*: $(v_1 \frown u_1)$ $i = (v_2 \frown u_2)$ i **for** i **by** *auto*
show *?rhs*
proof (*intro conjI ext nth-equalityI*)
show *length* $v_1 =$ *length* v_2 **by** (*rule assms*(*1*))
next
fix i
assume *2*: $i <$ *length* v_1
have *3*: $i <$ *length* v_2 **using** *assms*(*1*) *2* **by** *simp*
show v_1 ! $i = v_2$! i **using** *1*[*of i*] *2 3* **by** *simp*

```
  next
    show u₁ i = u₂ i for i
      using 1[of length v₁ + i] assms(1) by simp
  qed
next
  assume ?rhs
  then show ?lhs by simp
qed

lemma same-concat-eq[iff]: u ⌢ v = u ⌢ w ⟷ v = w
  by simp

lemma comp-concat[simp]: f ∘ u ⌢ v = map f u ⌢ (f ∘ v)
proof
  fix i
  show (f ∘ u ⌢ v) i = (map f u ⌢ (f ∘ v)) i
    by (cases i < length u) simp-all
qed
```

73.3 Prepending

```
primrec build :: 'a ⇒ 'a word ⇒ 'a word  (infixr ‹##› 65)
  where (a ## w) 0 = a | (a ## w) (Suc i) = w i

lemma build-eq[iff]: a₁ ## w₁ = a₂ ## w₂ ⟷ a₁ = a₂ ∧ w₁ = w₂
proof
  assume 1: a₁ ## w₁ = a₂ ## w₂
  have 2: (a₁ ## w₁) i = (a₂ ## w₂) i for i
    using 1 by auto
  show a₁ = a₂ ∧ w₁ = w₂
  proof (intro conjI ext)
    show a₁ = a₂
      using 2[of 0] by simp
    show w₁ i = w₂ i for i
      using 2[of Suc i] by simp
  qed
next
  assume 1: a₁ = a₂ ∧ w₁ = w₂
  show a₁ ## w₁ = a₂ ## w₂ using 1 by simp
qed

lemma build-cons[simp]: (a # u) ⌢ v = a ## u ⌢ v
proof
  fix i
  show ((a # u) ⌢ v) i = (a ## u ⌢ v) i
  proof (cases i)
    case 0
    show ?thesis unfolding 0 by simp
  next
```

```
    case (Suc j)
    show ?thesis unfolding Suc by (cases j < length u, simp+)
  qed
qed

lemma build-append[simp]: (w @ a # u) ⌢ v = w ⌢ a ## u ⌢ v
  unfolding conc-conc[symmetric] by simp

lemma build-first[simp]: w 0 ## suffix (Suc 0) w = w
proof
  show (w 0 ## suffix (Suc 0) w) i = w i for i
    by (cases i) simp-all
qed

lemma build-split[intro]: w = w 0 ## suffix 1 w
  by simp

lemma build-range[simp]: range (a ## w) = insert a (range w)
proof safe
  show (a ## w) i ∉ range w ⟹ (a ## w) i = a for i
    by (cases i) auto
  show a ∈ range (a ## w)
  proof (rule range-eqI)
    show a = (a ## w) 0 by simp
  qed
  show w i ∈ range (a ## w) for i
  proof (rule range-eqI)
    show w i = (a ## w) (Suc i) by simp
  qed
qed

lemma suffix-singleton-suffix[simp]: w i ## suffix (Suc i) w = suffix i w
  using suffix-subseq-join[of i Suc i w]
  by simp
```

Find the first occurrence of a letter from a given set

```
lemma word-first-split-set:
  assumes A ∩ range w ≠ {}
  obtains u a v where w = u ⌢ [a] ⌢ v A ∩ set u = {} a ∈ A
proof −
  define i where i = (LEAST i. w i ∈ A)
  show ?thesis
  proof
    show w = prefix i w ⌢ [w i] ⌢ suffix (Suc i) w
      by simp
    show A ∩ set (prefix i w) = {}
      apply safe
      subgoal premises prems for a
      proof −
```

```
      from prems obtain k where 3: k < i w k = a
        by auto
      have 4: w k ∉ A
        using not-less-Least 3(1) unfolding i-def .
      show ?thesis
        using prems(1) 3(2) 4 by auto
    qed
    done
  show w i ∈ A
    using LeastI assms(1) unfolding i-def by fast
 qed
qed
```

73.4 The limit set of an ω-word

The limit set (also called infinity set) of an ω-word is the set of letters that appear infinitely often in the word. This set plays an important role in defining acceptance conditions of ω-automata.

```
definition limit :: 'a word ⇒ 'a set
  where limit x ≡ {a . ∃∞n . x n = a}

lemma limit-iff-frequent: a ∈ limit x ⟷ (∃∞n . x n = a)
  by (simp add: limit-def)
```

The following is a different way to define the limit, using the reverse image, making the laws about reverse image applicable to the limit set. (Might want to change the definition above?)

```
lemma limit-vimage: (a ∈ limit x) = infinite (x -` {a})
  by (simp add: limit-def Inf-many-def vimage-def)

lemma two-in-limit-iff:
  ({a, b} ⊆ limit x) =
    ((∃n. x n =a ) ∧ (∀n. x n = a ⟶ (∃m>n. x m = b)) ∧ (∀m. x m = b ⟶
(∃n>m. x n = a)))
  (is ?lhs = (?r1 ∧ ?r2 ∧ ?r3))
proof
  assume lhs: ?lhs
  hence 1: ?r1 by (auto simp: limit-def elim: INFM-EX)
  from lhs have ∀n. ∃m>n. x m = b by (auto simp: limit-def INFM-nat)
  hence 2: ?r2 by simp
  from lhs have ∀m. ∃n>m. x n = a by (auto simp: limit-def INFM-nat)
  hence 3: ?r3 by simp
  from 1 2 3 show ?r1 ∧ ?r2 ∧ ?r3 by simp
next
  assume ?r1 ∧ ?r2 ∧ ?r3
  hence 1: ?r1 and 2: ?r2 and 3: ?r3 by simp+
  have infa: ∀m. ∃n≥m. x n = a
  proof
```

```
    fix m
    show ∃ n≥m. x n = a (is ?A m)
    proof (induct m)
      from 1 show ?A 0 by simp
    next
      fix m
      assume ih: ?A m
      then obtain n where n: n ≥ m x n = a by auto
      with 2 obtain k where k: k>n x k = b by auto
      with 3 obtain l where l: l>k x l = a by auto
      from n k l have l ≥ Suc m by auto
      with l show ?A (Suc m) by auto
    qed
  qed
  hence infa': ∃∞n. x n = a by (simp add: INFM-nat-le)
  have ∀ n. ∃ m>n. x m = b
  proof
    fix n
    from infa obtain k where k1: k≥n and k2: x k = a by auto
    from 2 k2 obtain l where l1: l>k and l2: x l = b by auto
    from k1 l1 have l > n by auto
    with l2 show ∃ m>n. x m = b by auto
  qed
  hence ∃∞m. x m = b by (simp add: INFM-nat)
  with infa' show ?lhs by (auto simp: limit-def)
qed
```

For ω-words over a finite alphabet, the limit set is non-empty. Moreover, from some position onward, any such word contains only letters from its limit set.

```
lemma limit-nonempty:
  assumes fin: finite (range x)
  shows ∃ a. a ∈ limit x
proof −
  from fin obtain a where a ∈ range x ∧ infinite (x −‘ {a})
    by (rule inf-img-fin-domE) auto
  hence a ∈ limit x
    by (auto simp add: limit-vimage)
  thus ?thesis ..
qed

lemmas limit-nonemptyE = limit-nonempty[THEN exE]

lemma limit-inter-INF:
  assumes hyp: limit w ∩ S ≠ {}
  shows ∃∞ n. w n ∈ S
proof −
  from hyp obtain x where ∃∞ n. w n = x and x ∈ S
    by (auto simp add: limit-def)
```

thus *?thesis*
by (*auto elim*: *INFM-mono*)
qed

The reverse implication is true only if S is finite.

lemma *INF-limit-inter*:
assumes *hyp*: $\exists_\infty n.\ w\ n \in S$
and *fin*: *finite* $(S \cap range\ w)$
shows $\exists a.\ a \in limit\ w \cap S$
proof (*rule ccontr*)
assume *contra*: $\neg(\exists a.\ a \in limit\ w \cap S)$
hence $\forall a{\in}S.\ finite\ \{n.\ w\ n = a\}$
by (*auto simp add*: *limit-def Inf-many-def*)
with *fin* **have** *finite* $(UN\ a{:}S \cap range\ w.\ \{n.\ w\ n = a\})$
by *auto*
moreover
have $(UN\ a{:}S \cap range\ w.\ \{n.\ w\ n = a\}) = \{n.\ w\ n \in S\}$
by *auto*
moreover
note *hyp*
ultimately show *False*
by (*simp add*: *Inf-many-def*)
qed

lemma *fin-ex-inf-eq-limit*: *finite* $A \Longrightarrow (\exists_\infty i.\ w\ i \in A) \longleftrightarrow limit\ w \cap A \neq \{\}$
by (*metis INF-limit-inter equals0D finite-Int limit-inter-INF*)

lemma *limit-in-range-suffix*: $limit\ x \subseteq range\ (suffix\ k\ x)$
proof
fix *a*
assume $a \in limit\ x$
then obtain *l* **where**
kl: $k < l$ **and** *xl*: $x\ l = a$
by (*auto simp add*: *limit-def INFM-nat*)
from *kl* **obtain** *m* **where** $l = k{+}m$
by (*auto simp add*: *less-iff-Suc-add*)
with *xl* **show** $a \in range\ (suffix\ k\ x)$
by *auto*
qed

lemma *limit-in-range*: $limit\ r \subseteq range\ r$
using *limit-in-range-suffix*[*of r 0*] **by** *simp*

lemmas *limit-in-range-suffixD* = *limit-in-range-suffix*[*THEN subsetD*]

lemma *limit-subset*: $limit\ f \subseteq f\ `\ \{n..\}$
using *limit-in-range-suffix*[*of f n*] **unfolding** *suffix-def* **by** *auto*

theorem *limit-is-suffix*:

assumes *fin*: *finite* (*range x*)
shows $\exists k.$ *limit x* = *range* (*suffix k x*)
proof −
have $\exists k.$ *range* (*suffix k x*) $\subseteq$ *limit x*
proof −
— The set of letters that are not in the limit is certainly finite.
from *fin* **have** *finite* (*range x* − *limit x*)
by *simp*
— Moreover, any such letter occurs only finitely often
moreover
have $\forall a \in$ *range x* − *limit x*. *finite* (*x* −‘ {*a*})
by (*auto simp add*: *limit-vimage*)
— Thus, there are only finitely many occurrences of such letters.
ultimately have *finite* (*UN a* : *range x* − *limit x*. *x* −‘ {*a*})
by (*blast intro*: *finite-UN-I*)
— Therefore these occurrences are within some initial interval.
then obtain *k* **where** (*UN a* : *range x* − *limit x*. *x* −‘ {*a*}) $\subseteq$ {..<*k*}
by (*blast dest*: *finite-nat-bounded*)
— This is just the bound we are looking for.
hence $\forall m.$ *k* $\leq$ *m* $\longrightarrow$ *x m* $\in$ *limit x*
by (*auto simp add*: *limit-vimage*)
hence *range* (*suffix k x*) $\subseteq$ *limit x*
by *auto*
thus *?thesis* **..**
qed
then obtain *k* **where** *range* (*suffix k x*) $\subseteq$ *limit x* **..**
with *limit-in-range-suffix*
have *limit x* = *range* (*suffix k x*)
by (*rule subset-antisym*)
thus *?thesis* **..**
qed

lemmas *limit-is-suffixE* = *limit-is-suffix*[*THEN exE*]

The limit set enjoys some simple algebraic laws with respect to concatenation, suffixes, iteration, and renaming.

theorem *limit-conc* [*simp*]: *limit* (*w* $\frown$ *x*) = *limit x*
proof (*auto*)
fix *a* **assume** *a*: *a* $\in$ *limit* (*w* $\frown$ *x*)
have $\forall m.$ $\exists n.$ *m*<*n* $\wedge$ *x n* = *a*
proof
fix *m*
from *a* **obtain** *n* **where** *m* + *length w* < *n* $\wedge$ (*w* $\frown$ *x*) *n* = *a*
by (*auto simp add*: *limit-def Inf-many-def infinite-nat-iff-unbounded*)
hence *m* < *n* − *length w* $\wedge$ *x* (*n* − *length w*) = *a*
by (*auto simp add*: *conc-def*)
thus $\exists n.$ *m*<*n* $\wedge$ *x n* = *a* **..**
qed
hence *infinite* {*n* . *x n* = *a*}

```
    by (simp add: infinite-nat-iff-unbounded)
  thus a ∈ limit x
    by (simp add: limit-def Inf-many-def)
next
  fix a assume a: a ∈ limit x
  have ∀ m. length w < m ⟶ (∃ n. m<n ∧ (w ⌢ x) n = a)
  proof (clarify)
    fix m
    assume m: length w < m
    with a obtain n where m − length w < n ∧ x n = a
      by (auto simp add: limit-def Inf-many-def infinite-nat-iff-unbounded)
    with m have m < n + length w ∧ (w ⌢ x) (n + length w) = a
      by (simp add: conc-def, arith)
    thus ∃ n. m<n ∧ (w ⌢ x) n = a ..
  qed
  hence infinite {n . (w ⌢ x) n = a}
    by (simp add: unbounded-k-infinite)
  thus a ∈ limit (w ⌢ x)
    by (simp add: limit-def Inf-many-def)
qed

theorem limit-suffix [simp]: limit (suffix n x) = limit x
proof −
  have x = (prefix n x) ⌢ (suffix n x)
    by (simp add: prefix-suffix)
  hence limit x = limit (prefix n x ⌢ suffix n x)
    by simp
  also have ... = limit (suffix n x)
    by (rule limit-conc)
  finally show ?thesis
    by (rule sym)
qed

theorem limit-iter [simp]:
  assumes nempty: 0 < length w
  shows limit w^ω = set w
proof
  have limit w^ω ⊆ range w^ω
    by (auto simp add: limit-def dest: INFM-EX)
  also from nempty have ... ⊆ set w
    by auto
  finally show limit w^ω ⊆ set w .
next
  {
    fix a assume a: a ∈ set w
    then obtain k where k: k < length w ∧ w!k = a
      by (auto simp add: set-conv-nth)
    — the following bound is terrible, but it simplifies the proof
    from nempty k have ∀ m. w^ω ((Suc m)*(length w) + k) = a
```

```
    by (simp add: mod-add-left-eq [symmetric])
  moreover
  — why is the following so hard to prove??
  have ∀ m. m < (Suc m)*(length w) + k
  proof
    fix m
    from nempty have 1 ≤ length w by arith
    hence m*1 ≤ m*length w by simp
    hence m ≤ m*length w by simp
    with nempty have m < length w + (m*length w) + k by arith
    thus m < (Suc m)*(length w) + k by simp
  qed
  moreover note nempty
  ultimately have a ∈ limit w^ω
    by (auto simp add: limit-iff-frequent INFM-nat)
 }
 then show set w ⊆ limit w^ω by auto
qed

lemma limit-o [simp]:
 assumes a: a ∈ limit w
 shows f a ∈ limit (f ∘ w)
proof −
 from a
 have ∃_∞n. w n = a
   by (simp add: limit-iff-frequent)
 hence ∃_∞n. f (w n) = f a
   by (rule INFM-mono, simp)
 thus f a ∈ limit (f ∘ w)
   by (simp add: limit-iff-frequent)
qed
```

The converse relation is not true in general: $f(a)$ can be in the limit of $f \circ w$ even though a is not in the limit of w. However, *limit* commutes with renaming if the function is injective. More generally, if $f(a)$ is the image of only finitely many elements, some of these must be in the limit of w.

```
lemma limit-o-inv:
 assumes fin: finite (f −‘ {x})
   and x: x ∈ limit (f ∘ w)
 shows ∃ a ∈ (f −‘ {x}). a ∈ limit w
proof (rule ccontr)
 assume contra: ¬ ?thesis
 — hence, every element in the pre-image occurs only finitely often
 then have ∀ a ∈ (f −‘ {x}). finite {n. w n = a}
   by (simp add: limit-def Inf-many-def)
 — so there are only finitely many occurrences of any such element
 with fin have finite (⋃ a ∈ (f −‘ {x}). {n. w n = a})
   by auto
 — these are precisely those positions where x occurs in f ∘ w
```

```
  moreover
  have (⋃ a ∈ (f −‘ {x}). {n. w n = a}) = {n. f(w n) = x}
    by auto
  ultimately
  — so x can occur only finitely often in the translated word
  have finite {n. f(w n) = x}
    by simp
  — ... which yields a contradiction
  with x show False
    by (simp add: limit-def Inf-many-def)
qed

theorem limit-inj [simp]:
  assumes inj: inj f
  shows limit (f ∘ w) = f ‘ (limit w)
proof
  show f ‘ limit w ⊆ limit (f ∘ w)
    by auto
  show limit (f ∘ w) ⊆ f ‘ limit w
  proof
    fix x
    assume x: x ∈ limit (f ∘ w)
    from inj have finite (f −‘ {x})
      by (blast intro: finite-vimageI)
    with x obtain a where a: a ∈ (f −‘ {x}) ∧ a ∈ limit w
      by (blast dest: limit-o-inv)
    thus x ∈ f ‘ (limit w)
      by auto
  qed
qed

lemma limit-inter-empty:
  assumes fin: finite (range w)
  assumes hyp: limit w ∩ S = {}
  shows ∀∞n. w n ∉ S
proof −
  from fin obtain k where k-def: limit w = range (suffix k w)
    using limit-is-suffix by blast
  have w (k + k') ∉ S for k'
    using hyp unfolding k-def suffix-def image-def by blast
  thus ?thesis
    unfolding MOST-nat-le using le-Suc-ex by blast
qed
```

If the limit is the suffix of the sequence's range, we may increase the suffix index arbitrarily

```
lemma limit-range-suffix-incr:
  assumes limit r = range (suffix i r)
  assumes j≥i
```

shows *limit r = range (suffix j r)*
(**is** *?lhs = ?rhs*)
proof −
have *?lhs = range (suffix i r)*
using *assms* **by** *simp*
moreover
have ... ⊇ *?rhs* **using** ‹*j*≥*i*›
by (*metis* (*mono-tags*, *lifting*) *assms*(*2*)
image-subsetI le-Suc-ex range-eqI suffix-def suffix-suffix)
moreover
have ... ⊇ *?lhs* **by** (*rule limit-in-range-suffix*)
ultimately
show *?lhs = ?rhs*
by (*metis antisym-conv limit-in-range-suffix*)
qed

For two finite sequences, we can find a common suffix index such that the limits can be represented as these suffixes' ranges.

lemma *common-range-limit*:
assumes *finite* (*range x*)
and *finite* (*range y*)
obtains *i* **where** *limit x = range* (*suffix i x*)
and *limit y = range* (*suffix i y*)
proof −
obtain *i j* **where** *1*: *limit x = range* (*suffix i x*)
and *2*: *limit y = range* (*suffix j y*)
using *assms limit-is-suffix* **by** *metis*
have *limit x = range* (*suffix* (*max i j*) *x*)
and *limit y = range* (*suffix* (*max i j*) *y*)
using *limit-range-suffix-incr*[*OF 1*] *limit-range-suffix-incr*[*OF 2*]
by *auto*
thus *?thesis*
using *that* **by** *metis*
qed

73.5 Index sequences and piecewise definitions

A word can be defined piecewise: given a sequence of words $w_0, w_1, \ldots$ and a strictly increasing sequence of integers $i_0, i_1, \ldots$ where $i_0 = 0$, a single word is obtained by concatenating subwords of the w_n as given by the integers: the resulting word is

$$(w_0)_{i_0} \ldots (w_0)_{i_1-1} (w_1)_{i_1} \ldots (w_1)_{i_2-1} \ldots$$

We prepare the field by proving some trivial facts about such sequences of indexes.

definition *idx-sequence* :: *nat word* ⇒ *bool*
where *idx-sequence idx* ≡ (*idx 0 = 0*) ∧ (∀ *n*. *idx n* < *idx* (*Suc n*))

lemma *idx-sequence-less*:
assumes *iseq*: *idx-sequence idx*
shows *idx n* < *idx* (*Suc*(*n*+*k*))
proof (*induct k*)
from *iseq* **show** *idx n* < *idx* (*Suc* (*n* + *0*))
by (*simp add*: *idx-sequence-def*)
next
fix *k*
assume *ih*: *idx n* < *idx* (*Suc*(*n*+*k*))
from *iseq* **have** *idx* (*Suc*(*n*+*k*)) < *idx* (*Suc*(*n* + *Suc k*))
by (*simp add*: *idx-sequence-def*)
with *ih* **show** *idx n* < *idx* (*Suc*(*n* + *Suc k*))
by (*rule less-trans*)
qed

lemma *idx-sequence-inj*:
assumes *iseq*: *idx-sequence idx*
and *eq*: *idx m* = *idx n*
shows *m* = *n*
proof (*cases m n rule*: *linorder-cases*)
case *greater*
then obtain *k* **where** *m* = *Suc*(*n*+*k*)
by (*auto simp add*: *less-iff-Suc-add*)
with *iseq* **have** *idx n* < *idx m*
by (*simp add*: *idx-sequence-less*)
with *eq* **show** *?thesis*
by *simp*
next
case *less*
then obtain *k* **where** *n* = *Suc*(*m*+*k*)
by (*auto simp add*: *less-iff-Suc-add*)
with *iseq* **have** *idx m* < *idx n*
by (*simp add*: *idx-sequence-less*)
with *eq* **show** *?thesis*
by *simp*
qed

lemma *idx-sequence-mono*:
assumes *iseq*: *idx-sequence idx*
and *m*: *m* ≤ *n*
shows *idx m* ≤ *idx n*
proof (*cases m*=*n*)
case *True*
thus *?thesis* **by** *simp*
next
case *False*
with *m* **have** *m* < *n* **by** *simp*
then obtain *k* **where** *n* = *Suc*(*m*+*k*)

```
    by (auto simp add: less-iff-Suc-add)
  with iseq have idx m < idx n
    by (simp add: idx-sequence-less)
  thus ?thesis by simp
qed
```

Given an index sequence, every natural number is contained in the interval defined by two adjacent indexes, and in fact this interval is determined uniquely.

```
lemma idx-sequence-idx:
  assumes idx-sequence idx
  shows idx k ∈ {idx k ..< idx (Suc k)}
using assms by (auto simp add: idx-sequence-def)

lemma idx-sequence-interval:
  assumes iseq: idx-sequence idx
  shows ∃ k. n ∈ {idx k ..< idx (Suc k) }
    (is ?P n is ∃ k. ?in n k)
proof (induct n)
  from iseq have 0 = idx 0
    by (simp add: idx-sequence-def)
  moreover
  from iseq have idx 0 ∈ {idx 0 ..< idx (Suc 0) }
    by (rule idx-sequence-idx)
  ultimately
  show ?P 0 by auto
next
  fix n
  assume ?P n
  then obtain k where k: ?in n k ..
  show ?P (Suc n)
  proof (cases Suc n < idx (Suc k))
    case True
    with k have ?in (Suc n) k
      by simp
    thus ?thesis ..
  next
    case False
    with k have Suc n = idx (Suc k)
      by auto
    with iseq have ?in (Suc n) (Suc k)
      by (simp add: idx-sequence-def)
    thus ?thesis ..
  qed
qed

lemma idx-sequence-interval-unique:
  assumes iseq: idx-sequence idx
    and k: n ∈ {idx k ..< idx (Suc k)}
```

```
    and m: n ∈ {idx m ..< idx (Suc m)}
  shows k = m
proof (cases k m rule: linorder-cases)
  case less
  hence Suc k ≤ m by simp
  with iseq have idx (Suc k) ≤ idx m
    by (rule idx-sequence-mono)
  with m have idx (Suc k) ≤ n
    by auto
  with k have False
    by simp
  thus ?thesis ..
next
  case greater
  hence Suc m ≤ k by simp
  with iseq have idx (Suc m) ≤ idx k
    by (rule idx-sequence-mono)
  with k have idx (Suc m) ≤ n
    by auto
  with m have False
    by simp
  thus ?thesis ..
qed

lemma idx-sequence-unique-interval:
  assumes iseq: idx-sequence idx
  shows ∃! k. n ∈ {idx k ..< idx (Suc k) }
proof (rule ex-ex1I)
  from iseq show ∃ k. n ∈ {idx k ..< idx (Suc k)}
    by (rule idx-sequence-interval)
next
  fix k y
  assume n ∈ {idx k..<idx (Suc k)} and n ∈ {idx y..<idx (Suc y)}
  with iseq show k = y by (auto elim: idx-sequence-interval-unique)
qed
```

Now we can define the piecewise construction of a word using an index sequence.

```
definition merge :: 'a word word ⇒ nat word ⇒ 'a word
  where merge ws idx ≡ λn. let i = THE i. n ∈ {idx i ..< idx (Suc i) } in ws i n

lemma merge:
  assumes idx: idx-sequence idx
    and n: n ∈ {idx i ..< idx (Suc i)}
  shows merge ws idx n = ws i n
proof −
  from n have (THE k. n ∈ {idx k ..< idx (Suc k) }) = i
     by (rule the-equality[OF - sym[OF idx-sequence-interval-unique[OF idx n]]])
simp
```

thus *?thesis*
by (*simp add*: *merge-def Let-def*)
qed

lemma *merge0*:
assumes *idx*: *idx-sequence idx*
shows *merge ws idx 0 = ws 0 0*
proof (*rule merge*[*OF idx*])
from *idx* **have** *idx 0 < idx* (*Suc 0*)
unfolding *idx-sequence-def* **by** *blast*
with *idx* **show** *0* ∈ {*idx 0 ..< idx* (*Suc 0*)}
by (*simp add*: *idx-sequence-def*)
qed

lemma *merge-Suc*:
assumes *idx*: *idx-sequence idx*
and *n*: *n* ∈ {*idx i ..< idx* (*Suc i*)}
shows *merge ws idx* (*Suc n*) = (*if Suc n = idx* (*Suc i*) *then ws* (*Suc i*) *else ws i*) (*Suc n*)
proof *auto*
assume *eq*: *Suc n = idx* (*Suc i*)
from *idx* **have** *idx* (*Suc i*) < *idx* (*Suc*(*Suc i*))
unfolding *idx-sequence-def* **by** *blast*
with *eq idx* **show** *merge ws idx* (*idx* (*Suc i*)) = *ws* (*Suc i*) (*idx* (*Suc i*))
by (*simp add*: *merge*)
next
assume *neq*: *Suc n* ≠ *idx* (*Suc i*)
with *n* **have** *Suc n* ∈ {*idx i ..< idx* (*Suc i*) }
by *auto*
with *idx* **show** *merge ws idx* (*Suc n*) = *ws i* (*Suc n*)
by (*rule merge*)
qed

end

74 Combinator syntax for generic, open state monads (single-threaded monads)

theory *Open-State-Syntax*
imports *Main*
begin

context
includes *state-combinator-syntax*
begin

74.1 Motivation

The logic HOL has no notion of constructor classes, so it is not possible to model monads the Haskell way in full genericity in Isabelle/HOL.

However, this theory provides substantial support for a very common class of monads: *state monads* (or *single-threaded monads*, since a state is transformed single-threadedly).

To enter from the Haskell world, https://www.engr.mun.ca/~theo/Misc/haskell_and_monads.htm makes a good motivating start. Here we just sketch briefly how those monads enter the game of Isabelle/HOL.

74.2 State transformations and combinators

We classify functions operating on states into two categories:

transformations with type signature $\sigma \Rightarrow \sigma'$, transforming a state.

"yielding" transformations with type signature $\sigma \Rightarrow \alpha \times \sigma'$, "yielding" a side result while transforming a state.

queries with type signature $\sigma \Rightarrow \alpha$, computing a result dependent on a state.

By convention we write σ for types representing states and α, β, γ, ... for types representing side results. Type changes due to transformations are not excluded in our scenario.

We aim to assert that values of any state type σ are used in a single-threaded way: after application of a transformation on a value of type σ, the former value should not be used again. To achieve this, we use a set of monad combinators:

Given two transformations f and g, they may be directly composed using the $(\circ>)$ combinator, forming a forward composition: $(f \circ> g)\ s = f\ (g\ s)$.

After any yielding transformation, we bind the side result immediately using a lambda abstraction. This is the purpose of the $(\circ\rightarrow)$ combinator: $(f \circ\rightarrow (\lambda x.\ g))\ s = (let\ (x,\ s') = f\ s\ in\ g\ s')$.

For queries, the existing *Let* is appropriate.

Naturally, a computation may yield a side result by pairing it to the state from the left; we introduce the suggestive abbreviation *return* for this purpose.

The most crucial distinction to Haskell is that we do not need to introduce distinguished type constructors for different kinds of state. This has two consequences:

- The monad model does not state anything about the kind of state; the model for the state is completely orthogonal and may be specified completely independently.

- There is no distinguished type constructor encapsulating away the state transformation, i.e. transformations may be applied directly without using any lifting or providing and dropping units ("open monad").
- The type of states may change due to a transformation.

74.3 Monad laws

The common monadic laws hold and may also be used as normalization rules for monadic expressions:

lemmas *monad-simp* = *Pair-scomp scomp-Pair id-fcomp fcomp-id scomp-scomp scomp-fcomp fcomp-scomp fcomp-assoc*

Evaluation of monadic expressions by force:

lemmas *monad-collapse* = *monad-simp fcomp-apply scomp-apply split-beta*

end

74.4 Do-syntax

nonterminal *sdo-binds* **and** *sdo-bind*

syntax
-sdo-block :: *sdo-binds* ⇒ *'a*
(‹(‹*open-block notation=‹mixfix exec block›› exec* {//(*2* -)//})› [*12*] *62*)
-sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind*
(‹(‹*indent=2 notation=‹infix exec bind››* - ←/ -)› *13*)
-sdo-let :: [*pttrn*, *'a*] ⇒ *sdo-bind*
(‹(‹*indent=2 notation=‹infix exec let›› let* - =/ -)› [*1000*, *13*] *13*)
-sdo-then :: *'a* ⇒ *sdo-bind* (‹-› [*14*] *13*)
-sdo-final :: *'a* ⇒ *sdo-binds* (‹-›)
-sdo-cons :: [*sdo-bind*, *sdo-binds*] ⇒ *sdo-binds*
(‹(‹*open-block notation=‹infix exec next››*-;//-)› [*13*, *12*] *12*)

syntax (*ASCII*)
-sdo-bind :: [*pttrn*, *'a*] ⇒ *sdo-bind*
(‹(‹*indent=2 notation=‹infix exec bind››* - <−/ -)› *13*)

syntax-consts
-sdo-let == *Let*

translations
-sdo-block (*-sdo-cons* (*-sdo-bind p t*) (*-sdo-final e*))
== *CONST scomp t* (λ*p*. *e*)
-sdo-block (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*))
=> *CONST fcomp t e*
-sdo-final (*-sdo-block* (*-sdo-cons* (*-sdo-then t*) (*-sdo-final e*)))

<= -sdo-final (CONST fcomp t e)
-sdo-block (-sdo-cons (-sdo-then t) e)
<= CONST fcomp t (-sdo-block e)
-sdo-block (-sdo-cons (-sdo-let p t) bs)
== let p = t in -sdo-block bs
-sdo-block (-sdo-cons b (-sdo-cons c cs))
== -sdo-block (-sdo-cons b (-sdo-final (-sdo-block (-sdo-cons c cs))))
-sdo-cons (-sdo-let p t) (-sdo-final s)
== -sdo-final (let p = t in s)
-sdo-block (-sdo-final e) => e

For an example, see `~~/src/HOL/Proofs/Extraction/Higman_Extraction.thy`.

end

75 Canonical order on option type

theory *Option-ord*
imports *Main*
begin

unbundle *lattice-syntax*

instantiation *option* :: (*preorder*) *preorder*
begin

definition *less-eq-option* **where**
$x \leq y \longleftrightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ $x \leq y$))

definition *less-option* **where**
$x < y \longleftrightarrow$ (*case y of None* $\Rightarrow$ *False* | *Some y* $\Rightarrow$ (*case x of None* $\Rightarrow$ *True* | *Some x* $\Rightarrow$ $x < y$))

lemma *less-eq-option-None* [*simp*]: *None* $\leq x$
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-None-code* [*code*]: *None* $\leq x \longleftrightarrow$ *True*
by *simp*

lemma *less-eq-option-None-is-None*: $x \leq$ *None* $\Longrightarrow x =$ *None*
by (*cases x*) (*simp-all add*: *less-eq-option-def*)

lemma *less-eq-option-Some-None* [*simp*, *code*]: *Some x* $\leq$ *None* $\longleftrightarrow$ *False*
by (*simp add*: *less-eq-option-def*)

lemma *less-eq-option-Some* [*simp*, *code*]: *Some x* $\leq$ *Some y* $\longleftrightarrow x \leq y$
by (*simp add*: *less-eq-option-def*)

lemma *less-option-None* [*simp*, *code*]: $x <$ *None* $\longleftrightarrow$ *False*

by (*simp add*: *less-option-def*)

lemma *less-option-None-is-Some*: $None < x \Longrightarrow \exists\, z.\ x = Some\ z$
by (*cases x*) (*simp-all add*: *less-option-def*)

lemma *less-option-None-Some* [*simp*]: $None < Some\ x$
by (*simp add*: *less-option-def*)

lemma *less-option-None-Some-code* [*code*]: $None < Some\ x \longleftrightarrow True$
by *simp*

lemma *less-option-Some* [*simp*, *code*]: $Some\ x < Some\ y \longleftrightarrow x < y$
by (*simp add*: *less-option-def*)

instance
by *standard*
(*auto simp add*: *less-eq-option-def less-option-def less-le-not-le*
elim: *order-trans split*: *option.splits*)

end

instance *option* :: (*order*) *order*
by *standard* (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instance *option* :: (*linorder*) *linorder*
by *standard* (*auto simp add*: *less-eq-option-def less-option-def split*: *option.splits*)

instantiation *option* :: (*order*) *order-bot*
begin

definition *bot-option* **where** $\bot = None$

instance
by *standard* (*simp add*: *bot-option-def*)

end

instantiation *option* :: (*order-top*) *order-top*
begin

definition *top-option* **where** $\top = Some\ \top$

instance
by *standard* (*simp add*: *top-option-def less-eq-option-def split*: *option.split*)

end

instance *option* :: (*wellorder*) *wellorder*
proof

```
  fix P :: 'a option ⇒ bool
  fix z :: 'a option
  assume H: ⋀x. (⋀y. y < x ⟹ P y) ⟹ P x
  have P None by (rule H) simp
  then have P-Some [case-names Some]: P z if ⋀x. z = Some x ⟹ (P ∘ Some)
x for z
    using ‹P None› that by (cases z) simp-all
  show P z
  proof (cases z rule: P-Some)
    case (Some w)
    show (P ∘ Some) w
    proof (induct rule: less-induct)
      case (less x)
      have P (Some x)
      proof (rule H)
        fix y :: 'a option
        assume y < Some x
        show P y
        proof (cases y rule: P-Some)
          case (Some v)
          with ‹y < Some x› have v < x by simp
          with less show (P ∘ Some) v .
        qed
      qed
      then show ?case by simp
    qed
  qed
qed

instantiation option :: (inf) inf
begin

definition inf-option where
  x ⊓ y = (case x of None ⇒ None | Some x ⇒ (case y of None ⇒ None | Some
y ⇒ Some (x ⊓ y)))

lemma inf-None-1 [simp, code]: None ⊓ y = None
  by (simp add: inf-option-def)

lemma inf-None-2 [simp, code]: x ⊓ None = None
  by (cases x) (simp-all add: inf-option-def)

lemma inf-Some [simp, code]: Some x ⊓ Some y = Some (x ⊓ y)
  by (simp add: inf-option-def)

instance ..

end
```

instantiation *option* :: (*sup*) *sup*
begin

definition *sup-option* **where**
$x \sqcup y = (\mathit{case}\ x\ \mathit{of}\ \mathit{None} \Rightarrow y \mid \mathit{Some}\ x' \Rightarrow (\mathit{case}\ y\ \mathit{of}\ \mathit{None} \Rightarrow x \mid \mathit{Some}\ y \Rightarrow \mathit{Some}\ (x' \sqcup y)))$

lemma *sup-None-1* [*simp*, *code*]: $\mathit{None} \sqcup y = y$
by (*simp add*: *sup-option-def*)

lemma *sup-None-2* [*simp*, *code*]: $x \sqcup \mathit{None} = x$
by (*cases x*) (*simp-all add*: *sup-option-def*)

lemma *sup-Some* [*simp*, *code*]: $\mathit{Some}\ x \sqcup \mathit{Some}\ y = \mathit{Some}\ (x \sqcup y)$
by (*simp add*: *sup-option-def*)

instance ..

end

instance *option* :: (*semilattice-inf*) *semilattice-inf*
proof
fix $x\ y\ z$ :: *'a option*
show $x \sqcap y \leq x$
by (*cases x*, *simp-all*, *cases y*, *simp-all*)
show $x \sqcap y \leq y$
by (*cases x*, *simp-all*, *cases y*, *simp-all*)
show $x \leq y \Longrightarrow x \leq z \Longrightarrow x \leq y \sqcap z$
by (*cases x*, *simp-all*, *cases y*, *simp-all*, *cases z*, *simp-all*)
qed

instance *option* :: (*semilattice-sup*) *semilattice-sup*
proof
fix $x\ y\ z$ :: *'a option*
show $x \leq x \sqcup y$
by (*cases x*, *simp-all*, *cases y*, *simp-all*)
show $y \leq x \sqcup y$
by (*cases x*, *simp-all*, *cases y*, *simp-all*)
fix $x\ y\ z$ :: *'a option*
show $y \leq x \Longrightarrow z \leq x \Longrightarrow y \sqcup z \leq x$
by (*cases y*, *simp-all*, *cases z*, *simp-all*, *cases x*, *simp-all*)
qed

instance *option* :: (*lattice*) *lattice* ..

instance *option* :: (*lattice*) *bounded-lattice-bot* ..

instance *option* :: (*bounded-lattice-top*) *bounded-lattice-top* ..

instance *option* :: (*bounded-lattice-top*) *bounded-lattice* **..**

instance *option* :: (*distrib-lattice*) *distrib-lattice*
proof
fix *x y z* :: *'a option*
show $x \sqcup y \sqcap z = (x \sqcup y) \sqcap (x \sqcup z)$
by (*cases x*, *simp-all*, *cases y*, *simp-all*, *cases z*, *simp-all add*: *sup-inf-distrib1 inf-commute*)
qed

instantiation *option* :: (*complete-lattice*) *complete-lattice*
begin

definition *Inf-option* :: *'a option set* ⇒ *'a option* **where**
$\bigsqcap A = (\textit{if } \textit{None} \in A \textit{ then None else Some } (\bigsqcap \textit{Option.these } A))$

lemma *None-in-Inf* [*simp*]: $\textit{None} \in A \Longrightarrow \bigsqcap A = \textit{None}$
by (*simp add*: *Inf-option-def*)

definition *Sup-option* :: *'a option set* ⇒ *'a option* **where**
$\bigsqcup A = (\textit{if } A = \{\} \vee A = \{\textit{None}\} \textit{ then None else Some } (\bigsqcup \textit{Option.these } A))$

lemma *empty-Sup* [*simp*]: $\bigsqcup \{\} = \textit{None}$
by (*simp add*: *Sup-option-def*)

lemma *singleton-None-Sup* [*simp*]: $\bigsqcup \{\textit{None}\} = \textit{None}$
by (*simp add*: *Sup-option-def*)

instance
proof
fix *x* :: *'a option* **and** *A*
assume $x \in A$
then show $\bigsqcap A \leq x$
by (*cases x*) (*auto simp add*: *Inf-option-def in-these-eq intro*: *Inf-lower*)
next
fix *z* :: *'a option* **and** *A*
assume ∗: $\bigwedge x.\ x \in A \Longrightarrow z \leq x$
show $z \leq \bigsqcap A$
proof (*cases z*)
case *None* **then show** *?thesis* **by** *simp*
next
case (*Some y*)
show *?thesis*
by (*auto simp add*: *Inf-option-def in-these-eq Some intro*!: *Inf-greatest dest*!: ∗)
qed
next
fix *x* :: *'a option* **and** *A*
assume $x \in A$

```
  then show x ≤ ⨆A
    by (cases x) (auto simp add: Sup-option-def in-these-eq intro: Sup-upper)
next
  fix z :: 'a option and A
  assume *: ⋀x. x ∈ A ⟹ x ≤ z
  show ⨆A ≤ z
  proof (cases z)
    case None
   with * have ⋀x. x ∈ A ⟹ x = None by (auto dest: less-eq-option-None-is-None)
    then have A = {} ∨ A = {None} by blast
    then show ?thesis by (simp add: Sup-option-def)
  next
    case (Some y)
    from * have ⋀w. Some w ∈ A ⟹ Some w ≤ z .
    with Some have ⋀w. w ∈ Option.these A ⟹ w ≤ y
      by (simp add: in-these-eq)
    then have ⨆Option.these A ≤ y by (rule Sup-least)
    with Some show ?thesis by (simp add: Sup-option-def)
  qed
next
  show ⨆{} = (⊥::'a option)
    by (auto simp: bot-option-def)
  show ⨅{} = (⊤::'a option)
    by (auto simp: top-option-def Inf-option-def)
qed

end

lemma Some-Inf:
  Some (⨅A) = ⨅(Some ‘ A)
  by (auto simp add: Inf-option-def)

lemma Some-Sup:
  A ≠ {} ⟹ Some (⨆A) = ⨆(Some ‘ A)
  by (auto simp add: Sup-option-def)

lemma Some-INF:
  Some (⨅x∈A. f x) = (⨅x∈A. Some (f x))
  by (simp add: Some-Inf image-comp)

lemma Some-SUP:
  A ≠ {} ⟹ Some (⨆x∈A. f x) = (⨆x∈A. Some (f x))
  by (simp add: Some-Sup image-comp)

lemma option-Inf-Sup: ⨅(Sup ‘ A) ≤ ⨆(Inf ‘ {f ‘ A |f. ∀ Y∈A. f Y ∈ Y})
  for A :: ('a::complete-distrib-lattice option) set set
proof (cases {} ∈ A)
  case True
  then show ?thesis
```

by (*rule INF-lower2*, *simp-all*)
next
case *False*
from *this* **have** *X*: $\{\} \notin A$
by *simp*
then show *?thesis*
proof (*cases* $\{None\} \in A$)
case *True*
then show *?thesis*
by (*rule INF-lower2*, *simp-all*)
next
case *False*

{fix *y*
assume *A*: $y \in A$
have *Sup* $(y - \{None\}) = Sup\ y$
by (*metis* (*no-types*, *lifting*) *Sup-option-def insert-Diff-single these-insert-None these-not-empty-eq*)
from *A* **and** *this* **have** $(\exists z.\ y - \{None\} = z - \{None\} \wedge z \in A) \wedge \bigsqcup y = \bigsqcup (y - \{None\})$
by *auto*
}
from *this* **have** *A*: *Sup* ' $A = (Sup\ `\{y - \{None\} \mid y.\ y \in A\})$
by (*auto simp add*: *image-def*)

have [*simp*]: $\bigwedge y.\ y \in A \Longrightarrow \exists ya.\ \{ya.\ \exists x.\ x \in y \wedge (\exists y.\ x = Some\ y) \wedge ya = the\ x\}$
$= \{y.\ \exists x \in ya - \{None\}.\ y = the\ x\} \wedge ya \in A$
by (*rule exI*, *auto*)

have [*simp*]: $\bigwedge y.\ y \in A \Longrightarrow$
$(\exists ya.\ y - \{None\} = ya - \{None\} \wedge ya \in A) \wedge \bigsqcup \{ya.\ \exists x \in y - \{None\}.\ ya = the\ x\}$
$= \bigsqcup \{ya.\ \exists x.\ x \in y \wedge (\exists y.\ x = Some\ y) \wedge ya = the\ x\}$
apply (*safe*, *blast*)
by (*rule arg-cong* [*of* - - *Sup*], *auto*)
{fix *y*
assume [*simp*]: $y \in A$
have $\exists x.\ (\exists y.\ x = \{ya.\ \exists x \in y - \{None\}.\ ya = the\ x\} \wedge y \in A) \wedge \bigsqcup \{ya.\ \exists x.\ x \in y \wedge (\exists y.\ x = Some\ y) \wedge ya = the\ x\} = \bigsqcup x$
and $\exists x.\ (\exists y.\ x = y - \{None\} \wedge y \in A) \wedge \bigsqcup \{ya.\ \exists x \in y - \{None\}.\ ya = the\ x\} = \bigsqcup \{y.\ \exists xa.\ xa \in x \wedge (\exists y.\ xa = Some\ y) \wedge y = the\ xa\}$
apply (*rule exI* [*of* - $\{ya.\ \exists x.\ x \in y \wedge (\exists y.\ x = Some\ y) \wedge ya = the\ x\}$], *simp*)
by (*rule exI* [*of* - $y - \{None\}$], *simp*)
}
from *this* **have** *C*: $(\lambda x.\ (\bigsqcup Option.these\ x))\ `\{y - \{None\} \mid y.\ y \in A\} = (Sup\ `\{the\ `(y - \{None\}) \mid y.\ y \in A\})$
by (*simp add*: *image-def Option.these-def*, *safe*, *simp-all*)

have *D*: $\forall\ f\ .\ \exists\ Y{\in}A.\ f\ Y \notin Y \Longrightarrow \mathit{False}$
by (*drule spec* [*of* - $\lambda\ Y\ .\ \mathit{SOME}\ x\ .\ x \in Y$], *simp add*: *X some-in-eq*)

define *F* **where** $F = (\lambda\ Y\ .\ \mathit{SOME}\ x{::}'a\ \mathit{option}\ .\ x \in (Y - \{\mathit{None}\}))$

have *G*: $\bigwedge\ Y\ .\ Y \in A \Longrightarrow \exists\ x\ .\ x \in Y - \{\mathit{None}\}$
by (*metis False X all-not-in-conv insert-Diff-single these-insert-None these-not-empty-eq*)

have *F*: $\bigwedge\ Y\ .\ Y \in A \Longrightarrow F\ Y \in (Y - \{\mathit{None}\})$
by (*metis F-def G empty-iff some-in-eq*)

have $\mathit{Some}\ \bot \leq \mathit{Inf}\ (F\ `\ A)$
by (*metis* (*no-types*, *lifting*) *Diff-iff F Inf-option-def bot.extremum image-iff less-eq-option-Some singletonI*)

from *this* **have** $\mathit{Inf}\ (F\ `\ A) \neq \mathit{None}$
by (*cases* $\prod x{\in}A.\ F\ x$, *simp-all*)

from *this* **have** $\mathit{Inf}\ (F\ `\ A) \neq \mathit{None} \wedge \mathit{Inf}\ (F\ `\ A) \in \mathit{Inf}\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\}$
using *F* **by** *auto*

from *this* **have** $\exists\ x\ .\ x \neq \mathit{None} \wedge x \in \mathit{Inf}\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\}$
by *blast*

from *this* **have** *E*: $\mathit{Inf}\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\} = \{\mathit{None}\} \Longrightarrow \mathit{False}$
by *blast*

have [*simp*]: $((\bigsqcup x{\in}\{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\}.\ \prod x) = \mathit{None}) = \mathit{False}$
by (*metis* (*no-types*, *lifting*) *E Sup-option-def* ‹$\exists\ x.\ x \neq \mathit{None} \wedge x \in \mathit{Inf}\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\}$›
ex-in-conv option.simps(*3*))

have *B*: $\mathit{Option.these}\ ((\lambda x.\ \mathit{Some}\ (\bigsqcup \mathit{Option.these}\ x))\ `\ \{y - \{\mathit{None}\}\ |y.\ y \in A\})$
$= ((\lambda x.\ (\bigsqcup\ \mathit{Option.these}\ x))\ `\ \{y - \{\mathit{None}\}\ |y.\ y \in A\})$
by (*metis image-image these-image-Some-eq*)
{
fix *f*
assume *A*: $\bigwedge\ Y\ .\ (\exists\ y.\ Y = \mathit{the}\ `\ (y - \{\mathit{None}\}) \wedge y \in A) \Longrightarrow f\ Y \in Y$

have $\bigwedge xa.\ xa \in A \Longrightarrow f\ \{y.\ \exists\ a{\in}xa - \{\mathit{None}\}.\ y = \mathit{the}\ a\} = f\ (\mathit{the}\ `\ (xa - \{\mathit{None}\}))$
by (*simp add*: *image-def*)
from *this* **have** [*simp*]: $\bigwedge xa.\ xa \in A \Longrightarrow \exists\ x{\in}A.\ f\ \{y.\ \exists\ a{\in}xa - \{\mathit{None}\}.\ y = \mathit{the}\ a\} = f\ (\mathit{the}\ `\ (x - \{\mathit{None}\}))$
by *blast*
have $\bigwedge xa.\ xa \in A \Longrightarrow f\ (\mathit{the}\ `\ (xa - \{\mathit{None}\})) = f\ \{y.\ \exists\ a \in xa - \{\mathit{None}\}.$

$y = the\ a\} \wedge xa \in A$
by (*simp add*: *image-def*)
from *this* **have** [*simp*]: $\bigwedge xa.\ xa \in A \Longrightarrow \exists x.\ f\ (the\ `\ (xa - \{None\})) = f\ \{y.\ \exists a{\in}x - \{None\}.\ y = the\ a\} \wedge x \in A$
by *blast*

{
fix *Y*
have $Y \in A \Longrightarrow Some\ (f\ (the\ `\ (Y - \{None\}))) \in Y$
using *A* [*of the* ` (*Y* − {*None*})] **apply** (*simp add*: *image-def*)
using *option.collapse* **by** *fastforce*
}
from *this* **have** [*simp*]: $\bigwedge\ Y\ .\ Y \in A \Longrightarrow Some\ (f\ (the\ `\ (Y - \{None\}))) \in Y$
by *blast*
have [*simp*]: $(\prod x{\in}A.\ Some\ (f\ \{y.\ \exists x{\in}x - \{None\}.\ y = the\ x\})) = \prod \{Some\ (f\ \{y.\ \exists a{\in}x - \{None\}.\ y = the\ a\})\ |x.\ x \in A\}$
by (*simp add*: *Setcompr-eq-image*)

have [*simp*]: $\exists x.\ (\exists f.\ x = \{y.\ \exists x{\in}A.\ y = f\ x\} \wedge (\forall\ Y{\in}A.\ f\ Y \in Y)) \wedge \prod \{Some\ (f\ \{y.\ \exists a{\in}x - \{None\}.\ y = the\ a\})\ |x.\ x \in A\} = \prod x$
apply (*rule exI* [*of* - {*Some* (*f* {*y*. ∃ *a*∈*x* − {*None*}. *y* = *the a*}) | *x* . *x*∈ *A*}], *safe*)
by (*rule exI* [*of* - (λ *Y* . *Some* (*f* (*the* ` (*Y* − {*None*}))))], *safe*, *simp-all*)

{
fix *xb*
have $xb \in A \Longrightarrow (\prod x{\in}\{\{ya.\ \exists x{\in}y - \{None\}.\ ya = the\ x\}\ |y.\ y \in A\}.\ f\ x) \leq f\ \{y.\ \exists x{\in}xb - \{None\}.\ y = the\ x\}$
apply (*rule INF-lower2* [*of* {*y*. ∃ *x*∈*xb* − {*None*}. *y* = *the x*}])
by *blast*+
}
from *this* **have** [*simp*]: $(\prod x{\in}\{the\ `\ (y - \{None\})\ |y.\ y \in A\}.\ f\ x) \leq the\ (\prod Y{\in}A.\ Some\ (f\ (the\ `\ (Y - \{None\}))))$
apply (*simp add*: *Inf-option-def image-def Option.these-def*)
by (*rule Inf-greatest*, *clarsimp*)
have [*simp*]: *the* $(\prod Y{\in}A.\ Some\ (f\ (the\ `\ (Y - \{None\})))) \in Option.these\ (Inf\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\})$
apply (*auto simp add*: *Option.these-def*)
apply (*rule imageI*)
apply *auto*
using ‹$\bigwedge Y.\ Y \in A \Longrightarrow Some\ (f\ (the\ `\ (Y - \{None\}))) \in Y$› **apply** *blast*
apply (*auto simp add*: *Some-INF* [*symmetric*])
done
have $(\prod x{\in}\{the\ `\ (y - \{None\})\ |y.\ y \in A\}.\ f\ x) \leq \bigsqcup Option.these\ (Inf\ `\ \{f\ `\ A\ |f.\ \forall\ Y{\in}A.\ f\ Y \in Y\})$
by (*rule Sup-upper2* [*of the* (*Inf* ((λ *Y* . *Some* (*f* (*the* ` (*Y* − {*None*})) )) ` *A*))], *simp-all*)
}

from *this* **have** X: $\bigwedge f\ .\ \forall Y.\ (\exists y.\ Y = the\ `\ (y - \{None\}) \wedge y \in A) \longrightarrow f\ Y \in Y \Longrightarrow$
$(\bigsqcap x{\in}\{the\ `\ (y - \{None\})\ |y.\ y \in A\}.\ f\ x) \leq \bigsqcup Option.these\ (Inf\ `\ \{f\ `\ A\ |f.\ \forall Y{\in}A.\ f\ Y \in Y\})$
by *blast*

have [*simp*]: $\bigwedge\ x\ .\ x{\in}\{y - \{None\}\ |y.\ y \in A\} \Longrightarrow\ x \neq \{\} \wedge x \neq \{None\}$
using F **by** *fastforce*

have $(Inf\ (Sup\ `A)) = (Inf\ (Sup\ `\ \{y - \{None\}\ |\ y.\ y{\in}A\}))$
by (*subst A*, *simp*)

also have ... $= (\bigsqcap x{\in}\{y - \{None\}\ |y.\ y \in A\}.\ if\ x = \{\} \vee x = \{None\}\ then\ None\ else\ Some\ (\bigsqcup Option.these\ x))$
by (*simp add*: *Sup-option-def*)

also have ... $= (\bigsqcap x{\in}\{y - \{None\}\ |y.\ y \in A\}.\ Some\ (\bigsqcup Option.these\ x))$
using G **by** *fastforce*

also have ... $= Some\ (\bigsqcap Option.these\ ((\lambda x.\ Some\ (\bigsqcup Option.these\ x))\ `\ \{y - \{None\}\ |y.\ y \in A\}))$
by (*simp add*: *Inf-option-def*, *safe*)

also have ... $=\ Some\ (\bigsqcap\ ((\lambda x.\ \ (\bigsqcup Option.these\ x))\ `\ \{y - \{None\}\ |y.\ y \in A\}))$
by (*simp add*: B)

also have ... $= Some\ (Inf\ (Sup\ `\ \{the\ `\ (y - \{None\})\ |y.\ y \in A\}))$
by (*unfold* C, *simp*)
thm *Inf-Sup*
also have ... $= Some\ (\bigsqcup x{\in}\{f\ `\ \{the\ `\ (y - \{None\})\ |y.\ y \in A\}\ |f.\ \forall Y.\ (\exists y.\ Y = the\ `\ (y - \{None\}) \wedge y \in A) \longrightarrow f\ Y \in Y\}.\ \bigsqcap x)$
by (*simp add*: *Inf-Sup*)

also have ... $\leq \bigsqcup\ (Inf\ `\ \{f\ `\ A\ |f.\ \forall Y{\in}A.\ f\ Y \in Y\})$
proof (*cases* $\bigsqcup\ (Inf\ `\ \{f\ `\ A\ |f.\ \forall Y{\in}A.\ f\ Y \in Y\}))$
case *None*
then show *?thesis* **by** (*simp add*: *less-eq-option-def*)
next
case (*Some a*)
then show *?thesis*
apply *simp*
apply (*rule Sup-least*, *safe*)
apply (*simp add*: *Sup-option-def*)
apply (*cases* $(\forall f.\ \exists Y{\in}A.\ f\ Y \notin Y) \vee Inf\ `\ \{f\ `\ A\ |f.\ \forall Y{\in}A.\ f\ Y \in Y\} = \{None\}$, *simp-all*)
by (*drule* X, *simp*)
qed

finally show *?thesis* **by** *simp*
qed
qed

instance *option* :: (*complete-distrib-lattice*) *complete-distrib-lattice*
by (*standard*, *simp add*: *option-Inf-Sup*)

instance *option* :: (*complete-linorder*) *complete-linorder* **..**

unbundle *no lattice-syntax*

end

76 Futures and parallel lists for code generated towards Isabelle/ML

theory *Parallel*
imports *Main*
begin

76.1 Futures

datatype *'a future* = *fork unit* ⇒ *'a*

primrec *join* :: *'a future* ⇒ *'a* **where**
join (*fork f*) = *f* ()

lemma *future-eqI* [*intro*!]:
assumes *join f* = *join g*
shows *f* = *g*
using *assms* **by** (*cases f*, *cases g*) (*simp add*: *ext*)

code-printing
type-constructor *future* ⇀ (*Eval*) - *future*
| **constant** *fork* ⇀ (*Eval*) *Future.fork*
| **constant** *join* ⇀ (*Eval*) *Future.join*

code-reserved (*Eval*) *Future future*

76.2 Parallel lists

definition *map* :: (*'a* ⇒ *'b*) ⇒ *'a list* ⇒ *'b list* **where**
[*simp*]: *map* = *List.map*

definition *forall* :: (*'a* ⇒ *bool*) ⇒ *'a list* ⇒ *bool* **where**
forall = *list-all*

lemma *forall-all* [*simp*]:
forall P xs ⟷ (∀ *x*∈*set xs*. *P x*)

by (*simp add*: *forall-def list-all-iff*)

definition *exists* :: ('*a* ⇒ *bool*) ⇒ '*a list* ⇒ *bool* **where**
exists = *list-ex*

lemma *exists-ex* [*simp*]:
exists P xs ⟷ (∃ *x*∈*set xs*. *P x*)
by (*simp add*: *exists-def list-ex-iff*)

code-printing
constant *map* ⇀ (*Eval*) *Par'-List.map*
| **constant** *forall* ⇀ (*Eval*) *Par'-List.forall*
| **constant** *exists* ⇀ (*Eval*) *Par'-List.exists*

code-reserved (*Eval*) *Par-List*

hide-const (**open**) *fork join map exists forall*

end

77 Input syntax for pattern aliases (or "as-patterns" in Haskell)

theory *Pattern-Aliases*
imports *Main*
begin

Most functional languages (Haskell, ML, Scala) support aliases in patterns. This allows to refer to a subpattern with a variable name. This theory implements this using a check phase. It works well for function definitions (see usage below). All features are packed into a **bundle**.

The following caveats should be kept in mind:

- The translation expects a term of the form *f x y* = *rhs*, where *x* and *y* are patterns that may contain aliases. The result of the translation is a nested *Let*-expression on the right hand side. The code generator *does not* print Isabelle pattern aliases to target language pattern aliases.
- The translation does not process nested equalities; only the top-level equality is translated.
- Terms that do not adhere to the above shape may either stay untranslated or produce an error message. The **fun** command will complain if pattern aliases are left untranslated. In particular, there are no checks whether the patterns are wellformed or linear.

- The corresponding uncheck phase attempts to reverse the translation (no guarantee). The additionally introduced variables are bound using a "fake quantifier" that does not appear in the output.
- To obtain reasonable induction principles in function definitions, the bundle also declares a custom congruence rule for *Let* that only affects **fun**. This congruence rule might lead to an explosion in term size (although that is rare)! In some circumstances (using *let* to destructure tuples), the internal construction of functions stumbles over this rule and prints an error. To mitigate this, either
  - activate the bundle locally (**context includes ... begin**) or
  - rewrite the *let*-expression to use *case*: *let* $(a, b) = x$ *in* (b, a) becomes *case* x *of* $(a, b) \Rightarrow (b, a)$.
- The bundle also adds the *Let ?s ?f* $\equiv$ *?f ?s* rule to the simpset.

77.1 Definition

```
consts
  as :: 'a ⇒ 'a ⇒ 'a
  fake-quant :: ('a ⇒ prop) ⇒ prop

lemma let-cong-unfolding: M = N ⟹ f N = g N ⟹ Let M f = Let N g
by simp

translations P <= CONST fake-quant (λx. P)

ML‹
local

fun let-typ a b = a --> (a --> b) --> b
fun as-typ a = a --> a --> a

fun strip-all t =
  case try Logic.dest-all-global t of
    NONE => ([], t)
  | SOME (var, t) => apfst (cons var) (strip-all t)

fun all-Frees t =
  fold-aterms (fn Free (x, t) => insert (op =) (x, t) | - => I) t []

fun subst-once (old, new) t =
  let
    fun go t =
      if t = old then
        (new, true)
```

```
      else
        case t of
          u $ v =>
            let
              val (u', substituted) = go u
            in
              if substituted then
                (u' $ v, true)
              else
                case go v of (v', substituted) => (u $ v', substituted)
            end
        | Abs (name, typ, t) =>
            (case go t of (t', substituted) => (Abs (name, typ, t'), substituted))
        | - => (t, false)
  in fst (go t) end

(* adapted from logic.ML *)
fun fake-const T = Const (const-name ‹fake-quant›, (T --> propT) --> propT);

fun dependent-fake-name v t =
  let
    val x = Term.term-name v
    val T = Term.fastype-of v
    val t' = Term.abstract-over (v, t)
  in if Term.is-dependent t' then fake-const T $ Abs (x, T, t') else t end

in

fun check-pattern-syntax t =
  case strip-all t of
    (vars, Const ‹Trueprop› $ (Const (const-name ‹HOL.eq›, -) $ lhs $ rhs)) =>
      let
        fun go (Const (const-name ‹as›, -) $ pat $ var, rhs) =
              let
                val (pat', rhs') = go (pat, rhs)
                 val - = if is-Free var then () else error Right−hand side of =: must
be a free variable
                val rhs'' =
                 Const (const-name ‹Let›, let-typ (fastype-of var) (fastype-of rhs)) $
                    pat' $ lambda var rhs'
              in
                (pat', rhs'')
              end
          | go (t $ u, rhs) =
              let
                val (t', rhs') = go (t, rhs)
                val (u', rhs'') = go (u, rhs')
              in (t' $ u', rhs'') end
          | go (t, rhs) = (t, rhs)
```

```
      val (lhs', rhs') = go (lhs, rhs)

      val res = HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs', rhs'))

      val frees = filter (member (op =) vars) (all-Frees res)
    in fold (fn v => Logic.dependent-all-name (, v)) (map Free frees) res end
  | - => t

fun uncheck-pattern-syntax ctxt t =
  case strip-all t of
    (vars, Const ‹Trueprop› $ (Const (const-name ‹HOL.eq›, -) $ lhs $ rhs)) =>
      let
        (* restricted to going down abstractions; ignores eta−contracted rhs *)
        fun go lhs (rhs as Const (const-name ‹Let›, -) $ pat $ Abs (name, typ, t))
ctxt frees =
              if exists-subterm (fn t' => t' = pat) lhs then
                let
                  val ([name'], ctxt') = Variable.variant-fixes [name] ctxt
                  val free = Free (name', typ)
                  val subst = (pat, Const (const-name ‹as›, as-typ typ) $ pat $ free)
                  val lhs' = subst-once subst lhs
                  val rhs' = subst-bound (free, t)
                in
                  go lhs' rhs' ctxt' (Free (name', typ) :: frees)
                end
              else
                (lhs, rhs, ctxt, frees)
          | go lhs rhs ctxt frees = (lhs, rhs, ctxt, frees)

        val (lhs', rhs', -, frees) = go lhs rhs ctxt []

        val res =
          HOLogic.mk-Trueprop (HOLogic.mk-eq (lhs', rhs'))
          |> fold (fn v => Logic.dependent-all-name (, v)) (map Free vars)
          |> fold dependent-fake-name frees
      in
        if null frees then t else res
      end
  | - => t

end
›

bundle pattern-aliases begin

  notation as (infixr ‹=:› 1)

  declaration ‹K (Syntax-Phases.term-check 98 pattern-syntax (K (map check-pattern-syntax)))›
```

declaration ‹*K* (*Syntax-Phases.term-uncheck 98 pattern-syntax* (*map o uncheck-pattern-syntax*))›

declare *let-cong-unfolding* [*fundef-cong*]
declare *Let-def* [*simp*]

end

hide-const *as*
hide-const *fake-quant*

77.2 Usage

context includes *pattern-aliases* **begin**

Not very useful for plain definitions, but works anyway.

private definition *test-1 x* (*y* =: *z*) = *y* + *z*

lemma *test-1 x y* = *y* + *y*
by (*rule test-1-def*[*unfolded Let-def*])

Very useful for function definitions.

private fun *test-2* **where**
test-2 (*y* # (*y'* # *ys* =: *x'*) =: *x*) = *x* @ *x'* @ *x'* |
test-2 - = []

lemma *test-2* (*y* # *y'* # *ys*) = (*y* # *y'* # *ys*) @ (*y'* # *ys*) @ (*y'* # *ys*)
by (*rule test-2.simps*[*unfolded Let-def*])

ML‹
let
  val actual =
    @{*thm test-2.simps*(*1*)}
    |> *Thm.prop-of*
    |> *Syntax.pretty-term* ***context***
    |> *Pretty.pure-string-of*
  val expected = *test-2* (*?y* # (*?y'* # *?ys* =: *x'*) =: *x*) = *x* @ *x'* @ *x'*
in ***assert*** (*actual* = *expected*) *end*
›

end

end

78 Periodic Functions

theory *Periodic-Fun*
imports *Complex-Main*
begin

A locale for periodic functions. The idea is that one proves $f(x+p) = f(x)$ for some period p and gets derived results like $f(x-p) = f(x)$ and $f(x+2p) = f(x)$ for free.

g and *gm* are "plus/minus k periods" functions. *g1* and *gn1* are "plus/minus one period" functions. This is useful e.g. if the period is one; the lemmas one gets are then $f\ (x\ +\ 1) = f\ x$ instead of $f\ (x\ +\ 1\ *\ 1) = f\ x$ etc.

locale *periodic-fun* =
  fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$ **and** $g\ gm :: 'a \Rightarrow 'a \Rightarrow 'a$ **and** $g1\ gn1 :: 'a \Rightarrow 'a$
  assumes *plus-1*: $f\ (g1\ x) = f\ x$
  assumes *periodic-arg-plus-0*: $g\ x\ 0 = x$
  assumes *periodic-arg-plus-distrib*: $g\ x\ (of\text{-}int\ (m + n)) = g\ (g\ x\ (of\text{-}int\ n))\ (of\text{-}int\ m)$
  assumes *plus-1-eq*: $g\ x\ 1 = g1\ x$ **and** *minus-1-eq*: $g\ x\ (-1) = gn1\ x$
    and *minus-eq*: $g\ x\ (-y) = gm\ x\ y$
begin

lemma *plus-of-nat*: $f\ (g\ x\ (of\text{-}nat\ n)) = f\ x$
  by (*induction n*) (*insert periodic-arg-plus-distrib*[*of - 1 int n* **for** *n*],
    simp-all add: *plus-1 periodic-arg-plus-0 plus-1-eq*)

lemma *minus-of-nat*: $f\ (gm\ x\ (of\text{-}nat\ n)) = f\ x$
proof −
  have $f\ (g\ x\ (-\ of\text{-}nat\ n)) = f\ (g\ (g\ x\ (-\ of\text{-}nat\ n))\ (of\text{-}nat\ n))$
    by (*rule plus-of-nat*[*symmetric*])
  also have $\ldots = f\ (g\ (g\ x\ (of\text{-}int\ (-\ of\text{-}nat\ n)))\ (of\text{-}int\ (of\text{-}nat\ n)))$ **by** *simp*
  also have $\ldots = f\ x$
    by (*subst periodic-arg-plus-distrib* [*symmetric*]) (*simp add*: *periodic-arg-plus-0*)
  finally show *?thesis* **by** (*simp add*: *minus-eq*)
qed

lemma *plus-of-int*: $f\ (g\ x\ (of\text{-}int\ n)) = f\ x$
  by (*induction n*) (*simp-all add*: *plus-of-nat minus-of-nat minus-eq del*: *of-nat-Suc*)

lemma *minus-of-int*: $f\ (gm\ x\ (of\text{-}int\ n)) = f\ x$
  using *plus-of-int*[*of x of-int* $(-n)$] **by** (*simp add*: *minus-eq*)

lemma *plus-numeral*: $f\ (g\ x\ (numeral\ n)) = f\ x$
  by (*subst of-nat-numeral*[*symmetric*], *subst plus-of-nat*) (*rule refl*)

lemma *minus-numeral*: $f\ (gm\ x\ (numeral\ n)) = f\ x$
  by (*subst of-nat-numeral*[*symmetric*], *subst minus-of-nat*) (*rule refl*)

lemma *minus-1*: $f\ (gn1\ x) = f\ x$
  using *minus-of-nat*[*of x 1*] **by** (*simp flip*: *minus-1-eq minus-eq*)

lemmas *periodic-simps* = *plus-of-nat minus-of-nat plus-of-int minus-of-int*
    plus-numeral minus-numeral plus-1 minus-1

end

Specialised case of the *periodic-fun* locale for periods that are not 1. Gives lemmas $f\ (x - period) = f\ x$ etc.

locale *periodic-fun-simple* =
  fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$ **and** $period :: 'a$
  assumes *plus-period*: $f\ (x + period) = f\ x$
begin
sublocale *periodic-fun* $f\ \lambda z\ x.\ z + x * period\ \lambda z\ x.\ z - x * period$
  $\lambda z.\ z + period\ \lambda z.\ z - period$
  by *standard* (*simp-all add*: *ring-distribs plus-period*)
end

Specialised case of the *periodic-fun* locale for period 1. Gives lemmas $f\ (x - 1) = f\ x$ etc.

locale *periodic-fun-simple'* =
  fixes $f :: ('a :: \{ring\text{-}1\}) \Rightarrow 'b$
  assumes *plus-period*: $f\ (x + 1) = f\ x$
begin
sublocale *periodic-fun* $f\ \lambda z\ x.\ z + x\ \lambda z\ x.\ z - x\ \lambda z.\ z + 1\ \lambda z.\ z - 1$
  by *standard* (*simp-all add*: *ring-distribs plus-period*)

lemma *of-nat*: $f\ (of\text{-}nat\ n) = f\ 0$ **using** *plus-of-nat*[*of 0 n*] **by** *simp*
lemma *uminus-of-nat*: $f\ (-of\text{-}nat\ n) = f\ 0$ **using** *minus-of-nat*[*of 0 n*] **by** *simp*
lemma *of-int*: $f\ (of\text{-}int\ n) = f\ 0$ **using** *plus-of-int*[*of 0 n*] **by** *simp*
lemma *uminus-of-int*: $f\ (-of\text{-}int\ n) = f\ 0$ **using** *minus-of-int*[*of 0 n*] **by** *simp*
lemma *of-numeral*: $f\ (numeral\ n) = f\ 0$ **using** *plus-numeral*[*of 0 n*] **by** *simp*
lemma *of-neg-numeral*: $f\ (-numeral\ n) = f\ 0$ **using** *minus-numeral*[*of 0 n*] **by** *simp*
lemma *of-1*: $f\ 1 = f\ 0$ **using** *plus-of-nat*[*of 0 1*] **by** *simp*
lemma *of-neg-1*: $f\ (-1) = f\ 0$ **using** *minus-of-nat*[*of 0 1*] **by** *simp*

lemmas *periodic-simps'* =
  of-nat uminus-of-nat of-int uminus-of-int of-numeral of-neg-numeral of-1 of-neg-1

end

lemma *sin-plus-pi*: $sin\ ((z :: 'a :: \{real\text{-}normed\text{-}field,banach\}) + of\text{-}real\ pi) = -sin\ z$
  by (*simp add*: *sin-add*)

lemma *cos-plus-pi*: $cos\ ((z :: 'a :: \{real\text{-}normed\text{-}field,banach\}) + of\text{-}real\ pi) = -cos\ z$
  by (*simp add*: *cos-add*)

interpretation *sin*: *periodic-fun-simple sin* $2 * of\text{-}real\ pi :: 'a :: \{real\text{-}normed\text{-}field,banach\}$
proof
  fix $z :: 'a$
  have $sin\ (z + 2 * of\text{-}real\ pi) = sin\ (z + of\text{-}real\ pi + of\text{-}real\ pi)$ **by** (*simp add*:

ac-simps)
also have ... = *sin z* **by** (*simp only*: *sin-plus-pi*) *simp*
finally show *sin* (*z* + *2* ∗ *of-real pi*) = *sin z* **.**
qed

interpretation *cos*: *periodic-fun-simple cos 2* ∗ *of-real pi* :: *'a* :: {*real-normed-field*,*banach*}
proof
fix *z* :: *'a*
have *cos* (*z* + *2* ∗ *of-real pi*) = *cos* (*z* + *of-real pi* + *of-real pi*) **by** (*simp add*: *ac-simps*)
also have ... = *cos z* **by** (*simp only*: *cos-plus-pi*) *simp*
finally show *cos* (*z* + *2* ∗ *of-real pi*) = *cos z* **.**
qed

interpretation *tan*: *periodic-fun-simple tan 2* ∗ *of-real pi* :: *'a* :: {*real-normed-field*,*banach*}
by *standard* (*simp only*: *tan-def* [*abs-def*] *sin.plus-1 cos.plus-1*)

interpretation *cot*: *periodic-fun-simple cot 2* ∗ *of-real pi* :: *'a* :: {*real-normed-field*,*banach*}
by *standard* (*simp only*: *cot-def* [*abs-def*] *sin.plus-1 cos.plus-1*)

lemma *cos-eq-neg-periodic-intro*:
assumes *x* − *y* = *2*∗(*of-int k*)∗*pi* + *pi* ∨ *x* + *y* = *2*∗(*of-int k*)∗*pi* + *pi*
shows *cos x* = − *cos y* **using** *assms*
proof
assume *x* − *y* = *2* ∗ (*of-int k*) ∗ *pi* + *pi*
then show *?thesis*
using *cos.periodic-simps*[*of y*+*pi*]
by (*auto simp add*:*algebra-simps*)
next
assume *x* + *y* = *2* ∗ *real-of-int k* ∗ *pi* + *pi*
then show *?thesis*
using *cos.periodic-simps*[*of* −*y*+*pi*]
by (*clarsimp simp add*: *algebra-simps*) (*smt* (*verit*))
qed

lemma *cos-eq-periodic-intro*:
assumes *x* − *y* = *2*∗(*of-int k*)∗*pi* ∨ *x* + *y* = *2*∗(*of-int k*)∗*pi*
shows *cos x* = *cos y*
by (*smt* (*verit*, *best*) *assms cos-eq-neg-periodic-intro cos-minus-pi cos-periodic-pi*)

lemma *cos-eq-arccos-Ex*:
cos x = *y* ⟷ −*1*≤*y* ∧ *y*≤*1* ∧ (∃ *k*::*int*. *x* = *arccos y* + *2*∗*k*∗*pi* ∨ *x* = − *arccos y* + *2*∗*k*∗*pi*) (**is** *?L*=*?R*)
proof
assume *?R* **then show** *cos x* = *y*
by (*metis cos.plus-of-int cos-arccos cos-minus id-apply mult.assoc mult.left-commute of-real-eq-id*)
next
assume *L*: *?L*

let *?goal* = (∃ *k*::*int*. *x* = *arccos y* + *2*∗*k*∗*pi* ∨ *x* = − *arccos y* + *2*∗*k*∗*pi*)
obtain *k*::*int* **where** *k*: −*pi* < *x* − *k*∗(*2*∗*pi*) *x* − *k*∗(*2*∗*pi*) ≤ *pi*
using *ceiling-divide-lower* [*of 2*∗*pi x*−*pi*] *ceiling-divide-upper* [*of 2*∗*pi x*−*pi*]
by (*simp add*: *divide-simps algebra-simps*) (*metis mult.commute*)
have ∗: *cos* (*x* − *k* ∗ *2*∗*pi*) = *y*
using *cos.periodic-simps*(*3*)[*of x* −*k*] *L* **by** (*auto simp add*:*field-simps*)
then have ∗∗: *?goal* **when** *x*−*k*∗*2*∗*pi* ≥ *0*
using *arccos-cos k that* **by** *force*
then show −*1*≤*y* ∧ *y*≤*1* ∧ *?goal*
using ∗ *arccos-cos2 k*(*1*) **by** *force*
qed

end

79 Polynomial mapping: combination of almost everywhere zero functions with an algebraic view

theory *Poly-Mapping*
imports *Groups-Big-Fun Fun-Lexorder More-List*
begin

79.1 Preliminary: auxiliary operations for *almost everywhere zero*

A central notion for polynomials are functions being *almost everywhere zero*. For these we provide some auxiliary definitions and lemmas.

lemma *finite-mult-not-eq-zero-leftI*:
fixes *f* :: *'b* ⇒ *'a* :: *mult-zero*
assumes *finite* {*a*. *f a* ≠ *0*}
shows *finite* {*a*. *g a* ∗ *f a* ≠ *0*}
by (*metis* (*mono-tags*, *lifting*) *Collect-mono assms mult-zero-right finite-subset*)

lemma *finite-mult-not-eq-zero-rightI*:
fixes *f* :: *'b* ⇒ *'a* :: *mult-zero*
assumes *finite* {*a*. *f a* ≠ *0*}
shows *finite* {*a*. *f a* ∗ *g a* ≠ *0*}
by (*metis* (*mono-tags*, *lifting*) *Collect-mono assms lambda-zero finite-subset*)

lemma *finite-mult-not-eq-zero-prodI*:
fixes *f g* :: *'a* ⇒ *'b*::*semiring-0*
assumes *finite* {*a*. *f a* ≠ *0*} (**is** *finite ?F*)
assumes *finite* {*b*. *g b* ≠ *0*} (**is** *finite ?G*)
shows *finite* {(*a*, *b*). *f a* ∗ *g b* ≠ *0*}
proof −
from *assms* **have** *finite* (*?F* × *?G*)
by *blast*
then have *finite* {(*a*, *b*). *f a* ≠ *0* ∧ *g b* ≠ *0*}

```
    by simp
  then show ?thesis
    by (rule rev-finite-subset) auto
qed

lemma finite-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add ⇒ 'b::semiring-0
  assumes finite {a. f a ≠ 0} (is finite ?F)
  assumes finite {b. g b ≠ 0} (is finite ?G)
  shows finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0} (is finite ?FG)
proof −
  from assms have finite (?F × ?G)
    by (simp add: finite-cartesian-product-iff)
  then have finite (case-prod plus ‘ (?F × ?G))
    by (rule finite-imageI)
  also have case-prod plus ‘ (?F × ?G) = ?FG
    by auto
  finally show ?thesis
    by simp
qed

lemma finite-mult-not-eq-zero-sumI:
  fixes f g :: 'a::monoid-add ⇒ 'b::semiring-0
  assumes finite {a. f a ≠ 0}
  assumes finite {b. g b ≠ 0}
  shows finite {a + b | a b. f a * g b ≠ 0}
proof −
  from assms
  have finite {a + b | a b. f a ≠ 0 ∧ g b ≠ 0}
    by (rule finite-not-eq-zero-sumI)
  then show ?thesis
    by (rule rev-finite-subset) (auto dest: mult-not-zero)
qed

lemma finite-Sum-any-not-eq-zero-weakenI:
  assumes finite {a. ∃ b. f a b ≠ 0}
  shows finite {a. Sum-any (f a) ≠ 0}
proof −
  have {a. Sum-any (f a) ≠ 0} ⊆ {a. ∃ b. f a b ≠ 0}
    by (auto elim: Sum-any.not-neutral-obtains-not-neutral)
  then show ?thesis using assms by (rule finite-subset)
qed

context zero
begin

definition when :: 'a ⇒ bool ⇒ 'a (infixl ‹when› 20)
where
  (a when P) = (if P then a else 0)
```

Case distinctions always complicate matters, particularly when nested. The (*when*) operation allows to minimise these if *0* is the false-case value and makes proof obligations much more readable.

lemma *when* [*simp*]:
$P \Longrightarrow (a\ when\ P) = a$
$\neg\ P \Longrightarrow (a\ when\ P) = 0$
by (*simp-all add*: *when-def*)

lemma *when-simps* [*simp*]:
$(a\ when\ True) = a$
$(a\ when\ False) = 0$
by *simp-all*

lemma *when-cong*:
assumes $P \longleftrightarrow Q$
and $Q \Longrightarrow a = b$
shows $(a\ when\ P) = (b\ when\ Q)$
using *assms* **by** (*simp add*: *when-def*)

lemma *zero-when* [*simp*]:
$(0\ when\ P) = 0$
by (*simp add*: *when-def*)

lemma *when-when*:
$(a\ when\ P\ when\ Q) = (a\ when\ P \wedge Q)$
by (*cases Q*) *simp-all*

lemma *when-commute*:
$(a\ when\ Q\ when\ P) = (a\ when\ P\ when\ Q)$
by (*simp add*: *when-when conj-commute*)

lemma *when-neq-zero* [*simp*]:
$(a\ when\ P) \neq 0 \longleftrightarrow P \wedge a \neq 0$
by (*cases P*) *simp-all*

end

context *monoid-add*
begin

lemma *when-add-distrib*:
$(a + b\ when\ P) = (a\ when\ P) + (b\ when\ P)$
by (*simp add*: *when-def*)

end

context *semiring-1*
begin

lemma *zero-power-eq*:
  $0 \mathbin{\hat{}} n = (1 \text{ when } n = 0)$
  by (*simp add*: *power-0-left*)

end

context *comm-monoid-add*
begin

lemma *Sum-any-when-equal* [*simp*]:
  $(\sum a.\ (f\ a \text{ when } a = b)) = f\ b$
  by (*simp add*: *when-def*)

lemma *Sum-any-when-equal'* [*simp*]:
  $(\sum a.\ (f\ a \text{ when } b = a)) = f\ b$
  by (*simp add*: *when-def*)

lemma *Sum-any-when-independent*:
  $(\sum a.\ g\ a \text{ when } P) = ((\sum a.\ g\ a) \text{ when } P)$
  by (*cases P*) *simp-all*

lemma *Sum-any-when-dependent-prod-right*:
  $(\sum (a,\ b).\ g\ a \text{ when } b = h\ a) = (\sum a.\ g\ a)$
proof −
  have *inj-on* $(\lambda a.\ (a,\ h\ a))\ \{a.\ g\ a \neq 0\}$
    by (*rule inj-onI*) *auto*
  then show *?thesis* **unfolding** *Sum-any.expand-set*
    by (*rule sum.reindex-cong*) *auto*
qed

lemma *Sum-any-when-dependent-prod-left*:
  $(\sum (a,\ b).\ g\ b \text{ when } a = h\ b) = (\sum b.\ g\ b)$
proof −
  have $(\sum (a,\ b).\ g\ b \text{ when } a = h\ b) = (\sum (b,\ a).\ g\ b \text{ when } a = h\ b)$
    by (*rule Sum-any.reindex-cong* [*of prod.swap*]) (*simp-all add*: *fun-eq-iff*)
  then show *?thesis* **by** (*simp add*: *Sum-any-when-dependent-prod-right*)
qed

end

context *cancel-comm-monoid-add*
begin

lemma *when-diff-distrib*:
  $(a - b \text{ when } P) = (a \text{ when } P) - (b \text{ when } P)$
  by (*simp add*: *when-def*)

end

context *group-add*
begin

lemma *when-uminus-distrib*:
(− *a when P*) = − (*a when P*)
by (*simp add*: *when-def*)

end

context *mult-zero*
begin

lemma *mult-when*:
a ∗ (*b when P*) = (*a* ∗ *b when P*)
by (*cases P*) *simp-all*

lemma *when-mult*:
(*a when P*) ∗ *b* = (*a* ∗ *b when P*)
by (*cases P*) *simp-all*

end

79.2 Type definition

The following type is of central importance:

typedef (**overloaded**) ('*a*, '*b*) *poly-mapping* (‹(- ⇒$_0$ /-)› [*1*, *0*] *0*) =
{*f* :: '*a* ⇒ '*b*::*zero*. *finite* {*x*. *f x* ≠ *0*}}
morphisms *lookup Abs-poly-mapping*
using *not-finite-existsD* **by** *force*

declare *lookup-inverse* [*simp*]
declare *lookup-inject* [*simp*]

lemma *lookup-Abs-poly-mapping* [*simp*]:
finite {*x*. *f x* ≠ *0*} ⟹ *lookup* (*Abs-poly-mapping f*) = *f*
using *Abs-poly-mapping-inverse* [*of f*] **by** *simp*

lemma *finite-lookup* [*simp*]:
finite {*k*. *lookup f k* ≠ *0*}
using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *finite-lookup-nat* [*simp*]:
fixes *f* :: '*a* ⇒$_0$ *nat*
shows *finite* {*k*. *0* < *lookup f k*}
using *poly-mapping.lookup* [*of f*] **by** *simp*

lemma *poly-mapping-eqI*:
assumes ⋀*k*. *lookup f k* = *lookup g k*
shows *f* = *g*

using *assms* **unfolding** *poly-mapping.lookup-inject* [*symmetric*]
by *blast*

lemma *poly-mapping-eq-iff*: $a = b \longleftrightarrow lookup\ a = lookup\ b$
by *auto*

We model the universe of functions being *almost everywhere zero* by means of a separate type $'a \Rightarrow_0 'b$. For convenience we provide a suggestive infix syntax which is a variant of the usual function space syntax. Conversion between both types happens through the morphisms

lookup

Abs-poly-mapping

satisfying

$Abs\text{-}poly\text{-}mapping\ (lookup\ ?x) = ?x$

$finite\ \{x.\ ?f\ x \neq 0\} \Longrightarrow lookup\ (Abs\text{-}poly\text{-}mapping\ ?f) = ?f$

Luckily, we have rarely to deal with those low-level morphisms explicitly but rely on Isabelle's *lifting* package with its method *transfer* and its specification tool *lift-definition*.

setup-lifting *type-definition-poly-mapping*
code-datatype *Abs-poly-mapping*— FIXME? workaround for preventing *code-abstype* setup

$'a \Rightarrow_0 'b$ serves distinctive purposes:

1. A clever nesting as $(nat \Rightarrow_0 nat) \Rightarrow_0 'a$ later in theory *MPoly* gives a suitable representation type for polynomials *almost for free*: Interpreting $nat \Rightarrow_0 nat$ as a mapping from variable identifiers to exponents yields monomials, and the whole type maps monomials to coefficients. Lets call this the *ultimate interpretation*.

2. A further more specialised type isomorphic to $nat \Rightarrow_0 'a$ is apt to direct implementation using code generation [1].

Note that despite the names *mapping* and *lookup* suggest something implementation-near, it is best to keep $'a \Rightarrow_0 'b$ as an abstract *algebraic* type providing operations like *addition*, *multiplication* without any notion of key-order, data structures etc. This implementations-specific notions are easily introduced later for particular implementations but do not provide any gain for specifying logical properties of polynomials.

79.3 Additive structure

The additive structure covers the usual operations *0*, + and (unary and binary) −. Recalling the ultimate interpretation, it is obvious that these have just lift the corresponding operations on values to mappings.

Isabelle has a rich hierarchy of algebraic type classes, and in such situations of pointwise lifting a typical pattern is to have instantiations for a considerable number of type classes.

The operations themselves are specified using *lift-definition*, where the proofs of the *almost everywhere zero* property can be significantly involved.

The *lookup* operation is supposed to be usable explicitly (unless in other situations where the morphisms between types are somehow internal to the *lifting* package). Hence it is good style to provide explicit rewrite rules how *lookup* acts on operations immediately.

instantiation *poly-mapping* :: (*type*, *zero*) *zero*
begin

lift-definition *zero-poly-mapping* :: $'a \Rightarrow_0 'b$
is $\lambda k.\ 0$
by *simp*

instance ..

end

lemma *Abs-poly-mapping* [*simp*]: *Abs-poly-mapping* $(\lambda k.\ 0) = 0$
by (*simp add*: *zero-poly-mapping.abs-eq*)

lemma *lookup-zero* [*simp*]: *lookup 0 k* = *0*
by *transfer rule*

instantiation *poly-mapping* :: (*type*, *monoid-add*) *monoid-add*
begin

lift-definition *plus-poly-mapping* ::
$('a \Rightarrow_0 'b) \Rightarrow ('a \Rightarrow_0 'b) \Rightarrow 'a \Rightarrow_0 'b$
is $\lambda f1\ f2\ k.\ f1\ k + f2\ k$
proof −
fix *f1 f2* :: $'a \Rightarrow 'b$
assume *finite* $\{k.\ f1\ k \neq 0\}$
and *finite* $\{k.\ f2\ k \neq 0\}$
then have *finite* $(\{k.\ f1\ k \neq 0\} \cup \{k.\ f2\ k \neq 0\})$ **by** *auto*
moreover have $\{x.\ f1\ x + f2\ x \neq 0\} \subseteq \{k.\ f1\ k \neq 0\} \cup \{k.\ f2\ k \neq 0\}$
by *auto*
ultimately show *finite* $\{x.\ f1\ x + f2\ x \neq 0\}$
by (*blast intro*: *finite-subset*)
qed

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

end

lemma *lookup-add*: *lookup* (*f* + *g*) *k* = *lookup f k* + *lookup g k*
by (*simp add*: *plus-poly-mapping.rep-eq*)

instance *poly-mapping* :: (*type*, *comm-monoid-add*) *comm-monoid-add*
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

lemma *lookup-sum*: *lookup* (*sum pp X*) *i* = *sum* (λ*x*. *lookup* (*pp x*) *i*) *X*
by (*induction rule*: *infinite-finite-induct*) (*auto simp*: *lookup-add*)

instantiation *poly-mapping* :: (*type*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
begin

lift-definition *minus-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ '*a* ⇒$_0$ '*b*
is λ*f1 f2 k*. *f1 k* − *f2 k*
proof −
fix *f1 f2* :: '*a* ⇒ '*b*
assume *finite* {*k*. *f1 k* ≠ *0*}
and *finite* {*k*. *f2 k* ≠ *0*}
then have *finite* ({*k*. *f1 k* ≠ *0*} ∪ {*k*. *f2 k* ≠ *0*}) **by** *auto*
moreover have {*x*. *f1 x* − *f2 x* ≠ *0*} ⊆ {*k*. *f1 k* ≠ *0*} ∪ {*k*. *f2 k* ≠ *0*}
by *auto*
ultimately show *finite* {*x*. *f1 x* − *f2 x* ≠ *0*} **by** (*blast intro*: *finite-subset*)
qed

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff diff-diff-add*)+

end

instantiation *poly-mapping* :: (*type*, *ab-group-add*) *ab-group-add*
begin

lift-definition *uminus-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ '*a* ⇒$_0$ '*b*
is *uminus*
by *simp*

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff ac-simps*)+

end

lemma *lookup-uminus* [*simp*]:
lookup (− *f*) *k* = − *lookup f k*
by *transfer simp*

lemma *lookup-minus*:
lookup (*f* − *g*) *k* = *lookup f k* − *lookup g k*
by *transfer rule*

79.4 Multiplicative structure

instantiation *poly-mapping* :: (*zero*, *zero-neq-one*) *zero-neq-one*
begin

lift-definition *one-poly-mapping* :: *'a* ⇒$_0$ *'b*
is λ*k*. *1 when k* = *0*
by *simp*

instance
by *intro-classes* (*transfer*, *simp add*: *fun-eq-iff*)

end

lemma *lookup-one*: *lookup 1 k* = (*1 when k* = *0*)
by (*meson one-poly-mapping.rep-eq*)

lemma *lookup-one-zero* [*simp*]:
lookup 1 0 = *1*
by (*simp add*: *one-poly-mapping.rep-eq*)

definition *prod-fun* :: (*'a* ⇒ *'b*) ⇒ (*'a* ⇒ *'b*) ⇒ *'a*::*monoid-add* ⇒ *'b*::*semiring-0*
where
prod-fun f1 f2 k = (∑ *l*. *f1 l* ∗ (∑ *q*. (*f2 q when k* = *l* + *q*)))

lemma *prod-fun-unfold-prod*:
fixes *f g* :: *'a* :: *monoid-add* ⇒ *'b*::*semiring-0*
assumes *fin-f*: *finite* {*a*. *f a* ≠ *0*}
assumes *fin-g*: *finite* {*b*. *g b* ≠ *0*}
shows *prod-fun f g k* = (∑ (*a*, *b*). *f a* ∗ *g b when k* = *a* + *b*)
proof −
let *?C* = {*a*. *f a* ≠ *0*} × {*b*. *g b* ≠ *0*}
from *fin-f fin-g* **have** *finite ?C* **by** *blast*
moreover
have {*a*. ∃ *b*. (*f a* ∗ *g b when k* = *a* + *b*) ≠ *0*} ×
{*b*. ∃ *a*. (*f a* ∗ *g b when k* = *a* + *b*) ≠ *0*} ⊆ {*a*. *f a* ≠ *0*} × {*b*. *g b* ≠ *0*}
by *auto*
ultimately show *?thesis* **using** *fin-g*
by (*auto simp*: *prod-fun-def*
Sum-any.cartesian-product [*of* {*a*. *f a* ≠ *0*} × {*b*. *g b* ≠ *0*}] *Sum-any-right-distrib mult-when*)

qed

lemma *finite-prod-fun*:
fixes *f1 f2* :: *'a* :: *monoid-add* ⇒ *'b* :: *semiring-0*
assumes *fin1*: *finite* {*l*. *f1 l* ≠ *0*}
and *fin2*: *finite* {*q*. *f2 q* ≠ *0*}
shows *finite* {*k*. *prod-fun f1 f2 k* ≠ *0*}
proof −
have ∗: *finite* {*k*. (∃ *l*. *f1 l* ≠ *0* ∧ (∃ *q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*))}
using *assms* **by** *simp*
have *aux*: *sum f2* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*} = (∑ *q*. (*f2 q when k* = *l* + *q*))
for *k l*
proof −
have {*q*. (*f2 q when k* = *l* + *q*) ≠ *0*} ⊆ {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*} **by** *auto*
with *fin2* **show** *?thesis*
by (*simp add*: *Sum-any.expand-superset* [*of* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*}])
qed
have {*k*. (∑ *l*. *f1 l* ∗ *sum f2* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*}) ≠ *0*}
⊆ {*k*. (∃ *l*. *f1 l* ∗ *sum f2* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*} ≠ *0*)}
by (*auto elim*!: *Sum-any.not-neutral-obtains-not-neutral*)
also have ... ⊆ {*k*. (∃ *l*. *f1 l* ≠ *0* ∧ *sum f2* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*} ≠ *0*)}
by (*auto dest*: *mult-not-zero*)
also have ... ⊆ {*k*. (∃ *l*. *f1 l* ≠ *0* ∧ (∃ *q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*))}
by (*auto elim*!: *sum.not-neutral-contains-not-neutral*)
finally have *finite* {*k*. (∑ *l*. *f1 l* ∗ *sum f2* {*q*. *f2 q* ≠ *0* ∧ *k* = *l* + *q*}) ≠ *0*}
using ∗ **by** (*rule finite-subset*)
with *aux* **have** *finite* {*k*. (∑ *l*. *f1 l* ∗ (∑ *q*. (*f2 q when k* = *l* + *q*))) ≠ *0*}
by *simp*
with *fin2* **show** *?thesis*
by (*simp add*: *prod-fun-def*)
qed

instantiation *poly-mapping* :: (*monoid-add*, *semiring-0*) *semiring-0*
begin

lift-definition *times-poly-mapping* :: (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *'a* ⇒$_0$ *'b*
is *prod-fun*
by(*rule finite-prod-fun*)

instance
proof
fix *a b c* :: *'a* ⇒$_0$ *'b*
show *a* ∗ *b* ∗ *c* = *a* ∗ (*b* ∗ *c*)
proof *transfer*
fix *f g h* :: *'a* ⇒ *'b*
assume *fin-f*: *finite* {*a*. *f a* ≠ *0*} (**is** *finite ?F*)
assume *fin-g*: *finite* {*b*. *g b* ≠ *0*} (**is** *finite ?G*)
assume *fin-h*: *finite* {*c*. *h c* ≠ *0*} (**is** *finite ?H*)
from *fin-f fin-g* **have** *fin-fg*: *finite* {(*a*, *b*). *f a* ∗ *g b* ≠ *0*} (**is** *finite ?FG*)

by (*rule finite-mult-not-eq-zero-prodI*)
from *fin-g fin-h* **have** *fin-gh*: *finite* {(*b*, *c*). *g b* ∗ *h c* ≠ *0*} (**is** *finite ?GH*)
by (*rule finite-mult-not-eq-zero-prodI*)
from *fin-f fin-g* **have** *fin-fg'*: *finite* {*a* + *b* | *a b*. *f a* ∗ *g b* ≠ *0*} (**is** *finite ?FG'*)
by (*rule finite-mult-not-eq-zero-sumI*)
then have *fin-fg''*: *finite* {*d*. (∑ (*a*, *b*). *f a* ∗ *g b when d* = *a* + *b*) ≠ *0*}
by (*auto intro*: *finite-Sum-any-not-eq-zero-weakenI*)
from *fin-g fin-h* **have** *fin-gh'*: *finite* {*b* + *c* | *b c*. *g b* ∗ *h c* ≠ *0*} (**is** *finite ?GH'*)
by (*rule finite-mult-not-eq-zero-sumI*)
then have *fin-gh''*: *finite* {*d*. (∑ (*b*, *c*). *g b* ∗ *h c when d* = *b* + *c*) ≠ *0*}
by (*auto intro*: *finite-Sum-any-not-eq-zero-weakenI*)
show *prod-fun* (*prod-fun f g*) *h* = *prod-fun f* (*prod-fun g h*) (**is** *?lhs* = *?rhs*)
proof
fix *k*
from *fin-f fin-g fin-h fin-fg''*
have *?lhs k* = (∑ (*ab*, *c*). (∑ (*a*, *b*). *f a* ∗ *g b when ab* = *a* + *b*) ∗ *h c when k* = *ab* + *c*)
by (*simp add*: *prod-fun-unfold-prod*)
also have ... = (∑ (*ab*, *c*). (∑ (*a*, *b*). *f a* ∗ *g b* ∗ *h c when k* = *ab* + *c when ab* = *a* + *b*))
using *fin-fg*
apply (*simp add*: *Sum-any-left-distrib split-def flip*: *Sum-any-when-independent*)
apply (*simp add*: *when-when when-mult mult-when conj-commute*)
done
also have ... = (∑ (*ab*, *c*, *a*, *b*). *f a* ∗ *g b* ∗ *h c when k* = *ab* + *c when ab* = *a* + *b*)
apply (*subst Sum-any.cartesian-product2* [*of* (*?FG'* × *?H*) × *?FG*])
apply (*auto simp*: *finite-cartesian-product-iff fin-fg fin-fg' fin-h dest*: *mult-not-zero*)
done
also have ... = (∑ (*ab*, *c*, *a*, *b*). *f a* ∗ *g b* ∗ *h c when k* = *a* + *b* + *c when ab* = *a* + *b*)
by (*rule Sum-any.cong*) (*simp add*: *split-def when-def*)
also have ... = (∑ (*ab*, *cab*). (*case cab of* (*c*, *a*, *b*) ⇒ *f a* ∗ *g b* ∗ *h c when k* = *a* + *b* + *c*)
when ab = (*case cab of* (*c*, *a*, *b*) ⇒ *a* + *b*))
by (*simp add*: *split-def*)
also have ... = (∑ (*c*, *a*, *b*). *f a* ∗ *g b* ∗ *h c when k* = *a* + *b* + *c*)
by (*simp add*: *Sum-any-when-dependent-prod-left*)
also have ... = (∑ (*bc*, *cab*). (*case cab of* (*c*, *a*, *b*) ⇒ *f a* ∗ *g b* ∗ *h c when k* = *a* + *b* + *c*)
when bc = (*case cab of* (*c*, *a*, *b*) ⇒ *b* + *c*))
by (*simp add*: *Sum-any-when-dependent-prod-left*)
also have ... = (∑ (*bc*, *c*, *a*, *b*). *f a* ∗ *g b* ∗ *h c when k* = *a* + *b* + *c when bc* = *b* + *c*)
by (*simp add*: *split-def*)
also have ... = (∑ (*bc*, *c*, *a*, *b*). *f a* ∗ *g b* ∗ *h c when bc* = *b* + *c when k* = *a* + *bc*)
by (*rule Sum-any.cong*) (*simp add*: *split-def when-def ac-simps*)
also have ... = (∑ (*a*, *bc*, *b*, *c*). *f a* ∗ *g b* ∗ *h c when bc* = *b* + *c when k* =

```
a + bc)
      proof −
        have bij (λ(a, d, b, c). (d, c, a, b))
          by (auto intro!: bijI injI surjI [of - λ(d, c, a, b). (a, d, b, c)] simp add:
split-def)
        then show ?thesis
          by (rule Sum-any.reindex-cong) auto
      qed
      also have ... = (∑(a, bc). (∑(b, c). f a * g b * h c when bc = b + c when
k = a + bc))
        apply (subst Sum-any.cartesian-product2 [of (?F × ?GH') × ?GH])
         apply (auto simp: finite-cartesian-product-iff fin-f fin-gh fin-gh' ac-simps
dest: mult-not-zero)
        done
      also have ... = (∑(a, bc). f a * (∑(b, c). g b * h c when bc = b + c) when
k = a + bc)
        apply (subst Sum-any-right-distrib)
        using fin-gh apply (simp add: split-def)
        apply (subst Sum-any-when-independent [symmetric])
        apply (simp add: when-when when-mult mult-when split-def ac-simps)
        done
      also from fin-f fin-g fin-h fin-gh''
      have ... = ?rhs k
        by (simp add: prod-fun-unfold-prod)
      finally show ?lhs k = ?rhs k .
    qed
  qed
  show (a + b) * c = a * c + b * c
  proof transfer
    fix f g h :: 'a ⇒ 'b
    assume fin-f: finite {k. f k ≠ 0}
    assume fin-g: finite {k. g k ≠ 0}
    assume fin-h: finite {k. h k ≠ 0}
    show prod-fun (λk. f k + g k) h = (λk. prod-fun f h k + prod-fun g h k)
      apply (rule ext)
      apply (simp add: prod-fun-def algebra-simps)
      by (simp add: Sum-any.distrib fin-f fin-g finite-mult-not-eq-zero-rightI)
  qed
  show a * (b + c) = a * b + a * c
  proof transfer
    fix f g h :: 'a ⇒ 'b
    assume fin-f: finite {k. f k ≠ 0}
    assume fin-g: finite {k. g k ≠ 0}
    assume fin-h: finite {k. h k ≠ 0}
    show prod-fun f (λk. g k + h k) = (λk. prod-fun f g k + prod-fun f h k)
      apply (rule ext)
     apply (auto simp: prod-fun-def Sum-any.distrib algebra-simps when-add-distrib
fin-g fin-h)
      by (simp add: Sum-any.distrib fin-f finite-mult-not-eq-zero-rightI)
```

```
  qed
  show 0 * a = 0
    by transfer (simp add: prod-fun-def [abs-def])
  show a * 0 = 0
    by transfer (simp add: prod-fun-def [abs-def])
qed

end

lemma lookup-mult:
  lookup (f * g) k = (∑ l. lookup f l * (∑ q. lookup g q when k = l + q))
  by transfer (simp add: prod-fun-def)

instance poly-mapping :: (comm-monoid-add, comm-semiring-0) comm-semiring-0
proof
  fix a b c :: 'a ⇒₀ 'b
  show a * b = b * a
  proof transfer
    fix f g :: 'a ⇒ 'b
    assume fin-f: finite {a. f a ≠ 0}
    assume fin-g: finite {b. g b ≠ 0}
    show prod-fun f g = prod-fun g f
    proof
      fix k
      have fin1: ⋀l. finite {a. (f a when k = l + a) ≠ 0}
        using fin-f by auto
      have fin2: ⋀l. finite {b. (g b when k = l + b) ≠ 0}
        using fin-g by auto
      from fin-f fin-g have finite {(a, b). f a ≠ 0 ∧ g b ≠ 0} (is finite ?AB)
        by simp
      have (∑ a. ∑ n. f a * (g n when k = a + n)) = (∑ a. ∑ n. g a * (f n when
k = a + n))
       by (subst Sum-any.swap [OF ‹finite ?AB›]) (auto simp: mult-when ac-simps)
      then show prod-fun f g k = prod-fun g f k
      by (simp add: prod-fun-def Sum-any-right-distrib [OF fin2] Sum-any-right-distrib
[OF fin1])
    qed
  qed
  show (a + b) * c = a * c + b * c
  proof transfer
    fix f g h :: 'a ⇒ 'b
    assume fin-f: finite {k. f k ≠ 0}
    assume fin-g: finite {k. g k ≠ 0}
    assume fin-h: finite {k. h k ≠ 0}
    show prod-fun (λk. f k + g k) h = (λk. prod-fun f h k + prod-fun g h k)
      by (auto simp: prod-fun-def fun-eq-iff algebra-simps
            Sum-any.distrib fin-f fin-g finite-mult-not-eq-zero-rightI)
  qed
qed
```

instance *poly-mapping* :: (*monoid-add*, *semiring-0-cancel*) *semiring-0-cancel*
  ..

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-0-cancel*) *comm-semiring-0-cancel*
  ..

instance *poly-mapping* :: (*monoid-add*, *semiring-1*) *semiring-1*
proof
  fix *a* :: *'a* ⇒$_0$ *'b*
  show *1 * a = a*
    by *transfer* (*simp add*: *prod-fun-def* [*abs-def*] *when-mult*)
  show *a * 1 = a*
    apply *transfer*
   apply (*simp add*: *prod-fun-def* [*abs-def*] *Sum-any-right-distrib Sum-any-left-distrib mult-when*)
    apply (*subst when-commute*)
    apply *simp*
    done
qed

instance *poly-mapping* :: (*comm-monoid-add*, *comm-semiring-1*) *comm-semiring-1*
proof
  fix *a* :: *'a* ⇒$_0$ *'b*
  show *1 * a = a*
    by *transfer* (*simp add*: *prod-fun-def* [*abs-def*])
qed

instance *poly-mapping* :: (*monoid-add*, *semiring-1-cancel*) *semiring-1-cancel*
  ..

instance *poly-mapping* :: (*monoid-add*, *ring*) *ring*
  ..

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring*) *comm-ring*
  ..

instance *poly-mapping* :: (*monoid-add*, *ring-1*) *ring-1*
  ..

instance *poly-mapping* :: (*comm-monoid-add*, *comm-ring-1*) *comm-ring-1*
  ..

79.5 Single-point mappings

lift-definition *single* :: *'a* ⇒ *'b* ⇒ *'a* ⇒$_0$ *'b*::*zero*
  is *λk v k'*. (*v when k = k'*)
  by *simp*

lemma *inj-single* [*iff*]:
inj (*single k*)
proof (*rule injI*, *transfer*)
fix *k* :: *'b* **and** *a b* :: *'a*::*zero*
assume (λ*k'*. *a when k* = *k'*) = (λ*k'*. *b when k* = *k'*)
then have (λ*k'*. *a when k* = *k'*) *k* = (λ*k'*. *b when k* = *k'*) *k*
by (*rule arg-cong*)
then show *a* = *b* **by** *simp*
qed

lemma *lookup-single*:
lookup (*single k v*) *k'* = (*v when k* = *k'*)
by (*simp add*: *single.rep-eq*)

lemma *lookup-single-eq* [*simp*]:
lookup (*single k v*) *k* = *v*
by (*simp add*: *single.rep-eq*)

lemma *lookup-single-not-eq*:
k ≠ *k'* ⟹ *lookup* (*single k v*) *k'* = *0*
by (*simp add*: *single.rep-eq*)

lemma *single-zero* [*simp*]:
single k 0 = *0*
by *transfer simp*

lemma *single-one* [*simp*]:
single 0 1 = *1*
by *transfer simp*

lemma *single-add*:
single k (*a* + *b*) = *single k a* + *single k b*
by *transfer* (*simp add*: *fun-eq-iff when-add-distrib*)

lemma *single-uminus*:
single k (− *a*) = − *single k a*
by *transfer* (*simp add*: *fun-eq-iff when-uminus-distrib*)

lemma *single-diff*:
single k (*a* − *b*) = *single k a* − *single k b*
by *transfer* (*simp add*: *fun-eq-iff when-diff-distrib*)

lemma *single-numeral* [*simp*]:
single 0 (*numeral n*) = *numeral n*
by (*induct n*) (*simp-all only*: *numeral.simps numeral-add single-zero single-one single-add*)

lemma *lookup-numeral*:
lookup (*numeral n*) *k* = (*numeral n when k* = *0*)

proof −
have *lookup* (*numeral n*) *k* = *lookup* (*single 0* (*numeral n*)) *k*
by *simp*
then show *?thesis* **unfolding** *lookup-single* **by** *simp*
qed

lemma *single-of-nat* [*simp*]:
single 0 (*of-nat n*) = *of-nat n*
by (*induct n*) (*simp-all add*: *single-add*)

lemma *lookup-of-nat*:
lookup (*of-nat n*) *k* = (*of-nat n when k* = *0*)
by (*metis lookup-single lookup-single-not-eq single-of-nat*)

lemma *of-nat-single*:
of-nat = *single 0* ∘ *of-nat*
by (*simp add*: *fun-eq-iff*)

lemma *mult-single*:
single k a ∗ *single l b* = *single* (*k* + *l*) (*a* ∗ *b*)
proof *transfer*
fix *k l* :: *'a* **and** *a b* :: *'b*
show *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) = (λ*k'*. *a* ∗ *b when k* + *l* = *k'*)
proof
fix *k'*
have *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) *k'* = (∑ *n*. *a* ∗ *b when l* = *n when k'* = *k* + *n*)
by (*simp add*: *prod-fun-def Sum-any-right-distrib mult-when when-mult*)
also have ... = (∑ *n*. *a* ∗ *b when k'* = *k* + *n when l* = *n*)
by (*simp add*: *when-when conj-commute*)
also have ... = (*a* ∗ *b when k'* = *k* + *l*)
by *simp*
also have ... = (*a* ∗ *b when k* + *l* = *k'*)
by (*simp add*: *when-def*)
finally show *prod-fun* (λ*k'*. *a when k* = *k'*) (λ*k'*. *b when l* = *k'*) *k'* =
(λ*k'*. *a* ∗ *b when k* + *l* = *k'*) *k'* **.**
qed
qed

instance *poly-mapping* :: (*monoid-add*, *semiring-char-0*) *semiring-char-0*
by *intro-classes* (*auto intro*: *inj-compose inj-of-nat simp add*: *of-nat-single*)

instance *poly-mapping* :: (*monoid-add*, *ring-char-0*) *ring-char-0*
..

lemma *single-of-int* [*simp*]:
single 0 (*of-int k*) = *of-int k*
by (*cases k*) (*simp-all add*: *single-diff single-uminus*)

lemma *lookup-of-int*:
lookup (*of-int l*) *k* = (*of-int l when k* = *0*)
by (*metis lookup-single-not-eq single.rep-eq single-of-int*)

79.6 Integral domains

instance *poly-mapping* :: ({*ordered-cancel-comm-monoid-add*, *linorder*}, *semiring-no-zero-divisors*) *semiring-no-zero-divisors*

The *linorder* constraint is a pragmatic device for the proof — maybe it can be dropped

proof
fix *f g* :: *'a* ⇒$_0$ *'b*
assume *f* ≠ *0* **and** *g* ≠ *0*
then show *f* ∗ *g* ≠ *0*
proof *transfer*
fix *f g* :: *'a* ⇒ *'b*
define *F* **where** *F* = {*a*. *f a* ≠ *0*}
moreover define *G* **where** *G* = {*a*. *g a* ≠ *0*}
ultimately have [*simp*]:
⋀*a*. *f a* ≠ *0* ⟷ *a* ∈ *F*
⋀*b*. *g b* ≠ *0* ⟷ *b* ∈ *G*
by *simp-all*
assume *finite* {*a*. *f a* ≠ *0*}
then have [*simp*]: *finite F*
by *simp*
assume *finite* {*a*. *g a* ≠ *0*}
then have [*simp*]: *finite G*
by *simp*
assume *f* ≠ (λ*a*. *0*)
then obtain *a* **where** *f a* ≠ *0*
by (*auto simp*: *fun-eq-iff*)
assume *g* ≠ (λ*b*. *0*)
then obtain *b* **where** *g b* ≠ *0*
by (*auto simp*: *fun-eq-iff*)
from ‹*f a* ≠ *0*› **and** ‹*g b* ≠ *0*› **have** *F* ≠ {} **and** *G* ≠ {}
by *auto*
note *Max-F* = ‹*finite F*› ‹*F* ≠ {}›
note *Max-G* = ‹*finite G*› ‹*G* ≠ {}›
from *Max-F* **and** *Max-G* **have** [*simp*]:
Max F ∈ *F*
Max G ∈ *G*
by *auto*
from *Max-F Max-G* **have** [*dest*!]:
⋀*a*. *a* ∈ *F* ⟹ *a* ≤ *Max F*
⋀*b*. *b* ∈ *G* ⟹ *b* ≤ *Max G*
by *auto*
define *q* **where** *q* = *Max F* + *Max G*

```
    have (∑(a, b). f a * g b when q = a + b) =
      (∑(a, b). f a * g b when q = a + b when a ∈ F ∧ b ∈ G)
      by (rule Sum-any.cong) (auto simp: split-def when-def q-def intro: ccontr)
    also have ... =
      (∑(a, b). f a * g b when (Max F, Max G) = (a, b))
    proof (rule Sum-any.cong)
      fix ab :: 'a × 'a
      obtain a b where [simp]: ab = (a, b)
        by (cases ab) simp-all
      have [dest!]:
        a ≤ Max F ⟹ Max F ≠ a ⟹ a < Max F
        b ≤ Max G ⟹ Max G ≠ b ⟹ b < Max G
        by auto
      show (case ab of (a, b) ⇒ f a * g b when q = a + b when a ∈ F ∧ b ∈ G) =
          (case ab of (a, b) ⇒ f a * g b when (Max F, Max G) = (a, b))
        by (auto simp: split-def when-def q-def dest: add-strict-mono [of a Max F b
Max G])
    qed
    also have ... = (∑ab. (case ab of (a, b) ⇒ f a * g b) when
      (Max F, Max G) = ab)
      unfolding split-def when-def by auto
    also have ... ≠ 0
      by simp
    finally have prod-fun f g q ≠ 0
      by (simp add: prod-fun-unfold-prod)
    then show prod-fun f g ≠ (λk. 0)
      by (auto simp: fun-eq-iff)
  qed
qed

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, ring-no-zero-divisors)
ring-no-zero-divisors
  ..

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, ring-1-no-zero-divisors)
ring-1-no-zero-divisors
  ..

instance poly-mapping :: ({ordered-cancel-comm-monoid-add, linorder}, idom) idom
  ..
```

79.7 Mapping order

```
instantiation poly-mapping :: (linorder, {zero, linorder}) linorder
begin

lift-definition less-poly-mapping :: ('a ⇒₀ 'b) ⇒ ('a ⇒₀ 'b) ⇒ bool
  is less-fun
  .
```

lift-definition *less-eq-poly-mapping* :: ('*a* ⇒$_0$ '*b*) ⇒ ('*a* ⇒$_0$ '*b*) ⇒ *bool*
is λ*f g*. *less-fun f g* ∨ *f* = *g*
.

instance proof (*rule linorder.intro-of-class*)
show *class.linorder* (*less-eq* :: (- ⇒$_0$ -) ⇒ -) *less*
proof (*rule linorder-strictI*, *rule order-strictI*)
fix *f g h* :: '*a* ⇒$_0$ '*b*
show *f* ≤ *g* ⟷ *f* < *g* ∨ *f* = *g*
by *transfer* (*rule refl*)
show ¬ *f* < *f*
by *transfer* (*rule less-fun-irrefl*)
show *f* < *g* ∨ *f* = *g* ∨ *g* < *f*
proof *transfer*
fix *f g* :: '*a* ⇒ '*b*
assume *finite* {*k*. *f k* ≠ *0*} **and** *finite* {*k*. *g k* ≠ *0*}
then have *finite* ({*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*})
by *simp*
moreover have {*k*. *f k* ≠ *g k*} ⊆ {*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*}
by *auto*
ultimately have *finite* {*k*. *f k* ≠ *g k*}
by (*rule rev-finite-subset*)
then show *less-fun f g* ∨ *f* = *g* ∨ *less-fun g f*
by (*rule less-fun-trichotomy*)
qed
assume *f* < *g* **then show** ¬ *g* < *f*
by *transfer* (*rule less-fun-asym*)
note ‹*f* < *g*› **moreover assume** *g* < *h*
ultimately show *f* < *h*
by *transfer* (*rule less-fun-trans*)
qed
qed

end

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *linorder*}) *ordered-ab-semigroup-add*
proof (*intro-classes*, *transfer*)
fix *f g h* :: '*a* ⇒ '*b*
assume ∗: *less-fun f g* ∨ *f* = *g*
have *less-fun* (λ*k*. *h k* + *f k*) (λ*k*. *h k* + *g k*) **if** *less-fun f g*
by (*metis* (*no-types*, *lifting*) *less-fun-def add-strict-left-mono that*)
with ∗ **show** *less-fun* (λ*k*. *h k* + *f k*) (λ*k*. *h k* + *g k*) ∨ (λ*k*. *h k* + *f k*) = (λ*k*. *h k* + *g k*)
by (*auto simp*: *fun-eq-iff*)
qed

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*,

cancel-comm-monoid-add, *linorder*}) *linordered-cancel-ab-semigroup-add*
..

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-comm-monoid-add*
..

instance *poly-mapping* :: (*linorder*, {*ordered-comm-monoid-add*, *ordered-ab-semigroup-add-imp-le*, *cancel-comm-monoid-add*, *linorder*}) *ordered-cancel-comm-monoid-add*
..

instance *poly-mapping* :: (*linorder*, *linordered-ab-group-add*) *linordered-ab-group-add*
..

For pragmatism we leave out the final elements in the hierarchy: *linordered-ring*, *linordered-ring-strict*, *linordered-idom*; remember that the order instance is a mere technical device, not a deeper algebraic property.

79.8 Fundamental mapping notions

lift-definition *keys* :: ('*a* ⇒$_0$ '*b*::*zero*) ⇒ '*a set*
is λ*f*. {*k*. *f k* ≠ *0*} .

lift-definition *range* :: ('*a* ⇒$_0$ '*b*::*zero*) ⇒ '*b set*
is λ*f* :: '*a* ⇒ '*b*. *Set.range f* − {*0*} .

lemma *finite-keys* [*simp*]:
finite (*keys f*)
by *transfer*

lemma *not-in-keys-iff-lookup-eq-zero*:
k ∉ *keys f* ⟷ *lookup f k* = *0*
by *transfer simp*

lemma *lookup-not-eq-zero-eq-in-keys*:
lookup f k ≠ *0* ⟷ *k* ∈ *keys f*
by *transfer simp*

lemma *lookup-eq-zero-in-keys-contradict* [*dest*]:
lookup f k = *0* ⟹ ¬ *k* ∈ *keys f*
by (*simp add*: *not-in-keys-iff-lookup-eq-zero*)

lemma *finite-range* [*simp*]: *finite* (*Poly-Mapping.range p*)
proof *transfer*
fix *f* :: '*b* ⇒ '*a*
assume ∗: *finite* {*x*. *f x* ≠ *0*}
have *Set.range f* − {*0*} ⊆ *f* ‘ {*x*. *f x* ≠ *0*}
by *auto*
thus *finite* (*Set.range f* − {*0*})

using ∗ *finite-surj* **by** *blast*
qed

lemma *in-keys-lookup-in-range* [*simp*]:
k ∈ *keys f* ⟹ *lookup f k* ∈ *range f*
by *transfer simp*

lemma *in-keys-iff*: *x* ∈ (*keys s*) = (*lookup s x* ≠ *0*)
by (*simp add*: *lookup-not-eq-zero-eq-in-keys*)

lemma *keys-zero* [*simp*]:
keys 0 = {}
by *transfer simp*

lemma *range-zero* [*simp*]:
range 0 = {}
by *transfer auto*

lemma *keys-add*:
keys (*f* + *g*) ⊆ *keys f* ∪ *keys g*
by *transfer auto*

lemma *keys-one* [*simp*]:
keys 1 = {*0*}
by *transfer simp*

lemma *range-one* [*simp*]:
range 1 = {*1*}
by *transfer* (*auto simp*: *when-def*)

lemma *keys-single* [*simp*]:
keys (*single k v*) = (*if v* = *0 then* {} *else* {*k*})
by *transfer simp*

lemma *range-single* [*simp*]:
range (*single k v*) = (*if v* = *0 then* {} *else* {*v*})
by *transfer* (*auto simp*: *when-def*)

lemma *keys-mult*:
keys (*f* ∗ *g*) ⊆ {*a* + *b* | *a b*. *a* ∈ *keys f* ∧ *b* ∈ *keys g*}
apply *transfer*
apply (*force simp*: *prod-fun-def dest*!: *mult-not-zero elim*!: *Sum-any.not-neutral-obtains-not-neutral*)
done

lemma *setsum-keys-plus-distrib*:
assumes *hom-0*: ⋀*k*. *f k 0* = *0*
and *hom-plus*: ⋀*k*. *k* ∈ *Poly-Mapping.keys p* ∪ *Poly-Mapping.keys q* ⟹ *f k* (*Poly-Mapping.lookup p k* + *Poly-Mapping.lookup q k*) = *f k* (*Poly-Mapping.lookup p k*) + *f k* (*Poly-Mapping.lookup q k*)

shows
$(\sum k{\in}$*Poly-Mapping.keys* $(p + q).\ f\ k\ ($*Poly-Mapping.lookup* $(p + q)\ k)) =$
$(\sum k{\in}$*Poly-Mapping.keys* $p.\ f\ k\ ($*Poly-Mapping.lookup* $p\ k)) +$
$(\sum k{\in}$*Poly-Mapping.keys* $q.\ f\ k\ ($*Poly-Mapping.lookup* $q\ k))$
(**is** *?lhs* $=$ *?p* $+$ *?q*)
proof −
let *?A* $=$ *Poly-Mapping.keys* $p \cup$ *Poly-Mapping.keys* q
have *?lhs* $= (\sum k{\in}?A.\ f\ k\ ($*Poly-Mapping.lookup* $p\ k +$ *Poly-Mapping.lookup* $q\ k))$
by(*intro sum.mono-neutral-cong-left*) (*auto simp*: *sum.mono-neutral-cong-left hom-0 in-keys-iff lookup-add*)
also have $\ldots = (\sum k{\in}?A.\ f\ k\ ($*Poly-Mapping.lookup* $p\ k) + f\ k\ ($*Poly-Mapping.lookup* $q\ k))$
by(*rule sum.cong*)(*simp-all add*: *hom-plus*)
also have $\ldots = (\sum k{\in}?A.\ f\ k\ ($*Poly-Mapping.lookup* $p\ k)) + (\sum k{\in}?A.\ f\ k\ ($*Poly-Mapping.lookup* $q\ k))$
(**is** - $=$ *?p′* $+$ *?q′*)
by(*simp add*: *sum.distrib*)
also have *?p′* $=$ *?p*
by (*simp add*: *hom-0 in-keys-iff sum.mono-neutral-cong-right*)
also have *?q′* $=$ *?q*
by (*simp add*: *hom-0 in-keys-iff sum.mono-neutral-cong-right*)
finally show *?thesis* **.**
qed

79.9 Degree

definition *degree* :: $(nat \Rightarrow_0 'a{::}zero) \Rightarrow nat$
where
degree f $=$ *Max* (*insert 0* (*Suc* ‘ *keys f*))

lemma *degree-zero* [*simp*]:
degree 0 $=$ *0*
unfolding *degree-def* **by** *transfer simp*

lemma *degree-one* [*simp*]:
degree 1 $=$ *1*
unfolding *degree-def* **by** *transfer simp*

lemma *degree-single-zero* [*simp*]:
degree (*single k 0*) $=$ *0*
unfolding *degree-def* **by** *transfer simp*

lemma *degree-single-not-zero* [*simp*]:
$v \neq 0 \Longrightarrow$ *degree* (*single k v*) $=$ *Suc k*
unfolding *degree-def* **by** *transfer simp*

lemma *degree-zero-iff* [*simp*]:
degree f $= 0 \longleftrightarrow f = 0$

```
unfolding degree-def proof transfer
  fix f :: nat ⇒ 'a
  assume finite {n. f n ≠ 0}
  then have fin: finite (insert 0 (Suc ` {n. f n ≠ 0})) by auto
  show Max (insert 0 (Suc ` {n. f n ≠ 0})) = 0 ⟷ f = (λn. 0) (is ?P ⟷ ?Q)
  proof
    assume ?P
    have {n. f n ≠ 0} = {}
    proof (rule ccontr)
      assume {n. f n ≠ 0} ≠ {}
      then obtain n where n ∈ {n. f n ≠ 0} by blast
      then have {n. f n ≠ 0} = insert n {n. f n ≠ 0} by auto
      then have Suc ` {n. f n ≠ 0} = insert (Suc n) (Suc ` {n. f n ≠ 0}) by auto
       with ‹?P› have Max (insert 0 (insert (Suc n) (Suc ` {n. f n ≠ 0}))) = 0
by simp
      then have Max (insert (Suc n) (insert 0 (Suc ` {n. f n ≠ 0}))) = 0
        by (simp add: insert-commute)
      with fin have max (Suc n) (Max (insert 0 (Suc ` {n. f n ≠ 0}))) = 0
        by simp
      then show False by simp
    qed
    then show ?Q by (simp add: fun-eq-iff)
  next
    assume ?Q then show ?P by simp
  qed
qed

lemma degree-greater-zero-in-keys:
  assumes 0 < degree f
  shows degree f − 1 ∈ keys f
proof −
  from assms have keys f ≠ {}
    by (auto simp: degree-def)
  then show ?thesis unfolding degree-def
    by (simp add: mono-Max-commute [symmetric] mono-Suc)
qed

lemma in-keys-less-degree:
  n ∈ keys f ⟹ n < degree f
unfolding degree-def by transfer (auto simp: Max-gr-iff)

lemma beyond-degree-lookup-zero:
  degree f ≤ n ⟹ lookup f n = 0
  unfolding degree-def by transfer auto

lemma degree-add:
  degree (f + g) ≤ max (degree f) (Poly-Mapping.degree g)
unfolding degree-def proof transfer
  fix f g :: nat ⇒ 'a
```

assume *f*: *finite* {*x*. *f x* ≠ *0*}
assume *g*: *finite* {*x*. *g x* ≠ *0*}
let *?f* = *Max* (*insert 0* (*Suc* ‘ {*k*. *f k* ≠ *0*}))
let *?g* = *Max* (*insert 0* (*Suc* ‘ {*k*. *g k* ≠ *0*}))
have *Max* (*insert 0* (*Suc* ‘ {*k*. *f k* + *g k* ≠ *0*})) ≤ *Max* (*insert 0* (*Suc* ‘ ({*k*. *f k* ≠ *0*} ∪ {*k*. *g k* ≠ *0*})))
by (*rule Max.subset-imp*) (*insert f g*, *auto*)
also have ... = *max ?f ?g*
using *f g* **by** (*simp-all add*: *image-Un Max-Un* [*symmetric*])
finally show *Max* (*insert 0* (*Suc* ‘ {*k*. *f k* + *g k* ≠ *0*}))
≤ *max* (*Max* (*insert 0* (*Suc* ‘ {*k*. *f k* ≠ *0*}))) (*Max* (*insert 0* (*Suc* ‘ {*k*. *g k* ≠ *0*})))
.
qed

lemma *sorted-list-of-set-keys*:
sorted-list-of-set (*keys f*) = *filter* (λ*k*. *k* ∈ *keys f*) [*0*..<*degree f*] (**is** - = *?r*)
proof −
have *keys f* = *set ?r*
by (*auto dest*: *in-keys-less-degree*)
moreover have *sorted-list-of-set* (*set ?r*) = *?r*
unfolding *sorted-list-of-set-sort-remdups*
by (*simp add*: *remdups-filter filter-sort* [*symmetric*])
ultimately show *?thesis* **by** *simp*
qed

79.10 Inductive structure

lift-definition *update* :: *'a* ⇒ *'b* ⇒ (*'a* ⇒$_0$ *'b*::*zero*) ⇒ *'a* ⇒$_0$ *'b*
is λ*k v f*. *f*(*k* := *v*)
proof −
fix *f* :: *'a* ⇒ *'b* **and** *k' v*
assume *finite* {*k*. *f k* ≠ *0*}
then have *finite* (*insert k'* {*k*. *f k* ≠ *0*})
by *simp*
then show *finite* {*k*. (*f*(*k'* := *v*)) *k* ≠ *0*}
by (*rule rev-finite-subset*) *auto*
qed

lemma *update-induct* [*case-names const update*]:
assumes *const'*: *P 0*
assumes *update'*: ⋀*f a b*. *a* ∉ *keys f* ⟹ *b* ≠ *0* ⟹ *P f* ⟹ *P* (*update a b f*)
shows *P f*
proof −
obtain *g* **where** *f* = *Abs-poly-mapping g* **and** *finite* {*a*. *g a* ≠ *0*}
by (*cases f*) *simp-all*
define *Q* **where** *Q g* = *P* (*Abs-poly-mapping g*) **for** *g*
from ‹*finite* {*a*. *g a* ≠ *0*}› **have** *Q g*
proof (*induct g rule*: *finite-update-induct*)

```
    case const with const' Q-def show ?case
      by simp
  next
    case (update a b g)
    from ‹finite {a. g a ≠ 0}› ‹g a = 0› have a ∉ keys (Abs-poly-mapping g)
      by (simp add: Abs-poly-mapping-inverse keys.rep-eq)
    moreover note ‹b ≠ 0›
    moreover from ‹Q g› have P (Abs-poly-mapping g)
      by (simp add: Q-def)
    ultimately have P (update a b (Abs-poly-mapping g))
      by (rule update')
    also from ‹finite {a. g a ≠ 0}›
    have update a b (Abs-poly-mapping g) = Abs-poly-mapping (g(a := b))
      by (simp add: update.abs-eq eq-onp-same-args)
    finally show ?case
      by (simp add: Q-def fun-upd-def)
  qed
  then show ?thesis by (simp add: Q-def ‹f = Abs-poly-mapping g›)
qed

lemma lookup-update:
  lookup (update k v f) k' = (if k = k' then v else lookup f k')
  by transfer simp

lemma keys-update:
  keys (update k v f) = (if v = 0 then keys f - {k} else insert k (keys f))
  by transfer auto
```

79.11 Quasi-functorial structure

```
lift-definition map :: ('b::zero ⇒ 'c::zero)
  ⇒ ('a ⇒₀ 'b) ⇒ ('a ⇒₀ 'c::zero)
  is λg f k. g (f k) when f k ≠ 0
  by simp

context
  fixes f :: 'b ⇒ 'a
  assumes inj-f: inj f
begin

lift-definition map-key :: ('a ⇒₀ 'c::zero) ⇒ 'b ⇒₀ 'c
  is λp. p ∘ f
proof -
  fix g :: 'c ⇒ 'd and p :: 'a ⇒ 'c
  assume finite {x. p x ≠ 0}
  hence finite (f ‘ {y. p (f y) ≠ 0})
    by (simp add: rev-finite-subset subset-eq)
  thus finite {x. (p ∘ f) x ≠ 0} unfolding o-def
    by (metis finite-imageD injD inj-f inj-on-def)
```

qed

end

lemma *map-key-compose*:
assumes [*transfer-rule*]: *inj f inj g*
shows *map-key f* (*map-key g p*) = *map-key* (*g* ∘ *f*) *p*
proof −
from *assms* **have** [*transfer-rule*]: *inj* (*g* ∘ *f*)
by(*simp add*: *inj-compose*)
show *?thesis* **by** *transfer*(*simp add*: *o-assoc*)
qed

lemma *map-key-id*:
map-key (λ*x*. *x*) *p* = *p*
proof −
have [*transfer-rule*]: *inj* (λ*x*. *x*) **by** *simp*
show *?thesis* **by** *transfer*(*simp add*: *o-def*)
qed

context
fixes *f* :: *'a* ⇒ *'b*
assumes *inj-f* [*transfer-rule*]: *inj f*
begin

lemma *map-key-map*:
map-key f (*map g p*) = *map g* (*map-key f p*)
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *map-key-plus*:
map-key f (*p* + *q*) = *map-key f p* + *map-key f q*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *keys-map-key*:
keys (*map-key f p*) = *f* −‘ *keys p*
by *transfer auto*

lemma *map-key-zero* [*simp*]:
map-key f 0 = *0*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *map-key-single* [*simp*]:
map-key f (*single* (*f k*) *v*) = *single k v*
by *transfer* (*simp add*: *fun-eq-iff inj-onD* [*OF inj-f*] *when-def*)

end

lemma *mult-map-scale-conv-mult*: *map* ((∗) *s*) *p* = *single 0 s* ∗ *p*
proof(*transfer fixing*: *s*)

fix $p :: 'a \Rightarrow 'b$
assume $*$: *finite* $\{x.\ p\ x \neq 0\}$
have *prod-fun* $(\lambda k'.\ s\ when\ 0 = k')\ p\ x = (\lambda k.\ s * p\ k\ when\ p\ k \neq 0)\ x$ (**is** *?lhs* = *?rhs*) **for** x
proof −
have *?lhs* $= (\sum l :: 'a.\ if\ l = 0\ then\ s * (\sum q.\ p\ q\ when\ x = q)\ else\ 0)$
by (*auto simp*: *prod-fun-def when-def intro*: *Sum-any.cong simp del*: *Sum-any.delta*)
also have ... = *?rhs*
by (*simp add*: *when-def*)
finally show *?thesis* **.**
qed
then show $(\lambda k.\ s * p\ k\ when\ p\ k \neq 0) = prod\text{-}fun\ (\lambda k'.\ s\ when\ 0 = k')\ p$
by(*simp add*: *fun-eq-iff*)
qed

lemma *map-single* [*simp*]:
$(c = 0 \Longrightarrow f\ 0 = 0) \Longrightarrow map\ f\ (single\ x\ c) = single\ x\ (f\ c)$
by *transfer*(*auto simp*: *fun-eq-iff when-def*)

lemma *map-eq-zero-iff*: $map\ f\ p = 0 \longleftrightarrow (\forall k \in keys\ p.\ f\ (lookup\ p\ k) = 0)$
by *transfer*(*auto simp*: *fun-eq-iff when-def*)

79.12 Canonical dense representation of $nat \Rightarrow_0 'a$

abbreviation *no-trailing-zeros* :: $'a$ :: *zero list* ⇒ *bool*
where
no-trailing-zeros ≡ *no-trailing* ((=) *0*)

lift-definition *nth* :: $'a$ *list* ⇒ ($nat \Rightarrow_0 'a$::*zero*)
is *nth-default 0*
by (*fact finite-nth-default-neq-default*)

The opposite direction is directly specified on (later) type *nat-mapping*.

lemma *nth-Nil* [*simp*]:
nth [] = *0*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *nth-singleton* [*simp*]:
nth [*v*] = *single 0 v*
proof (*transfer*, *rule ext*)
fix *n* :: *nat* **and** *v* :: $'a$
show *nth-default 0* [*v*] *n* = (*v when 0* = *n*)
by (*auto simp*: *nth-default-def nth-append*)
qed

lemma *nth-replicate* [*simp*]:
nth (*replicate n 0* @ [*v*]) = *single n v*
proof (*transfer*, *rule ext*)
fix *m n* :: *nat* **and** *v* :: $'a$
show *nth-default 0* (*replicate n 0* @ [*v*]) *m* = (*v when n* = *m*)

by (*auto simp*: *nth-default-def nth-append*)
qed

lemma *nth-strip-while* [*simp*]:
nth (*strip-while* ((=) *0*) *xs*) = *nth xs*
by *transfer* (*fact nth-default-strip-while-dflt*)

lemma *nth-strip-while′* [*simp*]:
nth (*strip-while* (λ*k*. *k* = *0*) *xs*) = *nth xs*
by (*subst eq-commute*) (*fact nth-strip-while*)

lemma *nth-eq-iff*:
nth xs = *nth ys* ⟷ *strip-while* (*HOL.eq 0*) *xs* = *strip-while* (*HOL.eq 0*) *ys*
by *transfer* (*simp add*: *nth-default-eq-iff*)

lemma *lookup-nth* [*simp*]:
lookup (*nth xs*) = *nth-default 0 xs*
by (*fact nth.rep-eq*)

lemma *keys-nth* [*simp*]:
keys (*nth xs*) = *fst* ‘ {(*n*, *v*) ∈ *set* (*indexed-from 0 xs*). *v* ≠ *0*}
proof *transfer*
fix *xs* :: *′a list*
have *n* ∈ *fst* ‘ {(*n*, *v*). (*n*, *v*) ∈ *set* (*indexed-from 0 xs*) ∧ *v* ≠ *0*}
if *nth-default 0 xs n* ≠ *0* **for** *n*
proof −
from *that* **have** *n* < *length xs* **and** *xs* ! *n* ≠ *0*
by (*auto simp*: *nth-default-def split*: *if-splits*)
then have (*n*, *xs* ! *n*) ∈ {(*n*, *v*). (*n*, *v*) ∈ *set* (*indexed-from 0 xs*) ∧ *v* ≠ *0*} (**is** *?x* ∈ *?A*)
by (*auto simp*: *in-set-conv-nth indexed-from-eq-zip*)
then have *fst ?x* ∈ *fst* ‘ *?A*
by *blast*
then show *?thesis*
by *simp*
qed
then show {*k*. *nth-default 0 xs k* ≠ *0*} = *fst* ‘ {(*n*, *v*). (*n*, *v*) ∈ *set* (*indexed-from 0 xs*) ∧ *v* ≠ *0*}
by (*auto simp*: *in-indexed-from-iff-nth-default-eq*)
qed

lemma *range-nth* [*simp*]:
range (*nth xs*) = *set xs* − {*0*}
by *transfer simp*

lemma *degree-nth*:
no-trailing-zeros xs ⟹ *degree* (*nth xs*) = *length xs*
unfolding *degree-def* **proof** *transfer*
fix *xs* :: *′a list*

```
  assume ∗: no-trailing-zeros xs
  let ?A = {n. nth-default 0 xs n ≠ 0}
  let ?f = nth-default 0 xs
  let ?bound = Max (insert 0 (Suc ` {n. ?f n ≠ 0}))
  show ?bound = length xs
  proof (cases xs = [])
    case False
    with ∗ obtain n where n: n < length xs xs ! n ≠ 0
      by (fastforce simp add: no-trailing-unfold last-conv-nth neq-Nil-conv)
    then have ?bound = Max (Suc ` {k. (k < length xs ⟶ xs ! k ≠ (0::'a)) ∧ k
< length xs})
      by (subst Max-insert) (auto simp: nth-default-def)
    also let ?A = {k. k < length xs ∧ xs ! k ≠ 0}
    have {k. (k < length xs ⟶ xs ! k ≠ (0::'a)) ∧ k < length xs} = ?A by auto
    also have Max (Suc ` ?A) = Suc (Max ?A) using n
       by (subst mono-Max-commute [where f = Suc, symmetric]) (auto simp:
mono-Suc)
    also {
      have Max ?A ∈ ?A using n Max-in [of ?A] by fastforce
      hence Suc (Max ?A) ≤ length xs by simp
      moreover from ∗ False have length xs − 1 ∈ ?A
        by(auto simp: no-trailing-unfold last-conv-nth)
      hence length xs − 1 ≤ Max ?A using Max-ge[of ?A length xs − 1] by auto
      hence length xs ≤ Suc (Max ?A) by simp
      ultimately have Suc (Max ?A) = length xs by simp }
    finally show ?thesis .
  qed simp
qed

lemma nth-trailing-zeros [simp]:
  nth (xs @ replicate n 0) = nth xs
  by (simp add: nth.abs-eq)

lemma nth-idem:
  nth (List.map (lookup f) [0..<degree f]) = f
  unfolding degree-def by transfer
    (auto simp: nth-default-def fun-eq-iff not-less)

lemma nth-idem-bound:
  assumes degree f ≤ n
  shows nth (List.map (lookup f) [0..<n]) = f
proof −
  from assms obtain m where n = degree f + m
    by (blast dest: le-Suc-ex)
  then have [0..<n] = [0..<degree f] @ [degree f..<degree f + m]
    by (simp add: upt-add-eq-append [of 0])
  moreover have List.map (lookup f) [degree f..<degree f + m] = replicate m 0
    by (rule replicate-eqI) (auto simp: beyond-degree-lookup-zero)
  ultimately show ?thesis by (simp add: nth-idem)
```

qed

79.13 Canonical sparse representation of *'a* $\Rightarrow_0$ *'b*

lift-definition *the-value* :: (*'a* × *'b*) *list* ⇒ *'a* $\Rightarrow_0$ *'b*::*zero*
  is *λxs k. case map-of xs k of None ⇒ 0 | Some v ⇒ v*
proof −
  fix *xs* :: (*'a* × *'b*) *list*
  have *fin*: *finite* {*k*. ∃ *v*. *map-of xs k* = *Some v*}
    using *finite-dom-map-of* [*of xs*] **unfolding** *dom-def* **by** *auto*
  then show *finite* {*k*. (*case map-of xs k of None ⇒ 0 | Some v ⇒ v*) ≠ *0*}
    using *fin* **by** (*simp split*: *option.split*)
qed

definition *items* :: (*'a*::*linorder* $\Rightarrow_0$ *'b*::*zero*) ⇒ (*'a* × *'b*) *list*
where
  items f = *List.map* (*λk*. (*k*, *lookup f k*)) (*sorted-list-of-set* (*keys f*))

For the canonical sparse representation we provide both directions of morphisms since the specification of ordered association lists in theory *OAL-ist* will support arbitrary linear orders *linorder* as keys, not just natural numbers *nat*.

lemma *the-value-items* [*simp*]:
  the-value (*items f*) = *f*
  unfolding *items-def*
  by *transfer* (*simp add*: *fun-eq-iff map-of-map-restrict restrict-map-def*)

lemma *lookup-the-value*:
  lookup (*the-value xs*) *k* = (*case map-of xs k of None ⇒ 0 | Some v ⇒ v*)
  by (*simp add*: *the-value.rep-eq*)

lemma *items-the-value*:
  assumes *sorted* (*List.map fst xs*) **and** *distinct* (*List.map fst xs*) **and** *0* ∉ *snd* ‘ *set xs*
  shows *items* (*the-value xs*) = *xs*
proof −
  from *assms* **have** *sorted-list-of-set* (*set* (*List.map fst xs*)) = *List.map fst xs*
    unfolding *sorted-list-of-set-sort-remdups* **by** (*simp add*: *distinct-remdups-id sort-key-id-if-sorted*)
  moreover from *assms* **have** *keys* (*the-value xs*) = *fst* ‘ *set xs*
    by *transfer* (*auto simp*: *image-def split*: *option.split dest*: *set-map-of-compr*)
  ultimately show *?thesis*
    unfolding *items-def* **using** *assms*
    by (*auto simp*: *lookup-the-value intro*: *map-idI*)
qed

lemma *the-value-Nil* [*simp*]:
  the-value [] = *0*
  by *transfer* (*simp add*: *fun-eq-iff*)

lemma *the-value-Cons* [*simp*]:
the-value (*x* # *xs*) = *update* (*fst x*) (*snd x*) (*the-value xs*)
by *transfer* (*simp add*: *fun-eq-iff*)

lemma *items-zero* [*simp*]:
items 0 = []
unfolding *items-def* **by** *simp*

lemma *items-one* [*simp*]:
items 1 = [(*0*, *1*)]
unfolding *items-def* **by** *transfer simp*

lemma *items-single* [*simp*]:
items (*single k v*) = (*if v* = *0 then* [] *else* [(*k*, *v*)])
unfolding *items-def* **by** *simp*

lemma *in-set-items-iff* [*simp*]:
(*k*, *v*) ∈ *set* (*items f*) ⟷ *k* ∈ *keys f* ∧ *lookup f k* = *v*
unfolding *items-def* **by** *transfer auto*

79.14 Size estimation

context
fixes *f* :: *'a* ⇒ *nat*
and *g* :: *'b* :: *zero* ⇒ *nat*
begin

definition *poly-mapping-size* :: (*'a* ⇒$_0$ *'b*) ⇒ *nat*
where
poly-mapping-size m = *g 0* + (∑ *k* ∈ *keys m*. *Suc* (*f k* + *g* (*lookup m k*)))

lemma *poly-mapping-size-0* [*simp*]:
poly-mapping-size 0 = *g 0*
by (*simp add*: *poly-mapping-size-def*)

lemma *poly-mapping-size-single* [*simp*]:
poly-mapping-size (*single k v*) = (*if v* = *0 then g 0 else g 0* + *f k* + *g v* + *1*)
unfolding *poly-mapping-size-def* **by** *transfer simp*

lemma *keys-less-poly-mapping-size*:
k ∈ *keys m* ⟹ *f k* + *g* (*lookup m k*) < *poly-mapping-size m*
unfolding *poly-mapping-size-def*
proof *transfer*
fix *k* :: *'a* **and** *m* :: *'a* ⇒ *'b* **and** *f* :: *'a* ⇒ *nat* **and** *g*
let *?keys* = {*k*. *m k* ≠ *0*}
assume ∗: *finite ?keys k* ∈ *?keys*
then have *f k* + *g* (*m k*) = (∑ *k'* ∈ *?keys*. *f k'* + *g* (*m k'*) *when k'* = *k*)
by (*simp add*: *sum.delta when-def*)

also have $\ldots < (\sum k' \in$ *?keys*. *Suc* $(f\ k' + g\ (m\ k')))$ **using** $*$
by (*intro sum-strict-mono*) (*auto simp*: *when-def*)
also have $\ldots \leq g\ 0 + \ldots$ **by** *simp*
finally have $f\ k + g\ (m\ k) < \ldots$ **.**
then show $f\ k + g\ (m\ k) < g\ 0 + (\sum k \mid m\ k \neq 0.\ Suc\ (f\ k + g\ (m\ k)))$
by *simp*
qed

lemma *lookup-le-poly-mapping-size*:
$g\ (lookup\ m\ k) \leq$ *poly-mapping-size* m
proof (*cases* $k \in$ *keys* m)
case *True*
with *keys-less-poly-mapping-size* [*of* $k\ m$]
show *?thesis* **by** *simp*
next
case *False*
then show *?thesis*
by (*simp add*: *Poly-Mapping.poly-mapping-size-def in-keys-iff*)
qed

lemma *poly-mapping-size-estimation*:
$k \in keys\ m \Longrightarrow y \leq f\ k + g\ (lookup\ m\ k) \Longrightarrow y <$ *poly-mapping-size* m
using *keys-less-poly-mapping-size* **by** (*auto intro*: *le-less-trans*)

lemma *poly-mapping-size-estimation2*:
assumes $v \in range\ m$ **and** $y \leq g\ v$
shows $y <$ *poly-mapping-size* m
proof −
from *assms* **obtain** k **where** $*$: $lookup\ m\ k = v\ v \neq 0$
by *transfer blast*
then have $k \in keys\ m$
by (*simp add*: *in-keys-iff*)
with $*$ **show** *?thesis*
by (*simp add*: *Poly-Mapping.poly-mapping-size-estimation assms*(*2*) *trans-le-add2*)
qed

end

lemma *poly-mapping-size-one* [*simp*]:
poly-mapping-size f g $1 = g\ 0 + f\ 0 + g\ 1 + 1$
unfolding *poly-mapping-size-def* **by** *transfer simp*

lemma *poly-mapping-size-cong* [*fundef-cong*]:
$m = m' \Longrightarrow g\ 0 = g'\ 0 \Longrightarrow (\bigwedge k.\ k \in keys\ m' \Longrightarrow f\ k = f'\ k)$
$\Longrightarrow (\bigwedge v.\ v \in range\ m' \Longrightarrow g\ v = g'\ v)$
$\Longrightarrow$ *poly-mapping-size f g* $m =$ *poly-mapping-size* $f'\ g'\ m'$
by (*auto simp*: *poly-mapping-size-def intro*!: *sum.cong*)

instantiation *poly-mapping* :: (*type*, *zero*) *size*

begin

definition *size* = *poly-mapping-size* (λ-. *0*) (λ-. *0*)

instance ..

end

79.15 Further mapping operations and properties

It is like in algebra: there are many definitions, some are also used

lift-definition *mapp* ::
(*'a* ⇒ *'b* :: *zero* ⇒ *'c* :: *zero*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'c*)
is λ*f p k*. (*if k* ∈ *keys p then f k* (*lookup p k*) *else 0*)
by *simp*

lemma *mapp-cong* [*fundef-cong*]:
⟦ *m* = *m'*; ⋀*k*. *k* ∈ *keys m'* ⟹ *f k* (*lookup m' k*) = *f' k* (*lookup m' k*) ⟧
⟹ *mapp f m* = *mapp f' m'*
by *transfer* (*auto simp*: *fun-eq-iff*)

lemma *lookup-mapp*:
lookup (*mapp f p*) *k* = (*f k* (*lookup p k*) *when k* ∈ *keys p*)
by (*simp add*: *mapp.rep-eq*)

lemma *keys-mapp-subset*: *keys* (*mapp f p*) ⊆ *keys p*
by (*meson in-keys-iff mapp.rep-eq subsetI*)

instantiation *poly-mapping* :: (*type*, {*zero*, *equal*}) *equal*
begin

definition *equal-poly-mapping* :: ‹(*'a* ⇒$_0$ *'b*) ⇒ (*'a* ⇒$_0$ *'b*) ⇒ *bool*›
where ‹*HOL.equal f g* ⟷
(*let K* = *keys f in K* = *keys g* ∧ (∀ *k*∈*K*. *HOL.equal* (*lookup f k*) (*lookup g k*)))›
for *f g* :: ‹*'a* ⇒$_0$ *'b*›

instance
apply *standard*
apply (*auto simp add*: *equal-poly-mapping-def equal Let-def intro*!: *poly-mapping-eqI*)
apply *transfer*
apply *auto*
done

end

79.16 Free Abelian Groups Over a Type

abbreviation *frag-of* :: *'a* ⇒ *'a* ⇒$_0$ *int*
where *frag-of c* ≡ *Poly-Mapping.single c* (*1*::*int*)

lemma *lookup-frag-of* [*simp*]:
Poly-Mapping.lookup(*frag-of c*) = (λ*x*. *if x* = *c then 1 else 0*)
by (*force simp add*: *lookup-single-not-eq*)

lemma *frag-of-nonzero* [*simp*]: *frag-of a* ≠ *0*
by (*metis lookup-single-eq lookup-zero zero-neq-one*)

definition *frag-cmul* :: *int* ⇒ (′*a* ⇒$_0$ *int*) ⇒ (′*a* ⇒$_0$ *int*)
where *frag-cmul c a* = *Abs-poly-mapping* (λ*x*. *c* ∗ *Poly-Mapping.lookup a x*)

lemma *frag-cmul-zero* [*simp*]: *frag-cmul 0 x* = *0*
by (*simp add*: *frag-cmul-def*)

lemma *frag-cmul-zero2* [*simp*]: *frag-cmul c 0* = *0*
by (*simp add*: *frag-cmul-def*)

lemma *frag-cmul-one* [*simp*]: *frag-cmul 1 x* = *x*
by (*simp add*: *frag-cmul-def*)

lemma *frag-cmul-minus-one* [*simp*]: *frag-cmul* (−*1*) *x* = −*x*
by (*simp add*: *frag-cmul-def uminus-poly-mapping-def poly-mapping-eqI*)

lemma *frag-cmul-cmul* [*simp*]: *frag-cmul c* (*frag-cmul d x*) = *frag-cmul* (*c*∗*d*) *x*
by (*simp add*: *frag-cmul-def mult-ac*)

lemma *lookup-frag-cmul* [*simp*]: *poly-mapping.lookup* (*frag-cmul c x*) *i* = *c* ∗ *poly-mapping.lookup x i*
by (*simp add*: *frag-cmul-def*)

lemma *minus-frag-cmul* [*simp*]: − *frag-cmul k x* = *frag-cmul* (−*k*) *x*
by (*simp add*: *poly-mapping-eqI*)

lemma *keys-frag-of*: *Poly-Mapping.keys*(*frag-of a*) = {*a*}
by *simp*

lemma *finite-cmul-nonzero*: *finite* {*x*. *c* ∗ *Poly-Mapping.lookup a x* ≠ (*0*::*int*)}
by *simp*

lemma *keys-cmul*: *Poly-Mapping.keys*(*frag-cmul c a*) ⊆ *Poly-Mapping.keys a*
using *finite-cmul-nonzero* [*of c a*]
by (*metis lookup-frag-cmul mult-zero-right not-in-keys-iff-lookup-eq-zero subsetI*)

lemma *keys-cmul-iff* [*iff*]: *i* ∈ *Poly-Mapping.keys* (*frag-cmul c x*) ⟷ *i* ∈ *Poly-Mapping.keys x* ∧ *c* ≠ *0*
by (*metis in-keys-iff lookup-frag-cmul mult-eq-0-iff*)

lemma *keys-minus* [*simp*]: *Poly-Mapping.keys*(−*a*) = *Poly-Mapping.keys a*

by (*metis* (*no-types*, *opaque-lifting*) *in-keys-iff lookup-uminus neg-equal-0-iff-equal subsetI subset-antisym*)

lemma *keys-diff*:
Poly-Mapping.keys(*a* − *b*) ⊆ *Poly-Mapping.keys a* ∪ *Poly-Mapping.keys b*
by (*auto simp*: *in-keys-iff lookup-minus*)

lemma *keys-eq-empty* [*simp*]: *Poly-Mapping.keys c* = {} ⟷ *c* = *0*
by (*metis in-keys-iff keys-zero lookup-zero poly-mapping-eqI*)

lemma *frag-cmul-eq-0-iff* [*simp*]: *frag-cmul k c* = *0* ⟷ *k*=*0* ∨ *c*=*0*
by *auto* (*metis subsetI subset-antisym keys-cmul-iff keys-eq-empty*)

lemma *frag-of-eq*: *frag-of x* = *frag-of y* ⟷ *x* = *y*
by (*metis lookup-single-eq lookup-single-not-eq zero-neq-one*)

lemma *frag-cmul-distrib*: *frag-cmul* (*c*+*d*) *a* = *frag-cmul c a* + *frag-cmul d a*
by (*simp add*: *frag-cmul-def plus-poly-mapping-def int-distrib*)

lemma *frag-cmul-distrib2*: *frag-cmul c* (*a*+*b*) = *frag-cmul c a* + *frag-cmul c b*
by (*simp add*: *int-distrib*(*2*) *lookup-add poly-mapping-eqI*)

lemma *frag-cmul-diff-distrib*: *frag-cmul* (*a* − *b*) *c* = *frag-cmul a c* − *frag-cmul b c*
by (*auto simp*: *left-diff-distrib lookup-minus poly-mapping-eqI*)

lemma *frag-cmul-sum*:
frag-cmul a (*sum b I*) = (∑ *i*∈*I*. *frag-cmul a* (*b i*))
proof (*induction rule*: *infinite-finite-induct*)
case (*insert i I*)
then show *?case*
by (*auto simp*: *algebra-simps frag-cmul-distrib2*)
qed *auto*

lemma *keys-sum*: *Poly-Mapping.keys*(*sum b I*) ⊆ (⋃ *i* ∈*I*. *Poly-Mapping.keys*(*b i*))
proof (*induction I rule*: *infinite-finite-induct*)
case (*insert i I*)
then show *?case*
using *keys-add* [*of b i sum b I*] **by** *auto*
qed *auto*

definition *frag-extend* :: ('*b* ⇒ '*a* ⇒$_0$ *int*) ⇒ ('*b* ⇒$_0$ *int*) ⇒ '*a* ⇒$_0$ *int*
where *frag-extend b x* ≡ (∑ *i* ∈ *Poly-Mapping.keys x*. *frag-cmul* (*Poly-Mapping.lookup x i*) (*b i*))

lemma *frag-extend-0* [*simp*]: *frag-extend b 0* = *0*
by (*simp add*: *frag-extend-def*)

lemma *frag-extend-of* [*simp*]: *frag-extend f* (*frag-of a*) = *f a*
by (*simp add*: *frag-extend-def*)

lemma *frag-extend-cmul*:
frag-extend f (*frag-cmul c x*) = *frag-cmul c* (*frag-extend f x*)
by (*auto simp*: *frag-extend-def frag-cmul-sum intro*: *sum.mono-neutral-cong-left*)

lemma *frag-extend-minus*:
frag-extend f (− *x*) = − (*frag-extend f x*)
using *frag-extend-cmul* [*of f* −*1*] **by** *simp*

lemma *frag-extend-add*:
frag-extend f (*a*+*b*) = (*frag-extend f a*) + (*frag-extend f b*)
proof −
have ∗: (∑ *i*∈*Poly-Mapping.keys a*. *frag-cmul* (*poly-mapping.lookup a i*) (*f i*))
= (∑ *i*∈*Poly-Mapping.keys a* ∪ *Poly-Mapping.keys b*. *frag-cmul* (*poly-mapping.lookup a i*) (*f i*))
(∑ *i*∈*Poly-Mapping.keys b*. *frag-cmul* (*poly-mapping.lookup b i*) (*f i*))
= (∑ *i*∈*Poly-Mapping.keys a* ∪ *Poly-Mapping.keys b*. *frag-cmul* (*poly-mapping.lookup b i*) (*f i*))
by (*auto simp*: *in-keys-iff intro*: *sum.mono-neutral-cong-left*)
have *frag-extend f* (*a*+*b*) = (∑ *i*∈*Poly-Mapping.keys* (*a* + *b*).
frag-cmul (*poly-mapping.lookup a i*) (*f i*) + *frag-cmul* (*poly-mapping.lookup b i*) (*f i*))
by (*auto simp*: *frag-extend-def Poly-Mapping.lookup-add frag-cmul-distrib*)
also have ... = (∑ *i* ∈ *Poly-Mapping.keys a* ∪ *Poly-Mapping.keys b*. *frag-cmul* (*poly-mapping.lookup a i*) (*f i*)
+ *frag-cmul* (*poly-mapping.lookup b i*) (*f i*))
proof (*rule sum.mono-neutral-cong-left*)
show ∀ *i*∈*keys a* ∪ *keys b* − *keys* (*a* + *b*).
frag-cmul (*lookup a i*) (*f i*) + *frag-cmul* (*lookup b i*) (*f i*) = *0*
by (*metis DiffD2 frag-cmul-distrib frag-cmul-zero in-keys-iff lookup-add*)
qed (*auto simp*: *keys-add*)
also have ... = (*frag-extend f a*) + (*frag-extend f b*)
by (*auto simp*: ∗ *sum.distrib frag-extend-def*)
finally show *?thesis* **.**
qed

lemma *frag-extend-diff*:
frag-extend f (*a*−*b*) = (*frag-extend f a*) − (*frag-extend f b*)
by (*metis* (*no-types*, *opaque-lifting*) *add-uminus-conv-diff frag-extend-add frag-extend-minus*)

lemma *frag-extend-sum*:
finite I ⟹ *frag-extend f* (∑ *i*∈*I*. *g i*) = *sum* (*frag-extend f* ∘ *g*) *I*
by (*induction I rule*: *finite-induct*) (*simp-all add*: *frag-extend-add*)

lemma *frag-extend-eq*:
(⋀*f*. *f* ∈ *Poly-Mapping.keys c* ⟹ *g f* = *h f*) ⟹ *frag-extend g c* = *frag-extend h c*

by (*simp add*: *frag-extend-def*)

lemma *frag-extend-eq-0*:
(⋀*x*. *x* ∈ *Poly-Mapping.keys c* ⟹ *f x* = *0*) ⟹ *frag-extend f c* = *0*
by (*simp add*: *frag-extend-def*)

lemma *keys-frag-extend*: *Poly-Mapping.keys*(*frag-extend f c*) ⊆ (⋃ *x* ∈ *Poly-Mapping.keys c*. *Poly-Mapping.keys*(*f x*))
unfolding *frag-extend-def*
using *keys-sum* **by** *fastforce*

lemma *frag-expansion*: *a* = *frag-extend frag-of a*
proof −
have ∗: *finite I*
⟹ *Poly-Mapping.lookup* (∑ *i*∈*I*. *frag-cmul* (*Poly-Mapping.lookup a i*) (*frag-of i*)) *j* =
(*if j* ∈ *I then Poly-Mapping.lookup a j else 0*) **for** *I j*
by (*induction I rule*: *finite-induct*) (*auto simp*: *lookup-single lookup-add*)
show *?thesis*
unfolding *frag-extend-def*
by (*rule poly-mapping-eqI*) (*fastforce simp add*: *in-keys-iff* ∗)
qed

lemma *frag-closure-minus-cmul*:
assumes *P 0* **and** *P*: ⋀*x y*. ⟦*P x*; *P y*⟧ ⟹ *P*(*x* − *y*) *P c*
shows *P*(*frag-cmul k c*)
proof −
have *P* (*frag-cmul* (*int n*) *c*) **for** *n*
proof (*induction n*)
case *0*
then show *?case*
by (*simp add*: *assms*)
next
case (*Suc n*)
then show *?case*
by (*metis assms diff-0 diff-minus-eq-add frag-cmul-distrib frag-cmul-one of-nat-Suc*)
qed
then show *?thesis*
by (*metis* (*no-types*, *opaque-lifting*) *add-diff-eq assms*(*2*) *diff-add-cancel frag-cmul-distrib int-diff-cases*)
qed

lemma *frag-induction* [*consumes 1*, *case-names zero one diff*]:
assumes *supp*: *Poly-Mapping.keys c* ⊆ *S*
and *0*: *P 0* **and** *sing*: ⋀*x*. *x* ∈ *S* ⟹ *P*(*frag-of x*)
and *diff*: ⋀*a b*. ⟦*P a*; *P b*⟧ ⟹ *P*(*a* − *b*)
shows *P c*
proof −

```
  have P (∑ i∈I. frag-cmul (poly-mapping.lookup c i) (frag-of i))
    if I ⊆ Poly-Mapping.keys c for I
    using finite-subset [OF that finite-keys [of c]] that supp
  proof (induction I arbitrary: c rule: finite-induct)
    case empty
    then show ?case
      by (auto simp: 0)
  next
    case (insert i I c)
    have ab: a+b = a - (0 - b) for a b :: 'a ⇒₀ int
      by simp
    have Pfrag: P (frag-cmul (poly-mapping.lookup c i) (frag-of i))
      by (metis 0 diff frag-closure-minus-cmul insert.prems insert-subset sing sub-
set-iff)
    with insert show ?case
      by (metis (mono-tags, lifting) 0 ab diff insert-subset sum.insert)
  qed
  then show ?thesis
    by (subst frag-expansion) (auto simp: frag-extend-def)
qed

lemma frag-extend-compose:
  frag-extend f (frag-extend (frag-of o g) c) = frag-extend (f o g) c
  using subset-UNIV
  by (induction c rule: frag-induction) (auto simp: frag-extend-diff)

lemma frag-split:
  fixes c :: 'a ⇒₀ int
  assumes Poly-Mapping.keys c ⊆ S ∪ T
  obtains d e where Poly-Mapping.keys d ⊆ S Poly-Mapping.keys e ⊆ T d + e
= c
proof
  let ?d = frag-extend (λf. if f ∈ S then frag-of f else 0) c
  let ?e = frag-extend (λf. if f ∈ S then 0 else frag-of f) c
  show Poly-Mapping.keys ?d ⊆ S Poly-Mapping.keys ?e ⊆ T
   using assms by (auto intro!: order-trans [OF keys-frag-extend] split: if-split-asm)
  show ?d + ?e = c
    using assms
  proof (induction c rule: frag-induction)
    case (diff a b)
    then show ?case
     by (metis (no-types, lifting) frag-extend-diff add-diff-eq diff-add-eq diff-add-eq-diff-diff-swap)
  qed auto
qed

hide-const (open) lookup single update keys range map map-key degree nth the-value
items foldr mapp

end
```

80 Exponentiation by Squaring

theory *Power-By-Squaring*
imports *Main*
begin

context
fixes $f :: 'a \Rightarrow 'a \Rightarrow 'a$
begin

function *efficient-funpow* $:: 'a \Rightarrow 'a \Rightarrow nat \Rightarrow 'a$ **where**
efficient-funpow y x 0 $= y$
| *efficient-funpow y x* (*Suc 0*) $= f\ x\ y$
| $n \neq 0 \Longrightarrow$ *even n* $\Longrightarrow$ *efficient-funpow y x n* $=$ *efficient-funpow y* $(f\ x\ x)$ (*n div 2*)
| $n \neq 1 \Longrightarrow$ *odd n* $\Longrightarrow$ *efficient-funpow y x n* $=$ *efficient-funpow* $(f\ x\ y)$ $(f\ x\ x)$ (*n div 2*)
by *force+*
termination by (*relation measure* (*snd* $\circ$ *snd*)) (*auto elim*: *oddE*)

lemma *efficient-funpow-code* [*code*]:
efficient-funpow y x n $=$
(*if n* $= 0$ *then y*
else if n $= 1$ *then f x y*
else if even n then efficient-funpow y $(f\ x\ x)$ (*n div 2*)
else efficient-funpow $(f\ x\ y)$ $(f\ x\ x)$ (*n div 2*))
by (*induction y x n rule*: *efficient-funpow.induct*) *auto*

end

lemma *efficient-funpow-correct*:
assumes *f-assoc*: $\bigwedge x\ z.\ f\ x\ (f\ x\ z) = f\ (f\ x\ x)\ z$
shows *efficient-funpow f y x n* $= (f\ x \mathbin{\widehat{\ }\widehat{\ }} n)\ y$
proof $-$
have [*simp*]: $f \mathbin{\widehat{\ }\widehat{\ }} 2 = (\lambda x.\ f\ (f\ x))$ **for** $f :: 'a \Rightarrow 'a$
by (*simp add*: *eval-nat-numeral o-def*)
show *?thesis*
by (*induction y x n rule*: *efficient-funpow.induct*[*of - f*])
(*auto elim*!: *evenE oddE simp*: *funpow-mult* [*symmetric*] *funpow-Suc-right f-assoc*
simp del: *funpow.simps*(*2*))
qed

context *monoid-mult*
begin

lemma *power-by-squaring*: *efficient-funpow* $(*)$ $(1 :: 'a) = (\widehat{\ })$
proof (*intro ext*)

fix $x :: 'a$ **and** n
have *efficient-funpow* $(*)$ *1* x $n = ((*)$ x $\frown\frown$ $n)$ *1*
by (*subst efficient-funpow-correct*) (*simp-all add*: *mult.assoc*)
also have $\ldots = x \frown n$
by (*induction n*) *simp-all*
finally show *efficient-funpow* $(*)$ *1* x $n = x \frown n$ **.**
qed

end

end

81 Preorders with explicit equivalence relation

theory *Preorder*
imports *Main*
begin

class *preorder-equiv* = *preorder*
begin

definition *equiv* :: $'a \Rightarrow 'a \Rightarrow$ *bool*
where *equiv* x $y \longleftrightarrow x \leq y \wedge y \leq x$

notation
equiv (‹$'(\approx')$›) **and**
equiv (‹(‹*notation*=‹*infix* $\approx$››-/ $\approx$ -)› [*51*, *51*] *50*)

lemma *equivD1*: $x \leq y$ **if** $x \approx y$
using *that* **by** (*simp add*: *equiv-def*)

lemma *equivD2*: $y \leq x$ **if** $x \approx y$
using *that* **by** (*simp add*: *equiv-def*)

lemma *equiv-refl* [*iff*]: $x \approx x$
by (*simp add*: *equiv-def*)

lemma *equiv-sym*: $x \approx y \longleftrightarrow y \approx x$
by (*auto simp add*: *equiv-def*)

lemma *equiv-trans*: $x \approx y \Longrightarrow y \approx z \Longrightarrow x \approx z$
by (*auto simp*: *equiv-def intro*: *order-trans*)

lemma *equiv-antisym*: $x \leq y \Longrightarrow y \leq x \Longrightarrow x \approx y$
by (*simp only*: *equiv-def*)

lemma *less-le*: $x < y \longleftrightarrow x \leq y \wedge \neg\, x \approx y$
by (*auto simp add*: *equiv-def less-le-not-le*)

lemma *le-less*: $x \leq y \longleftrightarrow x < y \vee x \approx y$
by (*auto simp add*: *equiv-def less-le*)

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

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

lemma *not-equiv-le-trans*: $\neg\ a \approx b \Longrightarrow a \leq b \Longrightarrow a < b$
by (*simp add*: *less-le*)

lemma *le-not-equiv-trans*: $a \leq b \Longrightarrow \neg\ a \approx b \Longrightarrow a < b$
by (*rule not-equiv-le-trans*)

lemma *antisym-conv*: $y \leq x \Longrightarrow x \leq y \longleftrightarrow x \approx y$
by (*simp add*: *equiv-def*)

end

ML-file ‹~~/*src/Provers/preorder.ML*›

ML ‹
structure Quasi = Quasi-Tac(
struct

val le-trans = @{thm order-trans};
val le-refl = @{thm order-refl};
val eqD1 = @{thm equivD1};
val eqD2 = @{thm equivD2};
val less-reflE = @{thm less-irrefl};
val less-imp-le = @{thm less-imp-le};
val le-neq-trans = @{thm le-not-equiv-trans};
val neq-le-trans = @{thm not-equiv-le-trans};
val less-imp-neq = @{thm less-imp-not-equiv};

fun decomp-quasi thy (Const (@{const-name less-eq}, -) $ t1 $ t2) = SOME (t1, <=, t2)
| decomp-quasi thy (Const (@{const-name less}, -) $ t1 $ t2) = SOME (t1, <, t2)
| decomp-quasi thy (Const (@{const-name equiv}, -) $ t1 $ t2) = SOME (t1, =, t2)
| decomp-quasi thy (Const (@{const-name Not}, -) $ (Const (@{const-name equiv}, -) $ t1 $ t2)) = SOME (t1, ~=, t2)
| decomp-quasi thy - = NONE;

fun decomp-trans thy t = case decomp-quasi thy t of
x as SOME (t1, <=, t2) => x
| - => NONE;

end
);
›

end

82 Additive group operations on product types

theory *Product-Plus*
imports *Main*
begin

82.1 Operations

instantiation *prod* :: (*zero*, *zero*) *zero*
begin

definition *zero-prod-def*: $0 = (0, 0)$

instance ..
end

instantiation *prod* :: (*plus*, *plus*) *plus*
begin

definition *plus-prod-def*:
$x + y = (fst\ x + fst\ y, snd\ x + snd\ y)$

instance ..
end

instantiation *prod* :: (*minus*, *minus*) *minus*
begin

definition *minus-prod-def*:
$x - y = (fst\ x - fst\ y, snd\ x - snd\ y)$

instance ..
end

instantiation *prod* :: (*uminus*, *uminus*) *uminus*
begin

definition *uminus-prod-def*:
$- x = (- fst\ x, - snd\ x)$

instance ..
end

lemma *fst-zero* [*simp*]: *fst 0 = 0*
unfolding *zero-prod-def* **by** *simp*

lemma *snd-zero* [*simp*]: *snd 0 = 0*
unfolding *zero-prod-def* **by** *simp*

lemma *fst-add* [*simp*]: *fst (x + y) = fst x + fst y*
unfolding *plus-prod-def* **by** *simp*

lemma *snd-add* [*simp*]: *snd (x + y) = snd x + snd y*
unfolding *plus-prod-def* **by** *simp*

lemma *fst-diff* [*simp*]: *fst (x − y) = fst x − fst y*
unfolding *minus-prod-def* **by** *simp*

lemma *snd-diff* [*simp*]: *snd (x − y) = snd x − snd y*
unfolding *minus-prod-def* **by** *simp*

lemma *fst-uminus* [*simp*]: *fst (− x) = − fst x*
unfolding *uminus-prod-def* **by** *simp*

lemma *snd-uminus* [*simp*]: *snd (− x) = − snd x*
unfolding *uminus-prod-def* **by** *simp*

lemma *add-Pair* [*simp*]: *(a, b) + (c, d) = (a + c, b + d)*
unfolding *plus-prod-def* **by** *simp*

lemma *diff-Pair* [*simp*]: *(a, b) − (c, d) = (a − c, b − d)*
unfolding *minus-prod-def* **by** *simp*

lemma *uminus-Pair* [*simp*, *code*]: *− (a, b) = (− a, − b)*
unfolding *uminus-prod-def* **by** *simp*

82.2 Class instances

instance *prod* :: (*semigroup-add*, *semigroup-add*) *semigroup-add*
by *standard* (*simp add*: *prod-eq-iff add.assoc*)

instance *prod* :: (*ab-semigroup-add*, *ab-semigroup-add*) *ab-semigroup-add*
by *standard* (*simp add*: *prod-eq-iff add.commute*)

instance *prod* :: (*monoid-add*, *monoid-add*) *monoid-add*
by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*comm-monoid-add*, *comm-monoid-add*) *comm-monoid-add*
by *standard* (*simp add*: *prod-eq-iff*)

instance *prod* :: (*cancel-semigroup-add*, *cancel-semigroup-add*) *cancel-semigroup-add*

by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*cancel-ab-semigroup-add*, *cancel-ab-semigroup-add*) *cancel-ab-semigroup-add*
by *standard* (*simp-all add*: *prod-eq-iff diff-diff-eq*)

instance *prod* :: (*cancel-comm-monoid-add*, *cancel-comm-monoid-add*) *cancel-comm-monoid-add*
..

instance *prod* :: (*group-add*, *group-add*) *group-add*
by *standard* (*simp-all add*: *prod-eq-iff*)

instance *prod* :: (*ab-group-add*, *ab-group-add*) *ab-group-add*
by *standard* (*simp-all add*: *prod-eq-iff*)

lemma *fst-sum*: *fst* ($\sum x{\in}A.\ f\ x$) = ($\sum x{\in}A.$ *fst* (*f x*))
proof (*cases finite A*)
case *True*
then show *?thesis* **by** *induct simp-all*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *snd-sum*: *snd* ($\sum x{\in}A.\ f\ x$) = ($\sum x{\in}A.$ *snd* (*f x*))
proof (*cases finite A*)
case *True*
then show *?thesis* **by** *induct simp-all*
next
case *False*
then show *?thesis* **by** *simp*
qed

lemma *sum-prod*: ($\sum x{\in}A.$ (*f x*, *g x*)) = ($\sum x{\in}A.\ f\ x$, $\sum x{\in}A.\ g\ x$)
proof (*cases finite A*)
case *True*
then show *?thesis* **by** *induct* (*simp-all add*: *zero-prod-def*)
next
case *False*
then show *?thesis* **by** (*simp add*: *zero-prod-def*)
qed

end

83 Roots of real quadratics

theory *Quadratic-Discriminant*
imports *Complex-Main*
begin

definition *discrim* :: *real* ⇒ *real* ⇒ *real* ⇒ *real*
  where *discrim a b c* ≡ $b^2 - 4 * a * c$

lemma *complete-square*:
  $a \neq 0 \Longrightarrow a * x^2 + b * x + c = 0 \longleftrightarrow (2 * a * x + b)^2 = discrim\ a\ b\ c$
by (*simp add*: *discrim-def*) *algebra*

lemma *discriminant-negative*:
  fixes *a b c x* :: *real*
  assumes $a \neq 0$
    and *discrim a b c* < 0
  shows $a * x^2 + b * x + c \neq 0$
  by (*metis assms complete-square power2-less-0*)

lemma *plus-or-minus-sqrt*:
  fixes *x y* :: *real*
  assumes $y \geq 0$
  shows $x^2 = y \longleftrightarrow x = sqrt\ y \vee x = -\ sqrt\ y$
  using *assms* **by** *fastforce*

lemma *divide-non-zero*:
  fixes *x y z* :: *real*
  assumes $x \neq 0$
  shows $x * y = z \longleftrightarrow y = z\ /\ x$
  by (*simp add*: *assms mult.commute nonzero-eq-divide-eq*)

lemma *discriminant-nonneg*:
  fixes *a b c x* :: *real*
  assumes $a \neq 0$
    and *discrim a b c* ≥ 0
  shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
    $x = (-b + sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a) \vee$
    $x = (-b - sqrt\ (discrim\ a\ b\ c))\ /\ (2 * a)$ (**is** *?L* = *?R*)
proof
  assume *?L* **with** *assms* **show** *?R*
    by (*smt* (*verit*, *ccfv-threshold*) *complete-square nonzero-mult-div-cancel-left real-sqrt-abs*)
qed (*use assms complete-square* **in** *auto*)

lemma *discriminant-zero*:
  fixes *a b c x* :: *real*
  assumes $a \neq 0$
    and *discrim a b c* = 0
  shows $a * x^2 + b * x + c = 0 \longleftrightarrow x = -b\ /\ (2 * a)$
  by (*simp add*: *discriminant-nonneg assms*)

theorem *discriminant-iff*:
  fixes *a b c x* :: *real*
  assumes $a \neq 0$

shows $a * x^2 + b * x + c = 0 \longleftrightarrow$
$discrim\ a\ b\ c \geq 0 \wedge$
$(x = (-b + sqrt\ (discrim\ a\ b\ c)) / (2 * a) \vee$
$x = (-b - sqrt\ (discrim\ a\ b\ c)) / (2 * a))$
by (*smt* (*verit*, *best*) *assms discriminant-negative discriminant-nonneg*)

lemma *discriminant-nonneg-ex*:
fixes *a b c* :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c \geq 0$
shows $\exists\ x.\ a * x^2 + b * x + c = 0$
by (*auto simp*: *discriminant-nonneg assms*)

lemma *discriminant-pos-ex*:
fixes *a b c* :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists x\ y.\ x \neq y \wedge a * x^2 + b * x + c = 0 \wedge a * y^2 + b * y + c = 0$
proof −
let $?x = (-b + sqrt\ (discrim\ a\ b\ c)) / (2 * a)$
let $?y = (-b - sqrt\ (discrim\ a\ b\ c)) / (2 * a)$
from ‹$discrim\ a\ b\ c > 0$› **have** $sqrt\ (discrim\ a\ b\ c) \neq 0$
by *simp*
then have $sqrt\ (discrim\ a\ b\ c) \neq - sqrt\ (discrim\ a\ b\ c)$
by *arith*
with ‹$a \neq 0$› **have** $?x \neq ?y$
by *simp*
moreover from *assms* **have** $a * ?x^2 + b * ?x + c = 0$ **and** $a * ?y^2 + b * ?y + c = 0$
using *discriminant-nonneg* [*of a b c ?x*]
and *discriminant-nonneg* [*of a b c ?y*]
by *simp-all*
ultimately show *?thesis*
by *blast*
qed

lemma *discriminant-pos-distinct*:
fixes *a b c x* :: *real*
assumes $a \neq 0$
and $discrim\ a\ b\ c > 0$
shows $\exists\ y.\ x \neq y \wedge a * y^2 + b * y + c = 0$
by (*metis assms discriminant-pos-ex*)

lemma *Rats-solution-QE*:
assumes $a \in \mathbb{Q}$ $b \in \mathbb{Q}$ $a \neq 0$
and $a*x\widehat{}2 + b*x + c = 0$
and $sqrt\ (discrim\ a\ b\ c) \in \mathbb{Q}$
shows $x \in \mathbb{Q}$
using *assms*(*1*,*2*,*5*) *discriminant-iff*[*THEN iffD1*, *OF assms*(*3*,*4*)] **by** *auto*

lemma *Rats-solution-QE-converse*:
assumes $a \in \mathbb{Q}$ $b \in \mathbb{Q}$
and $a*x\hat{}2 + b*x + c = 0$
and $x \in \mathbb{Q}$
shows *sqrt* (*discrim a b c*) $\in \mathbb{Q}$
proof −
from *assms*(*3*) **have** *discrim a b c* = $(2*a*x+b)\hat{}2$ **unfolding** *discrim-def* **by** *algebra*
hence *sqrt* (*discrim a b c*) = $|2*a*x+b|$ **by** (*simp*)
thus *?thesis* **using** ‹$a \in \mathbb{Q}$› ‹$b \in \mathbb{Q}$› ‹$x \in \mathbb{Q}$› **by** (*simp*)
qed

end

84 Pretty syntax for Quotient operations

theory *Quotient-Syntax*
imports *Main*
begin

notation
rel-conj (**infixr** ‹*OOO*› *75*) **and**
map-fun (**infixr** ‹−−−>› *55*) **and**
rel-fun (**infixr** ‹===>› *55*)

end

85 Quotient infrastructure for the set type

theory *Quotient-Set*
imports *Quotient-Syntax*
begin

85.1 Contravariant set map (vimage) and set relator, rules for the Quotient package

definition *rel-vset R xs ys* ≡ ∀ *x y*. *R x y* ⟶ *x* ∈ *xs* ⟷ *y* ∈ *ys*

lemma *rel-vset-eq* [*id-simps*]:
rel-vset (=) = (=)
by (*subst fun-eq-iff*, *subst fun-eq-iff*) (*simp add*: *set-eq-iff rel-vset-def*)

lemma *rel-vset-equivp*:
assumes *e*: *equivp R*
shows *rel-vset R xs ys* ⟷ *xs* = *ys* ∧ (∀ *x y*. *x* ∈ *xs* ⟶ *R x y* ⟶ *y* ∈ *xs*)
unfolding *rel-vset-def*
using *equivp-reflp*[*OF e*]

by *auto* (*metis*, *metis equivp-symp*[*OF e*])

lemma *set-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-vset R*) (*vimage Rep*) (*vimage Abs*)
proof (*rule Quotient3I*)
from *assms* **have** ⋀*x*. *Abs* (*Rep x*) = *x* **by** (*rule Quotient3-abs-rep*)
then show ⋀*xs*. *Rep* −‘ (*Abs* −‘ *xs*) = *xs*
unfolding *vimage-def* **by** *auto*
next
show ⋀*xs*. *rel-vset R* (*Abs* −‘ *xs*) (*Abs* −‘ *xs*)
unfolding *rel-vset-def vimage-def*
by *auto* (*metis Quotient3-rel-abs*[*OF assms*])+
next
fix *r s*
show *rel-vset R r s* = (*rel-vset R r r* ∧ *rel-vset R s s* ∧ *Rep* −‘ *r* = *Rep* −‘ *s*)
unfolding *rel-vset-def vimage-def set-eq-iff*
by *auto* (*metis rep-abs-rsp*[*OF assms*] *assms*[*simplified Quotient3-def*])+
qed

declare [[*mapQ3 set* = (*rel-vset*, *set-quotient*)]]

lemma *empty-set-rsp*[*quot-respect*]:
rel-vset R {} {}
unfolding *rel-vset-def* **by** *simp*

lemma *collect-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows ((*R* ===> (=)) ===> *rel-vset R*) *Collect Collect*
by (*intro rel-funI*) (*simp add*: *rel-fun-def rel-vset-def*)

lemma *collect-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((*Abs* −−−> *id*) −−−> (−‘) *Rep*) *Collect* = *Collect*
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF assms*])

lemma *union-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R* ===> *rel-vset R* ===> *rel-vset R*) (∪) (∪)
by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

lemma *union-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((−‘) *Abs* −−−> (−‘) *Abs* −−−> (−‘) *Rep*) (∪) = (∪)
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]])

lemma *diff-rsp*[*quot-respect*]:

assumes *Quotient3 R Abs Rep*
shows (*rel-vset R ===> rel-vset R ===> rel-vset R*) (−) (−)
by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

lemma *diff-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((− ‘) *Abs* −−−> (− ‘) *Abs* −−−> (− ‘) *Rep*) (−) = (−)
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]] *vimage-Diff*)

lemma *inter-rsp*[*quot-respect*]:
assumes *Quotient3 R Abs Rep*
shows (*rel-vset R ===> rel-vset R ===> rel-vset R*) (∩) (∩)
by (*intro rel-funI*) (*auto simp add*: *rel-vset-def*)

lemma *inter-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows ((− ‘) *Abs* −−−> (− ‘) *Abs* −−−> (− ‘) *Rep*) (∩) = (∩)
unfolding *fun-eq-iff*
by (*simp add*: *Quotient3-abs-rep*[*OF set-quotient*[*OF assms*]])

lemma *mem-prs*[*quot-preserve*]:
assumes *Quotient3 R Abs Rep*
shows (*Rep* −−−> (− ‘) *Abs* −−−> *id*) (∈) = (∈)
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF assms*])

lemma *mem-rsp*[*quot-respect*]:
shows (*R ===> rel-vset R ===>* (=)) (∈) (∈)
by (*intro rel-funI*) (*simp add*: *rel-vset-def*)

end

86 Quotient infrastructure for the product type

theory *Quotient-Product*
imports *Quotient-Syntax*
begin

86.1 Rules for the Quotient package

lemma *map-prod-id* [*id-simps*]:
shows *map-prod id id = id*
by (*simp add*: *fun-eq-iff*)

lemma *rel-prod-eq* [*id-simps*]:
shows *rel-prod* (=) (=) = (=)
by (*simp add*: *fun-eq-iff*)

lemma *prod-equivp* [*quot-equiv*]:

assumes *equivp R1*
assumes *equivp R2*
shows *equivp* (*rel-prod R1 R2*)
using *assms* **by** (*auto intro*!: *equivpI reflpI sympI transpI elim*!: *equivpE elim*: *reflpE sympE transpE*)

lemma *prod-quotient* [*quot-thm*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows *Quotient3* (*rel-prod R1 R2*) (*map-prod Abs1 Abs2*) (*map-prod Rep1 Rep2*)
apply (*rule Quotient3I*)
apply (*simp add*: *map-prod.compositionality comp-def map-prod.identity*
Quotient3-abs-rep [*OF assms*(*1*)] *Quotient3-abs-rep* [*OF assms*(*2*)])
apply (*simp add*: *split-paired-all Quotient3-rel-rep* [*OF assms*(*1*)] *Quotient3-rel-rep* [*OF assms*(*2*)])
using *Quotient3-rel* [*OF assms*(*1*)] *Quotient3-rel* [*OF assms*(*2*)]
apply (*auto simp add*: *split-paired-all*)
done

declare [[*mapQ3 prod* = (*rel-prod*, *prod-quotient*)]]

lemma *Pair-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R1* ===> *R2* ===> *rel-prod R1 R2*) *Pair Pair*
by (*rule Pair-transfer*)

lemma *Pair-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* ---> *Rep2* ---> (*map-prod Abs1 Abs2*)) *Pair* = *Pair*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])
done

lemma *fst-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*
assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2* ===> *R1*) *fst fst*
by *auto*

lemma *fst-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*map-prod Rep1 Rep2* ---> *Abs1*) *fst* = *fst*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*])

lemma *snd-rsp* [*quot-respect*]:
assumes *Quotient3 R1 Abs1 Rep1*

assumes *Quotient3 R2 Abs2 Rep2*
shows (*rel-prod R1 R2 ===> R2*) *snd snd*
by *auto*

lemma *snd-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*map-prod Rep1 Rep2 ---> Abs2*) *snd = snd*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q2*])

lemma *case-prod-rsp* [*quot-respect*]:
shows ((*R1 ===> R2 ===> (=)*) *===> (rel-prod R1 R2) ===> (=)*) *case-prod case-prod*
by (*rule case-prod-transfer*)

lemma *split-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows (((*Abs1 ---> Abs2 ---> id*) *---> map-prod Rep1 Rep2 ---> id*) *case-prod*) = *case-prod*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])

lemma [*quot-respect*]:
shows ((*R2 ===> R2 ===> (=)*) *===> (R1 ===> R1 ===> (=)) ===>*
rel-prod R2 R1 ===> rel-prod R2 R1 ===> (=)) *rel-prod rel-prod*
by (*rule prod.rel-transfer*)

lemma [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 abs1 rep1*
and *q2*: *Quotient3 R2 abs2 rep2*
shows ((*abs1 ---> abs1 ---> id*) *---> (abs2 ---> abs2 ---> id*) *--->*
map-prod rep1 rep2 ---> map-prod rep1 rep2 ---> id) *rel-prod = rel-prod*
by (*simp add*: *fun-eq-iff Quotient3-abs-rep*[*OF q1*] *Quotient3-abs-rep*[*OF q2*])

lemma [*quot-preserve*]:
shows(*rel-prod* ((*rep1 ---> rep1 ---> id*) *R1*) ((*rep2 ---> rep2 ---> id*) *R2*)
(*l1*, *l2*) (*r1*, *r2*)) = (*R1* (*rep1 l1*) (*rep1 r1*) ∧ *R2* (*rep2 l2*) (*rep2 r2*))
by *simp*

declare *prod.inject*[*quot-preserve*]

end

87 Quotient infrastructure for the option type

theory *Quotient-Option*
imports *Quotient-Syntax*

begin

87.1 Rules for the Quotient package

lemma *rel-option-map1*:
rel-option R (*map-option f x*) *y* ⟷ *rel-option* (λ*x*. *R* (*f x*)) *x y*
by (*simp add*: *rel-option-iff split*: *option.split*)

lemma *rel-option-map2*:
rel-option R x (*map-option f y*) ⟷ *rel-option* (λ*x y*. *R x* (*f y*)) *x y*
by (*simp add*: *rel-option-iff split*: *option.split*)

declare
map-option.id [*id-simps*]
option.rel-eq [*id-simps*]

lemma *reflp-rel-option*:
reflp R ⟹ *reflp* (*rel-option R*)
unfolding *reflp-def split-option-all* **by** *simp*

lemma *option-symp*:
symp R ⟹ *symp* (*rel-option R*)
unfolding *symp-def split-option-all*
by (*simp only*: *option.rel-inject option.rel-distinct*) *fast*

lemma *option-transp*:
transp R ⟹ *transp* (*rel-option R*)
unfolding *transp-def split-option-all*
by (*simp only*: *option.rel-inject option.rel-distinct*) *fast*

lemma *option-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*rel-option R*)
by (*blast intro*: *equivpI reflp-rel-option option-symp option-transp elim*: *equivpE*)

lemma *option-quotient* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*rel-option R*) (*map-option Abs*) (*map-option Rep*)
apply (*rule Quotient3I*)
apply (*simp-all add*: *option.map-comp comp-def option.map-id*[*unfolded id-def*] *option.rel-eq rel-option-map1 rel-option-map2 Quotient3-abs-rep* [*OF assms*] *Quotient3-rel-rep* [*OF assms*])
using *Quotient3-rel* [*OF assms*]
apply (*simp add*: *rel-option-iff split*: *option.split*)
done

declare [[*mapQ3 option* = (*rel-option*, *option-quotient*)]]

lemma *option-None-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*

shows *rel-option R None None*
by (*rule option.ctr-transfer*(*1*))

lemma *option-Some-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *rel-option R*) *Some Some*
by (*rule option.ctr-transfer*(*2*))

lemma *option-None-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map-option Abs None* = *None*
by (*rule Option.option.map*(*1*))

lemma *option-Some-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> *map-option Abs*) *Some* = *Some*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q*])
done

end

88 Quotient infrastructure for the list type

theory *Quotient-List*
imports *Quotient-Set Quotient-Product Quotient-Option*
begin

88.1 Rules for the Quotient package

lemma *map-id* [*id-simps*]:
map id = *id*
by (*fact List.map.id*)

lemma *list-all2-eq* [*id-simps*]:
list-all2 (=) = (=)
proof (*rule ext*)+
fix *xs ys*
show *list-all2* (=) *xs ys* ⟷ *xs* = *ys*
by (*induct xs ys rule*: *list-induct2'*) *simp-all*
qed

lemma *reflp-list-all2*:
assumes *reflp R*
shows *reflp* (*list-all2 R*)
proof (*rule reflpI*)
from *assms* **have** ∗: ⋀*xs*. *R xs xs* **by** (*rule reflpE*)
fix *xs*
show *list-all2 R xs xs*

by (*induct xs*) (*simp-all add*: ∗)
qed

lemma *list-symp*:
assumes *symp R*
shows *symp* (*list-all2 R*)
proof (*rule sympI*)
from *assms* **have** ∗: ⋀*xs ys*. *R xs ys* ⟹ *R ys xs* **by** (*rule sympE*)
fix *xs ys*
assume *list-all2 R xs ys*
then show *list-all2 R ys xs*
by (*induct xs ys rule*: *list-induct2′*) (*simp-all add*: ∗)
qed

lemma *list-transp*:
assumes *transp R*
shows *transp* (*list-all2 R*)
proof (*rule transpI*)
from *assms* **have** ∗: ⋀*xs ys zs*. *R xs ys* ⟹ *R ys zs* ⟹ *R xs zs* **by** (*rule transpE*)
fix *xs ys zs*
assume *list-all2 R xs ys* **and** *list-all2 R ys zs*
then show *list-all2 R xs zs*
by (*induct arbitrary*: *zs*) (*auto simp*: *list-all2-Cons1 intro*: ∗)
qed

lemma *list-equivp* [*quot-equiv*]:
equivp R ⟹ *equivp* (*list-all2 R*)
by (*blast intro*: *equivpI reflp-list-all2 list-symp list-transp elim*: *equivpE*)

lemma *list-quotient3* [*quot-thm*]:
assumes *Quotient3 R Abs Rep*
shows *Quotient3* (*list-all2 R*) (*map Abs*) (*map Rep*)
proof (*rule Quotient3I*)
from *assms* **have** ⋀*x*. *Abs* (*Rep x*) = *x* **by** (*rule Quotient3-abs-rep*)
then show ⋀*xs*. *map Abs* (*map Rep xs*) = *xs* **by** (*simp add*: *comp-def*)
next
from *assms* **have** ⋀*x y*. *R* (*Rep x*) (*Rep y*) ⟷ *x* = *y* **by** (*rule Quotient3-rel-rep*)
then show ⋀*xs*. *list-all2 R* (*map Rep xs*) (*map Rep xs*)
by (*simp add*: *list-all2-map1 list-all2-map2 list-all2-eq*)
next
fix *xs ys*
from *assms* **have** ⋀*x y*. *R x x* ∧ *R y y* ∧ *Abs x* = *Abs y* ⟷ *R x y* **by** (*rule Quotient3-rel*)
then show *list-all2 R xs ys* ⟷ *list-all2 R xs xs* ∧ *list-all2 R ys ys* ∧ *map Abs xs* = *map Abs ys*
by (*induct xs ys rule*: *list-induct2′*) *auto*
qed

declare [[*mapQ3 list* = (*list-all2*, *list-quotient3*)]]

lemma *cons-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*Rep* −−−> (*map Rep*) −−−> (*map Abs*)) (#) = (#)
by (*auto simp add*: *fun-eq-iff comp-def Quotient3-abs-rep* [*OF q*])

lemma *cons-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows (*R* ===> *list-all2 R* ===> *list-all2 R*) (#) (#)
by *auto*

lemma *nil-prs* [*quot-preserve*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *map Abs* [] = []
by *simp*

lemma *nil-rsp* [*quot-respect*]:
assumes *q*: *Quotient3 R Abs Rep*
shows *list-all2 R* [] []
by *simp*

lemma *map-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows (*map abs2*) (*map* ((*abs1* −−−> *rep2*) *f*) (*map rep1 l*)) = *map f l*
by (*induct l*)
(*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *map-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *rep2*) −−−> (*map rep1*) −−−> (*map abs2*)) *map* = *map*
and ((*abs1* −−−> *id*) −−−> *map rep1* −−−> *id*) *map* = *map*
by (*simp-all only*: *fun-eq-iff map-prs-aux*[*OF a b*] *comp-def*)
(*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *map-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2*) ===> (*list-all2 R1*) ===> *list-all2 R2*) *map map*
and ((*R1* ===> (=)) ===> (*list-all2 R1*) ===> (=)) *map map*
unfolding *list-all2-eq* [*symmetric*] **by** (*rule list.map-transfer*)+

lemma *foldr-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs2* (*foldr* ((*abs1* −−−> *abs2* −−−> *rep2*) *f*) (*map rep1 l*) (*rep2 e*))
= *foldr f l e*
by (*induct l*) (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *foldr-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep2*) −−−> (*map rep1*) −−−> *rep2* −−−> *abs2*) *foldr* = *foldr*
apply (*simp add*: *fun-eq-iff*)
by (*simp only*: *fun-eq-iff foldr-prs-aux*[*OF a b*])
(*simp*)

lemma *foldl-prs-aux*:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows *abs1* (*foldl* ((*abs1* −−−> *abs2* −−−> *rep1*) *f*) (*rep1 e*) (*map rep2 l*)) = *foldl f e l*
by (*induct l arbitrary*:*e*) (*simp-all add*: *Quotient3-abs-rep*[*OF a*] *Quotient3-abs-rep*[*OF b*])

lemma *foldl-prs* [*quot-preserve*]:
assumes *a*: *Quotient3 R1 abs1 rep1*
and *b*: *Quotient3 R2 abs2 rep2*
shows ((*abs1* −−−> *abs2* −−−> *rep1*) −−−> *rep1* −−−> (*map rep2*) −−−> *abs1*) *foldl* = *foldl*
by (*simp add*: *fun-eq-iff foldl-prs-aux* [*OF a b*])

lemma *foldl-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R1*) ===> *R1* ===> *list-all2 R2* ===> *R1*) *foldl foldl*
by (*rule foldl-transfer*)

lemma *foldr-rsp*[*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
and *q2*: *Quotient3 R2 Abs2 Rep2*
shows ((*R1* ===> *R2* ===> *R2*) ===> *list-all2 R1* ===> *R2* ===> *R2*) *foldr foldr*
by (*rule foldr-transfer*)

lemma *list-all2-rsp*:
assumes *r*: ∀ *x y*. *R x y* ⟶ (∀ *a b*. *R a b* ⟶ *S x a* = *T y b*)
and *l1*: *list-all2 R x y*
and *l2*: *list-all2 R a b*
shows *list-all2 S x a* = *list-all2 T y b*
using *l1 l2*
by (*induct arbitrary*: *a b rule*: *list-all2-induct*,
auto simp: *list-all2-Cons1 list-all2-Cons2 r*)

lemma [*quot-respect*]:

((*R* ===> *R* ===> (=)) ===> *list-all2 R* ===> *list-all2 R* ===> (=)) *list-all2 list-all2*
by (*rule list.rel-transfer*)

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows ((*abs1* ---> *abs1* ---> *id*) ---> *map rep1* ---> *map rep1* ---> *id*) *list-all2* = *list-all2*
apply (*simp add*: *fun-eq-iff*)
apply *clarify*
apply (*induct-tac xa xb rule*: *list-induct2'*)
apply (*simp-all add*: *Quotient3-abs-rep*[*OF a*])
done

lemma [*quot-preserve*]:
assumes *a*: *Quotient3 R abs1 rep1*
shows (*list-all2* ((*rep1* ---> *rep1* ---> *id*) *R*) *l m*) = (*l* = *m*)
by (*induct l m rule*: *list-induct2'*) (*simp-all add*: *Quotient3-rel-rep*[*OF a*])

lemma *list-all2-find-element*:
assumes *a*: *x* ∈ *set a*
and *b*: *list-all2 R a b*
shows ∃ *y*. (*y* ∈ *set b* ∧ *R x y*)
using *b a* **by** *induct auto*

lemma *list-all2-refl*:
assumes *a*: ⋀*x y*. *R x y* = (*R x* = *R y*)
shows *list-all2 R x x*
by (*induct x*) (*auto simp add*: *a*)

end

89 Quotient infrastructure for the sum type

theory *Quotient-Sum*
imports *Quotient-Syntax*
begin

89.1 Rules for the Quotient package

lemma *rel-sum-map1*:
rel-sum R1 R2 (*map-sum f1 f2 x*) *y* ⟷ *rel-sum* (λ*x*. *R1* (*f1 x*)) (λ*x*. *R2* (*f2 x*)) *x y*
by (*rule sum.rel-map*(*1*))

lemma *rel-sum-map2*:
rel-sum R1 R2 x (*map-sum f1 f2 y*) ⟷ *rel-sum* (λ*x y*. *R1 x* (*f1 y*)) (λ*x y*. *R2 x* (*f2 y*)) *x y*
by (*rule sum.rel-map*(*2*))

lemma *map-sum-id* [*id-simps*]:
map-sum id id = id
by (*simp add*: *id-def map-sum.identity fun-eq-iff*)

lemma *rel-sum-eq* [*id-simps*]:
rel-sum (=) (=) = (=)
by (*rule sum.rel-eq*)

lemma *reflp-rel-sum*:
reflp R1 ⟹ reflp R2 ⟹ reflp (rel-sum R1 R2)
unfolding *reflp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-symp*:
symp R1 ⟹ symp R2 ⟹ symp (rel-sum R1 R2)
unfolding *symp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-transp*:
transp R1 ⟹ transp R2 ⟹ transp (rel-sum R1 R2)
unfolding *transp-def split-sum-all rel-sum-simps* **by** *fast*

lemma *sum-equivp* [*quot-equiv*]:
equivp R1 ⟹ equivp R2 ⟹ equivp (rel-sum R1 R2)
by (*blast intro*: *equivpI reflp-rel-sum sum-symp sum-transp elim*: *equivpE*)

lemma *sum-quotient* [*quot-thm*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows *Quotient3* (*rel-sum R1 R2*) (*map-sum Abs1 Abs2*) (*map-sum Rep1 Rep2*)
apply (*rule Quotient3I*)
apply (*simp-all add*: *map-sum.compositionality comp-def map-sum.identity rel-sum-eq rel-sum-map1 rel-sum-map2*
Quotient3-abs-rep [*OF q1*] *Quotient3-rel-rep* [*OF q1*] *Quotient3-abs-rep* [*OF q2*] *Quotient3-rel-rep* [*OF q2*])
using *Quotient3-rel* [*OF q1*] *Quotient3-rel* [*OF q2*]
apply (*fastforce elim*!: *rel-sum.cases simp add*: *comp-def split*: *sum.split*)
done

declare [[*mapQ3 sum* = (*rel-sum*, *sum-quotient*)]]

lemma *sum-Inl-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*R1 ===> rel-sum R1 R2*) *Inl Inl*
by *auto*

lemma *sum-Inr-rsp* [*quot-respect*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*

shows (*R2* ===> *rel-sum R1 R2*) *Inr Inr*
by *auto*

lemma *sum-Inl-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep1* −−−> *map-sum Abs1 Abs2*) *Inl* = *Inl*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q1*])
done

lemma *sum-Inr-prs* [*quot-preserve*]:
assumes *q1*: *Quotient3 R1 Abs1 Rep1*
assumes *q2*: *Quotient3 R2 Abs2 Rep2*
shows (*Rep2* −−−> *map-sum Abs1 Abs2*) *Inr* = *Inr*
apply(*simp add*: *fun-eq-iff*)
apply(*simp add*: *Quotient3-abs-rep*[*OF q2*])
done

end

90 Quotient types

theory *Quotient-Type*
imports *Main*
begin

We introduce the notion of quotient types over equivalence relations via type classes.

90.1 Equivalence relations and quotient types

Type class *equiv* models equivalence relations $\sim :: 'a \Rightarrow 'a \Rightarrow bool$.

class *eqv* =
fixes *eqv* :: $'a \Rightarrow 'a \Rightarrow bool$ (**infixl** ‹∼› *50*)

class *equiv* = *eqv* +
assumes *equiv-refl* [*intro*]: $x \sim x$
and *equiv-trans* [*trans*]: $x \sim y \Longrightarrow y \sim z \Longrightarrow x \sim z$
and *equiv-sym* [*sym*]: $x \sim y \Longrightarrow y \sim x$
begin

lemma *equiv-not-sym* [*sym*]: $\neg\ x \sim y \Longrightarrow \neg\ y \sim x$
proof −
assume $\neg\ x \sim y$
then show $\neg\ y \sim x$ **by** (*rule contrapos-nn*) (*rule equiv-sym*)
qed

lemma *not-equiv-trans1* [*trans*]: $\neg\ x \sim y \Longrightarrow y \sim z \Longrightarrow \neg\ x \sim z$
proof −
  assume $\neg\ x \sim y$ **and** $y \sim z$
  show $\neg\ x \sim z$
  proof
    assume $x \sim z$
    also from ‹$y \sim z$› **have** $z \sim y$ **..**
    finally have $x \sim y$ **.**
    with ‹$\neg\ x \sim y$› **show** *False* **by** *contradiction*
  qed
qed

lemma *not-equiv-trans2* [*trans*]: $x \sim y \Longrightarrow \neg\ y \sim z \Longrightarrow \neg\ x \sim z$
proof −
  assume $\neg\ y \sim z$
  then have $\neg\ z \sim y$ **..**
  also
  assume $x \sim y$
  then have $y \sim x$ **..**
  finally have $\neg\ z \sim x$ **.**
  then show $\neg\ x \sim z$ **..**
qed

end

The quotient type *'a quot* consists of all *equivalence classes* over elements of the base type *'a*.

definition (**in** *eqv*) *quot* = $\{\{x.\ a \sim x\} \mid a.\ True\}$

typedef (**overloaded**) *'a quot* = *quot* :: *'a*::*eqv set set*
  unfolding *quot-def* **by** *blast*

lemma *quotI* [*intro*]: $\{x.\ a \sim x\} \in quot$
  unfolding *quot-def* **by** *blast*

lemma *quotE* [*elim*]:
  assumes $R \in quot$
  obtains *a* **where** $R = \{x.\ a \sim x\}$
  using *assms* **unfolding** *quot-def* **by** *blast*

Abstracted equivalence classes are the canonical representation of elements of a quotient type.

definition *class* :: *'a*::*equiv* ⇒ *'a quot* (‹(‹*open-block notation*=‹*mixfix class*››⌊-⌋)›)
  where $\lfloor a \rfloor$ = *Abs-quot* $\{x.\ a \sim x\}$

theorem *quot-exhaust*: $\exists\, a.\ A = \lfloor a \rfloor$
proof (*cases A*)
  fix *R*
  assume *R*: *A* = *Abs-quot R*

assume $R \in$ *quot*
then have $\exists\, a.\ R = \{x.\ a \sim x\}$ **by** *blast*
with R **have** $\exists\, a.\ A =$ *Abs-quot* $\{x.\ a \sim x\}$ **by** *blast*
then show *?thesis* **unfolding** *class-def* **.**
qed

lemma *quot-cases* [*cases type*: *quot*]:
obtains a **where** $A = \lfloor a \rfloor$
using *quot-exhaust* **by** *blast*

90.2 Equality on quotients

Equality of canonical quotient elements coincides with the original relation.

theorem *quot-equality* [*iff?*]: $\lfloor a \rfloor = \lfloor b \rfloor \longleftrightarrow a \sim b$
proof
assume *eq*: $\lfloor a \rfloor = \lfloor b \rfloor$
show $a \sim b$
proof −
from *eq* **have** $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
by (*simp only*: *class-def Abs-quot-inject quotI*)
moreover have $a \sim a$ **..**
ultimately have $a \in \{x.\ b \sim x\}$ **by** *blast*
then have $b \sim a$ **by** *blast*
then show *?thesis* **..**
qed
next
assume *ab*: $a \sim b$
show $\lfloor a \rfloor = \lfloor b \rfloor$
proof −
have $\{x.\ a \sim x\} = \{x.\ b \sim x\}$
proof (*rule Collect-cong*)
fix x **show** $(a \sim x) = (b \sim x)$
proof
from *ab* **have** $b \sim a$ **..**
also assume $a \sim x$
finally show $b \sim x$ **.**
next
note *ab*
also assume $b \sim x$
finally show $a \sim x$ **.**
qed
qed
then show *?thesis* **by** (*simp only*: *class-def*)
qed
qed

90.3 Picking representing elements

definition *pick* :: *'a*::*equiv quot* ⇒ *'a*

where *pick A* = (*SOME a*. *A* = ⌊*a*⌋)

theorem *pick-equiv* [*intro*]: *pick* ⌊*a*⌋ ∼ *a*
proof (*unfold pick-def*)
show (*SOME x*. ⌊*a*⌋ = ⌊*x*⌋) ∼ *a*
proof (*rule someI2*)
show ⌊*a*⌋ = ⌊*a*⌋ **..**
fix *x* **assume** ⌊*a*⌋ = ⌊*x*⌋
then have *a* ∼ *x* **..**
then show *x* ∼ *a* **..**
qed
qed

theorem *pick-inverse* [*intro*]: ⌊*pick A*⌋ = *A*
proof (*cases A*)
fix *a* **assume** *a*: *A* = ⌊*a*⌋
then have *pick A* ∼ *a* **by** (*simp only*: *pick-equiv*)
then have ⌊*pick A*⌋ = ⌊*a*⌋ **..**
with *a* **show** *?thesis* **by** *simp*
qed

The following rules support canonical function definitions on quotient types (with up to two arguments). Note that the stripped-down version without additional conditions is sufficient most of the time.

theorem *quot-cond-function*:
assumes *eq*: ⋀*X Y*. *P X Y* ⟹ *f X Y* ≡ *g* (*pick X*) (*pick Y*)
and *cong*: ⋀*x x′ y y′*. ⌊*x*⌋ = ⌊*x′*⌋ ⟹ ⌊*y*⌋ = ⌊*y′*⌋
⟹ *P* ⌊*x*⌋ ⌊*y*⌋ ⟹ *P* ⌊*x′*⌋ ⌊*y′*⌋ ⟹ *g x y* = *g x′ y′*
and *P*: *P* ⌊*a*⌋ ⌊*b*⌋
shows *f* ⌊*a*⌋ ⌊*b*⌋ = *g a b*
proof −
from *eq* **and** *P* **have** *f* ⌊*a*⌋ ⌊*b*⌋ = *g* (*pick* ⌊*a*⌋) (*pick* ⌊*b*⌋) **by** (*simp only*:)
also have ... = *g a b*
proof (*rule cong*)
show ⌊*pick* ⌊*a*⌋⌋ = ⌊*a*⌋ **..**
moreover
show ⌊*pick* ⌊*b*⌋⌋ = ⌊*b*⌋ **..**
moreover
show *P* ⌊*a*⌋ ⌊*b*⌋ **by** (*rule P*)
ultimately show *P* ⌊*pick* ⌊*a*⌋⌋ ⌊*pick* ⌊*b*⌋⌋ **by** (*simp only*:)
qed
finally show *?thesis* **.**
qed

theorem *quot-function*:
assumes ⋀*X Y*. *f X Y* ≡ *g* (*pick X*) (*pick Y*)
and ⋀*x x′ y y′*. ⌊*x*⌋ = ⌊*x′*⌋ ⟹ ⌊*y*⌋ = ⌊*y′*⌋ ⟹ *g x y* = *g x′ y′*
shows *f* ⌊*a*⌋ ⌊*b*⌋ = *g a b*
using *assms* **and** *TrueI*

by (*rule quot-cond-function*)

theorem *quot-function'*:
(⋀*X Y*. *f X Y* ≡ *g* (*pick X*) (*pick Y*)) ⟹
(⋀*x x' y y'*. *x* ∼ *x'* ⟹ *y* ∼ *y'* ⟹ *g x y* = *g x' y'*) ⟹
f ⌊*a*⌋ ⌊*b*⌋ = *g a b*
by (*rule quot-function*) (*simp-all only*: *quot-equality*)

end

91 Ramsey's Theorem

theory *Ramsey*
imports *Infinite-Set Equipollence FuncSet*
begin

91.1 Preliminary definitions

abbreviation *strict-sorted* :: *'a*::*linorder list* ⇒ *bool* **where**
strict-sorted ≡ *sorted-wrt* (<)

91.1.1 The n-element subsets of a set A

definition *nsets* :: [*'a set*, *nat*] ⇒ *'a set set* (‹(‹*notation*=‹*mixfix nsets*››[-]⁻)› [*0*,*999*] *999*)
where *nsets A n* ≡ {*N*. *N* ⊆ *A* ∧ *finite N* ∧ *card N* = *n*}

lemma *finite-imp-finite-nsets*: *finite A* ⟹ *finite* ([*A*]^*k*)
by (*simp add*: *nsets-def*)

lemma *nsets-mono*: *A* ⊆ *B* ⟹ *nsets A n* ⊆ *nsets B n*
by (*auto simp*: *nsets-def*)

lemma *nsets-Pi-contra*: *A'* ⊆ *A* ⟹ *Pi* ([*A*]^*n*) *B* ⊆ *Pi* ([*A'*]^*n*) *B*
by (*auto simp*: *nsets-def*)

lemma *nsets-2-eq*: [*A*]^2 = (⋃*x*∈*A*. ⋃*y*∈*A* − {*x*}. {{*x*, *y*}})
by (*auto simp*: *nsets-def card-2-iff*)

lemma *nsets2-E*:
assumes *e* ∈ [*A*]^2
obtains *x y* **where** *e* = {*x*,*y*} *x* ∈ *A* *y* ∈ *A* *x*≠*y*
using *assms* **by** (*auto simp*: *nsets-def card-2-iff*)

lemma *nsets-doubleton-2-eq* [*simp*]: [{*x*, *y*}]^2 = (*if x*=*y then* {} *else* {{*x*, *y*}})
by (*auto simp*: *nsets-2-eq*)

lemma *doubleton-in-nsets-2* [*simp*]: {*x*,*y*} ∈ [*A*]^2 ⟷ *x* ∈ *A* ∧ *y* ∈ *A* ∧ *x* ≠ *y*
by (*auto simp*: *nsets-2-eq Set.doubleton-eq-iff*)

lemma *nsets-3-eq*: $[A]^3 = (\bigcup x{\in}A.\ \bigcup y{\in}A - \{x\}.\ \bigcup z{\in}A - \{x,y\}.\ \{\{x,y,z\}\})$
by (*simp add*: *eval-nat-numeral nsets-def card-Suc-eq*) *blast*

lemma *nsets-4-eq*: $[A]^4 = (\bigcup u{\in}A.\ \bigcup x{\in}A - \{u\}.\ \bigcup y{\in}A - \{u,x\}.\ \bigcup z{\in}A - \{u,x,y\}.\ \{\{u,x,y,z\}\})$
(**is** - = *?rhs*)
proof
show $[A]^4 \subseteq$ *?rhs*
by (*clarsimp simp add*: *nsets-def eval-nat-numeral card-Suc-eq*) *blast*
have $\bigwedge X.\ X \in$ *?rhs* $\Longrightarrow$ *card* $X = 4$
by (*force simp*: *card-2-iff*)
then show *?rhs* $\subseteq [A]^4$
by (*auto simp*: *nsets-def*)
qed

lemma *nsets-disjoint-2*:
$X \cap Y = \{\} \Longrightarrow [X \cup Y]^2 = [X]^2 \cup [Y]^2 \cup (\bigcup x{\in}X.\ \bigcup y{\in}Y.\ \{\{x,y\}\})$
by (*fastforce simp*: *nsets-2-eq Set.doubleton-eq-iff*)

lemma *ordered-nsets-2-eq*:
fixes A :: *'a*::*linorder set*
shows $[A]^2 = \{\{x,y\} \mid x\ y.\ x \in A \wedge y \in A \wedge x{<}y\}$
(**is** - = *?rhs*)
proof
show $[A]^2 \subseteq$ *?rhs*
by (*auto simp*: *nsets-def card-2-iff doubleton-eq-iff neq-iff*)
show *?rhs* $\subseteq [A]^2$
unfolding *numeral-nat* **by** (*auto simp*: *nsets-def card-Suc-eq*)
qed

lemma *ordered-nsets-3-eq*:
fixes A :: *'a*::*linorder set*
shows $[A]^3 = \{\{x,y,z\} \mid x\ y\ z.\ x \in A \wedge y \in A \wedge z \in A \wedge x{<}y \wedge y{<}z\}$
(**is** - = *?rhs*)
proof
show $[A]^3 \subseteq$ *?rhs*
unfolding *nsets-def card-3-iff*
by (*smt* (*verit*, *del-insts*) *Collect-mono-iff insert-commute insert-subset linorder-less-linear*)
have $\bigwedge X.\ X \in$ *?rhs* $\Longrightarrow$ *card* $X = 3$
by (*force simp*: *card-3-iff*)
then show *?rhs* $\subseteq [A]^3$
by (*auto simp*: *nsets-def*)
qed

lemma *ordered-nsets-4-eq*:
fixes A :: *'a*::*linorder set*
defines *rhs* $\equiv \lambda U.\ \exists u\ x\ y\ z.\ U = \{u,x,y,z\} \wedge u \in A \wedge x \in A \wedge y \in A \wedge z \in A$

$\wedge\ u < x \wedge x < y \wedge y < z$
shows $[A]^4 = \mathit{Collect}\ \mathit{rhs}$
proof −
have *rhs U* **if** $U \in [A]^4$ **for** *U*
proof −
from *that* **obtain** *l* **where** *strict-sorted l List.set l = U length l = 4 U ⊆ A*
by (*simp add*: *nsets-def*) (*metis finite-set-strict-sorted*)
then show *?thesis*
unfolding *numeral-nat length-Suc-conv rhs-def* **by** *auto blast*
qed
moreover
have $\bigwedge X.\ X \in \mathit{Collect}\ \mathit{rhs} \Longrightarrow \mathit{card}\ X = 4 \wedge \mathit{finite}\ X \wedge X \subseteq A$
by (*auto simp*: *rhs-def card-insert-if*)
ultimately show *?thesis*
unfolding *nsets-def* **by** *blast*
qed

lemma *ordered-nsets-5-eq*:
fixes *A* :: *'a*::*linorder set*
defines *rhs* ≡ λ*U*. ∃ *u v x y z*. *U* = {*u*,*v*,*x*,*y*,*z*} ∧ *u* ∈ *A* ∧ *v* ∈ *A* ∧ *x* ∈ *A* ∧ *y* ∈ *A* ∧ *z* ∈ *A* ∧ *u* < *v* ∧ *v* < *x* ∧ *x* < *y* ∧ *y* < *z*
shows $[A]^5 = \mathit{Collect}\ \mathit{rhs}$
proof −
have *rhs U* **if** $U \in [A]^5$ **for** *U*
proof −
from *that* **obtain** *l* **where** *strict-sorted l List.set l = U length l = 5 U ⊆ A*
by (*simp add*: *nsets-def*) (*metis finite-set-strict-sorted*)
then show *?thesis*
unfolding *numeral-nat length-Suc-conv rhs-def* **by** *auto blast*
qed
moreover
have $\bigwedge X.\ X \in \mathit{Collect}\ \mathit{rhs} \Longrightarrow \mathit{card}\ X = 5 \wedge \mathit{finite}\ X \wedge X \subseteq A$
by (*auto simp*: *rhs-def card-insert-if*)
ultimately show *?thesis*
unfolding *nsets-def* **by** *blast*
qed

lemma *binomial-eq-nsets*: *n choose k = card* (*nsets* {*0*..<*n*} *k*)
proof −
have {*K*. *K* ⊆ {*0*..<*n*} ∧ *card K* = *k*} = {*N*. *N* ⊆ {*0*..<*n*} ∧ *finite N* ∧ *card N* = *k*}
using *infinite-super* **by** *blast*
then show *?thesis*
by (*simp add*: *binomial-def nsets-def*)
qed

lemma *nsets-eq-empty-iff*: *nsets A r* = {} ⟷ *finite A* ∧ *card A* < *r*
unfolding *nsets-def*
proof (*intro iffI conjI*)

```
  assume that: {N. N ⊆ A ∧ finite N ∧ card N = r} = {}
  show finite A
    using infinite-arbitrarily-large that by auto
  then have ¬ r ≤ card A
    using that by (simp add: set-eq-iff) (metis obtain-subset-with-card-n)
  then show card A < r
    using not-less by blast
next
  show {N. N ⊆ A ∧ finite N ∧ card N = r} = {}
    if finite A ∧ card A < r
    using that card-mono leD by auto
qed

lemma nsets-eq-empty: ⟦finite A; card A < r⟧ ⟹ nsets A r = {}
  by (simp add: nsets-eq-empty-iff)

lemma nsets-empty-iff: nsets {} r = (if r=0 then {{}} else {})
  by (auto simp: nsets-def)

lemma nsets-singleton-iff: nsets {a} r = (if r=0 then {{}} else if r=1 then {{a}}
else {})
  by (auto simp: nsets-def card-gt-0-iff subset-singleton-iff)

lemma nsets-self [simp]: nsets {..<m} m = {{..<m}}
proof
  show [{..<m}]^m ⊆ {{..<m}}
    by (force simp add: card-subset-eq nsets-def)
qed (simp add: nsets-def)

lemma nsets-zero [simp]: nsets A 0 = {{}}
  by (auto simp: nsets-def)

lemma nsets-one: nsets A (Suc 0) = (λx. {x}) ` A
  using card-eq-SucD by (force simp: nsets-def)

lemma inj-on-nsets:
  assumes inj-on f A
  shows inj-on (λX. f ` X) ([A]^n)
  using assms by (simp add: nsets-def inj-on-def inj-on-image-eq-iff)

lemma bij-betw-nsets:
  assumes bij-betw f A B
  shows bij-betw (λX. f ` X) ([A]^n) ([B]^n)
proof −
  have ⋀X. ⟦X ⊆ f ` A; finite X⟧ ⟹ ∃ Y⊆A. finite Y ∧ card Y = card X ∧ X
= f ` Y
    by (metis card-image inj-on-finite order-refl subset-image-inj)
  then have (`) f ` [A]^n = [f ` A]^n
   using assms by (auto simp: nsets-def bij-betw-def image-iff card-image inj-on-subset)
```

with *assms* **show** *?thesis*
by (*auto simp*: *bij-betw-def inj-on-nsets*)
qed

lemma *nset-image-obtains*:
assumes $X \in [f`A]^k$ *inj-on f A*
obtains Y **where** $Y \in [A]^k$ $X = f`Y$
proof
show $X = f`(A \cap f-`X)$
using *assms* **by** (*auto simp*: *nsets-def*)
then show A *Int* $(f-`X) \in [A]^k$
using *assms*
unfolding *nsets-def mem-Collect-eq*
by (*metis card-image finite-image-iff inf-le1 inj-on-subset*)
qed

lemma *nsets-image-funcset*:
assumes $g \in S \to T$ **and** *inj-on g S*
shows $(\lambda X.\ g`X) \in [S]^k \to [T]^k$
using *assms*
by (*fastforce simp*: *nsets-def card-image inj-on-subset subset-iff simp flip*: *image-subset-iff-funcset*)

lemma *nsets-compose-image-funcset*:
assumes *f*: $f \in [T]^k \to D$ **and** $g \in S \to T$ **and** *inj-on g S*
shows $f \circ (\lambda X.\ g`X) \in [S]^k \to D$
proof −
have $(\lambda X.\ g`X) \in [S]^k \to [T]^k$
using *assms* **by** (*simp add*: *nsets-image-funcset*)
then show *?thesis*
using *f* **by** *fastforce*
qed

91.1.2 Further properties, involving equipollence

lemma *nsets-lepoll-cong*:
assumes $A \lesssim B$
shows $[A]^k \lesssim [B]^k$
proof −
obtain *f* **where** *f*: *inj-on f A* $f`A \subseteq B$
by (*meson assms lepoll-def*)
define *F* **where** $F \equiv \lambda N.\ f`N$
have *inj-on F* $([A]^k)$
using *F-def f inj-on-nsets* **by** *blast*
moreover
have $F`([A]^k) \subseteq [B]^k$
by (*metis F-def bij-betw-def bij-betw-nsets f nsets-mono*)
ultimately show *?thesis*
by (*meson lepoll-def*)

qed

lemma *nsets-eqpoll-cong*:
assumes $A \approx B$
shows $[A]^k \approx [B]^k$
by (*meson assms eqpoll-imp-lepoll eqpoll-sym lepoll-antisym nsets-lepoll-cong*)

lemma *infinite-imp-infinite-nsets*:
assumes *inf*: *infinite A* **and** *k>0*
shows *infinite* $([A]^k)$
proof −
obtain *B* **where** $B \subset A$ $A \approx B$
by (*meson inf infinite-iff-psubset*)
then obtain *a* **where** *a*: $a \in A$ $a \notin B$
by *blast*
then obtain *N* **where** $N \subseteq B$ *finite N card N* = *k*−*1* $a \notin N$
by (*metis* ‹$A \approx B$› *inf eqpoll-finite-iff infinite-arbitrarily-large subset-eq*)
with *a* ‹*k>0*› ‹$B \subset A$› **have** *insert a N* $\in [A]^k$
by (*simp add*: *nsets-def*)
with *a* **have** *nsets B k* ≠ *nsets A k*
by (*metis* (*no-types*, *lifting*) *in-mono insertI1 mem-Collect-eq nsets-def*)
moreover have *nsets B k* ⊆ *nsets A k*
using ‹$B \subset A$› *nsets-mono* **by** *auto*
ultimately show *?thesis*
unfolding *infinite-iff-psubset-le*
by (*meson* ‹$A \approx B$› *eqpoll-imp-lepoll nsets-eqpoll-cong psubsetI*)
qed

lemma *finite-nsets-iff*:
assumes *k>0*
shows *finite* $([A]^k) \longleftrightarrow$ *finite A*
using *assms finite-imp-finite-nsets infinite-imp-infinite-nsets* **by** *blast*

lemma *card-nsets* [*simp*]: *card* (*nsets A k*) = *card A choose k*
proof (*cases finite A*)
case *True*
then show *?thesis*
by (*metis bij-betw-nsets bij-betw-same-card binomial-eq-nsets ex-bij-betw-nat-finite*)
next
case *False*
then show *?thesis*
by (*cases k=0*; *simp add*: *finite-nsets-iff*)
qed

91.1.3 Partition predicates

definition *monochromatic* ≡ $\lambda\beta\ \alpha\ \gamma\ f\ i.\ \exists H \in nsets\ \beta\ \alpha.\ f\ `\ (nsets\ H\ \gamma) \subseteq \{i\}$

uniform partition sizes

definition *partn* :: *'a set* ⇒ *nat* ⇒ *nat* ⇒ *'b set* ⇒ *bool*

where *partn* β α γ $\delta \equiv \forall f \in nsets\ \beta\ \gamma\ \rightarrow\ \delta.\ \exists \xi{\in}\delta.\ monochromatic\ \beta\ \alpha\ \gamma\ f\ \xi$

partition sizes enumerated in a list

definition *partn-lst* :: *'a set* ⇒ *nat list* ⇒ *nat* ⇒ *bool*
where *partn-lst* β α $\gamma \equiv \forall f \in nsets\ \beta\ \gamma\ \rightarrow\ \{..{<}length\ \alpha\}.\ \exists i < length\ \alpha.$ *monochromatic* β $(\alpha!i)$ γ *f i*

There's always a 0-clique

lemma *partn-lst-0*: $\gamma > 0 \Longrightarrow$ *partn-lst* β $(0\#\alpha)$ γ
by (*force simp*: *partn-lst-def monochromatic-def nsets-empty-iff*)

lemma *partn-lst-0'*: $\gamma > 0 \Longrightarrow$ *partn-lst* β $(a\#0\#\alpha)$ γ
by (*force simp*: *partn-lst-def monochromatic-def nsets-empty-iff*)

lemma *partn-lst-greater-resource*:
fixes *M*::*nat*
assumes *M*: *partn-lst* $\{..{<}M\}$ α γ **and** $M \leq N$
shows *partn-lst* $\{..{<}N\}$ α γ
proof (*clarsimp simp*: *partn-lst-def*)
fix *f*
assume $f \in nsets\ \{..{<}N\}\ \gamma \rightarrow \{..{<}length\ \alpha\}$
then have $f \in nsets\ \{..{<}M\}\ \gamma \rightarrow \{..{<}length\ \alpha\}$
by (*meson Pi-anti-mono* ‹$M \leq N$› *lessThan-subset-iff nsets-mono subsetD*)
then obtain *i H* **where** *i*: $i < length\ \alpha$ **and** *H*: $H \in nsets\ \{..{<}M\}\ (\alpha\ !\ i)$ **and** *subi*: $f\ `\ nsets\ H\ \gamma \subseteq \{i\}$
using *M* **unfolding** *partn-lst-def monochromatic-def* **by** *blast*
have $H \in nsets\ \{..{<}N\}\ (\alpha\ !\ i)$
using ‹$M \leq N$› *H* **by** (*auto simp*: *nsets-def subset-iff*)
then show $\exists i{<}length\ \alpha.$ *monochromatic* $\{..{<}N\}$ $(\alpha!i)$ γ *f i*
using *i subi* **unfolding** *monochromatic-def* **by** *blast*
qed

lemma *partn-lst-fewer-colours*:
assumes *major*: *partn-lst* β $(n\#\alpha)$ γ **and** $n \geq \gamma$
shows *partn-lst* β α γ
proof (*clarsimp simp*: *partn-lst-def*)
fix *f* :: *'a set* ⇒ *nat*
assume *f*: $f \in [\beta]^{\gamma} \rightarrow \{..{<}length\ \alpha\}$
then obtain *i H* **where** *i*: $i < Suc\ (length\ \alpha)$
and *H*: $H \in [\beta]^{((n\ \#\ \alpha)\ !\ i)}$
and *hom*: $\forall x{\in}[H]^{\gamma}.\ Suc\ (f\ x) = i$
using ‹$n \geq \gamma$› *major* [*unfolded partn-lst-def*, *rule-format*, *of Suc o f*]
by (*fastforce simp*: *image-subset-iff nsets-eq-empty-iff monochromatic-def*)
show $\exists i{<}length\ \alpha.$ *monochromatic* β $(\alpha!i)$ γ *f i*
proof (*cases i*)
case *0*
then have $[H]^{\gamma} = \{\}$
using *hom* **by** *blast*
then show *?thesis*

```
    using 0 H ‹n ≥ γ›
    by (simp add: nsets-eq-empty-iff) (simp add: nsets-def)
  next
    case (Suc i')
    then show ?thesis
      unfolding monochromatic-def using i H hom by auto
  qed
qed

lemma partn-lst-eq-partn: partn-lst {..<n} [m,m] 2 = partn {..<n} m 2 {..<2::nat}
proof −
  have ⋀i. i < 2 ⟹ [m, m] ! i = m
    using less-2-cases-iff by force
  then show ?thesis
    by (auto simp: partn-lst-def partn-def numeral-2-eq-2 cong: conj-cong)
qed

lemma partn-lstE:
  assumes partn-lst β α γ f ∈ nsets β γ → {..<l} length α = l
  obtains i H where i < length α H ∈ nsets β (α!i) f ‘ (nsets H γ) ⊆ {i}
  using partn-lst-def monochromatic-def assms by metis

lemma partn-lst-less:
  assumes M: partn-lst β α n and eq: length α' = length α
    and le: ⋀i. i < length α ⟹ α'!i ≤ α!i
  shows partn-lst β α' n
proof (clarsimp simp: partn-lst-def)
  fix f
  assume f ∈ [β]^n → {..<length α'}
  then obtain i H where i: i < length α
                  and H ⊆ β and H: card H = (α!i) and finite H
                  and fi: f ‘ nsets H n ⊆ {i}
    using assms by (auto simp: partn-lst-def monochromatic-def nsets-def)
  then obtain bij where bij: bij-betw bij H {0..<α!i}
    by (metis ex-bij-betw-finite-nat)
  then have inj: inj-on (inv-into H bij) {0..<α' ! i}
    by (metis bij-betw-def dual-order.refl i inj-on-inv-into ivl-subset le)
  define H' where H' = inv-into H bij ‘ {0..<α'!i}
  show ∃i<length α'. monochromatic β (α'!i) n f i
    unfolding monochromatic-def
  proof (intro exI bexI conjI)
    show i < length α'
      by (simp add: assms(2) i)
    have H' ⊆ H
      using bij ‹i < length α› bij-betw-imp-surj-on le
      by (force simp: H'-def image-subset-iff intro: inv-into-into)
    then have finite H'
      by (simp add: ‹finite H› finite-subset)
    with ‹H' ⊆ H› have cardH': card H' = (α'!i)
```

unfolding *H'-def* **by** (*simp add*: *inj card-image*)
show f ' $[H]^n \subseteq \{i\}$
by (*meson* ‹$H' \subseteq H$› *dual-order.trans fi image-mono nsets-mono*)
show $H' \in [\beta]^{(\alpha' !\ i)}$
using ‹$H \subseteq \beta$› ‹$H' \subseteq H$› ‹*finite H'*› *cardH' nsets-def* **by** *fastforce*
qed
qed

91.2 Finite versions of Ramsey's theorem

To distinguish the finite and infinite ones, lower and upper case names are used (ramsey vs Ramsey).

91.2.1 The Erds–Szekeres theorem exhibits an upper bound for Ramsey numbers

The Erds–Szekeres bound, essentially extracted from the proof

fun *ES* :: [*nat*,*nat*,*nat*] ⇒ *nat*
where *ES 0 k l* = *max k l*
| *ES* (*Suc r*) *k l* =
(*if r=0 then k+l−1*
else if k=0 ∨ *l=0 then 1 else Suc* (*ES r* (*ES* (*Suc r*) (*k−1*) *l*) (*ES* (*Suc r*) *k* (*l−1*))))

declare *ES.simps* [*simp del*]

lemma *ES-0* [*simp*]: *ES 0 k l* = *max k l*
using *ES.simps*(*1*) **by** *blast*

lemma *ES-1* [*simp*]: *ES 1 k l* = *k+l−1*
using *ES.simps*(*2*) [*of 0 k l*] **by** *simp*

lemma *ES-2*: *ES 2 k l* = (*if k=0* ∨ *l=0 then 1 else ES 2* (*k−1*) *l* + *ES 2 k* (*l−1*))
unfolding *numeral-2-eq-2*
by (*smt* (*verit*) *ES.elims One-nat-def Suc-pred add-gr-0 neq0-conv nat.inject zero-less-Suc*)

The Erds–Szekeres upper bound

lemma *ES2-choose*: *ES 2 k l* = (*k+l*) *choose k*
proof (*induct n* ≡ *k+l arbitrary*: *k l*)
case *0*
then show *?case*
by (*auto simp*: *ES-2*)
next
case (*Suc n*)
then have *k>0* ⟹ *l>0* ⟹ *ES 2* (*k* − *1*) *l* + *ES 2 k* (*l* − *1*) = *k* + *l choose k*
using *choose-reduce-nat* **by** *force*
then show *?case*

by (*metis ES-2 Nat.add-0-right binomial-n-0 binomial-n-n gr0I*)
qed

91.2.2 Trivial cases

Vacuous, since we are dealing with 0-sets!

lemma *ramsey0*: ∃ *N*::*nat*. *partn-lst* {..<*N*} [*q1*,*q2*] *0*
by (*force simp*: *partn-lst-def monochromatic-def ex-in-conv less-Suc-eq nsets-eq-empty-iff*)

Just the pigeon hole principle, since we are dealing with 1-sets

lemma *ramsey1-explicit*: *partn-lst* {..<*q0* + *q1* − *Suc 0*} [*q0*,*q1*] *1*
proof −
have ∃ *i*<*Suc* (*Suc 0*). ∃ *H*∈*nsets* {..<*q0* + *q1* − *1*} ([*q0*, *q1*] ! *i*). *f* ‘ *nsets H 1* ⊆ {*i*}
if *f* ∈ *nsets* {..<*q0* + *q1* − *1*} (*Suc 0*) → {..<*Suc* (*Suc 0*)} **for** *f*
proof −
define *A* **where** *A* ≡ λ*i*. {*q*. *q* < *q0*+*q1*−*1* ∧ *f* {*q*} = *i*}
have *A 0* ∪ *A 1* = {..<*q0* + *q1*−*1*}
using *that* **by** (*auto simp*: *A-def PiE-iff nsets-one lessThan-Suc-atMost le-Suc-eq*)
moreover have *A 0* ∩ *A 1* = {}
by (*auto simp*: *A-def*)
ultimately have *q0* + *q1* ≤ *card* (*A 0*) + *card* (*A 1*) + *1*
by (*metis card-Un-le card-lessThan le-diff-conv*)
then consider *card* (*A 0*) ≥ *q0* | *card* (*A 1*) ≥ *q1*
by *linarith*
then obtain *i* **where** *i* < *Suc* (*Suc 0*) *card* (*A i*) ≥ [*q0*, *q1*] ! *i*
by (*metis One-nat-def lessI nth-Cons-0 nth-Cons-Suc zero-less-Suc*)
then obtain *B* **where** *B* ⊆ *A i card B* = [*q0*, *q1*] ! *i finite B*
by (*meson obtain-subset-with-card-n*)
then have *B* ∈ *nsets* {..<*q0* + *q1* − *1*} ([*q0*, *q1*] ! *i*) ∧ *f* ‘ *nsets B* (*Suc 0*) ⊆ {*i*}
by (*auto simp*: *A-def nsets-def card-1-singleton-iff*)
then show *?thesis*
using ‹*i* < *Suc* (*Suc 0*)› **by** *auto*
qed
then show *?thesis*
by (*simp add*: *partn-lst-def monochromatic-def*)
qed

lemma *ramsey1*: ∃ *N*::*nat*. *partn-lst* {..<*N*} [*q0*,*q1*] *1*
using *ramsey1-explicit* **by** *blast*

91.2.3 Ramsey's theorem with TWO colours and arbitrary exponents (hypergraph version)

lemma *ramsey-induction-step*:
fixes *p*::*nat*

```
  assumes p1: partn-lst {..<p1} [q1−1,q2] (Suc r) and p2: partn-lst {..<p2}
[q1,q2−1] (Suc r)
    and p: partn-lst {..<p} [p1,p2] r
    and q1>0 q2>0
  shows partn-lst {..<Suc p} [q1, q2] (Suc r)
proof −
  have ∃ i<Suc (Suc 0). ∃ H∈nsets {..p} ([q1,q2] ! i). f ‘ nsets H (Suc r) ⊆ {i}
    if f: f ∈ nsets {..p} (Suc r) → {..<Suc (Suc 0)} for f
  proof −
    define g where g ≡ λR. f (insert p R)
    have f (insert p i) ∈ {..<Suc (Suc 0)} if i ∈ nsets {..<p} r for i
      using that card-insert-if by (fastforce simp: nsets-def intro!: Pi-mem [OF f])
    then have g: g ∈ nsets {..<p} r → {..<Suc (Suc 0)}
      by (force simp: g-def PiE-iff)
    then obtain i U where i: i < Suc (Suc 0) and gi: g ‘ nsets U r ⊆ {i}
      and U: U ∈ nsets {..<p} ([p1, p2] ! i)
      using p by (auto simp: partn-lst-def monochromatic-def)
    then have Usub: U ⊆ {..<p}
      by (auto simp: nsets-def)
    consider (izero) i = 0 | (ione) i = Suc 0
      using i by linarith
    then show ?thesis
    proof cases
      case izero
      then have U ∈ nsets {..<p} p1
        using U by simp
      then obtain u where u: bij-betw u {..<p1} U
       using ex-bij-betw-nat-finite lessThan-atLeast0 by (fastforce simp: nsets-def)
      have u-nsets: u ‘ X ∈ nsets {..p} n if X ∈ nsets {..<p1} n for X n
      proof −
        have inj-on u X
          using u that bij-betw-imp-inj-on inj-on-subset by (force simp: nsets-def)
        then show ?thesis
          using Usub u that bij-betwE
          by (fastforce simp: nsets-def card-image)
      qed
      define h where h ≡ λR. f (u ‘ R)
      have h ∈ nsets {..<p1} (Suc r) → {..<Suc (Suc 0)}
        unfolding h-def using f u-nsets by auto
      then obtain j V where j: j <Suc (Suc 0) and hj: h ‘ nsets V (Suc r) ⊆ {j}
        and V: V ∈ nsets {..<p1} ([q1 − Suc 0, q2] ! j)
        using p1 by (auto simp: partn-lst-def monochromatic-def)
      then have Vsub: V ⊆ {..<p1}
        by (auto simp: nsets-def)
      have invinv-eq: u ‘ inv-into {..<p1} u ‘ X = X if X ⊆ u ‘ {..<p1} for X
        by (simp add: image-inv-into-cancel that)
      let ?W = insert p (u ‘ V)
      consider (jzero) j = 0 | (jone) j = Suc 0
        using j by linarith
```

```
    then show ?thesis
    proof cases
      case jzero
      then have V ∈ nsets {..<p1} (q1 − Suc 0)
        using V by simp
      then have u ‘ V ∈ nsets {..<p} (q1 − Suc 0)
        using u-nsets [of - q1 − Suc 0] nsets-mono [OF Vsub] Usub u
        unfolding bij-betw-def nsets-def
        by (fastforce elim!: subsetD)
      then have inq1: ?W ∈ nsets {..p} q1
        unfolding nsets-def using ‹q1 > 0› card-insert-if by fastforce
      have invu-nsets: inv-into {..<p1} u ‘ X ∈ nsets V r
        if X ∈ nsets (u ‘ V) r for X r
      proof −
        have X ⊆ u ‘ V ∧ finite X ∧ card X = r
          using nsets-def that by auto
        then have [simp]: card (inv-into {..<p1} u ‘ X) = card X
            by (meson Vsub bij-betw-def bij-betw-inv-into card-image image-mono
inj-on-subset u)
        show ?thesis
          using that u Vsub by (fastforce simp: nsets-def bij-betw-def)
      qed
      have f X = i if X: X ∈ nsets ?W (Suc r) for X
      proof (cases p ∈ X)
        case True
        then have Xp: X − {p} ∈ nsets (u ‘ V) r
          using X by (auto simp: nsets-def)
        moreover have u ‘ V ⊆ U
          using Vsub bij-betwE u by blast
        ultimately have X − {p} ∈ nsets U r
          by (meson in-mono nsets-mono)
        then have g (X − {p}) = i
          using gi by blast
        have f X = i
          using gi True ‹X − {p} ∈ nsets U r› insert-Diff
          by (fastforce simp: g-def image-subset-iff)
        then show ?thesis
          by (simp add: ‹f X = i› ‹g (X − {p}) = i›)
      next
        case False
        then have Xim: X ∈ nsets (u ‘ V) (Suc r)
          using X by (auto simp: nsets-def subset-insert)
        then have u ‘ inv-into {..<p1} u ‘ X = X
          using Vsub bij-betw-imp-inj-on u
          by (fastforce simp: nsets-def image-mono invinv-eq subset-trans)
        then show ?thesis
          using izero jzero hj Xim invu-nsets unfolding h-def
          by (fastforce simp: image-subset-iff)
      qed
```

```
      moreover have insert p (u ‘ V) ∈ nsets {..p} q1
        by (simp add: izero inq1)
      ultimately show ?thesis
        by (metis izero image-subsetI insertI1 nth-Cons-0 zero-less-Suc)
    next
      case jone
      then have u ‘ V ∈ nsets {..p} q2
        using V u-nsets by auto
      moreover have f ‘ nsets (u ‘ V) (Suc r) ⊆ {j}
        using hj
       by (force simp: h-def image-subset-iff nsets-def subset-image-inj card-image
dest: finite-imageD)
      ultimately show ?thesis
        using jone not-less-eq by fastforce
    qed
  next
    case ione
    then have U ∈ nsets {..<p} p2
      using U by simp
    then obtain u where u: bij-betw u {..<p2} U
     using ex-bij-betw-nat-finite lessThan-atLeast0 by (fastforce simp: nsets-def)
    have u-nsets: u ‘ X ∈ nsets {..p} n if X ∈ nsets {..<p2} n for X n
    proof −
      have inj-on u X
        using u that bij-betw-imp-inj-on inj-on-subset by (force simp: nsets-def)
      then show ?thesis
        using Usub u that bij-betwE
        by (fastforce simp: nsets-def card-image)
    qed
    define h where h ≡ λR. f (u ‘ R)
    have h ∈ nsets {..<p2} (Suc r) → {..<Suc (Suc 0)}
      unfolding h-def using f u-nsets by auto
   then obtain j V where j: j <Suc (Suc 0) and hj: h ‘ nsets V (Suc r) ⊆ {j}
      and V: V ∈ nsets {..<p2} ([q1, q2 − Suc 0] ! j)
      using p2 by (auto simp: partn-lst-def monochromatic-def)
    then have Vsub: V ⊆ {..<p2}
      by (auto simp: nsets-def)
    have invinv-eq: u ‘ inv-into {..<p2} u ‘ X = X if X ⊆ u ‘ {..<p2} for X
      by (simp add: image-inv-into-cancel that)
    let ?W = insert p (u ‘ V)
    consider (jzero) j = 0 | (jone) j = Suc 0
      using j by linarith
    then show ?thesis
    proof cases
      case jone
      then have V ∈ nsets {..<p2} (q2 − Suc 0)
        using V by simp
      then have u ‘ V ∈ nsets {..<p} (q2 − Suc 0)
        using u-nsets [of - q2 − Suc 0] nsets-mono [OF Vsub] Usub u
```

```
      unfolding bij-betw-def nsets-def by blast
    then have inq1: ?W ∈ nsets {..p} q2
      unfolding nsets-def using ‹q2 > 0› card-insert-if by fastforce
    have invu-nsets: inv-into {..<p2} u ` X ∈ nsets V r
      if X ∈ nsets (u ` V) r for X r
    proof −
      have X ⊆ u ` V ∧ finite X ∧ card X = r
        using nsets-def that by auto
      then have [simp]: card (inv-into {..<p2} u ` X) = card X
          by (meson Vsub bij-betw-def bij-betw-inv-into card-image image-mono
inj-on-subset u)
      show ?thesis
        using that u Vsub by (fastforce simp: nsets-def bij-betw-def)
    qed
    have f X = i if X: X ∈ nsets ?W (Suc r) for X
    proof (cases p ∈ X)
      case True
      then have Xp: X − {p} ∈ nsets (u ` V) r
        using X by (auto simp: nsets-def)
      moreover have u ` V ⊆ U
        using Vsub bij-betwE u by blast
      ultimately have X − {p} ∈ nsets U r
        by (meson in-mono nsets-mono)
      then have g (X − {p}) = i
        using gi by blast
      have f X = i
        using gi True ‹X − {p} ∈ nsets U r› insert-Diff
        by (fastforce simp: g-def image-subset-iff)
      then show ?thesis
        by (simp add: ‹f X = i› ‹g (X − {p}) = i›)
    next
      case False
      then have Xim: X ∈ nsets (u ` V) (Suc r)
        using X by (auto simp: nsets-def subset-insert)
      then have u ` inv-into {..<p2} u ` X = X
        using Vsub bij-betw-imp-inj-on u
        by (fastforce simp: nsets-def image-mono invinv-eq subset-trans)
      then show ?thesis
        using ione jone hj Xim invu-nsets unfolding h-def
        by (fastforce simp: image-subset-iff)
    qed
    moreover have insert p (u ` V) ∈ nsets {..p} q2
      by (simp add: ione inq1)
    ultimately show ?thesis
      by (metis ione image-subsetI insertI1 lessI nth-Cons-0 nth-Cons-Suc)
  next
    case jzero
    then have u ` V ∈ nsets {..p} q1
      using V u-nsets by auto
```

```
      moreover have f ‘ nsets (u ‘ V) (Suc r) ⊆ {j}
        using hj unfolding h-def image-subset-iff nsets-def
        apply (clarsimp simp add: h-def image-subset-iff nsets-def)
        by (metis card-image finite-imageD subset-image-inj)
      ultimately show ?thesis
        using jzero not-less-eq by fastforce
    qed
   qed
  qed
  then show ?thesis
   using lessThan-Suc lessThan-Suc-atMost
   by (auto simp: partn-lst-def monochromatic-def insert-commute)
qed

proposition ramsey2-full: partn-lst {..<ES r q1 q2} [q1,q2] r
proof (induction r arbitrary: q1 q2)
  case 0
  then show ?case
   by (auto simp: partn-lst-def monochromatic-def less-Suc-eq ex-in-conv nsets-eq-empty-iff)
next
  case (Suc r)
  note outer = this
  show ?case
  proof (cases r = 0)
    case True
    then show ?thesis
      using ramsey1-explicit by (force simp: ES.simps)
  next
    case False
    then have r > 0
      by simp
    show ?thesis
      using Suc.prems
    proof (induct k ≡ q1 + q2 arbitrary: q1 q2)
      case 0
      with partn-lst-0 show ?case by auto
    next
      case (Suc k)
      consider q1 = 0 ∨ q2 = 0 | q1 ≠ 0 q2 ≠ 0 by auto
      then show ?case
      proof cases
        case 1
        with False partn-lst-0 partn-lst-0' show ?thesis
          by blast
      next
        define p1 where p1 ≡ ES (Suc r) (q1−1) q2
        define p2 where p2 ≡ ES (Suc r) q1 (q2−1)
        define p where p ≡ ES r p1 p2
        case 2
```

```
      with Suc have k = (q1−1) + q2 k = q1 + (q2 − 1) by auto
      then have p1: partn-lst {..<p1} [q1−1,q2] (Suc r)
           and p2: partn-lst {..<p2} [q1,q2−1] (Suc r)
        using Suc.hyps unfolding p1-def p2-def by blast+
      then have p: partn-lst {..<p} [p1,p2] r
        using outer Suc.prems unfolding p-def by auto
      show ?thesis
         using ramsey-induction-step [OF p1 p2 p] 2 ES.simps(2) False p1-def
p2-def p-def by auto
    qed
   qed
  qed
qed
```

91.2.4 Full Ramsey's theorem with multiple colours and arbitrary exponents

```
theorem ramsey-full: ∃ N::nat. partn-lst {..<N} qs r
proof (induction k ≡ length qs arbitrary: qs)
  case 0
  then show ?case
    by (rule-tac x= r in exI) (simp add: partn-lst-def)
next
  case (Suc k)
  note IH = this
  show ?case
  proof (cases k)
    case 0
    with Suc obtain q where qs = [q]
      by (metis length-0-conv length-Suc-conv)
    then show ?thesis
      by (rule-tac x=q in exI) (auto simp: partn-lst-def monochromatic-def func-
set-to-empty-iff)
  next
    case (Suc k')
    then obtain q1 q2 l where qs: qs = q1#q2#l
      by (metis Suc.hyps(2) length-Suc-conv)
    then obtain q::nat where q: partn-lst {..<q} [q1,q2] r
      using ramsey2-full by blast
    then obtain p::nat where p: partn-lst {..<p} (q#l) r
      using IH ‹qs = q1 # q2 # l› by fastforce
    have keq: Suc (length l) = k
      using IH qs by auto
    show ?thesis
    proof (intro exI conjI)
      show partn-lst {..<p} qs r
      proof (auto simp: partn-lst-def)
        fix f
        assume f: f ∈ nsets {..<p} r → {..<length qs}
```

define g **where** $g \equiv \lambda X.$ *if* $f\ X < Suc\ (Suc\ 0)$ *then* 0 *else* $f\ X - Suc\ 0$
have $g \in nsets\ \{..<p\}\ r \to \{..<k\}$
unfolding *g-def* **using** *f Suc IH*
by (*auto simp*: *Pi-def not-less*)
then obtain $i\ U$ **where** i: $i < k$ **and** *gi*: g ‘ $nsets\ U\ r \subseteq \{i\}$
and U: $U \in nsets\ \{..<p\}\ ((q\#l)\ !\ i)$
using p *keq* **by** (*auto simp*: *partn-lst-def monochromatic-def*)
show $\exists i{<}length\ qs.$ *monochromatic* $\{..<p\}\ (qs!i)\ r\ f\ i$
proof (*cases* $i = 0$)
case *True*
then have $U \in nsets\ \{..<p\}\ q$ **and** *f01*: f ‘ $nsets\ U\ r \subseteq \{0, Suc\ 0\}$
using U *gi* **unfolding** *g-def* **by** (*auto simp*: *image-subset-iff*)
then obtain u **where** u: *bij-betw* $u\ \{..<q\}\ U$
using *ex-bij-betw-nat-finite lessThan-atLeast0* **by** (*fastforce simp*: *nsets-def*)
then have *Usub*: $U \subseteq \{..<p\}$
by (*smt* (*verit*) U *mem-Collect-eq nsets-def*)
have *u-nsets*: u ‘ $X \in nsets\ \{..<p\}\ n$ **if** $X \in nsets\ \{..<q\}\ n$ **for** $X\ n$
proof −
have *inj-on u X*
using *u that bij-betw-imp-inj-on inj-on-subset*
by (*force simp*: *nsets-def*)
then show *?thesis*
using *Usub u that bij-betwE*
by (*fastforce simp*: *nsets-def card-image*)
qed
define h **where** $h \equiv \lambda X.\ f\ (u$ ‘ $X)$
have $f\ (u$ ‘ $X) < Suc\ (Suc\ 0)$ **if** $X \in nsets\ \{..<q\}\ r$ **for** X
proof −
have u ‘ $X \in nsets\ U\ r$
using u *u-nsets that* **by** (*auto simp*: *nsets-def bij-betwE subset-eq*)
then show *?thesis*
using *f01* **by** *auto*
qed
then have $h \in nsets\ \{..<q\}\ r \to \{..<Suc\ (Suc\ 0)\}$
unfolding *h-def* **by** *blast*
then obtain $j\ V$ **where** j: $j < Suc\ (Suc\ 0)$ **and** *hj*: h ‘ $nsets\ V\ r \subseteq \{j\}$
and V: $V \in nsets\ \{..<q\}\ ([q1,q2]\ !\ j)$
using q **by** (*auto simp*: *partn-lst-def monochromatic-def*)
show *?thesis*
unfolding *monochromatic-def*
proof (*intro exI conjI bexI*)
show $j < length\ qs$
using *Suc Suc.hyps*(*2*) j **by** *linarith*
have *nsets* $(u$ ‘ $V)\ r \subseteq (\lambda x.\ (u$ ‘ $x))$ ‘ $nsets\ V\ r$
apply (*clarsimp simp add*: *nsets-def image-iff*)
by (*metis card-image finite-imageD subset-image-inj*)
then have f ‘ $nsets\ (u$ ‘ $V)\ r \subseteq h$ ‘ $nsets\ V\ r$
by (*auto simp*: *h-def*)

```
          then show f ‘ nsets (u ‘ V) r ⊆ {j}
            using hj by auto
          show (u ‘ V) ∈ nsets {..<p} (qs ! j)
            using V j less-2-cases numeral-2-eq-2 qs u-nsets by fastforce
        qed
      next
        case False
        then have eq: ⋀A. ⟦A ∈ [U]^r⟧ ⟹ f A = Suc i
        by (metis Suc-pred diff-0-eq-0 g-def gi image-subset-iff not-gr0 singletonD)
        show ?thesis
          unfolding monochromatic-def
        proof (intro exI conjI bexI)
          show Suc i < length qs
            using Suc.hyps(2) i by auto
          show f ‘ nsets U r ⊆ {Suc i}
            using False by (auto simp: eq)
          show U ∈ nsets {..<p} (qs ! (Suc i))
            using False U qs by auto
        qed
      qed
    qed
  qed
qed
```

91.2.5 Simple graph version

This is the most basic version in terms of cliques and independent sets, i.e. the version for graphs and 2 colours.

```
definition clique V E ⟷ (∀ v∈V. ∀ w∈V. v ≠ w ⟶ {v, w} ∈ E)
definition indep V E ⟷ (∀ v∈V. ∀ w∈V. v ≠ w ⟶ {v, w} ∉ E)

lemma clique-Un: ⟦clique K F; clique L F; ∀ v∈K. ∀ w∈L. v≠w ⟶ {v,w} ∈ F⟧
⟹ clique (K ∪ L) F
  by (metis UnE clique-def doubleton-eq-iff)

lemma null-clique[simp]: clique {} E and null-indep[simp]: indep {} E
  by (auto simp: clique-def indep-def)

lemma smaller-clique: ⟦clique R E; R' ⊆ R⟧ ⟹ clique R' E
  by (auto simp: clique-def)

lemma smaller-indep: ⟦indep R E; R' ⊆ R⟧ ⟹ indep R' E
  by (auto simp: indep-def)

lemma ramsey2:
  ∃ r≥1. ∀ (V::'a set) (E::'a set set). finite V ∧ card V ≥ r ⟶
    (∃ R ⊆ V. card R = m ∧ clique R E ∨ card R = n ∧ indep R E)
proof −
```

```
  obtain N where N ≥ Suc 0 and N: partn-lst {..<N} [m,n] 2
    using ramsey2-full nat-le-linear partn-lst-greater-resource by blast
  have ∃ R⊆V. card R = m ∧ clique R E ∨ card R = n ∧ indep R E
    if finite V N ≤ card V for V :: 'a set and E :: 'a set set
  proof −
    from that
    obtain v where u: inj-on v {..<N} v ‘ {..<N} ⊆ V
      by (metis card-le-inj card-lessThan finite-lessThan)
    define f where f ≡ λe. if v ‘ e ∈ E then 0 else Suc 0
    have f: f ∈ nsets {..<N} 2 → {..<Suc (Suc 0)}
      by (simp add: f-def)
    then obtain i U where i: i < 2 and gi: f ‘ nsets U 2 ⊆ {i}
      and U: U ∈ nsets {..<N} ([m,n] ! i)
      using N numeral-2-eq-2 by (auto simp: partn-lst-def monochromatic-def)
    show ?thesis
    proof (intro exI conjI)
      show v ‘ U ⊆ V
        using U u by (auto simp: image-subset-iff nsets-def)
      show card (v ‘ U) = m ∧ clique (v ‘ U) E ∨ card (v ‘ U) = n ∧ indep (v ‘
U) E
        using i unfolding numeral-2-eq-2
          using gi U u
          unfolding image-subset-iff nsets-2-eq clique-def indep-def less-Suc-eq
          by (auto simp: f-def nsets-def card-image inj-on-subset split: if-splits)
    qed
  qed
  then show ?thesis
    using ‹Suc 0 ≤ N› by auto
qed
```

91.3 Preliminaries for the infinitary version

91.3.1 "Axiom" of Dependent Choice

```
primrec choice :: ('a ⇒ bool) ⇒ ('a × 'a) set ⇒ nat ⇒ 'a
  where — An integer-indexed chain of choices
    choice-0: choice P r 0 = (SOME x. P x)
  | choice-Suc: choice P r (Suc n) = (SOME y. P y ∧ (choice P r n, y) ∈ r)

lemma choice-n:
  assumes P0: P x0
    and Pstep: ⋀x. P x ⟹ ∃ y. P y ∧ (x, y) ∈ r
  shows P (choice P r n)
proof (induct n)
  case 0
  show ?case by (force intro: someI P0)
next
  case Suc
  then show ?case by (auto intro: someI2-ex [OF Pstep])
qed
```

lemma *dependent-choice*:
assumes *trans*: *trans r*
and *P0*: *P x0*
and *Pstep*: $\bigwedge x.\ P\ x \Longrightarrow \exists\, y.\ P\ y \wedge (x,\ y) \in r$
obtains *f* :: *nat* ⇒ *'a* **where** $\bigwedge n.\ P\ (f\ n)$ **and** $\bigwedge n\ m.\ n < m \Longrightarrow (f\ n,\ f\ m) \in r$
proof
fix *n*
show *P* (*choice P r n*)
by (*blast intro*: *choice-n* [*OF P0 Pstep*])
next
fix *n m* :: *nat*
assume *n* < *m*
from *Pstep* [*OF choice-n* [*OF P0 Pstep*]] **have** (*choice P r k*, *choice P r* (*Suc k*)) ∈ *r* **for** *k*
by (*auto intro*: *someI2-ex*)
then show (*choice P r n*, *choice P r m*) ∈ *r*
by (*auto intro*: *less-Suc-induct* [*OF* ‹*n* < *m*›] *transD* [*OF trans*])
qed

91.3.2 Partition functions

definition *part-fn* :: *nat* ⇒ *nat* ⇒ *'a set* ⇒ (*'a set* ⇒ *nat*) ⇒ *bool*
— the function *f* partitions the *r*-subsets of the typically infinite set *Y* into *s* distinct categories.
where *part-fn r s Y f* ⟷ (*f* ∈ *nsets Y r* → {..<*s*})

For induction, we decrease the value of *r* in partitions.

lemma *part-fn-Suc-imp-part-fn*:
⟦*infinite Y*; *part-fn* (*Suc r*) *s Y f*; *y* ∈ *Y*⟧ ⟹ *part-fn r s* (*Y* − {*y*}) (λ*u*. *f* (*insert y u*))
by (*simp add*: *part-fn-def nsets-def Pi-def subset-Diff-insert*)

lemma *part-fn-subset*: *part-fn r s YY f* ⟹ *Y* ⊆ *YY* ⟹ *part-fn r s Y f*
unfolding *part-fn-def nsets-def* **by** *blast*

91.4 Ramsey's Theorem: Infinitary Version

lemma *Ramsey-induction*:
fixes *s r* :: *nat*
and *YY* :: *'a set*
and *f* :: *'a set* ⇒ *nat*
assumes *infinite YY part-fn r s YY f*
shows ∃ *Y' t'*. *Y'* ⊆ *YY* ∧ *infinite Y'* ∧ *t'* < *s* ∧ (∀ *X*. *X* ⊆ *Y'* ∧ *finite X* ∧ *card X* = *r* ⟶ *f X* = *t'*)
using *assms*
proof (*induct r arbitrary*: *YY f*)
case *0*
then show *?case*
by (*auto simp add*: *part-fn-def card-eq-0-iff cong*: *conj-cong*)

```
next
  case (Suc r)
  show ?case
  proof −
    from Suc.prems infinite-imp-nonempty obtain yy where yy: yy ∈ YY
      by blast
    let ?ramr = {((y, Y, t), (y', Y', t')). y' ∈ Y ∧ Y' ⊆ Y}
    let ?propr = λ(y, Y, t).
                 y ∈ YY ∧ y ∉ Y ∧ Y ⊆ YY ∧ infinite Y ∧ t < s
                 ∧ (∀X. X⊆Y ∧ finite X ∧ card X = r ⟶ (f ∘ insert y) X = t)
    from Suc.prems have infYY': infinite (YY − {yy}) by auto
    from Suc.prems have partf': part-fn r s (YY − {yy}) (f ∘ insert yy)
      by (simp add: o-def part-fn-Suc-imp-part-fn yy)
    have transr: trans ?ramr by (force simp: trans-def)
    from Suc.hyps [OF infYY' partf']
    obtain Y0 and t0 where Y0 ⊆ YY − {yy} infinite Y0 t0 < s
      X ⊆ Y0 ∧ finite X ∧ card X = r ⟶ (f ∘ insert yy) X = t0 for X
      by blast
    with yy have propr0: ?propr(yy, Y0, t0) by blast
    have proprstep: ∃y. ?propr y ∧ (x, y) ∈ ?ramr if x: ?propr x for x
    proof (cases x)
      case (fields yx Yx tx)
      with x obtain yx' where yx': yx' ∈ Yx
        by (blast dest: infinite-imp-nonempty)
      from fields x have infYx': infinite (Yx − {yx'}) by auto
     with fields x yx' Suc.prems have partfx': part-fn r s (Yx − {yx'}) (f ∘ insert
yx')
       by (simp add: o-def part-fn-Suc-imp-part-fn part-fn-subset [where YY=YY
and Y=Yx])
      from Suc.hyps [OF infYx' partfx'] obtain Y' and t'
        where Y': Y' ⊆ Yx − {yx'} infinite Y' t' < s
          X ⊆ Y' ∧ finite X ∧ card X = r ⟶ (f ∘ insert yx') X = t' for X
        by blast
      from fields x Y' yx' have ?propr (yx', Y', t') ∧ (x, (yx', Y', t')) ∈ ?ramr
        by blast
      then show ?thesis ..
    qed
    from dependent-choice [OF transr propr0 proprstep]
    obtain g where pg: ?propr (g n) and rg: n < m ⟹ (g n, g m) ∈ ?ramr for
n m :: nat
      by blast
    let ?gy = fst ∘ g
    let ?gt = snd ∘ snd ∘ g
    have rangeg: ∃k. range ?gt ⊆ {..<k}
    proof (intro exI subsetI)
      fix x
      assume x ∈ range ?gt
      then obtain n where x = ?gt n ..
      with pg [of n] show x ∈ {..<s} by (cases g n) auto
```

```
    qed
    from rangeg have finite (range ?gt)
      by (simp add: finite-nat-iff-bounded)
    then obtain s' and n' where s': s' = ?gt n' and infeqs': infinite {n. ?gt n =
s'}
      by (rule inf-img-fin-domE) (auto simp add: vimage-def intro: infinite-UNIV-nat)
    with pg [of n'] have less': s'<s by (cases g n') auto
    have inj-gy: inj ?gy
    proof (rule linorder-injI)
      fix m m' :: nat
      assume m < m'
      from rg [OF this] pg [of m] show ?gy m ≠ ?gy m'
        by (cases g m, cases g m') auto
    qed
    show ?thesis
    proof (intro exI conjI)
      from pg show ?gy ` {n. ?gt n = s'} ⊆ YY
        by (auto simp add: Let-def split-beta)
      from infeqs' show infinite (?gy ` {n. ?gt n = s'})
        by (blast intro: inj-gy [THEN inj-on-subset] dest: finite-imageD)
      show s' < s by (rule less')
      show ∀ X. X ⊆ ?gy ` {n. ?gt n = s'} ∧ finite X ∧ card X = Suc r ⟶ f X
= s'
      proof −
        have f X = s'
          if X: X ⊆ ?gy ` {n. ?gt n = s'}
          and cardX: finite X card X = Suc r
          for X
        proof −
          from X obtain AA where AA: AA ⊆ {n. ?gt n = s'} and Xeq: X =
?gy`AA
            by (auto simp add: subset-image-iff)
          with cardX have AA ≠ {} by auto
          then have AAleast: (LEAST x. x ∈ AA) ∈ AA by (auto intro: LeastI-ex)
          show ?thesis
          proof (cases g (LEAST x. x ∈ AA))
            case (fields ya Ya ta)
            with AAleast Xeq have ya: ya ∈ X by (force intro!: rev-image-eqI)
            then have f X = f (insert ya (X − {ya})) by (simp add: insert-absorb)
            also have ... = ta
            proof −
              have ∗: X − {ya} ⊆ Ya
              proof
                fix x assume x: x ∈ X − {ya}
                then obtain a' where xeq: x = ?gy a' and a': a' ∈ AA
                  by (auto simp add: Xeq)
                with fields x have a' ≠ (LEAST x. x ∈ AA) by auto
                with Least-le [of λx. x ∈ AA, OF a'] have (LEAST x. x ∈ AA) < a'
                  by arith
```

```
            from xeq fields rg [OF this] show x ∈ Ya by auto
          qed
          have card (X − {ya}) = r
            by (simp add: cardX ya)
          with pg [of LEAST x. x ∈ AA] fields cardX ∗ show ?thesis
            by (auto simp del: insert-Diff-single)
        qed
        also from AA AAleast fields have ... = s' by auto
        finally show ?thesis .
      qed
    qed
    then show ?thesis by blast
  qed
 qed
qed


theorem Ramsey:
 fixes s r :: nat
   and Z :: 'a set
   and f :: 'a set ⇒ nat
 shows
  ⟦infinite Z;
    ∀ X. X ⊆ Z ∧ finite X ∧ card X = r ⟶ f X < s⟧
  ⟹ ∃ Y t. Y ⊆ Z ∧ infinite Y ∧ t < s
        ∧ (∀ X. X ⊆ Y ∧ finite X ∧ card X = r ⟶ f X = t)
 by (blast intro: Ramsey-induction [unfolded part-fn-def nsets-def])

corollary Ramsey2:
 fixes s :: nat
   and Z :: 'a set
   and f :: 'a set ⇒ nat
 assumes infZ: infinite Z
   and part: ∀ x∈Z. ∀ y∈Z. x ≠ y ⟶ f {x, y} < s
 shows ∃ Y t. Y ⊆ Z ∧ infinite Y ∧ t < s ∧ (∀ x∈Y. ∀ y∈Y. x≠y ⟶ f {x, y}
= t)
proof −
 from part have part2: ∀ X. X ⊆ Z ∧ finite X ∧ card X = 2 ⟶ f X < s
   by (fastforce simp: eval-nat-numeral card-Suc-eq)
 obtain Y t where ∗:
   Y ⊆ Z infinite Y t < s (∀ X. X ⊆ Y ∧ finite X ∧ card X = 2 ⟶ f X = t)
   by (insert Ramsey [OF infZ part2]) auto
 then have ∀ x∈Y. ∀ y∈Y. x ≠ y ⟶ f {x, y} = t by auto
 with ∗ show ?thesis by iprover
qed

corollary Ramsey-nsets:
 fixes f :: 'a set ⇒ nat
```

assumes *infinite Z f ' nsets Z r* ⊆ {..<*s*}
obtains *Y t* **where** *Y* ⊆ *Z infinite Y t* < *s f ' nsets Y r* ⊆ {*t*}
using *Ramsey* [*of Z r f s*] *assms* **by** (*auto simp*: *nsets-def image-subset-iff*)

Every infinite sequence of linearly ordered elements has an infinite monotone subsequence:

corollary *monotone-subsequence*:
fixes *f* :: *nat* ⇒ *'a*::*linorder*
shows ∃ *I*::*nat set*. *infinite I* ∧
((∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f i* ≤ *f j*) ∨ (∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f i* > *f j*))
proof −
let *?f* = λ*M*. *if* ∃ *i j*. *M* = {*i*,*j*} ∧ *i* < *j* ∧ *f i* ≤ *f j then 0 else 1* :: *nat*
have *?f i* < *2* **for** *i* **by** *auto*
with *Ramsey2*[*OF infinite-UNIV-nat*, *of ?f 2*]
obtain *I t* **where** *It*:
infinite I t < *2* ∀ *x*∈*I*. ∀ *y*∈*I*. *x* ≠ *y* ⟶ *?f* {*x*,*y*} = *t* **by** *auto*
show *?thesis*
proof (*cases t* = *0*)
case *True*
hence ∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f i* ≤ *f j* **using** *It*(*3*)
by (*metis doubleton-eq-iff not-one-le-zero order-less-imp-not-less order-refl*)
then show *?thesis* **using** *It*(*1*) **by** *blast*
next
case *False*
hence ⋀*i j*. ⟦ *i* ∈ *I*; *j* ∈ *I*; *i* < *j* ⟧ ⟹ *f i* > *f j*
by (*metis It*(*3*) *linorder-not-less nat-neq-iff*)
then show *?thesis* **by** (*metis It*(*1*))
qed
qed

Every infinite sequence of *nat*s has an infinite monotonically increasing subsequence:

corollary *mono-incr-subsequence-nat*:
fixes *f* :: *nat* ⇒ *nat*
shows ∃ *I*::*nat set*. *infinite I* ∧ (∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f i* ≤ *f j*)
proof −
from *monotone-subsequence*[*of f*]
obtain *I* **where** *I*: *infinite I*
(∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f i* ≤ *f j*) ∨ (∀ *i*∈*I*. ∀ *j*∈*I*. *i* < *j* ⟶ *f j* < *f i*) (**is** *?A* ∨ *?B*)
by *auto*
from *I*(*2*) **show** *?thesis*
proof
assume *?A* **thus** *?thesis* **using** *I*(*1*) **by** *blast*
next
assume *?B*
hence *False*
using *no-infinite-less-nat-down-chain*[*of f o enumerate I*]
by (*simp add*: *I*(*1*) *enumerate-in-set*)

then show *?thesis* **..**
qed
qed

91.5 Disjunctive Well-Foundedness

An application of Ramsey's theorem to program termination. See [4].

definition *disj-wf* :: $('a \times 'a)$ *set* $\Rightarrow$ *bool*
where *disj-wf r* $\longleftrightarrow$ $(\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf\ (T\ i)) \wedge r = (\bigcup i{<}n.\ T\ i))$

definition *transition-idx* :: $(nat \Rightarrow 'a) \Rightarrow (nat \Rightarrow ('a \times 'a)\ set) \Rightarrow nat\ set \Rightarrow nat$
where *transition-idx s T A* = $(LEAST\ k.\ \exists i\ j.\ A = \{i, j\} \wedge i < j \wedge (s\ j, s\ i) \in T\ k)$

lemma *transition-idx-less*:
assumes $i < j$ $(s\ j, s\ i) \in T\ k$ $k < n$
shows *transition-idx s T* $\{i, j\} < n$
proof −
from *assms*(*1*,*2*) **have** *transition-idx s T* $\{i, j\} \leq k$
by (*simp add*: *transition-idx-def*, *blast intro*: *Least-le*)
with *assms*(*3*) **show** *?thesis* **by** *simp*
qed

lemma *transition-idx-in*:
assumes $i < j$ $(s\ j, s\ i) \in T\ k$
shows $(s\ j, s\ i) \in T$ (*transition-idx s T* $\{i, j\}$)
using *assms*
by (*simp add*: *transition-idx-def doubleton-eq-iff conj-disj-distribR cong*: *conj-cong*) (*erule LeastI*)

To be equal to the union of some well-founded relations is equivalent to being the subset of such a union.

lemma *disj-wf*: *disj-wf r* $\longleftrightarrow$ $(\exists T.\ \exists n{::}nat.\ (\forall i{<}n.\ wf(T\ i)) \wedge r \subseteq (\bigcup i{<}n.\ T\ i))$
proof −
have $*$: $\bigwedge T\ n.\ [\![\forall i{<}n.\ wf\ (T\ i); r \subseteq \bigcup\ (T\ `\ \{..{<}n\})]\!]$
$\Longrightarrow (\forall i{<}n.\ wf\ (T\ i \cap r)) \wedge r = (\bigcup i{<}n.\ T\ i \cap r)$
by (*force simp*: *wf-Int1*)
show *?thesis*
unfolding *disj-wf-def* **by** *auto* (*metis* $*$)
qed

theorem *trans-disj-wf-implies-wf*:
assumes *trans r*
and *disj-wf r*
shows *wf r*
proof (*simp only*: *wf-iff-no-infinite-down-chain*, *rule notI*)
assume $\exists s.\ \forall i.\ (s\ (Suc\ i), s\ i) \in r$

then obtain s **where** *sSuc*: $\forall i.\ (s\ (Suc\ i),\ s\ i) \in r$ **..**
from ‹*disj-wf r*› **obtain** T **and** $n :: nat$ **where** *wfT*: $\forall k{<}n.\ wf(T\ k)$ **and** r: $r = (\bigcup k{<}n.\ T\ k)$
by (*auto simp add*: *disj-wf-def*)
have *s-in-T*: $\exists k.\ (s\ j,\ s\ i) \in T\ k \wedge k{<}n$ **if** $i < j$ **for** $i\ j$
proof −
from ‹$i < j$› **have** $(s\ j,\ s\ i) \in r$
proof (*induct rule*: *less-Suc-induct*)
case *1*
then show *?case* **by** (*simp add*: *sSuc*)
next
case *2*
with ‹*trans r*› **show** *?case*
unfolding *trans-def* **by** *blast*
qed
then show *?thesis* **by** (*auto simp add*: *r*)
qed
have $i < j \Longrightarrow$ *transition-idx s T* $\{i, j\} < n$ **for** $i\ j$
using *s-in-T transition-idx-less* **by** *blast*
then have *trless*: $i \neq j \Longrightarrow$ *transition-idx s T* $\{i, j\} < n$ **for** $i\ j$
by (*metis doubleton-eq-iff less-linear*)
have $\exists K\ k.\ K \subseteq UNIV \wedge$ *infinite* $K \wedge k < n \wedge$
$(\forall i{\in}K.\ \forall j{\in}K.\ i \neq j \longrightarrow$ *transition-idx s T* $\{i, j\} = k)$
by (*rule Ramsey2*) (*auto intro*: *trless infinite-UNIV-nat*)
then obtain K **and** k **where** *infK*: *infinite* K **and** $k < n$
and *allk*: $\forall i{\in}K.\ \forall j{\in}K.\ i \neq j \longrightarrow$ *transition-idx s T* $\{i, j\} = k$
by *auto*
have $(s\ (enumerate\ K\ (Suc\ m)),\ s\ (enumerate\ K\ m)) \in T\ k$ **for** $m :: nat$
proof −
let *?j* = *enumerate K* (*Suc m*)
let *?i* = *enumerate K m*
have *ij*: *?i* < *?j* **by** (*simp add*: *enumerate-step infK*)
have *?j* ∈ *K ?i* ∈ *K* **by** (*simp-all add*: *enumerate-in-set infK*)
with *ij* **have** *k*: *k* = *transition-idx s T* {*?i*, *?j*} **by** (*simp add*: *allk*)
from *s-in-T* [*OF ij*] **obtain** k' **where** $(s\ ?j,\ s\ ?i) \in T\ k'\ k'{<}n$ **by** *blast*
then show $(s\ ?j,\ s\ ?i) \in T\ k$ **by** (*simp add*: *k transition-idx-in ij*)
qed
then have ¬ *wf* (*T k*)
unfolding *wf-iff-no-infinite-down-chain* **by** *iprover*
with *wfT* ‹$k < n$› **show** *False* **by** *blast*
qed

end

92 Modulo and congruence on the reals

theory *Real-Mod*
imports *Complex-Main*
begin

definition *rmod* :: *real* ⇒ *real* ⇒ *real* (**infixl** ‹*rmod*› *70*) **where**
x rmod y = *x* − |*y*| ∗ *of-int* ⌊*x* / |*y*|⌋

lemma *rmod-conv-frac*: *y* ≠ *0* ⟹ *x rmod y* = *frac* (*x* / |*y*|) ∗ |*y*|
by (*simp add*: *rmod-def frac-def algebra-simps*)

lemma *rmod-conv-frac′*: *x rmod y* = (*if y* = *0 then x else frac* (*x* / |*y*|) ∗ |*y*|)
by (*simp add*: *rmod-def frac-def algebra-simps*)

lemma *rmod-rmod* [*simp*]: (*x rmod y*) *rmod y* = *x rmod y*
by (*simp add*: *rmod-conv-frac′*)

lemma *rmod-0-right* [*simp*]: *x rmod 0* = *x*
by (*simp add*: *rmod-def*)

lemma *rmod-less*: *m* > *0* ⟹ *x rmod m* < *m*
by (*simp add*: *rmod-conv-frac′ frac-lt-1*)

lemma *rmod-less-abs*: *m* ≠ *0* ⟹ *x rmod m* < |*m*|
by (*simp add*: *rmod-conv-frac′ frac-lt-1*)

lemma *rmod-le*: *m* > *0* ⟹ *x rmod m* ≤ *m*
by (*intro less-imp-le rmod-less*)

lemma *rmod-nonneg*: *m* ≠ *0* ⟹ *x rmod m* ≥ *0*
unfolding *rmod-def*
by (*metis abs-le-zero-iff diff-ge-0-iff-ge floor-divide-lower linorder-not-le mult.commute*)

lemma *rmod-unique*:
assumes *z* ∈ {*0*..<|*y*|} *x* = *z* + *of-int n* ∗ *y*
shows *x rmod y* = *z*
proof −
have (*x* − *z*) / *y* = *of-int n*
using *assms* **by** *auto*
hence (*x* − *z*) / |*y*| = *of-int* ((*if y* > *0 then 1 else* −*1*) ∗ *n*)
using *assms*(*1*) **by** (*cases y 0* :: *real rule*: *linorder-cases*) (*auto split*: *if-splits*)
also have ... ∈ ℤ
by *auto*
finally have *frac* (*x* / |*y*|) = *z* / |*y*|
using *assms*(*1*) **by** (*subst frac-unique-iff*) (*auto simp*: *field-simps*)
thus *?thesis*
using *assms*(*1*) **by** (*auto simp*: *rmod-conv-frac′*)
qed

lemma *rmod-0* [*simp*]: *0 rmod z* = *0*
by (*simp add*: *rmod-def*)

lemma *rmod-add*: $(x \ rmod \ z + y \ rmod \ z) \ rmod \ z = (x + y) \ rmod \ z$
proof (*cases* $z = 0$)
  case [*simp*]: *False*
  show *?thesis*
  proof (*rule sym*, *rule rmod-unique*)
    define n **where** $n = (\textit{if } z > 0 \textit{ then } 1 \textit{ else } -1) * (\lfloor x \ / \ |z| \rfloor + \lfloor y \ / \ |z| \rfloor + \lfloor (x + y - (|z| * \textit{real-of-int } \lfloor x \ / \ |z| \rfloor + |z| * \textit{real-of-int } \lfloor y \ / \ |z| \rfloor)) \ / \ |z| \rfloor)$
    show $x + y = (x \ rmod \ z + y \ rmod \ z) \ rmod \ z + \textit{real-of-int } n * z$
      by (*simp add*: *rmod-def algebra-simps n-def*)
  qed (*auto simp*: *rmod-less-abs rmod-nonneg*)
qed *auto*

lemma *rmod-diff*: $(x \ rmod \ z - y \ rmod \ z) \ rmod \ z = (x - y) \ rmod \ z$
proof (*cases* $z = 0$)
  case [*simp*]: *False*
  show *?thesis*
  proof (*rule sym*, *rule rmod-unique*)
    define n **where** $n = (\textit{if } z > 0 \textit{ then } 1 \textit{ else } -1) * (\lfloor x \ / \ |z| \rfloor + \lfloor (x + |z| * \textit{real-of-int } \lfloor y \ / \ |z| \rfloor - (y + |z| * \textit{real-of-int } \lfloor x \ / \ |z| \rfloor)) \ / \ |z| \rfloor - \lfloor y \ / \ |z| \rfloor)$
    show $x - y = (x \ rmod \ z - y \ rmod \ z) \ rmod \ z + \textit{real-of-int } n * z$
      by (*simp add*: *rmod-def algebra-simps n-def*)
  qed (*auto simp*: *rmod-less-abs rmod-nonneg*)
qed *auto*

lemma *rmod-self* [*simp*]: $x \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-int* [*simp*]: $(\textit{of-int } n * x) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-nat* [*simp*]: $(\textit{of-nat } n * x) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-numeral* [*simp*]: $(\textit{numeral } n * x) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-int'* [*simp*]: $(x * \textit{of-int } n) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-nat'* [*simp*]: $(x * \textit{of-nat } n) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-self-multiple-numeral'* [*simp*]: $(x * \textit{numeral } n) \ rmod \ x = 0$
  by (*cases x 0* :: *real rule*: *linorder-cases*) (*auto simp*: *rmod-conv-frac*)

lemma *rmod-idem* [*simp*]: $x \in \{0..<|y|\} \Longrightarrow x \ rmod \ y = x$
  by (*rule rmod-unique*[*of - - - 0*]) *auto*

definition *rcong* :: *real* ⇒ *real* ⇒ *real* ⇒ *bool*
(‹(‹*indent=1 notation=‹mixfix rcong›*›[- = -] ′(′ *rmod* -′)›)
where [*x* = *y*] (*rmod m*) ⟷ *x rmod m* = *y rmod m*

named-theorems *rcong-intros*

lemma *rcong-0-right* [*simp*]: [*x* = *y*] (*rmod 0*) ⟷ *x* = *y*
by (*simp add*: *rcong-def*)

lemma *rcong-0-iff*: [*x* = *0*] (*rmod m*) ⟷ *x rmod m* = *0*
and *rcong-0-iff′*: [*0* = *x*] (*rmod m*) ⟷ *x rmod m* = *0*
by (*simp-all add*: *rcong-def*)

lemma *rcong-refl* [*simp*, *intro*!, *rcong-intros*]: [*x* = *x*] (*rmod m*)
by (*simp add*: *rcong-def*)

lemma *rcong-sym*: [*y* = *x*] (*rmod m*) ⟹ [*x* = *y*] (*rmod m*)
by (*simp add*: *rcong-def*)

lemma *rcong-sym-iff*: [*y* = *x*] (*rmod m*) ⟷ [*x* = *y*] (*rmod m*)
unfolding *rcong-def* **by** (*simp add*: *eq-commute del*: *rmod-idem*)

lemma *rcong-trans* [*trans*]: [*x* = *y*] (*rmod m*) ⟹ [*y* = *z*] (*rmod m*) ⟹ [*x* = *z*] (*rmod m*)
by (*simp add*: *rcong-def*)

lemma *rcong-add* [*rcong-intros*]:
[*a* = *b*] (*rmod m*) ⟹ [*c* = *d*] (*rmod m*) ⟹ [*a* + *c* = *b* + *d*] (*rmod m*)
unfolding *rcong-def* **using** *rmod-add* **by** *metis*

lemma *rcong-diff* [*rcong-intros*]:
[*a* = *b*] (*rmod m*) ⟹ [*c* = *d*] (*rmod m*) ⟹ [*a* − *c* = *b* − *d*] (*rmod m*)
unfolding *rcong-def* **using** *rmod-diff* **by** *metis*

lemma *rcong-uminus* [*rcong-intros*]:
[*a* = *b*] (*rmod m*) ⟹ [−*a* = −*b*] (*rmod m*)
using *rcong-diff*[*of 0 0 m a b*] **by** *simp*

lemma *rcong-uminus-uminus-iff* [*simp*]: [−*x* = −*y*] (*rmod m*) ⟷ [*x* = *y*] (*rmod m*)
using *rcong-uminus minus-minus* **by** *metis*

lemma *rcong-uminus-left-iff*: [−*x* = *y*] (*rmod m*) ⟷ [*x* = −*y*] (*rmod m*)
using *rcong-uminus minus-minus* **by** *metis*

lemma *rcong-add-right-cancel* [*simp*]: [*a* + *c* = *b* + *c*] (*rmod m*) ⟷ [*a* = *b*] (*rmod*

m)
using *rcong-add*[*of a b m c c*] *rcong-add*[*of a + c b + c m −c −c*] **by** *auto*

lemma *rcong-add-left-cancel* [*simp*]: [*c + a = c + b*] (*rmod m*) ⟷ [*a = b*] (*rmod m*)
by (*subst* (*1 2*) *add.commute*) *simp*

lemma *rcong-diff-right-cancel* [*simp*]: [*a − c = b − c*] (*rmod m*) ⟷ [*a = b*] (*rmod m*)
by (*metis rcong-add-left-cancel uminus-add-conv-diff*)

lemma *rcong-diff-left-cancel* [*simp*]: [*c − a = c − b*] (*rmod m*) ⟷ [*a = b*] (*rmod m*)
by (*metis minus-diff-eq rcong-diff-right-cancel rcong-uminus-uminus-iff*)

lemma *rcong-rmod-right-iff* [*simp*]: [*a = (b rmod m)*] (*rmod m*) ⟷ [*a = b*] (*rmod m*)
and *rcong-rmod-left-iff* [*simp*]: [(*a rmod m*) *= b*] (*rmod m*) ⟷ [*a = b*] (*rmod m*)
by (*simp-all add*: *rcong-def*)

lemma *rcong-rmod-left* [*rcong-intros*]: [*a = b*] (*rmod m*) ⟹ [(*a rmod m*) *= b*] (*rmod m*)
and *rcong-rmod-right* [*rcong-intros*]: [*a = b*] (*rmod m*) ⟹ [*a = (b rmod m)*] (*rmod m*)
by *simp-all*

lemma *rcong-mult-of-int-0-left-left* [*rcong-intros*]: [*0 = of-int n ∗ m*] (*rmod m*)
and *rcong-mult-of-int-0-right-left* [*rcong-intros*]: [*0 = m ∗ of-int n*] (*rmod m*)
and *rcong-mult-of-int-0-left-right* [*rcong-intros*]: [*of-int n ∗ m = 0*] (*rmod m*)
and *rcong-mult-of-int-0-right-right* [*rcong-intros*]: [*m ∗ of-int n = 0*] (*rmod m*)
by (*simp-all add*: *rcong-def*)

lemma *rcong-altdef*: [*a = b*] (*rmod m*) ⟷ (∃ *n*. *b = a + of-int n ∗ m*)
proof (*cases m = 0*)
case *False*
show *?thesis*
proof
assume [*a = b*] (*rmod m*)
hence [*a − b = b − b*] (*rmod m*)
by (*intro rcong-intros*)
hence (*a − b*) *rmod m = 0*
by (*simp add*: *rcong-def*)
then obtain *n* **where** *of-int n* = (*a − b*) / |*m*|
using *False* **by** (*auto simp*: *rmod-conv-frac elim*!: *Ints-cases*)
thus ∃ *n*. *b = a + of-int n ∗ m* **using** *False*
by (*intro exI*[*of - if m > 0 then −n else n*]) (*auto simp*: *field-simps*)
next
assume ∃ *n*. *b = a + of-int n ∗ m*

```
    then obtain n where n: b = a + of-int n * m
      by auto
    have [a + 0 = a + of-int n * m] (rmod m)
      by (intro rcong-intros)
    with n show [a = b] (rmod m)
      by simp
  qed
qed auto

lemma rcong-conv-diff-rmod-eq-0: [x = y] (rmod m) ⟷ (x - y) rmod m = 0
  by (metis cancel-comm-monoid-add-class.diff-cancel rcong-0-iff rcong-diff-right-cancel)

lemma rcong-imp-eq:
  assumes [x = y] (rmod m) |x - y| < |m|
  shows   x = y
proof -
  from assms obtain n where n: y = x + of-int n * m
    unfolding rcong-altdef by blast
  have of-int |n| * |m| = |x - y|
    by (simp add: n abs-mult)
  also have ... < 1 * |m|
    using assms(2) by simp
  finally have n = 0
    by (subst (asm) mult-less-cancel-right) auto
  with n show ?thesis
    by simp
qed

lemma rcong-mult-modulus:
  assumes [a = b] (rmod (m / c)) c ≠ 0
  shows   [a * c = b * c] (rmod m)
proof -
  from assms obtain k where k: b = a + of-int k * (m / c)
    by (auto simp: rcong-altdef)
  have b * c = (a + of-int k * (m / c)) * c
    by (simp only: k)
  also have ... = a * c + of-int k * m
    using assms(2) by (auto simp: divide-simps)
  finally show ?thesis
    unfolding rcong-altdef by blast
qed

lemma rcong-divide-modulus:
  assumes [a = b] (rmod (m * c)) c ≠ 0
  shows   [a / c = b / c] (rmod m)
  using rcong-mult-modulus[of a b m 1 / c] assms by (auto simp: field-simps)

lemma sin-rmod [simp]: sin (x rmod (2*pi)) = sin x
  and cos-rmod [simp]: cos (x rmod (2*pi)) = cos x
```

by (*simp-all add*: *rmod-def sin-diff cos-diff*)

lemma *tan-rmod* [*simp*]: *tan* (*x rmod* (*2*∗*pi*)) = *tan x*
and *cot-rmod* [*simp*]: *cot* (*x rmod* (*2*∗*pi*)) = *cot x*
and *cis-rmod* [*simp*]: *cis* (*x rmod* (*2*∗*pi*)) = *cis x*
and *rcis-rmod* [*simp*]: *rcis r* (*x rmod* (*2*∗*pi*)) = *rcis r x*
by (*simp-all add*: *tan-def cot-def complex-eq-iff*)

lemma *cis-eq-iff*: *cis a* = *cis b* ⟷ [*a* = *b*] (*rmod* (*2* ∗ *pi*))
proof −
have *cis a* = *cis b* ⟷ *sin a* = *sin b* ∧ *cos a* = *cos b*
by (*auto simp*: *complex-eq-iff*)
also have ... ⟷ (∃ *x*. *a* = *b* + *2* ∗ *pi* ∗ *real-of-int x*)
by (*rule sin-cos-eq-iff*)
also have ... ⟷ [*b* = *a*] (*rmod* (*2* ∗ *pi*))
by (*simp add*: *rcong-altdef mult-ac*)
finally show *?thesis*
by (*simp add*: *rcong-sym-iff*)
qed

lemma *cis-eq-1-iff*: *cis a* = *1* ⟷ (∃ *n*. *a* = *of-int n* ∗ (*2* ∗ *pi*))
proof −
have *cis 0* = *cis a* ⟷ [*0* = *a*] (*rmod* (*2* ∗ *pi*))
by (*rule cis-eq-iff*)
also have ... ⟷ (∃ *n*. *a* = *of-int n* ∗ (*2* ∗ *pi*))
unfolding *rcong-altdef* **by** *simp*
finally show *?thesis*
by *simp*
qed

lemma *cis-cong*:
assumes [*a* = *b*] (*rmod 2* ∗ *pi*)
shows *cis a* = *cis b*
using *cis-eq-iff assms* **by** *blast*

lemma *rcis-cong*:
assumes [*a* = *b*] (*rmod 2* ∗ *pi*)
shows *rcis r a* = *rcis r b*
using *assms* **by** (*auto simp*: *rcis-def intro*!: *cis-cong*)

lemma *sin-rcong*: [*x* = *y*] (*rmod* (*2* ∗ *pi*)) ⟹ *sin x* = *sin y*
and *cos-rcong*: [*x* = *y*] (*rmod* (*2* ∗ *pi*)) ⟹ *cos x* = *cos y*
using *cis-cong*[*of x y*] **by** (*simp-all add*: *complex-eq-iff*)

lemma *sin-eq-sin-conv-rmod*:
assumes *sin x* = *sin y*
shows [*x* = *y*] (*rmod 2* ∗ *pi*) ∨ [*x* = *pi* − *y*] (*rmod 2* ∗ *pi*)
proof −
have *0* = *sin y* − *sin x*

```
    using assms by simp
  hence sin ((y − x) / 2) = 0 ∨ cos ((y + x) / 2) = 0
    unfolding sin-diff-sin by simp
  hence (∃ i. y = x + real-of-int i ∗ (2 ∗ pi)) ∨
        (∃ i. pi − y = x + real-of-int (−i) ∗ (2 ∗ pi))
    unfolding sin-zero-iff-int2 cos-zero-iff-int2
    by (auto simp: algebra-simps)
  thus ?thesis
    unfolding rcong-altdef by blast
qed

lemma cos-eq-cos-conv-rmod:
  assumes cos x = cos y
  shows   [x = y] (rmod 2 ∗ pi) ∨ [x = −y] (rmod 2 ∗ pi)
proof −
  have 0 = cos y − cos x
    using assms by simp
  hence sin ((y + x) / 2) = 0 ∨ sin ((x − y) / 2) = 0
    unfolding cos-diff-cos by simp
  hence (∃ i. −y = x + real-of-int (−i) ∗ (2 ∗ pi)) ∨
        (∃ i. y = x + real-of-int (−i) ∗ (2 ∗ pi))
    unfolding sin-zero-iff-int2
    by (auto simp: algebra-simps)
  thus ?thesis
    unfolding rcong-altdef by blast
qed

lemma sin-eq-sin-conv-rmod-iff:
  sin x = sin y ⟷ [x = y] (rmod 2 ∗ pi) ∨ [x = pi − y] (rmod 2 ∗ pi)
  by (metis sin-eq-sin-conv-rmod sin-pi-minus sin-rcong)

lemma cos-eq-cos-conv-rmod-iff:
  cos x = cos y ⟷ [x = y] (rmod 2 ∗ pi) ∨ [x = −y] (rmod 2 ∗ pi)
  by (metis cos-eq-cos-conv-rmod cos-minus cos-rcong)

end
```

93 Generic reflection and reification

```
theory Reflection
imports Main
begin

ML-file ‹~~/src/HOL/Tools/reflection.ML›

method-setup reify = ‹
  Attrib.thms −−
    Scan.option (Scan.lift (Args.$$$ () |−− Args.term −−| Scan.lift (Args.$$$ )))
>>
```

```
    (fn (user-eqs, to) => fn ctxt => SIMPLE-METHOD' (Reflection.default-reify-tac
ctxt user-eqs to))
› partial automatic reification

method-setup reflection = ‹
let
  fun keyword k = Scan.lift (Args.$$$ k -- Args.colon) >> K ();
  val onlyN = only;
  val rulesN = rules;
  val any-keyword = keyword onlyN || keyword rulesN;
  val thms = Scan.repeats (Scan.unless any-keyword Attrib.multi-thm);
  val terms = thms >> map (Thm.term-of o Drule.dest-term);
in
  thms -- Scan.optional (keyword rulesN |-- thms) [] --
    Scan.option (keyword onlyN |-- Args.term) >>
  (fn ((user-eqs, user-thms), to) => fn ctxt =>
      SIMPLE-METHOD' (Reflection.default-reflection-tac ctxt user-thms user-eqs
to))
end
› partial automatic reflection

end


theory Rewrite
imports Main
begin

consts rewrite-HOLE :: 'a::{}  (‹¤›)

lemma eta-expand:
  fixes f :: 'a::{} ⇒ 'b::{}
  shows f ≡ λx. f x .

lemma imp-cong-eq:
  (PROP A ⟹ (PROP B ⟹ PROP C) ≡ (PROP B' ⟹ PROP C')) ≡
    ((PROP B ⟹ PROP A ⟹ PROP C) ≡ (PROP B' ⟹ PROP A ⟹ PROP
C'))
  apply (intro Pure.equal-intr-rule)
      apply (drule (1) cut-rl; drule Pure.equal-elim-rule1 Pure.equal-elim-rule2;
assumption)+
   apply (drule Pure.equal-elim-rule1 Pure.equal-elim-rule2; assumption)+
  done

ML-file ‹cconv.ML›
ML-file ‹rewrite.ML›

end
```

94 Assigning lengths to types by type classes

theory *Type-Length*
imports *Numeral-Type*
begin

The aim of this is to allow any type as index type, but to provide a default instantiation for numeral types. This independence requires some duplication with the definitions in `Numeral_Type.thy`.

class *len0* =
  fixes *len-of* :: *'a itself* ⇒ *nat*

syntax *-type-length* :: *type* ⇒ *nat* (‹(*1LENGTH/(1'(-'))*)›)
syntax-consts *-type-length* ⇌ *len-of*
translations *LENGTH('a)* ↽ *CONST len-of TYPE('a)*
print-translation ‹
  let
    fun len-of-itself-tr' ctxt [*Const* (***const-syntax*** ‹*Pure.type*›, *Type* (-, [*T*]))] =
      Syntax.const ***syntax-const*** ‹*-type-length*› $ *Syntax-Phases.term-of-typ ctxt T*
  in [(***const-syntax*** ‹*len-of*›, *len-of-itself-tr'*)] *end*
›

Some theorems are only true on words with length greater 0.

class *len* = *len0* +
  assumes *len-gt-0* [*iff*]: *0* < *LENGTH('a)*
begin

lemma *len-not-eq-0* [*simp*]:
  LENGTH('a) ≠ *0*
  by *simp*

end

instantiation *num0* **and** *num1* :: *len0*
begin

definition *len-num0*: *len-of* (- :: *num0 itself*) = *0*
definition *len-num1*: *len-of* (- :: *num1 itself*) = *1*

instance ..

end

instantiation *bit0* **and** *bit1* :: (*len0*) *len0*
begin

definition *len-bit0*: *len-of* (- :: *'a*::*len0 bit0 itself*) = *2* ∗ *LENGTH('a)*
definition *len-bit1*: *len-of* (- :: *'a*::*len0 bit1 itself*) = *2* ∗ *LENGTH('a)* + *1*

instance ..

end

lemmas *len-of-numeral-defs* [*simp*] = *len-num0 len-num1 len-bit0 len-bit1*

instance *num1* :: *len*
  by *standard simp*
instance *bit0* :: (*len*) *len*
  by *standard simp*
instance *bit1* :: (*len0*) *len*
  by *standard simp*

instantiation *Enum.finite-1* :: *len*
begin

definition
  len-of-finite-1 (*x* :: *Enum.finite-1 itself*) ≡ (*1* :: *nat*)

instance
  by *standard* (*auto simp*: *len-of-finite-1-def*)

end

instantiation *Enum.finite-2* :: *len*
begin

definition
  len-of-finite-2 (*x* :: *Enum.finite-2 itself*) ≡ (*2* :: *nat*)

instance
  by *standard* (*auto simp*: *len-of-finite-2-def*)

end

instantiation *Enum.finite-3* :: *len*
begin

definition
  len-of-finite-3 (*x* :: *Enum.finite-3 itself*) ≡ (*4* :: *nat*)

instance
  by *standard* (*auto simp*: *len-of-finite-3-def*)

end

lemma *length-less-eq-Suc-0-iff* [*simp*]:
  ‹*LENGTH*('*a*::*len*) ≤ *Suc 0* ⟷ *LENGTH*('*a*) = *Suc 0*›
  by (*simp add*: *le-Suc-eq*)

lemma *length-not-greater-eq-2-iff* [*simp*]:
‹¬ 2 ≤ *LENGTH*('a::*len*) ⟷ *LENGTH*('a) = *Suc* 0›
by (*auto simp add*: *not-le dest*: *less-2-cases*)

lemma *less-eq-decr-length-iff* [*simp*]:
‹*n* ≤ *LENGTH*('a::*len*) − *Suc* 0 ⟷ *n* < *LENGTH*('a)›
by (*cases* ‹*LENGTH*('a)›) (*simp-all add*: *less-Suc-eq le-less*)

lemma *decr-length-less-iff* [*simp*]:
‹*LENGTH*('a::*len*) − *Suc* 0 < *n* ⟷ *LENGTH*('a) ≤ *n*›
by (*cases* ‹*LENGTH*('a)›) *auto*

context *linordered-idom*
begin

lemma *two-less-eq-exp-length* [*simp*]:
‹2 ≤ 2 ^ *LENGTH*('b::*len*)›
using *mult-left-mono* [*of* 1 ‹2 ^ (*LENGTH*('b::*len*) − 1)› 2]
by (*cases* ‹*LENGTH*('b::*len*)›) *simp-all*

end

end

95 Saturated arithmetic

theory *Saturated*
imports *Numeral-Type Type-Length*
begin

95.1 The type of saturated naturals

typedef (**overloaded**) ('a::*len*) *sat* = {.. *LENGTH*('a)}
morphisms *nat-of Abs-sat*
by *auto*

lemma *sat-eqI*:
nat-of m = *nat-of n* ⟹ *m* = *n*
by (*simp add*: *nat-of-inject*)

lemma *sat-eq-iff*:
m = *n* ⟷ *nat-of m* = *nat-of n*
by (*simp add*: *nat-of-inject*)

lemma *Abs-sat-nat-of* [*code abstype*]:
Abs-sat (*nat-of n*) = *n*
by (*fact nat-of-inverse*)

definition *Abs-sat'* :: *nat* ⇒ *'a*::*len sat* **where**
Abs-sat' n = *Abs-sat* (*min* (*LENGTH*(*'a*)) *n*)

lemma *nat-of-Abs-sat'* [*simp*]:
nat-of (*Abs-sat' n* :: (*'a*::*len*) *sat*) = *min* (*LENGTH*(*'a*)) *n*
unfolding *Abs-sat'-def* **by** (*rule Abs-sat-inverse*) *simp*

lemma *nat-of-le-len-of* [*simp*]:
nat-of (*n* :: (*'a*::*len*) *sat*) ≤ *LENGTH*(*'a*)
using *nat-of* [**where** *x* = *n*] **by** *simp*

lemma *min-len-of-nat-of* [*simp*]:
min (*LENGTH*(*'a*)) (*nat-of* (*n*::(*'a*::*len*) *sat*)) = *nat-of n*
by (*rule min.absorb2* [*OF nat-of-le-len-of*])

lemma *min-nat-of-len-of* [*simp*]:
min (*nat-of* (*n*::(*'a*::*len*) *sat*)) (*LENGTH*(*'a*)) = *nat-of n*
by (*subst min.commute*) *simp*

lemma *Abs-sat'-nat-of* [*simp*]:
Abs-sat' (*nat-of n*) = *n*
by (*simp add*: *Abs-sat'-def nat-of-inverse*)

instantiation *sat* :: (*len*) *linorder*
begin

definition
less-eq-sat-def: *x* ≤ *y* ⟷ *nat-of x* ≤ *nat-of y*

definition
less-sat-def: *x* < *y* ⟷ *nat-of x* < *nat-of y*

instance
by *standard*
(*auto simp add*: *less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1 mult.commute*)

end

instantiation *sat* :: (*len*) {*minus*, *comm-semiring-1*}
begin

definition
0 = *Abs-sat' 0*

definition
1 = *Abs-sat' 1*

lemma *nat-of-zero-sat* [*simp*, *code abstract*]:

nat-of 0 = 0
by (*simp add*: *zero-sat-def*)

lemma *nat-of-one-sat* [*simp*, *code abstract*]:
nat-of 1 = min 1 (*LENGTH*('*a*))
by (*simp add*: *one-sat-def*)

definition
x + y = Abs-sat' (*nat-of x + nat-of y*)

lemma *nat-of-plus-sat* [*simp*, *code abstract*]:
nat-of (*x + y*) = *min* (*nat-of x + nat-of y*) (*LENGTH*('*a*))
by (*simp add*: *plus-sat-def*)

definition
x − y = Abs-sat' (*nat-of x − nat-of y*)

lemma *nat-of-minus-sat* [*simp*, *code abstract*]:
nat-of (*x − y*) = *nat-of x − nat-of y*
proof −
from *nat-of-le-len-of* [*of x*] **have** *nat-of x − nat-of y* ≤ *LENGTH*('*a*) **by** *arith*
then show *?thesis* **by** (*simp add*: *minus-sat-def*)
qed

definition
x ∗ y = Abs-sat' (*nat-of x ∗ nat-of y*)

lemma *nat-of-times-sat* [*simp*, *code abstract*]:
nat-of (*x ∗ y*) = *min* (*nat-of x ∗ nat-of y*) (*LENGTH*('*a*))
by (*simp add*: *times-sat-def*)

instance
proof
fix *a b c* :: '*a*::*len sat*
show *a ∗ b ∗ c = a ∗* (*b ∗ c*)
proof(*cases a = 0*)
case *True* **thus** *?thesis* **by** (*simp add*: *sat-eq-iff*)
next
case *False* **show** *?thesis*
proof(*cases c = 0*)
case *True* **thus** *?thesis* **by** (*simp add*: *sat-eq-iff*)
next
case *False* **with** ‹*a ≠ 0*› **show** *?thesis*
by (*simp add*: *sat-eq-iff nat-mult-min-left nat-mult-min-right mult.assoc min.assoc min.absorb2*)
qed
qed
show *1 ∗ a = a*
by (*simp add*: *sat-eq-iff min-def not-le not-less*)

show $(a + b) * c = a * c + b * c$
proof(*cases* $c = 0$)
case *True* **thus** *?thesis* **by** (*simp add*: *sat-eq-iff*)
next
case *False* **thus** *?thesis*
by (*simp add*: *sat-eq-iff nat-mult-min-left add-mult-distrib min-add-distrib-left min-add-distrib-right min.assoc*)
qed
qed (*simp-all add*: *sat-eq-iff mult.commute*)

end

instantiation *sat* :: (*len*) *ordered-comm-semiring*
begin

instance
by *standard*
(*auto simp add*: *less-eq-sat-def less-sat-def not-le sat-eq-iff min.coboundedI1 mult.commute*)

end

lemma *Abs-sat'-eq-of-nat*: *Abs-sat'* n = *of-nat* n
by (*rule sat-eqI*, *induct n*, *simp-all*)

abbreviation *Sat* :: *nat* ⇒ *'a*::*len sat* **where**
Sat ≡ *of-nat*

lemma *nat-of-Sat* [*simp*]:
nat-of (*Sat n* :: (*'a*::*len*) *sat*) = *min* (*LENGTH*(*'a*)) *n*
by (*rule nat-of-Abs-sat'* [*unfolded Abs-sat'-eq-of-nat*])

lemma [*code-abbrev*]:
of-nat (*numeral k*) = (*numeral k* :: *'a*::*len sat*)
by *simp*

context
begin

qualified definition *sat-of-nat* :: *nat* ⇒ (*'a*::*len*) *sat*
where [*code-abbrev*]: *sat-of-nat* = *of-nat*

lemma [*code abstract*]:
nat-of (*sat-of-nat n* :: (*'a*::*len*) *sat*) = *min* (*LENGTH*(*'a*)) *n*
by (*simp add*: *sat-of-nat-def*)

end

instance *sat* :: (*len*) *finite*

proof
show *finite* (*UNIV*::*'a sat set*)
using *type-definition.Abs-image* [*OF type-definition-sat*]
by (*metis finite-atMost finite-imageI*)
qed

instantiation *sat* :: (*len*) *equal*
begin

definition *HOL.equal A B* ⟷ *nat-of A* = *nat-of B*

instance
by *standard* (*simp add*: *equal-sat-def nat-of-inject*)

end

instantiation *sat* :: (*len*) {*bounded-lattice*, *distrib-lattice*}
begin

definition (*inf* :: *'a sat* ⇒ *'a sat* ⇒ *'a sat*) = *min*
definition (*sup* :: *'a sat* ⇒ *'a sat* ⇒ *'a sat*) = *max*
definition *bot* = (*0* :: *'a sat*)
definition *top* = *Sat* (*LENGTH*(*'a*))

instance
by *standard*
(*simp-all add*: *inf-sat-def sup-sat-def bot-sat-def top-sat-def max-min-distrib2*,
simp-all add: *less-eq-sat-def*)

end

instantiation *sat* :: (*len*) {*Inf*, *Sup*}
begin

global-interpretation *Inf-sat*: *semilattice-neutr-set min* ‹*top* :: *'a sat*›
defines *Inf-sat* = *Inf-sat.F*
by *standard* (*simp add*: *min-def*)

global-interpretation *Sup-sat*: *semilattice-neutr-set max* ‹*bot* :: *'a sat*›
defines *Sup-sat* = *Sup-sat.F*
by *standard* (*simp add*: *max-def bot.extremum-unique*)

instance ..

end

instance *sat* :: (*len*) *complete-lattice*
proof
fix *x* :: *'a sat*

fix $A :: 'a\ sat\ set$
note *finite*
moreover assume $x \in A$
ultimately show $Inf\ A \leq x$
by (*induct A*) (*auto intro*: *min.coboundedI2*)
next
fix $z :: 'a\ sat$
fix $A :: 'a\ sat\ set$
note *finite*
moreover assume z: $\bigwedge x.\ x \in A \Longrightarrow z \leq x$
ultimately show $z \leq Inf\ A$ **by** (*induct A*) *simp-all*
next
fix $x :: 'a\ sat$
fix $A :: 'a\ sat\ set$
note *finite*
moreover assume $x \in A$
ultimately show $x \leq Sup\ A$
by (*induct A*) (*auto intro*: *max.coboundedI2*)
next
fix $z :: 'a\ sat$
fix $A :: 'a\ sat\ set$
note *finite*
moreover assume z: $\bigwedge x.\ x \in A \Longrightarrow x \leq z$
ultimately show $Sup\ A \leq z$ **by** (*induct A*) *auto*
next
show $Inf\ \{\} = (top::'a\ sat)$
by (*auto simp*: *top-sat-def*)
show $Sup\ \{\} = (bot::'a\ sat)$
by (*auto simp*: *bot-sat-def*)
qed

95.2 Enumeration

lemma *inj-on-sat-of-nat*:
shows *inj-on* (*of-nat* :: *nat* ⇒ *'a*::*len sat*) {*0*..<*LENGTH*(*'a*)}
by (*rule inj-onI*) (*simp add*: *sat-eq-iff*)

lemma *UNIV-sat-eq-of-nat*:
shows (*UNIV* :: *'a*::*len sat set*) = *of-nat* ' {*0*..*LENGTH*(*'a*)} (**is** *?lhs* = *?rhs*)
proof −
have $x \in$ *?rhs* **for** $x :: 'a\ sat$
by (*simp add*: *image-eqI*[**where** *x*=*nat-of x*] *sat-eq-iff*)
then show *?thesis*
by *blast*
qed

instantiation *sat* :: (*len*) *enum*
begin

definition *enum-sat* :: *'a sat list* **where**
enum-sat = *map of-nat* [*0*..<*Suc*(*LENGTH*('*a*))]

definition *enum-all-sat* :: (*'a sat* ⇒ *bool*) ⇒ *bool* **where**
enum-all-sat = *All*

definition *enum-ex-sat* :: (*'a sat* ⇒ *bool*) ⇒ *bool* **where**
enum-ex-sat = *Ex*

instance
proof *intro-classes*
show *UNIV* = *set* (*enum-class.enum* :: *'a sat list*)
by (*simp only*: *enum-sat-def UNIV-sat-eq-of-nat set-map flip*: *atLeastAtMost-upt*)
show *distinct* (*enum-class.enum* :: *'a sat list*)
by (*clarsimp simp*: *enum-sat-def distinct-map inj-on-sat-of-nat sat-eq-iff*)
qed (*simp-all add*: *enum-all-sat-def enum-ex-sat-def*)

end

lemma *enum-sat-code* [*code*]:
fixes *P* :: *'a*::*len sat* ⇒ *bool*
shows *Enum.enum-all P* ⟷ *list-all P Enum.enum*
and *Enum.enum-ex P* ⟷ *list-ex P Enum.enum*
by (*simp-all add*: *enum-all-sat-def enum-ex-sat-def enum-UNIV list-all-iff list-ex-iff*)

end

96 Set Idioms

theory *Set-Idioms*
imports *Countable-Set*

begin

96.1 Idioms for being a suitable union/intersection of something

definition *union-of* :: (*'a set set* ⇒ *bool*) ⇒ (*'a set* ⇒ *bool*) ⇒ *'a set* ⇒ *bool*
(**infixr** ‹*union'-of*› *60*)
where *P union-of Q* ≡ λ*S*. ∃*𝒰*. *P 𝒰* ∧ *𝒰* ⊆ *Collect Q* ∧ ⋃*𝒰* = *S*

definition *intersection-of* :: (*'a set set* ⇒ *bool*) ⇒ (*'a set* ⇒ *bool*) ⇒ *'a set* ⇒ *bool*
(**infixr** ‹*intersection'-of*› *60*)
where *P intersection-of Q* ≡ λ*S*. ∃*𝒰*. *P 𝒰* ∧ *𝒰* ⊆ *Collect Q* ∧ ⋂*𝒰* = *S*

definition *arbitrary*:: *'a set set* ⇒ *bool* **where** *arbitrary 𝒰* ≡ *True*

lemma *union-of-inc*: ⟦*P* {*S*}; *Q S*⟧ ⟹ (*P union-of Q*) *S*
by (*auto simp*: *union-of-def*)

lemma *intersection-of-inc*:
$[\![P\ \{S\};\ Q\ S]\!] \Longrightarrow (P\ \textit{intersection-of}\ Q)\ S$
by (*auto simp*: *intersection-of-def*)

lemma *union-of-mono*:
$[\![(P\ \textit{union-of}\ Q)\ S;\ \bigwedge x.\ Q\ x \Longrightarrow Q'\ x]\!] \Longrightarrow (P\ \textit{union-of}\ Q')\ S$
by (*auto simp*: *union-of-def*)

lemma *intersection-of-mono*:
$[\![(P\ \textit{intersection-of}\ Q)\ S;\ \bigwedge x.\ Q\ x \Longrightarrow Q'\ x]\!] \Longrightarrow (P\ \textit{intersection-of}\ Q')\ S$
by (*auto simp*: *intersection-of-def*)

lemma *all-union-of*:
$(\forall S.\ (P\ \textit{union-of}\ Q)\ S \longrightarrow R\ S) \longleftrightarrow (\forall T.\ P\ T \wedge T \subseteq \textit{Collect}\ Q \longrightarrow R(\bigcup T))$
by (*auto simp*: *union-of-def*)

lemma *all-intersection-of*:
$(\forall S.\ (P\ \textit{intersection-of}\ Q)\ S \longrightarrow R\ S) \longleftrightarrow (\forall T.\ P\ T \wedge T \subseteq \textit{Collect}\ Q \longrightarrow R(\bigcap T))$
by (*auto simp*: *intersection-of-def*)

lemma *intersection-ofE*:
$[\![(P\ \textit{intersection-of}\ Q)\ S;\ \bigwedge T.\ [\![P\ T;\ T \subseteq \textit{Collect}\ Q]\!] \Longrightarrow R(\bigcap T)]\!] \Longrightarrow R\ S$
by (*auto simp*: *intersection-of-def*)

lemma *union-of-empty*:
$P\ \{\} \Longrightarrow (P\ \textit{union-of}\ Q)\ \{\}$
by (*auto simp*: *union-of-def*)

lemma *intersection-of-empty*:
$P\ \{\} \Longrightarrow (P\ \textit{intersection-of}\ Q)\ \textit{UNIV}$
by (*auto simp*: *intersection-of-def*)

The arbitrary and finite cases

lemma *arbitrary-union-of-alt*:
$(\textit{arbitrary union-of}\ Q)\ S \longleftrightarrow (\forall x \in S.\ \exists U.\ Q\ U \wedge x \in U \wedge U \subseteq S)$
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
by (*force simp*: *union-of-def arbitrary-def*)
next
assume *?rhs*
then have $\{U.\ Q\ U \wedge U \subseteq S\} \subseteq \textit{Collect}\ Q \bigcup \{U.\ Q\ U \wedge U \subseteq S\} = S$
by *auto*
then show *?lhs*
unfolding *union-of-def arbitrary-def* **by** *blast*
qed

lemma *arbitrary-union-of-empty* [*simp*]: (*arbitrary union-of P*) {}
by (*force simp*: *union-of-def arbitrary-def*)

lemma *arbitrary-intersection-of-empty* [*simp*]:
(*arbitrary intersection-of P*) *UNIV*
by (*force simp*: *intersection-of-def arbitrary-def*)

lemma *arbitrary-union-of-inc*:
P S ⟹ (*arbitrary union-of P*) *S*
by (*force simp*: *union-of-inc arbitrary-def*)

lemma *arbitrary-intersection-of-inc*:
P S ⟹ (*arbitrary intersection-of P*) *S*
by (*force simp*: *intersection-of-inc arbitrary-def*)

lemma *arbitrary-union-of-complement*:
(*arbitrary union-of P*) *S* ⟷ (*arbitrary intersection-of* (λ*S*. *P*(− *S*))) (− *S*)
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then obtain 𝒰 **where** 𝒰 ⊆ *Collect P S* = ⋃𝒰
by (*auto simp*: *union-of-def arbitrary-def*)
then show *?rhs*
unfolding *intersection-of-def arbitrary-def*
by (*rule-tac x*=*uminus* ' 𝒰 **in** *exI*) *auto*
next
assume *?rhs*
then obtain 𝒰 **where** 𝒰 ⊆ {*S*. *P* (− *S*)} ⋂𝒰 = − *S*
by (*auto simp*: *union-of-def intersection-of-def arbitrary-def*)
then show *?lhs*
unfolding *union-of-def arbitrary-def*
by (*rule-tac x*=*uminus* ' 𝒰 **in** *exI*) *auto*
qed

lemma *arbitrary-intersection-of-complement*:
(*arbitrary intersection-of P*) *S* ⟷ (*arbitrary union-of* (λ*S*. *P*(− *S*))) (− *S*)
by (*simp add*: *arbitrary-union-of-complement*)

lemma *arbitrary-union-of-idempot* [*simp*]:
arbitrary union-of arbitrary union-of P = *arbitrary union-of P*
proof −
have *1*: ∃𝒰′⊆*Collect P*. ⋃𝒰′ = ⋃𝒰 **if** 𝒰 ⊆ {*S*. ∃𝒱⊆*Collect P*. ⋃𝒱 = *S*} **for**
𝒰
proof −
let *?𝒲* = {*V*. ∃𝒱. 𝒱⊆*Collect P* ∧ *V* ∈ 𝒱 ∧ (∃ *S* ∈ 𝒰. ⋃𝒱 = *S*)}
have ∗: ⋀*x U*. ⟦*x* ∈ *U*; *U* ∈ 𝒰⟧ ⟹ *x* ∈ ⋃ *?𝒲*
using *that*
apply *simp*

apply (*drule subsetD*, *assumption*, *auto*)
done
show *?thesis*
apply (*rule-tac x*=$\{V.\ \exists \mathcal{V}.\ \mathcal{V} \subseteq Collect\ P \wedge V \in \mathcal{V} \wedge (\exists S \in \mathcal{U}.\ \bigcup \mathcal{V} = S)\}$ **in** *exI*)
using *that* **by** (*blast intro*: $*$)
qed
have *2*: $\exists \mathcal{U}' \subseteq \{S.\ \exists \mathcal{U} \subseteq Collect\ P.\ \bigcup \mathcal{U} = S\}.\ \bigcup \mathcal{U}' = \bigcup \mathcal{U}$ **if** $\mathcal{U} \subseteq Collect\ P$ **for** $\mathcal{U}$
by (*metis* (*mono-tags*, *lifting*) *union-of-def arbitrary-union-of-inc that*)
show *?thesis*
unfolding *union-of-def arbitrary-def* **by** (*force simp*: *1 2*)
qed

lemma *arbitrary-intersection-of-idempot*:
arbitrary intersection-of arbitrary intersection-of P = *arbitrary intersection-of P* (**is** *?lhs* = *?rhs*)
proof −
have − *?lhs* = − *?rhs*
unfolding *arbitrary-intersection-of-complement* **by** *simp*
then show *?thesis*
by *simp*
qed

lemma *arbitrary-union-of-Union*:
$(\bigwedge S.\ S \in \mathcal{U} \Longrightarrow (arbitrary\ union\text{-}of\ P)\ S) \Longrightarrow (arbitrary\ union\text{-}of\ P)\ (\bigcup \mathcal{U})$
by (*metis union-of-def arbitrary-def arbitrary-union-of-idempot mem-Collect-eq subsetI*)

lemma *arbitrary-union-of-Un*:
$[\![(arbitrary\ union\text{-}of\ P)\ S;\ (arbitrary\ union\text{-}of\ P)\ T]\!]$
$\Longrightarrow (arbitrary\ union\text{-}of\ P)\ (S \cup T)$
using *arbitrary-union-of-Union* [*of* $\{S, T\}$] **by** *auto*

lemma *arbitrary-intersection-of-Inter*:
$(\bigwedge S.\ S \in \mathcal{U} \Longrightarrow (arbitrary\ intersection\text{-}of\ P)\ S) \Longrightarrow (arbitrary\ intersection\text{-}of\ P)\ (\bigcap \mathcal{U})$
by (*metis intersection-of-def arbitrary-def arbitrary-intersection-of-idempot mem-Collect-eq subsetI*)

lemma *arbitrary-intersection-of-Int*:
$[\![(arbitrary\ intersection\text{-}of\ P)\ S;\ (arbitrary\ intersection\text{-}of\ P)\ T]\!]$
$\Longrightarrow (arbitrary\ intersection\text{-}of\ P)\ (S \cap T)$
using *arbitrary-intersection-of-Inter* [*of* $\{S, T\}$] **by** *auto*

lemma *arbitrary-union-of-Int-eq*:
$(\forall S\ T.\ (arbitrary\ union\text{-}of\ P)\ S \wedge (arbitrary\ union\text{-}of\ P)\ T \longrightarrow (arbitrary\ union\text{-}of\ P)\ (S \cap T))$
$\longleftrightarrow (\forall S\ T.\ P\ S \wedge P\ T \longrightarrow (arbitrary\ union\text{-}of\ P)\ (S \cap T))$ (**is** *?lhs* = *?rhs*)

proof
assume *?lhs*
then show *?rhs*
by (*simp add*: *arbitrary-union-of-inc*)
next
assume *R*: *?rhs*
show *?lhs*
proof *clarify*
fix S :: *'a set* **and** T :: *'a set*
assume (*arbitrary union-of P*) S **and** (*arbitrary union-of P*) T
then obtain $\mathcal{U}$ $\mathcal{V}$ **where** $*$: $\mathcal{U} \subseteq$ *Collect P* $\bigcup\mathcal{U} = S$ $\mathcal{V} \subseteq$ *Collect P* $\bigcup\mathcal{V} = T$
by (*auto simp*: *union-of-def*)
then have (*arbitrary union-of P*) $(\bigcup C{\in}\mathcal{U}.\ \bigcup D{\in}\mathcal{V}.\ C \cap D)$
using *R* **by** (*blast intro*: *arbitrary-union-of-Union*)
then show (*arbitrary union-of P*) $(S \cap T)$
by (*simp add*: *Int-UN-distrib2* $*$)
qed
qed

lemma *arbitrary-intersection-of-Un-eq*:
$(\forall S\ T.$ (*arbitrary intersection-of P*) $S \wedge$ (*arbitrary intersection-of P*) T
$\longrightarrow$ (*arbitrary intersection-of P*) $(S \cup T)) \longleftrightarrow$
$(\forall S\ T.\ P\ S \wedge P\ T \longrightarrow$ (*arbitrary intersection-of P*) $(S \cup T))$
apply (*simp add*: *arbitrary-intersection-of-complement*)
using *arbitrary-union-of-Int-eq* [*of* $\lambda S.\ P\ (-\ S)$]
by (*metis* (*no-types*, *lifting*) *arbitrary-def double-compl union-of-inc*)

lemma *finite-union-of-empty* [*simp*]: (*finite union-of P*) $\{\}$
by (*simp add*: *union-of-empty*)

lemma *finite-intersection-of-empty* [*simp*]: (*finite intersection-of P*) *UNIV*
by (*simp add*: *intersection-of-empty*)

lemma *finite-union-of-inc*:
$P\ S \Longrightarrow$ (*finite union-of P*) S
by (*simp add*: *union-of-inc*)

lemma *finite-intersection-of-inc*:
$P\ S \Longrightarrow$ (*finite intersection-of P*) S
by (*simp add*: *intersection-of-inc*)

lemma *finite-union-of-complement*:
(*finite union-of P*) $S \longleftrightarrow$ (*finite intersection-of* $(\lambda S.\ P(-\ S))$) $(-\ S)$
unfolding *union-of-def intersection-of-def*
apply *safe*
apply (*rule-tac x*=*uminus* '$\mathcal{U}$ **in** *exI*, *fastforce*)+
done

lemma *finite-intersection-of-complement*:

(*finite intersection-of P*) $S \longleftrightarrow$ (*finite union-of* ($\lambda S.\ P(-\ S)$)) ($-\ S$)
by (*simp add*: *finite-union-of-complement*)

lemma *finite-union-of-idempot* [*simp*]:
finite union-of finite union-of P = *finite union-of P*
proof −
have (*finite union-of P*) *S* **if** *S*: (*finite union-of finite union-of P*) *S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** *finite* $\mathcal{U}$ $S = \bigcup \mathcal{U}$ **and** $\mathcal{U}$: $\forall\, U{\in}\mathcal{U}.\ \exists \mathcal{U}.$ *finite* $\mathcal{U} \wedge (\mathcal{U} \subseteq$ *Collect P*$) \wedge \bigcup \mathcal{U} = U$
using *S* **unfolding** *union-of-def* **by** (*auto simp*: *subset-eq*)
then obtain *f* **where** $\forall\, U{\in}\mathcal{U}.$ *finite* (*f U*) $\wedge$ (*f U* $\subseteq$ *Collect P*) $\wedge \bigcup$(*f U*) = *U*
by *metis*
then show *?thesis*
unfolding *union-of-def* ‹$S = \bigcup \mathcal{U}$›
by (*rule-tac x* = *snd* ‘ *Sigma* $\mathcal{U}$ *f* **in** *exI*) (*fastforce simp*: ‹*finite* $\mathcal{U}$›)
qed
moreover
have (*finite union-of finite union-of P*) *S* **if** (*finite union-of P*) *S* **for** *S*
by (*simp add*: *finite-union-of-inc that*)
ultimately show *?thesis*
by *force*
qed

lemma *finite-intersection-of-idempot* [*simp*]:
finite intersection-of finite intersection-of P = *finite intersection-of P*
by (*force simp*: *finite-intersection-of-complement*)

lemma *finite-union-of-Union*:
⟦*finite* $\mathcal{U}$; $\bigwedge S.\ S \in \mathcal{U} \Longrightarrow$ (*finite union-of P*) *S*⟧ $\Longrightarrow$ (*finite union-of P*) ($\bigcup \mathcal{U}$)
using *finite-union-of-idempot* [*of P*]
by (*metis mem-Collect-eq subsetI union-of-def*)

lemma *finite-union-of-Un*:
⟦(*finite union-of P*) *S*; (*finite union-of P*) *T*⟧ $\Longrightarrow$ (*finite union-of P*) ($S \cup T$)
by (*auto simp*: *union-of-def*)

lemma *finite-intersection-of-Inter*:
⟦*finite* $\mathcal{U}$; $\bigwedge S.\ S \in \mathcal{U} \Longrightarrow$ (*finite intersection-of P*) *S*⟧ $\Longrightarrow$ (*finite intersection-of P*) ($\bigcap \mathcal{U}$)
using *finite-intersection-of-idempot* [*of P*]
by (*metis intersection-of-def mem-Collect-eq subsetI*)

lemma *finite-intersection-of-Int*:
⟦(*finite intersection-of P*) *S*; (*finite intersection-of P*) *T*⟧
$\Longrightarrow$ (*finite intersection-of P*) ($S \cap T$)
by (*auto simp*: *intersection-of-def*)

lemma *finite-union-of-Int-eq*:

(∀ *S T*. (*finite union-of P*) *S* ∧ (*finite union-of P*) *T* ⟶ (*finite union-of P*) (*S* ∩ *T*))
⟷ (∀ *S T*. *P S* ∧ *P T* ⟶ (*finite union-of P*) (*S* ∩ *T*))
(**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
then show *?rhs*
by (*simp add*: *finite-union-of-inc*)
next
assume *R*: *?rhs*
show *?lhs*
proof *clarify*
fix *S* :: *'a set* **and** *T* :: *'a set*
assume (*finite union-of P*) *S* **and** (*finite union-of P*) *T*
then obtain *U V* **where** ∗: *U* ⊆ *Collect P* ⋃*U* = *S finite U V* ⊆ *Collect P* ⋃*V* = *T finite V*
by (*auto simp*: *union-of-def*)
then have (*finite union-of P*) (⋃ *C*∈*U*. ⋃ *D*∈*V*. *C* ∩ *D*)
using *R*
by (*blast intro*: *finite-union-of-Union*)
then show (*finite union-of P*) (*S* ∩ *T*)
by (*simp add*: *Int-UN-distrib2* ∗)
qed
qed

lemma *finite-intersection-of-Un-eq*:
(∀ *S T*. (*finite intersection-of P*) *S* ∧
(*finite intersection-of P*) *T*
⟶ (*finite intersection-of P*) (*S* ∪ *T*)) ⟷
(∀ *S T*. *P S* ∧ *P T* ⟶ (*finite intersection-of P*) (*S* ∪ *T*))
apply (*simp add*: *finite-intersection-of-complement*)
using *finite-union-of-Int-eq* [*of* λ*S*. *P* (− *S*)]
by (*metis* (*no-types*, *lifting*) *double-compl*)

abbreviation *finite'* :: *'a set* ⇒ *bool*
where *finite' A* ≡ *finite A* ∧ *A* ≠ {}

lemma *finite'-intersection-of-Int*:
⟦(*finite' intersection-of P*) *S*; (*finite' intersection-of P*) *T*⟧
⟹ (*finite' intersection-of P*) (*S* ∩ *T*)
by (*auto simp*: *intersection-of-def*)

lemma *finite'-intersection-of-inc*:
P S ⟹ (*finite' intersection-of P*) *S*
by (*simp add*: *intersection-of-inc*)

96.2 The "Relative to" operator

A somewhat cheap but handy way of getting localized forms of various topological concepts (open, closed, borel, fsigma, gdelta etc.)

definition *relative-to* :: ['*a set* ⇒ *bool*, '*a set*, '*a set*] ⇒ *bool* (**infixl** ‹*relative'-to*› *55*)
where P *relative-to* $S \equiv \lambda T.\ \exists U.\ P\ U \wedge S \cap U = T$

lemma *relative-to-UNIV* [*simp*]: (P *relative-to UNIV*) $S \longleftrightarrow P\ S$
by (*simp add*: *relative-to-def*)

lemma *relative-to-imp-subset*:
(P *relative-to* S) $T \Longrightarrow T \subseteq S$
by (*auto simp*: *relative-to-def*)

lemma *all-relative-to*: $(\forall S.\ (P \text{ relative-to } U)\ S \longrightarrow Q\ S) \longleftrightarrow (\forall S.\ P\ S \longrightarrow Q(U \cap S))$
by (*auto simp*: *relative-to-def*)

lemma *relative-toE*: $[\![(P \text{ relative-to } U)\ S; \bigwedge S.\ P\ S \Longrightarrow Q(U \cap S)]\!] \Longrightarrow Q\ S$
by (*auto simp*: *relative-to-def*)

lemma *relative-to-inc*:
$P\ S \Longrightarrow (P \text{ relative-to } U)\ (U \cap S)$
by (*auto simp*: *relative-to-def*)

lemma *relative-to-relative-to* [*simp*]:
P *relative-to* S *relative-to* $T = P$ *relative-to* $(S \cap T)$
unfolding *relative-to-def*
by *auto*

lemma *relative-to-compl*:
$S \subseteq U \Longrightarrow ((P \text{ relative-to } U)\ (U - S) \longleftrightarrow ((\lambda c.\ P(-\ c)) \text{ relative-to } U)\ S)$
unfolding *relative-to-def*
by (*metis Diff-Diff-Int Diff-eq double-compl inf.absorb-iff2*)

lemma *relative-to-subset-trans*:
$[\![(P \text{ relative-to } U)\ S; S \subseteq T; T \subseteq U]\!] \Longrightarrow (P \text{ relative-to } T)\ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-mono*:
$[\![(P \text{ relative-to } U)\ S; \bigwedge S.\ P\ S \Longrightarrow Q\ S]\!] \Longrightarrow (Q \text{ relative-to } U)\ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-subset-inc*: $[\![S \subseteq U; P\ S]\!] \Longrightarrow (P \text{ relative-to } U)\ S$
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-Int*:
$[\![(P \text{ relative-to } S)\ C; (P \text{ relative-to } S)\ D; \bigwedge X\ Y.\ [\![P\ X; P\ Y]\!] \Longrightarrow P(X \cap Y)]\!]$

$\Longrightarrow$ (*P relative-to S*) (*C* $\cap$ *D*)
unfolding *relative-to-def* **by** *auto*

lemma *relative-to-Un*:
⟦(*P relative-to S*) *C*; (*P relative-to S*) *D*; $\bigwedge X\ Y.$ ⟦*P X*; *P Y*⟧ $\Longrightarrow$ *P*(*X* $\cup$ *Y*)⟧
$\Longrightarrow$ (*P relative-to S*) (*C* $\cup$ *D*)
unfolding *relative-to-def* **by** *auto*

lemma *arbitrary-union-of-relative-to*:
((*arbitrary union-of P*) *relative-to U*) = (*arbitrary union-of* (*P relative-to U*))
(**is** *?lhs* = *?rhs*)
proof −
have *?rhs S* **if** *L*: *?lhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** *S* = *U* $\cap$ $\bigcup\mathcal{U}$ $\mathcal{U}$ $\subseteq$ *Collect P*
using *L* **unfolding** *relative-to-def union-of-def* **by** *auto*
then show *?thesis*
unfolding *relative-to-def union-of-def arbitrary-def*
by (*rule-tac x*=(λ*X*. *U* $\cap$ *X*) ‘ $\mathcal{U}$ **in** *exI*) *auto*
qed
moreover have *?lhs S* **if** *R*: *?rhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** *S* = $\bigcup\mathcal{U}$ ∀ *T*∈$\mathcal{U}$. ∃ *V*. *P V* ∧ *U* $\cap$ *V* = *T*
using *R* **unfolding** *relative-to-def union-of-def* **by** *auto*
then obtain *f* **where** *f*: $\bigwedge T.$ *T* $\in$ $\mathcal{U}$ $\Longrightarrow$ *P* (*f T*) $\bigwedge T.$ *T* $\in$ $\mathcal{U}$ $\Longrightarrow$ *U* $\cap$ (*f T*)
= *T*
by *metis*
then have ∃$\mathcal{U}'$⊆*Collect P*. $\bigcup\mathcal{U}'$ = $\bigcup$ (*f* ‘ $\mathcal{U}$)
by (*metis image-subset-iff mem-Collect-eq*)
moreover have *eq*: *U* $\cap$ $\bigcup$ (*f* ‘ $\mathcal{U}$) = $\bigcup\mathcal{U}$
using *f* **by** *auto*
ultimately show *?thesis*
unfolding *relative-to-def union-of-def arbitrary-def* ‹*S* = $\bigcup\mathcal{U}$›
by *metis*
qed
ultimately show *?thesis*
by *blast*
qed

lemma *finite-union-of-relative-to*:
((*finite union-of P*) *relative-to U*) = (*finite union-of* (*P relative-to U*)) (**is** *?lhs*
= *?rhs*)
proof −
have *?rhs S* **if** *L*: *?lhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** *S* = *U* $\cap$ $\bigcup\mathcal{U}$ $\mathcal{U}$ $\subseteq$ *Collect P finite* $\mathcal{U}$
using *L* **unfolding** *relative-to-def union-of-def* **by** *auto*
then show *?thesis*
unfolding *relative-to-def union-of-def*

```
    by (rule-tac x=(λX. U ∩ X) ` 𝒰 in exI) auto
  qed
  moreover have ?lhs S if R: ?rhs S for S
  proof −
    obtain 𝒰 where S = ⋃𝒰 ∀ T∈𝒰. ∃ V. P V ∧ U ∩ V = T finite 𝒰
      using R unfolding relative-to-def union-of-def by auto
    then obtain f where f: ⋀T. T ∈ 𝒰 ⟹ P (f T) ⋀T. T ∈ 𝒰 ⟹ U ∩ (f T)
= T
      by metis
    then have ∃𝒰'⊆Collect P. ⋃𝒰' = ⋃ (f ` 𝒰)
      by (metis image-subset-iff mem-Collect-eq)
    moreover have eq: U ∩ ⋃ (f ` 𝒰) = ⋃𝒰
      using f by auto
    ultimately show ?thesis
      using ‹finite 𝒰› f
      unfolding relative-to-def union-of-def ‹S = ⋃𝒰›
    by (rule-tac x=⋃ (f ` 𝒰) in exI) (metis finite-imageI image-subsetI mem-Collect-eq)
  qed
  ultimately show ?thesis
    by blast
qed

lemma countable-union-of-relative-to:
  ((countable union-of P) relative-to U) = (countable union-of (P relative-to U))
(is ?lhs = ?rhs)
proof −
  have ?rhs S if L: ?lhs S for S
  proof −
    obtain 𝒰 where S = U ∩ ⋃𝒰 𝒰 ⊆ Collect P countable 𝒰
      using L unfolding relative-to-def union-of-def by auto
    then show ?thesis
      unfolding relative-to-def union-of-def
      by (rule-tac x=(λX. U ∩ X) ` 𝒰 in exI) auto
  qed
  moreover have ?lhs S if R: ?rhs S for S
  proof −
    obtain 𝒰 where S = ⋃𝒰 ∀ T∈𝒰. ∃ V. P V ∧ U ∩ V = T countable 𝒰
      using R unfolding relative-to-def union-of-def by auto
    then obtain f where f: ⋀T. T ∈ 𝒰 ⟹ P (f T) ⋀T. T ∈ 𝒰 ⟹ U ∩ (f T)
= T
      by metis
    then have ∃𝒰'⊆Collect P. ⋃𝒰' = ⋃ (f ` 𝒰)
      by (metis image-subset-iff mem-Collect-eq)
    moreover have eq: U ∩ ⋃ (f ` 𝒰) = ⋃𝒰
      using f by auto
    ultimately show ?thesis
      using ‹countable 𝒰› f
      unfolding relative-to-def union-of-def ‹S = ⋃𝒰›
        by (rule-tac x=⋃ (f ` 𝒰) in exI) (metis countable-image image-subsetI
```

mem-Collect-eq)
qed
ultimately show *?thesis*
by *blast*
qed

lemma *arbitrary-intersection-of-relative-to*:
(*(arbitrary intersection-of P) relative-to U*) = (*(arbitrary intersection-of (P relative-to U)) relative-to U*) (**is** *?lhs = ?rhs*)
proof −
have *?rhs S* **if** *L*: *?lhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** $\mathcal{U}$: $S = U \cap \bigcap \mathcal{U}$ $\mathcal{U} \subseteq$ *Collect P*
using *L* **unfolding** *relative-to-def intersection-of-def* **by** *auto*
show *?thesis*
unfolding *relative-to-def intersection-of-def arbitrary-def*
proof (*intro exI conjI*)
show $U \cap (\bigcap X{\in}\mathcal{U}.\ U \cap X) = S$ $(\cap)$ U ‘ $\mathcal{U} \subseteq \{T.\ \exists Ua.\ P\ Ua \wedge U \cap Ua = T\}$
using $\mathcal{U}$ **by** *blast+*
qed *auto*
qed
moreover have *?lhs S* **if** *R*: *?rhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** $S = U \cap \bigcap \mathcal{U}$ $\forall T{\in}\mathcal{U}.\ \exists V.\ P\ V \wedge U \cap V = T$
using *R* **unfolding** *relative-to-def intersection-of-def* **by** *auto*
then obtain *f* **where** *f*: $\bigwedge T.\ T \in \mathcal{U} \Longrightarrow P\ (f\ T)$ $\bigwedge T.\ T \in \mathcal{U} \Longrightarrow U \cap (f\ T) = T$
by *metis*
then have *f* ‘ $\mathcal{U} \subseteq$ *Collect P*
by *auto*
moreover have *eq*: $U \cap \bigcap (f\ ‘\ \mathcal{U}) = U \cap \bigcap \mathcal{U}$
using *f* **by** *auto*
ultimately show *?thesis*
unfolding *relative-to-def intersection-of-def arbitrary-def* ‹$S = U \cap \bigcap \mathcal{U}$›
by *auto*
qed
ultimately show *?thesis*
by *blast*
qed

lemma *finite-intersection-of-relative-to*:
(*(finite intersection-of P) relative-to U*) = (*(finite intersection-of (P relative-to U)) relative-to U*) (**is** *?lhs = ?rhs*)
proof −
have *?rhs S* **if** *L*: *?lhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** $\mathcal{U}$: $S = U \cap \bigcap \mathcal{U}$ $\mathcal{U} \subseteq$ *Collect P finite* $\mathcal{U}$

```
    using L unfolding relative-to-def intersection-of-def by auto
  show ?thesis
    unfolding relative-to-def intersection-of-def
  proof (intro exI conjI)
    show U ∩ (⋂X∈𝒰. U ∩ X) = S (∩) U ‘ 𝒰 ⊆ {T. ∃Ua. P Ua ∧ U ∩ Ua =
T}
      using 𝒰 by blast+
    show finite ((∩) U ‘ 𝒰)
      by (simp add: ‹finite 𝒰›)
  qed auto
 qed
 moreover have ?lhs S if R: ?rhs S for S
 proof −
  obtain 𝒰 where S = U ∩ ⋂𝒰 ∀T∈𝒰. ∃V. P V ∧ U ∩ V = T finite 𝒰
    using R unfolding relative-to-def intersection-of-def by auto
  then obtain f where f: ⋀T. T ∈ 𝒰 ⟹ P (f T) ⋀T. T ∈ 𝒰 ⟹ U ∩ (f T)
= T
    by metis
  then have f ‘ 𝒰 ⊆ Collect P
    by auto
  moreover have eq: U ∩ ⋂ (f ‘ 𝒰) = U ∩ ⋂ 𝒰
    using f by auto
  ultimately show ?thesis
    unfolding relative-to-def intersection-of-def ‹S = U ∩ ⋂𝒰›
    using ‹finite 𝒰›
    by auto
 qed
 ultimately show ?thesis
  by blast
qed

lemma countable-intersection-of-relative-to:
  ((countable intersection-of P) relative-to U) = ((countable intersection-of (P
relative-to U)) relative-to U) (is ?lhs = ?rhs)
proof −
 have ?rhs S if L: ?lhs S for S
 proof −
  obtain 𝒰 where 𝒰: S = U ∩ ⋂𝒰 𝒰 ⊆ Collect P countable 𝒰
    using L unfolding relative-to-def intersection-of-def by auto
  show ?thesis
    unfolding relative-to-def intersection-of-def
  proof (intro exI conjI)
    show U ∩ (⋂X∈𝒰. U ∩ X) = S (∩) U ‘ 𝒰 ⊆ {T. ∃Ua. P Ua ∧ U ∩ Ua =
T}
      using 𝒰 by blast+
    show countable ((∩) U ‘ 𝒰)
      by (simp add: ‹countable 𝒰›)
  qed auto
 qed
```

moreover have *?lhs S* **if** *R*: *?rhs S* **for** *S*
proof −
obtain $\mathcal{U}$ **where** $S = U \cap \bigcap \mathcal{U}$ $\forall T{\in}\mathcal{U}.\ \exists V.\ P\ V \wedge U \cap V = T$ *countable* $\mathcal{U}$
using *R* **unfolding** *relative-to-def intersection-of-def* **by** *auto*
then obtain *f* **where** *f*: $\bigwedge T.\ T \in \mathcal{U} \Longrightarrow P\ (f\ T)$ $\bigwedge T.\ T \in \mathcal{U} \Longrightarrow U \cap (f\ T) = T$
by *metis*
then have $f\ `\ \mathcal{U} \subseteq$ *Collect P*
by *auto*
moreover have *eq*: $U \cap \bigcap\ (f\ `\mathcal{U}) = U \cap \bigcap\ \mathcal{U}$
using *f* **by** *auto*
ultimately show *?thesis*
unfolding *relative-to-def intersection-of-def* ‹$S = U \cap \bigcap \mathcal{U}$›
using ‹*countable* $\mathcal{U}$› *countable-image*
by *auto*
qed
ultimately show *?thesis*
by *blast*
qed

lemma *countable-union-of-empty* [*simp*]: (*countable union-of P*) {}
by (*simp add*: *union-of-empty*)

lemma *countable-intersection-of-empty* [*simp*]: (*countable intersection-of P*) *UNIV*
by (*simp add*: *intersection-of-empty*)

lemma *countable-union-of-inc*: *P S* ⟹ (*countable union-of P*) *S*
by (*simp add*: *union-of-inc*)

lemma *countable-intersection-of-inc*: *P S* ⟹ (*countable intersection-of P*) *S*
by (*simp add*: *intersection-of-inc*)

lemma *countable-union-of-complement*:
(*countable union-of P*) *S* ⟷ (*countable intersection-of* (λ*S*. *P*(−*S*))) (−*S*)
(**is** *?lhs*=*?rhs*)
proof
assume *?lhs*
then obtain $\mathcal{U}$ **where** *countable* $\mathcal{U}$ **and** $\mathcal{U}$: $\mathcal{U} \subseteq$ *Collect P* $\bigcup \mathcal{U} = S$
by (*metis union-of-def*)
define $\mathcal{U}'$ **where** $\mathcal{U}' \equiv (\lambda C.\ -C)\ `\mathcal{U}$
have $\mathcal{U}' \subseteq \{S.\ P\ (-\ S)\}$ $\bigcap \mathcal{U}' = -S$
using $\mathcal{U}'$*-def* $\mathcal{U}$ **by** *auto*
then show *?rhs*
unfolding *intersection-of-def* **by** (*metis* $\mathcal{U}'$*-def* ‹*countable* $\mathcal{U}$› *countable-image*)
next
assume *?rhs*
then obtain $\mathcal{U}$ **where** *countable* $\mathcal{U}$ **and** $\mathcal{U}$: $\mathcal{U} \subseteq \{S.\ P\ (-\ S)\}$ $\bigcap \mathcal{U} = -S$
by (*metis intersection-of-def*)
define $\mathcal{U}'$ **where** $\mathcal{U}' \equiv (\lambda C.\ -C)\ `\mathcal{U}$

have $\mathcal{U}' \subseteq$ *Collect* $P \bigcup \mathcal{U}' = S$
using $\mathcal{U}'$*-def* $\mathcal{U}$ **by** *auto*
then show *?lhs*
unfolding *union-of-def*
by (*metis* $\mathcal{U}'$*-def* ‹*countable* $\mathcal{U}$› *countable-image*)
qed

lemma *countable-intersection-of-complement*:
(*countable intersection-of* P) $S \longleftrightarrow$ (*countable union-of* $(\lambda S.\ P(-\ S))$) $(-\ S)$
by (*simp add*: *countable-union-of-complement*)

lemma *countable-union-of-explicit*:
assumes $P\ \{\}$
shows (*countable union-of* P) $S \longleftrightarrow$
$(\exists\, T.\ (\forall\, n{::}nat.\ P(T\ n)) \wedge \bigcup(range\ T) = S)$ (**is** *?lhs=?rhs*)
proof
assume *?lhs*
then obtain $\mathcal{U}$ **where** *countable* $\mathcal{U}$ **and** $\mathcal{U}$: $\mathcal{U} \subseteq$ *Collect* $P \bigcup \mathcal{U} = S$
by (*metis union-of-def*)
then show *?rhs*
by (*metis SUP-bot Sup-empty assms from-nat-into mem-Collect-eq range-from-nat-into subsetD*)
next
assume *?rhs*
then show *?lhs*
by (*metis countableI-type countable-image image-subset-iff mem-Collect-eq union-of-def*)
qed

lemma *countable-union-of-ascending*:
assumes *empty*: $P\ \{\}$ **and** *Un*: $\bigwedge T\ U.\ [\![P\ T;\ P\ U]\!] \Longrightarrow P(T \cup U)$
shows (*countable union-of* P) $S \longleftrightarrow$
$(\exists\, T.\ (\forall\, n.\ P(T\ n)) \wedge (\forall\, n.\ T\ n \subseteq T(Suc\ n)) \wedge \bigcup(range\ T) = S)$ (**is** *?lhs=?rhs*)
proof
assume *?lhs*
then obtain T **where** T: $\bigwedge n{::}nat.\ P(T\ n) \bigcup(range\ T) = S$
by (*meson empty countable-union-of-explicit*)
have $P\ (\bigcup\ (T\ `\ \{..n\}))$ **for** n
by (*induction n*) (*auto simp*: *atMost-Suc Un T*)
with T **show** *?rhs*
by (*rule-tac* $x{=}\lambda n.\ \bigcup k{\leq}n.\ T\ k$ **in** *exI*) *force*
next
assume *?rhs*
then show *?lhs*
using *empty countable-union-of-explicit* **by** *auto*
qed

lemma *countable-union-of-idem* [*simp*]:
countable union-of countable union-of P = *countable union-of* P (**is** *?lhs=?rhs*)

proof
fix S
show (*countable union-of countable union-of P*) S = (*countable union-of P*) S
proof
assume L: *?lhs S*
then obtain $\mathcal{U}$ **where** *countable* $\mathcal{U}$ **and** $\mathcal{U}$: $\mathcal{U} \subseteq$ *Collect* (*countable union-of P*) $\bigcup\mathcal{U} = S$
by (*metis union-of-def*)
then have $\forall\, U \in \mathcal{U}.\ \exists\mathcal{V}.$ *countable* $\mathcal{V} \wedge \mathcal{V} \subseteq$ *Collect P* $\wedge\ U = \bigcup\mathcal{V}$
by (*metis Ball-Collect union-of-def*)
then obtain $\mathcal{F}$ **where** $\mathcal{F}$: $\forall\, U \in \mathcal{U}.$ *countable* ($\mathcal{F}$ U) $\wedge$ $\mathcal{F}$ $U \subseteq$ *Collect P* $\wedge$ $U = \bigcup(\mathcal{F}\ U)$
by *metis*
have *countable* ($\bigcup$ ($\mathcal{F}$ ‘ $\mathcal{U}$))
using $\mathcal{F}$ ‹*countable* $\mathcal{U}$› **by** *blast*
moreover have $\bigcup$ ($\mathcal{F}$ ‘ $\mathcal{U}$) $\subseteq$ *Collect P*
by (*simp add*: *Sup-le-iff* $\mathcal{F}$)
moreover have $\bigcup$ ($\bigcup$ ($\mathcal{F}$ ‘ $\mathcal{U}$)) = S
by *auto* (*metis Union-iff* $\mathcal{F}$ $\mathcal{U}$(*2*))+
ultimately show *?rhs S*
by (*meson union-of-def*)
qed (*simp add*: *countable-union-of-inc*)
qed

lemma *countable-intersection-of-idem* [*simp*]:
countable intersection-of countable intersection-of P =
countable intersection-of P
by (*force simp*: *countable-intersection-of-complement*)

lemma *countable-union-of-Union*:
⟦*countable* $\mathcal{U}$; $\bigwedge S.\ S \in \mathcal{U} \Longrightarrow$ (*countable union-of P*) S⟧
$\Longrightarrow$ (*countable union-of P*) ($\bigcup$ $\mathcal{U}$)
by (*metis Ball-Collect countable-union-of-idem union-of-def*)

lemma *countable-union-of-UN*:
⟦*countable* I; $\bigwedge i.\ i \in I \Longrightarrow$ (*countable union-of P*) (U i)⟧
$\Longrightarrow$ (*countable union-of P*) ($\bigcup i{\in}I.\ U\ i$)
by (*metis* (*mono-tags*, *lifting*) *countable-image countable-union-of-Union imageE*)

lemma *countable-union-of-Un*:
⟦(*countable union-of P*) S; (*countable union-of P*) T⟧
$\Longrightarrow$ (*countable union-of P*) ($S \cup T$)
by (*smt* (*verit*) *Union-Un-distrib countable-Un le-sup-iff union-of-def*)

lemma *countable-intersection-of-Inter*:
⟦*countable* $\mathcal{U}$; $\bigwedge S.\ S \in \mathcal{U} \Longrightarrow$ (*countable intersection-of P*) S⟧
$\Longrightarrow$ (*countable intersection-of P*) ($\bigcap$ $\mathcal{U}$)
by (*metis countable-intersection-of-idem intersection-of-def mem-Collect-eq subsetI*)

lemma *countable-intersection-of-INT*:
⟦*countable I*; ⋀*i*. *i* ∈ *I* ⟹ (*countable intersection-of P*) (*U i*)⟧
⟹ (*countable intersection-of P*) (⋂ *i*∈*I*. *U i*)
by (*metis* (*mono-tags*, *lifting*) *countable-image countable-intersection-of-Inter imageE*)

lemma *countable-intersection-of-inter*:
⟦(*countable intersection-of P*) *S*; (*countable intersection-of P*) *T*⟧
⟹ (*countable intersection-of P*) (*S* ∩ *T*)
by (*simp add*: *countable-intersection-of-complement countable-union-of-Un*)

lemma *countable-union-of-Int*:
assumes *S*: (*countable union-of P*) *S* **and** *T*: (*countable union-of P*) *T*
and *Int*: ⋀*S T*. *P S* ∧ *P T* ⟹ *P*(*S* ∩ *T*)
shows (*countable union-of P*) (*S* ∩ *T*)
proof −
obtain *U* **where** *countable U* **and** *U*: *U* ⊆ *Collect P* ⋃*U* = *S*
using *S* **by** (*metis union-of-def*)
obtain *V* **where** *countable V* **and** *V*: *V* ⊆ *Collect P* ⋃*V* = *T*
using *T* **by** (*metis union-of-def*)
have ⋀*U V*. ⟦*U* ∈ *U*; *V* ∈ *V*⟧ ⟹ (*countable union-of P*) (*U* ∩ *V*)
using *U V* **by** (*metis Ball-Collect countable-union-of-inc local.Int*)
then have (*countable union-of P*) (⋃ *U*∈*U*. ⋃ *V*∈*V*. *U* ∩ *V*)
by (*meson* ‹*countable U*› ‹*countable V*› *countable-union-of-UN*)
moreover have *S* ∩ *T* = (⋃ *U*∈*U*. ⋃ *V*∈*V*. *U* ∩ *V*)
by (*simp add*: *U V*)
ultimately show *?thesis*
by *presburger*
qed

lemma *countable-intersection-of-union*:
assumes *S*: (*countable intersection-of P*) *S* **and** *T*: (*countable intersection-of P*) *T*
and *Un*: ⋀*S T*. *P S* ∧ *P T* ⟹ *P*(*S* ∪ *T*)
shows (*countable intersection-of P*) (*S* ∪ *T*)
by (*metis* (*mono-tags*, *lifting*) *Compl-Int S T Un compl-sup countable-intersection-of-complement countable-union-of-Int*)

end

97 Signed division: negative results rounded towards zero rather than minus infinity.

theory *Signed-Division*
imports *Main*
begin

class *signed-divide* =
  fixes *signed-divide* :: ‹'*a* ⇒ '*a* ⇒ '*a*› (**infixl** ‹*sdiv*› *70*)

class *signed-modulo* =
  fixes *signed-modulo* :: ‹'*a* ⇒ '*a* ⇒ '*a*› (**infixl** ‹*smod*› *70*)

class *signed-division* = *comm-semiring-1-cancel* + *signed-divide* + *signed-modulo* +
  assumes *sdiv-mult-smod-eq*: ‹*a sdiv b* ∗ *b* + *a smod b* = *a*›
begin

lemma *mult-sdiv-smod-eq*:
  ‹*b* ∗ (*a sdiv b*) + *a smod b* = *a*›
  using *sdiv-mult-smod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *smod-sdiv-mult-eq*:
  ‹*a smod b* + *a sdiv b* ∗ *b* = *a*›
  using *sdiv-mult-smod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *smod-mult-sdiv-eq*:
  ‹*a smod b* + *b* ∗ (*a sdiv b*) = *a*›
  using *sdiv-mult-smod-eq* [*of a b*] **by** (*simp add*: *ac-simps*)

lemma *minus-sdiv-mult-eq-smod*:
  ‹*a* − *a sdiv b* ∗ *b* = *a smod b*›
  by (*rule add-implies-diff* [*symmetric*]) (*fact smod-sdiv-mult-eq*)

lemma *minus-mult-sdiv-eq-smod*:
  ‹*a* − *b* ∗ (*a sdiv b*) = *a smod b*›
  by (*rule add-implies-diff* [*symmetric*]) (*fact smod-mult-sdiv-eq*)

lemma *minus-smod-eq-sdiv-mult*:
  ‹*a* − *a smod b* = *a sdiv b* ∗ *b*›
  by (*rule add-implies-diff* [*symmetric*]) (*fact sdiv-mult-smod-eq*)

lemma *minus-smod-eq-mult-sdiv*:
  ‹*a* − *a smod b* = *b* ∗ (*a sdiv b*)›
  by (*rule add-implies-diff* [*symmetric*]) (*fact mult-sdiv-smod-eq*)

end

The following specification of division is named "T-division" in [2]. It is motivated by ISO C99, which in turn adopted the typical behavior of hardware modern in the beginning of the 1990ies; but note ISO C99 describes the instance on machine words, not mathematical integers.

instantiation *int* :: *signed-division*
begin

definition *signed-divide-int* :: ‹*int* ⇒ *int* ⇒ *int*›

where ‹*k sdiv l = sgn k * sgn l * (|k| div |l|)*› **for** *k l :: int*

definition *signed-modulo-int* :: ‹*int ⇒ int ⇒ int*›
where ‹*k smod l = sgn k * (|k| mod |l|)*› **for** *k l :: int*

instance by *standard*
(*simp add*: *signed-divide-int-def signed-modulo-int-def div-abs-eq mod-abs-eq algebra-simps*)

end

lemma *divide-int-eq-signed-divide-int*:
‹*k div l = k sdiv l − of-bool (l ≠ 0 ∧ sgn k ≠ sgn l ∧ ¬ l dvd k)*›
for *k l :: int*
by (*simp add*: *div-eq-div-abs* [*of k l*] *signed-divide-int-def*)

lemma *signed-divide-int-eq-divide-int*:
‹*k sdiv l = k div l + of-bool (l ≠ 0 ∧ sgn k ≠ sgn l ∧ ¬ l dvd k)*›
for *k l :: int*
by (*simp add*: *divide-int-eq-signed-divide-int*)

lemma *modulo-int-eq-signed-modulo-int*:
‹*k mod l = k smod l + l * of-bool (sgn k ≠ sgn l ∧ ¬ l dvd k)*›
for *k l :: int*
by (*simp add*: *mod-eq-mod-abs* [*of k l*] *signed-modulo-int-def*)

lemma *signed-modulo-int-eq-modulo-int*:
‹*k smod l = k mod l − l * of-bool (sgn k ≠ sgn l ∧ ¬ l dvd k)*›
for *k l :: int*
by (*simp add*: *modulo-int-eq-signed-modulo-int*)

lemma *sdiv-int-div-0*:
(*x :: int*) *sdiv 0 = 0*
by (*clarsimp simp*: *signed-divide-int-def*)

lemma *sdiv-int-0-div* [*simp*]:
0 sdiv (*x :: int*) *= 0*
by (*clarsimp simp*: *signed-divide-int-def*)

lemma *smod-int-alt-def*:
(*a*::*int*) *smod b = sgn* (*a*) * (*abs a mod abs b*)
by (*fact signed-modulo-int-def*)

lemma *int-sdiv-simps* [*simp*]:
(*a :: int*) *sdiv 1 = a*
(*a :: int*) *sdiv 0 = 0*
(*a :: int*) *sdiv −1 = −a*
by (*auto simp*: *signed-divide-int-def sgn-if*)

lemma *smod-int-mod-0* [*simp*]:
  x smod (*0* :: *int*) = *x*
  by (*clarsimp simp*: *signed-modulo-int-def abs-mult-sgn ac-simps*)

lemma *smod-int-0-mod* [*simp*]:
  0 smod (*x* :: *int*) = *0*
  by (*clarsimp simp*: *smod-int-alt-def*)

lemma *sgn-sdiv-eq-sgn-mult*:
  a sdiv b ≠ *0* ⟹ *sgn* ((*a* :: *int*) *sdiv b*) = *sgn* (*a* ∗ *b*)
  by (*auto simp*: *signed-divide-int-def sgn-div-eq-sgn-mult sgn-mult*)

lemma *int-sdiv-same-is-1* [*simp*]:
  assumes *a* ≠ *0*
  shows ((*a* :: *int*) *sdiv b* = *a*) = (*b* = *1*)
proof −
  have *b* = *1* **if** *a sdiv b* = *a*
  proof −
    have *b*>*0*
      by (*smt* (*verit*, *ccfv-threshold*) *assms mult-cancel-left2 sgn-if sgn-mult sgn-sdiv-eq-sgn-mult that*)
    then show *?thesis*
      by (*smt* (*verit*) *assms dvd-eq-mod-eq-0 int-div-less-self of-bool-eq*(*1*,*2*) *sgn-if signed-divide-int-eq-divide-int that zdiv-zminus1-eq-if*)
  qed
  then show *?thesis*
    by *auto*
qed

lemma *int-sdiv-negated-is-minus1* [*simp*]:
    a ≠ *0* ⟹ ((*a* :: *int*) *sdiv b* = − *a*) = (*b* = −*1*)
  using *int-sdiv-same-is-1* [*of* - −*b*]
  using *signed-divide-int-def* **by** *fastforce*

lemma *sdiv-int-range*:
  ‹*a sdiv b* ∈ {− |*a*|..|*a*|}› **for** *a b* :: *int*
  using *zdiv-mono2* [*of* ‹|*a*|› *1* ‹|*b*|›]
  by (*cases* ‹*b* = *0*›; *cases* ‹*sgn b* = *sgn a*›)
     (*auto simp add*: *signed-divide-int-def pos-imp-zdiv-nonneg-iff*
     dest!: *sgn-not-eq-imp intro*: *order-trans* [*of* - *0*])

lemma *smod-int-range*:
  ‹*a smod b* ∈ {− |*b*| + *1*..|*b*| − *1*}›
  if ‹*b* ≠ *0*› **for** *a b* :: *int*
proof −
  define *m n* **where** ‹*m* = *nat* |*a*|› ‹*n* = *nat* |*b*|›
  then have ‹|*a*| = *int m*› ‹|*b*| = *int n*›
    by *simp-all*
  with *that* **have** ‹*n* > *0*›

by *simp*
with *signed-modulo-int-def* [*of a b*] ‹|*a*| = *int m*› ‹|*b*| = *int n*›
show *?thesis*
by (*auto simp add*: *sgn-if diff-le-eq int-one-le-iff-zero-less simp flip*: *of-nat-mod of-nat-diff*)
qed

lemma *smod-int-compares*:
⟦ *0* ≤ *a*; *0* < *b* ⟧ ⟹ (*a* :: *int*) *smod b* < *b*
⟦ *0* ≤ *a*; *0* < *b* ⟧ ⟹ *0* ≤ (*a* :: *int*) *smod b*
⟦ *a* ≤ *0*; *0* < *b* ⟧ ⟹ −*b* < (*a* :: *int*) *smod b*
⟦ *a* ≤ *0*; *0* < *b* ⟧ ⟹ (*a* :: *int*) *smod b* ≤ *0*
⟦ *0* ≤ *a*; *b* < *0* ⟧ ⟹ (*a* :: *int*) *smod b* < − *b*
⟦ *0* ≤ *a*; *b* < *0* ⟧ ⟹ *0* ≤ (*a* :: *int*) *smod b*
⟦ *a* ≤ *0*; *b* < *0* ⟧ ⟹ (*a* :: *int*) *smod b* ≤ *0*
⟦ *a* ≤ *0*; *b* < *0* ⟧ ⟹ *b* ≤ (*a* :: *int*) *smod b*
using *smod-int-range* [**where** *a*=*a* **and** *b*=*b*]
by (*auto simp*: *add1-zle-eq smod-int-alt-def sgn-if*)

lemma *smod-mod-positive*:
⟦ *0* ≤ (*a* :: *int*); *0* ≤ *b* ⟧ ⟹ *a smod b* = *a mod b*
by (*clarsimp simp*: *smod-int-alt-def zsgn-def*)

lemma *minus-sdiv-eq* [*simp*]:
‹− *k sdiv l* = − (*k sdiv l*)› **for** *k l* :: *int*
by (*simp add*: *signed-divide-int-def*)

lemma *sdiv-minus-eq* [*simp*]:
‹*k sdiv* − *l* = − (*k sdiv l*)› **for** *k l* :: *int*
by (*simp add*: *signed-divide-int-def*)

lemma *sdiv-int-numeral-numeral* [*simp*]:
‹*numeral m sdiv numeral n* = *numeral m div* (*numeral n* :: *int*)›
by (*simp add*: *signed-divide-int-def*)

lemma *minus-smod-eq* [*simp*]:
‹− *k smod l* = − (*k smod l*)› **for** *k l* :: *int*
by (*simp add*: *smod-int-alt-def*)

lemma *smod-minus-eq* [*simp*]:
‹*k smod* − *l* = *k smod l*› **for** *k l* :: *int*
by (*simp add*: *smod-int-alt-def*)

lemma *smod-int-numeral-numeral* [*simp*]:
‹*numeral m smod numeral n* = *numeral m mod* (*numeral n* :: *int*)›
by (*simp add*: *smod-int-alt-def*)

end

98 State monad

theory *State-Monad*
imports *Monad-Syntax*
begin

datatype $('s, 'a)$ *state* = *State* (*run-state*: $'s \Rightarrow ('a \times 's)$)

lemma *set-state-iff*: $x \in$ *set-state* $m \longleftrightarrow (\exists\, s\ s'.$ *run-state* $m\ s = (x, s'))$
by (*cases* m) (*simp add*: *prod-set-defs eq-fst-iff*)

lemma *pred-stateI*[*intro*]:
  assumes $\bigwedge a\ s\ s'.$ *run-state* $m\ s = (a, s') \Longrightarrow P\ a$
  shows *pred-state* $P\ m$
proof (*subst state.pred-set*, *rule*)
  fix x
  assume $x \in$ *set-state* m
  then obtain $s\ s'$ **where** *run-state* $m\ s = (x, s')$
    by (*auto simp*: *set-state-iff*)
  with *assms* **show** $P\ x$ **.**
qed

lemma *pred-stateD*[*dest*]:
  assumes *pred-state* $P\ m$ *run-state* $m\ s = (a, s')$
  shows $P\ a$
proof (*rule state.exhaust*[*of* m])
  fix f
  assume $m =$ *State* f
  with *assms* **have** *pred-fun* $(\lambda$-. *True*) (*pred-prod* P *top*) f
    by (*metis state.pred-inject*)
  moreover have $f\ s = (a, s')$
    using *assms* **unfolding** ‹$m =$ -› **by** *auto*
  ultimately show $P\ a$
    unfolding *pred-prod-beta pred-fun-def*
    by (*metis fst-conv*)
qed

lemma *pred-state-run-state*: *pred-state* $P\ m \Longrightarrow P$ (*fst* (*run-state* $m\ s$))
by (*meson pred-stateD prod.exhaust-sel*)

definition *state-io-rel* :: $('s \Rightarrow 's \Rightarrow$ *bool*$) \Rightarrow ('s, 'a)$ *state* $\Rightarrow$ *bool* **where**
state-io-rel $P\ m = (\forall\, s.\ P\ s$ (*snd* (*run-state* $m\ s$)))

lemma *state-io-relI*[*intro*]:
  assumes $\bigwedge a\ s\ s'.$ *run-state* $m\ s = (a, s') \Longrightarrow P\ s\ s'$
  shows *state-io-rel* $P\ m$
using *assms* **unfolding** *state-io-rel-def*
by (*metis prod.collapse*)

lemma *state-io-relD*[*dest*]:
assumes *state-io-rel P m run-state m s* = (*a*, *s*′)
shows *P s s*′
using *assms* **unfolding** *state-io-rel-def*
by (*metis snd-conv*)

lemma *state-io-rel-mono*[*mono*]: *P* ≤ *Q* ⟹ *state-io-rel P* ≤ *state-io-rel Q*
by *blast*

lemma *state-ext*:
assumes ⋀*s*. *run-state m s* = *run-state n s*
shows *m* = *n*
using *assms*
by (*cases m*; *cases n*) *auto*

context begin

qualified definition *return* :: ′*a* ⇒ (′*s*, ′*a*) *state* **where**
return a = *State* (*Pair a*)

lemma *run-state-return*[*simp*]: *run-state* (*return x*) *s* = (*x*, *s*)
unfolding *return-def*
by *simp*

qualified definition *ap* :: (′*s*, ′*a* ⇒ ′*b*) *state* ⇒ (′*s*, ′*a*) *state* ⇒ (′*s*, ′*b*) *state* **where**
ap f x = *State* (λ*s*. *case run-state f s of* (*g*, *s*′) ⇒ *case run-state x s*′ *of* (*y*, *s*′′) ⇒ (*g y*, *s*′′))

lemma *run-state-ap*[*simp*]:
run-state (*ap f x*) *s* = (*case run-state f s of* (*g*, *s*′) ⇒ *case run-state x s*′ *of* (*y*, *s*′′) ⇒ (*g y*, *s*′′))
unfolding *ap-def* **by** *auto*

qualified definition *bind* :: (′*s*, ′*a*) *state* ⇒ (′*a* ⇒ (′*s*, ′*b*) *state*) ⇒ (′*s*, ′*b*) *state* **where**
bind x f = *State* (λ*s*. *case run-state x s of* (*a*, *s*′) ⇒ *run-state* (*f a*) *s*′)

lemma *run-state-bind*[*simp*]:
run-state (*bind x f*) *s* = (*case run-state x s of* (*a*, *s*′) ⇒ *run-state* (*f a*) *s*′)
unfolding *bind-def* **by** *auto*

adhoc-overloading *Monad-Syntax.bind* ⇌ *bind*

lemma *bind-left-identity*[*simp*]: *bind* (*return a*) *f* = *f a*
unfolding *return-def bind-def* **by** *simp*

lemma *bind-right-identity*[*simp*]: *bind m return* = *m*
unfolding *return-def bind-def* **by** *simp*

lemma *bind-assoc*[*simp*]: *bind* (*bind m f*) *g* = *bind m* (λ*x*. *bind* (*f x*) *g*)
unfolding *bind-def* **by** (*auto split*: *prod.splits*)

lemma *bind-predI*[*intro*]:
assumes *pred-state* (λ*x*. *pred-state P* (*f x*)) *m*
shows *pred-state P* (*bind m f*)
apply (*rule pred-stateI*)
unfolding *bind-def*
using *assms* **by** (*auto split*: *prod.splits*)

qualified definition *get* :: ('*s*, '*s*) *state* **where**
get = *State* (λ*s*. (*s*, *s*))

lemma *run-state-get*[*simp*]: *run-state get s* = (*s*, *s*)
unfolding *get-def* **by** *simp*

qualified definition *set* :: '*s* ⇒ ('*s*, *unit*) *state* **where**
set s' = *State* (λ-. ((), *s'*))

lemma *run-state-set*[*simp*]: *run-state* (*set s'*) *s* = ((), *s'*)
unfolding *set-def* **by** *simp*

lemma *get-set*[*simp*]: *bind get set* = *return* ()
unfolding *bind-def get-def set-def return-def*
by *simp*

lemma *set-set*[*simp*]: *bind* (*set s*) (λ-. *set s'*) = *set s'*
unfolding *bind-def set-def*
by *simp*

lemma *get-bind-set*[*simp*]: *bind get* (λ*s*. *bind* (*set s*) (*f s*)) = *bind get* (λ*s*. *f s* ())
unfolding *bind-def get-def set-def*
by *simp*

lemma *get-const*[*simp*]: *bind get* (λ-. *m*) = *m*
unfolding *get-def bind-def*
by *simp*

fun *traverse-list* :: ('*a* ⇒ ('*b*, '*c*) *state*) ⇒ '*a list* ⇒ ('*b*, '*c list*) *state* **where**
traverse-list - [] = *return* [] |
traverse-list f (*x* # *xs*) = *do* {
x ← *f x*;
xs ← *traverse-list f xs*;
return (*x* # *xs*)
}

lemma *traverse-list-app*[*simp*]: *traverse-list f* (*xs* @ *ys*) = *do* {
xs ← *traverse-list f xs*;

ys ← *traverse-list f ys*;
return (*xs* @ *ys*)
}
by (*induction xs*) *auto*

lemma *traverse-comp*[*simp*]: *traverse-list* (*g* ∘ *f*) *xs* = *traverse-list g* (*map f xs*)
by (*induction xs*) *auto*

abbreviation *mono-state* :: ('*s*::*preorder*, '*a*) *state* ⇒ *bool* **where**
mono-state ≡ *state-io-rel* (≤)

abbreviation *strict-mono-state* :: ('*s*::*preorder*, '*a*) *state* ⇒ *bool* **where**
strict-mono-state ≡ *state-io-rel* (<)

corollary *strict-mono-implies-mono*: *strict-mono-state m* ⟹ *mono-state m*
unfolding *state-io-rel-def*
by (*simp add*: *less-imp-le*)

lemma *return-mono*[*simp*, *intro*]: *mono-state* (*return x*)
unfolding *return-def* **by** *auto*

lemma *get-mono*[*simp*, *intro*]: *mono-state get*
unfolding *get-def* **by** *auto*

lemma *put-mono*:
assumes ⋀*x*. *s'* ≥ *x*
shows *mono-state* (*set s'*)
using *assms* **unfolding** *set-def*
by *auto*

lemma *map-mono*[*intro*]: *mono-state m* ⟹ *mono-state* (*map-state f m*)
by (*auto intro*!: *state-io-relI split*: *prod.splits simp*: *map-prod-def state.map-sel*)

lemma *map-strict-mono*[*intro*]: *strict-mono-state m* ⟹ *strict-mono-state* (*map-state f m*)
by (*auto intro*!: *state-io-relI split*: *prod.splits simp*: *map-prod-def state.map-sel*)

lemma *bind-mono-strong*:
assumes *mono-state m*
assumes ⋀*x s s'*. *run-state m s* = (*x*, *s'*) ⟹ *mono-state* (*f x*)
shows *mono-state* (*bind m f*)
unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split*: *prod.splits dest*!: *state-io-relD intro*: *order-trans*)

lemma *bind-strict-mono-strong1*:
assumes *mono-state m*
assumes ⋀*x s s'*. *run-state m s* = (*x*, *s'*) ⟹ *strict-mono-state* (*f x*)
shows *strict-mono-state* (*bind m f*)

unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split*: *prod.splits dest*!: *state-io-relD intro*: *le-less-trans*)

lemma *bind-strict-mono-strong2*:
assumes *strict-mono-state m*
assumes $\bigwedge x\ s\ s'$. *run-state m s* $= (x, s') \Longrightarrow$ *mono-state* (*f x*)
shows *strict-mono-state* (*bind m f*)
unfolding *bind-def*
apply (*rule state-io-relI*)
using *assms* **by** (*auto split*: *prod.splits dest*!: *state-io-relD intro*: *less-le-trans*)

corollary *bind-strict-mono-strong*:
assumes *strict-mono-state m*
assumes $\bigwedge x\ s\ s'$. *run-state m s* $= (x, s') \Longrightarrow$ *strict-mono-state* (*f x*)
shows *strict-mono-state* (*bind m f*)
using *assms* **by** (*auto intro*: *bind-strict-mono-strong1 strict-mono-implies-mono*)

qualified definition *update* :: ('*s* ⇒ '*s*) ⇒ ('*s*, *unit*) *state* **where**
update f = *bind get* (*set* ∘ *f*)

lemma *update-id*[*simp*]: *update* (λ*x*. *x*) = *return* ()
unfolding *update-def return-def get-def set-def bind-def*
by *auto*

lemma *update-comp*[*simp*]: *bind* (*update f*) (λ-. *update g*) = *update* (*g* ∘ *f*)
unfolding *update-def return-def get-def set-def bind-def*
by *auto*

lemma *set-update*[*simp*]: *bind* (*set s*) (λ-. *update f*) = *set* (*f s*)
unfolding *set-def update-def bind-def get-def set-def*
by *simp*

lemma *set-bind-update*[*simp*]: *bind* (*set s*) (λ-. *bind* (*update f*) *g*) = *bind* (*set* (*f s*)) *g*
unfolding *set-def update-def bind-def get-def set-def*
by *simp*

lemma *update-mono*:
assumes $\bigwedge x$. *x* ≤ *f x*
shows *mono-state* (*update f*)
using *assms* **unfolding** *update-def get-def set-def bind-def*
by (*auto intro*!: *state-io-relI*)

lemma *update-strict-mono*:
assumes $\bigwedge x$. *x* < *f x*
shows *strict-mono-state* (*update f*)
using *assms* **unfolding** *update-def get-def set-def bind-def*
by (*auto intro*!: *state-io-relI*)

end

end

theory *Comparator*
imports *Main*
begin

99 Comparators on linear quasi-orders

99.1 Basic properties

datatype *comp* = *Less* | *Equiv* | *Greater*

locale *comparator* =
fixes *cmp* :: ‹'*a* ⇒ '*a* ⇒ *comp*›
assumes *refl* [*simp*]: ‹⋀*a*. *cmp a a* = *Equiv*›
and *trans-equiv*: ‹⋀*a b c*. *cmp a b* = *Equiv* ⟹ *cmp b c* = *Equiv* ⟹ *cmp a c* = *Equiv*›
assumes *trans-less*: ‹*cmp a b* = *Less* ⟹ *cmp b c* = *Less* ⟹ *cmp a c* = *Less*›
and *greater-iff-sym-less*: ‹⋀*b a*. *cmp b a* = *Greater* ⟷ *cmp a b* = *Less*›
begin

Dual properties

lemma *trans-greater*:
‹*cmp a c* = *Greater*› **if** ‹*cmp a b* = *Greater*› ‹*cmp b c* = *Greater*›
using *that greater-iff-sym-less trans-less* **by** *blast*

lemma *less-iff-sym-greater*:
‹*cmp b a* = *Less* ⟷ *cmp a b* = *Greater*›
by (*simp add*: *greater-iff-sym-less*)

The equivalence part

lemma *sym*:
‹*cmp b a* = *Equiv* ⟷ *cmp a b* = *Equiv*›
by (*metis* (*full-types*) *comp.exhaust greater-iff-sym-less*)

lemma *reflp*:
‹*reflp* (λ*a b*. *cmp a b* = *Equiv*)›
by (*rule reflpI*) *simp*

lemma *symp*:
‹*symp* (λ*a b*. *cmp a b* = *Equiv*)›
by (*rule sympI*) (*simp add*: *sym*)

lemma *transp*:
‹*transp* (λ*a b*. *cmp a b* = *Equiv*)›

by (*rule transpI*) (*fact trans-equiv*)

lemma *equivp*:
‹*equivp* (λ*a b*. *cmp a b* = *Equiv*)›
using *reflp symp transp* **by** (*rule equivpI*)

The strict part

lemma *irreflp-less*:
‹*irreflp* (λ*a b*. *cmp a b* = *Less*)›
by (*rule irreflpI*) *simp*

lemma *irreflp-greater*:
‹*irreflp* (λ*a b*. *cmp a b* = *Greater*)›
by (*rule irreflpI*) *simp*

lemma *asym-less*:
‹*cmp b a* ≠ *Less*› **if** ‹*cmp a b* = *Less*›
using *that greater-iff-sym-less* **by** *force*

lemma *asym-greater*:
‹*cmp b a* ≠ *Greater*› **if** ‹*cmp a b* = *Greater*›
using *that greater-iff-sym-less* **by** *force*

lemma *asymp-less*:
‹*asymp* (λ*a b*. *cmp a b* = *Less*)›
using *irreflp-less* **by** (*auto dest*: *asym-less*)

lemma *asymp-greater*:
‹*asymp* (λ*a b*. *cmp a b* = *Greater*)›
using *irreflp-greater* **by** (*auto dest*: *asym-greater*)

lemma *trans-equiv-less*:
‹*cmp a c* = *Less*› **if** ‹*cmp a b* = *Equiv*› **and** ‹*cmp b c* = *Less*›
using *that*
by (*metis* (*full-types*) *comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-less-equiv*:
‹*cmp a c* = *Less*› **if** ‹*cmp a b* = *Less*› **and** ‹*cmp b c* = *Equiv*›
using *that*
by (*metis* (*full-types*) *comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-equiv-greater*:
‹*cmp a c* = *Greater*› **if** ‹*cmp a b* = *Equiv*› **and** ‹*cmp b c* = *Greater*›
using *that* **by** (*simp add*: *sym* [*of a b*] *greater-iff-sym-less trans-less-equiv*)

lemma *trans-greater-equiv*:
‹*cmp a c* = *Greater*› **if** ‹*cmp a b* = *Greater*› **and** ‹*cmp b c* = *Equiv*›
using *that* **by** (*simp add*: *sym* [*of b c*] *greater-iff-sym-less trans-equiv-less*)

lemma *transp-less*:
‹*transp* (λ*a b*. *cmp a b* = *Less*)›
by (*rule transpI*) (*fact trans-less*)

lemma *transp-greater*:
‹*transp* (λ*a b*. *cmp a b* = *Greater*)›
by (*rule transpI*) (*fact trans-greater*)

The reflexive part

lemma *reflp-not-less*:
‹*reflp* (λ*a b*. *cmp a b* ≠ *Less*)›
by (*rule reflpI*) *simp*

lemma *reflp-not-greater*:
‹*reflp* (λ*a b*. *cmp a b* ≠ *Greater*)›
by (*rule reflpI*) *simp*

lemma *quasisym-not-less*:
‹*cmp a b* = *Equiv*› **if** ‹*cmp a b* ≠ *Less*› **and** ‹*cmp b a* ≠ *Less*›
using *that comp.exhaust greater-iff-sym-less* **by** *auto*

lemma *quasisym-not-greater*:
‹*cmp a b* = *Equiv*› **if** ‹*cmp a b* ≠ *Greater*› **and** ‹*cmp b a* ≠ *Greater*›
using *that comp.exhaust greater-iff-sym-less* **by** *auto*

lemma *trans-not-less*:
‹*cmp a c* ≠ *Less*› **if** ‹*cmp a b* ≠ *Less*› ‹*cmp b c* ≠ *Less*›
using *that* **by** (*metis comp.exhaust greater-iff-sym-less trans-equiv trans-less*)

lemma *trans-not-greater*:
‹*cmp a c* ≠ *Greater*› **if** ‹*cmp a b* ≠ *Greater*› ‹*cmp b c* ≠ *Greater*›
using *that greater-iff-sym-less trans-not-less* **by** *blast*

lemma *transp-not-less*:
‹*transp* (λ*a b*. *cmp a b* ≠ *Less*)›
by (*rule transpI*) (*fact trans-not-less*)

lemma *transp-not-greater*:
‹*transp* (λ*a b*. *cmp a b* ≠ *Greater*)›
by (*rule transpI*) (*fact trans-not-greater*)

Substitution under equivalences

lemma *equiv-subst-left*:
‹*cmp z y* = *comp* ⟷ *cmp x y* = *comp*› **if** ‹*cmp z x* = *Equiv*› **for** *comp*
proof −
from *that* **have** ‹*cmp x z* = *Equiv*›
by (*simp add*: *sym*)
with *that* **show** *?thesis*
by (*cases comp*) (*auto intro*: *trans-equiv trans-equiv-less trans-equiv-greater*)

qed

lemma *equiv-subst-right*:
‹*cmp x z = comp* ⟷ *cmp x y = comp*› **if** ‹*cmp z y = Equiv*› **for** *comp*
proof −
from *that* **have** ‹*cmp y z = Equiv*›
by (*simp add*: *sym*)
with *that* **show** *?thesis*
by (*cases comp*) (*auto intro*: *trans-equiv trans-less-equiv trans-greater-equiv*)
qed

end

typedef *'a comparator* = ‹{*cmp* :: *'a* ⇒ *'a* ⇒ *comp*. *comparator cmp*}›
morphisms *compare Abs-comparator*
proof −
have ‹*comparator* (λ- -. *Equiv*)›
by *standard simp-all*
then show *?thesis*
by *auto*
qed

setup-lifting *type-definition-comparator*

global-interpretation *compare*: *comparator* ‹*compare cmp*›
using *compare* [*of cmp*] **by** *simp*

lift-definition *flat* :: ‹*'a comparator*›
is ‹λ- -. *Equiv*› **by** *standard simp-all*

instantiation *comparator* :: (*linorder*) *default*
begin

lift-definition *default-comparator* :: ‹*'a comparator*›
is ‹λ*x y*. *if x < y then Less else if x > y then Greater else Equiv*›
by *standard* (*auto split*: *if-splits*)

instance ..

end

lemma *compare-default-eq-Less-iff* [*simp*]:
‹*compare default x y = Less* ⟷ *x < y*›
by *transfer simp*

lemma *compare-default-eq-Equiv-iff* [*simp*]:
‹*compare default x y = Equiv* ⟷ *x = y*›
by *transfer auto*

lemma *compare-default-eq-Greater-iff* [*simp*]:
‹*compare default x y = Greater* ⟷ *x > y*›
by *transfer auto*

A rudimentary quickcheck setup

instantiation *comparator* :: (*enum*) *equal*
begin

lift-definition *equal-comparator* :: ‹*'a comparator* ⇒ *'a comparator* ⇒ *bool*›
is ‹*λf g. ∀ x ∈ set Enum.enum. f x = g x*› **.**

instance
by (*standard*; *transfer*) (*auto simp add*: *enum-UNIV*)

end

lemma [*code nbe*]:
‹*HOL.equal* (*cmp* :: *'a*::*enum comparator*) *cmp* ⟷ *True*›
by (*fact equal-refl*)

lemma [*code*]:
‹*HOL.equal cmp1 cmp2* ⟷ *Enum.enum-all* (*λx. compare cmp1 x = compare cmp2 x*)›
by *transfer* (*simp add*: *enum-UNIV*)

instantiation *comparator* :: ({*linorder*, *typerep*}) *full-exhaustive*
begin

definition *full-exhaustive-comparator* ::
‹(*'a comparator* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*)
⇒ *natural* ⇒ (*bool* × *term list*) *option*›
where ‹*full-exhaustive-comparator f s* =
Quickcheck-Exhaustive.orelse
(*f* (*flat*, (*λu. Code-Evaluation.Const* (*STR ''Comparator.flat''*) *TYPEREP*(*'a comparator*))))
(*f* (*default*, (*λu. Code-Evaluation.Const* (*STR ''HOL.default-class.default''*) *TYPEREP*(*'a comparator*))))›

instance ..

end

99.2 Fundamental comparator combinators

lift-definition *reversed* :: ‹*'a comparator* ⇒ *'a comparator*›
is ‹*λcmp a b. cmp b a*›
proof −
fix *cmp* :: ‹*'a* ⇒ *'a* ⇒ *comp*›
assume ‹*comparator cmp*›
then interpret *comparator cmp* **.**

show ‹*comparator* (λ*a b*. *cmp b a*)›
by *standard* (*auto intro*: *trans-equiv trans-less simp*: *greater-iff-sym-less*)
qed

lemma *compare-reversed-apply* [*simp*]:
‹*compare* (*reversed cmp*) *x y* = *compare cmp y x*›
by *transfer simp*

lift-definition *key* :: ‹('*b* ⇒ '*a*) ⇒ '*a comparator* ⇒ '*b comparator*›
is ‹λ*f cmp a b*. *cmp* (*f a*) (*f b*)›
proof −
fix *cmp* :: ‹'*a* ⇒ '*a* ⇒ *comp*› **and** *f* :: ‹'*b* ⇒ '*a*›
assume ‹*comparator cmp*›
then interpret *comparator cmp* **.**
show ‹*comparator* (λ*a b*. *cmp* (*f a*) (*f b*))›
by *standard* (*auto intro*: *trans-equiv trans-less simp*: *greater-iff-sym-less*)
qed

lemma *compare-key-apply* [*simp*]:
‹*compare* (*key f cmp*) *x y* = *compare cmp* (*f x*) (*f y*)›
by *transfer simp*

lift-definition *prod-lex* :: ‹'*a comparator* ⇒ '*b comparator* ⇒ ('*a* × '*b*) *comparator*›
is ‹λ*f g* (*a*, *c*) (*b*, *d*). *case f a b of Less* ⇒ *Less* | *Equiv* ⇒ *g c d* | *Greater* ⇒ *Greater*›
proof −
fix *f* :: ‹'*a* ⇒ '*a* ⇒ *comp*› **and** *g* :: ‹'*b* ⇒ '*b* ⇒ *comp*›
assume ‹*comparator f*›
then interpret *f*: *comparator f* **.**
assume ‹*comparator g*›
then interpret *g*: *comparator g* **.**
define *h* **where** ‹*h* = (λ(*a*, *c*) (*b*, *d*). *case f a b of Less* ⇒ *Less* | *Equiv* ⇒ *g c d* | *Greater* ⇒ *Greater*)›
then have *h-apply* [*simp*]: ‹*h* (*a*, *c*) (*b*, *d*) = (*case f a b of Less* ⇒ *Less* | *Equiv* ⇒ *g c d* | *Greater* ⇒ *Greater*)› **for** *a b c d*
by *simp*
have *h-equiv*: ‹*h p q* = *Equiv* ⟷ *f* (*fst p*) (*fst q*) = *Equiv* ∧ *g* (*snd p*) (*snd q*) = *Equiv*› **for** *p q*
by (*cases p*; *cases q*) (*simp split*: *comp.split*)
have *h-less*: ‹*h p q* = *Less* ⟷ *f* (*fst p*) (*fst q*) = *Less* ∨ *f* (*fst p*) (*fst q*) = *Equiv* ∧ *g* (*snd p*) (*snd q*) = *Less*› **for** *p q*
by (*cases p*; *cases q*) (*simp split*: *comp.split*)
have *h-greater*: ‹*h p q* = *Greater* ⟷ *f* (*fst p*) (*fst q*) = *Greater* ∨ *f* (*fst p*) (*fst q*) = *Equiv* ∧ *g* (*snd p*) (*snd q*) = *Greater*› **for** *p q*
by (*cases p*; *cases q*) (*simp split*: *comp.split*)
have ‹*comparator h*›
apply *standard*
apply (*simp-all add*: *h-equiv h-less h-greater f.greater-iff-sym-less g.greater-iff-sym-less f.sym g.sym*)

apply (*auto intro*: *f.trans-equiv g.trans-equiv f.trans-less g.trans-less f.trans-less-equiv f.trans-equiv-less*)
done
then show ‹*comparator* (λ(*a*, *c*) (*b*, *d*). *case f a b of Less ⇒ Less*
| *Equiv ⇒ g c d*
| *Greater ⇒ Greater*)›
by (*simp add*: *h-def*)
qed

lemma *compare-prod-lex-apply*:
‹*compare* (*prod-lex cmp1 cmp2*) *p q* =
(*case compare* (*key fst cmp1*) *p q of Less ⇒ Less* | *Equiv ⇒ compare* (*key snd cmp2*) *p q* | *Greater ⇒ Greater*)›
by *transfer* (*simp add*: *split-def*)

99.3 Direct implementations for linear orders on selected types

definition *comparator-bool* :: ‹*bool comparator*›
where [*simp*, *code-abbrev*]: ‹*comparator-bool* = *default*›

lemma *compare-comparator-bool* [*code abstract*]:
‹*compare comparator-bool* = (λ*p q*.
if p then if q then Equiv else Greater
else if q then Less else Equiv)›
by (*auto simp add*: *fun-eq-iff*)

definition *raw-comparator-nat* :: ‹*nat ⇒ nat ⇒ comp*›
where [*simp*]: ‹*raw-comparator-nat* = *compare default*›

lemma *default-comparator-nat* [*simp*, *code*]:
‹*raw-comparator-nat* (*0*::*nat*) *0* = *Equiv*›
‹*raw-comparator-nat* (*Suc m*) *0* = *Greater*›
‹*raw-comparator-nat 0* (*Suc n*) = *Less*›
‹*raw-comparator-nat* (*Suc m*) (*Suc n*) = *raw-comparator-nat m n*›
by (*transfer*; *simp*)+

definition *comparator-nat* :: ‹*nat comparator*›
where [*simp*, *code-abbrev*]: ‹*comparator-nat* = *default*›

lemma *compare-comparator-nat* [*code abstract*]:
‹*compare comparator-nat* = *raw-comparator-nat*›
by *simp*

definition *comparator-linordered-group* :: ‹'*a*::*linordered-ab-group-add comparator*›
where [*simp*, *code-abbrev*]: ‹*comparator-linordered-group* = *default*›

lemma *comparator-linordered-group* [*code abstract*]:
‹*compare comparator-linordered-group* = (λ*a b*.

let c = a − b in if c < 0 then Less
else if c = 0 then Equiv else Greater)›
proof (*rule ext*)+
fix *a b* :: *'a*
show ‹*compare comparator-linordered-group a b* =
(*let c = a − b in if c < 0 then Less*
else if c = 0 then Equiv else Greater)›
by (*simp add*: *Let-def not-less*) (*transfer*; *auto*)
qed

end

theory *Sorting-Algorithms*
imports *Main Multiset Comparator*
begin

100 Stably sorted lists

abbreviation (*input*) *stable-segment* :: ‹*'a comparator* ⇒ *'a* ⇒ *'a list* ⇒ *'a list*›
where ‹*stable-segment cmp x* ≡ *filter* (λ*y*. *compare cmp x y* = *Equiv*)›

fun *sorted* :: ‹*'a comparator* ⇒ *'a list* ⇒ *bool*›
where *sorted-Nil*: ‹*sorted cmp* [] ⟷ *True*›
| *sorted-single*: ‹*sorted cmp* [*x*] ⟷ *True*›
| *sorted-rec*: ‹*sorted cmp* (*y* # *x* # *xs*) ⟷ *compare cmp y x* ≠ *Greater* ∧ *sorted cmp* (*x* # *xs*)›

lemma *sorted-ConsI*:
‹*sorted cmp* (*x* # *xs*)› **if** ‹*sorted cmp xs*›
and ‹⋀*y ys*. *xs* = *y* # *ys* ⟹ *compare cmp x y* ≠ *Greater*›
using *that* **by** (*cases xs*) *simp-all*

lemma *sorted-Cons-imp-sorted*:
‹*sorted cmp xs*› **if** ‹*sorted cmp* (*x* # *xs*)›
using *that* **by** (*cases xs*) *simp-all*

lemma *sorted-Cons-imp-not-less*:
‹*compare cmp y x* ≠ *Greater*› **if** ‹*sorted cmp* (*y* # *xs*)›
and ‹*x* ∈ *set xs*›
using *that* **by** (*induction xs arbitrary*: *y*) (*auto dest*: *compare.trans-not-greater*)

lemma *sorted-induct* [*consumes 1*, *case-names Nil Cons*, *induct pred*: *sorted*]:
‹*P xs*› **if** ‹*sorted cmp xs*› **and** ‹*P* []›
and ∗: ‹⋀*x xs*. *sorted cmp xs* ⟹ *P xs*
⟹ (⋀*y*. *y* ∈ *set xs* ⟹ *compare cmp x y* ≠ *Greater*) ⟹ *P* (*x* # *xs*)›
using ‹*sorted cmp xs*› **proof** (*induction xs*)
case *Nil*
show *?case*

```
    by (rule ‹P []›)
next
  case (Cons x xs)
  from ‹sorted cmp (x # xs)› have ‹sorted cmp xs›
    by (cases xs) simp-all
  moreover have ‹P xs› using ‹sorted cmp xs›
    by (rule Cons.IH)
  moreover have ‹compare cmp x y ≠ Greater› if ‹y ∈ set xs› for y
  using that ‹sorted cmp (x # xs)› proof (induction xs)
    case Nil
    then show ?case
      by simp
  next
    case (Cons z zs)
    then show ?case
    proof (cases zs)
      case Nil
      with Cons.prems show ?thesis
        by simp
    next
      case (Cons w ws)
       with Cons.prems have ‹compare cmp z w ≠ Greater› ‹compare cmp x z ≠
Greater›
        by auto
      then have ‹compare cmp x w ≠ Greater›
        by (auto dest: compare.trans-not-greater)
      with Cons show ?thesis
        using Cons.prems Cons.IH by auto
    qed
  qed
  ultimately show ?case
    by (rule ∗)
qed

lemma sorted-induct-remove1 [consumes 1, case-names Nil minimum]:
  ‹P xs› if ‹sorted cmp xs› and ‹P []›
    and ∗: ‹⋀x xs. sorted cmp xs ⟹ P (remove1 x xs)
      ⟹ x ∈ set xs ⟹ hd (stable-segment cmp x xs) = x ⟹ (⋀y. y ∈ set xs ⟹
compare cmp x y ≠ Greater)
    ⟹ P xs›
using ‹sorted cmp xs› proof (induction xs)
  case Nil
  show ?case
    by (rule ‹P []›)
next
  case (Cons x xs)
  then have ‹sorted cmp (x # xs)›
    by (simp add: sorted-ConsI)
  moreover note Cons.IH
```

```
  moreover have ‹⋀y. compare cmp x y = Greater ⟹ y ∈ set xs ⟹ False›
    using Cons.hyps by simp
  ultimately show ?case
    by (auto intro!: * [of ‹x # xs› x]) blast
qed

lemma sorted-remove1:
  ‹sorted cmp (remove1 x xs)› if ‹sorted cmp xs›
proof (cases ‹x ∈ set xs›)
  case False
  with that show ?thesis
    by (simp add: remove1-idem)
next
  case True
  with that show ?thesis proof (induction xs)
    case Nil
    then show ?case
      by simp
  next
    case (Cons y ys)
    show ?case proof (cases ‹x = y›)
      case True
      with Cons.hyps show ?thesis
        by simp
    next
      case False
      then have ‹sorted cmp (remove1 x ys)›
        using Cons.IH Cons.prems by auto
      then have ‹sorted cmp (y # remove1 x ys)›
      proof (rule sorted-ConsI)
        fix z zs
        assume ‹remove1 x ys = z # zs›
        with ‹x ≠ y› have ‹z ∈ set ys›
          using notin-set-remove1 [of z ys x] by auto
        then show ‹compare cmp y z ≠ Greater›
          by (rule Cons.hyps(2))
      qed
      with False show ?thesis
        by simp
    qed
  qed
qed

lemma sorted-stable-segment:
  ‹sorted cmp (stable-segment cmp x xs)›
proof (induction xs)
  case Nil
  show ?case
    by simp
```

next
case (*Cons y ys*)
then show *?case*
by (*auto intro*!: *sorted-ConsI simp add*: *filter-eq-Cons-iff compare.sym*)
(*auto dest*: *compare.trans-equiv simp add*: *compare.sym compare.greater-iff-sym-less*)

qed

primrec *insort* :: ‹*'a comparator* ⇒ *'a* ⇒ *'a list* ⇒ *'a list*›
where ‹*insort cmp y* [] = [*y*]›
| ‹*insort cmp y* (*x* # *xs*) = (*if compare cmp y x* ≠ *Greater*
then y # *x* # *xs*
else x # *insort cmp y xs*)›

lemma *mset-insort* [*simp*]:
‹*mset* (*insort cmp x xs*) = *add-mset x* (*mset xs*)›
by (*induction xs*) *simp-all*

lemma *length-insort* [*simp*]:
‹*length* (*insort cmp x xs*) = *Suc* (*length xs*)›
by (*induction xs*) *simp-all*

lemma *sorted-insort*:
‹*sorted cmp* (*insort cmp x xs*)› **if** ‹*sorted cmp xs*›
using *that* **proof** (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons y ys*)
then show *?case* **by** (*cases ys*)
(*auto*, *simp-all add*: *compare.greater-iff-sym-less*)
qed

lemma *stable-insort-equiv*:
‹*stable-segment cmp y* (*insort cmp x xs*) = *x* # *stable-segment cmp y xs*›
if ‹*compare cmp y x* = *Equiv*›
proof (*induction xs*)
case *Nil*
from *that* **show** *?case*
by *simp*
next
case (*Cons z xs*)
moreover from *that* **have** ‹*compare cmp y z* = *Equiv* ⟹ *compare cmp z x* = *Equiv*›
by (*auto intro*: *compare.trans-equiv simp add*: *compare.sym*)
ultimately show *?case*
using *that* **by** (*auto simp add*: *compare.greater-iff-sym-less*)
qed

lemma *stable-insort-not-equiv*:
‹*stable-segment cmp y* (*insort cmp x xs*) = *stable-segment cmp y xs*›
if ‹*compare cmp y x* ≠ *Equiv*›
using *that* **by** (*induction xs*) *simp-all*

lemma *remove1-insort-same-eq* [*simp*]:
‹*remove1 x* (*insort cmp x xs*) = *xs*›
by (*induction xs*) *simp-all*

lemma *insort-eq-ConsI*:
‹*insort cmp x xs* = *x* # *xs*›
if ‹*sorted cmp xs*› ‹⋀*y*. *y* ∈ *set xs* ⟹ *compare cmp x y* ≠ *Greater*›
using *that* **by** (*induction xs*) (*simp-all add*: *compare.greater-iff-sym-less*)

lemma *remove1-insort-not-same-eq* [*simp*]:
‹*remove1 y* (*insort cmp x xs*) = *insort cmp x* (*remove1 y xs*)›
if ‹*sorted cmp xs*› ‹*x* ≠ *y*›
using *that* **proof** (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons z zs*)
show *?case*
proof (*cases* ‹*compare cmp x z* = *Greater*›)
case *True*
with *Cons* **show** *?thesis*
by *simp*
next
case *False*
then have ‹*compare cmp x y* ≠ *Greater*› **if** ‹*y* ∈ *set zs*› **for** *y*
using *that Cons.hyps*
by (*auto dest*: *compare.trans-not-greater*)
with *Cons* **show** *?thesis*
by (*simp add*: *insort-eq-ConsI*)
qed
qed

lemma *insort-remove1-same-eq*:
‹*insort cmp x* (*remove1 x xs*) = *xs*›
if ‹*sorted cmp xs*› **and** ‹*x* ∈ *set xs*› **and** ‹*hd* (*stable-segment cmp x xs*) = *x*›
using *that* **proof** (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons y ys*)
then have ‹*compare cmp x y* ≠ *Less*›

```
    by (auto simp add: compare.greater-iff-sym-less)
  then consider ‹compare cmp x y = Greater› | ‹compare cmp x y = Equiv›
    by (cases ‹compare cmp x y›) auto
  then show ?case proof cases
    case 1
    with Cons.prems Cons.IH show ?thesis
      by auto
  next
    case 2
    with Cons.prems have ‹x = y›
      by simp
    with Cons.hyps show ?thesis
      by (simp add: insort-eq-ConsI)
  qed
qed

lemma sorted-append-iff:
  ‹sorted cmp (xs @ ys) ⟷ sorted cmp xs ∧ sorted cmp ys
    ∧ (∀x ∈ set xs. ∀y ∈ set ys. compare cmp x y ≠ Greater)› (is ‹?P ⟷ ?R ∧
?S ∧ ?Q›)
proof
  assume ?P
  have ?R
    using ‹?P› by (induction xs)
      (auto simp add: sorted-Cons-imp-not-less,
        auto simp add: sorted-Cons-imp-sorted intro: sorted-ConsI)
  moreover have ?S
    using ‹?P› by (induction xs) (auto dest: sorted-Cons-imp-sorted)
  moreover have ?Q
    using ‹?P› by (induction xs) (auto simp add: sorted-Cons-imp-not-less,
      simp add: sorted-Cons-imp-sorted)
  ultimately show ‹?R ∧ ?S ∧ ?Q›
    by simp
next
  assume ‹?R ∧ ?S ∧ ?Q›
  then have ?R ?S ?Q
    by simp-all
  then show ?P
    by (induction xs)
      (auto simp add: append-eq-Cons-conv intro!: sorted-ConsI)
qed

definition sort :: ‹'a comparator ⇒ 'a list ⇒ 'a list›
  where ‹sort cmp xs = foldr (insort cmp) xs []›

lemma sort-simps [simp]:
  ‹sort cmp [] = []›
  ‹sort cmp (x # xs) = insort cmp x (sort cmp xs)›
  by (simp-all add: sort-def)
```

lemma *mset-sort* [*simp*]:
‹*mset* (*sort cmp xs*) = *mset xs*›
by (*induction xs*) *simp-all*

lemma *length-sort* [*simp*]:
‹*length* (*sort cmp xs*) = *length xs*›
by (*induction xs*) *simp-all*

lemma *sorted-sort* [*simp*]:
‹*sorted cmp* (*sort cmp xs*)›
by (*induction xs*) (*simp-all add*: *sorted-insort*)

lemma *stable-sort*:
‹*stable-segment cmp x* (*sort cmp xs*) = *stable-segment cmp x xs*›
by (*induction xs*) (*simp-all add*: *stable-insort-equiv stable-insort-not-equiv*)

lemma *sort-remove1-eq* [*simp*]:
‹*sort cmp* (*remove1 x xs*) = *remove1 x* (*sort cmp xs*)›
by (*induction xs*) *simp-all*

lemma *set-insort* [*simp*]:
‹*set* (*insort cmp x xs*) = *insert x* (*set xs*)›
by (*induction xs*) *auto*

lemma *set-sort* [*simp*]:
‹*set* (*sort cmp xs*) = *set xs*›
by (*induction xs*) *auto*

lemma *sort-eqI*:
‹*sort cmp ys* = *xs*›
if *permutation*: ‹*mset ys* = *mset xs*›
and *sorted*: ‹*sorted cmp xs*›
and *stable*: ‹⋀*y*. *y* ∈ *set ys* ⟹
stable-segment cmp y ys = *stable-segment cmp y xs*›
proof −
have *stable′*: ‹*stable-segment cmp y ys* =
stable-segment cmp y xs› **for** *y*
proof (*cases* ‹∃ *x*∈*set ys*. *compare cmp y x* = *Equiv*›)
case *True*
then obtain *z* **where** ‹*z* ∈ *set ys*› **and** ‹*compare cmp y z* = *Equiv*›
by *auto*
then have ‹*compare cmp y x* = *Equiv* ⟷ *compare cmp z x* = *Equiv*› **for** *x*
by (*meson compare.sym compare.trans-equiv*)
moreover have ‹*stable-segment cmp z ys* =
stable-segment cmp z xs›
using ‹*z* ∈ *set ys*› **by** (*rule stable*)
ultimately show *?thesis*
by *simp*

```
  next
    case False
    moreover from permutation have ‹set ys = set xs›
      by (rule mset-eq-setD)
    ultimately show ?thesis
      by simp
  qed
  show ?thesis
  using sorted permutation stable' proof (induction xs arbitrary: ys rule: sorted-induct-remove1)
    case Nil
    then show ?case
      by simp
  next
    case (minimum x xs)
    from ‹mset ys = mset xs› have ys: ‹set ys = set xs›
      by (rule mset-eq-setD)
    then have ‹compare cmp x y ≠ Greater› if ‹y ∈ set ys› for y
      using that minimum.hyps by simp
    from minimum.prems have stable: ‹stable-segment cmp x ys = stable-segment
cmp x xs›
      by simp
    have ‹sort cmp (remove1 x ys) = remove1 x xs›
      by (rule minimum.IH) (simp-all add: minimum.prems filter-remove1)
    then have ‹remove1 x (sort cmp ys) = remove1 x xs›
      by simp
    then have ‹insort cmp x (remove1 x (sort cmp ys)) =
      insort cmp x (remove1 x xs)›
      by simp
    also from minimum.hyps ys stable have ‹insort cmp x (remove1 x (sort cmp
ys)) = sort cmp ys›
      by (simp add: stable-sort insort-remove1-same-eq)
    also from minimum.hyps have ‹insort cmp x (remove1 x xs) = xs›
      by (simp add: insort-remove1-same-eq)
    finally show ?case .
  qed
qed

lemma filter-insort:
  ‹filter P (insort cmp x xs) = insort cmp x (filter P xs)›
    if ‹sorted cmp xs› and ‹P x›
  using that by (induction xs)
    (auto simp add: compare.trans-not-greater insort-eq-ConsI)

lemma filter-insort-triv:
  ‹filter P (insort cmp x xs) = filter P xs›
    if ‹¬ P x›
  using that by (induction xs) simp-all

lemma filter-sort:
```

‹*filter P* (*sort cmp xs*) = *sort cmp* (*filter P xs*)›
by (*induction xs*) (*auto simp add*: *filter-insort filter-insort-triv*)

101 Alternative sorting algorithms

101.1 Quicksort

definition *quicksort* :: ‹'*a comparator* ⇒ '*a list* ⇒ '*a list*›
where *quicksort-is-sort* [*simp*]: ‹*quicksort* = *sort*›

lemma *sort-by-quicksort*:
‹*sort* = *quicksort*›
by *simp*

lemma *sort-by-quicksort-rec*:
‹*sort cmp xs* = *sort cmp* [*x*←*xs*. *compare cmp x* (*xs* ! (*length xs div 2*)) = *Less*]
@ *stable-segment cmp* (*xs* ! (*length xs div 2*)) *xs*
@ *sort cmp* [*x*←*xs*. *compare cmp x* (*xs* ! (*length xs div 2*)) = *Greater*]› (**is** ‹- = *?rhs*›)
proof (*rule sort-eqI*)
show ‹*mset xs* = *mset ?rhs*›
by (*rule multiset-eqI*) (*auto simp add*: *compare.sym intro*: *comp.exhaust*)
next
show ‹*sorted cmp ?rhs*›
by (*auto simp add*: *sorted-append-iff sorted-stable-segment compare.equiv-subst-right dest*: *compare.trans-greater*)
next
let *?pivot* = ‹*xs* ! (*length xs div 2*)›
fix *l*
have ‹*compare cmp x ?pivot* = *comp* ∧ *compare cmp l x* = *Equiv*
⟷ *compare cmp l ?pivot* = *comp* ∧ *compare cmp l x* = *Equiv*› **for** *x comp*
proof −
have ‹*compare cmp x ?pivot* = *comp* ⟷ *compare cmp l ?pivot* = *comp*›
if ‹*compare cmp l x* = *Equiv*›
using *that* **by** (*simp add*: *compare.equiv-subst-left compare.sym*)
then show *?thesis* **by** *blast*
qed
then show ‹*stable-segment cmp l xs* = *stable-segment cmp l ?rhs*›
by (*simp add*: *stable-sort compare.sym* [*of* - *?pivot*])
(*cases* ‹*compare cmp l ?pivot*›, *simp-all*)
qed

context
begin

qualified definition *partition* :: ‹'*a comparator* ⇒ '*a* ⇒ '*a list* ⇒ '*a list* × '*a list* × '*a list*›
where ‹*partition cmp pivot xs* =
([*x* ← *xs*. *compare cmp x pivot* = *Less*], *stable-segment cmp pivot xs*, [*x* ← *xs*.

compare cmp x pivot = Greater])›

qualified lemma *partition-code* [*code*]:
‹*partition cmp pivot* [] = ([], [], [])›
‹*partition cmp pivot* (*x* # *xs*) =
(*let* (*lts*, *eqs*, *gts*) = *partition cmp pivot xs*
in case compare cmp x pivot of
Less ⇒ (*x* # *lts*, *eqs*, *gts*)
| *Equiv* ⇒ (*lts*, *x* # *eqs*, *gts*)
| *Greater* ⇒ (*lts*, *eqs*, *x* # *gts*))›
using *comp.exhaust* **by** (*auto simp add*: *partition-def Let-def compare.sym* [*of* - *pivot*])

lemma *quicksort-code* [*code*]:
‹*quicksort cmp xs* =
(*case xs of*
[] ⇒ []
| [*x*] ⇒ *xs*
| [*x*, *y*] ⇒ (*if compare cmp x y* ≠ *Greater then xs else* [*y*, *x*])
| - ⇒
let (*lts*, *eqs*, *gts*) = *partition cmp* (*xs* ! (*length xs div 2*)) *xs*
in quicksort cmp lts @ *eqs* @ *quicksort cmp gts*)›
proof (*cases* ‹*length xs* ≥ *3*›)
case *False*
then have ‹*length xs* ∈ {*0*, *1*, *2*}›
by (*auto simp add*: *not-le le-less less-antisym*)
then consider ‹*xs* = []› | *x* **where** ‹*xs* = [*x*]› | *x y* **where** ‹*xs* = [*x*, *y*]›
by (*auto simp add*: *length-Suc-conv numeral-2-eq-2*)
then show *?thesis*
by *cases simp-all*
next
case *True*
then obtain *x y z zs* **where** ‹*xs* = *x* # *y* # *z* # *zs*›
by (*metis le-0-eq length-0-conv length-Cons list.exhaust not-less-eq-eq numeral-3-eq-3*)
moreover have ‹*quicksort cmp xs* =
(*let* (*lts*, *eqs*, *gts*) = *partition cmp* (*xs* ! (*length xs div 2*)) *xs*
in quicksort cmp lts @ *eqs* @ *quicksort cmp gts*)›
using *sort-by-quicksort-rec* [*of cmp xs*] **by** (*simp add*: *partition-def*)
ultimately show *?thesis*
by *simp*
qed

end

101.2 Mergesort

definition *mergesort* :: ‹*'a comparator* ⇒ *'a list* ⇒ *'a list*›
where *mergesort-is-sort* [*simp*]: ‹*mergesort* = *sort*›

lemma *sort-by-mergesort*:
‹*sort* = *mergesort*›
by *simp*

context
fixes *cmp* :: ‹'*a comparator*›
begin

qualified function *merge* :: ‹'*a list* ⇒ '*a list* ⇒ '*a list*›
where ‹*merge* [] *ys* = *ys*›
| ‹*merge xs* [] = *xs*›
| ‹*merge* (*x* # *xs*) (*y* # *ys*) = (*if compare cmp x y* = *Greater*
then y # *merge* (*x* # *xs*) *ys else x* # *merge xs* (*y* # *ys*))›
by *pat-completeness auto*

qualified termination by *lexicographic-order*

lemma *mset-merge*:
‹*mset* (*merge xs ys*) = *mset xs* + *mset ys*›
by (*induction xs ys rule*: *merge.induct*) *simp-all*

lemma *merge-eq-Cons-imp*:
‹*xs* ≠ [] ∧ *z* = *hd xs* ∨ *ys* ≠ [] ∧ *z* = *hd ys*›
if ‹*merge xs ys* = *z* # *zs*›
using *that* **by** (*induction xs ys rule*: *merge.induct*) (*auto split*: *if-splits*)

lemma *filter-merge*:
‹*filter P* (*merge xs ys*) = *merge* (*filter P xs*) (*filter P ys*)›
if ‹*sorted cmp xs*› **and** ‹*sorted cmp ys*›
using *that* **proof** (*induction xs ys rule*: *merge.induct*)
case (*1 ys*)
then show *?case*
by *simp*
next
case (*2 xs*)
then show *?case*
by *simp*
next
case (*3 x xs y ys*)
show *?case*
proof (*cases* ‹*compare cmp x y* = *Greater*›)
case *True*
with *3* **have** *hyp*: ‹*filter P* (*merge* (*x* # *xs*) *ys*) =
merge (*filter P* (*x* # *xs*)) (*filter P ys*)›
by (*simp add*: *sorted-Cons-imp-sorted*)
show *?thesis*
proof (*cases* ‹¬ *P x* ∧ *P y*›)
case *False*
with ‹*compare cmp x y* = *Greater*› **show** *?thesis*

```
      by (auto simp add: hyp)
    next
      case True
      from ‹compare cmp x y = Greater› 3.prems
      have ∗: ‹compare cmp z y = Greater› if ‹z ∈ set (filter P xs)› for z
      using that by (auto dest: compare.trans-not-greater sorted-Cons-imp-not-less)
      from ‹compare cmp x y = Greater› show ?thesis
        by (cases ‹filter P xs›) (simp-all add: hyp ∗)
    qed
  next
    case False
    with 3 have hyp: ‹filter P (merge xs (y # ys)) =
      merge (filter P xs) (filter P (y # ys))›
      by (simp add: sorted-Cons-imp-sorted)
    show ?thesis
    proof (cases ‹P x ∧ ¬ P y›)
      case False
      with ‹compare cmp x y ≠ Greater› show ?thesis
        by (auto simp add: hyp)
    next
      case True
      from ‹compare cmp x y ≠ Greater› 3.prems
      have ∗: ‹compare cmp x z ≠ Greater› if ‹z ∈ set (filter P ys)› for z
      using that by (auto dest: compare.trans-not-greater sorted-Cons-imp-not-less)
      from ‹compare cmp x y ≠ Greater› show ?thesis
        by (cases ‹filter P ys›) (simp-all add: hyp ∗)
    qed
  qed
qed

lemma sorted-merge:
  ‹sorted cmp (merge xs ys)› if ‹sorted cmp xs› and ‹sorted cmp ys›
using that proof (induction xs ys rule: merge.induct)
  case (1 ys)
  then show ?case
    by simp
next
  case (2 xs)
  then show ?case
    by simp
next
  case (3 x xs y ys)
  show ?case
  proof (cases ‹compare cmp x y = Greater›)
    case True
    with 3 have ‹sorted cmp (merge (x # xs) ys)›
      by (simp add: sorted-Cons-imp-sorted)
    then have ‹sorted cmp (y # merge (x # xs) ys)›
    proof (rule sorted-ConsI)
```

```
    fix z zs
    assume ‹merge (x # xs) ys = z # zs›
    with 3(4) True show ‹compare cmp y z ≠ Greater›
      by (clarsimp simp add: sorted-Cons-imp-sorted dest!: merge-eq-Cons-imp)
        (auto simp add: compare.asym-greater sorted-Cons-imp-not-less)
  qed
  with True show ?thesis
    by simp
 next
  case False
  with 3 have ‹sorted cmp (merge xs (y # ys))›
    by (simp add: sorted-Cons-imp-sorted)
  then have ‹sorted cmp (x # merge xs (y # ys))›
  proof (rule sorted-ConsI)
    fix z zs
    assume ‹merge xs (y # ys) = z # zs›
    with 3(3) False show ‹compare cmp x z ≠ Greater›
      by (clarsimp simp add: sorted-Cons-imp-sorted dest!: merge-eq-Cons-imp)
        (auto simp add: compare.asym-greater sorted-Cons-imp-not-less)
  qed
  with False show ?thesis
    by simp
 qed
qed

lemma merge-eq-appendI:
 ‹merge xs ys = xs @ ys›
  if ‹⋀x y. x ∈ set xs ⟹ y ∈ set ys ⟹ compare cmp x y ≠ Greater›
 using that by (induction xs ys rule: merge.induct) simp-all

lemma merge-stable-segments:
 ‹merge (stable-segment cmp l xs) (stable-segment cmp l ys) =
   stable-segment cmp l xs @ stable-segment cmp l ys›
 by (rule merge-eq-appendI) (auto dest: compare.trans-equiv-greater)

lemma sort-by-mergesort-rec:
 ‹sort cmp xs =
   merge (sort cmp (take (length xs div 2) xs))
     (sort cmp (drop (length xs div 2) xs))› (is ‹- = ?rhs›)
proof (rule sort-eqI)
 have ‹mset (take (length xs div 2) xs) + mset (drop (length xs div 2) xs) =
   mset (take (length xs div 2) xs @ drop (length xs div 2) xs)›
   by (simp only: mset-append)
 then show ‹mset xs = mset ?rhs›
   by (simp add: mset-merge)
next
 show ‹sorted cmp ?rhs›
   by (simp add: sorted-merge)
next
```

```
  fix l
  have ‹stable-segment cmp l (take (length xs div 2) xs) @ stable-segment cmp l
(drop (length xs div 2) xs)
    = stable-segment cmp l xs›
    by (simp only: filter-append [symmetric] append-take-drop-id)
  have ‹merge (stable-segment cmp l (take (length xs div 2) xs))
    (stable-segment cmp l (drop (length xs div 2) xs)) =
     stable-segment cmp l (take (length xs div 2) xs) @ stable-segment cmp l (drop
(length xs div 2) xs)›
    by (rule merge-eq-appendI) (auto simp add: compare.trans-equiv-greater)
  also have ‹... = stable-segment cmp l xs›
    by (simp only: filter-append [symmetric] append-take-drop-id)
  finally show ‹stable-segment cmp l xs = stable-segment cmp l ?rhs›
    by (simp add: stable-sort filter-merge)
qed

lemma mergesort-code [code]:
  ‹mergesort cmp xs =
    (case xs of
      [] ⇒ []
    | [x] ⇒ xs
    | [x, y] ⇒ (if compare cmp x y ≠ Greater then xs else [y, x])
    | - ⇒
        let
          half = length xs div 2;
          ys = take half xs;
          zs = drop half xs
        in merge (mergesort cmp ys) (mergesort cmp zs))›
proof (cases ‹length xs ≥ 3›)
  case False
  then have ‹length xs ∈ {0, 1, 2}›
    by (auto simp add: not-le le-less less-antisym)
  then consider ‹xs = []› | x where ‹xs = [x]› | x y where ‹xs = [x, y]›
    by (auto simp add: length-Suc-conv numeral-2-eq-2)
  then show ?thesis
    by cases simp-all
next
  case True
  then obtain x y z zs where ‹xs = x # y # z # zs›
   by (metis le-0-eq length-0-conv length-Cons list.exhaust not-less-eq-eq numeral-3-eq-3)
  moreover have ‹mergesort cmp xs =
    (let
       half = length xs div 2;
       ys = take half xs;
       zs = drop half xs
     in merge (mergesort cmp ys) (mergesort cmp zs))›
    using sort-by-mergesort-rec [of xs] by (simp add: Let-def)
  ultimately show ?thesis
    by simp
```

qed

end

101.3 Lexicographic products

lemma *sorted-prod-lex-imp-sorted-fst*:
‹*sorted* (*key fst cmp1*) *ps*› **if** ‹*sorted* (*prod-lex cmp1 cmp2*) *ps*›
using *that* **proof** (*induction rule*: *sorted-induct*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons p ps*)
have ‹*compare* (*key fst cmp1*) *p q* ≠ *Greater*› **if** ‹*ps* = *q* # *qs*› **for** *q qs*
using *that Cons.hyps*(*2*) [*of q*] **by** (*simp add*: *compare-prod-lex-apply split*: *comp.splits*)
with *Cons.IH* **show** *?case*
by (*rule sorted-ConsI*) *simp*
qed

lemma *sorted-prod-lex-imp-sorted-snd*:
‹*sorted* (*key snd cmp2*) *ps*› **if** ‹*sorted* (*prod-lex cmp1 cmp2*) *ps*› ‹⋀*a′ b′*. (*a′*, *b′*) ∈ *set ps* ⟹ *compare cmp1 a a′* = *Equiv*›
using *that* **proof** (*induction rule*: *sorted-induct*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons p ps*)
then show *?case*
apply (*cases p*)
apply (*rule sorted-ConsI*)
apply (*simp-all add*: *compare-prod-lex-apply*)
apply (*auto cong del*: *comp.case-cong-weak*)
apply (*metis comp.simps*(*8*) *compare.equiv-subst-left*)
done
qed

lemma *sort-comp-fst-snd-eq-sort-prod-lex*:
‹*sort* (*key fst cmp1*) ∘ *sort* (*key snd cmp2*) = *sort* (*prod-lex cmp1 cmp2*)› (**is** ‹*sort ?cmp1* ∘ *sort ?cmp2* = *sort ?cmp*›)
proof
fix *ps* :: ‹(*'a* × *'b*) *list*›
have ‹*sort ?cmp1* (*sort ?cmp2 ps*) = *sort ?cmp ps*›
proof (*rule sort-eqI*)
show ‹*mset* (*sort ?cmp2 ps*) = *mset* (*sort ?cmp ps*)›
by *simp*
show ‹*sorted ?cmp1* (*sort ?cmp ps*)›

```
    by (rule sorted-prod-lex-imp-sorted-fst [of - cmp2]) simp
  next
    fix p :: ‹'a × 'b›
    define a b where ab: ‹a = fst p› ‹b = snd p›
    moreover assume ‹p ∈ set (sort ?cmp2 ps)›
    ultimately have ‹(a, b) ∈ set (sort ?cmp2 ps)›
      by simp
    let ?qs = ‹filter (λ(a', -). compare cmp1 a a' = Equiv) ps›
    have ‹sort ?cmp2 ?qs = sort ?cmp ?qs›
    proof (rule sort-eqI)
      show ‹mset ?qs = mset (sort ?cmp ?qs)›
        by simp
      show ‹sorted ?cmp2 (sort ?cmp ?qs)›
        by (rule sorted-prod-lex-imp-sorted-snd) auto
    next
      fix q :: ‹'a × 'b›
      define c d where ‹c = fst q› ‹d = snd q›
      moreover assume ‹q ∈ set ?qs›
      ultimately have ‹(c, d) ∈ set ?qs›
        by simp
      from sorted-stable-segment [of ?cmp ‹(a, d)› ps]
      have ‹sorted ?cmp (filter (λ(c, b). compare (prod-lex cmp1 cmp2) (a, d) (c,
b) = Equiv) ps)›
        by (simp only: case-prod-unfold prod.collapse)
      also have ‹(λ(c, b). compare (prod-lex cmp1 cmp2) (a, d) (c, b) = Equiv) =
        (λ(c, b). compare cmp1 a c = Equiv ∧ compare cmp2 d b = Equiv)›
        by (simp add: fun-eq-iff compare-prod-lex-apply split: comp.split)
      finally have ∗: ‹sorted ?cmp (filter (λ(c, b). compare cmp1 a c = Equiv ∧
compare cmp2 d b = Equiv) ps)› .
      let ?rs = ‹filter (λ(-, d'). compare cmp2 d d' = Equiv) ?qs›
      have ‹sort ?cmp ?rs = ?rs›
        by (rule sort-eqI) (use ∗ in ‹simp-all add: case-prod-unfold›)
      then show ‹filter (λr. compare ?cmp2 q r = Equiv) ?qs =
        filter (λr. compare ?cmp2 q r = Equiv) (sort ?cmp ?qs)›
        by (simp add: filter-sort case-prod-unfold flip: ‹d = snd q›)
    qed
    then show ‹filter (λq. compare ?cmp1 p q = Equiv) (sort ?cmp2 ps) =
      filter (λq. compare ?cmp1 p q = Equiv) (sort ?cmp ps)›
      by (simp add: filter-sort case-prod-unfold flip: ab)
  qed
  then show ‹(sort (key fst cmp1) ∘ sort (key snd cmp2)) ps = sort (prod-lex
cmp1 cmp2) ps›
    by simp
qed

end
```

102 A decision procedure for universal multivariate real arithmetic with addition, multiplication and ordering using semidefinite programming

theory *Sum-of-Squares*
imports *Complex-Main*
begin

ML-file ‹*Sum-of-Squares/positivstellensatz.ML*›
ML-file ‹*Sum-of-Squares/positivstellensatz-tools.ML*›
ML-file ‹*Sum-of-Squares/sum-of-squares.ML*›
ML-file ‹*Sum-of-Squares/sos-wrapper.ML*›

end

theory *Time-Commands*
imports *Main*
keywords *time-fun* :: *thy-decl*
and *time-function* :: *thy-decl*
and *time-definition* :: *thy-decl*
and *time-partial-function* :: *thy-decl*
and *equations*
and *time-fun-0* :: *thy-decl*
begin

ML-file *time-commands.ML*

declare [[*time-prefix* = *T-*]]

This theory provides commands for the automatic definition of step-counting running-time functions from HOL functions following the translation described in Section 1.5, Running Time, of the book "Functional Data Structures and Algorithms. A Proof Assistant Approach." See https://functional-algorithms-verified.org

Command *time-fun f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T-f*. For all auxiliary functions used by *f* (excluding constructors), running time functions must already have been defined. If the definition of the function requires a manual termination proof, use *time-function* accompanied by a *termination* command.

The pre-defined functions below are assumed to have constant running time. In fact, we make that constant 0. This does not change the asymptotic running time of user-defined functions using the pre-defined functions because 1 is added for every user-defined function call.

Many of the functions below are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.

The constant-time assumption for (=) is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example $xs = []$.

Users of this running time framework need to ensure that 0-time functions are used only within the above restrictions.

time-fun-0 *min*
time-fun-0 *max*
time-fun-0 (+)
time-fun-0 (−)
time-fun-0 (∗)
time-fun-0 (/)
time-fun-0 (*div*)
time-fun-0 (<)
time-fun-0 (≤)
time-fun-0 *Not*
time-fun-0 (∧)
time-fun-0 (∨)
time-fun-0 *Num.numeral-class.numeral*
time-fun-0 (=)

end

103 Time functions for various standard library operations

theory *Time-Functions*
imports *Time-Commands*
begin

time-fun *fst*
time-fun *snd*

lemma *T-fst-0*[*simp*]: *T-fst x = 0*
by (*metis T-fst.elims*)

lemma *T-snd-0*[*simp*]: *T-snd x = 0*
by (*metis T-snd.elims*)

time-fun *the*

103.1 List

time-fun *hd*

lemma *T-hd*[*simp*]: $xs \neq [] \Longrightarrow$ *T-hd xs* = *0*
by (*cases xs*) *simp-all*

time-fun *tl*

lemma *T-tl*[*simp*]: *T-tl xs* = *0*
by (*cases xs*) *simp-all*

declare *T-tl.simps*[*simp del*]

time-fun (@)

lemma *T-append*[*simp*]: *T-append xs ys* = *length xs* + *1*
by(*induction xs*) *auto*

class *T-size* =
  fixes *T-size* :: $'a \Rightarrow$ *nat*

instantiation *list* :: (-) *T-size*
begin

time-fun *length*

instance ..

end

abbreviation *T-length* :: $'a$ *list* $\Rightarrow$ *nat* **where**
T-length $\equiv$ *T-size*

lemma *T-length*: *T-length xs* = *length xs* + *1*
  by (*induction xs*) *auto*

lemmas [*simp del*] = *T-size-list.simps*

time-fun *map*

lemma *T-map-simps* [*simp*,*code*]:
  T-map T-f [] = *1*
  T-map T-f (*x* # *xs*) = *T-f x* + *T-map T-f xs* + *1*
by (*simp-all add*: *T-map-def*)

lemma *T-map*: *T-map T-f xs* = ($\sum$ *x*←*xs*. *T-f x*) + *length xs* + *1*
  by (*induction xs*) *auto*

lemmas [*simp del*] = *T-map-simps*

lemma *T-map-bound*:

$\forall x \in set\ xs.\ T\text{-}P\ x \leq k \Longrightarrow T\text{-}map\ T\text{-}P\ xs \leq k * length\ xs + length\ xs + 1$
using *sum-list-bound*[*of map T-P xs*] **by**(*simp add*: *T-map*)

time-fun *filter*

lemma *T-filter-simps* [*code*]:
T-filter T-P [] = *1*
T-filter T-P (*x* # *xs*) = *T-P x* + *T-filter T-P xs* + *1*
by (*simp-all add*: *T-filter-def*)

lemma *T-filter*: *T-filter T-P xs* = $(\sum x \leftarrow xs.\ T\text{-}P\ x)$ + *length xs* + *1*
by (*induction xs*) (*auto simp*: *T-filter-simps*)

lemma *T-filter-eq-T-map*: *T-filter T-f xs* = *T-map T-f xs*
by (*simp add*: *T-filter T-map*)

lemma *T-filter-bound*:
$\forall x \in set\ xs.\ T\text{-}P\ x \leq k \Longrightarrow T\text{-}filter\ T\text{-}P\ xs \leq k * length\ xs + length\ xs + 1$
by (*metis T-filter-eq-T-map T-map-bound*)

time-fun *nth*

lemma *T-nth*: *n* < *length xs* $\Longrightarrow$ *T-nth xs n* = *n* + *1*
by (*induction xs n rule*: *T-nth.induct*) (*auto split*: *nat.splits*)

lemmas [*simp del*] = *T-nth.simps*

time-fun *take*
time-fun *drop*

lemma *T-take*: *T-take n xs* = *min n* (*length xs*) + *1*
by (*induction xs arbitrary*: *n*) (*auto split*: *nat.splits*)

lemma *T-drop*: *T-drop n xs* = *min n* (*length xs*) + *1*
by (*induction xs arbitrary*: *n*) (*auto split*: *nat.splits*)

time-fun *zip*

lemma *T-zip*: *length xs* = *length ys* $\Longrightarrow$ *T-zip xs ys* = *length ys* + *1*
by (*induction xs ys rule*: *list-induct2*) *auto*

time-fun *rev*

lemma *T-rev*: *T-rev xs* $\leq$ (*length xs* + *1*)^2
by(*induction xs*) (*auto simp*: *T-append power2-eq-square*)

fun *itrev* :: *'a list* $\Rightarrow$ *'a list* $\Rightarrow$ *'a list* **where**
itrev [] *ys* = *ys* |
itrev (*x*#*xs*) *ys* = *itrev xs* (*x* # *ys*)

lemma *itrev*: *itrev xs ys = rev xs @ ys*
by(*induction xs arbitrary*: *ys*) *auto*

lemma *itrev-Nil*: *itrev xs [] = rev xs*
by(*simp add*: *itrev*)

time-fun *itrev*

lemma *T-itrev*: *T-itrev xs ys = length xs + 1*
by(*induction xs arbitrary*: *ys*) *auto*

time-fun *list-update*

lemma *T-list-update*[*simp*]: *i < length xs ⟹ T-list-update xs i x = i + 1*
by(*induction xs arbitrary*: *i*) (*auto split*: *nat.splits*)

time-fun *last*

lemma *T-last*[*simp*]: *as ≠ [] ⟹ T-last as = length as*
by (*induction as*) *auto*

end

104 A table-based implementation of the reflexive transitive closure

theory *Transitive-Closure-Table*
imports *Main*
begin

inductive *rtrancl-path* :: (*'a ⇒ 'a ⇒ bool*) *⇒ 'a ⇒ 'a list ⇒ 'a ⇒ bool*
for *r* :: *'a ⇒ 'a ⇒ bool*
where
base: *rtrancl-path r x [] x*
| *step*: *r x y ⟹ rtrancl-path r y ys z ⟹ rtrancl-path r x (y # ys) z*

lemma *rtranclp-eq-rtrancl-path*: *r** x y ⟷ (∃ xs. rtrancl-path r x xs y)*
proof
show *∃ xs. rtrancl-path r x xs y* **if** *r** x y*
using *that*
proof (*induct rule*: *converse-rtranclp-induct*)
case *base*
have *rtrancl-path r y [] y* **by** (*rule rtrancl-path.base*)
then show *?case* **..**
next
case (*step x z*)
from ‹*∃ xs. rtrancl-path r z xs y*›

```
    obtain xs where rtrancl-path r z xs y ..
    with ‹r x z› have rtrancl-path r x (z # xs) y
      by (rule rtrancl-path.step)
    then show ?case ..
  qed
  show r** x y if ∃xs. rtrancl-path r x xs y
  proof −
    from that obtain xs where rtrancl-path r x xs y ..
    then show ?thesis
    proof induct
      case (base x)
      show ?case
        by (rule rtranclp.rtrancl-refl)
    next
      case (step x y ys z)
      from ‹r x y› ‹r** y z› show ?case
        by (rule converse-rtranclp-into-rtranclp)
    qed
  qed
qed

lemma rtrancl-path-trans:
  assumes xy: rtrancl-path r x xs y
    and yz: rtrancl-path r y ys z
  shows rtrancl-path r x (xs @ ys) z using xy yz
proof (induct arbitrary: z)
  case (base x)
  then show ?case by simp
next
  case (step x y xs)
  then have rtrancl-path r y (xs @ ys) z
    by simp
  with ‹r x y› have rtrancl-path r x (y # (xs @ ys)) z
    by (rule rtrancl-path.step)
  then show ?case by simp
qed

lemma rtrancl-path-appendE:
  assumes xz: rtrancl-path r x (xs @ y # ys) z
  obtains rtrancl-path r x (xs @ [y]) y and rtrancl-path r y ys z
  using xz
proof (induct xs arbitrary: x)
  case Nil
  then have rtrancl-path r x (y # ys) z by simp
  then obtain xy: r x y and yz: rtrancl-path r y ys z
    by cases auto
  from xy have rtrancl-path r x [y] y
    by (rule rtrancl-path.step [OF - rtrancl-path.base])
  then have rtrancl-path r x ([] @ [y]) y by simp
```

```
  then show thesis using yz by (rule Nil)
next
  case (Cons a as)
  then have rtrancl-path r x (a # (as @ y # ys)) z by simp
  then obtain xa: r x a and az: rtrancl-path r a (as @ y # ys) z
    by cases auto
  show thesis
  proof (rule Cons(1) [OF - az])
    assume rtrancl-path r y ys z
    assume rtrancl-path r a (as @ [y]) y
    with xa have rtrancl-path r x (a # (as @ [y])) y
      by (rule rtrancl-path.step)
    then have rtrancl-path r x ((a # as) @ [y]) y
      by simp
    then show thesis using ‹rtrancl-path r y ys z›
      by (rule Cons(2))
  qed
qed

lemma rtrancl-path-distinct:
  assumes xy: rtrancl-path r x xs y
  obtains xs' where rtrancl-path r x xs' y and distinct (x # xs') and set xs' ⊆
set xs
  using xy
proof (induct xs rule: measure-induct-rule [of length])
  case (less xs)
  show ?case
  proof (cases distinct (x # xs))
    case True
    with ‹rtrancl-path r x xs y› show ?thesis by (rule less) simp
  next
    case False
    then have ∃ as bs cs a. x # xs = as @ [a] @ bs @ [a] @ cs
      by (rule not-distinct-decomp)
    then obtain as bs cs a where xxs: x # xs = as @ [a] @ bs @ [a] @ cs
      by iprover
    show ?thesis
    proof (cases as)
      case Nil
      with xxs have x: x = a and xs: xs = bs @ a # cs
        by auto
      from x xs ‹rtrancl-path r x xs y› have cs: rtrancl-path r x cs y set cs ⊆ set xs
        by (auto elim: rtrancl-path-appendE)
      from xs have length cs < length xs by simp
      then show ?thesis
        by (rule less(1))(blast intro: cs less(2) order-trans del: subsetI)+
    next
      case (Cons d ds)
      with xxs have xs: xs = ds @ a # (bs @ [a] @ cs)
```

```
      by auto
    with ‹rtrancl-path r x xs y› obtain xa: rtrancl-path r x (ds @ [a]) a
      and ay: rtrancl-path r a (bs @ a # cs) y
      by (auto elim: rtrancl-path-appendE)
    from ay have rtrancl-path r a cs y by (auto elim: rtrancl-path-appendE)
    with xa have xy: rtrancl-path r x ((ds @ [a]) @ cs) y
      by (rule rtrancl-path-trans)
    from xs have set: set ((ds @ [a]) @ cs) ⊆ set xs by auto
    from xs have length ((ds @ [a]) @ cs) < length xs by simp
    then show ?thesis
      by (rule less(1))(blast intro: xy less(2) set[THEN subsetD])+
  qed
 qed
qed

inductive rtrancl-tab :: ('a ⇒ 'a ⇒ bool) ⇒ 'a list ⇒ 'a ⇒ 'a ⇒ bool
 for r :: 'a ⇒ 'a ⇒ bool
where
 base: rtrancl-tab r xs x x
| step: x ∉ set xs ⟹ r x y ⟹ rtrancl-tab r (x # xs) y z ⟹ rtrancl-tab r xs x z

lemma rtrancl-path-imp-rtrancl-tab:
 assumes path: rtrancl-path r x xs y
   and x: distinct (x # xs)
   and ys: ({x} ∪ set xs) ∩ set ys = {}
 shows rtrancl-tab r ys x y
 using path x ys
proof (induct arbitrary: ys)
 case base
 show ?case
   by (rule rtrancl-tab.base)
next
 case (step x y zs z)
 then have x ∉ set ys
   by auto
 from step have distinct (y # zs)
   by simp
 moreover from step have ({y} ∪ set zs) ∩ set (x # ys) = {}
   by auto
 ultimately have rtrancl-tab r (x # ys) y z
   by (rule step)
 with ‹x ∉ set ys› ‹r x y› show ?case
   by (rule rtrancl-tab.step)
qed

lemma rtrancl-tab-imp-rtrancl-path:
 assumes tab: rtrancl-tab r ys x y
 obtains xs where rtrancl-path r x xs y
 using tab
```

proof *induct*
case *base*
from *rtrancl-path.base* **show** *?case*
by (*rule base*)
next
case *step*
show *?case*
by (*iprover intro*: *step rtrancl-path.step*)
qed

lemma *rtranclp-eq-rtrancl-tab-nil*: $r^{**}\ x\ y \longleftrightarrow$ *rtrancl-tab r* [] *x y*
proof
show *rtrancl-tab r* [] *x y* **if** $r^{**}\ x\ y$
proof −
from *that* **obtain** *xs* **where** *rtrancl-path r x xs y*
by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
then obtain *xs'* **where** *xs'*: *rtrancl-path r x xs' y* **and** *distinct*: *distinct* (*x* # *xs'*)
by (*rule rtrancl-path-distinct*)
have ({*x*} ∪ *set xs'*) ∩ *set* [] = {}
by *simp*
with *xs' distinct* **show** *?thesis*
by (*rule rtrancl-path-imp-rtrancl-tab*)
qed
show $r^{**}\ x\ y$ **if** *rtrancl-tab r* [] *x y*
proof −
from *that* **obtain** *xs* **where** *rtrancl-path r x xs y*
by (*rule rtrancl-tab-imp-rtrancl-path*)
then show *?thesis*
by (*auto simp add*: *rtranclp-eq-rtrancl-path*)
qed
qed

declare *rtranclp-eq-rtrancl-tab-nil* [*code*]
declare *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD2*, *code-pred-intro*]

code-pred *rtranclp*
using *rtranclp-eq-rtrancl-tab-nil* [*THEN iffD1*] **by** *fastforce*

lemma *rtrancl-path-Range*: ⟦ *rtrancl-path R x xs y*; *z* ∈ *set xs* ⟧ ⟹ *Rangep R z*
by(*induction rule*: *rtrancl-path.induct*) *auto*

lemma *rtrancl-path-Range-end*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *Rangep R y*
by(*induction rule*: *rtrancl-path.induct*)(*auto elim*: *rtrancl-path.cases*)

lemma *rtrancl-path-nth*:
⟦ *rtrancl-path R x xs y*; *i* < *length xs* ⟧ ⟹ *R* ((*x* # *xs*) ! *i*) (*xs* ! *i*)
proof(*induction arbitrary*: *i rule*: *rtrancl-path.induct*)
case *step* **thus** *?case* **by**(*cases i*) *simp-all*

qed *simp*

lemma *rtrancl-path-last*: ⟦ *rtrancl-path R x xs y*; *xs* ≠ [] ⟧ ⟹ *last xs* = *y*
by(*induction rule*: *rtrancl-path.induct*)(*auto elim*: *rtrancl-path.cases*)

lemma *rtrancl-path-mono*:
⟦ *rtrancl-path R x p y*; ⋀*x y*. *R x y* ⟹ *S x y* ⟧ ⟹ *rtrancl-path S x p y*
by(*induction rule*: *rtrancl-path.induct*)(*auto intro*: *rtrancl-path.intros*)

end

105 Binary Tree

theory *Tree*
imports *Main*
begin

datatype *'a tree* =
Leaf (‹⟨⟩›) |
Node 'a tree (*value*: *'a*) *'a tree* (‹(‹*indent=1 notation=‹mixfix Node›*›⟨-,/ -,/ -⟩)›)
datatype-compat *tree*

primrec *left* :: *'a tree* ⇒ *'a tree* **where**
left (*Node l v r*) = *l* |
left Leaf = *Leaf*

primrec *right* :: *'a tree* ⇒ *'a tree* **where**
right (*Node l v r*) = *r* |
right Leaf = *Leaf*

Counting the number of leaves rather than nodes:

fun *size1* :: *'a tree* ⇒ *nat* **where**
size1 ⟨⟩ = *1* |
size1 ⟨*l*, *x*, *r*⟩ = *size1 l* + *size1 r*

fun *subtrees* :: *'a tree* ⇒ *'a tree set* **where**
subtrees ⟨⟩ = {⟨⟩} |
subtrees (⟨*l*, *a*, *r*⟩) = {⟨*l*, *a*, *r*⟩} ∪ *subtrees l* ∪ *subtrees r*

fun *mirror* :: *'a tree* ⇒ *'a tree* **where**
mirror ⟨⟩ = *Leaf* |
mirror ⟨*l*,*x*,*r*⟩ = ⟨*mirror r*, *x*, *mirror l*⟩

class *height* = **fixes** *height* :: *'a* ⇒ *nat*

instantiation *tree* :: (*type*)*height*
begin

fun *height-tree* :: *'a tree* => *nat* **where**

height Leaf = 0 |
height (Node l a r) = max (height l) (height r) + 1

instance ..

end

fun *min-height* :: *'a tree* ⇒ *nat* **where**
min-height Leaf = 0 |
min-height (Node l - r) = min (min-height l) (min-height r) + 1

fun *complete* :: *'a tree* ⇒ *bool* **where**
complete Leaf = True |
complete (Node l x r) = (height l = height r ∧ complete l ∧ complete r)

Almost complete:

definition *acomplete* :: *'a tree* ⇒ *bool* **where**
acomplete t = (height t − min-height t ≤ 1)

Weight balanced:

fun *wbalanced* :: *'a tree* ⇒ *bool* **where**
wbalanced Leaf = True |
wbalanced (Node l x r) = (abs(int(size l) − int(size r)) ≤ 1 ∧ wbalanced l ∧ wbalanced r)

Internal path length:

fun *ipl* :: *'a tree* ⇒ *nat* **where**
ipl Leaf = 0 |
ipl (Node l - r) = ipl l + size l + ipl r + size r

fun *preorder* :: *'a tree* ⇒ *'a list* **where**
preorder ⟨⟩ = [] |
preorder ⟨l, x, r⟩ = x # preorder l @ preorder r

fun *inorder* :: *'a tree* ⇒ *'a list* **where**
inorder ⟨⟩ = [] |
inorder ⟨l, x, r⟩ = inorder l @ [x] @ inorder r

A linear version avoiding append:

fun *inorder2* :: *'a tree* ⇒ *'a list* ⇒ *'a list* **where**
inorder2 ⟨⟩ xs = xs |
inorder2 ⟨l, x, r⟩ xs = inorder2 l (x # inorder2 r xs)

fun *postorder* :: *'a tree* ⇒ *'a list* **where**
postorder ⟨⟩ = [] |
postorder ⟨l, x, r⟩ = postorder l @ postorder r @ [x]

Binary Search Tree:

fun *bst-wrt* :: *('a ⇒ 'a ⇒ bool) ⇒ 'a tree ⇒ bool* **where**

bst-wrt P ⟨⟩ ⟷ *True* |
bst-wrt P ⟨*l*, *a*, *r*⟩ ⟷
(∀ *x*∈*set-tree l*. *P x a*) ∧ (∀ *x*∈*set-tree r*. *P a x*) ∧ *bst-wrt P l* ∧ *bst-wrt P r*

abbreviation *bst* :: ('*a*::*linorder*) *tree* ⇒ *bool* **where**
bst ≡ *bst-wrt* (<)

fun (**in** *linorder*) *heap* :: '*a tree* ⇒ *bool* **where**
heap Leaf = *True* |
heap (*Node l m r*) =
((∀ *x* ∈ *set-tree l* ∪ *set-tree r*. *m* ≤ *x*) ∧ *heap l* ∧ *heap r*)

105.1 *map-tree*

lemma *eq-map-tree-Leaf*[*simp*]: *map-tree f t* = *Leaf* ⟷ *t* = *Leaf*
by (*rule tree.map-disc-iff*)

lemma *eq-Leaf-map-tree*[*simp*]: *Leaf* = *map-tree f t* ⟷ *t* = *Leaf*
by (*cases t*) *auto*

105.2 *size*

lemma *size1-size*: *size1 t* = *size t* + *1*
by (*induction t*) *simp-all*

lemma *size1-ge0*[*simp*]: *0* < *size1 t*
by (*simp add*: *size1-size*)

lemma *eq-size-0*[*simp*]: *size t* = *0* ⟷ *t* = *Leaf*
by(*cases t*) *auto*

lemma *eq-0-size*[*simp*]: *0* = *size t* ⟷ *t* = *Leaf*
by(*cases t*) *auto*

lemma *neq-Leaf-iff*: (*t* ≠ ⟨⟩) = (∃ *l a r*. *t* = ⟨*l*, *a*, *r*⟩)
by (*cases t*) *auto*

lemma *size-map-tree*[*simp*]: *size* (*map-tree f t*) = *size t*
by (*induction t*) *auto*

lemma *size1-map-tree*[*simp*]: *size1* (*map-tree f t*) = *size1 t*
by (*simp add*: *size1-size*)

105.3 *set-tree*

lemma *eq-set-tree-empty*[*simp*]: *set-tree t* = {} ⟷ *t* = *Leaf*
by (*cases t*) *auto*

lemma *eq-empty-set-tree*[*simp*]: {} = *set-tree t* ⟷ *t* = *Leaf*
by (*cases t*) *auto*

lemma *finite-set-tree*[*simp*]: *finite*(*set-tree t*)
by(*fact tree.set-finite*)

lemma *finite-trees-height-le*:
assumes *finite L*
shows *finite* {*t* :: *'a tree. height t* ≤ *n* ∧ *set-tree t* ⊆ *L*}
proof (*induction n*)
case *0*
have {*t* :: *'a tree. height t* ≤ *0* ∧ *set-tree t* ⊆ *L*} = {*Leaf*}
using *height-tree.elims* **by** *auto*
then show *?case* **by** *simp*
next
case (*Suc n*)

let *?T* = λ*n*. {*t* :: *'a tree. height t* ≤ *n* ∧ *set-tree t* ⊆ *L*}

have *?T* (*Suc n*) ⊆ {*Leaf*} ∪ (λ(*l*,*a*,*r*). *Node l a r*) ' (*?T n* × *L* × *?T n*)
apply (*auto split*: *prod.splits simp*: *image-def*)
by (*smt* (*verit*, *best*) *Suc-eq-plus1 add-diff-cancel-right' height-tree.elims le-diff-conv le-sup-iff subsetD*
sup-nat-def tree.sel(*2*) *tree.set-sel*(*2*) *tree.simps*(*15*))

then show *finite* (*?T* (*Suc n*))
using *Suc.IH assms*
by (*metis* (*no-types*, *lifting*) *finite.emptyI finite-SigmaI finite-Un finite-imageI finite-insert*
rev-finite-subset)
qed

105.4 *subtrees*

lemma *neq-subtrees-empty*[*simp*]: *subtrees t* ≠ {}
by (*cases t*)(*auto*)

lemma *neq-empty-subtrees*[*simp*]: {} ≠ *subtrees t*
by (*cases t*)(*auto*)

lemma *size-subtrees*: *s* ∈ *subtrees t* ⟹ *size s* ≤ *size t*
by(*induction t*)(*auto*)

lemma *set-treeE*: *a* ∈ *set-tree t* ⟹ ∃ *l r*. ⟨*l*, *a*, *r*⟩ ∈ *subtrees t*
by (*induction t*)(*auto*)

lemma *Node-notin-subtrees-if*[*simp*]: *a* ∉ *set-tree t* ⟹ *Node l a r* ∉ *subtrees t*
by (*induction t*) *auto*

lemma *in-set-tree-if*: ⟨*l*, *a*, *r*⟩ ∈ *subtrees t* ⟹ *a* ∈ *set-tree t*
by (*metis Node-notin-subtrees-if*)

105.5 *height* and *min-height*

lemma *eq-height-0*[*simp*]: *height t = 0 ⟷ t = Leaf*
by(*cases t*) *auto*

lemma *eq-0-height*[*simp*]: *0 = height t ⟷ t = Leaf*
by(*cases t*) *auto*

lemma *height-map-tree*[*simp*]: *height* (*map-tree f t*) = *height t*
by (*induction t*) *auto*

lemma *height-le-size-tree*: *height t* ≤ *size* (*t*::'*a tree*)
by (*induction t*) *auto*

lemma *size1-height*: *size1 t* ≤ *2* ^ *height* (*t*::'*a tree*)
proof(*induction t*)
  case (*Node l a r*)
  show *?case*
  proof (*cases height l* ≤ *height r*)
    case *True*
    have *size1*(*Node l a r*) = *size1 l* + *size1 r* **by** *simp*
    also have ... ≤ *2* ^ *height l* + *2* ^ *height r* **using** *Node.IH* **by** *arith*
    also have ... ≤ *2* ^ *height r* + *2* ^ *height r* **using** *True* **by** *simp*
    also have ... = *2* ^ *height* (*Node l a r*)
      using *True* **by** (*auto simp*: *max-def mult-2*)
    finally show *?thesis* **.**
  next
    case *False*
    have *size1*(*Node l a r*) = *size1 l* + *size1 r* **by** *simp*
    also have ... ≤ *2* ^ *height l* + *2* ^ *height r* **using** *Node.IH* **by** *arith*
    also have ... ≤ *2* ^ *height l* + *2* ^ *height l* **using** *False* **by** *simp*
    finally show *?thesis* **using** *False* **by** (*auto simp*: *max-def mult-2*)
  qed
qed *simp*

corollary *size-height*: *size t* ≤ *2* ^ *height* (*t*::'*a tree*) − *1*
using *size1-height*[*of t*, *unfolded size1-size*] **by**(*arith*)

lemma *height-subtrees*: *s* ∈ *subtrees t* ⟹ *height s* ≤ *height t*
by (*induction t*) *auto*

lemma *min-height-le-height*: *min-height t* ≤ *height t*
by(*induction t*) *auto*

lemma *min-height-map-tree*[*simp*]: *min-height* (*map-tree f t*) = *min-height t*
by (*induction t*) *auto*

lemma *min-height-size1*: *2* ^ *min-height t* ≤ *size1 t*
proof(*induction t*)

case (*Node l a r*)
have (*2*::*nat*) ^ *min-height* (*Node l a r*) ≤ *2* ^ *min-height l* + *2* ^ *min-height r*
by (*simp add*: *min-def*)
also have ... ≤ *size1*(*Node l a r*) **using** *Node.IH* **by** *simp*
finally show *?case* **.**
qed *simp*

105.6 *complete*

lemma *complete-iff-height*: *complete t* ⟷ (*min-height t* = *height t*)
apply(*induction t*)
apply *simp*
apply (*simp add*: *min-def max-def*)
by (*metis le-antisym le-trans min-height-le-height*)

lemma *size1-if-complete*: *complete t* ⟹ *size1 t* = *2* ^ *height t*
by (*induction t*) *auto*

lemma *size-if-complete*: *complete t* ⟹ *size t* = *2* ^ *height t* − *1*
using *size1-if-complete*[*simplified size1-size*] **by** *fastforce*

lemma *size1-height-if-incomplete*:
¬ *complete t* ⟹ *size1 t* < *2* ^ *height t*
proof(*induction t*)
case *Leaf* **thus** *?case* **by** *simp*
next
case (*Node l x r*)
have *1*: *?case* **if** *h*: *height l* < *height r*
using *h size1-height*[*of l*] *size1-height*[*of r*] *power-strict-increasing*[*OF h*, *of* *2*::*nat*]
by(*auto simp*: *max-def simp del*: *power-strict-increasing-iff*)
have *2*: *?case* **if** *h*: *height l* > *height r*
using *h size1-height*[*of l*] *size1-height*[*of r*] *power-strict-increasing*[*OF h*, *of* *2*::*nat*]
by(*auto simp*: *max-def simp del*: *power-strict-increasing-iff*)
have *3*: *?case* **if** *h*: *height l* = *height r* **and** *c*: ¬ *complete l*
using *h size1-height*[*of r*] *Node.IH*(*1*)[*OF c*] **by**(*simp*)
have *4*: *?case* **if** *h*: *height l* = *height r* **and** *c*: ¬ *complete r*
using *h size1-height*[*of l*] *Node.IH*(*2*)[*OF c*] **by**(*simp*)
from *1 2 3 4 Node.prems* **show** *?case* **apply** (*simp add*: *max-def*) **by** *linarith*
qed

lemma *complete-iff-min-height*: *complete t* ⟷ (*height t* = *min-height t*)
by(*auto simp add*: *complete-iff-height*)

lemma *min-height-size1-if-incomplete*:
¬ *complete t* ⟹ *2* ^ *min-height t* < *size1 t*
proof(*induction t*)
case *Leaf* **thus** *?case* **by** *simp*

next
case (*Node l x r*)
have *1*: *?case* **if** *h*: *min-height l < min-height r*
using *h min-height-size1*[*of l*] *min-height-size1*[*of r*] *power-strict-increasing*[*OF h*, *of 2*::*nat*]
by(*auto simp*: *max-def simp del*: *power-strict-increasing-iff*)
have *2*: *?case* **if** *h*: *min-height l > min-height r*
using *h min-height-size1*[*of l*] *min-height-size1*[*of r*] *power-strict-increasing*[*OF h*, *of 2*::*nat*]
by(*auto simp*: *max-def simp del*: *power-strict-increasing-iff*)
have *3*: *?case* **if** *h*: *min-height l = min-height r* **and** *c*: ¬ *complete l*
using *h min-height-size1*[*of r*] *Node.IH*(*1*)[*OF c*] **by**(*simp add*: *complete-iff-min-height*)
have *4*: *?case* **if** *h*: *min-height l = min-height r* **and** *c*: ¬ *complete r*
using *h min-height-size1*[*of l*] *Node.IH*(*2*)[*OF c*] **by**(*simp add*: *complete-iff-min-height*)
from *1 2 3 4 Node.prems* **show** *?case*
by (*fastforce simp*: *complete-iff-min-height*[*THEN iffD1*])
qed

lemma *complete-if-size1-height*: *size1 t = 2 ^ height t ⟹ complete t*
using *size1-height-if-incomplete* **by** *fastforce*

lemma *complete-if-size1-min-height*: *size1 t = 2 ^ min-height t ⟹ complete t*
using *min-height-size1-if-incomplete* **by** *fastforce*

lemma *complete-iff-size1*: *complete t ⟷ size1 t = 2 ^ height t*
using *complete-if-size1-height size1-if-complete* **by** *blast*

105.7 *acomplete*

lemma *acomplete-subtreeL*: *acomplete* (*Node l x r*) *⟹ acomplete l*
by(*simp add*: *acomplete-def*)

lemma *acomplete-subtreeR*: *acomplete* (*Node l x r*) *⟹ acomplete r*
by(*simp add*: *acomplete-def*)

lemma *acomplete-subtrees*: ⟦ *acomplete t*; *s ∈ subtrees t* ⟧ *⟹ acomplete s*
using [[*simp-depth-limit=1*]]
by(*induction t arbitrary*: *s*)
(*auto simp add*: *acomplete-subtreeL acomplete-subtreeR*)

Balanced trees have optimal height:

lemma *acomplete-optimal*:
fixes *t* :: *'a tree* **and** *t'* :: *'b tree*
assumes *acomplete t size t ≤ size t'* **shows** *height t ≤ height t'*
proof (*cases complete t*)
case *True*
have (*2*::*nat*) *^ height t ≤ 2 ^ height t'*
proof −
have *2 ^ height t = size1 t*
using *True* **by** (*simp add*: *size1-if-complete*)

```
    also have ... ≤ size1 t' using assms(2) by(simp add: size1-size)
    also have ... ≤ 2 ^ height t' by (rule size1-height)
    finally show ?thesis .
  qed
  thus ?thesis by (simp)
next
  case False
  have (2::nat) ^ min-height t < 2 ^ height t'
  proof −
    have (2::nat) ^ min-height t < size1 t
      by(rule min-height-size1-if-incomplete[OF False])
    also have ... ≤ size1 t' using assms(2) by (simp add: size1-size)
    also have ... ≤ 2 ^ height t' by(rule size1-height)
    finally have (2::nat) ^ min-height t < (2::nat) ^ height t' .
    thus ?thesis .
  qed
  hence ∗: min-height t < height t' by simp
  have min-height t + 1 = height t
    using min-height-le-height[of t] assms(1) False
    by (simp add: complete-iff-height acomplete-def)
  with ∗ show ?thesis by arith
qed
```

105.8 *wbalanced*

```
lemma wbalanced-subtrees: ⟦ wbalanced t; s ∈ subtrees t ⟧ ⟹ wbalanced s
using [[simp-depth-limit=1]] by(induction t arbitrary: s) auto
```

105.9 *ipl*

The internal path length of a tree:

```
lemma ipl-if-complete-int:
  complete t ⟹ int(ipl t) = (int(height t) − 2) ∗ 2^(height t) + 2
apply(induction t)
 apply simp
apply simp
apply (simp add: algebra-simps size-if-complete of-nat-diff)
done
```

105.10 List of entries

```
lemma eq-inorder-Nil[simp]: inorder t = [] ⟷ t = Leaf
by (cases t) auto

lemmas eq-Nil-inorder[simp] = eq-inorder-Nil[THEN eq-iff-swap]

lemma set-inorder[simp]: set (inorder t) = set-tree t
by (induction t) auto
```

lemma *preorder-eq-Nil-iff*[*simp*]: (*preorder t* = []) = (*t* = ⟨⟩)
by (*cases t*) *auto*

lemmas *Nil-eq-preorder-iff* [*simp*] = *preorder-eq-Nil-iff*[*THEN eq-iff-swap*]

lemma *preorder-eq-Cons-iff*:
preorder t = *x* # *xs* ⟷ (∃ *l r*. *t* = ⟨*l*, *x*, *r*⟩ ∧ *xs* = *preorder l* @ *preorder r*)
by (*cases t*) *auto*

lemmas *Cons-eq-preorder-iff* = *preorder-eq-Cons-iff*[*THEN eq-iff-swap*]

lemma *set-preorder*[*simp*]: *set* (*preorder t*) = *set-tree t*
by (*induction t*) *auto*

lemma *set-postorder*[*simp*]: *set* (*postorder t*) = *set-tree t*
by (*induction t*) *auto*

lemma *length-preorder*[*simp*]: *length* (*preorder t*) = *size t*
by (*induction t*) *auto*

lemma *length-inorder*[*simp*]: *length* (*inorder t*) = *size t*
by (*induction t*) *auto*

lemma *length-postorder*[*simp*]: *length* (*postorder t*) = *size t*
by (*induction t*) *auto*

lemma *preorder-map*: *preorder* (*map-tree f t*) = *map f* (*preorder t*)
by (*induction t*) *auto*

lemma *inorder-map*: *inorder* (*map-tree f t*) = *map f* (*inorder t*)
by (*induction t*) *auto*

lemma *postorder-map*: *postorder* (*map-tree f t*) = *map f* (*postorder t*)
by (*induction t*) *auto*

lemma *inorder2-inorder*: *inorder2 t xs* = *inorder t* @ *xs*
by (*induction t arbitrary*: *xs*) *auto*

105.11 Binary Search Tree

lemma *bst-wrt-mono*: (⋀*x y*. *P x y* ⟹ *Q x y*) ⟹ *bst-wrt P t* ⟹ *bst-wrt Q t*
by (*induction t*) (*auto*)

lemma *bst-wrt-le-if-bst*: *bst t* ⟹ *bst-wrt* (≤) *t*
using *bst-wrt-mono less-imp-le* **by** *blast*

lemma *bst-wrt-le-iff-sorted*: *bst-wrt* (≤) *t* ⟷ *sorted* (*inorder t*)
apply (*induction t*)
apply(*simp*)

by (*fastforce simp*: *sorted-append intro*: *less-imp-le less-trans*)

lemma *bst-iff-sorted-wrt-less*: *bst t* ⟷ *sorted-wrt* (<) (*inorder t*)
apply (*induction t*)
apply *simp*
apply (*fastforce simp*: *sorted-wrt-append*)
done

105.12 *heap*

105.13 *mirror*

lemma *mirror-Leaf*[*simp*]: *mirror t* = ⟨⟩ ⟷ *t* = ⟨⟩
by (*induction t*) *simp-all*

lemma *Leaf-mirror*[*simp*]: ⟨⟩ = *mirror t* ⟷ *t* = ⟨⟩
using *mirror-Leaf* **by** *fastforce*

lemma *size-mirror*[*simp*]: *size*(*mirror t*) = *size t*
by (*induction t*) *simp-all*

lemma *size1-mirror*[*simp*]: *size1*(*mirror t*) = *size1 t*
by (*simp add*: *size1-size*)

lemma *height-mirror*[*simp*]: *height*(*mirror t*) = *height t*
by (*induction t*) *simp-all*

lemma *min-height-mirror* [*simp*]: *min-height* (*mirror t*) = *min-height t*
by (*induction t*) *simp-all*

lemma *ipl-mirror* [*simp*]: *ipl* (*mirror t*) = *ipl t*
by (*induction t*) *simp-all*

lemma *inorder-mirror*: *inorder*(*mirror t*) = *rev*(*inorder t*)
by (*induction t*) *simp-all*

lemma *map-mirror*: *map-tree f* (*mirror t*) = *mirror* (*map-tree f t*)
by (*induction t*) *simp-all*

lemma *mirror-mirror*[*simp*]: *mirror*(*mirror t*) = *t*
by (*induction t*) *simp-all*

end

106 Multiset of Elements of Binary Tree

theory *Tree-Multiset*
imports *Multiset Tree*
begin

Kept separate from theory *HOL−Library.Tree* to avoid importing all of theory *HOL−Library.Multiset* into *HOL−Library.Tree*. Should be merged if *HOL−Library.Multiset* ever becomes part of *Main*.

fun *mset-tree* :: *'a tree* ⇒ *'a multiset* **where**
mset-tree Leaf = {#} |
mset-tree (*Node l a r*) = {#*a*#} + *mset-tree l* + *mset-tree r*

fun *subtrees-mset* :: *'a tree* ⇒ *'a tree multiset* **where**
subtrees-mset Leaf = {#*Leaf*#} |
subtrees-mset (*Node l x r*) = *add-mset* (*Node l x r*) (*subtrees-mset l* + *subtrees-mset r*)

lemma *mset-tree-empty-iff*[*simp*]: *mset-tree t* = {#} ⟷ *t* = *Leaf*
by (*cases t*) *auto*

lemma *set-mset-tree*[*simp*]: *set-mset* (*mset-tree t*) = *set-tree t*
by(*induction t*) *auto*

lemma *size-mset-tree*[*simp*]: *size*(*mset-tree t*) = *size t*
by(*induction t*) *auto*

lemma *mset-map-tree*: *mset-tree* (*map-tree f t*) = *image-mset f* (*mset-tree t*)
by (*induction t*) *auto*

lemma *mset-iff-set-tree*: *x* ∈# *mset-tree t* ⟷ *x* ∈ *set-tree t*
by(*induction t arbitrary*: *x*) *auto*

lemma *mset-preorder*[*simp*]: *mset* (*preorder t*) = *mset-tree t*
by (*induction t*) (*auto simp*: *ac-simps*)

lemma *mset-inorder*[*simp*]: *mset* (*inorder t*) = *mset-tree t*
by (*induction t*) (*auto simp*: *ac-simps*)

lemma *map-mirror*: *mset-tree* (*mirror t*) = *mset-tree t*
by (*induction t*) (*simp-all add*: *ac-simps*)

lemma *in-subtrees-mset-iff*[*simp*]: *s* ∈# *subtrees-mset t* ⟷ *s* ∈ *subtrees t*
by(*induction t*) *auto*

end

theory *Tree-Real*
imports
Complex-Main
Tree
begin

This theory is separate from *HOL−Library.Tree* because the former is discrete and builds on *Main* whereas this theory builds on *Complex-Main*.

lemma *size1-height-log*: *log 2 (size1 t) ≤ height t*
by (*simp add*: *log2-of-power-le size1-height*)

lemma *min-height-size1-log*: *min-height t ≤ log 2 (size1 t)*
by (*simp add*: *le-log2-of-power min-height-size1*)

lemma *size1-log-if-complete*: *complete t ⟹ height t = log 2 (size1 t)*
by (*simp add*: *size1-if-complete*)

lemma *min-height-size1-log-if-incomplete*:
¬ complete t ⟹ min-height t < log 2 (size1 t)
by (*simp add*: *less-log2-of-power min-height-size1-if-incomplete*)

lemma *min-height-acomplete*: **assumes** *acomplete t*
shows *min-height t = nat(floor(log 2 (size1 t)))*
proof *cases*
assume ∗: *complete t*
hence *size1 t = 2 ^ min-height t*
by (*simp add*: *complete-iff-height size1-if-complete*)
from *log2-of-power-eq*[*OF this*] **show** *?thesis* **by** *linarith*
next
assume ∗: *¬ complete t*
hence *height t = min-height t + 1*
using *assms min-height-le-height*[*of t*]
by(*auto simp*: *acomplete-def complete-iff-height*)
hence *size1 t < 2 ^ (min-height t + 1)* **by** (*metis* ∗ *size1-height-if-incomplete*)
from *floor-log-nat-eq-if*[*OF min-height-size1 this*] **show** *?thesis* **by** *simp*
qed

lemma *height-acomplete*: **assumes** *acomplete t*
shows *height t = nat(ceiling(log 2 (size1 t)))*
proof *cases*
assume ∗: *complete t*
hence *size1 t = 2 ^ height t* **by** (*simp add*: *size1-if-complete*)
from *log2-of-power-eq*[*OF this*] **show** *?thesis* **by** *linarith*
next
assume ∗: *¬ complete t*
hence ∗∗: *height t = min-height t + 1*
using *assms min-height-le-height*[*of t*]
by(*auto simp add*: *acomplete-def complete-iff-height*)
hence *size1 t ≤ 2 ^ (min-height t + 1)* **by** (*metis size1-height*)
from *log2-of-power-le*[*OF this size1-ge0*] *min-height-size1-log-if-incomplete*[*OF* ∗]
∗∗
show *?thesis* **by** *linarith*
qed

lemma *acomplete-Node-if-wbal1*:
assumes *acomplete l acomplete r size l = size r + 1*
shows *acomplete ⟨l, x, r⟩*
proof −
from *assms(3)* **have** [*simp*]: *size1 l = size1 r + 1* **by**(*simp add*: *size1-size*)
have *nat ⌈log 2 (1 + size1 r)⌉ ≥ nat ⌈log 2 (size1 r)⌉*
by(*rule nat-mono*[*OF ceiling-mono*]) *simp*
hence *1*: *height*(*Node l x r*) = *nat ⌈log 2 (1 + size1 r)⌉ + 1*
using *height-acomplete*[*OF assms(1)*] *height-acomplete*[*OF assms(2)*]
by (*simp del*: *nat-ceiling-le-eq add*: *max-def*)
have *nat ⌊log 2 (1 + size1 r)⌋ ≥ nat ⌊log 2 (size1 r)⌋*
by(*rule nat-mono*[*OF floor-mono*]) *simp*
hence *2*: *min-height*(*Node l x r*) = *nat ⌊log 2 (size1 r)⌋ + 1*
using *min-height-acomplete*[*OF assms(1)*] *min-height-acomplete*[*OF assms(2)*]
by (*simp*)
have *size1 r ≥ 1* **by**(*simp add*: *size1-size*)
then obtain *i* **where** *i*: *2 ^ i ≤ size1 r size1 r < 2 ^ (i + 1)*
using *ex-power-ivl1*[*of 2 size1 r*] **by** *auto*
hence *i1*: *2 ^ i < size1 r + 1 size1 r + 1 ≤ 2 ^ (i + 1)* **by** *auto*
from *1 2 floor-log-nat-eq-if*[*OF i*] *ceiling-log-nat-eq-if*[*OF i1*]
show *?thesis* **by**(*simp add*:*acomplete-def*)
qed

lemma *acomplete-sym*: *acomplete ⟨l, x, r⟩ ⟹ acomplete ⟨r, y, l⟩*
by(*auto simp*: *acomplete-def*)

lemma *acomplete-Node-if-wbal2*:
assumes *acomplete l acomplete r abs*(*int*(*size l*) − *int*(*size r*)) ≤ *1*
shows *acomplete ⟨l, x, r⟩*
proof −
have *size l = size r ∨ (size l = size r + 1 ∨ size r = size l + 1)* (**is** *?A ∨ ?B*)
using *assms(3)* **by** *linarith*
thus *?thesis*
proof
assume *?A*
thus *?thesis* **using** *assms(1,2)*
apply(*simp add*: *acomplete-def min-def max-def*)
by (*metis assms(1,2) acomplete-optimal le-antisym le-less*)
next
assume *?B*
thus *?thesis*
by (*meson assms(1,2) acomplete-sym acomplete-Node-if-wbal1*)
qed
qed

lemma *acomplete-if-wbalanced*: *wbalanced t ⟹ acomplete t*
proof(*induction t*)
case *Leaf* **show** *?case* **by** (*simp add*: *acomplete-def*)
next

case (*Node l x r*)
thus *?case* **by**(*simp add*: *acomplete-Node-if-wbal2*)
qed

end

107 Unordered pairs

theory *Uprod* **imports** *Main* **begin**

typedef ('*a*, '*b*) *commute* = {*f* :: '*a* ⇒ '*a* ⇒ '*b*. ∀ *x y*. *f x y* = *f y x*}
morphisms *apply-commute Abs-commute*
by *auto*

setup-lifting *type-definition-commute*

lemma *apply-commute-commute*: *apply-commute f x y* = *apply-commute f y x*
by(*transfer*) *simp*

context includes *lifting-syntax* **begin**

lift-definition *rel-commute* :: ('*a* ⇒ '*b* ⇒ *bool*) ⇒ ('*c* ⇒ '*d* ⇒ *bool*) ⇒ ('*a*, '*c*) *commute* ⇒ ('*b*, '*d*) *commute* ⇒ *bool*
is λ*A B*. *A* ===> *A* ===> *B* **.**

end

definition *eq-upair* :: ('*a* × '*a*) ⇒ ('*a* × '*a*) ⇒ *bool*
where *eq-upair* = (λ(*a*, *b*) (*c*, *d*). *a* = *c* ∧ *b* = *d* ∨ *a* = *d* ∧ *b* = *c*)

lemma *eq-upair-simps* [*simp*]:
eq-upair (*a*, *b*) (*c*, *d*) ⟷ *a* = *c* ∧ *b* = *d* ∨ *a* = *d* ∧ *b* = *c*
by(*simp add*: *eq-upair-def*)

lemma *equivp-eq-upair*: *equivp eq-upair*
by(*auto simp add*: *equivp-def fun-eq-iff*)

quotient-type '*a uprod* = '*a* × '*a* / *eq-upair* **by**(*rule equivp-eq-upair*)

lift-definition *Upair* :: '*a* ⇒ '*a* ⇒ '*a uprod* **is** *Pair* **parametric** *Pair-transfer*[*of A A* **for** *A*] **.**

lemma *uprod-exhaust* [*case-names Upair*, *cases type*: *uprod*]:
obtains *a b* **where** *x* = *Upair a b*
by *transfer fastforce*

lemma *Upair-inject* [*simp*]: *Upair a b* = *Upair c d* ⟷ *a* = *c* ∧ *b* = *d* ∨ *a* = *d* ∧ *b* = *c*
by *transfer auto*

code-datatype *Upair*

lift-definition *case-uprod* :: ('*a*, '*b*) *commute* ⇒ '*a uprod* ⇒ '*b* **is** *case-prod*
  parametric *case-prod-transfer*[*of A A* **for** *A*] **by** *auto*

lemma *case-uprod-simps* [*simp*, *code*]: *case-uprod f* (*Upair x y*) = *apply-commute f x y*
by *transfer auto*

lemma *uprod-split*: *P* (*case-uprod f x*) ⟷ (∀ *a b*. *x* = *Upair a b* ⟶ *P* (*apply-commute f a b*))
by *transfer auto*

lemma *uprod-split-asm*: *P* (*case-uprod f x*) ⟷ ¬ (∃ *a b*. *x* = *Upair a b* ∧ ¬ *P* (*apply-commute f a b*))
by *transfer auto*

lift-definition *not-equal* :: ('*a*, *bool*) *commute* **is** (≠) **by** *auto*

lemma *apply-not-equal* [*simp*]: *apply-commute not-equal x y* ⟷ *x* ≠ *y*
by *transfer simp*

definition *proper-uprod* :: '*a uprod* ⇒ *bool*
where *proper-uprod* = *case-uprod not-equal*

lemma *proper-uprod-simps* [*simp*, *code*]: *proper-uprod* (*Upair x y*) ⟷ *x* ≠ *y*
by(*simp add*: *proper-uprod-def*)

context includes *lifting-syntax* **begin**

private lemma *set-uprod-parametric'*:
  (*rel-prod A A* ===> *rel-set A*) (λ(*a*, *b*). {*a*, *b*}) (λ(*a*, *b*). {*a*, *b*})
by *transfer-prover*

lift-definition *set-uprod* :: '*a uprod* ⇒ '*a set* **is** λ(*a*, *b*). {*a*, *b*}
  parametric *set-uprod-parametric'* **by** *auto*

lemma *set-uprod-simps* [*simp*, *code*]: *set-uprod* (*Upair x y*) = {*x*, *y*}
by *transfer simp*

lemma *finite-set-uprod* [*simp*]: *finite* (*set-uprod x*)
by(*cases x*) *simp*

private lemma *map-uprod-parametric'*:
  ((*A* ===> *B*) ===> *rel-prod A A* ===> *rel-prod B B*) (λ*f*. *map-prod f f*) (λ*f*. *map-prod f f*)
by *transfer-prover*

lift-definition *map-uprod* :: $('a \Rightarrow 'b) \Rightarrow 'a$ *uprod* $\Rightarrow 'b$ *uprod* **is** $\lambda f.$ *map-prod f f*
parametric *map-uprod-parametric'* **by** *auto*

lemma *map-uprod-simps* [*simp*, *code*]: *map-uprod f* (*Upair x y*) = *Upair* (*f x*) (*f y*)
by *transfer simp*

private lemma *rel-uprod-transfer'*:
((*A* ===> *B* ===> (=)) ===> *rel-prod A A* ===> *rel-prod B B* ===> (=))
(λR (*a*, *b*) (*c*, *d*). *R a c* $\wedge$ *R b d* $\vee$ *R a d* $\wedge$ *R b c*) (λR (*a*, *b*) (*c*, *d*). *R a c* $\wedge$ *R b d* $\vee$ *R a d* $\wedge$ *R b c*)
by *transfer-prover*

lift-definition *rel-uprod* :: $('a \Rightarrow 'b \Rightarrow$ *bool*$) \Rightarrow 'a$ *uprod* $\Rightarrow 'b$ *uprod* $\Rightarrow$ *bool*
is λR (*a*, *b*) (*c*, *d*). *R a c* $\wedge$ *R b d* $\vee$ *R a d* $\wedge$ *R b c* **parametric** *rel-uprod-transfer'*
by *auto*

lemma *rel-uprod-simps* [*simp*, *code*]:
rel-uprod R (*Upair a b*) (*Upair c d*) $\longleftrightarrow$ *R a c* $\wedge$ *R b d* $\vee$ *R a d* $\wedge$ *R b c*
by *transfer auto*

lemma *Upair-parametric* [*transfer-rule*]: (*A* ===> *A* ===> *rel-uprod A*) *Upair Upair*
unfolding *rel-fun-def* **by** *transfer auto*

lemma *case-uprod-parametric* [*transfer-rule*]:
(*rel-commute A B* ===> *rel-uprod A* ===> *B*) *case-uprod case-uprod*
unfolding *rel-fun-def* **by** *transfer*(*force dest*: *rel-funD*)

end

bnf *uprod*: *'a uprod*
map: *map-uprod*
sets: *set-uprod*
bd: *natLeq*
rel: *rel-uprod*
proof −
show *map-uprod id* = *id* **unfolding** *fun-eq-iff* **by** *transfer auto*
show *map-uprod* (*g* ∘ *f*) = *map-uprod g* ∘ *map-uprod f* **for** *f* :: $'a \Rightarrow 'b$ **and** *g* :: $'b \Rightarrow 'c$
unfolding *fun-eq-iff* **by** *transfer auto*
show *map-uprod f x* = *map-uprod g x* **if** $\bigwedge z.$ *z* ∈ *set-uprod x* $\Longrightarrow$ *f z* = *g z*
for *f* :: $'a \Rightarrow 'b$ **and** *g x* **using** *that* **by** *transfer auto*
show *set-uprod* ∘ *map-uprod f* = (‘) *f* ∘ *set-uprod* **for** *f* :: $'a \Rightarrow 'b$ **by** *transfer auto*
show *card-order natLeq* **by**(*rule natLeq-card-order*)
show *BNF-Cardinal-Arithmetic.cinfinite natLeq* **by**(*rule natLeq-cinfinite*)
show *regularCard natLeq* **by**(*rule regularCard-natLeq*)

show *ordLess2* (*card-of* (*set-uprod x*)) *natLeq* **for** *x* :: *'a uprod*
by (*auto simp flip*: *finite-iff-ordLess-natLeq*)
show *rel-uprod R OO rel-uprod S* ≤ *rel-uprod* (*R OO S*)
for *R* :: *'a* ⇒ *'b* ⇒ *bool* **and** *S* :: *'b* ⇒ *'c* ⇒ *bool* **by**(*rule predicate2I*)(*transfer*; *auto*)
show *rel-uprod R* = (λ*x y*. ∃ *z*. *set-uprod z* ⊆ {(*x*, *y*). *R x y*} ∧ *map-uprod fst z* = *x* ∧ *map-uprod snd z* = *y*)
for *R* :: *'a* ⇒ *'b* ⇒ *bool* **by** *transfer*(*auto simp add*: *fun-eq-iff*)
qed

lemma *pred-uprod-code* [*simp*, *code*]: *pred-uprod P* (*Upair x y*) ⟷ *P x* ∧ *P y*
by(*simp add*: *pred-uprod-def*)

instantiation *uprod* :: (*equal*) *equal* **begin**

definition *equal-uprod* :: *'a uprod* ⇒ *'a uprod* ⇒ *bool*
where *equal-uprod* = (=)

lemma *equal-uprod-code* [*code*]:
HOL.equal (*Upair x y*) (*Upair z u*) ⟷ *x* = *z* ∧ *y* = *u* ∨ *x* = *u* ∧ *y* = *z*
unfolding *equal-uprod-def* **by** *simp*

instance by *standard*(*simp add*: *equal-uprod-def*)
end

quickcheck-generator *uprod constructors*: *Upair*

lemma *UNIV-uprod*: *UNIV* = (λ*x*. *Upair x x*) ‘ *UNIV* ∪ (λ(*x*, *y*). *Upair x y*) ‘ *Sigma UNIV* (λ*x*. *UNIV* − {*x*})
apply(*rule set-eqI*)
subgoal for *x* **by**(*cases x*) *auto*
done

context begin
private lift-definition *upair-inv* :: *'a uprod* ⇒ *'a*
is λ(*x*, *y*). *if x* = *y then x else undefined* **by** *auto*

lemma *finite-UNIV-prod* [*simp*]:
finite (*UNIV* :: *'a uprod set*) ⟷ *finite* (*UNIV* :: *'a set*) (**is** *?lhs* = *?rhs*)
proof
assume *?lhs*
hence *finite* (*range* (λ*x* :: *'a. Upair x x*)) **by**(*rule finite-subset*[*rotated*]) *simp*
hence *finite* (*upair-inv* ‘ *range* (λ*x* :: *'a. Upair x x*)) **by**(*rule finite-imageI*)
also have *upair-inv* (*Upair x x*) = *x* **for** *x* :: *'a* **by** *transfer simp*
then have *upair-inv* ‘ *range* (λ*x* :: *'a. Upair x x*) = *UNIV* **by**(*auto simp add*: *image-image*)
finally show *?rhs* **.**
qed(*simp add*: *UNIV-uprod*)

end

lemma *card-UNIV-uprod*:
card (*UNIV* :: *'a uprod set*) = *card* (*UNIV* :: *'a set*) * (*card* (*UNIV* :: *'a set*) + *1*) *div 2*
(**is** *?UPROD* = *?A* * - *div* -)
proof(*cases finite* (*UNIV* :: *'a set*))
case *True*
from *True* **obtain** *f* :: *nat* ⇒ *'a* **where** *bij*: *bij-betw f* {*0*..<*?A*} *UNIV*
by (*blast dest*: *ex-bij-betw-nat-finite*)
hence [*simp*]: *f* ' {*0*..<*?A*} = *UNIV* **by**(*rule bij-betw-imp-surj-on*)
have *UNIV* = (λ(*x*, *y*). *Upair* (*f x*) (*f y*)) ' (*SIGMA x*:{*0*..<*?A*}. {..*x*})
apply(*rule set-eqI*)
subgoal for *x*
apply(*cases x*)
apply(*clarsimp*)
subgoal for *a b*
apply(*cases inv-into* {*0*..<*?A*} *f a* ≤ *inv-into* {*0*..<*?A*} *f b*)
subgoal by(*rule rev-image-eqI*[**where** *x*=(*inv-into* {*0*..<*?A*} *f* -, *inv-into* {*0*..<*?A*} *f* -)])
(*auto simp add*: *inv-into-into*[**where** *A*={*0*..<*?A*} **and** *f*=*f*, *simplified*] *intro*: *f-inv-into-f*[**where** *f*=*f*, *symmetric*])
subgoal
apply(*simp only*: *not-le*)
apply(*drule less-imp-le*)
apply(*rule rev-image-eqI*[**where** *x*=(*inv-into* {*0*..<*?A*} *f* -, *inv-into* {*0*..<*?A*} *f* -)])
apply(*auto simp add*: *inv-into-into*[**where** *A*={*0*..<*?A*} **and** *f*=*f*, *simplified*] *intro*: *f-inv-into-f*[**where** *f*=*f*, *symmetric*])
done
done
done
done
hence *?UPROD* = *card* ... **by** *simp*
also have ... = *card* (*SIGMA x*:{*0*..<*?A*}. {..*x*})
apply(*rule card-image*)
using *bij*[*THEN bij-betw-imp-inj-on*]
by(*simp add*: *inj-on-def Ball-def*)(*metis leD le-eq-less-or-eq le-less-trans*)
also have ... = *sum Suc* {*0*..<*?A*}
by (*subst card-SigmaI*) *simp-all*
also have ... = *sum of-nat* {*Suc 0*..*?A*}
using *sum.atLeastLessThan-reindex* [*symmetric*, *of Suc 0 ?A id*]
by (*simp del*: *sum.op-ivl-Suc add*: *atLeastLessThanSuc-atLeastAtMost*)
also have ... = *?A* * (*?A* + *1*) *div 2*
using *gauss-sum-from-Suc-0* [*of ?A*, **where** *?'a* = *nat*] **by** *simp*
finally show *?thesis* **.**
qed *simp*

end

108 A type of finite bit strings

theory *Word*
imports
HOL−Library.Type-Length
begin

108.1 Preliminaries

lemma *signed-take-bit-decr-length-iff*:
‹*signed-take-bit* (*LENGTH*('*a*::*len*) − *Suc 0*) *k* = *signed-take-bit* (*LENGTH*('*a*) − *Suc 0*) *l*
⟷ *take-bit LENGTH*('*a*) *k* = *take-bit LENGTH*('*a*) *l*›
by (*simp add*: *signed-take-bit-eq-iff-take-bit-eq*)

108.2 Fundamentals

108.2.1 Type definition

quotient-type (**overloaded**) '*a word* = *int* / ‹λ*k l*. *take-bit LENGTH*('*a*) *k* = *take-bit LENGTH*('*a*::*len*) *l*›
morphisms *rep Word* **by** (*auto intro*!: *equivpI reflpI sympI transpI*)

hide-const (**open**) *rep* — only for foundational purpose
hide-const (**open**) *Word* — only for code generation

108.2.2 Basic arithmetic

instantiation *word* :: (*len*) *comm-ring-1*
begin

lift-definition *zero-word* :: ‹'*a word*›
is *0* .

lift-definition *one-word* :: ‹'*a word*›
is *1* .

lift-definition *plus-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is ‹(+)›
by (*auto simp*: *take-bit-eq-mod intro*: *mod-add-cong*)

lift-definition *minus-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is ‹(−)›
by (*auto simp*: *take-bit-eq-mod intro*: *mod-diff-cong*)

lift-definition *uminus-word* :: ‹'*a word* ⇒ '*a word*›
is *uminus*
by (*auto simp*: *take-bit-eq-mod intro*: *mod-minus-cong*)

lift-definition *times-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›

is ‹(∗)›
by (*auto simp*: *take-bit-eq-mod intro*: *mod-mult-cong*)

instance
by (*standard*; *transfer*) (*simp-all add*: *algebra-simps*)

end

context
includes *lifting-syntax*
notes
power-transfer [*transfer-rule*]
transfer-rule-of-bool [*transfer-rule*]
transfer-rule-numeral [*transfer-rule*]
transfer-rule-of-nat [*transfer-rule*]
transfer-rule-of-int [*transfer-rule*]
begin

lemma *power-transfer-word* [*transfer-rule*]:
‹(*pcr-word* ===> (=) ===> *pcr-word*) (^) (^)›
by *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *of-bool of-bool*›
by *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *numeral numeral*›
by *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) *int of-nat*›
by *transfer-prover*

lemma [*transfer-rule*]:
‹((=) ===> *pcr-word*) (λ*k*. *k*) *of-int*›
proof −
have ‹((=) ===> *pcr-word*) *of-int of-int*›
by *transfer-prover*
then show *?thesis* **by** (*simp add*: *id-def*)
qed

lemma [*transfer-rule*]:
‹(*pcr-word* ===> (⟷)) *even* ((*dvd*) *2* :: ′*a*::*len word* ⇒ *bool*)›
proof −
have *even-word-unfold*: *even k* ⟷ (∃*l*. *take-bit LENGTH*(′*a*) *k* = *take-bit LENGTH*(′*a*) (*2* ∗ *l*)) (**is** *?P* ⟷ *?Q*)
for *k* :: *int*
by (*metis dvd-triv-left evenE even-take-bit-eq len-not-eq-0*)

show *?thesis*
unfolding *even-word-unfold* [*abs-def*] *dvd-def* [**where** *?'a = 'a word*, *abs-def*]
by *transfer-prover*
qed

end

lemma *exp-eq-zero-iff* [*simp*]:
‹*2 ^ n = (0 :: 'a::len word) ⟷ n ≥ LENGTH('a)*›
by *transfer auto*

lemma *word-exp-length-eq-0* [*simp*]:
‹*(2 :: 'a::len word) ^ LENGTH('a) = 0*›
by *simp*

108.2.3 Basic tool setup

ML-file ‹*Tools/word-lib.ML*›

108.2.4 Basic code generation setup

context
begin

qualified lift-definition *the-int* :: ‹*'a::len word ⇒ int*›
is ‹*take-bit LENGTH('a)*› **.**

end

lemma [*code abstype*]:
‹*Word.Word (Word.the-int w) = w*›
by *transfer simp*

lemma *Word-eq-word-of-int* [*code-post*, *simp*]:
‹*Word.Word = of-int*›
by (*rule*; *transfer*) *simp*

quickcheck-generator *word*
constructors:
‹*0 :: 'a::len word*›,
‹*numeral :: num ⇒ 'a::len word*›

instantiation *word* :: (*len*) *equal*
begin

lift-definition *equal-word* :: ‹*'a word ⇒ 'a word ⇒ bool*›
is ‹*λk l. take-bit LENGTH('a) k = take-bit LENGTH('a) l*›
by *simp*

instance

by (*standard*; *transfer*) *rule*

end

lemma [*code*]:
‹*HOL.equal v w* ⟷ *HOL.equal* (*Word.the-int v*) (*Word.the-int w*)›
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
‹*Word.the-int 0* = *0*›
by *transfer simp*

lemma [*code*]:
‹*Word.the-int 1* = *1*›
by *transfer simp*

lemma [*code*]:
‹*Word.the-int* (*v* + *w*) = *take-bit LENGTH*('*a*) (*Word.the-int v* + *Word.the-int w*)›
for *v w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *take-bit-add*)

lemma [*code*]:
‹*Word.the-int* (− *w*) = (*let k* = *Word.the-int w in if w* = *0 then 0 else 2* ^ *LENGTH*('*a*) − *k*)›
for *w* :: ‹'*a*::*len word*›
by *transfer* (*auto simp*: *take-bit-eq-mod zmod-zminus1-eq-if*)

lemma [*code*]:
‹*Word.the-int* (*v* − *w*) = *take-bit LENGTH*('*a*) (*Word.the-int v* − *Word.the-int w*)›
for *v w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *take-bit-diff*)

lemma [*code*]:
‹*Word.the-int* (*v* ∗ *w*) = *take-bit LENGTH*('*a*) (*Word.the-int v* ∗ *Word.the-int w*)›
for *v w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *take-bit-mult*)

108.2.5 Basic conversions

abbreviation *word-of-nat* :: ‹*nat* ⇒ '*a*::*len word*›
where ‹*word-of-nat* ≡ *of-nat*›

abbreviation *word-of-int* :: ‹*int* ⇒ '*a*::*len word*›
where ‹*word-of-int* ≡ *of-int*›

lemma *word-of-nat-eq-iff*:

‹*word-of-nat m = (word-of-nat n :: 'a::len word) ⟷ take-bit LENGTH('a) m = take-bit LENGTH('a) n*›
by *transfer* (*simp add*: *take-bit-of-nat*)

lemma *word-of-int-eq-iff*:
‹*word-of-int k = (word-of-int l :: 'a::len word) ⟷ take-bit LENGTH('a) k = take-bit LENGTH('a) l*›
by *transfer rule*

lemma *word-of-nat-eq-0-iff*:
‹*word-of-nat n = (0 :: 'a::len word) ⟷ 2 ^ LENGTH('a) dvd n*›
using *word-of-nat-eq-iff* [**where** *?'a = 'a*, *of n 0*] **by** (*simp add*: *take-bit-eq-0-iff*)

lemma *word-of-int-eq-0-iff*:
‹*word-of-int k = (0 :: 'a::len word) ⟷ 2 ^ LENGTH('a) dvd k*›
using *word-of-int-eq-iff* [**where** *?'a = 'a*, *of k 0*] **by** (*simp add*: *take-bit-eq-0-iff*)

context *semiring-1*
begin

lift-definition *unsigned* :: ‹*'b::len word ⇒ 'a*›
is ‹*of-nat ∘ nat ∘ take-bit LENGTH('b)*›
by *simp*

lemma *unsigned-0* [*simp*]:
‹*unsigned 0 = 0*›
by *transfer simp*

lemma *unsigned-1* [*simp*]:
‹*unsigned 1 = 1*›
by *transfer simp*

lemma *unsigned-numeral* [*simp*]:
‹*unsigned (numeral n :: 'b::len word) = of-nat (take-bit LENGTH('b) (numeral n))*›
by *transfer* (*simp add*: *nat-take-bit-eq*)

lemma *unsigned-neg-numeral* [*simp*]:
‹*unsigned (- numeral n :: 'b::len word) = of-nat (nat (take-bit LENGTH('b) (- numeral n)))*›
by *transfer simp*

end

context *semiring-1*
begin

lemma *unsigned-of-nat*:
‹*unsigned (word-of-nat n :: 'b::len word) = of-nat (take-bit LENGTH('b) n)*›

by *transfer* (*simp add*: *nat-eq-iff take-bit-of-nat*)

lemma *unsigned-of-int*:
‹*unsigned* (*word-of-int k* :: *'b*::*len word*) = *of-nat* (*nat* (*take-bit LENGTH*('*b*) *k*))›
by *transfer simp*

end

context *semiring-char-0*
begin

lemma *unsigned-word-eqI*:
‹*v* = *w*› **if** ‹*unsigned v* = *unsigned w*›
using *that* **by** *transfer* (*simp add*: *eq-nat-nat-iff*)

lemma *word-eq-iff-unsigned*:
‹*v* = *w* ⟷ *unsigned v* = *unsigned w*›
by (*auto intro*: *unsigned-word-eqI*)

lemma *inj-unsigned* [*simp*]:
‹*inj unsigned*›
by (*rule injI*) (*simp add*: *unsigned-word-eqI*)

lemma *unsigned-eq-0-iff*:
‹*unsigned w* = *0* ⟷ *w* = *0*›
using *word-eq-iff-unsigned* [*of w 0*] **by** *simp*

end

context *ring-1*
begin

lift-definition *signed* :: ‹*'b*::*len word* ⇒ *'a*›
is ‹*of-int* ∘ *signed-take-bit* (*LENGTH*('*b*) − *Suc 0*)›
by (*simp flip*: *signed-take-bit-decr-length-iff*)

lemma *signed-0* [*simp*]:
‹*signed 0* = *0*›
by *transfer simp*

lemma *signed-1* [*simp*]:
‹*signed* (*1* :: *'b*::*len word*) = (*if LENGTH*('*b*) = *1 then* − *1 else 1*)›
by (*transfer fixing*: *uminus*; *cases* ‹*LENGTH*('*b*)›) (*auto dest*: *gr0-implies-Suc*)

lemma *signed-minus-1* [*simp*]:
‹*signed* (− *1* :: *'b*::*len word*) = − *1*›
by (*transfer fixing*: *uminus*) *simp*

lemma *signed-numeral* [*simp*]:

‹*signed* (*numeral n* :: *'b*::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('*b*) − *1*) (*numeral n*))›
by *transfer simp*

lemma *signed-neg-numeral* [*simp*]:
‹*signed* (− *numeral n* :: *'b*::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('*b*) − *1*) (− *numeral n*))›
by *transfer simp*

lemma *signed-of-nat*:
‹*signed* (*word-of-nat n* :: *'b*::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('*b*) − *Suc 0*) (*int n*))›
by *transfer simp*

lemma *signed-of-int*:
‹*signed* (*word-of-int n* :: *'b*::*len word*) = *of-int* (*signed-take-bit* (*LENGTH*('*b*) − *Suc 0*) *n*)›
by *transfer simp*

end

context *ring-char-0*
begin

lemma *signed-word-eqI*:
‹*v* = *w*› **if** ‹*signed v* = *signed w*›
using *that* **by** *transfer* (*simp flip*: *signed-take-bit-decr-length-iff*)

lemma *word-eq-iff-signed*:
‹*v* = *w* ⟷ *signed v* = *signed w*›
by (*auto intro*: *signed-word-eqI*)

lemma *inj-signed* [*simp*]:
‹*inj signed*›
by (*rule injI*) (*simp add*: *signed-word-eqI*)

lemma *signed-eq-0-iff*:
‹*signed w* = *0* ⟷ *w* = *0*›
using *word-eq-iff-signed* [*of w 0*] **by** *simp*

end

abbreviation *unat* :: ‹*'a*::*len word* ⇒ *nat*›
where ‹*unat* ≡ *unsigned*›

abbreviation *uint* :: ‹*'a*::*len word* ⇒ *int*›
where ‹*uint* ≡ *unsigned*›

abbreviation *sint* :: ‹*'a*::*len word* ⇒ *int*›

where ‹*sint* ≡ *signed*›

abbreviation *ucast* :: ‹'*a*::*len word* ⇒ '*b*::*len word*›
where ‹*ucast* ≡ *unsigned*›

abbreviation *scast* :: ‹'*a*::*len word* ⇒ '*b*::*len word*›
where ‹*scast* ≡ *signed*›

context
includes *lifting-syntax*
begin

lemma [*transfer-rule*]:
‹(*pcr-word* ===> (=)) (*nat* ∘ *take-bit LENGTH*('*a*)) (*unat* :: '*a*::*len word* ⇒ *nat*)›
using *unsigned.transfer* [**where** *?'a* = *nat*] **by** *simp*

lemma [*transfer-rule*]:
‹(*pcr-word* ===> (=)) (*take-bit LENGTH*('*a*)) (*uint* :: '*a*::*len word* ⇒ *int*)›
using *unsigned.transfer* [**where** *?'a* = *int*] **by** (*simp add*: *comp-def*)

lemma [*transfer-rule*]:
‹(*pcr-word* ===> (=)) (*signed-take-bit* (*LENGTH*('*a*) − *Suc 0*)) (*sint* :: '*a*::*len word* ⇒ *int*)›
using *signed.transfer* [**where** *?'a* = *int*] **by** *simp*

lemma [*transfer-rule*]:
‹(*pcr-word* ===> *pcr-word*) (*take-bit LENGTH*('*a*)) (*ucast* :: '*a*::*len word* ⇒ '*b*::*len word*)›
proof (*rule rel-funI*)
fix *k* :: *int* **and** *w* :: ‹'*a word*›
assume ‹*pcr-word k w*›
then have ‹*w* = *word-of-int k*›
by (*simp add*: *pcr-word-def cr-word-def relcompp-apply*)
moreover have ‹*pcr-word* (*take-bit LENGTH*('*a*) *k*) (*ucast* (*word-of-int k* :: '*a word*))›
by *transfer* (*simp add*: *pcr-word-def cr-word-def relcompp-apply*)
ultimately show ‹*pcr-word* (*take-bit LENGTH*('*a*) *k*) (*ucast w*)›
by *simp*
qed

lemma [*transfer-rule*]:
‹(*pcr-word* ===> *pcr-word*) (*signed-take-bit* (*LENGTH*('*a*) − *Suc 0*)) (*scast* :: '*a*::*len word* ⇒ '*b*::*len word*)›
proof (*rule rel-funI*)
fix *k* :: *int* **and** *w* :: ‹'*a word*›
assume ‹*pcr-word k w*›
then have ‹*w* = *word-of-int k*›
by (*simp add*: *pcr-word-def cr-word-def relcompp-apply*)

moreover have ‹*pcr-word* (*signed-take-bit* (*LENGTH*('*a*) − *Suc 0*) *k*) (*scast* (*word-of-int k* :: '*a word*))›
by *transfer* (*simp add*: *pcr-word-def cr-word-def relcompp-apply*)
ultimately show ‹*pcr-word* (*signed-take-bit* (*LENGTH*('*a*) − *Suc 0*) *k*) (*scast w*)›
by *simp*
qed

end

lemma *of-nat-unat* [*simp*]:
‹*of-nat* (*unat w*) = *unsigned w*›
by *transfer simp*

lemma *of-int-uint* [*simp*]:
‹*of-int* (*uint w*) = *unsigned w*›
by *transfer simp*

lemma *of-int-sint* [*simp*]:
‹*of-int* (*sint a*) = *signed a*›
by *transfer* (*simp-all add*: *take-bit-signed-take-bit*)

lemma *nat-uint-eq* [*simp*]:
‹*nat* (*uint w*) = *unat w*›
by *transfer simp*

lemma *nat-of-natural-unsigned-eq* [*simp*]:
‹*nat-of-natural* (*unsigned w*) = *unat w*›
by *transfer simp*

lemma *int-of-integer-unsigned-eq* [*simp*]:
‹*int-of-integer* (*unsigned w*) = *uint w*›
by *transfer simp*

lemma *int-of-integer-signed-eq* [*simp*]:
‹*int-of-integer* (*signed w*) = *sint w*›
by *transfer simp*

lemma *sgn-uint-eq* [*simp*]:
‹*sgn* (*uint w*) = *of-bool* (*w* ≠ *0*)›
by *transfer* (*simp add*: *less-le*)

Aliasses only for code generation

context
begin

qualified lift-definition *of-int* :: ‹*int* ⇒ '*a*::*len word*›
is ‹*take-bit LENGTH*('*a*)› **.**

qualified lift-definition *of-nat* :: ‹*nat* ⇒ *'a*::*len word*›
is ‹*int* ∘ *take-bit LENGTH*(*'a*)› **.**

qualified lift-definition *the-nat* :: ‹*'a*::*len word* ⇒ *nat*›
is ‹*nat* ∘ *take-bit LENGTH*(*'a*)› **by** *simp*

qualified lift-definition *the-signed-int* :: ‹*'a*::*len word* ⇒ *int*›
is ‹*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)› **by** (*simp add*: *signed-take-bit-decr-length-iff*)

qualified lift-definition *cast* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
is ‹*take-bit LENGTH*(*'a*)› **by** *simp*

qualified lift-definition *signed-cast* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
is ‹*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*)› **by** (*metis signed-take-bit-decr-length-iff*)

end

lemma [*code-abbrev*, *simp*]:
‹*Word.the-int* = *uint*›
by *transfer rule*

lemma [*code*]:
‹*Word.the-int* (*Word.of-int k* :: *'a*::*len word*) = *take-bit LENGTH*(*'a*) *k*›
by *transfer simp*

lemma [*code-abbrev*, *simp*]:
‹*Word.of-int* = *word-of-int*›
by (*rule*; *transfer*) *simp*

lemma [*code*]:
‹*Word.the-int* (*Word.of-nat n* :: *'a*::*len word*) = *take-bit LENGTH*(*'a*) (*int n*)›
by *transfer* (*simp add*: *take-bit-of-nat*)

lemma [*code-abbrev*, *simp*]:
‹*Word.of-nat* = *word-of-nat*›
by (*rule*; *transfer*) (*simp add*: *take-bit-of-nat*)

lemma [*code*]:
‹*Word.the-nat w* = *nat* (*Word.the-int w*)›
by *transfer simp*

lemma [*code-abbrev*, *simp*]:
‹*Word.the-nat* = *unat*›
by (*rule*; *transfer*) *simp*

lemma [*code*]:
‹*Word.the-signed-int w* = (*let k* = *Word.the-int w*
in if bit k (*LENGTH*(*'a*) − *Suc 0*) *then k* + *push-bit LENGTH*(*'a*) (− *1*) *else*
k)›

for *w* :: ‹*'a*::*len word*›
by *transfer* (*simp add*: *bit-simps signed-take-bit-eq-take-bit-add*)

lemma [*code-abbrev*, *simp*]:
‹*Word.the-signed-int* = *sint*›
by (*rule*; *transfer*) *simp*

lemma [*code*]:
‹*Word.the-int* (*Word.cast w* :: *'b*::*len word*) = *take-bit LENGTH*(*'b*) (*Word.the-int w*)›
for *w* :: ‹*'a*::*len word*›
by *transfer simp*

lemma [*code-abbrev*, *simp*]:
‹*Word.cast* = *ucast*›
by (*rule*; *transfer*) *simp*

lemma [*code*]:
‹*Word.the-int* (*Word.signed-cast w* :: *'b*::*len word*) = *take-bit LENGTH*(*'b*) (*Word.the-signed-int w*)›
for *w* :: ‹*'a*::*len word*›
by *transfer simp*

lemma [*code-abbrev*, *simp*]:
‹*Word.signed-cast* = *scast*›
by (*rule*; *transfer*) *simp*

lemma [*code*]:
‹*unsigned w* = *of-nat* (*nat* (*Word.the-int w*))›
by *transfer simp*

lemma [*code*]:
‹*signed w* = *of-int* (*Word.the-signed-int w*)›
by *transfer simp*

108.3 Elementary case distinctions

lemma *word-length-one* [*case-names zero minus-one length-beyond*]:
fixes *w* :: ‹*'a*::*len word*›
obtains (*zero*) ‹*LENGTH*(*'a*) = *Suc 0*› ‹*w* = *0*›
| (*minus-one*) ‹*LENGTH*(*'a*) = *Suc 0*› ‹*w* = − *1*›
| (*length-beyond*) ‹*2* ≤ *LENGTH*(*'a*)›
proof (*cases* ‹*2* ≤ *LENGTH*(*'a*)›)
case *True*
with *length-beyond* **show** *?thesis* **.**
next
case *False*
then have ‹*LENGTH*(*'a*) = *Suc 0*›
by *simp*

then have ‹$w = 0 \vee w = -1$›
by *transfer auto*
with ‹$LENGTH('a) = Suc\ 0$› *zero minus-one* **show** *?thesis*
by *blast*
qed

108.3.1 Basic ordering

instantiation *word* :: (*len*) *linorder*
begin

lift-definition *less-eq-word* :: $'a\ word \Rightarrow 'a\ word \Rightarrow bool$
is $\lambda a\ b.\ take\text{-}bit\ LENGTH('a)\ a \leq take\text{-}bit\ LENGTH('a)\ b$
by *simp*

lift-definition *less-word* :: $'a\ word \Rightarrow 'a\ word \Rightarrow bool$
is $\lambda a\ b.\ take\text{-}bit\ LENGTH('a)\ a < take\text{-}bit\ LENGTH('a)\ b$
by *simp*

instance
by (*standard*; *transfer*) *auto*

end

interpretation *word-order*: *ordering-top* ‹$(\leq)$› ‹$(<)$› ‹-1 :: $'a$::*len word*›
by (*standard*; *transfer*) (*simp add*: *take-bit-eq-mod zmod-minus1*)

interpretation *word-coorder*: *ordering-top* ‹$(\geq)$› ‹$(>)$› ‹0 :: $'a$::*len word*›
by (*standard*; *transfer*) *simp*

lemma *word-of-nat-less-eq-iff*:
‹*word-of-nat* $m \leq$ (*word-of-nat* n :: $'a$::*len word*) $\longleftrightarrow$ *take-bit* $LENGTH('a)\ m \leq$ *take-bit* $LENGTH('a)\ n$›
by *transfer* (*simp add*: *take-bit-of-nat*)

lemma *word-of-int-less-eq-iff*:
‹*word-of-int* $k \leq$ (*word-of-int* l :: $'a$::*len word*) $\longleftrightarrow$ *take-bit* $LENGTH('a)\ k \leq$ *take-bit* $LENGTH('a)\ l$›
by *transfer rule*

lemma *word-of-nat-less-iff*:
‹*word-of-nat* $m <$ (*word-of-nat* n :: $'a$::*len word*) $\longleftrightarrow$ *take-bit* $LENGTH('a)\ m <$ *take-bit* $LENGTH('a)\ n$›
by *transfer* (*simp add*: *take-bit-of-nat*)

lemma *word-of-int-less-iff*:
‹*word-of-int* $k <$ (*word-of-int* l :: $'a$::*len word*) $\longleftrightarrow$ *take-bit* $LENGTH('a)\ k <$ *take-bit* $LENGTH('a)\ l$›
by *transfer rule*

lemma *word-le-def* [*code*]:
  $a \leq b \longleftrightarrow uint\ a \leq uint\ b$
  by *transfer rule*

lemma *word-less-def* [*code*]:
  $a < b \longleftrightarrow uint\ a < uint\ b$
  by *transfer rule*

lemma *word-greater-zero-iff*:
  $\langle a > 0 \longleftrightarrow a \neq 0 \rangle$ **for** $a :: \langle 'a{::}len\ word \rangle$
  by *transfer* (*simp add*: *less-le*)

lemma *of-nat-word-less-eq-iff*:
  $\langle of\text{-}nat\ m \leq (of\text{-}nat\ n :: 'a{::}len\ word) \longleftrightarrow take\text{-}bit\ LENGTH('a)\ m \leq take\text{-}bit\ LENGTH('a)\ n \rangle$
  by *transfer* (*simp add*: *take-bit-of-nat*)

lemma *of-nat-word-less-iff*:
  $\langle of\text{-}nat\ m < (of\text{-}nat\ n :: 'a{::}len\ word) \longleftrightarrow take\text{-}bit\ LENGTH('a)\ m < take\text{-}bit\ LENGTH('a)\ n \rangle$
  by *transfer* (*simp add*: *take-bit-of-nat*)

lemma *of-int-word-less-eq-iff*:
  $\langle of\text{-}int\ k \leq (of\text{-}int\ l :: 'a{::}len\ word) \longleftrightarrow take\text{-}bit\ LENGTH('a)\ k \leq take\text{-}bit\ LENGTH('a)\ l \rangle$
  by *transfer rule*

lemma *of-int-word-less-iff*:
  $\langle of\text{-}int\ k < (of\text{-}int\ l :: 'a{::}len\ word) \longleftrightarrow take\text{-}bit\ LENGTH('a)\ k < take\text{-}bit\ LENGTH('a)\ l \rangle$
  by *transfer rule*

instantiation *word* :: (*len*) *order-bot*
begin

lift-definition *bot-word* :: $\langle 'a\ word \rangle$
  is *0* **.**

instance
  by (*standard*; *transfer*) *simp*

end

lemma *bot-word-eq*:
  $\langle bot = (0 :: 'a{::}len\ word) \rangle$
  by *transfer rule*

instantiation *word* :: (*len*) *order-top*

begin

lift-definition *top-word* :: ‹*'a word*›
  is ‹*− 1*› **.**

instance
  by (*standard*; *transfer*) (*simp add*: *take-bit-int-less-eq-mask*)

end

lemma *top-word-eq*:
  ‹*top* = (*− 1* :: *'a*::*len word*)›
  by *transfer rule*

108.4 Enumeration

lemma *inj-on-word-of-nat*:
  ‹*inj-on* (*word-of-nat* :: *nat* ⇒ *'a*::*len word*) {*0*..<*2* ^ *LENGTH*(*'a*)}›
  by (*rule inj-onI*; *transfer*) (*simp-all add*: *take-bit-int-eq-self*)

lemma *UNIV-word-eq-word-of-nat*:
  ‹(*UNIV* :: *'a*::*len word set*) = *word-of-nat* ‘ {*0*..<*2* ^ *LENGTH*(*'a*)}› (**is** ‹- = *?A*›)
proof
  show ‹*word-of-nat* ‘ {*0*..<*2* ^ *LENGTH*(*'a*)} ⊆ *UNIV*›
    by *simp*
  show ‹*UNIV* ⊆ *?A*›
  proof
    fix *w* :: ‹*'a word*›
    show ‹*w* ∈ (*word-of-nat* ‘ {*0*..<*2* ^ *LENGTH*(*'a*)} :: *'a word set*)›
      by (*rule image-eqI* [*of* - - ‹*unat w*›]; *transfer*) *simp-all*
  qed
qed

instantiation *word* :: (*len*) *enum*
begin

definition *enum-word* :: ‹*'a word list*›
  where ‹*enum-word* = *map word-of-nat* [*0*..<*2* ^ *LENGTH*(*'a*)]›

definition *enum-all-word* :: ‹(*'a word* ⇒ *bool*) ⇒ *bool*›
  where ‹*enum-all-word* = *All*›

definition *enum-ex-word* :: ‹(*'a word* ⇒ *bool*) ⇒ *bool*›
  where ‹*enum-ex-word* = *Ex*›

instance
  by *standard*
   (*simp-all add*: *enum-all-word-def enum-ex-word-def enum-word-def distinct-map*

inj-on-word-of-nat flip: *UNIV-word-eq-word-of-nat*)

end

lemma [*code*]:
‹*Enum.enum-all P* ⟷ *list-all P Enum.enum*›
‹*Enum.enum-ex P* ⟷ *list-ex P Enum.enum*› **for** *P* :: ‹'*a*::*len word* ⇒ *bool*›
by (*simp-all add*: *enum-all-word-def enum-ex-word-def enum-UNIV list-all-iff list-ex-iff*)

108.5 Bit-wise operations

The following specification of word division just lifts the pre-existing division on integers named "F-Division" in [2].

instantiation *word* :: (*len*) *semiring-modulo*
begin

lift-definition *divide-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is ‹λ*a b*. *take-bit LENGTH*('*a*) *a div take-bit LENGTH*('*a*) *b*›
by *simp*

lift-definition *modulo-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is ‹λ*a b*. *take-bit LENGTH*('*a*) *a mod take-bit LENGTH*('*a*) *b*›
by *simp*

instance proof
show *a div b* ∗ *b* + *a mod b* = *a* **for** *a b* :: '*a word*
proof *transfer*
fix *k l* :: *int*
define *r* :: *int* **where** *r* = *2* ^ *LENGTH*('*a*)
then have *r*: *take-bit LENGTH*('*a*) *k* = *k mod r* **for** *k*
by (*simp add*: *take-bit-eq-mod*)
have *k mod r* = ((*k mod r*) *div* (*l mod r*) ∗ (*l mod r*)
+ (*k mod r*) *mod* (*l mod r*)) *mod r*
by (*simp add*: *div-mult-mod-eq*)
also have ... = (((*k mod r*) *div* (*l mod r*) ∗ (*l mod r*)) *mod r*
+ (*k mod r*) *mod* (*l mod r*)) *mod r*
by (*simp add*: *mod-add-left-eq*)
also have ... = (((*k mod r*) *div* (*l mod r*) ∗ *l*) *mod r*
+ (*k mod r*) *mod* (*l mod r*)) *mod r*
by (*simp add*: *mod-mult-right-eq*)
finally have *k mod r* = ((*k mod r*) *div* (*l mod r*) ∗ *l*
+ (*k mod r*) *mod* (*l mod r*)) *mod r*
by (*simp add*: *mod-simps*)
with *r* **show** *take-bit LENGTH*('*a*) (*take-bit LENGTH*('*a*) *k div take-bit LENGTH*('*a*) *l* ∗ *l*
+ *take-bit LENGTH*('*a*) *k mod take-bit LENGTH*('*a*) *l*) = *take-bit LENGTH*('*a*) *k*
by *simp*

```
  qed
qed

end

lemma unat-div-distrib:
  ‹unat (v div w) = unat v div unat w›
proof transfer
  fix k l
  have ‹nat (take-bit LENGTH('a) k) div nat (take-bit LENGTH('a) l) ≤ nat
(take-bit LENGTH('a) k)›
    by (rule div-le-dividend)
  also have ‹nat (take-bit LENGTH('a) k) < 2 ^ LENGTH('a)›
    by (simp add: nat-less-iff)
 finally show ‹(nat ∘ take-bit LENGTH('a)) (take-bit LENGTH('a) k div take-bit
LENGTH('a) l) =
    (nat ∘ take-bit LENGTH('a)) k div (nat ∘ take-bit LENGTH('a)) l›
   by (simp add: nat-take-bit-eq div-int-pos-iff nat-div-distrib take-bit-nat-eq-self-iff)
qed

lemma unat-mod-distrib:
  ‹unat (v mod w) = unat v mod unat w›
proof transfer
  fix k l
  have ‹nat (take-bit LENGTH('a) k) mod nat (take-bit LENGTH('a) l) ≤ nat
(take-bit LENGTH('a) k)›
    by (rule mod-less-eq-dividend)
  also have ‹nat (take-bit LENGTH('a) k) < 2 ^ LENGTH('a)›
    by (simp add: nat-less-iff)
 finally show ‹(nat ∘ take-bit LENGTH('a)) (take-bit LENGTH('a) k mod take-bit
LENGTH('a) l) =
    (nat ∘ take-bit LENGTH('a)) k mod (nat ∘ take-bit LENGTH('a)) l›
   by (simp add: nat-take-bit-eq mod-int-pos-iff less-le nat-mod-distrib take-bit-nat-eq-self-iff)
qed

instance word :: (len) semiring-parity
  by (standard; transfer)
    (auto simp: mod-2-eq-odd take-bit-Suc elim: evenE dest: le-Suc-ex)

lemma word-bit-induct [case-names zero even odd]:
  ‹P a› if word-zero: ‹P 0›
    and word-even: ‹⋀a. P a ⟹ 0 < a ⟹ a < 2 ^ (LENGTH('a) - Suc 0) ⟹
P (2 * a)›
    and word-odd: ‹⋀a. P a ⟹ a < 2 ^ (LENGTH('a) - Suc 0) ⟹ P (1 + 2
* a)›
  for P and a :: ‹'a::len word›
proof -
  define m :: nat where ‹m = LENGTH('a) - Suc 0›
  then have l: ‹LENGTH('a) = Suc m›
```

```
    by simp
  define n :: nat where ‹n = unat a›
  then have ‹n < 2 ^ LENGTH('a)›
    by transfer (simp add: take-bit-eq-mod)
  then have ‹n < 2 * 2 ^ m›
    by (simp add: l)
  then have ‹P (of-nat n)›
  proof (induction n rule: nat-bit-induct)
    case zero
    show ?case
      by simp (rule word-zero)
  next
    case (even n)
    then have ‹n < 2 ^ m›
      by simp
    with even.IH have ‹P (of-nat n)›
      by simp
    moreover from ‹n < 2 ^ m› even.hyps have ‹0 < (of-nat n :: 'a word)›
      by (auto simp: word-greater-zero-iff l word-of-nat-eq-0-iff)
    moreover from ‹n < 2 ^ m› have ‹(of-nat n :: 'a word) < 2 ^ (LENGTH('a)
- Suc 0)›
      using of-nat-word-less-iff [where ?'a = 'a, of n ‹2 ^ m›]
      by (simp add: l take-bit-eq-mod)
    ultimately have ‹P (2 * of-nat n)›
      by (rule word-even)
    then show ?case
      by simp
  next
    case (odd n)
    then have ‹Suc n ≤ 2 ^ m›
      by simp
    with odd.IH have ‹P (of-nat n)›
      by simp
   moreover from ‹Suc n ≤ 2 ^ m› have ‹(of-nat n :: 'a word) < 2 ^ (LENGTH('a)
- Suc 0)›
      using of-nat-word-less-iff [where ?'a = 'a, of n ‹2 ^ m›]
      by (simp add: l take-bit-eq-mod)
    ultimately have ‹P (1 + 2 * of-nat n)›
      by (rule word-odd)
    then show ?case
      by simp
  qed
  moreover have ‹of-nat (nat (uint a)) = a›
    by transfer simp
  ultimately show ?thesis
    by (simp add: n-def)
qed

lemma bit-word-half-eq:
```

```
  ‹(of-bool b + a * 2) div 2 = a›
    if ‹a < 2 ^ (LENGTH('a) - Suc 0)›
    for a :: ‹'a::len word›
proof (cases ‹2 ≤ LENGTH('a::len)›)
  case False
  with that show ?thesis
    by transfer simp
next
  case True
  obtain n where length: ‹LENGTH('a) = Suc n›
    by (cases ‹LENGTH('a)›) simp-all
  show ?thesis proof (cases b)
    case False
    moreover have ‹a * 2 div 2 = a›
    using that proof transfer
      fix k :: int
      from length have ‹k * 2 mod 2 ^ LENGTH('a) = (k mod 2 ^ n) * 2›
        by simp
       moreover assume ‹take-bit LENGTH('a) k < take-bit LENGTH('a) (2 ^
(LENGTH('a) - Suc 0))›
      with ‹LENGTH('a) = Suc n› have ‹take-bit LENGTH('a) k = take-bit n k›
        by (auto simp: take-bit-Suc-from-most)
       ultimately have ‹take-bit LENGTH('a) (k * 2) = take-bit LENGTH('a) k
* 2›
        by (simp add: take-bit-eq-mod)
       with True show ‹take-bit LENGTH('a) (take-bit LENGTH('a) (k * 2) div
take-bit LENGTH('a) 2)
        = take-bit LENGTH('a) k›
        by simp
    qed
    ultimately show ?thesis
      by simp
  next
    case True
    moreover have ‹(1 + a * 2) div 2 = a›
    using that proof transfer
      fix k :: int
      from length have ‹(1 + k * 2) mod 2 ^ LENGTH('a) = 1 + (k mod 2 ^ n)
* 2›
        using pos-zmod-mult-2 [of ‹2 ^ n› k] by (simp add: ac-simps)
       moreover assume ‹take-bit LENGTH('a) k < take-bit LENGTH('a) (2 ^
(LENGTH('a) - Suc 0))›
      with ‹LENGTH('a) = Suc n› have ‹take-bit LENGTH('a) k = take-bit n k›
        by (auto simp: take-bit-Suc-from-most)
        ultimately have ‹take-bit LENGTH('a) (1 + k * 2) = 1 + take-bit
LENGTH('a) k * 2›
        by (simp add: take-bit-eq-mod)
      with True show ‹take-bit LENGTH('a) (take-bit LENGTH('a) (1 + k * 2)
div take-bit LENGTH('a) 2)
```

```
      = take-bit LENGTH('a) k›
      by (auto simp: take-bit-Suc)
    qed
    ultimately show ?thesis
      by simp
  qed
qed

lemma even-mult-exp-div-word-iff:
  ‹even (a * 2 ^ m div 2 ^ n) ⟷ ¬ (
    m ≤ n ∧
    n < LENGTH('a) ∧ odd (a div 2 ^ (n − m)))› for a :: ‹'a::len word›
  by transfer
   (auto simp flip: drop-bit-eq-div simp add: even-drop-bit-iff-not-bit bit-take-bit-iff,
     simp-all flip: push-bit-eq-mult add: bit-push-bit-iff-int)

instantiation word :: (len) semiring-bits
begin

lift-definition bit-word :: ‹'a word ⇒ nat ⇒ bool›
  is ‹λk n. n < LENGTH('a) ∧ bit k n›
proof
  fix k l :: int and n :: nat
  assume *: ‹take-bit LENGTH('a) k = take-bit LENGTH('a) l›
  show ‹n < LENGTH('a) ∧ bit k n ⟷ n < LENGTH('a) ∧ bit l n›
  proof (cases ‹n < LENGTH('a)›)
    case True
    from * have ‹bit (take-bit LENGTH('a) k) n ⟷ bit (take-bit LENGTH('a)
l) n›
      by simp
    then show ?thesis
      by (simp add: bit-take-bit-iff)
  next
    case False
    then show ?thesis
      by simp
  qed
qed

instance proof
  show ‹P a› if stable: ‹⋀a. a div 2 = a ⟹ P a›
    and rec: ‹⋀a b. P a ⟹ (of-bool b + 2 * a) div 2 = a ⟹ P (of-bool b + 2 *
a)›
  for P and a :: ‹'a word›
  proof (induction a rule: word-bit-induct)
    case zero
    have ‹0 div 2 = (0::'a word)›
      by transfer simp
    with stable [of 0] show ?case
```

```
    by simp
  next
    case (even a)
    with rec [of a False] show ?case
      using bit-word-half-eq [of a False] by (simp add: ac-simps)
  next
    case (odd a)
    with rec [of a True] show ?case
      using bit-word-half-eq [of a True] by (simp add: ac-simps)
  qed
  show ‹bit a n ⟷ odd (a div 2 ^ n)› for a :: ‹'a word› and n
    by transfer (simp flip: drop-bit-eq-div add: drop-bit-take-bit bit-iff-odd-drop-bit)
  show ‹a div 0 = 0›
    for a :: ‹'a word›
    by transfer simp
  show ‹a div 1 = a›
    for a :: ‹'a word›
    by transfer simp
  show ‹0 div a = 0›
    for a :: ‹'a word›
    by transfer simp
  show ‹a mod b div b = 0›
    for a b :: ‹'a word›
   by (simp add: word-eq-iff-unsigned [where ?'a = nat] unat-div-distrib unat-mod-distrib)
  show ‹a div 2 div 2 ^ n = a div 2 ^ Suc n›
    for a :: ‹'a word› and m n :: nat
    apply transfer
    using drop-bit-eq-div [symmetric, where ?'a = int,of - 1]
    apply (auto simp:  not-less take-bit-drop-bit ac-simps simp flip: drop-bit-eq-div
simp del: power.simps)
    apply (simp add: drop-bit-take-bit)
    done
  show ‹even (2 * a div 2 ^ Suc n) ⟷ even (a div 2 ^ n)› if ‹2 ^ Suc n ≠ (0::'a
word)›
    for a :: ‹'a word› and n :: nat
    using that by transfer
    (simp add: even-drop-bit-iff-not-bit bit-simps flip: drop-bit-eq-div del: power.simps)
qed

end

lemma bit-word-eqI:
  ‹a = b› if ‹⋀n. n < LENGTH('a) ⟹ bit a n ⟷ bit b n›
  for a b :: ‹'a::len word›
  using that by transfer (auto simp: nat-less-le bit-eq-iff bit-take-bit-iff)

lemma bit-imp-le-length: ‹n < LENGTH('a)› if ‹bit w n› for w :: ‹'a::len word›
  by (meson bit-word.rep-eq that)
```

lemma *not-bit-length* [*simp*]:
‹¬ *bit w LENGTH*('*a*)› **for** *w* :: ‹'*a*::*len word*›
using *bit-imp-le-length* **by** *blast*

lemma *finite-bit-word* [*simp*]:
‹*finite* {*n*. *bit w n*}›
for *w* :: ‹'*a*::*len word*›
by (*metis bit-imp-le-length bounded-nat-set-is-finite mem-Collect-eq*)

lemma *bit-numeral-word-iff* [*simp*]:
‹*bit* (*numeral w* :: '*a*::*len word*) *n*
⟷ *n* < *LENGTH*('*a*) ∧ *bit* (*numeral w* :: *int*) *n*›
by *transfer simp*

lemma *bit-neg-numeral-word-iff* [*simp*]:
‹*bit* (− *numeral w* :: '*a*::*len word*) *n*
⟷ *n* < *LENGTH*('*a*) ∧ *bit* (− *numeral w* :: *int*) *n*›
by *transfer simp*

instantiation *word* :: (*len*) *ring-bit-operations*
begin

lift-definition *not-word* :: ‹'*a word* ⇒ '*a word*›
is *not*
by (*simp add*: *take-bit-not-iff*)

lift-definition *and-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is ‹*and*›
by *simp*

lift-definition *or-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is *or*
by *simp*

lift-definition *xor-word* :: ‹'*a word* ⇒ '*a word* ⇒ '*a word*›
is *xor*
by *simp*

lift-definition *mask-word* :: ‹*nat* ⇒ '*a word*›
is *mask*
.

lift-definition *set-bit-word* :: ‹*nat* ⇒ '*a word* ⇒ '*a word*›
is *set-bit*
by (*simp add*: *set-bit-def*)

lift-definition *unset-bit-word* :: ‹*nat* ⇒ '*a word* ⇒ '*a word*›
is *unset-bit*
by (*simp add*: *unset-bit-def*)

lift-definition *flip-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
is *flip-bit*
by (*simp add*: *flip-bit-def*)

lift-definition *push-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
is *push-bit*
proof −
show ‹*take-bit LENGTH('a)* (*push-bit n k*) = *take-bit LENGTH('a)* (*push-bit n l*)›
if ‹*take-bit LENGTH('a) k* = *take-bit LENGTH('a) l*› **for** *k l* :: *int* **and** *n* :: *nat*
by (*metis le-add2 push-bit-take-bit take-bit-tightened that*)
qed

lift-definition *drop-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
is ‹λ*n*. *drop-bit n* ∘ *take-bit LENGTH('a)*›
by (*simp add*: *take-bit-eq-mod*)

lift-definition *take-bit-word* :: ‹*nat* ⇒ *'a word* ⇒ *'a word*›
is ‹λ*n*. *take-bit* (*min LENGTH('a) n*)›
by (*simp add*: *ac-simps*) (*simp only*: *flip*: *take-bit-take-bit*)

context
includes *bit-operations-syntax*
begin

instance proof
fix *v w* :: ‹*'a word*› **and** *n m* :: *nat*
show ‹*NOT v* = − *v* − *1*›
by *transfer* (*simp add*: *not-eq-complement*)
show ‹*v AND w* = *of-bool* (*odd v* ∧ *odd w*) + *2* ∗ (*v div 2 AND w div 2*)›
apply *transfer*
by (*rule bit-eqI*) (*auto simp*: *even-bit-succ-iff bit-simps bit-0 simp flip*: *bit-Suc*)
show ‹*v OR w* = *of-bool* (*odd v* ∨ *odd w*) + *2* ∗ (*v div 2 OR w div 2*)›
apply *transfer*
by (*rule bit-eqI*) (*auto simp*: *even-bit-succ-iff bit-simps bit-0 simp flip*: *bit-Suc*)
show ‹*v XOR w* = *of-bool* (*odd v* ≠ *odd w*) + *2* ∗ (*v div 2 XOR w div 2*)›
apply *transfer*
apply (*rule bit-eqI*)
subgoal for *k l n*
apply (*cases n*)
apply (*auto simp*: *even-bit-succ-iff bit-simps bit-0 even-xor-iff simp flip*: *bit-Suc*)
done
done
show ‹*mask n* = *2* ^ *n* − (*1* :: *'a word*)›
by *transfer* (*simp flip*: *mask-eq-exp-minus-1*)
show ‹*set-bit n v* = *v OR push-bit n 1*›

by *transfer* (*simp add*: *set-bit-eq-or*)
show ‹*unset-bit n v* = (*v OR push-bit n 1*) *XOR push-bit n 1*›
by *transfer* (*simp add*: *unset-bit-eq-or-xor*)
show ‹*flip-bit n v* = *v XOR push-bit n 1*›
by *transfer* (*simp add*: *flip-bit-eq-xor*)
show ‹*push-bit n v* = *v* ∗ 2 ^ *n*›
by *transfer* (*simp add*: *push-bit-eq-mult*)
show ‹*drop-bit n v* = *v div* 2 ^ *n*›
by *transfer* (*simp add*: *drop-bit-take-bit flip*: *drop-bit-eq-div*)
show ‹*take-bit n v* = *v mod* 2 ^ *n*›
by *transfer* (*simp flip*: *take-bit-eq-mod*)
qed

end

end

lemma [*code*]:
‹*push-bit n w* = *w* ∗ 2 ^ *n*› **for** *w* :: ‹'*a*::*len word*›
by (*fact push-bit-eq-mult*)

lemma [*code*]:
‹*Word.the-int* (*drop-bit n w*) = *drop-bit n* (*Word.the-int w*)›
by *transfer* (*simp add*: *drop-bit-take-bit min-def le-less less-diff-conv*)

lemma [*code*]:
‹*Word.the-int* (*take-bit n w*) = (*if n* < *LENGTH*('*a*::*len*) *then take-bit n* (*Word.the-int w*) *else Word.the-int w*)›
for *w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *not-le not-less ac-simps min-absorb2*)

lemma [*code-abbrev*]:
‹*push-bit n* 1 = (2 :: '*a*::*len word*) ^ *n*›
by (*fact push-bit-of-1*)

context
includes *bit-operations-syntax*
begin

lemma [*code*]:
‹*NOT w* = *Word.of-int* (*NOT* (*Word.the-int w*))›
for *w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *take-bit-not-take-bit*)

lemma [*code*]:
‹*Word.the-int* (*v AND w*) = *Word.the-int v AND Word.the-int w*›
by *transfer simp*

lemma [*code*]:

‹*Word.the-int* (*v OR w*) = *Word.the-int v OR Word.the-int w*›
by *transfer simp*

lemma [*code*]:
‹*Word.the-int* (*v XOR w*) = *Word.the-int v XOR Word.the-int w*›
by *transfer simp*

lemma [*code*]:
‹*Word.the-int* (*mask n* :: *'a*::*len word*) = *mask* (*min LENGTH*(*'a*) *n*)›
by *transfer simp*

lemma [*code*]:
‹*set-bit n w* = *w OR push-bit n 1*› **for** *w* :: ‹*'a*::*len word*›
by (*fact set-bit-eq-or*)

lemma [*code*]:
‹*unset-bit n w* = *w AND NOT* (*push-bit n 1*)› **for** *w* :: ‹*'a*::*len word*›
by (*fact unset-bit-eq-and-not*)

lemma [*code*]:
‹*flip-bit n w* = *w XOR push-bit n 1*› **for** *w* :: ‹*'a*::*len word*›
by (*fact flip-bit-eq-xor*)

context
includes *lifting-syntax*
begin

lemma *set-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *set-bit set-bit*›
by (*unfold set-bit-def*) *transfer-prover*

lemma *unset-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *unset-bit unset-bit*›
by (*unfold unset-bit-def*) *transfer-prover*

lemma *flip-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*) *flip-bit flip-bit*›
by (*unfold flip-bit-def*) *transfer-prover*

lemma *signed-take-bit-word-transfer* [*transfer-rule*]:
‹((=) ===> *pcr-word* ===> *pcr-word*)
(*λn k. signed-take-bit n* (*take-bit LENGTH*(*'a*::*len*) *k*))
(*signed-take-bit* :: *nat* ⇒ *'a word* ⇒ *'a word*)›
proof −
let *?K* = ‹*λn* (*k* :: *int*). *take-bit* (*min LENGTH*(*'a*) *n*) *k OR of-bool* (*n* < *LENGTH*(*'a*) ∧ *bit k n*) ∗ *NOT* (*mask n*)›
let *?W* = ‹*λn* (*w* :: *'a word*). *take-bit n w OR of-bool* (*bit w n*) ∗ *NOT* (*mask n*)›
have ‹((=) ===> *pcr-word* ===> *pcr-word*) *?K ?W*›

by *transfer-prover*
also have ‹*?K* = (λ*n k*. *signed-take-bit n* (*take-bit LENGTH*('*a*::*len*) *k*))›
by (*simp add*: *fun-eq-iff signed-take-bit-def bit-take-bit-iff ac-simps*)
also have ‹*?W* = *signed-take-bit*›
by (*simp add*: *fun-eq-iff signed-take-bit-def*)
finally show *?thesis* **.**
qed

end

end

108.6 Conversions including casts

108.6.1 Generic unsigned conversion

context *semiring-bits*
begin

lemma *bit-unsigned-iff* [*bit-simps*]:
‹*bit* (*unsigned w*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ *bit w n*›
for *w* :: ‹'*b*::*len word*›
by (*transfer fixing*: *bit*) (*simp add*: *bit-of-nat-iff bit-nat-iff bit-take-bit-iff*)

end

lemma *possible-bit-word*[*simp*]:
‹*possible-bit TYPE*(('*a* :: *len*) *word*) *m* ⟷ *m* < *LENGTH*('*a*)›
by (*simp add*: *possible-bit-def linorder-not-le*)

context *semiring-bit-operations*
begin

lemma *unsigned-minus-1-eq-mask*:
‹*unsigned* (− *1* :: '*b*::*len word*) = *mask LENGTH*('*b*)›
by (*transfer fixing*: *mask*) (*simp add*: *nat-mask-eq of-nat-mask-eq*)

lemma *unsigned-push-bit-eq*:
‹*unsigned* (*push-bit n w*) = *take-bit LENGTH*('*b*) (*push-bit n* (*unsigned w*))›
for *w* :: ‹'*b*::*len word*›
proof (*rule bit-eqI*)
fix *m*
assume ‹*possible-bit TYPE*('*a*) *m*›
show ‹*bit* (*unsigned* (*push-bit n w*)) *m* = *bit* (*take-bit LENGTH*('*b*) (*push-bit n* (*unsigned w*))) *m*›
proof (*cases* ‹*n* ≤ *m*›)
case *True*
with ‹*possible-bit TYPE*('*a*) *m*› **have** ‹*possible-bit TYPE*('*a*) (*m* − *n*)›
by (*simp add*: *possible-bit-less-imp*)
with *True* **show** *?thesis*

by (*simp add*: *bit-unsigned-iff bit-push-bit-iff Bit-Operations.bit-push-bit-iff bit-take-bit-iff not-le ac-simps*)
next
case *False*
then show *?thesis*
by (*simp add*: *not-le bit-unsigned-iff bit-push-bit-iff Bit-Operations.bit-push-bit-iff bit-take-bit-iff*)
qed
qed

lemma *unsigned-take-bit-eq*:
‹*unsigned* (*take-bit n w*) = *take-bit n* (*unsigned w*)›
for *w* :: ‹*'b*::*len word*›
by (*rule bit-eqI*) (*simp add*: *bit-unsigned-iff bit-take-bit-iff Bit-Operations.bit-take-bit-iff*)

end

context *linordered-euclidean-semiring-bit-operations*
begin

lemma *unsigned-drop-bit-eq*:
‹*unsigned* (*drop-bit n w*) = *drop-bit n* (*take-bit LENGTH*(*'b*) (*unsigned w*))›
for *w* :: ‹*'b*::*len word*›
by (*rule bit-eqI*) (*auto simp*: *bit-unsigned-iff bit-take-bit-iff bit-drop-bit-eq Bit-Operations.bit-drop-bit-eq possible-bit-def dest*: *bit-imp-le-length*)

end

lemma *ucast-drop-bit-eq*:
‹*ucast* (*drop-bit n w*) = *drop-bit n* (*ucast w* :: *'b*::*len word*)›
if ‹*LENGTH*(*'a*) ≤ *LENGTH*(*'b*)› **for** *w* :: ‹*'a*::*len word*›
by (*rule bit-word-eqI*) (*use that* **in** ‹*auto simp*: *bit-unsigned-iff bit-drop-bit-eq dest*: *bit-imp-le-length*›)

context *semiring-bit-operations*
begin

context
includes *bit-operations-syntax*
begin

lemma *unsigned-and-eq*:
‹*unsigned* (*v AND w*) = *unsigned v AND unsigned w*›
for *v w* :: ‹*'b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps*)

lemma *unsigned-or-eq*:
‹*unsigned* (*v OR w*) = *unsigned v OR unsigned w*›
for *v w* :: ‹*'b*::*len word*›

by (*simp add*: *bit-eq-iff bit-simps*)

lemma *unsigned-xor-eq*:
‹*unsigned* (*v XOR w*) = *unsigned v XOR unsigned w*›
for *v w* :: ‹'*b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps*)

end

end

context *ring-bit-operations*
begin

context
includes *bit-operations-syntax*
begin

lemma *unsigned-not-eq*:
‹*unsigned* (*NOT w*) = *take-bit LENGTH*('*b*) (*NOT* (*unsigned w*))›
for *w* :: ‹'*b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps*)

end

end

context *linordered-euclidean-semiring*
begin

lemma *unsigned-greater-eq* [*simp*]:
‹*0* ≤ *unsigned w*› **for** *w* :: ‹'*b*::*len word*›
by (*transfer fixing*: *less-eq*) *simp*

lemma *unsigned-less* [*simp*]:
‹*unsigned w* < *2* ^ *LENGTH*('*b*)› **for** *w* :: ‹'*b*::*len word*›
by (*transfer fixing*: *less*) *simp*

end

context *linordered-semidom*
begin

lemma *word-less-eq-iff-unsigned*:
a ≤ *b* ⟷ *unsigned a* ≤ *unsigned b*
by (*transfer fixing*: *less-eq*) (*simp add*: *nat-le-eq-zle*)

lemma *word-less-iff-unsigned*:
a < *b* ⟷ *unsigned a* < *unsigned b*

by (*transfer fixing*: *less*) (*auto dest*: *preorder-class.le-less-trans* [*OF take-bit-nonnegative*])

end

108.6.2 Generic signed conversion

context *ring-bit-operations*
begin

lemma *bit-signed-iff* [*bit-simps*]:
‹*bit* (*signed w*) *n* ⟷ *possible-bit TYPE*('*a*) *n* ∧ *bit w* (*min* (*LENGTH*('*b*) − *Suc 0*) *n*)›
for *w* :: ‹'*b*::*len word*›
by (*transfer fixing*: *bit*)
(*auto simp*: *bit-of-int-iff Bit-Operations.bit-signed-take-bit-iff min-def*)

lemma *signed-push-bit-eq*:
‹*signed* (*push-bit n w*) = *signed-take-bit* (*LENGTH*('*b*) − *Suc 0*) (*push-bit n* (*signed w* :: '*a*))›
for *w* :: ‹'*b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps possible-bit-less-imp min-less-iff-disj*) *auto*

lemma *signed-take-bit-eq*:
‹*signed* (*take-bit n w*) = (*if n* < *LENGTH*('*b*) *then take-bit n* (*signed w*) *else signed w*)›
for *w* :: ‹'*b*::*len word*›
by (*transfer fixing*: *take-bit*; *cases* ‹*LENGTH*('*b*)›)
(*auto simp*: *Bit-Operations.signed-take-bit-take-bit Bit-Operations.take-bit-signed-take-bit take-bit-of-int min-def less-Suc-eq*)

context
includes *bit-operations-syntax*
begin

lemma *signed-not-eq*:
‹*signed* (*NOT w*) = *signed-take-bit LENGTH*('*b*) (*NOT* (*signed w*))›
for *w* :: ‹'*b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps possible-bit-less-imp min-less-iff-disj*)
(*auto simp*: *min-def*)

lemma *signed-and-eq*:
‹*signed* (*v AND w*) = *signed v AND signed w*›
for *v w* :: ‹'*b*::*len word*›
by (*rule bit-eqI*) (*simp add*: *bit-signed-iff bit-and-iff Bit-Operations.bit-and-iff*)

lemma *signed-or-eq*:
‹*signed* (*v OR w*) = *signed v OR signed w*›
for *v w* :: ‹'*b*::*len word*›
by (*rule bit-eqI*) (*simp add*: *bit-signed-iff bit-or-iff Bit-Operations.bit-or-iff*)

lemma *signed-xor-eq*:
‹*signed* (*v XOR w*) = *signed v XOR signed w*›
for *v w* :: ‹'*b*::*len word*›
by (*rule bit-eqI*) (*simp add*: *bit-signed-iff bit-xor-iff Bit-Operations.bit-xor-iff*)

end

end

108.6.3 More

lemma *sint-greater-eq*:
‹− (2 ^ (*LENGTH*('*a*) − *Suc 0*)) ≤ *sint w*› **for** *w* :: ‹'*a*::*len word*›
proof (*cases* ‹*bit w* (*LENGTH*('*a*) − *Suc 0*)›)
case *True*
then show *?thesis*
by *transfer* (*simp add*: *signed-take-bit-eq-if-negative minus-exp-eq-not-mask or-greater-eq ac-simps*)
next
have ∗: ‹− (2 ^ (*LENGTH*('*a*) − *Suc 0*)) ≤ (*0*::*int*)›
by *simp*
case *False*
then show *?thesis*
by *transfer* (*auto simp*: *signed-take-bit-eq intro*: *order-trans* ∗)
qed

lemma *sint-less*:
‹*sint w* < 2 ^ (*LENGTH*('*a*) − *Suc 0*)› **for** *w* :: ‹'*a*::*len word*›
by (*cases* ‹*bit w* (*LENGTH*('*a*) − *Suc 0*)›; *transfer*)
(*simp-all add*: *signed-take-bit-eq signed-take-bit-def not-eq-complement mask-eq-exp-minus-1 OR-upper*)

lemma *uint-div-distrib*:
‹*uint* (*v div w*) = *uint v div uint w*›
by (*metis int-ops*(*8*) *of-nat-unat unat-div-distrib*)

lemma *unat-drop-bit-eq*:
‹*unat* (*drop-bit n w*) = *drop-bit n* (*unat w*)›
by (*rule bit-eqI*) (*simp add*: *bit-unsigned-iff bit-drop-bit-eq*)

lemma *uint-mod-distrib*:
‹*uint* (*v mod w*) = *uint v mod uint w*›
by (*metis int-ops*(*9*) *of-nat-unat unat-mod-distrib*)

context *semiring-bit-operations*
begin

lemma *unsigned-ucast-eq*:

‹*unsigned* (*ucast w* :: *'c*::*len word*) = *take-bit LENGTH*('*c*) (*unsigned w*)›
for *w* :: ‹*'b*::*len word*›
by (*rule bit-eqI*) (*simp add*: *bit-unsigned-iff Word.bit-unsigned-iff bit-take-bit-iff not-le*)

end

context *ring-bit-operations*
begin

lemma *signed-ucast-eq*:
‹*signed* (*ucast w* :: *'c*::*len word*) = *signed-take-bit* (*LENGTH*('*c*) − *Suc 0*) (*unsigned w*)›
for *w* :: ‹*'b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps min-less-iff-disj*)

lemma *signed-scast-eq*:
‹*signed* (*scast w* :: *'c*::*len word*) = *signed-take-bit* (*LENGTH*('*c*) − *Suc 0*) (*signed w*)›
for *w* :: ‹*'b*::*len word*›
by (*simp add*: *bit-eq-iff bit-simps min-less-iff-disj*)

end

lemma *uint-nonnegative*: *0* ≤ *uint w*
by (*fact unsigned-greater-eq*)

lemma *uint-bounded*: *uint w* < *2* ^ *LENGTH*('*a*)
for *w* :: *'a*::*len word*
by (*fact unsigned-less*)

lemma *uint-idem*: *uint w mod 2* ^ *LENGTH*('*a*) = *uint w*
for *w* :: *'a*::*len word*
by *transfer* (*simp add*: *take-bit-eq-mod*)

lemma *word-uint-eqI*: *uint a* = *uint b* ⟹ *a* = *b*
by (*fact unsigned-word-eqI*)

lemma *word-uint-eq-iff*: *a* = *b* ⟷ *uint a* = *uint b*
by (*fact word-eq-iff-unsigned*)

lemma *uint-word-of-int-eq*:
‹*uint* (*word-of-int k* :: *'a*::*len word*) = *take-bit LENGTH*('*a*) *k*›
by *transfer rule*

lemma *uint-word-of-int*: *uint* (*word-of-int k* :: *'a*::*len word*) = *k mod 2* ^ *LENGTH*('*a*)
by (*simp add*: *uint-word-of-int-eq take-bit-eq-mod*)

lemma *word-of-int-uint*: *word-of-int* (*uint w*) = *w*

by *transfer simp*

lemma *word-div-def* [*code*]:
a div b = word-of-int (uint a div uint b)
by *transfer rule*

lemma *word-mod-def* [*code*]:
a mod b = word-of-int (uint a mod uint b)
by *transfer rule*

lemma *split-word-all*: $(\bigwedge x::'a::len\ word.\ PROP\ P\ x) \equiv (\bigwedge x.\ PROP\ P\ (word\text{-}of\text{-}int\ x))$
proof
fix *x* :: *'a word*
assume $\bigwedge x.\ PROP\ P\ (word\text{-}of\text{-}int\ x)$
then have *PROP P (word-of-int (uint x))* **.**
then show *PROP P x*
by (*simp only*: *word-of-int-uint*)
qed

lemma *sint-uint*:
‹*sint w = signed-take-bit (LENGTH('a) − Suc 0) (uint w)*›
for *w* :: ‹*'a::len word*›
by (*cases* ‹*LENGTH('a)*›; *transfer*) (*simp-all add*: *signed-take-bit-take-bit*)

lemma *unat-eq-nat-uint*:
‹*unat w = nat (uint w)*›
by *simp*

lemma *ucast-eq*:
‹*ucast w = word-of-int (uint w)*›
by *transfer simp*

lemma *scast-eq*:
‹*scast w = word-of-int (sint w)*›
by *transfer simp*

lemma *uint-0-eq*:
‹*uint 0 = 0*›
by (*fact unsigned-0*)

lemma *uint-1-eq*:
‹*uint 1 = 1*›
by (*fact unsigned-1*)

lemma *word-m1-wi*: *− 1 = word-of-int (− 1)*
by *simp*

lemma *uint-0-iff*: *uint x = 0 ⟷ x = 0*

by (*auto simp*: *unsigned-word-eqI*)

lemma *unat-0-iff*: *unat x = 0 ⟷ x = 0*
by (*auto simp*: *unsigned-word-eqI*)

lemma *unat-0*: *unat 0 = 0*
by (*fact unsigned-0*)

lemma *unat-gt-0*: *0 < unat x ⟷ x ≠ 0*
by (*auto simp*: *unat-0-iff* [*symmetric*])

lemma *ucast-0*: *ucast 0 = 0*
by (*fact unsigned-0*)

lemma *sint-0*: *sint 0 = 0*
by (*fact signed-0*)

lemma *scast-0*: *scast 0 = 0*
by (*fact signed-0*)

lemma *sint-n1*: *sint (− 1) = − 1*
by (*fact signed-minus-1*)

lemma *scast-n1*: *scast (− 1) = − 1*
by (*fact signed-minus-1*)

lemma *uint-1*: *uint (1::'a::len word) = 1*
by (*fact uint-1-eq*)

lemma *unat-1*: *unat (1::'a::len word) = 1*
by (*fact unsigned-1*)

lemma *ucast-1*: *ucast (1::'a::len word) = 1*
by (*fact unsigned-1*)

instantiation *word* :: (*len*) *size*
begin

lift-definition *size-word* :: ‹*'a word ⇒ nat*›
is ‹*λ-. LENGTH('a)*› **..**

instance ..

end

lemma *word-size* [*code*]:
‹*size w = LENGTH('a)*› **for** *w* :: ‹*'a::len word*›
by (*fact size-word.rep-eq*)

lemma *word-size-gt-0* [*iff*]: *0 < size w*
for *w* :: *'a::len word*
by (*simp add*: *word-size*)

lemmas *lens-gt-0 = word-size-gt-0 len-gt-0*

lemma *lens-not-0* [*iff*]:
‹*size w ≠ 0*› **for** *w* :: ‹*'a::len word*›
by *auto*

lift-definition *source-size* :: ‹(*'a::len word ⇒ 'b*) ⇒ *nat*›
is ‹λ-. *LENGTH('a)*› **.**

lift-definition *target-size* :: ‹(*'a ⇒ 'b::len word*) ⇒ *nat*›
is ‹λ-. *LENGTH('b)*› **..**

lift-definition *is-up* :: ‹(*'a::len word ⇒ 'b::len word*) ⇒ *bool*›
is ‹λ-. *LENGTH('a) ≤ LENGTH('b)*› **..**

lift-definition *is-down* :: ‹(*'a::len word ⇒ 'b::len word*) ⇒ *bool*›
is ‹λ-. *LENGTH('a) ≥ LENGTH('b)*› **..**

lemma *is-up-eq*:
‹*is-up f ⟷ source-size f ≤ target-size f*›
for *f* :: ‹*'a::len word ⇒ 'b::len word*›
by (*simp add*: *source-size.rep-eq target-size.rep-eq is-up.rep-eq*)

lemma *is-down-eq*:
‹*is-down f ⟷ target-size f ≤ source-size f*›
for *f* :: ‹*'a::len word ⇒ 'b::len word*›
by (*simp add*: *source-size.rep-eq target-size.rep-eq is-down.rep-eq*)

lift-definition *word-int-case* :: ‹(*int ⇒ 'b*) ⇒ *'a::len word ⇒ 'b*›
is ‹λ*f*. *f ∘ take-bit LENGTH('a)*› **by** *simp*

lemma *word-int-case-eq-uint* [*code*]:
‹*word-int-case f w = f (uint w)*›
by *transfer simp*

translations
case x of XCONST of-int y ⇒ b ⇌ CONST word-int-case (λ*y. b*) *x*
case x of (*XCONST of-int* :: *'a*) *y ⇒ b ↽ CONST word-int-case* (λ*y. b*) *x*

108.7 Arithmetic operations

lemma *div-word-self*:
‹*w div w = 1*› **if** ‹*w ≠ 0*› **for** *w* :: ‹*'a::len word*›
using *that* **by** *transfer simp*

lemma *mod-word-self* [*simp*]:
‹*w mod w = 0*› **for** *w* :: ‹*'a*::*len word*›
by (*simp add*: *word-mod-def*)

lemma *div-word-less*:
‹*w div v = 0*› **if** ‹*w < v*› **for** *w v* :: ‹*'a*::*len word*›
using *that* **by** *transfer simp*

lemma *mod-word-less*:
‹*w mod v = w*› **if** ‹*w < v*› **for** *w v* :: ‹*'a*::*len word*›
using *div-mult-mod-eq* [*of w v*] **using** *that* **by** (*simp add*: *div-word-less*)

lemma *div-word-one* [*simp*]:
‹*1 div w = of-bool* (*w = 1*)› **for** *w* :: ‹*'a*::*len word*›
proof *transfer*
fix *k* :: *int*
show ‹*take-bit LENGTH*('*a*) (*take-bit LENGTH*('*a*) *1 div take-bit LENGTH*('*a*) *k*) =
take-bit LENGTH('*a*) (*of-bool* (*take-bit LENGTH*('*a*) *k* = *take-bit LENGTH*('*a*) *1*))›
using *take-bit-nonnegative* [*of* ‹*LENGTH*('*a*)› *k*]
by (*smt* (*verit*, *best*) *div-by-1 of-bool-eq take-bit-of-0 take-bit-of-1 zdiv-eq-0-iff*)
qed

lemma *mod-word-one* [*simp*]:
‹*1 mod w = 1 − w ∗ of-bool* (*w = 1*)› **for** *w* :: ‹*'a*::*len word*›
using *div-mult-mod-eq* [*of 1 w*] **by** *auto*

lemma *div-word-by-minus-1-eq* [*simp*]:
‹*w div − 1 = of-bool* (*w = − 1*)› **for** *w* :: ‹*'a*::*len word*›
by (*auto intro*: *div-word-less simp add*: *div-word-self word-order.not-eq-extremum*)

lemma *mod-word-by-minus-1-eq* [*simp*]:
‹*w mod − 1 = w ∗ of-bool* (*w < − 1*)› **for** *w* :: ‹*'a*::*len word*›
using *mod-word-less word-order.not-eq-extremum* **by** *fastforce*

Legacy theorems:

lemma *word-add-def* [*code*]:
a + b = word-of-int (*uint a + uint b*)
by *transfer* (*simp add*: *take-bit-add*)

lemma *word-sub-wi* [*code*]:
a − b = word-of-int (*uint a − uint b*)
by *transfer* (*simp add*: *take-bit-diff*)

lemma *word-mult-def* [*code*]:
a ∗ b = word-of-int (*uint a ∗ uint b*)
by *transfer* (*simp add*: *take-bit-eq-mod mod-simps*)

lemma *word-minus-def* [*code*]:
  $- a = word\text{-}of\text{-}int\ (-\ uint\ a)$
  by *transfer* (*simp add*: *take-bit-minus*)

lemma *word-0-wi*:
  $0 = word\text{-}of\text{-}int\ 0$
  by *transfer simp*

lemma *word-1-wi*:
  $1 = word\text{-}of\text{-}int\ 1$
  by *transfer simp*

lift-definition *word-succ* :: *'a*::*len word* ⇒ *'a word* **is** $\lambda x.\ x + 1$
  by (*auto simp*: *take-bit-eq-mod intro*: *mod-add-cong*)

lift-definition *word-pred* :: *'a*::*len word* ⇒ *'a word* **is** $\lambda x.\ x - 1$
  by (*auto simp*: *take-bit-eq-mod intro*: *mod-diff-cong*)

lemma *word-succ-alt* [*code*]:
  $word\text{-}succ\ a = word\text{-}of\text{-}int\ (uint\ a + 1)$
  by *transfer* (*simp add*: *take-bit-eq-mod mod-simps*)

lemma *word-pred-alt* [*code*]:
  $word\text{-}pred\ a = word\text{-}of\text{-}int\ (uint\ a - 1)$
  by *transfer* (*simp add*: *take-bit-eq-mod mod-simps*)

lemmas *word-arith-wis* =
  word-add-def word-sub-wi word-mult-def
  word-minus-def word-succ-alt word-pred-alt
  word-0-wi word-1-wi

lemma *wi-homs*:
  shows *wi-hom-add*: $word\text{-}of\text{-}int\ a + word\text{-}of\text{-}int\ b = word\text{-}of\text{-}int\ (a + b)$
    and *wi-hom-sub*: $word\text{-}of\text{-}int\ a - word\text{-}of\text{-}int\ b = word\text{-}of\text{-}int\ (a - b)$
    and *wi-hom-mult*: $word\text{-}of\text{-}int\ a * word\text{-}of\text{-}int\ b = word\text{-}of\text{-}int\ (a * b)$
    and *wi-hom-neg*: $-\ word\text{-}of\text{-}int\ a = word\text{-}of\text{-}int\ (-\ a)$
    and *wi-hom-succ*: $word\text{-}succ\ (word\text{-}of\text{-}int\ a) = word\text{-}of\text{-}int\ (a + 1)$
    and *wi-hom-pred*: $word\text{-}pred\ (word\text{-}of\text{-}int\ a) = word\text{-}of\text{-}int\ (a - 1)$
  by (*transfer*, *simp*)+

lemmas *wi-hom-syms* = *wi-homs* [*symmetric*]

lemmas *word-of-int-homs* = *wi-homs word-0-wi word-1-wi*

lemmas *word-of-int-hom-syms* = *word-of-int-homs* [*symmetric*]

lemma *double-eq-zero-iff*:
  $‹2 * a = 0 \longleftrightarrow a = 0 \vee a = 2\ \widehat{}\ (LENGTH('a) - Suc\ 0)›$
  for *a* :: ‹*'a*::*len word*›

proof −
define *n* **where** ‹*n* = *LENGTH*('*a*) − *Suc* 0›
then have ∗: ‹*LENGTH*('*a*) = *Suc n*›
by *simp*
have ‹*a* = 0› **if** ‹2 ∗ *a* = 0› **and** ‹*a* ≠ 2 ^ (*LENGTH*('*a*) − *Suc* 0)›
using *that* **by** *transfer*
(*auto simp*: *take-bit-eq-0-iff take-bit-eq-mod* ∗)
moreover have ‹2 ^ *LENGTH*('*a*) = (0 :: '*a word*)›
by *transfer simp*
then have ‹2 ∗ 2 ^ (*LENGTH*('*a*) − *Suc* 0) = (0 :: '*a word*)›
by (*simp add*: ∗)
ultimately show *?thesis*
by *auto*
qed

108.8 Ordering

instance *word* :: (*len*) *wellorder*
proof
fix *P* :: '*a word* ⇒ *bool* **and** *a*
assume ∗: (⋀*b*. (⋀*a*. *a* < *b* ⟹ *P a*) ⟹ *P b*)
have *wf* (*measure unat*) **..**
moreover have {(*a*, *b* :: ('*a*::*len*) *word*). *a* < *b*} ⊆ *measure unat*
by (*auto simp add*: *word-less-iff-unsigned* [**where** *?'a* = *nat*])
ultimately have *wf* {(*a*, *b* :: ('*a*::*len*) *word*). *a* < *b*}
by (*rule wf-subset*)
then show *P a* **using** ∗
by *induction blast*
qed

lemma *word-m1-ge* [*simp*]:
word-pred 0 ≥ *y*
by *transfer* (*simp add*: *mask-eq-exp-minus-1*)

lemma *word-less-alt*:
a < *b* ⟷ *uint a* < *uint b*
by (*fact word-less-def*)

lemma *word-zero-le* [*simp*]:
0 ≤ *y* **for** *y* :: '*a*::*len word*
by (*fact word-coorder.extremum*)

lemma *word-n1-ge* [*simp*]:
y ≤ − 1 **for** *y* :: '*a*::*len word*
by (*fact word-order.extremum*)

lemmas *word-not-simps* [*simp*] =
word-zero-le [*THEN leD*] *word-m1-ge* [*THEN leD*] *word-n1-ge* [*THEN leD*]

lemma *word-gt-0*:
0 < y ⟷ 0 ≠ y
for *y* :: *'a*::*len word*
by (*simp add*: *less-le*)

lemma *word-gt-0-no* [*simp*]:
‹(*0* :: *'a*::*len word*) < *numeral y ⟷* (*0* :: *'a*::*len word*) ≠ *numeral y*›
by (*fact word-gt-0*)

lemma *word-le-nat-alt*:
a ≤ b ⟷ unat a ≤ unat b
by *transfer* (*simp add*: *nat-le-eq-zle*)

lemma *word-less-nat-alt*:
a < b ⟷ unat a < unat b
by *transfer* (*auto simp*: *less-le* [*of 0*])

lemmas *unat-mono* = *word-less-nat-alt* [*THEN iffD1*]

lemma *wi-less*:
(*word-of-int n* < (*word-of-int m* :: *'a*::*len word*)) =
(*n mod 2* ^ *LENGTH*(*'a*) < *m mod 2* ^ *LENGTH*(*'a*))
by (*simp add*: *uint-word-of-int word-less-def*)

lemma *wi-le*:
(*word-of-int n* ≤ (*word-of-int m* :: *'a*::*len word*)) =
(*n mod 2* ^ *LENGTH*(*'a*) ≤ *m mod 2* ^ *LENGTH*(*'a*))
by (*simp add*: *uint-word-of-int word-le-def*)

lift-definition *word-sle* :: ‹*'a*::*len word* ⇒ *'a word* ⇒ *bool*›
is ‹λ*k l*. *signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*) *k* ≤ *signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*) *l*›
by (*simp flip*: *signed-take-bit-decr-length-iff*)

lift-definition *word-sless* :: ‹*'a*::*len word* ⇒ *'a word* ⇒ *bool*›
is ‹λ*k l*. *signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*) *k* < *signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*) *l*›
by (*simp flip*: *signed-take-bit-decr-length-iff*)

notation
word-sle (‹*'*(≤*s'*)›) **and**
word-sle (‹(‹*notation*=‹*infix* ≤*s*››-/ ≤*s* -)› [*51*, *51*] *50*) **and**
word-sless (‹*'*(<*s'*)›) **and**
word-sless (‹(‹*notation*=‹*infix* <*s*››-/ <*s* -)› [*51*, *51*] *50*)

notation (*input*)
word-sle (‹(‹*notation*=‹*infix* <=*s*››-/ <=*s* -)› [*51*, *51*] *50*)

lemma *word-sle-eq* [*code*]:

‹*a <=s b ⟷ sint a ≤ sint b*›
by *transfer simp*

lemma *word-sless-alt* [*code*]:
a <s b ⟷ sint a < sint b
by *transfer simp*

lemma *signed-ordering*: ‹*ordering word-sle word-sless*›
by (*standard*; *transfer*) (*auto simp flip*: *signed-take-bit-decr-length-iff*)

lemma *signed-linorder*: ‹*class.linorder word-sle word-sless*›
by (*standard*; *transfer*) (*auto simp*: *signed-take-bit-decr-length-iff*)

interpretation *signed*: *linorder word-sle word-sless*
by (*fact signed-linorder*)

lemma *word-sless-eq*:
‹*x <s y ⟷ x <=s y ∧ x ≠ y*›
by (*fact signed.less-le*)

lemma *minus-1-sless-0* [*simp*]:
‹*− 1 <s 0*›
by *transfer simp*

lemma *not-0-sless-minus-1* [*simp*]:
‹*¬ 0 <s − 1*›
by *transfer simp*

lemma *minus-1-sless-eq-0* [*simp*]:
‹*− 1 ≤s 0*›
by *transfer simp*

lemma *not-0-sless-eq-minus-1* [*simp*]:
‹*¬ 0 ≤s − 1*›
by *transfer simp*

108.9 Bit-wise operations

context
includes *bit-operations-syntax*
begin

lemma *take-bit-word-eq-self*:
‹*take-bit n w = w*› **if** ‹*LENGTH('a) ≤ n*› **for** *w* :: ‹*'a::len word*›
using *that* **by** *transfer simp*

lemma *take-bit-length-eq* [*simp*]:
‹*take-bit LENGTH('a) w = w*› **for** *w* :: ‹*'a::len word*›
by (*simp add*: *nat-le-linear take-bit-word-eq-self*)

lemma *bit-word-of-int-iff*:
‹*bit* (*word-of-int k* :: *'a*::*len word*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *bit k n*›
by (*simp add*: *bit-simps*)

lemma *bit-uint-iff*:
‹*bit* (*uint w*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *bit w n*›
for *w* :: ‹*'a*::*len word*›
by *transfer* (*simp add*: *bit-take-bit-iff*)

lemma *bit-sint-iff*:
‹*bit* (*sint w*) *n* ⟷ *n* ≥ *LENGTH*(*'a*) ∧ *bit w* (*LENGTH*(*'a*) − *1*) ∨ *bit w n*›
for *w* :: ‹*'a*::*len word*›
by *transfer* (*auto simp*: *bit-signed-take-bit-iff min-def le-less not-less*)

lemma *bit-word-ucast-iff*:
‹*bit* (*ucast w* :: *'b*::*len word*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *n* < *LENGTH*(*'b*) ∧
bit w n›
for *w* :: ‹*'a*::*len word*›
by (*auto simp add*: *bit-unsigned-iff bit-imp-le-length*)

lemma *bit-word-scast-iff*:
‹*bit* (*scast w* :: *'b*::*len word*) *n* ⟷
n < *LENGTH*(*'b*) ∧ (*bit w n* ∨ *LENGTH*(*'a*) ≤ *n* ∧ *bit w* (*LENGTH*(*'a*) −
Suc 0))›
for *w* :: ‹*'a*::*len word*›
by (*auto simp add*: *bit-signed-iff bit-imp-le-length min-def le-less dest*: *bit-imp-possible-bit*)

lemma *bit-word-iff-drop-bit-and* [*code*]:
‹*bit a n* ⟷ *drop-bit n a AND 1* = *1*› **for** *a* :: ‹*'a*::*len word*›
by (*simp add*: *bit-iff-odd-drop-bit odd-iff-mod-2-eq-one and-one-eq*)

lemma
word-not-def: *NOT* (*a*::*'a*::*len word*) = *word-of-int* (*NOT* (*uint a*))
and *word-and-def*: (*a*::*'a word*) *AND b* = *word-of-int* (*uint a AND uint b*)
and *word-or-def*: (*a*::*'a word*) *OR b* = *word-of-int* (*uint a OR uint b*)
and *word-xor-def*: (*a*::*'a word*) *XOR b* = *word-of-int* (*uint a XOR uint b*)
by (*transfer*, *simp add*: *take-bit-not-take-bit*)+

definition *even-word* :: ‹*'a*::*len word* ⇒ *bool*›
where [*code-abbrev*]: ‹*even-word* = *even*›

lemma *even-word-iff* [*code*]:
‹*even-word a* ⟷ *a AND 1* = *0*›
by (*simp add*: *and-one-eq even-iff-mod-2-eq-zero even-word-def*)

lemma *map-bit-range-eq-if-take-bit-eq*:
‹*map* (*bit k*) [*0*..<*n*] = *map* (*bit l*) [*0*..<*n*]›
if ‹*take-bit n k* = *take-bit n l*› **for** *k l* :: *int*

using *that*
proof (*induction n arbitrary*: *k l*)
case *0*
then show *?case*
by *simp*
next
case (*Suc n*)
from *Suc.prems* **have** ‹*take-bit n* (*k div 2*) = *take-bit n* (*l div 2*)›
by (*simp add*: *take-bit-Suc*)
then have ‹*map* (*bit* (*k div 2*)) [*0*..<*n*] = *map* (*bit* (*l div 2*)) [*0*..<*n*]›
by (*rule Suc.IH*)
moreover have ‹*bit* (*r div 2*) = *bit r* ∘ *Suc*› **for** *r* :: *int*
by (*simp add*: *fun-eq-iff bit-Suc*)
moreover from *Suc.prems* **have** ‹*even k* ⟷ *even l*›
by (*metis Zero-neq-Suc even-take-bit-eq*)
ultimately show *?case*
by (*simp only*: *map-Suc-upt upt-conv-Cons flip*: *list.map-comp*) (*simp add*: *bit-0*)
qed

lemma
take-bit-word-Bit0-eq [*simp*]: ‹*take-bit* (*numeral n*) (*numeral* (*num.Bit0 m*) :: *'a*::*len word*)
= *2* ∗ *take-bit* (*pred-numeral n*) (*numeral m*)› (**is** *?P*)
and *take-bit-word-Bit1-eq* [*simp*]: ‹*take-bit* (*numeral n*) (*numeral* (*num.Bit1 m*) :: *'a*::*len word*)
= *1* + *2* ∗ *take-bit* (*pred-numeral n*) (*numeral m*)› (**is** *?Q*)
and *take-bit-word-minus-Bit0-eq* [*simp*]: ‹*take-bit* (*numeral n*) (− *numeral* (*num.Bit0 m*) :: *'a*::*len word*)
= *2* ∗ *take-bit* (*pred-numeral n*) (− *numeral m*)› (**is** *?R*)
and *take-bit-word-minus-Bit1-eq* [*simp*]: ‹*take-bit* (*numeral n*) (− *numeral* (*num.Bit1 m*) :: *'a*::*len word*)
= *1* + *2* ∗ *take-bit* (*pred-numeral n*) (− *numeral* (*Num.inc m*))› (**is** *?S*)
proof −
define *w* :: ‹*'a*::*len word*›
where ‹*w* = *numeral m*›
moreover define *q* :: *nat*
where ‹*q* = *pred-numeral n*›
ultimately have *num*:
‹*numeral m* = *w*›
‹*numeral* (*num.Bit0 m*) = *2* ∗ *w*›
‹*numeral* (*num.Bit1 m*) = *1* + *2* ∗ *w*›
‹*numeral* (*Num.inc m*) = *1* + *w*›
‹*pred-numeral n* = *q*›
‹*numeral n* = *Suc q*›
by (*simp-all only*: *w-def q-def numeral-Bit0* [*of m*] *numeral-Bit1* [*of m*] *ac-simps numeral-inc numeral-eq-Suc flip*: *mult-2*)
have *even*: ‹*take-bit* (*Suc q*) (*2* ∗ *w*) = *2* ∗ *take-bit q w*› **for** *w* :: ‹*'a*::*len word*›
by (*rule bit-word-eqI*)

(*auto simp*: *bit-take-bit-iff bit-double-iff*)
have *odd*: ‹*take-bit* (*Suc q*) (*1 + 2 * w*) = *1 + 2 * take-bit q w*› **for** *w* :: ‹'*a*::*len word*›
by (*rule bit-eqI*)
(*auto simp*: *bit-take-bit-iff bit-double-iff even-bit-succ-iff*)
show *?P*
using *even* [*of w*] **by** (*simp add*: *num*)
show *?Q*
using *odd* [*of w*] **by** (*simp add*: *num*)
show *?R*
using *even* [*of* ‹*− w*›] **by** (*simp add*: *num*)
show *?S*
using *odd* [*of* ‹*− (1 + w)*›] **by** (*simp add*: *num*)
qed

108.10 More shift operations

lift-definition *signed-drop-bit* :: ‹*nat* ⇒ '*a word* ⇒ '*a*::*len word*›
is ‹λ*n*. *drop-bit n* ∘ *signed-take-bit* (*LENGTH*('*a*) − *Suc 0*)›
using *signed-take-bit-decr-length-iff*
by (*simp add*: *take-bit-drop-bit*) *force*

lemma *bit-signed-drop-bit-iff* [*bit-simps*]:
‹*bit* (*signed-drop-bit m w*) *n* ⟷ *bit w* (*if LENGTH*('*a*) − *m* ≤ *n* ∧ *n* < *LENGTH*('*a*) *then LENGTH*('*a*) − *1 else m + n*)›
for *w* :: ‹'*a*::*len word*›
by *transfer* (*simp add*: *bit-drop-bit-eq bit-signed-take-bit-iff min-def le-less not-less*, *auto*)

lemma [*code*]:
‹*Word.the-int* (*signed-drop-bit n w*) = *take-bit LENGTH*('*a*) (*drop-bit n* (*Word.the-signed-int w*))›
for *w* :: ‹'*a*::*len word*›
by *transfer simp*

lemma *signed-drop-bit-of-0* [*simp*]:
‹*signed-drop-bit n 0* = *0*›
by *transfer simp*

lemma *signed-drop-bit-of-minus-1* [*simp*]:
‹*signed-drop-bit n* (*− 1*) = *− 1*›
by *transfer simp*

lemma *signed-drop-bit-signed-drop-bit* [*simp*]:
‹*signed-drop-bit m* (*signed-drop-bit n w*) = *signed-drop-bit* (*m + n*) *w*›
for *w* :: ‹'*a*::*len word*›
proof (*cases* ‹*LENGTH*('*a*)›)
case *0*
then show *?thesis*

using *len-not-eq-0* **by** *blast*
next
case (*Suc n*)
then show *?thesis*
by (*force simp*: *bit-signed-drop-bit-iff not-le less-diff-conv ac-simps intro*!: *bit-word-eqI*)
qed

lemma *signed-drop-bit-0* [*simp*]:
‹*signed-drop-bit 0 w = w*›
by *transfer* (*simp add*: *take-bit-signed-take-bit*)

lemma *sint-signed-drop-bit-eq*:
‹*sint* (*signed-drop-bit n w*) = *drop-bit n* (*sint w*)›
by (*rule bit-eqI*; *cases n*) (*auto simp add*: *bit-simps not-less*)

108.11 Single-bit operations

lemma *set-bit-eq-idem-iff*:
‹*set-bit n w = w* ⟷ *bit w n* ∨ *n* ≥ *LENGTH*('*a*)›
for *w* :: ‹'*a*::*len word*›
unfolding *bit-eq-iff*
by (*auto simp*: *bit-simps not-le*)

lemma *unset-bit-eq-idem-iff*:
‹*unset-bit n w = w* ⟷ *bit w n* ⟶ *n* ≥ *LENGTH*('*a*)›
for *w* :: ‹'*a*::*len word*›
unfolding *bit-eq-iff*
by (*auto simp*: *bit-simps dest*: *bit-imp-le-length*)

lemma *flip-bit-eq-idem-iff*:
‹*flip-bit n w = w* ⟷ *n* ≥ *LENGTH*('*a*)›
for *w* :: ‹'*a*::*len word*›
by (*simp add*: *flip-bit-eq-if set-bit-eq-idem-iff unset-bit-eq-idem-iff*)

108.12 Rotation

lift-definition *word-rotr* :: ‹*nat* ⇒ '*a*::*len word* ⇒ '*a*::*len word*›
is ‹λ*n k*. *concat-bit* (*LENGTH*('*a*) − *n mod LENGTH*('*a*))
(*drop-bit* (*n mod LENGTH*('*a*)) (*take-bit LENGTH*('*a*) *k*))
(*take-bit* (*n mod LENGTH*('*a*)) *k*)›
using *take-bit-tightened* **by** *fastforce*

lift-definition *word-rotl* :: ‹*nat* ⇒ '*a*::*len word* ⇒ '*a*::*len word*›
is ‹λ*n k*. *concat-bit* (*n mod LENGTH*('*a*))
(*drop-bit* (*LENGTH*('*a*) − *n mod LENGTH*('*a*)) (*take-bit LENGTH*('*a*) *k*))
(*take-bit* (*LENGTH*('*a*) − *n mod LENGTH*('*a*)) *k*)›
using *take-bit-tightened* **by** *fastforce*

lift-definition *word-roti* :: ‹*int* ⇒ '*a*::*len word* ⇒ '*a*::*len word*›
is ‹λ*r k*. *concat-bit* (*LENGTH*('*a*) − *nat* (*r mod int LENGTH*('*a*)))

```
    (drop-bit (nat (r mod int LENGTH('a))) (take-bit LENGTH('a) k))
    (take-bit (nat (r mod int LENGTH('a))) k)›
  by (smt (verit, best) len-gt-0 nat-le-iff of-nat-0-less-iff pos-mod-bound
      take-bit-tightened)

lemma word-rotl-eq-word-rotr [code]:
  ‹word-rotl n = (word-rotr (LENGTH('a) − n mod LENGTH('a)) :: 'a::len word
⇒ 'a word)›
  by (rule ext, cases ‹n mod LENGTH('a) = 0›; transfer) simp-all

lemma word-roti-eq-word-rotr-word-rotl [code]:
  ‹word-roti i w =
    (if i ≥ 0 then word-rotr (nat i) w else word-rotl (nat (− i)) w)›
proof (cases ‹i ≥ 0›)
  case True
  moreover define n where ‹n = nat i›
  ultimately have ‹i = int n›
    by simp
  moreover have ‹word-roti (int n) = (word-rotr n :: - ⇒ 'a word)›
    by (rule ext, transfer) (simp add: nat-mod-distrib)
  ultimately show ?thesis
    by simp
next
  case False
  moreover define n where ‹n = nat (− i)›
  ultimately have ‹i = − int n› ‹n > 0›
    by simp-all
  moreover have ‹word-roti (− int n) = (word-rotl n :: - ⇒ 'a word)›
    by (rule ext, transfer)
      (simp add: zmod-zminus1-eq-if flip: of-nat-mod of-nat-diff)
  ultimately show ?thesis
    by simp
qed

lemma bit-word-rotr-iff [bit-simps]:
  ‹bit (word-rotr m w) n ⟷
    n < LENGTH('a) ∧ bit w ((n + m) mod LENGTH('a))›
  for w :: ‹'a::len word›
proof transfer
  fix k :: int and m n :: nat
  define q where ‹q = m mod LENGTH('a)›
  have ‹q < LENGTH('a)›
    by (simp add: q-def)
  then have ‹q ≤ LENGTH('a)›
    by simp
  have ‹m mod LENGTH('a) = q›
    by (simp add: q-def)
  moreover have ‹(n + m) mod LENGTH('a) = (n + q) mod LENGTH('a)›
    by (subst mod-add-right-eq [symmetric]) (simp add: ‹m mod LENGTH('a) =
```

q›)
moreover have ‹*n* < *LENGTH*('*a*) ∧
bit (*concat-bit* (*LENGTH*('*a*) − *q*) (*drop-bit q* (*take-bit LENGTH*('*a*) *k*)) (*take-bit q k*)) *n* ⟷
n < *LENGTH*('*a*) ∧ *bit k* ((*n* + *q*) *mod LENGTH*('*a*))›
using ‹*q* < *LENGTH*('*a*)›
by (*cases* ‹*q* + *n* ≥ *LENGTH*('*a*)›)
(*auto simp*: *bit-concat-bit-iff bit-drop-bit-eq*
bit-take-bit-iff le-mod-geq ac-simps)
ultimately show ‹*n* < *LENGTH*('*a*) ∧
bit (*concat-bit* (*LENGTH*('*a*) − *m mod LENGTH*('*a*))
(*drop-bit* (*m mod LENGTH*('*a*)) (*take-bit LENGTH*('*a*) *k*))
(*take-bit* (*m mod LENGTH*('*a*)) *k*)) *n*
⟷ *n* < *LENGTH*('*a*) ∧
(*n* + *m*) *mod LENGTH*('*a*) < *LENGTH*('*a*) ∧
bit k ((*n* + *m*) *mod LENGTH*('*a*))›
by *simp*
qed

lemma *bit-word-rotl-iff* [*bit-simps*]:
‹*bit* (*word-rotl m w*) *n* ⟷
n < *LENGTH*('*a*) ∧ *bit w* ((*n* + (*LENGTH*('*a*) − *m mod LENGTH*('*a*))) *mod LENGTH*('*a*))›
for *w* :: ‹'*a*::*len word*›
by (*simp add*: *word-rotl-eq-word-rotr bit-word-rotr-iff*)

lemma *bit-word-roti-iff* [*bit-simps*]:
‹*bit* (*word-roti k w*) *n* ⟷
n < *LENGTH*('*a*) ∧ *bit w* (*nat* ((*int n* + *k*) *mod int LENGTH*('*a*)))›
for *w* :: ‹'*a*::*len word*›
proof *transfer*
fix *k l* :: *int* **and** *n* :: *nat*
define *m* **where** ‹*m* = *nat* (*k mod int LENGTH*('*a*))›
have ‹*m* < *LENGTH*('*a*)›
by (*simp add*: *nat-less-iff m-def*)
then have ‹*m* ≤ *LENGTH*('*a*)›
by *simp*
have ‹*k mod int LENGTH*('*a*) = *int m*›
by (*simp add*: *nat-less-iff m-def*)
moreover have ‹(*int n* + *k*) *mod int LENGTH*('*a*) = *int* ((*n* + *m*) *mod LENGTH*('*a*))›
by (*subst mod-add-right-eq* [*symmetric*]) (*simp add*: *of-nat-mod* ‹*k mod int LENGTH*('*a*) = *int m*›)
moreover have ‹*n* < *LENGTH*('*a*) ∧
bit (*concat-bit* (*LENGTH*('*a*) − *m*) (*drop-bit m* (*take-bit LENGTH*('*a*) *l*)) (*take-bit m l*)) *n* ⟷
n < *LENGTH*('*a*) ∧ *bit l* ((*n* + *m*) *mod LENGTH*('*a*))›
using ‹*m* < *LENGTH*('*a*)›
by (*cases* ‹*m* + *n* ≥ *LENGTH*('*a*)›)

```
    (auto simp: bit-concat-bit-iff bit-drop-bit-eq
      bit-take-bit-iff nat-less-iff not-le not-less ac-simps
      le-diff-conv le-mod-geq)
  ultimately show ‹n < LENGTH('a)
    ∧ bit (concat-bit (LENGTH('a) − nat (k mod int LENGTH('a)))
          (drop-bit (nat (k mod int LENGTH('a))) (take-bit LENGTH('a) l))
          (take-bit (nat (k mod int LENGTH('a))) l)) n ⟷
      n < LENGTH('a)
    ∧ nat ((int n + k) mod int LENGTH('a)) < LENGTH('a)
    ∧ bit l (nat ((int n + k) mod int LENGTH('a)))›
    by simp
qed

lemma uint-word-rotr-eq:
  ‹uint (word-rotr n w) = concat-bit (LENGTH('a) − n mod LENGTH('a))
    (drop-bit (n mod LENGTH('a)) (uint w))
    (uint (take-bit (n mod LENGTH('a)) w))›
  for w :: ‹'a::len word›
  by transfer (simp add: take-bit-concat-bit-eq)

lemma [code]:
  ‹Word.the-int (word-rotr n w) = concat-bit (LENGTH('a) − n mod LENGTH('a))
    (drop-bit (n mod LENGTH('a)) (Word.the-int w))
    (Word.the-int (take-bit (n mod LENGTH('a)) w))›
  for w :: ‹'a::len word›
  using uint-word-rotr-eq [of n w] by simp
```

108.13 Split and cat operations

```
lift-definition word-cat :: ‹'a::len word ⇒ 'b::len word ⇒ 'c::len word›
  is ‹λk l. concat-bit LENGTH('b) l (take-bit LENGTH('a) k)›
  by (simp add: bit-eq-iff bit-concat-bit-iff bit-take-bit-iff)

lemma word-cat-eq:
  ‹(word-cat v w :: 'c::len word) = push-bit LENGTH('b) (ucast v) + ucast w›
  for v :: ‹'a::len word› and w :: ‹'b::len word›
  by transfer (simp add: concat-bit-eq ac-simps)

lemma word-cat-eq' [code]:
  ‹word-cat a b = word-of-int (concat-bit LENGTH('b) (uint b) (uint a))›
  for a :: ‹'a::len word› and b :: ‹'b::len word›
  by transfer (simp add: concat-bit-take-bit-eq)

lemma bit-word-cat-iff [bit-simps]:
  ‹bit (word-cat v w :: 'c::len word) n ⟷ n < LENGTH('c) ∧ (if n < LENGTH('b)
then bit w n else bit v (n − LENGTH('b)))›
  for v :: ‹'a::len word› and w :: ‹'b::len word›
  by transfer (simp add: bit-concat-bit-iff bit-take-bit-iff)
```

definition *word-split* :: ‹*'a*::*len word* ⇒ *'b*::*len word* × *'c*::*len word*›
where ‹*word-split w* =
(*ucast* (*drop-bit LENGTH*(*'c*) *w*) :: *'b*::*len word*, *ucast w* :: *'c*::*len word*)›

definition *word-rcat* :: ‹*'a*::*len word list* ⇒ *'b*::*len word*›
where ‹*word-rcat* = *word-of-int* ∘ *horner-sum uint* (*2* ^ *LENGTH*(*'a*)) ∘ *rev*›

108.14 More on conversions

lemma *int-word-sint*:
‹*sint* (*word-of-int x* :: *'a*::*len word*) = (*x* + *2* ^ (*LENGTH*(*'a*) − *1*)) *mod* *2* ^ *LENGTH*(*'a*) − *2* ^ (*LENGTH*(*'a*) − *1*)›
by *transfer* (*simp flip*: *take-bit-eq-mod add*: *signed-take-bit-eq-take-bit-shift*)

lemma *sint-sbintrunc'*: *sint* (*word-of-int bin* :: *'a word*) = *signed-take-bit* (*LENGTH*(*'a*::*len*) − *1*) *bin*
by (*simp add*: *signed-of-int*)

lemma *uint-sint*: *uint w* = *take-bit LENGTH*(*'a*) (*sint w*)
for *w* :: *'a*::*len word*
by *transfer* (*simp add*: *take-bit-signed-take-bit*)

lemma *bintr-uint*: *LENGTH*(*'a*) ≤ *n* ⟹ *take-bit n* (*uint w*) = *uint w*
for *w* :: *'a*::*len word*
by *transfer* (*simp add*: *min-def*)

lemma *wi-bintr*:
LENGTH(*'a*::*len*) ≤ *n* ⟹
word-of-int (*take-bit n w*) = (*word-of-int w* :: *'a word*)
by *transfer simp*

lemma *word-numeral-alt*: *numeral b* = *word-of-int* (*numeral b*)
by *simp*

declare *word-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *word-neg-numeral-alt*: − *numeral b* = *word-of-int* (− *numeral b*)
by *simp*

declare *word-neg-numeral-alt* [*symmetric*, *code-abbrev*]

lemma *uint-bintrunc* [*simp*]:
uint (*numeral bin* :: *'a word*) = *take-bit LENGTH*(*'a*::*len*) (*numeral bin*)
by *transfer rule*

lemma *uint-bintrunc-neg* [*simp*]:
uint (− *numeral bin* :: *'a word*) = *take-bit LENGTH*(*'a*::*len*) (− *numeral bin*)
by *transfer rule*

lemma *sint-sbintrunc* [*simp*]:
sint (*numeral bin* :: *'a word*) = *signed-take-bit* (*LENGTH*('a::*len*) − *1*) (*numeral bin*)
by *transfer simp*

lemma *sint-sbintrunc-neg* [*simp*]:
sint (− *numeral bin* :: *'a word*) = *signed-take-bit* (*LENGTH*('a::*len*) − *1*) (− *numeral bin*)
by *transfer simp*

lemma *unat-bintrunc* [*simp*]:
unat (*numeral bin* :: *'a*::*len word*) = *take-bit LENGTH*('a) (*numeral bin*)
by *transfer simp*

lemma *unat-bintrunc-neg* [*simp*]:
unat (− *numeral bin* :: *'a*::*len word*) = *nat* (*take-bit LENGTH*('a) (− *numeral bin*))
by *transfer simp*

lemma *size-0-eq*: *size w* = *0* ⟹ *v* = *w*
for *v w* :: *'a*::*len word*
by *transfer simp*

lemma *uint-ge-0* [*iff*]: *0* ≤ *uint x*
by (*fact unsigned-greater-eq*)

lemma *uint-lt2p* [*iff*]: *uint x* < *2* ^ *LENGTH*('a)
for *x* :: *'a*::*len word*
by (*fact unsigned-less*)

lemma *sint-ge*: − (*2* ^ (*LENGTH*('a) − *1*)) ≤ *sint x*
for *x* :: *'a*::*len word*
using *sint-greater-eq* [*of x*] **by** *simp*

lemma *sint-lt*: *sint x* < *2* ^ (*LENGTH*('a) − *1*)
for *x* :: *'a*::*len word*
using *sint-less* [*of x*] **by** *simp*

lemma *uint-m2p-neg*: *uint x* − *2* ^ *LENGTH*('a) < *0*
for *x* :: *'a*::*len word*
by (*simp only*: *diff-less-0-iff-less uint-lt2p*)

lemma *uint-m2p-not-non-neg*: ¬ *0* ≤ *uint x* − *2* ^ *LENGTH*('a)
for *x* :: *'a*::*len word*
by (*simp only*: *not-le uint-m2p-neg*)

lemma *lt2p-lem*: *LENGTH*('a) ≤ *n* ⟹ *uint w* < *2* ^ *n*
for *w* :: *'a*::*len word*
using *uint-bounded* [*of w*] **by** (*rule less-le-trans*) *simp*

lemma *uint-le-0-iff* [*simp*]: *uint x ≤ 0 ⟷ uint x = 0*
by (*fact uint-ge-0* [*THEN leD*, *THEN antisym-conv1*])

lemma *uint-nat*: *uint w = int* (*unat w*)
by *transfer simp*

lemma *uint-numeral*: *uint* (*numeral b* :: *'a*::*len word*) = *numeral b mod 2 ^ LENGTH*('*a*)
by (*simp flip*: *take-bit-eq-mod add*: *of-nat-take-bit*)

lemma *uint-neg-numeral*: *uint* (− *numeral b* :: *'a*::*len word*) = − *numeral b mod 2 ^ LENGTH*('*a*)
by (*simp flip*: *take-bit-eq-mod add*: *of-nat-take-bit*)

lemma *unat-numeral*: *unat* (*numeral b* :: *'a*::*len word*) = *numeral b mod 2 ^ LENGTH*('*a*)
by *transfer* (*simp add*: *take-bit-eq-mod nat-mod-distrib nat-power-eq*)

lemma *sint-numeral*:
sint (*numeral b* :: *'a*::*len word*) =
(*numeral b* + *2 ^*(*LENGTH*('*a*) − *1*)) *mod 2 ^ LENGTH*('*a*) − *2 ^*(*LENGTH*('*a*) − *1*)
by (*metis int-word-sint word-numeral-alt*)

lemma *word-of-int-0* [*simp*, *code-post*]: *word-of-int 0 = 0*
by (*fact of-int-0*)

lemma *word-of-int-1* [*simp*, *code-post*]: *word-of-int 1 = 1*
by (*fact of-int-1*)

lemma *word-of-int-neg-1* [*simp*]: *word-of-int* (− *1*) = − *1*
by *simp*

lemma *word-of-int-numeral* [*simp*] : (*word-of-int* (*numeral bin*) :: *'a*::*len word*) = *numeral bin*
by *simp*

lemma *word-int-case-wi*:
word-int-case f (*word-of-int i* :: *'b word*) = *f* (*i mod 2 ^ LENGTH*('*b*::*len*))
by (*simp add*: *uint-word-of-int word-int-case-eq-uint*)

lemma *word-int-split*:
P (*word-int-case f x*) =
(∀ *i*. *x* = (*word-of-int i* :: *'b*::*len word*) ∧ *0* ≤ *i* ∧ *i* < *2 ^ LENGTH*('*b*) ⟶ *P* (*f i*))
by *transfer* (*auto simp*: *take-bit-eq-mod*)

lemma *word-int-split-asm*:

P (*word-int-case f x*) =
(∄*n*. *x* = (*word-of-int n* :: *'b*::*len word*) ∧ *0* ≤ *n* ∧ *n* < *2* ^ *LENGTH*(*'b*::*len*) ∧ ¬ *P* (*f n*))
using *word-int-split* **by** *auto*

lemma *uint-range-size*: *0* ≤ *uint w* ∧ *uint w* < *2* ^ *size w*
by *transfer simp*

lemma *sint-range-size*: − (*2* ^ (*size w* − *Suc 0*)) ≤ *sint w* ∧ *sint w* < *2* ^ (*size w* − *Suc 0*)
by (*simp add*: *word-size sint-greater-eq sint-less*)

lemma *sint-above-size*: *2* ^ (*size w* − *1*) ≤ *x* ⟹ *sint w* < *x*
for *w* :: *'a*::*len word*
unfolding *word-size* **by** (*rule less-le-trans* [*OF sint-lt*])

lemma *sint-below-size*: *x* ≤ − (*2* ^ (*size w* − *1*)) ⟹ *x* ≤ *sint w*
for *w* :: *'a*::*len word*
unfolding *word-size* **by** (*rule order-trans* [*OF - sint-ge*])

lemma *word-unat-eq-iff*:
‹*v* = *w* ⟷ *unat v* = *unat w*›
for *v w* :: ‹*'a*::*len word*›
by (*fact word-eq-iff-unsigned*)

108.15 Testing bits

lemma *bin-nth-uint-imp*: *bit* (*uint w*) *n* ⟹ *n* < *LENGTH*(*'a*)
for *w* :: *'a*::*len word*
by (*simp add*: *bit-uint-iff*)

lemma *bin-nth-sint*:
LENGTH(*'a*) ≤ *n* ⟹
bit (*sint w*) *n* = *bit* (*sint w*) (*LENGTH*(*'a*) − *1*)
for *w* :: *'a*::*len word*
by (*transfer fixing*: *n*) (*simp add*: *bit-signed-take-bit-iff le-diff-conv min-def*)

lemma *num-of-bintr'*:
take-bit (*LENGTH*(*'a*::*len*)) (*numeral a* :: *int*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: *'a word*)
proof (*transfer fixing*: *a b*)
assume ‹*take-bit LENGTH*(*'a*) (*numeral a* :: *int*) = *numeral b*›
then have ‹*take-bit LENGTH*(*'a*) (*take-bit LENGTH*(*'a*) (*numeral a* :: *int*)) = *take-bit LENGTH*(*'a*) (*numeral b*)›
by *simp*
then show ‹*take-bit LENGTH*(*'a*) (*numeral a* :: *int*) = *take-bit LENGTH*(*'a*) (*numeral b*)›
by *simp*
qed

lemma *num-of-sbintr'*:
signed-take-bit (*LENGTH*('*a*::*len*) − *1*) (*numeral a* :: *int*) = (*numeral b*) ⟹
numeral a = (*numeral b* :: '*a word*)
proof (*transfer fixing*: *a b*)
assume ‹*signed-take-bit* (*LENGTH*('*a*) − *1*) (*numeral a* :: *int*) = *numeral b*›
then have ‹*take-bit LENGTH*('*a*) (*signed-take-bit* (*LENGTH*('*a*) − *1*) (*numeral a* :: *int*)) = *take-bit LENGTH*('*a*) (*numeral b*)›
by *simp*
then show ‹*take-bit LENGTH*('*a*) (*numeral a* :: *int*) = *take-bit LENGTH*('*a*) (*numeral b*)›
by (*simp add*: *take-bit-signed-take-bit*)
qed

lemma *num-abs-bintr*:
(*numeral x* :: '*a word*) =
word-of-int (*take-bit* (*LENGTH*('*a*::*len*)) (*numeral x*))
by *transfer simp*

lemma *num-abs-sbintr*:
(*numeral x* :: '*a word*) =
word-of-int (*signed-take-bit* (*LENGTH*('*a*::*len*) − *1*) (*numeral x*))
by *transfer* (*simp add*: *take-bit-signed-take-bit*)

cast – note, no arg for new length, as it's determined by type of result, thus in *cast w* = *w*, the type means cast to length of *w*!

lemma *bit-ucast-iff*:
‹*bit* (*ucast a* :: '*a*::*len word*) *n* ⟷ *n* < *LENGTH*('*a*::*len*) ∧ *bit a n*›
by *transfer* (*simp add*: *bit-take-bit-iff*)

lemma *ucast-id* [*simp*]: *ucast w* = *w*
by *transfer simp*

lemma *scast-id* [*simp*]: *scast w* = *w*
by *transfer* (*simp add*: *take-bit-signed-take-bit*)

lemma *ucast-mask-eq*:
‹*ucast* (*mask n* :: '*b word*) = *mask* (*min LENGTH*('*b*::*len*) *n*)›
by (*simp add*: *bit-eq-iff*) (*auto simp*: *bit-mask-iff bit-ucast-iff*)

— literal u(s)cast
lemma *ucast-bintr* [*simp*]:
ucast (*numeral w* :: '*a*::*len word*) =
word-of-int (*take-bit* (*LENGTH*('*a*)) (*numeral w*))
by *transfer simp*

lemma *scast-sbintr* [*simp*]:

scast (*numeral w* ::*'a*::*len word*) =
word-of-int (*signed-take-bit* (*LENGTH*(*'a*) − *Suc 0*) (*numeral w*))
by *transfer simp*

lemma *source-size*: *source-size* (*c*::*'a*::*len word* ⇒ -) = *LENGTH*(*'a*)
by *transfer simp*

lemma *target-size*: *target-size* (*c*::- ⇒ *'b*::*len word*) = *LENGTH*(*'b*)
by *transfer simp*

lemma *is-down*: *is-down c* ⟷ *LENGTH*(*'b*) ≤ *LENGTH*(*'a*)
for *c* :: *'a*::*len word* ⇒ *'b*::*len word*
by *transfer simp*

lemma *is-up*: *is-up c* ⟷ *LENGTH*(*'a*) ≤ *LENGTH*(*'b*)
for *c* :: *'a*::*len word* ⇒ *'b*::*len word*
by *transfer simp*

lemma *is-up-down*:
‹*is-up c* ⟷ *is-down d*›
for *c* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
and *d* :: ‹*'b*::*len word* ⇒ *'a*::*len word*›
by *transfer simp*

context
fixes *dummy-types* :: ‹*'a*::*len* × *'b*::*len*›
begin

private abbreviation (*input*) *UCAST* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
where ‹*UCAST* == *ucast*›

private abbreviation (*input*) *SCAST* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
where ‹*SCAST* == *scast*›

lemma *down-cast-same*:
‹*UCAST* = *scast*› **if** ‹*is-down UCAST*›
by (*rule ext*, *use that* **in** *transfer*) (*simp add*: *take-bit-signed-take-bit*)

lemma *sint-up-scast*:
‹*sint* (*SCAST w*) = *sint w*› **if** ‹*is-up SCAST*›
using *that* **by** *transfer* (*simp add*: *min-def Suc-leI le-diff-iff*)

lemma *uint-up-ucast*:
‹*uint* (*UCAST w*) = *uint w*› **if** ‹*is-up UCAST*›
using *that* **by** *transfer* (*simp add*: *min-def*)

lemma *ucast-up-ucast*:
‹*ucast* (*UCAST w*) = *ucast w*› **if** ‹*is-up UCAST*›
using *that* **by** *transfer* (*simp add*: *ac-simps*)

lemma *ucast-up-ucast-id*:
‹*ucast* (*UCAST w*) = *w*› **if** ‹*is-up UCAST*›
using *that* **by** (*simp add*: *ucast-up-ucast*)

lemma *scast-up-scast*:
‹*scast* (*SCAST w*) = *scast w*› **if** ‹*is-up SCAST*›
using *that* **by** *transfer* (*simp add*: *ac-simps*)

lemma *scast-up-scast-id*:
‹*scast* (*SCAST w*) = *w*› **if** ‹*is-up SCAST*›
using *that* **by** (*simp add*: *scast-up-scast*)

lemma *isduu*:
‹*is-up UCAST*› **if** ‹*is-down d*›
for *d* :: ‹*'b word* ⇒ *'a word*›
using *that is-up-down* [*of UCAST d*] **by** *simp*

lemma *isdus*:
‹*is-up SCAST*› **if** ‹*is-down d*›
for *d* :: ‹*'b word* ⇒ *'a word*›
using *that is-up-down* [*of SCAST d*] **by** *simp*

lemmas *ucast-down-ucast-id* = *isduu* [*THEN ucast-up-ucast-id*]
lemmas *scast-down-scast-id* = *isdus* [*THEN scast-up-scast-id*]

lemma *up-ucast-surj*:
‹*surj* (*ucast* :: *'b word* ⇒ *'a word*)› **if** ‹*is-up UCAST*›
by (*rule surjI*) (*use that* **in** ‹*rule ucast-up-ucast-id*›)

lemma *up-scast-surj*:
‹*surj* (*scast* :: *'b word* ⇒ *'a word*)› **if** ‹*is-up SCAST*›
by (*rule surjI*) (*use that* **in** ‹*rule scast-up-scast-id*›)

lemma *down-ucast-inj*:
‹*inj-on UCAST A*› **if** ‹*is-down* (*ucast* :: *'b word* ⇒ *'a word*)›
by (*rule inj-on-inverseI*) (*use that* **in** ‹*rule ucast-down-ucast-id*›)

lemma *down-scast-inj*:
‹*inj-on SCAST A*› **if** ‹*is-down* (*scast* :: *'b word* ⇒ *'a word*)›
by (*rule inj-on-inverseI*) (*use that* **in** ‹*rule scast-down-scast-id*›)

lemma *ucast-down-wi*:
‹*UCAST* (*word-of-int x*) = *word-of-int x*› **if** ‹*is-down UCAST*›
using *that* **by** *transfer simp*

lemma *ucast-down-no*:
‹*UCAST* (*numeral bin*) = *numeral bin*› **if** ‹*is-down UCAST*›
using *that* **by** *transfer simp*

end

lemmas *word-log-defs = word-and-def word-or-def word-xor-def word-not-def*

lemma *bit-last-iff*:
  ‹*bit w (LENGTH('a) − Suc 0) ⟷ sint w < 0*› (**is** ‹*?P ⟷ ?Q*›)
  for *w* :: ‹*'a*::*len word*›
  by (*simp add*: *bit-unsigned-iff sint-uint*)

lemma *drop-bit-eq-zero-iff-not-bit-last*:
  ‹*drop-bit (LENGTH('a) − Suc 0) w = 0 ⟷ ¬ bit w (LENGTH('a) − Suc 0)*›
  for *w* :: *'a*::*len word*
proof (*cases* ‹*LENGTH('a)*›)
  case (*Suc n*)
  then show *?thesis*
    apply *transfer*
    apply (*simp add*: *take-bit-drop-bit*)
    by (*simp add*: *bit-iff-odd-drop-bit drop-bit-take-bit odd-iff-mod-2-eq-one*)
qed *auto*

lemma *unat-div*:
  ‹*unat (x div y) = unat x div unat y*›
  by (*fact unat-div-distrib*)

lemma *unat-mod*:
  ‹*unat (x mod y) = unat x mod unat y*›
  by (*fact unat-mod-distrib*)

108.16 Word Arithmetic

lemmas *less-eq-word-numeral-numeral* [*simp*] =
  word-le-def [*of* ‹*numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
  for *a b*
lemmas *less-word-numeral-numeral* [*simp*] =
  word-less-def [*of* ‹*numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
  for *a b*
lemmas *less-eq-word-minus-numeral-numeral* [*simp*] =
  word-le-def [*of* ‹*− numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
  for *a b*
lemmas *less-word-minus-numeral-numeral* [*simp*] =
  word-less-def [*of* ‹*− numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
  for *a b*
lemmas *less-eq-word-numeral-minus-numeral* [*simp*] =
  word-le-def [*of* ‹*numeral a*› ‹*− numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg*

unsigned-minus-1-eq-mask mask-eq-exp-minus-1]
for *a b*
lemmas *less-word-numeral-minus-numeral* [*simp*] =
word-less-def [*of* ‹*numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *less-eq-word-minus-numeral-minus-numeral* [*simp*] =
word-le-def [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *less-word-minus-numeral-minus-numeral* [*simp*] =
word-less-def [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *less-word-numeral-minus-1* [*simp*] =
word-less-def [*of* ‹*numeral a*› ‹− *1*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *less-word-minus-numeral-minus-1* [*simp*] =
word-less-def [*of* ‹− *numeral a*› ‹− *1*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*

lemmas *sless-eq-word-numeral-numeral* [*simp*] =
word-sle-eq [*of* ‹*numeral a*› ‹*numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-word-numeral-numeral* [*simp*] =
word-sless-alt [*of* ‹*numeral a*› ‹*numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-eq-word-minus-numeral-numeral* [*simp*] =
word-sle-eq [*of* ‹− *numeral a*› ‹*numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-word-minus-numeral-numeral* [*simp*] =
word-sless-alt [*of* ‹− *numeral a*› ‹*numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-eq-word-numeral-minus-numeral* [*simp*] =
word-sle-eq [*of* ‹*numeral a*› ‹− *numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-word-numeral-minus-numeral* [*simp*] =
word-sless-alt [*of* ‹*numeral a*› ‹− *numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-eq-word-minus-numeral-minus-numeral* [*simp*] =
word-sle-eq [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*
lemmas *sless-word-minus-numeral-minus-numeral* [*simp*] =
word-sless-alt [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified sint-sbintrunc sint-sbintrunc-neg*]
for *a b*

lemmas *div-word-numeral-numeral* [*simp*] =

word-div-def [*of* ‹*numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *div-word-minus-numeral-numeral* [*simp*] =
word-div-def [*of* ‹− *numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *div-word-numeral-minus-numeral* [*simp*] =
word-div-def [*of* ‹*numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *div-word-minus-numeral-minus-numeral* [*simp*] =
word-div-def [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *div-word-minus-1-numeral* [*simp*] =
word-div-def [*of* ‹− 1› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *div-word-minus-1-minus-numeral* [*simp*] =
word-div-def [*of* ‹− 1› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*

lemmas *mod-word-numeral-numeral* [*simp*] =
word-mod-def [*of* ‹*numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *mod-word-minus-numeral-numeral* [*simp*] =
word-mod-def [*of* ‹− *numeral a*› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *mod-word-numeral-minus-numeral* [*simp*] =
word-mod-def [*of* ‹*numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *mod-word-minus-numeral-minus-numeral* [*simp*] =
word-mod-def [*of* ‹− *numeral a*› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *mod-word-minus-1-numeral* [*simp*] =
word-mod-def [*of* ‹− 1› ‹*numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*
lemmas *mod-word-minus-1-minus-numeral* [*simp*] =
word-mod-def [*of* ‹− 1› ‹− *numeral b*›, *simplified uint-bintrunc uint-bintrunc-neg unsigned-minus-1-eq-mask mask-eq-exp-minus-1*]
for *a b*

lemma *signed-drop-bit-of-1* [*simp*]:
‹*signed-drop-bit n* (*1* :: *'a*::*len word*) = *of-bool* (*LENGTH*('*a*) = *1* ∨ *n* = *0*)›
apply (*transfer fixing*: *n*)
apply (*cases* ‹*LENGTH*('*a*)›)
apply (*auto simp*: *take-bit-signed-take-bit*)
apply (*auto simp*: *take-bit-drop-bit gr0-conv-Suc simp flip*: *take-bit-eq-self-iff-drop-bit-eq-0*)
done

lemma *take-bit-word-beyond-length-eq*:
‹*take-bit n w* = *w*› **if** ‹*LENGTH*('*a*) ≤ *n*› **for** *w* :: ‹*'a*::*len word*›
by (*simp add*: *take-bit-word-eq-self that*)

lemmas *word-div-no* [*simp*] = *word-div-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-mod-no* [*simp*] = *word-mod-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-less-no* [*simp*] = *word-less-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-le-no* [*simp*] = *word-le-def* [*of numeral a numeral b*] **for** *a b*
lemmas *word-sless-no* [*simp*] = *word-sless-eq* [*of numeral a numeral b*] **for** *a b*
lemmas *word-sle-no* [*simp*] = *word-sle-eq* [*of numeral a numeral b*] **for** *a b*

lemma *size-0-same'*: *size w* = *0* ⟹ *w* = *v*
for *v w* :: *'a*::*len word*
by (*unfold word-size*) *simp*

lemmas *size-0-same* = *size-0-same'* [*unfolded word-size*]

lemmas *unat-eq-0* = *unat-0-iff*
lemmas *unat-eq-zero* = *unat-0-iff*

lemma *mask-1*: *mask 1* = *1*
by *simp*

lemma *mask-Suc-0*: *mask* (*Suc 0*) = *1*
by *simp*

lemma *bin-last-bintrunc*: *odd* (*take-bit l n*) ⟷ *l* > *0* ∧ *odd n*
by *simp*

lemma *push-bit-word-beyond* [*simp*]:
‹*push-bit n w* = *0*› **if** ‹*LENGTH*('*a*) ≤ *n*› **for** *w* :: ‹*'a*::*len word*›
using *that* **by** (*transfer fixing*: *n*) (*simp add*: *take-bit-push-bit*)

lemma *drop-bit-word-beyond* [*simp*]:
‹*drop-bit n w* = *0*› **if** ‹*LENGTH*('*a*) ≤ *n*› **for** *w* :: ‹*'a*::*len word*›
using *that* **by** (*transfer fixing*: *n*) (*simp add*: *drop-bit-take-bit*)

lemma *signed-drop-bit-beyond*:
‹*signed-drop-bit n w* = (*if bit w* (*LENGTH*('*a*) − *Suc 0*) *then* − *1 else 0*)›
if ‹*LENGTH*('*a*) ≤ *n*› **for** *w* :: ‹*'a*::*len word*›
by (*rule bit-word-eqI*) (*simp add*: *bit-signed-drop-bit-iff that*)

lemma *take-bit-numeral-minus-numeral-word* [*simp*]:
  ‹*take-bit* (*numeral m*) (− *numeral n* :: *'a*::*len word*) =
    (*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* ‹*LENGTH*(*'a*) ≤ *numeral m*›)
  case *True*
  then have ∗: ‹(*take-bit* (*numeral m*) :: *'a word* ⇒ *'a word*) = *id*›
    by (*simp add*: *fun-eq-iff take-bit-word-eq-self*)
  have ∗∗: ‹*2* ^ *numeral m* = (*0* :: *'a word*)›
    using *True* **by** (*simp flip*: *exp-eq-zero-iff*)
  show *?thesis*
    by (*auto simp only*: ∗ ∗∗ *split*: *option.split*
    dest!: *take-bit-num-eq-None-imp* [**where** *?'a* = ‹*'a word*›] *take-bit-num-eq-Some-imp* [**where** *?'a* = ‹*'a word*›])
      simp-all
next
  case *False*
  then show *?thesis*
    by (*transfer fixing*: *m n*) *simp*
qed

lemma *of-nat-inverse*:
  ‹*word-of-nat r* = *a* ⟹ *r* < *2* ^ *LENGTH*(*'a*) ⟹ *unat a* = *r*›
  for *a* :: ‹*'a*::*len word*›
  by (*metis id-apply of-nat-eq-id take-bit-nat-eq-self-iff unsigned-of-nat*)

108.17 Transferring goals from words to ints

lemma *word-ths*:
  shows *word-succ-p1*: *word-succ a* = *a* + *1*
    and *word-pred-m1*: *word-pred a* = *a* − *1*
    and *word-pred-succ*: *word-pred* (*word-succ a*) = *a*
    and *word-succ-pred*: *word-succ* (*word-pred a*) = *a*
    and *word-mult-succ*: *word-succ a* ∗ *b* = *b* + *a* ∗ *b*
  by (*transfer*, *simp add*: *algebra-simps*)+

lemma *uint-cong*: *x* = *y* ⟹ *uint x* = *uint y*
  by *simp*

lemma *uint-word-ariths*:
  fixes *a b* :: *'a*::*len word*
  shows *uint* (*a* + *b*) = (*uint a* + *uint b*) *mod* *2* ^ *LENGTH*(*'a*::*len*)
    and *uint* (*a* − *b*) = (*uint a* − *uint b*) *mod* *2* ^ *LENGTH*(*'a*)
    and *uint* (*a* ∗ *b*) = *uint a* ∗ *uint b mod* *2* ^ *LENGTH*(*'a*)
    and *uint* (− *a*) = − *uint a mod* *2* ^ *LENGTH*(*'a*)
    and *uint* (*word-succ a*) = (*uint a* + *1*) *mod* *2* ^ *LENGTH*(*'a*)
    and *uint* (*word-pred a*) = (*uint a* − *1*) *mod* *2* ^ *LENGTH*(*'a*)
    and *uint* (*0* :: *'a word*) = *0 mod* *2* ^ *LENGTH*(*'a*)

and *uint* (*1* :: *'a word*) = *1 mod 2* ^ *LENGTH*(*'a*)
by (*simp-all only*: *word-arith-wis uint-word-of-int-eq flip*: *take-bit-eq-mod*)

lemma *uint-word-arith-bintrs*:
fixes *a b* :: *'a*::*len word*
shows *uint* (*a* + *b*) = *take-bit* (*LENGTH*(*'a*)) (*uint a* + *uint b*)
and *uint* (*a* − *b*) = *take-bit* (*LENGTH*(*'a*)) (*uint a* − *uint b*)
and *uint* (*a* ∗ *b*) = *take-bit* (*LENGTH*(*'a*)) (*uint a* ∗ *uint b*)
and *uint* (− *a*) = *take-bit* (*LENGTH*(*'a*)) (− *uint a*)
and *uint* (*word-succ a*) = *take-bit* (*LENGTH*(*'a*)) (*uint a* + *1*)
and *uint* (*word-pred a*) = *take-bit* (*LENGTH*(*'a*)) (*uint a* − *1*)
and *uint* (*0* :: *'a word*) = *take-bit* (*LENGTH*(*'a*)) *0*
and *uint* (*1* :: *'a word*) = *take-bit* (*LENGTH*(*'a*)) *1*
by (*simp-all add*: *uint-word-ariths take-bit-eq-mod*)

context
fixes *a b* :: *'a*::*len word*
begin

lemma *sint-word-add*: *sint* (*a* + *b*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (*sint a* + *sint b*)
by *transfer* (*simp add*: *signed-take-bit-add*)

lemma *sint-word-diff*: *sint* (*a* − *b*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (*sint a* − *sint b*)
by *transfer* (*simp add*: *signed-take-bit-diff*)

lemma *sint-word-mult*: *sint* (*a* ∗ *b*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (*sint a* ∗ *sint b*)
by *transfer* (*simp add*: *signed-take-bit-mult*)

lemma *sint-word-minus*: *sint* (− *a*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (− *sint a*)
by *transfer* (*simp add*: *signed-take-bit-minus*)

lemma *sint-word-succ*: *sint* (*word-succ a*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (*sint a* + *1*)
by (*metis of-int-sint scast-id sint-sbintrunc' wi-hom-succ*)

lemma *sint-word-pred*: *sint* (*word-pred a*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) (*sint a* − *1*)
by (*metis of-int-sint scast-id sint-sbintrunc' wi-hom-pred*)

lemma *sint-word-01*:
sint (*0* :: *'a word*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) *0*
sint (*1* :: *'a word*) = *signed-take-bit* (*LENGTH*(*'a*) − *1*) *1*
by (*simp-all add*: *sint-uint*)

end

lemmas *sint-word-ariths* =
sint-word-add sint-word-diff sint-word-mult sint-word-minus
sint-word-succ sint-word-pred sint-word-01

lemma *word-pred-0-n1*: *word-pred 0 = word-of-int (− 1)*
unfolding *word-pred-m1* **by** *simp*

lemma *succ-pred-no* [*simp*]:
word-succ (numeral w) = numeral w + 1
word-pred (numeral w) = numeral w − 1
word-succ (− numeral w) = − numeral w + 1
word-pred (− numeral w) = − numeral w − 1
by (*simp-all add*: *word-succ-p1 word-pred-m1*)

lemma *word-sp-01* [*simp*]:
word-succ (− 1) = 0 ∧ word-succ 0 = 1 ∧ word-pred 0 = − 1 ∧ word-pred 1 = 0
by (*simp-all add*: *word-succ-p1 word-pred-m1*)

— alternative approach to lifting arithmetic equalities
lemma *word-of-int-Ex*: *∃ y. x = word-of-int y*
by (*rule-tac x=uint x* **in** *exI*) *simp*

108.18 Order on fixed-length words

lift-definition *udvd* :: ‹*'a*::*len word ⇒ 'a*::*len word ⇒ bool*› (**infixl** ‹*udvd*› *50*)
is ‹*λk l. take-bit LENGTH('a) k dvd take-bit LENGTH('a) l*› **by** *simp*

lemma *udvd-iff-dvd*:
‹*x udvd y ⟷ unat x dvd unat y*›
by *transfer* (*simp add*: *nat-dvd-iff*)

lemma *udvd-iff-dvd-int*:
‹*v udvd w ⟷ uint v dvd uint w*›
by *transfer rule*

lemma *udvdI* [*intro*]:
‹*v udvd w*› **if** ‹*unat w = unat v ∗ unat u*›
proof −
from *that* **have** ‹*unat v dvd unat w*› **..**
then show *?thesis*
by (*simp add*: *udvd-iff-dvd*)
qed

lemma *udvdE* [*elim*]:
fixes *v w* :: ‹*'a*::*len word*›
assumes ‹*v udvd w*›

```
  obtains u :: ‹'a word› where ‹unat w = unat v * unat u›
proof (cases ‹v = 0›)
  case True
  moreover from True ‹v udvd w› have ‹w = 0›
    by transfer simp
  ultimately show thesis
    using that by simp
next
  case False
  then have ‹unat v > 0›
    by (simp add: unat-gt-0)
  from ‹v udvd w› have ‹unat v dvd unat w›
    by (simp add: udvd-iff-dvd)
  then obtain n where ‹unat w = unat v * n› ..
  moreover have ‹n < 2 ^ LENGTH('a)›
  proof (rule ccontr)
    assume ‹¬ n < 2 ^ LENGTH('a)›
    then have ‹n ≥ 2 ^ LENGTH('a)›
      by (simp add: not-le)
    then have ‹unat v * n ≥ 2 ^ LENGTH('a)›
      using ‹unat v > 0› mult-le-mono [of 1 ‹unat v› ‹2 ^ LENGTH('a)› n]
      by simp
    with ‹unat w = unat v * n›
    have ‹unat w ≥ 2 ^ LENGTH('a)›
      by simp
    with unsigned-less [of w, where ?'a = nat] show False
      by linarith
  qed
  ultimately have ‹unat w = unat v * unat (word-of-nat n :: 'a word)›
    by (auto simp: take-bit-nat-eq-self-iff unsigned-of-nat intro: sym)
  with that show thesis .
qed

lemma udvd-imp-mod-eq-0:
  ‹w mod v = 0› if ‹v udvd w›
  using that by transfer simp

lemma mod-eq-0-imp-udvd [intro?]:
  ‹v udvd w› if ‹w mod v = 0›
  by (metis mod-0-imp-dvd that udvd-iff-dvd unat-0 unat-mod-distrib)

lemma udvd-imp-dvd:
  ‹v dvd w› if ‹v udvd w› for v w :: ‹'a::len word›
proof −
  from that obtain u :: ‹'a word› where ‹unat w = unat v * unat u› ..
  then have ‹w = v * u›
    by (metis of-nat-mult of-nat-unat word-mult-def word-of-int-uint)
  then show ‹v dvd w› ..
qed
```

lemma *exp-dvd-iff-exp-udvd*:
‹*2 ^ n dvd w* ⟷ *2 ^ n udvd w*› **for** *v w* :: ‹*'a*::*len word*›
proof
assume ‹*2 ^ n udvd w*› **then show** ‹*2 ^ n dvd w*›
by (*rule udvd-imp-dvd*)
next
assume ‹*2 ^ n dvd w*›
then obtain *u* :: ‹*'a word*› **where** ‹*w* = *2 ^ n* ∗ *u*› **..**
then have ‹*w* = *push-bit n u*›
by (*simp add*: *push-bit-eq-mult*)
then show ‹*2 ^ n udvd w*›
by *transfer* (*simp add*: *take-bit-push-bit dvd-eq-mod-eq-0 flip*: *take-bit-eq-mod*)
qed

lemma *udvd-nat-alt*:
‹*a udvd b* ⟷ (∃ *n*. *unat b* = *n* ∗ *unat a*)›
by (*auto simp*: *udvd-iff-dvd*)

lemma *udvd-unfold-int*:
‹*a udvd b* ⟷ (∃ *n*≥*0*. *uint b* = *n* ∗ *uint a*)›
unfolding *udvd-iff-dvd-int*
by (*metis dvd-div-mult-self dvd-triv-right uint-div-distrib uint-ge-0*)

lemma *unat-minus-one*:
‹*unat* (*w* − *1*) = *unat w* − *1*› **if** ‹*w* ≠ *0*›
proof −
have *0* ≤ *uint w* **by** (*fact uint-nonnegative*)
moreover from *that* **have** *0* ≠ *uint w*
by (*simp add*: *uint-0-iff*)
ultimately have *1* ≤ *uint w*
by *arith*
from *uint-lt2p* [*of w*] **have** *uint w* − *1* < *2 ^ LENGTH*('*a*)
by *arith*
with ‹*1* ≤ *uint w*› **have** (*uint w* − *1*) *mod 2 ^ LENGTH*('*a*) = *uint w* − *1*
by (*auto intro*: *mod-pos-pos-trivial*)
with ‹*1* ≤ *uint w*› **have** *nat* ((*uint w* − *1*) *mod 2 ^ LENGTH*('*a*)) = *nat* (*uint w*) − *1*
by (*auto simp del*: *nat-uint-eq*)
then show *?thesis*
by (*metis uint-word-ariths*(*6*) *unat-eq-nat-uint word-pred-m1*)
qed

lemma *measure-unat*: *p* ≠ *0* ⟹ *unat* (*p* − *1*) < *unat p*
by (*simp add*: *unat-minus-one*) (*simp add*: *unat-0-iff* [*symmetric*])

lemmas *uint-add-ge0* [*simp*] = *add-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]
lemmas *uint-mult-ge0* [*simp*] = *mult-nonneg-nonneg* [*OF uint-ge-0 uint-ge-0*]

lemma *uint-sub-lt2p* [*simp*]: *uint x − uint y < 2 ^ LENGTH('a)*
for *x :: 'a::len word* **and** *y :: 'b::len word*
using *uint-ge-0* [*of y*] *uint-lt2p* [*of x*] **by** *arith*

108.19 Conditions for the addition (etc) of two words to overflow

lemma *uint-add-lem*:
(uint x + uint y < 2 ^ LENGTH('a)) =
(uint (x + y) = uint x + uint y)
for *x y :: 'a::len word*
by (*metis add.right-neutral add-mono-thms-linordered-semiring(1) mod-pos-pos-trivial of-nat-0-le-iff uint-lt2p uint-nat uint-word-ariths(1)*)

lemma *uint-mult-lem*:
*(uint x * uint y < 2 ^ LENGTH('a)) =*
*(uint (x * y) = uint x * uint y)*
for *x y :: 'a::len word*
by (*metis mod-pos-pos-trivial uint-lt2p uint-mult-ge0 uint-word-ariths(3)*)

lemma *uint-sub-lem*: *uint x ≥ uint y ⟷ uint (x − y) = uint x − uint y*
by (*simp add*: *uint-word-arith-bintrs take-bit-int-eq-self-iff*)

lemma *uint-add-le*: *uint (x + y) ≤ uint x + uint y*
unfolding *uint-word-ariths* **by** (*simp add*: *zmod-le-nonneg-dividend*)

lemma *uint-sub-ge*: *uint (x − y) ≥ uint x − uint y*
by (*smt* (*verit*, *ccfv-SIG*) *uint-nonnegative uint-sub-lem*)

lemma *int-mod-ge*: *‹a ≤ a mod n›* **if** *‹a < n›* *‹0 < n›*
for *a n :: int*
using *that order.trans* [*of a 0 ‹a mod n›*] **by** (*cases ‹a < 0›*) *auto*

lemma *mod-add-if-z*:
⟦x < z; y < z; 0 ≤ y; 0 ≤ x; 0 ≤ z⟧ ⟹
(x + y) mod z = (if x + y < z then x + y else x + y − z)
for *x y z :: int*
by (*smt* (*verit*, *best*) *minus-mod-self2 mod-pos-pos-trivial*)

lemma *uint-plus-if'*:
uint (a + b) =
(if uint a + uint b < 2 ^ LENGTH('a) then uint a + uint b
else uint a + uint b − 2 ^ LENGTH('a))
for *a b :: 'a::len word*
using *mod-add-if-z* [*of uint a - uint b*] **by** (*simp add*: *uint-word-ariths*)

lemma *mod-sub-if-z*:
⟦x < z; y < z; 0 ≤ y; 0 ≤ x; 0 ≤ z⟧ ⟹
(x − y) mod z = (if y ≤ x then x − y else x − y + z)

for *x y z* :: *int*
using *mod-pos-pos-trivial* [*of x − y + z z*] **by** (*auto simp*: *not-le*)

lemma *uint-sub-if'*:
uint (*a − b*) =
(*if uint b* ≤ *uint a then uint a − uint b*
else uint a − uint b + 2 ^ *LENGTH*('*a*))
for *a b* :: '*a*::*len word*
using *mod-sub-if-z* [*of uint a - uint b*] **by** (*simp add*: *uint-word-ariths*)

lemma *word-of-int-inverse*:
word-of-int r = a ⟹ *0* ≤ *r* ⟹ *r < 2* ^ *LENGTH*('*a*) ⟹ *uint a = r*
for *a* :: '*a*::*len word*
by *transfer* (*simp add*: *take-bit-int-eq-self*)

lemma *unat-split*: *P* (*unat x*) ⟷ (∀ *n*. *of-nat n = x* ∧ *n < 2*^*LENGTH*('*a*) ⟶ *P n*)
for *x* :: '*a*::*len word*
by (*auto simp*: *unsigned-of-nat take-bit-nat-eq-self*)

lemma *unat-split-asm*: *P* (*unat x*) ⟷ (∄ *n*. *of-nat n = x* ∧ *n < 2*^*LENGTH*('*a*) ∧ ¬ *P n*)
for *x* :: '*a*::*len word*
using *unat-split* **by** *auto*

lemma *un-ui-le*:
‹*unat a* ≤ *unat b* ⟷ *uint a* ≤ *uint b*›
by *transfer* (*simp add*: *nat-le-iff*)

lemma *unat-plus-if'*:
‹*unat* (*a + b*) =
(*if unat a + unat b < 2* ^ *LENGTH*('*a*)
then unat a + unat b
else unat a + unat b − 2 ^ *LENGTH*('*a*))› **for** *a b* :: ‹'*a*::*len word*›
apply (*auto simp*: *not-less le-iff-add*)
using *of-nat-inverse* **apply** *force*
by (*smt* (*verit*, *ccfv-SIG*) *numeral-Bit0 numerals*(*1*) *of-nat-0-le-iff of-nat-1 of-nat-add of-nat-eq-iff of-nat-power of-nat-unat uint-plus-if'*)

lemma *unat-sub-if-size*:
unat (*x − y*) =
(*if unat y* ≤ *unat x*
then unat x − unat y
else unat x + 2 ^ *size x − unat y*)
proof −
have *nat* (*uint x − uint y + 2* ^ *LENGTH*('*a*)) = *nat* (*uint x*) + *2* ^ *LENGTH*('*a*) − *nat* (*uint y*)
proof −
have *nat* (*uint x − uint y + 2* ^ *LENGTH*('*a*)) = *nat* (*uint x + 2* ^

LENGTH('a) − uint y)
by *simp*
also have ... = *nat (uint x + 2 ^ LENGTH('a)) − nat (uint y)*
by (*simp add*: *nat-diff-distrib'*)
also have ... = *nat (uint x) + 2 ^ LENGTH('a) − nat (uint y)*
by (*simp add*: *nat-add-distrib nat-power-eq*)
finally show *?thesis* **.**
qed
then show *?thesis*
by (*metis nat-diff-distrib' uint-range-size uint-sub-if' un-ui-le unat-eq-nat-uint word-size*)
qed

lemmas *unat-sub-if' = unat-sub-if-size* [*unfolded word-size*]

lemma *uint-split*:
P (uint x) = (∀ i. word-of-int i = x ∧ 0 ≤ i ∧ i < 2^LENGTH('a) ⟶ P i)
for *x* :: *'a*::*len word*
by *transfer* (*auto simp*: *take-bit-eq-mod*)

lemma *uint-split-asm*:
P (uint x) = (∄ i. word-of-int i = x ∧ 0 ≤ i ∧ i < 2^LENGTH('a) ∧ ¬ P i)
for *x* :: *'a*::*len word*
by (*auto simp*: *unsigned-of-int take-bit-int-eq-self*)

108.20 Some proof tool support

lemma *power-False-cong*: *False ⟹ a ^ b = c ^ d*
by *auto*

lemmas *unat-splits = unat-split unat-split-asm*

lemmas *unat-arith-simps =*
word-le-nat-alt word-less-nat-alt
word-unat-eq-iff
unat-sub-if' unat-plus-if' unat-div unat-mod

lemmas *uint-splits = uint-split uint-split-asm*

lemmas *uint-arith-simps =*
word-le-def word-less-alt
word-uint-eq-iff
uint-sub-if' uint-plus-if'

— *unat-arith-tac*: tactic to reduce word arithmetic to *nat*, try to solve via *arith*
ML ‹
val unat-arith-simpset =
@{*context*} (∗ *TODO*: *completely explicitly determined simpset* ∗)
|> *Simplifier.add-simps* @{*thms unat-arith-simps*}

```
  |> fold Splitter.add-split @{thms if-split-asm}
  |> fold Simplifier.add-cong @{thms power-False-cong}
  |> Simplifier.simpset-of

fun unat-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (put-simpset unat-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
        (put-simpset HOL-ss ctxt
          |> fold Splitter.add-split @{thms unat-splits}
          |> fold Simplifier.add-cong @{thms power-False-cong})),
      rewrite-goals-tac ctxt @{thms word-size},
      ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                        REPEAT (eresolve-tac ctxt [conjE] n) THEN
                        REPEAT (dresolve-tac ctxt @{thms of-nat-inverse} n THEN
assume-tac ctxt n)),
      TRYALL arith-tac' ]
  end

fun unat-arith-tac ctxt = SELECT-GOAL (EVERY (unat-arith-tacs ctxt))
›

method-setup unat-arith =
  ‹Scan.succeed (SIMPLE-METHOD' o unat-arith-tac)›
  solving word arithmetic via natural numbers and arith

— uint-arith-tac: reduce to arithmetic on int, try to solve by arith
ML ‹
val uint-arith-simpset =
  @{context} (* TODO: completely explicitly determined simpset *)
  |> Simplifier.add-simps @{thms uint-arith-simps}
  |> fold Splitter.add-split @{thms if-split-asm}
  |> fold Simplifier.add-cong @{thms power-False-cong}
  |> Simplifier.simpset-of;

fun uint-arith-tacs ctxt =
  let
    fun arith-tac' n t =
      Arith-Data.arith-tac ctxt n t
        handle Cooper.COOPER - => Seq.empty;
  in
    [ clarify-tac ctxt 1,
      full-simp-tac (put-simpset uint-arith-simpset ctxt) 1,
      ALLGOALS (full-simp-tac
```

```
      (put-simpset HOL-ss ctxt
        |> fold Splitter.add-split @{thms uint-splits}
        |> fold Simplifier.add-cong @{thms power-False-cong})),
    rewrite-goals-tac ctxt @{thms word-size},
    ALLGOALS (fn n => REPEAT (resolve-tac ctxt [allI, impI] n) THEN
                  REPEAT (eresolve-tac ctxt [conjE] n) THEN
                  REPEAT (dresolve-tac ctxt @{thms word-of-int-inverse} n
                        THEN assume-tac ctxt n
                        THEN assume-tac ctxt n)),
    TRYALL arith-tac' ]
  end

fun uint-arith-tac ctxt = SELECT-GOAL (EVERY (uint-arith-tacs ctxt))
›

method-setup uint-arith =
  ‹Scan.succeed (SIMPLE-METHOD' o uint-arith-tac)›
  solving word arithmetic via integers and arith
```

108.21 More on overflows and monotonicity

lemma *no-plus-overflow-uint-size*: $x \leq x + y \longleftrightarrow uint\ x + uint\ y < 2\ \hat{}\ size\ x$
  for $x\ y :: 'a::len\ word$
  by (*auto simp*: *word-size word-le-def uint-add-lem uint-sub-lem*)

lemmas *no-olen-add* = *no-plus-overflow-uint-size* [*unfolded word-size*]

lemma *no-ulen-sub*: $x \geq x - y \longleftrightarrow uint\ y \leq uint\ x$
  for $x\ y :: 'a::len\ word$
  by (*auto simp*: *word-size word-le-def uint-add-lem uint-sub-lem*)

lemma *no-olen-add'*: $x \leq y + x \longleftrightarrow uint\ y + uint\ x < 2\ \hat{}\ LENGTH('a)$
  for $x\ y :: 'a::len\ word$
  by (*simp add*: *ac-simps no-olen-add*)

lemmas *olen-add-eqv* = *trans* [*OF no-olen-add no-olen-add'* [*symmetric*]]

lemmas *uint-plus-simple-iff* = *trans* [*OF no-olen-add uint-add-lem*]
lemmas *uint-plus-simple* = *uint-plus-simple-iff* [*THEN iffD1*]
lemmas *uint-minus-simple-iff* = *trans* [*OF no-ulen-sub uint-sub-lem*]
lemmas *uint-minus-simple-alt* = *uint-sub-lem* [*folded word-le-def*]
lemmas *word-sub-le-iff* = *no-ulen-sub* [*folded word-le-def*]
lemmas *word-sub-le* = *word-sub-le-iff* [*THEN iffD2*]

lemma *word-less-sub1*: $x \neq 0 \Longrightarrow 1 < x \longleftrightarrow 0 < x - 1$
  for $x :: 'a::len\ word$
  by *transfer* (*simp add*: *take-bit-decr-eq*)

lemma *word-le-sub1*: $x \neq 0 \Longrightarrow 1 \leq x \longleftrightarrow 0 \leq x - 1$

for $x :: 'a::len\ word$
by *transfer* (*simp add*: *int-one-le-iff-zero-less less-le*)

lemma *sub-wrap-lt*: $x < x - z \longleftrightarrow x < z$
for $x\ z :: 'a::len\ word$
by (*meson linorder-not-le word-sub-le-iff*)

lemma *sub-wrap*: $x \leq x - z \longleftrightarrow z = 0 \vee x < z$
for $x\ z :: 'a::len\ word$
by (*simp add*: *le-less sub-wrap-lt ac-simps*)

lemma *plus-minus-not-NULL-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow c \neq 0 \Longrightarrow x + c \neq 0$
for $x\ ab\ c :: 'a::len\ word$
by *uint-arith*

lemma *plus-minus-no-overflow-ab*: $x \leq ab - c \Longrightarrow c \leq ab \Longrightarrow x \leq x + c$
for $x\ ab\ c :: 'a::len\ word$
by *uint-arith*

lemma *le-minus'*: $a + c \leq b \Longrightarrow a \leq a + c \Longrightarrow c \leq b - a$
for $a\ b\ c :: 'a::len\ word$
by *uint-arith*

lemma *le-plus'*: $a \leq b \Longrightarrow c \leq b - a \Longrightarrow a + c \leq b$
for $a\ b\ c :: 'a::len\ word$
by *uint-arith*

lemmas *le-plus* = *le-plus'* [*rotated*]

lemmas *le-minus* = *leD* [*THEN thin-rl*, *THEN le-minus'*]

lemma *word-plus-mono-right*: $y \leq z \Longrightarrow x \leq x + z \Longrightarrow x + y \leq x + z$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-cancel*: $y - x < z - x \Longrightarrow x \leq z \Longrightarrow y < z$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-mono-left*: $y < z \Longrightarrow x \leq y \Longrightarrow y - x < z - x$
for $x\ y\ z :: 'a::len\ word$
by *uint-arith*

lemma *word-less-minus-mono*: $a < c \Longrightarrow d < b \Longrightarrow a - b < a \Longrightarrow c - d < c \Longrightarrow a - b < c - d$
for $a\ b\ c\ d :: 'a::len\ word$
by *uint-arith*

lemma *word-le-minus-cancel*: $y - x \leq z - x \Longrightarrow x \leq z \Longrightarrow y \leq z$
for *x y z* :: *'a*::*len word*
by *uint-arith*

lemma *word-le-minus-mono-left*: $y \leq z \Longrightarrow x \leq y \Longrightarrow y - x \leq z - x$
for *x y z* :: *'a*::*len word*
by *uint-arith*

lemma *word-le-minus-mono*:
$a \leq c \Longrightarrow d \leq b \Longrightarrow a - b \leq a \Longrightarrow c - d \leq c \Longrightarrow a - b \leq c - d$
for *a b c d* :: *'a*::*len word*
by *uint-arith*

lemma *plus-le-left-cancel-wrap*: $x + y' < x \Longrightarrow x + y < x \Longrightarrow x + y' < x + y \longleftrightarrow y' < y$
for *x y y'* :: *'a*::*len word*
by *uint-arith*

lemma *plus-le-left-cancel-nowrap*: $x \leq x + y' \Longrightarrow x \leq x + y \Longrightarrow x + y' < x + y \longleftrightarrow y' < y$
for *x y y'* :: *'a*::*len word*
by *uint-arith*

lemma *word-plus-mono-right2*: $a \leq a + b \Longrightarrow c \leq b \Longrightarrow a \leq a + c$
for *a b c* :: *'a*::*len word*
by *uint-arith*

lemma *word-less-add-right*: $x < y - z \Longrightarrow z \leq y \Longrightarrow x + z < y$
for *x y z* :: *'a*::*len word*
by *uint-arith*

lemma *word-less-sub-right*: $x < y + z \Longrightarrow y \leq x \Longrightarrow x - y < z$
for *x y z* :: *'a*::*len word*
by *uint-arith*

lemma *word-le-plus-either*: $x \leq y \vee x \leq z \Longrightarrow y \leq y + z \Longrightarrow x \leq y + z$
for *x y z* :: *'a*::*len word*
by *uint-arith*

lemma *word-less-nowrapI*: $x < z - k \Longrightarrow k \leq z \Longrightarrow 0 < k \Longrightarrow x < x + k$
for *x z k* :: *'a*::*len word*
by *uint-arith*

lemma *inc-le*: $i < m \Longrightarrow i + 1 \leq m$
for *i m* :: *'a*::*len word*
by *uint-arith*

lemma *less-imp-less-eq-dec*:
‹*v* ≤ *w* − *1*› **if** ‹*v* < *w*› **for** *v w* :: ‹*'a*::*len word*›

using *that* **proof** *transfer*
show ‹*take-bit LENGTH('a) k* ≤ *take-bit LENGTH('a) (l − 1)*›
if ‹*take-bit LENGTH('a) k* < *take-bit LENGTH('a) l*›
for *k l* :: *int*
using *that* **by** (*cases* ‹*take-bit LENGTH('a) l* = *0*›)
(*auto simp add*: *take-bit-decr-eq*)
qed

lemma *inc-less-eq-triv-imp*:
‹*w* = *− 1*› **if** ‹*w* + *1* ≤ *w*› **for** *w* :: ‹*'a*::*len word*›
proof (*rule ccontr*)
assume ‹*w* ≠ *− 1*›
with *that* **show** *False*
by *transfer* (*auto simp add*: *take-bit-eq-mask-iff dest*: *take-bit-decr-eq*)
qed

lemma *less-eq-dec-triv-imp*:
‹*w* = *0*› **if** ‹*w* ≤ *w* − *1*› **for** *w* :: ‹*'a*::*len word*›
proof (*rule ccontr*)
assume ‹*w* ≠ *0*›
with *that* **show** *False*
by *transfer* (*auto simp add*: *take-bit-eq-mask-iff dest*: *take-bit-decr-eq*)
qed

lemma *inc-less-eq-iff*:
‹*v* + *1* ≤ *w* ⟷ *v* = *− 1* ∨ *v* < *w*› **for** *v w* :: ‹*'a*::*len word*›
by (*auto intro*: *inc-less-eq-triv-imp inc-le*)

lemma *less-eq-dec-iff*:
‹*v* ≤ *w* − *1* ⟷ *w* = *0* ∨ *v* < *w*› **for** *v w* :: ‹*'a*::*len word*›
by (*auto intro*: *less-eq-dec-triv-imp less-imp-less-eq-dec*)

lemma *inc-i*: *1* ≤ *i* ⟹ *i* < *m* ⟹ *1* ≤ *i* + *1* ∧ *i* + *1* ≤ *m*
for *i m* :: *'a*::*len word*
by *uint-arith*

lemma *dec-less-imp-less-eq*:
‹*v* ≤ *w*› **if** ‹*v* − *1* < *w*› **for** *v w* :: ‹*'a*::*len word*›
using *that inc-le* [*of* ‹*v* − *1*› *w*] **by** *simp*

lemma *less-inc-imp-less-eq*:
‹*v* ≤ *w*› **if** ‹*v* < *w* + *1*› **for** *v w* :: ‹*'a*::*len word*›
using *that less-imp-less-eq-dec* [*of v* ‹*w* + *1*›] **by** *simp*

lemma *less-eq-dec-self-iff-eq*:
‹*w* ≤ *w* − *1* ⟷ *w* = *0*› **for** *w* :: ‹*'a*::*len word*›
using *less-eq-dec-iff* [*of w w*] **by** *simp*

lemma *inc-less-eq-self-iff-eq*:

‹$w + 1 \leq w \longleftrightarrow w = -1$› **for** w :: ‹$'a$::*len word*›
using *inc-less-eq-triv-imp* [*of w*] **by** *auto*

lemma *udvd-incr-lem*:
⟦$up < uq$; $up = ua + n * uint\ K$; $uq = ua + n' * uint\ K$⟧
$\Longrightarrow up + uint\ K \leq uq$
by *auto* (*metis int-distrib*(*1*) *linorder-not-less mult.left-neutral mult-right-mono uint-nonnegative zless-imp-add1-zle*)

lemma *udvd-incr'*:
$p < q \Longrightarrow uint\ p = ua + n * uint\ K \Longrightarrow$
$uint\ q = ua + n' * uint\ K \Longrightarrow p + K \leq q$
unfolding *word-less-alt word-le-def*
by (*metis* (*full-types*) *order-trans udvd-incr-lem uint-add-le*)

lemma *udvd-decr'*:
assumes $p < q$ *uint* $p = ua + n * uint\ K$ *uint* $q = ua + n' * uint\ K$
shows *uint* $q = ua + n' * uint\ K \Longrightarrow p \leq q - K$
proof −
have $\bigwedge w\ v.\ uint\ (w::'a\ word) \leq uint\ v + uint\ (w - v)$
by (*metis* (*no-types*) *add-diff-cancel-left' diff-add-cancel uint-add-le*)
moreover have $uint\ K + uint\ p \leq uint\ q$
using *assms* **by** (*metis* (*no-types*) *add-diff-cancel-left' diff-add-cancel udvd-incr-lem word-less-def*)
ultimately show *?thesis*
by (*meson add-le-cancel-left order-trans word-less-eq-iff-unsigned*)
qed

lemmas *udvd-incr-lem0* = *udvd-incr-lem* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-incr0* = *udvd-incr'* [**where** *ua=0*, *unfolded add-0-left*]
lemmas *udvd-decr0* = *udvd-decr'* [**where** *ua=0*, *unfolded add-0-left*]

lemma *udvd-minus-le'*: $xy < k \Longrightarrow z\ udvd\ xy \Longrightarrow z\ udvd\ k \Longrightarrow xy \leq k - z$
unfolding *udvd-unfold-int*
by (*meson udvd-decr0*)

lemma *udvd-incr2-K*:
$p < a + s \Longrightarrow a \leq a + s \Longrightarrow K\ udvd\ s \Longrightarrow K\ udvd\ p - a \Longrightarrow a \leq p \Longrightarrow$
$0 < K \Longrightarrow p \leq p + K \wedge p + K \leq a + s$
unfolding *udvd-unfold-int*
by (*smt* (*verit*, *best*) *diff-add-cancel leD udvd-incr-lem uint-plus-if' word-less-eq-iff-unsigned word-sub-le*)

108.22 Arithmetic type class instantiations

lemmas *word-le-0-iff* [*simp*] =
word-zero-le [*THEN leD*, *THEN antisym-conv1*]

lemma *word-of-int-nat*: $0 \leq x \Longrightarrow word\text{-}of\text{-}int\ x = of\text{-}nat\ (nat\ x)$

by *simp*

note that *iszero-def* is only for class *comm-semiring-1-cancel*, which requires word length $\geq$ *1*, ie *'a::len word*

lemma *iszero-word-no* [*simp*]:
iszero (*numeral bin* :: *'a::len word*) =
iszero (*take-bit LENGTH('a)* (*numeral bin* :: *int*))
by (*metis iszero-def uint-0-iff uint-bintrunc*)

Use *iszero* to simplify equalities between word numerals.

lemmas *word-eq-numeral-iff-iszero* [*simp*] =
eq-numeral-iff-iszero [**where** *'a='a::len word*]

lemma *word-less-eq-imp-half-less-eq*:
‹*v div 2* $\leq$ *w div 2*› **if** ‹*v* $\leq$ *w*› **for** *v w* :: ‹*'a::len word*›
using *that* **by** (*simp add*: *word-le-nat-alt unat-div div-le-mono*)

lemma *word-half-less-imp-less-eq*:
‹*v* $\leq$ *w*› **if** ‹*v div 2* < *w div 2*› **for** *v w* :: ‹*'a::len word*›
using *that linorder-linear word-less-eq-imp-half-less-eq* **by** *fastforce*

108.23 Word and nat

lemma *word-nchotomy*: $\forall$ *w* :: *'a::len word*. $\exists$ *n*. *w* = *of-nat n* $\wedge$ *n* < *2* ^ *LENGTH('a)*
by (*metis of-nat-unat ucast-id unsigned-less*)

lemma *of-nat-eq*: *of-nat n* = *w* $\longleftrightarrow$ ($\exists$ *q*. *n* = *unat w* + *q* * *2* ^ *LENGTH('a)*)
for *w* :: *'a::len word*
using *mod-div-mult-eq* [*of n 2* ^ *LENGTH('a)*, *symmetric*]
by (*auto simp flip*: *take-bit-eq-mod simp add*: *unsigned-of-nat*)

lemma *of-nat-eq-size*: *of-nat n* = *w* $\longleftrightarrow$ ($\exists$ *q*. *n* = *unat w* + *q* * *2* ^ *size w*)
unfolding *word-size* **by** (*rule of-nat-eq*)

lemma *of-nat-0*: *of-nat m* = (*0::'a::len word*) $\longleftrightarrow$ ($\exists$ *q*. *m* = *q* * *2* ^ *LENGTH('a)*)
by (*simp add*: *of-nat-eq*)

lemma *of-nat-2p* [*simp*]: *of-nat* (*2* ^ *LENGTH('a)*) = (*0::'a::len word*)
by (*fact mult-1* [*symmetric*, *THEN iffD2* [*OF of-nat-0 exI*]])

lemma *of-nat-gt-0*: *of-nat k* $\neq$ *0* $\Longrightarrow$ *0* < *k*
by (*cases k*) *auto*

lemma *of-nat-neq-0*: *0* < *k* $\Longrightarrow$ *k* < *2* ^ *LENGTH('a::len)* $\Longrightarrow$ *of-nat k* $\neq$ (*0* :: *'a word*)
by (*auto simp* : *of-nat-0*)

lemma *Abs-fnat-hom-add*: *of-nat a* + *of-nat b* = *of-nat* (*a* + *b*)
by *simp*

lemma *Abs-fnat-hom-mult*: *of-nat a* ∗ *of-nat b* = (*of-nat* (*a* ∗ *b*) :: *'a*::*len word*)
by (*simp add*: *wi-hom-mult*)

lemma *Abs-fnat-hom-Suc*: *word-succ* (*of-nat a*) = *of-nat* (*Suc a*)
by *transfer* (*simp add*: *ac-simps*)

lemma *Abs-fnat-hom-0*: (*0*::*'a*::*len word*) = *of-nat 0*
by *simp*

lemma *Abs-fnat-hom-1*: (*1*::*'a*::*len word*) = *of-nat* (*Suc 0*)
by *simp*

lemmas *Abs-fnat-homs* =
Abs-fnat-hom-add Abs-fnat-hom-mult Abs-fnat-hom-Suc
Abs-fnat-hom-0 Abs-fnat-hom-1

lemma *word-arith-nat-add*: *a* + *b* = *of-nat* (*unat a* + *unat b*)
by *simp*

lemma *word-arith-nat-mult*: *a* ∗ *b* = *of-nat* (*unat a* ∗ *unat b*)
by *simp*

lemma *word-arith-nat-Suc*: *word-succ a* = *of-nat* (*Suc* (*unat a*))
by (*subst Abs-fnat-hom-Suc* [*symmetric*]) *simp*

lemma *word-arith-nat-div*: *a div b* = *of-nat* (*unat a div unat b*)
by (*metis of-int-of-nat-eq of-nat-unat of-nat-div word-div-def*)

lemma *word-arith-nat-mod*: *a mod b* = *of-nat* (*unat a mod unat b*)
by (*metis of-int-of-nat-eq of-nat-mod of-nat-unat word-mod-def*)

the standard theorems for addition, subtraction and multiplication just work for words

lemma *word-of-nat-div*:
assumes *m*: ‹*m* < *2*^ *LENGTH*(*'a*::*len*)› **and** *n*: ‹*n* < *2*^ *LENGTH*(*'a*)›
shows ‹(*of-nat* (*m div n*)::*'a word*) = *of-nat m div of-nat n*›
proof (*cases* ‹*n*=*0*›)
case *False*
show *?thesis*
using *m*
proof (*induction m rule*: *less-induct*)
case (*less m*)
then show *?case*
by (*simp add*: *n of-nat-inverse word-arith-nat-div*)
qed
qed *auto*

lemmas *word-arith-nat-defs* =

word-arith-nat-add word-arith-nat-mult
word-arith-nat-Suc Abs-fnat-hom-0
Abs-fnat-hom-1 word-arith-nat-div
word-arith-nat-mod

lemma *unat-of-nat*:
‹*unat* (*word-of-nat x* :: *'a*::*len word*) = *x mod 2* ^ *LENGTH*(*'a*)›
by *transfer* (*simp flip*: *take-bit-eq-mod add*: *nat-take-bit-eq*)

lemma *unat-cong*: *x* = *y* ⟹ *unat x* = *unat y*
by (*fact arg-cong*)

lemmas *unat-word-ariths* = *word-arith-nat-defs*
[*THEN trans* [*OF unat-cong unat-of-nat*]]

lemmas *word-sub-less-iff* = *word-sub-le-iff*
[*unfolded linorder-not-less* [*symmetric*] *Not-eq-iff*]

lemma *unat-add-lem*:
unat x + *unat y* < *2* ^ *LENGTH*(*'a*) ⟷ *unat* (*x* + *y*) = *unat x* + *unat y*
for *x y* :: *'a*::*len word*
by (*metis mod-less unat-word-ariths*(*1*) *unsigned-less*)

lemma *unat-mult-lem*:
unat x ∗ *unat y* < *2* ^ *LENGTH*(*'a*) ⟷ *unat* (*x* ∗ *y*) = *unat x* ∗ *unat y*
for *x y* :: *'a*::*len word*
by (*metis mod-less unat-word-ariths*(*2*) *unsigned-less*)

lemma *le-no-overflow*: *x* ≤ *b* ⟹ *a* ≤ *a* + *b* ⟹ *x* ≤ *a* + *b*
for *a b x* :: *'a*::*len word*
using *word-le-plus-either* **by** *blast*

lemma *uint-div*:
‹*uint* (*x div y*) = *uint x div uint y*›
by (*fact uint-div-distrib*)

lemma *uint-mod*:
‹*uint* (*x mod y*) = *uint x mod uint y*›
by (*fact uint-mod-distrib*)

lemma *no-plus-overflow-unat-size*: *x* ≤ *x* + *y* ⟷ *unat x* + *unat y* < *2* ^ *size x*
for *x y* :: *'a*::*len word*
unfolding *word-size* **by** *unat-arith*

lemmas *no-olen-add-nat* =
no-plus-overflow-unat-size [*unfolded word-size*]

lemmas *unat-plus-simple* =
trans [*OF no-olen-add-nat unat-add-lem*]

lemma *word-div-mult*: $[\![0 < y;\ unat\ x * unat\ y < 2 \hat{}\ LENGTH('a)]\!] \Longrightarrow x * y\ div\ y = x$
for *x y* :: *'a*::*len word*
by (*simp add*: *unat-eq-zero unat-mult-lem word-arith-nat-div*)

lemma *div-lt'*: $i \leq k\ div\ x \Longrightarrow unat\ i * unat\ x < 2 \hat{}\ LENGTH('a)$
for *i k x* :: *'a*::*len word*
by *unat-arith* (*meson le-less-trans less-mult-imp-div-less not-le unsigned-less*)

lemmas *div-lt''* = *order-less-imp-le* [*THEN div-lt'*]

lemma *div-lt-mult*: $[\![i < k\ div\ x;\ 0 < x]\!] \Longrightarrow i * x < k$
for *i k x* :: *'a*::*len word*
by (*metis div-le-mono div-lt'' not-le unat-div word-div-mult word-less-iff-unsigned*)

lemma *div-le-mult*: $[\![i \leq k\ div\ x;\ 0 < x]\!] \Longrightarrow i * x \leq k$
for *i k x* :: *'a*::*len word*
by (*metis div-lt' less-mult-imp-div-less not-less unat-arith-simps*(*2*) *unat-div unat-mult-lem*)

lemma *div-lt-uint'*: $i \leq k\ div\ x \Longrightarrow uint\ i * uint\ x < 2 \hat{}\ LENGTH('a)$
for *i k x* :: *'a*::*len word*
unfolding *uint-nat*
by (*metis div-lt' int-ops*(*7*) *of-nat-unat uint-mult-lem unat-mult-lem*)

lemmas *div-lt-uint''* = *order-less-imp-le* [*THEN div-lt-uint'*]

lemma *word-le-exists'*: $x \leq y \Longrightarrow \exists z.\ y = x + z \wedge uint\ x + uint\ z < 2 \hat{}\ LENGTH('a)$
for *x y z* :: *'a*::*len word*
by (*metis add.commute diff-add-cancel no-olen-add*)

lemmas *plus-minus-not-NULL* = *order-less-imp-le* [*THEN plus-minus-not-NULL-ab*]

lemmas *plus-minus-no-overflow* =
order-less-imp-le [*THEN plus-minus-no-overflow-ab*]

lemmas *mcs* = *word-less-minus-cancel word-less-minus-mono-left*
word-le-minus-cancel word-le-minus-mono-left

lemmas *word-l-diffs* = *mcs* [**where** *y* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-diff-ls* = *mcs* [**where** *z* = *w* + *x*, *unfolded add-diff-cancel*] **for** *w x*
lemmas *word-plus-mcs* = *word-diff-ls* [**where** *y* = *v* + *x*, *unfolded add-diff-cancel*]
for *v x*

lemma *le-unat-uoi*:
‹*y* ≤ *unat z* ⟹ *unat* (*word-of-nat y* :: *'a word*) = *y*›
for *z* :: ‹*'a*::*len word*›
by *transfer* (*simp add*: *nat-take-bit-eq take-bit-nat-eq-self-iff le-less-trans*)

lemmas *thd = times-div-less-eq-dividend*

lemmas *uno-simps* [*THEN le-unat-uoi*] = *mod-le-divisor div-le-dividend*

lemma *word-mod-div-equality*: (*n div b*) * *b* + (*n mod b*) = *n*
for *n b* :: *'a*::*len word*
by (*fact div-mult-mod-eq*)

lemma *word-div-mult-le*: *a div b* * *b* ≤ *a*
for *a b* :: *'a*::*len word*
by (*metis div-le-mult mult-not-zero order.not-eq-order-implies-strict order-refl word-zero-le*)

lemma *word-mod-less-divisor*: *0* < *n* ⟹ *m mod n* < *n*
for *m n* :: *'a*::*len word*
by (*simp add*: *unat-arith-simps*)

lemma *word-of-int-power-hom*: *word-of-int a* ^ *n* = (*word-of-int* (*a* ^ *n*) :: *'a*::*len word*)
by (*induct n*) (*simp-all add*: *wi-hom-mult* [*symmetric*])

lemma *word-arith-power-alt*: *a* ^ *n* = (*word-of-int* (*uint a* ^ *n*) :: *'a*::*len word*)
by (*simp add* : *word-of-int-power-hom* [*symmetric*])

lemma *unatSuc*: *1* + *n* ≠ *0* ⟹ *unat* (*1* + *n*) = *Suc* (*unat n*)
for *n* :: *'a*::*len word*
by *unat-arith*

108.24 Cardinality, finiteness of set of words

lemma *inj-on-word-of-int*: ‹*inj-on* (*word-of-int* :: *int* ⇒ *'a word*) {*0*..<*2* ^ *LENGTH*(*'a*::*len*)}›
unfolding *inj-on-def*
by (*metis atLeastLessThan-iff word-of-int-inverse*)

lemma *range-uint*: ‹*range* (*uint* :: *'a word* ⇒ *int*) = {*0*..<*2* ^ *LENGTH*(*'a*::*len*)}›
apply *transfer*
apply (*auto simp*: *image-iff*)
apply (*metis take-bit-int-eq-self-iff*)
done

lemma *UNIV-eq*: ‹(*UNIV* :: *'a word set*) = *word-of-int* ' {*0*..<*2* ^ *LENGTH*(*'a*::*len*)}›
by (*auto simp*: *image-iff*) (*metis atLeastLessThan-iff linorder-not-le uint-split*)

lemma *card-word*: *CARD*(*'a word*) = *2* ^ *LENGTH*(*'a*::*len*)
by (*simp add*: *UNIV-eq card-image inj-on-word-of-int*)

lemma *card-word-size*: *CARD*(*'a word*) = *2* ^ *size x*
for *x* :: *'a*::*len word*

unfolding *word-size* **by** (*rule card-word*)

end

instance *word* :: (*len*) *finite*
by *standard* (*simp add*: *UNIV-eq*)

108.25 Bitwise Operations on Words

context
includes *bit-operations-syntax*
begin

lemma *word-wi-log-defs*:
NOT (*word-of-int a*) = *word-of-int* (*NOT a*)
word-of-int a AND word-of-int b = *word-of-int* (*a AND b*)
word-of-int a OR word-of-int b = *word-of-int* (*a OR b*)
word-of-int a XOR word-of-int b = *word-of-int* (*a XOR b*)
by (*transfer*, *rule refl*)+

lemma *word-no-log-defs* [*simp*]:
NOT (*numeral a*) = *word-of-int* (*NOT* (*numeral a*))
NOT (− *numeral a*) = *word-of-int* (*NOT* (− *numeral a*))
numeral a AND numeral b = *word-of-int* (*numeral a AND numeral b*)
numeral a AND − *numeral b* = *word-of-int* (*numeral a AND* − *numeral b*)
− *numeral a AND numeral b* = *word-of-int* (− *numeral a AND numeral b*)
− *numeral a AND* − *numeral b* = *word-of-int* (− *numeral a AND* − *numeral b*)
numeral a OR numeral b = *word-of-int* (*numeral a OR numeral b*)
numeral a OR − *numeral b* = *word-of-int* (*numeral a OR* − *numeral b*)
− *numeral a OR numeral b* = *word-of-int* (− *numeral a OR numeral b*)
− *numeral a OR* − *numeral b* = *word-of-int* (− *numeral a OR* − *numeral b*)
numeral a XOR numeral b = *word-of-int* (*numeral a XOR numeral b*)
numeral a XOR − *numeral b* = *word-of-int* (*numeral a XOR* − *numeral b*)
− *numeral a XOR numeral b* = *word-of-int* (− *numeral a XOR numeral b*)
− *numeral a XOR* − *numeral b* = *word-of-int* (− *numeral a XOR* − *numeral b*)
by (*transfer*, *rule refl*)+

Special cases for when one of the arguments equals 1.

lemma *word-bitwise-1-simps* [*simp*]:
NOT (*1*::*'a*::*len word*) = −*2*
1 AND numeral b = *word-of-int* (*1 AND numeral b*)
1 AND − *numeral b* = *word-of-int* (*1 AND* − *numeral b*)
numeral a AND 1 = *word-of-int* (*numeral a AND 1*)
− *numeral a AND 1* = *word-of-int* (− *numeral a AND 1*)
1 OR numeral b = *word-of-int* (*1 OR numeral b*)
1 OR − *numeral b* = *word-of-int* (*1 OR* − *numeral b*)
numeral a OR 1 = *word-of-int* (*numeral a OR 1*)
− *numeral a OR 1* = *word-of-int* (− *numeral a OR 1*)
1 XOR numeral b = *word-of-int* (*1 XOR numeral b*)
1 XOR − *numeral b* = *word-of-int* (*1 XOR* − *numeral b*)

numeral a XOR 1 = word-of-int (numeral a XOR 1)
− numeral a XOR 1 = word-of-int (− numeral a XOR 1)
apply (*simp-all add*: *word-uint-eq-iff unsigned-not-eq unsigned-and-eq unsigned-or-eq*
unsigned-xor-eq of-nat-take-bit ac-simps unsigned-of-int)
apply (*simp-all add*: *minus-numeral-eq-not-sub-one*)
apply (*simp-all only*: *sub-one-eq-not-neg bit.xor-compl-right take-bit-xor bit.double-compl*)
apply *simp-all*
done

Special cases for when one of the arguments equals -1.

lemma *word-bitwise-m1-simps* [*simp*]:
NOT (−1::'a::len word) = 0
(−1::'a::len word) AND x = x
x AND (−1::'a::len word) = x
(−1::'a::len word) OR x = −1
x OR (−1::'a::len word) = −1
(−1::'a::len word) XOR x = NOT x
x XOR (−1::'a::len word) = NOT x
by (*transfer*, *simp*)+

lemma *word-of-int-not-numeral-eq* [*simp*]:
‹(*word-of-int* (*NOT* (*numeral bin*)) :: *'a::len word*) = *− numeral bin − 1*›
by *transfer* (*simp add*: *not-eq-complement*)

lemma *uint-and*:
‹*uint* (*x AND y*) = *uint x AND uint y*›
by *transfer simp*

lemma *uint-or*:
‹*uint* (*x OR y*) = *uint x OR uint y*›
by *transfer simp*

lemma *uint-xor*:
‹*uint* (*x XOR y*) = *uint x XOR uint y*›
by *transfer simp*

— get from commutativity, associativity etc of *int-and* etc to same for *word-and etc*
lemmas *bwsimps* =
wi-hom-add
word-wi-log-defs

lemma *word-bw-assocs*:
(x AND y) AND z = x AND y AND z
(x OR y) OR z = x OR y OR z
(x XOR y) XOR z = x XOR y XOR z
for *x* :: *'a::len word*
by (*fact ac-simps*)+

lemma *word-bw-comms*:
x AND y = y AND x
x OR y = y OR x
x XOR y = y XOR x
for *x* :: *'a*::*len word*
by (*fact ac-simps*)+

lemma *word-bw-lcs*:
y AND x AND z = x AND y AND z
y OR x OR z = x OR y OR z
y XOR x XOR z = x XOR y XOR z
for *x* :: *'a*::*len word*
by (*fact ac-simps*)+

lemma *word-log-esimps*:
x AND 0 = 0
x AND −1 = x
x OR 0 = x
x OR −1 = −1
x XOR 0 = x
x XOR −1 = NOT x
0 AND x = 0
−1 AND x = x
0 OR x = x
−1 OR x = −1
0 XOR x = x
−1 XOR x = NOT x
for *x* :: *'a*::*len word*
by *simp-all*

lemma *word-not-dist*:
NOT (x OR y) = NOT x AND NOT y
NOT (x AND y) = NOT x OR NOT y
for *x* :: *'a*::*len word*
by *simp-all*

lemma *word-bw-same*:
x AND x = x
x OR x = x
x XOR x = 0
for *x* :: *'a*::*len word*
by *simp-all*

lemma *word-ao-absorbs* [*simp*]:
x AND (y OR x) = x
x OR y AND x = x
x AND (x OR y) = x
y AND x OR x = x

$(y\ OR\ x)\ AND\ x = x$
$x\ OR\ x\ AND\ y = x$
$(x\ OR\ y)\ AND\ x = x$
$x\ AND\ y\ OR\ x = x$
for $x :: 'a::len\ word$
by (*auto intro*: *bit-eqI simp add*: *bit-and-iff bit-or-iff*)

lemma *word-not-not* [*simp*]: $NOT\ (NOT\ x) = x$
for $x :: 'a::len\ word$
by (*fact bit.double-compl*)

lemma *word-ao-dist*: $(x\ OR\ y)\ AND\ z = x\ AND\ z\ OR\ y\ AND\ z$
for $x :: 'a::len\ word$
by (*fact bit.conj-disj-distrib2*)

lemma *word-oa-dist*: $x\ AND\ y\ OR\ z = (x\ OR\ z)\ AND\ (y\ OR\ z)$
for $x :: 'a::len\ word$
by (*fact bit.disj-conj-distrib2*)

lemma *word-add-not* [*simp*]: $x + NOT\ x = -1$
for $x :: 'a::len\ word$
by (*simp add*: *not-eq-complement*)

lemma *word-plus-and-or* [*simp*]: $(x\ AND\ y) + (x\ OR\ y) = x + y$
for $x :: 'a::len\ word$
by *transfer* (*simp add*: *plus-and-or*)

lemma *leoa*: $w = x\ OR\ y \Longrightarrow y = w\ AND\ y$
for $x :: 'a::len\ word$
by *auto*

lemma *leao*: $w' = x'\ AND\ y' \Longrightarrow x' = x'\ OR\ w'$
for $x' :: 'a::len\ word$
by *auto*

lemma *word-ao-equiv*: $w = w\ OR\ w' \longleftrightarrow w' = w\ AND\ w'$
for $w\ w' :: 'a::len\ word$
by (*auto intro*: *leoa leao*)

lemma *le-word-or2*: $x \leq x\ OR\ y$
for $x\ y :: 'a::len\ word$
by (*simp add*: *or-greater-eq uint-or word-le-def*)

lemmas *le-word-or1* = *xtrans*(*3*) [*OF word-bw-comms* (*2*) *le-word-or2*]
lemmas *word-and-le1* = *xtrans*(*3*) [*OF word-ao-absorbs* (*4*) [*symmetric*] *le-word-or2*]
lemmas *word-and-le2* = *xtrans*(*3*) [*OF word-ao-absorbs* (*8*) [*symmetric*] *le-word-or2*]

lemma *bit-horner-sum-bit-word-iff* [*bit-simps*]:
‹*bit* (*horner-sum of-bool* (*2* :: *'a::len word*) *bs*) *n*

$\longleftrightarrow n < min\ LENGTH('a)\ (length\ bs) \wedge bs\ !\ n\rangle$
by *transfer* (*simp add*: *bit-horner-sum-bit-iff*)

definition *word-reverse* :: ‹*'a*::*len word* ⇒ *'a word*›
where ‹*word-reverse w* = *horner-sum of-bool 2* (*rev* (*map* (*bit w*) [*0*..<*LENGTH*(*'a*)]))›

lemma *bit-word-reverse-iff* [*bit-simps*]:
‹*bit* (*word-reverse w*) *n* ⟷ *n* < *LENGTH*(*'a*) ∧ *bit w* (*LENGTH*(*'a*) − *Suc n*)›
for *w* :: ‹*'a*::*len word*›
by (*cases* ‹*n* < *LENGTH*(*'a*)›)
(*simp-all add*: *word-reverse-def bit-horner-sum-bit-word-iff rev-nth*)

lemma *word-rev-rev* [*simp*] : *word-reverse* (*word-reverse w*) = *w*
by (*rule bit-word-eqI*)
(*auto simp*: *bit-word-reverse-iff bit-imp-le-length Suc-diff-Suc*)

lemma *word-rev-gal*: *word-reverse w* = *u* ⟹ *word-reverse u* = *w*
by (*metis word-rev-rev*)

lemma *word-rev-gal'*: *u* = *word-reverse w* ⟹ *w* = *word-reverse u*
by *simp*

lemma *word-eq-reverseI*:
‹*v* = *w*› **if** ‹*word-reverse v* = *word-reverse w*›
by (*metis that word-rev-rev*)

lemma *uint-2p*: (*0*::*'a*::*len word*) < *2* ^ *n* ⟹ *uint* (*2* ^ *n*::*'a*::*len word*) = *2* ^ *n*
by (*cases* ‹*n* < *LENGTH*(*'a*)›; *transfer*; *force*)

lemma *word-of-int-2p*: (*word-of-int* (*2* ^ *n*) :: *'a*::*len word*) = *2* ^ *n*
by *simp*

108.25.1 shift functions in terms of lists of bools

lemma *drop-bit-word-numeral* [*simp*]:
‹*drop-bit* (*numeral n*) (*numeral k*) =
(*word-of-int* (*drop-bit* (*numeral n*) (*take-bit LENGTH*(*'a*) (*numeral k*))) :: *'a*::*len word*)›
by *transfer simp*

lemma *drop-bit-word-Suc-numeral* [*simp*]:
‹*drop-bit* (*Suc n*) (*numeral k*) =
(*word-of-int* (*drop-bit* (*Suc n*) (*take-bit LENGTH*(*'a*) (*numeral k*))) :: *'a*::*len word*)›
by *transfer simp*

lemma *drop-bit-word-minus-numeral* [*simp*]:
‹*drop-bit* (*numeral n*) (− *numeral k*) =
(*word-of-int* (*drop-bit* (*numeral n*) (*take-bit LENGTH*(*'a*) (− *numeral k*))) ::

'a::len word)›
by *transfer simp*

lemma *drop-bit-word-Suc-minus-numeral* [*simp*]:
‹*drop-bit* (*Suc n*) (− *numeral k*) =
(*word-of-int* (*drop-bit* (*Suc n*) (*take-bit LENGTH('a)* (− *numeral k*))) :: *'a::len word*)›
by *transfer simp*

lemma *signed-drop-bit-word-numeral* [*simp*]:
‹*signed-drop-bit* (*numeral n*) (*numeral k*) =
(*word-of-int* (*drop-bit* (*numeral n*) (*signed-take-bit* (*LENGTH('a)* − *1*) (*numeral k*))) :: *'a::len word*)›
by *transfer simp*

lemma *signed-drop-bit-word-Suc-numeral* [*simp*]:
‹*signed-drop-bit* (*Suc n*) (*numeral k*) =
(*word-of-int* (*drop-bit* (*Suc n*) (*signed-take-bit* (*LENGTH('a)* − *1*) (*numeral k*))) :: *'a::len word*)›
by *transfer simp*

lemma *signed-drop-bit-word-minus-numeral* [*simp*]:
‹*signed-drop-bit* (*numeral n*) (− *numeral k*) =
(*word-of-int* (*drop-bit* (*numeral n*) (*signed-take-bit* (*LENGTH('a)* − *1*) (− *numeral k*))) :: *'a::len word*)›
by *transfer simp*

lemma *signed-drop-bit-word-Suc-minus-numeral* [*simp*]:
‹*signed-drop-bit* (*Suc n*) (− *numeral k*) =
(*word-of-int* (*drop-bit* (*Suc n*) (*signed-take-bit* (*LENGTH('a)* − *1*) (− *numeral k*))) :: *'a::len word*)›
by *transfer simp*

lemma *take-bit-word-numeral* [*simp*]:
‹*take-bit* (*numeral n*) (*numeral k*) =
(*word-of-int* (*take-bit* (*min LENGTH('a)* (*numeral n*)) (*numeral k*)) :: *'a::len word*)›
by *transfer rule*

lemma *take-bit-word-Suc-numeral* [*simp*]:
‹*take-bit* (*Suc n*) (*numeral k*) =
(*word-of-int* (*take-bit* (*min LENGTH('a)* (*Suc n*)) (*numeral k*)) :: *'a::len word*)›
by *transfer rule*

lemma *take-bit-word-minus-numeral* [*simp*]:
‹*take-bit* (*numeral n*) (− *numeral k*) =
(*word-of-int* (*take-bit* (*min LENGTH('a)* (*numeral n*)) (− *numeral k*)) :: *'a::len word*)›
by *transfer rule*

lemma *take-bit-word-Suc-minus-numeral* [*simp*]:
‹*take-bit* (*Suc n*) (− *numeral k*) =
(*word-of-int* (*take-bit* (*min LENGTH*('*a*) (*Suc n*)) (− *numeral k*)) :: '*a*::*len word*)›
by *transfer rule*

lemma *signed-take-bit-word-numeral* [*simp*]:
‹*signed-take-bit* (*numeral n*) (*numeral k*) =
(*word-of-int* (*signed-take-bit* (*numeral n*) (*take-bit LENGTH*('*a*) (*numeral k*))) :: '*a*::*len word*)›
by *transfer rule*

lemma *signed-take-bit-word-Suc-numeral* [*simp*]:
‹*signed-take-bit* (*Suc n*) (*numeral k*) =
(*word-of-int* (*signed-take-bit* (*Suc n*) (*take-bit LENGTH*('*a*) (*numeral k*))) :: '*a*::*len word*)›
by *transfer rule*

lemma *signed-take-bit-word-minus-numeral* [*simp*]:
‹*signed-take-bit* (*numeral n*) (− *numeral k*) =
(*word-of-int* (*signed-take-bit* (*numeral n*) (*take-bit LENGTH*('*a*) (− *numeral k*))) :: '*a*::*len word*)›
by *transfer rule*

lemma *signed-take-bit-word-Suc-minus-numeral* [*simp*]:
‹*signed-take-bit* (*Suc n*) (− *numeral k*) =
(*word-of-int* (*signed-take-bit* (*Suc n*) (*take-bit LENGTH*('*a*) (− *numeral k*))) :: '*a*::*len word*)›
by *transfer rule*

lemma *False-map2-or*: ⟦*set xs* ⊆ {*False*}; *length ys* = *length xs*⟧ ⟹ *map2* (∨) *xs ys* = *ys*
by (*induction xs arbitrary*: *ys*) (*auto simp*: *length-Suc-conv*)

lemma *align-lem-or*:
assumes *length xs* = *n* + *m length ys* = *n* + *m*
and *drop m xs* = *replicate n False take m ys* = *replicate m False*
shows *map2* (∨) *xs ys* = *take m xs* @ *drop m ys*
using *assms*
proof (*induction xs arbitrary*: *ys m*)
case (*Cons a xs*)
then show *?case*
by (*cases m*) (*auto simp*: *length-Suc-conv False-map2-or*)
qed *auto*

lemma *False-map2-and*: ⟦*set xs* ⊆ {*False*}; *length ys* = *length xs*⟧ ⟹ *map2* (∧) *xs ys* = *xs*
by (*induction xs arbitrary*: *ys*) (*auto simp*: *length-Suc-conv*)

lemma *align-lem-and*:
assumes *length xs = n + m length ys = n + m*
and *drop m xs = replicate n False take m ys = replicate m False*
shows *map2 (∧) xs ys = replicate (n + m) False*
using *assms*
proof (*induction xs arbitrary*: *ys m*)
case (*Cons a xs*)
then show *?case*
by (*cases m*) (*auto simp*: *length-Suc-conv set-replicate-conv-if False-map2-and*)
qed *auto*

108.25.2 Mask

lemma *minus-1-eq-mask*:
‹*− 1 = (mask LENGTH('a) :: 'a::len word)*›
by (*rule bit-eqI*) (*simp add*: *bit-exp-iff bit-mask-iff*)

lemma *mask-eq-decr-exp*:
‹*mask n = 2 ^ n − (1 :: 'a::len word)*›
by (*fact mask-eq-exp-minus-1*)

lemma *mask-Suc-rec*:
‹*mask (Suc n) = 2 * mask n + (1 :: 'a::len word)*›
by (*simp add*: *mask-eq-exp-minus-1*)

context
begin

qualified lemma *bit-mask-iff* [*bit-simps*]:
‹*bit (mask m :: 'a::len word) n ⟷ n < min LENGTH('a) m*›
by (*simp add*: *bit-mask-iff not-le*)

end

lemma *mask-bin*: *mask n = word-of-int (take-bit n (− 1))*
by *transfer simp*

lemma *and-mask-bintr*: *w AND mask n = word-of-int (take-bit n (uint w))*
by *transfer* (*simp add*: *ac-simps take-bit-eq-mask*)

lemma *and-mask-wi*: *word-of-int i AND mask n = word-of-int (take-bit n i)*
by (*simp add*: *take-bit-eq-mask of-int-and-eq of-int-mask-eq*)

lemma *and-mask-wi'*:
word-of-int i AND mask n = (word-of-int (take-bit (min LENGTH('a) n) i) :: 'a::len word)
by (*auto simp*: *and-mask-wi min-def wi-bintr*)

lemma *and-mask-no*: *numeral i AND mask n = word-of-int* (*take-bit n* (*numeral i*))
unfolding *word-numeral-alt* **by** (*rule and-mask-wi*)

lemma *and-mask-mod-2p*: *w AND mask n = word-of-int* (*uint w mod 2 ^ n*)
by (*simp only*: *and-mask-bintr take-bit-eq-mod*)

lemma *uint-mask-eq*:
‹*uint* (*mask n* :: *'a*::*len word*) = *mask* (*min LENGTH*('*a*) *n*)›
by *transfer simp*

lemma *and-mask-lt-2p*: *uint* (*w AND mask n*) < *2 ^ n*
by (*metis take-bit-eq-mask take-bit-int-less-exp unsigned-take-bit-eq*)

lemma *mask-eq-iff*: *w AND mask n = w* ⟷ *uint w < 2 ^ n*
by (*metis and-mask-bintr and-mask-lt-2p take-bit-int-eq-self take-bit-nonnegative uint-sint word-of-int-uint*)

lemma *and-mask-dvd*: *2 ^ n dvd uint w* ⟷ *w AND mask n = 0*
by (*simp flip*: *take-bit-eq-mask take-bit-eq-mod unsigned-take-bit-eq add*: *dvd-eq-mod-eq-0 uint-0-iff*)

lemma *and-mask-dvd-nat*: *2 ^ n dvd unat w* ⟷ *w AND mask n = 0*
by (*simp flip*: *take-bit-eq-mask take-bit-eq-mod unsigned-take-bit-eq add*: *dvd-eq-mod-eq-0 unat-0-iff uint-0-iff*)

lemma *word-2p-lem*: *n < size w* ⟹ *w < 2 ^ n =* (*uint w < 2 ^ n*)
for *w* :: *'a*::*len word*
by *transfer simp*

lemma *less-mask-eq*:
fixes *x* :: *'a*::*len word*
assumes *x < 2 ^ n* **shows** *x AND mask n = x*
by (*metis* (*no-types*) *assms lt2p-lem mask-eq-iff not-less word-2p-lem word-size*)

lemmas *mask-eq-iff-w2p = trans* [*OF mask-eq-iff word-2p-lem* [*symmetric*]]

lemmas *and-mask-less'* = *iffD2* [*OF word-2p-lem and-mask-lt-2p*, *simplified word-size*]

lemma *and-mask-less-size*: *n < size x* ⟹ *x AND mask n < 2 ^ n*
for *x* :: ‹*'a*::*len word*›
unfolding *word-size* **by** (*erule and-mask-less'*)

lemma *word-mod-2p-is-mask* [*OF refl*]: *c = 2 ^ n* ⟹ *c > 0* ⟹ *x mod c = x AND mask n*
for *c x* :: *'a*::*len word*
by (*auto simp*: *word-mod-def uint-2p and-mask-mod-2p*)

lemma *mask-eqs*:

(a AND mask n) + b AND mask n = a + b AND mask n
a + (b AND mask n) AND mask n = a + b AND mask n
(a AND mask n) − b AND mask n = a − b AND mask n
a − (b AND mask n) AND mask n = a − b AND mask n
a ∗ (b AND mask n) AND mask n = a ∗ b AND mask n
(b AND mask n) ∗ a AND mask n = b ∗ a AND mask n
(a AND mask n) + (b AND mask n) AND mask n = a + b AND mask n
(a AND mask n) − (b AND mask n) AND mask n = a − b AND mask n
(a AND mask n) ∗ (b AND mask n) AND mask n = a ∗ b AND mask n
− (a AND mask n) AND mask n = − a AND mask n
word-succ (a AND mask n) AND mask n = word-succ a AND mask n
word-pred (a AND mask n) AND mask n = word-pred a AND mask n
using *word-of-int-Ex* [**where** *x=a*] *word-of-int-Ex* [**where** *x=b*]
unfolding *take-bit-eq-mask* [*symmetric*]
by (*transfer*; *simp add*: *take-bit-eq-mod mod-simps*)+

lemma *mask-power-eq*: *(x AND mask n) ^ k AND mask n = x ^ k AND mask n*
for *x* :: ‹*'a*::*len word*›
using *word-of-int-Ex* [**where** *x=x*]
unfolding *take-bit-eq-mask* [*symmetric*]
by (*transfer*; *simp add*: *take-bit-eq-mod mod-simps*)+

lemma *mask-full* [*simp*]: *mask LENGTH('a) = (− 1 :: 'a*::*len word)*
by *transfer simp*

108.25.3 Slices

definition *slice1* :: ‹*nat ⇒ 'a*::*len word ⇒ 'b*::*len word*›
where ‹*slice1 n w = (if n < LENGTH('a)*
then ucast (drop-bit (LENGTH('a) − n) w)
else push-bit (n − LENGTH('a)) (ucast w))›

lemma *bit-slice1-iff* [*bit-simps*]:
‹*bit (slice1 m w :: 'b*::*len word) n ⟷ m − LENGTH('a) ≤ n ∧ n < min LENGTH('b) m*
∧ bit w (n + (LENGTH('a) − m) − (m − LENGTH('a)))›
for *w* :: ‹*'a*::*len word*›
by (*auto simp*: *slice1-def bit-ucast-iff bit-drop-bit-eq bit-push-bit-iff not-less not-le ac-simps*
dest: *bit-imp-le-length*)

definition *slice* :: ‹*nat ⇒ 'a*::*len word ⇒ 'b*::*len word*›
where ‹*slice n = slice1 (LENGTH('a) − n)*›

lemma *bit-slice-iff* [*bit-simps*]:
‹*bit (slice m w :: 'b*::*len word) n ⟷ n < min LENGTH('b) (LENGTH('a) − m) ∧ bit w (n + LENGTH('a) − (LENGTH('a) − m))*›
for *w* :: ‹*'a*::*len word*›
by (*simp add*: *slice-def word-size bit-slice1-iff*)

lemma *slice1-0* [*simp*] : *slice1 n 0 = 0*
unfolding *slice1-def* **by** *simp*

lemma *slice-0* [*simp*] : *slice n 0 = 0*
unfolding *slice-def* **by** *auto*

lemma *ucast-slice1*: *ucast w = slice1* (*size w*) *w*
unfolding *slice1-def* **by** (*simp add*: *size-word.rep-eq*)

lemma *ucast-slice*: *ucast w = slice 0 w*
by (*simp add*: *slice-def slice1-def*)

lemma *slice-id*: *slice 0 t = t*
by (*simp only*: *ucast-slice* [*symmetric*] *ucast-id*)

lemma *rev-slice1*:
‹*slice1 n* (*word-reverse w* :: *'b*::*len word*) = *word-reverse* (*slice1 k w* :: *'a*::*len word*)›
if ‹*n* + *k* = *LENGTH*(*'a*) + *LENGTH*(*'b*)›
proof (*rule bit-word-eqI*)
fix *m*
assume ∗: ‹*m* < *LENGTH*(*'a*)›
from *that* **have** ∗∗: ‹*LENGTH*(*'b*) = *n* + *k* − *LENGTH*(*'a*)›
by *simp*
show ‹*bit* (*slice1 n* (*word-reverse w* :: *'b word*) :: *'a word*) *m* ⟷ *bit* (*word-reverse* (*slice1 k w* :: *'a word*)) *m*›
unfolding *bit-slice1-iff bit-word-reverse-iff*
using ∗ ∗∗
by (*cases* ‹*n* ≤ *LENGTH*(*'a*)›; *cases* ‹*k* ≤ *LENGTH*(*'a*)›) *auto*
qed

lemma *rev-slice*:
n + *k* + *LENGTH*(*'a*::*len*) = *LENGTH*(*'b*::*len*) ⟹
slice n (*word-reverse* (*w*::*'b word*)) = *word-reverse* (*slice k w* :: *'a word*)
unfolding *slice-def word-size*
by (*simp add*: *rev-slice1*)

108.25.4 Revcast

definition *revcast* :: ‹*'a*::*len word* ⇒ *'b*::*len word*›
where ‹*revcast* = *slice1 LENGTH*(*'b*)›

lemma *bit-revcast-iff* [*bit-simps*]:
‹*bit* (*revcast w* :: *'b*::*len word*) *n* ⟷ *LENGTH*(*'b*) − *LENGTH*(*'a*) ≤ *n* ∧ *n* < *LENGTH*(*'b*)
∧ *bit w* (*n* + (*LENGTH*(*'a*) − *LENGTH*(*'b*)) − (*LENGTH*(*'b*) − *LENGTH*(*'a*)))›
for *w* :: ‹*'a*::*len word*›
by (*simp add*: *revcast-def bit-slice1-iff*)

lemma *revcast-slice1* [*OF refl*]: *rc* = *revcast w* ⟹ *slice1* (*size rc*) *w* = *rc*
by (*simp add*: *revcast-def word-size*)

lemma *revcast-rev-ucast* [*OF refl refl refl*]:
cs = [*rc*, *uc*] ⟹ *rc* = *revcast* (*word-reverse w*) ⟹ *uc* = *ucast w* ⟹
rc = *word-reverse uc*
by (*metis rev-slice1 revcast-slice1 ucast-slice1 word-size*)

lemma *revcast-ucast*: *revcast w* = *word-reverse* (*ucast* (*word-reverse w*))
using *revcast-rev-ucast* [*of word-reverse w*] **by** *simp*

lemma *ucast-revcast*: *ucast w* = *word-reverse* (*revcast* (*word-reverse w*))
by (*fact revcast-rev-ucast* [*THEN word-rev-gal'*])

lemma *ucast-rev-revcast*: *ucast* (*word-reverse w*) = *word-reverse* (*revcast w*)
by (*fact revcast-ucast* [*THEN word-rev-gal'*])

linking revcast and cast via shift

lemmas *wsst-TYs* = *source-size target-size word-size*

lemmas *sym-notr* =
not-iff [*THEN iffD2*, *THEN not-sym*, *THEN not-iff* [*THEN iffD1*]]

108.26 Split and cat

lemmas *word-split-bin'* = *word-split-def*
lemmas *word-cat-bin'* = *word-cat-eq*

— this odd result is analogous to *ucast-id*, result to the length given by the result type

lemma *word-cat-id*: *word-cat a b* = *b*
by *transfer* (*simp add*: *take-bit-concat-bit-eq*)

lemma *word-cat-split-alt*: ⟦*size w* ≤ *size u* + *size v*; *word-split w* = (*u*,*v*)⟧ ⟹ *word-cat u v* = *w*
unfolding *word-split-def*
by (*rule bit-word-eqI*) (*auto simp*: *bit-word-cat-iff not-less word-size bit-ucast-iff bit-drop-bit-eq*)

lemmas *word-cat-split-size* = *sym* [*THEN* [*2*] *word-cat-split-alt* [*symmetric*]]

108.26.1 Split and slice

lemma *split-slices*:
assumes *word-split w* = (*u*, *v*)
shows *u* = *slice* (*size v*) *w* ∧ *v* = *slice 0 w*
unfolding *word-size*
proof (*intro conjI*)

have §: $\bigwedge n.$ ⟦*ucast* (*drop-bit LENGTH*('*b*) *w*) = *u*; *LENGTH*('*c*) < *LENGTH*('*b*)⟧ ⟹ ¬ *bit u n*
by (*metis bit-take-bit-iff bit-word-of-int-iff diff-is-0-eq' drop-bit-take-bit less-imp-le less-nat-zero-code of-int-uint unsigned-drop-bit-eq*)
show *u* = *slice LENGTH*('*b*) *w*
proof (*rule bit-word-eqI*)
show *bit u n* = *bit* ((*slice LENGTH*('*b*) *w*)::'*a word*) *n* **if** *n* < *LENGTH*('*a*) **for** *n*
using *assms bit-imp-le-length*
unfolding *word-split-def bit-slice-iff*
by (*fastforce simp*: § *ac-simps word-size bit-ucast-iff bit-drop-bit-eq*)
qed
show *v* = *slice 0 w*
by (*metis Pair-inject assms ucast-slice word-split-bin'*)
qed

lemma *slice-cat1* [*OF refl*]:
⟦*wc* = *word-cat a b*; *size a* + *size b* ≤ *size wc*⟧ ⟹ *slice* (*size b*) *wc* = *a*
by (*rule bit-word-eqI*) (*auto simp*: *bit-slice-iff bit-word-cat-iff word-size*)

lemmas *slice-cat2* = *trans* [*OF slice-id word-cat-id*]

lemma *cat-slices*:
⟦*a* = *slice n c*; *b* = *slice 0 c*; *n* = *size b*; *size c* ≤ *size a* + *size b*⟧ ⟹ *word-cat a b* = *c*
by (*rule bit-word-eqI*) (*auto simp*: *bit-slice-iff bit-word-cat-iff word-size*)

lemma *word-split-cat-alt*:
assumes *w* = *word-cat u v* **and** *size*: *size u* + *size v* ≤ *size w*
shows *word-split w* = (*u*,*v*)
proof −
have *ucast* ((*drop-bit LENGTH*('*c*) (*word-cat u v*))::'*a word*) = *u ucast* ((*word-cat u v*)::'*a word*) = *v*
using *assms*
by (*auto simp*: *word-size bit-ucast-iff bit-drop-bit-eq bit-word-cat-iff intro*: *bit-eqI*)
then show *?thesis*
by (*simp add*: *assms*(*1*) *word-split-bin'*)
qed

lemma *horner-sum-uint-exp-Cons-eq*:
‹*horner-sum uint* (*2* ^ *LENGTH*('*a*)) (*w* # *ws*) =
concat-bit LENGTH('*a*) (*uint w*) (*horner-sum uint* (*2* ^ *LENGTH*('*a*)) *ws*)›
for *ws* :: ‹'*a*::*len word list*›
by (*simp add*: *bintr-uint concat-bit-eq push-bit-eq-mult*)

lemma *bit-horner-sum-uint-exp-iff*:
‹*bit* (*horner-sum uint* (*2* ^ *LENGTH*('*a*)) *ws*) *n* ⟷
n div LENGTH('*a*) < *length ws* ∧ *bit* (*ws* ! (*n div LENGTH*('*a*))) (*n mod*

LENGTH('a)›
for *ws* :: ‹*'a::len word list*›
proof (*induction ws arbitrary*: *n*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons w ws*)
then show *?case*
by (*cases* ‹*n* ≥ *LENGTH('a)*›)
(*simp-all only*: *horner-sum-uint-exp-Cons-eq*, *simp-all add*: *bit-concat-bit-iff le-div-geq le-mod-geq bit-uint-iff Cons*)
qed

108.27 Rotation

lemma *word-rotr-word-rotr-eq*: ‹*word-rotr m* (*word-rotr n w*) = *word-rotr* (*m* + *n*) *w*›
by (*rule bit-word-eqI*) (*simp add*: *bit-word-rotr-iff ac-simps mod-add-right-eq*)

lemma *word-rot-lem*: ⟦*l* + *k* = *d* + *k mod l*; *n* < *l*⟧ ⟹ ((*d* + *n*) *mod l*) = *n* **for** *l*::*nat*
by (*metis* (*no-types*, *lifting*) *add.commute add.right-neutral add-diff-cancel-left' mod-if mod-mult-div-eq mod-mult-self2 mod-self*)

lemma *word-rot-rl* [*simp*]: ‹*word-rotl k* (*word-rotr k v*) = *v*›
proof (*rule bit-word-eqI*)
show *bit* (*word-rotl k* (*word-rotr k v*)) *n* = *bit v n* **if** *n* < *LENGTH('a)* **for** *n*
using *that*
by (*auto simp*: *word-rot-lem word-rotl-eq-word-rotr word-rotr-word-rotr-eq bit-word-rotr-iff algebra-simps split*: *nat-diff-split*)
qed

lemma *word-rot-lr* [*simp*]: ‹*word-rotr k* (*word-rotl k v*) = *v*›
proof (*rule bit-word-eqI*)
show *bit* (*word-rotr k* (*word-rotl k v*)) *n* = *bit v n* **if** *n* < *LENGTH('a)* **for** *n*
using *that*
by (*auto simp*: *word-rot-lem word-rotl-eq-word-rotr word-rotr-word-rotr-eq bit-word-rotr-iff algebra-simps split*: *nat-diff-split*)
qed

lemma *word-rot-gal*:
‹*word-rotr n v* = *w* ⟷ *word-rotl n w* = *v*›
by *auto*

lemma *word-rot-gal'*:
‹*w* = *word-rotr n v* ⟷ *v* = *word-rotl n w*›
by *auto*

lemma *word-reverse-word-rotl*:
‹*word-reverse* (*word-rotl n w*) = *word-rotr n* (*word-reverse w*)› (**is** ‹*?lhs* = *?rhs*›)
proof (*rule bit-word-eqI*)
fix *m*
assume ‹*m* < *LENGTH*('*a*)›
then have ‹*int* (*LENGTH*('*a*) − *Suc* ((*m* + *n*) *mod LENGTH*('*a*))) =
int ((*LENGTH*('*a*) + *LENGTH*('*a*) − *Suc* (*m* + *n mod LENGTH*('*a*))) *mod LENGTH*('*a*))›
unfolding *of-nat-diff of-nat-mod*
apply (*simp add*: *Suc-le-eq add-less-le-mono of-nat-mod algebra-simps*)
apply (*simp only*: *mod-diff-left-eq* [*symmetric*, *of* ‹*int LENGTH*('*a*) * *2*›] *mod-mult-self1-is-0 diff-0 minus-mod-int-eq*)
apply (*simp add*: *mod-simps*)
done
then have ‹*LENGTH*('*a*) − *Suc* ((*m* + *n*) *mod LENGTH*('*a*)) =
(*LENGTH*('*a*) + *LENGTH*('*a*) − *Suc* (*m* + *n mod LENGTH*('*a*))) *mod LENGTH*('*a*)›
by *simp*
with ‹*m* < *LENGTH*('*a*)› **show** ‹*bit ?lhs m* ⟷ *bit ?rhs m*›
by (*simp add*: *bit-simps*)
qed

lemma *word-reverse-word-rotr*:
‹*word-reverse* (*word-rotr n w*) = *word-rotl n* (*word-reverse w*)›
by (*rule word-eq-reverseI*) (*simp add*: *word-reverse-word-rotl*)

lemma *word-rotl-rev*:
‹*word-rotl n w* = *word-reverse* (*word-rotr n* (*word-reverse w*))›
by (*simp add*: *word-reverse-word-rotr*)

lemma *word-rotr-rev*:
‹*word-rotr n w* = *word-reverse* (*word-rotl n* (*word-reverse w*))›
by (*simp add*: *word-reverse-word-rotl*)

lemma *word-roti-0* [*simp*]: *word-roti 0 w* = *w*
by *transfer simp*

lemma *word-roti-add*: *word-roti* (*m* + *n*) *w* = *word-roti m* (*word-roti n w*)
by (*rule bit-word-eqI*)
(*simp add*: *bit-word-roti-iff nat-less-iff mod-simps ac-simps*)

lemma *word-roti-conv-mod'*:
word-roti n w = *word-roti* (*n mod int* (*size w*)) *w*
by *transfer simp*

lemmas *word-roti-conv-mod* = *word-roti-conv-mod'* [*unfolded word-size*]

end

108.27.1 "Word rotation commutes with bit-wise operations

locale *word-rotate*
begin

context
includes *bit-operations-syntax*
begin

lemma *word-rot-logs*:
word-rotl n (NOT v) = NOT (word-rotl n v)
word-rotr n (NOT v) = NOT (word-rotr n v)
word-rotl n (x AND y) = word-rotl n x AND word-rotl n y
word-rotr n (x AND y) = word-rotr n x AND word-rotr n y
word-rotl n (x OR y) = word-rotl n x OR word-rotl n y
word-rotr n (x OR y) = word-rotr n x OR word-rotr n y
word-rotl n (x XOR y) = word-rotl n x XOR word-rotl n y
word-rotr n (x XOR y) = word-rotr n x XOR word-rotr n y
by (*rule bit-word-eqI*, *auto simp*: *bit-word-rotl-iff bit-word-rotr-iff bit-and-iff bit-or-iff bit-xor-iff bit-not-iff algebra-simps not-le*)+

end

end

lemmas *word-rot-logs = word-rotate.word-rot-logs*

lemma *word-rotx-0* [*simp*] : *word-rotr i 0 = 0 ∧ word-rotl i 0 = 0*
by *transfer simp-all*

lemma *word-roti-0'* [*simp*] : *word-roti n 0 = 0*
by *transfer simp*

declare *word-roti-eq-word-rotr-word-rotl* [*simp*]

108.28 Maximum machine word

context
includes *bit-operations-syntax*
begin

lemma *word-int-cases*:
fixes *x* :: *'a*::*len word*
obtains *n* **where** *x = word-of-int n* **and** *0 ≤ n* **and** *n < 2^LENGTH('a)*
by (*rule that* [*of* ‹*uint x*›]) *simp-all*

lemma *word-nat-cases* [*cases type*: *word*]:
fixes *x* :: *'a*::*len word*
obtains *n* **where** *x = of-nat n* **and** *n < 2^LENGTH('a)*
by (*rule that* [*of* ‹*unat x*›]) *simp-all*

lemma *max-word-max* [*intro*!]:
‹$n \leq -1$› **for** n :: ‹'*a*::*len word*›
by (*fact word-order.extremum*)

lemma *word-of-int-2p-len*: *word-of-int* ($2 \hat{} LENGTH('a)$) = (0::'*a*::*len word*)
by *simp*

lemma *word-pow-0*: (2::'*a*::*len word*) $\hat{}$ $LENGTH('a) = 0$
by (*fact word-exp-length-eq-0*)

lemma *max-word-wrap*:
‹$x + 1 = 0 \Longrightarrow x = -1$› **for** x :: ‹'*a*::*len word*›
by (*simp add*: *eq-neg-iff-add-eq-0*)

lemma *word-and-max*:
‹x *AND* $-1 = x$› **for** x :: ‹'*a*::*len word*›
by (*fact word-log-esimps*)

lemma *word-or-max*:
‹x *OR* $-1 = -1$› **for** x :: ‹'*a*::*len word*›
by (*fact word-log-esimps*)

lemma *word-ao-dist2*: x *AND* (y *OR* z) = x *AND* y *OR* x *AND* z
for x y z :: '*a*::*len word*
by (*fact bit.conj-disj-distrib*)

lemma *word-oa-dist2*: x *OR* y *AND* z = (x *OR* y) *AND* (x *OR* z)
for x y z :: '*a*::*len word*
by (*fact bit.disj-conj-distrib*)

lemma *word-and-not* [*simp*]: x *AND* *NOT* $x = 0$
for x :: '*a*::*len word*
by (*fact bit.conj-cancel-right*)

lemma *word-or-not* [*simp*]:
‹x *OR* *NOT* $x = -1$› **for** x :: ‹'*a*::*len word*›
by (*fact bit.disj-cancel-right*)

lemma *word-xor-and-or*: x *XOR* y = x *AND* *NOT* y *OR* *NOT* x *AND* y
for x y :: '*a*::*len word*
by (*fact bit.xor-def*)

lemma *uint-lt-0* [*simp*]: *uint* $x < 0$ = *False*
by (*simp add*: *linorder-not-less*)

lemma *word-less-1* [*simp*]: $x < 1 \longleftrightarrow x = 0$
for x :: '*a*::*len word*
by (*simp add*: *word-less-nat-alt unat-0-iff*)

lemma *uint-plus-if-size*:
uint (*x* + *y*) =
(*if uint x* + *uint y* < *2^size x*
then uint x + *uint y*
else uint x + *uint y* − *2^size x*)
by (*simp add*: *take-bit-eq-mod word-size uint-word-of-int-eq uint-plus-if'*)

lemma *unat-plus-if-size*:
unat (*x* + *y*) =
(*if unat x* + *unat y* < *2^size x*
then unat x + *unat y*
else unat x + *unat y* − *2^size x*)
for *x y* :: *'a*::*len word*
by (*simp add*: *size-word.rep-eq unat-arith-simps*)

lemma *word-neq-0-conv*: *w* ≠ *0* ⟷ *0* < *w*
for *w* :: *'a*::*len word*
by (*fact word-coorder.not-eq-extremum*)

lemma *max-lt*: *unat* (*max a b div c*) = *unat* (*max a b*) *div unat c*
for *c* :: *'a*::*len word*
by (*fact unat-div*)

lemma *uint-sub-if-size*:
uint (*x* − *y*) =
(*if uint y* ≤ *uint x*
then uint x − *uint y*
else uint x − *uint y* + *2^size x*)
by (*simp add*: *size-word.rep-eq uint-sub-if'*)

lemma *unat-sub*:
‹*unat* (*a* − *b*) = *unat a* − *unat b*›
if ‹*b* ≤ *a*›
by (*meson that unat-sub-if-size word-le-nat-alt*)

lemmas *word-less-sub1-numberof* [*simp*] = *word-less-sub1* [*of numeral w*] **for** *w*
lemmas *word-le-sub1-numberof* [*simp*] = *word-le-sub1* [*of numeral w*] **for** *w*

lemma *word-of-int-minus*: *word-of-int* (*2^LENGTH('a)* − *i*) = (*word-of-int* (−*i*)::*'a*::*len word*)
by *simp*

lemma *word-of-int-inj*:
‹(*word-of-int x* :: *'a*::*len word*) = *word-of-int y* ⟷ *x* = *y*›
if ‹*0* ≤ *x* ∧ *x* < *2* ^ *LENGTH('a)*› ‹*0* ≤ *y* ∧ *y* < *2* ^ *LENGTH('a)*›
using *that* **by** (*transfer fixing*: *x y*) (*simp add*: *take-bit-int-eq-self*)

lemma *word-le-less-eq*: *x* ≤ *y* ⟷ *x* = *y* ∨ *x* < *y*

```
  for x y :: 'z::len word
  by (auto simp: order-class.le-less)

lemma mod-plus-cong:
  fixes b b' :: int
  assumes 1: b = b'
    and 2: x mod b' = x' mod b'
    and 3: y mod b' = y' mod b'
    and 4: x' + y' = z'
  shows (x + y) mod b = z' mod b'
proof -
  from 1 2[symmetric] 3[symmetric]
  have (x + y) mod b = (x' mod b' + y' mod b') mod b'
    by (simp add: mod-add-eq)
  also have ... = (x' + y') mod b'
    by (simp add: mod-add-eq)
  finally show ?thesis
    by (simp add: 4)
qed

lemma mod-minus-cong:
  fixes b b' :: int
  assumes b = b'
    and x mod b' = x' mod b'
    and y mod b' = y' mod b'
    and x' - y' = z'
  shows (x - y) mod b = z' mod b'
  using assms [symmetric] by (auto intro: mod-diff-cong)

lemma word-induct-less [case-names zero less]:
  ‹P m› if zero: ‹P 0› and less: ‹⋀n. n < m ⟹ P n ⟹ P (1 + n)›
  for m :: ‹'a::len word›
proof -
  define q where ‹q = unat m›
  with less have ‹⋀n. n < word-of-nat q ⟹ P n ⟹ P (1 + n)›
    by simp
  then have ‹P (word-of-nat q :: 'a word)›
  proof (induction q)
    case 0
    show ?case
      by (simp add: zero)
  next
    case (Suc q)
    show ?case
    proof (cases ‹1 + word-of-nat q = (0 :: 'a word)›)
      case True
      then show ?thesis
        by (simp add: zero)
    next
```

case *False*
then have ∗: ‹*word-of-nat q* < (*word-of-nat* (*Suc q*) :: *'a word*)›
by (*simp add*: *unatSuc word-less-nat-alt*)
then have ∗∗: ‹*n* < (*1* + *word-of-nat q* :: *'a word*) ⟷ *n* ≤ (*word-of-nat q* :: *'a word*)› **for** *n*
by (*metis* (*no-types*, *lifting*) *add.commute inc-le le-less-trans not-less of-nat-Suc*)
have ‹*P* (*word-of-nat q*)›
by (*simp add*: ∗∗ *Suc.IH Suc.prems*)
with ∗ **have** ‹*P* (*1* + *word-of-nat q*)›
by (*rule Suc.prems*)
then show *?thesis*
by *simp*
qed
qed
with ‹*q* = *unat m*› **show** *?thesis*
by *simp*
qed

lemma *word-induct*: *P 0* ⟹ (⋀*n*. *P n* ⟹ *P* (*1* + *n*)) ⟹ *P m*
for *P* :: *'a*::*len word* ⇒ *bool*
by (*rule word-induct-less*)

lemma *word-induct2* [*case-names zero suc*, *induct type*]: *P 0* ⟹ (⋀*n*. *1* + *n* ≠ *0* ⟹ *P n* ⟹ *P* (*1* + *n*)) ⟹ *P n*
for *P* :: *'b*::*len word* ⇒ *bool*
by (*induction rule*: *word-induct-less*; *force*)

108.29 Recursion combinator for words

definition *word-rec* :: *'a* ⇒ (*'b*::*len word* ⇒ *'a* ⇒ *'a*) ⇒ *'b word* ⇒ *'a*
where *word-rec forZero forSuc n* = *rec-nat forZero* (*forSuc* ∘ *of-nat*) (*unat n*)

lemma *word-rec-0* [*simp*]: *word-rec z s 0* = *z*
by (*simp add*: *word-rec-def*)

lemma *word-rec-Suc* [*simp*]: *1* + *n* ≠ *0* ⟹ *word-rec z s* (*1* + *n*) = *s n* (*word-rec z s n*)
for *n* :: *'a*::*len word*
by (*simp add*: *unatSuc word-rec-def*)

lemma *word-rec-Pred*: *n* ≠ *0* ⟹ *word-rec z s n* = *s* (*n* − *1*) (*word-rec z s* (*n* − *1*))
by (*metis add.commute diff-add-cancel word-rec-Suc*)

lemma *word-rec-in*: *f* (*word-rec z* (λ-. *f*) *n*) = *word-rec* (*f z*) (λ-. *f*) *n*
by (*induct n*) *simp-all*

lemma *word-rec-in2*: *f n* (*word-rec z f n*) = *word-rec* (*f 0 z*) (*f* ∘ (+) *1*) *n*

by (*induct n*) *simp-all*

lemma *word-rec-twice*:
$m \leq n \Longrightarrow$ *word-rec z f n* = *word-rec* (*word-rec z f* (*n* − *m*)) (*f* ∘ (+) (*n* − *m*)) *m*
proof (*induction n arbitrary*: *z f*)
case *zero*
then show *?case*
by (*metis diff-0-right word-le-0-iff word-rec-0*)
next
case (*suc n z f*)
show *?case*
proof (*cases 1* + (*n* − *m*) = *0*)
case *True*
then show *?thesis*
by (*simp add*: *add-diff-eq*)
next
case *False*
then have *eq*: *1* + *n* − *m* = *1* + (*n* − *m*)
by *simp*
with *False* **have** *m* ≤ *n*
using *inc-le linorder-not-le suc.prems word-le-minus-mono-left* **by** *fastforce*
with *False suc.hyps* **show** *?thesis*
using *suc.IH* [*of f 0 z f* ∘ (+) *1*]
by (*simp add*: *word-rec-in2 eq add.assoc o-def*)
qed
qed

lemma *word-rec-id*: *word-rec z* (λ-. *id*) *n* = *z*
by (*induct n*) *auto*

lemma *word-rec-id-eq*: (⋀*m*. *m* < *n* ⟹ *f m* = *id*) ⟹ *word-rec z f n* = *z*
by (*induction n*) (*auto simp*: *unatSuc unat-arith-simps*(*2*))

lemma *word-rec-max*:
assumes ∀ *m*≥*n*. *m* ≠ − *1* ⟶ *f m* = *id*
shows *word-rec z f* (− *1*) = *word-rec z f n*
proof −
have §: ⋀*m*. ⟦*m* < − *1* − *n*⟧ ⟹ (*f* ∘ (+) *n*) *m* = *id*
using *assms*
by (*metis* (*mono-tags*, *lifting*) *add.commute add-diff-cancel-left' comp-apply less-le olen-add-eqv plus-minus-no-overflow word-n1-ge*)
have *word-rec z f* (− *1*) = *word-rec* (*word-rec z f* (− *1* − (− *1* − *n*))) (*f* ∘ (+) (− *1* − (− *1* − *n*))) (− *1* − *n*)
by (*meson word-n1-ge word-rec-twice*)
also have ... = *word-rec z f n*
by (*metis* (*no-types*, *lifting*) § *diff-add-cancel minus-diff-eq uminus-add-conv-diff word-rec-id-eq*)
finally show *?thesis* **.**

qed

end

108.30 Some more naive computations rules

lemma *drop-bit-of-minus-1-eq* [*simp*]:
‹*drop-bit n* (− *1* :: *'a*::*len word*) = *mask* (*LENGTH*('*a*) − *n*)›
by (*rule bit-word-eqI*) (*auto simp add*: *bit-simps*)

context
includes *bit-operations-syntax*
begin

lemma *word-cat-eq-push-bit-or*:
‹*word-cat v w* = (*push-bit LENGTH*('*b*) (*ucast v*) *OR ucast w* :: *'c*::*len word*)›
for *v* :: ‹*'a*::*len word*› **and** *w* :: ‹*'b*::*len word*›
by *transfer* (*simp add*: *concat-bit-def ac-simps*)

end

context *semiring-bit-operations*
begin

lemma *of-nat-take-bit-numeral-eq* [*simp*]:
‹*of-nat* (*take-bit m* (*numeral n*)) = *take-bit m* (*numeral n*)›
by (*simp add*: *of-nat-take-bit*)

end

context *ring-bit-operations*
begin

lemma *signed-take-bit-of-int*:
‹*signed-take-bit n* (*of-int k*) = *of-int* (*signed-take-bit n k*)›
by (*rule bit-eqI*) (*simp add*: *bit-simps*)

lemma *of-int-signed-take-bit*:
‹*of-int* (*signed-take-bit n k*) = *signed-take-bit n* (*of-int k*)›
by (*simp add*: *signed-take-bit-of-int*)

lemma *of-int-take-bit-minus-numeral-eq* [*simp*]:
‹*of-int* (*take-bit m* (*numeral n*)) = *take-bit m* (*numeral n*)›
‹*of-int* (*take-bit m* (− *numeral n*)) = *take-bit m* (− *numeral n*)›
by (*simp-all add*: *of-int-take-bit*)

end

context

includes *bit-operations-syntax*
begin

lemma *concat-bit-numeral-of-one-1* [*simp*]:
‹*concat-bit* (*numeral m*) *1 l* = *1 OR push-bit* (*numeral m*) *l*›
by (*rule bit-eqI*) (*auto simp add*: *bit-simps*)

lemma *concat-bit-of-one-2* [*simp*]:
‹*concat-bit n k 1* = *set-bit n* (*take-bit n k*)›
by (*rule bit-eqI*) (*auto simp add*: *bit-simps*)

lemma *concat-bit-numeral-of-minus-one-1* [*simp*]:
‹*concat-bit* (*numeral m*) (− *1*) *l* = *push-bit* (*numeral m*) *l OR mask* (*numeral m*)›
by (*rule bit-eqI*) (*auto simp add*: *bit-simps*)

lemma *concat-bit-numeral-of-minus-one-2* [*simp*]:
‹*concat-bit* (*numeral m*) *k* (− *1*) = *take-bit* (*numeral m*) *k OR NOT* (*mask* (*numeral m*))›
by (*rule bit-eqI*) (*auto simp add*: *bit-simps*)

lemma *concat-bit-numeral* [*simp*]:
‹*concat-bit* (*numeral m*) (*numeral n*) (*numeral q*) = *take-bit* (*numeral m*) (*numeral n*) *OR push-bit* (*numeral m*) (*numeral q*)›
‹*concat-bit* (*numeral m*) (− *numeral n*) (*numeral q*) = *take-bit* (*numeral m*) (− *numeral n*) *OR push-bit* (*numeral m*) (*numeral q*)›
‹*concat-bit* (*numeral m*) (*numeral n*) (− *numeral q*) = *take-bit* (*numeral m*) (*numeral n*) *OR push-bit* (*numeral m*) (− *numeral q*)›
‹*concat-bit* (*numeral m*) (− *numeral n*) (− *numeral q*) = *take-bit* (*numeral m*) (− *numeral n*) *OR push-bit* (*numeral m*) (− *numeral q*)›
by (*fact concat-bit-def*)+

end

lemma *word-cat-0-left* [*simp*]:
‹*word-cat 0 w* = *ucast w*›
by (*simp add*: *word-cat-eq*)

108.31 Executable intervals

instance *word* :: (*len*) ‹{*interval-top*, *interval-bot*}›
by *standard*
(*simp-all add*: *less-eq-dec-self-iff-eq inc-less-eq-self-iff-eq less-inc-imp-less-eq dec-less-imp-less-eq*)

108.32 Tool support

ML-file ‹*Tools/smt-word.ML*›

end

109 The Field of Integers mod 2

theory *Z2*
imports *Main*
begin

Note that in most cases *bool* is appropriate when a binary type is needed; the type provided here, for historical reasons named *bit*, is only needed if proper field operations are required.

typedef *bit* = ‹*UNIV* :: *bool set*› **..**

instantiation *bit* :: *zero-neq-one*
begin

definition *zero-bit* :: *bit*
  where ‹*0* = *Abs-bit False*›

definition *one-bit* :: *bit*
  where ‹*1* = *Abs-bit True*›

instance
  by *standard* (*simp add*: *zero-bit-def one-bit-def Abs-bit-inject*)

end

free-constructors *case-bit* **for** ‹*0*::*bit*› | ‹*1*::*bit*›
proof −
  fix *P* :: *bool*
  fix *a* :: *bit*
  assume ‹*a* = *0* ⟹ *P*› **and** ‹*a* = *1* ⟹ *P*›
  then show *P*
    by (*cases a*) (*auto simp add*: *zero-bit-def one-bit-def Abs-bit-inject*)
qed *simp*

lemma *bit-not-zero-iff* [*simp*]:
  ‹*a* ≠ *0* ⟷ *a* = *1*› **for** *a* :: *bit*
  by (*cases a*) *simp-all*

lemma *bit-not-one-iff* [*simp*]:
  ‹*a* ≠ *1* ⟷ *a* = *0*› **for** *a* :: *bit*
  by (*cases a*) *simp-all*

instantiation *bit* :: *semidom-modulo*
begin

definition *plus-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
  where ‹*a* + *b* = *Abs-bit* (*Rep-bit a* ≠ *Rep-bit b*)›

definition *minus-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›

where [*simp*]: ‹*minus-bit* = *plus*›

definition *times-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where ‹*a* ∗ *b* = *Abs-bit* (*Rep-bit a* ∧ *Rep-bit b*)›

definition *divide-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where [*simp*]: ‹*divide-bit* = *times*›

definition *modulo-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where ‹*a mod b* = *Abs-bit* (*Rep-bit a* ∧ ¬ *Rep-bit b*)›

instance
by *standard*
(*auto simp flip*: *Rep-bit-inject*
simp add: *zero-bit-def one-bit-def plus-bit-def times-bit-def modulo-bit-def Abs-bit-inverse Rep-bit-inverse*)

end

lemma *bit-2-eq-0* [*simp*]:
‹*2* = (*0*::*bit*)›
by (*simp flip*: *one-add-one add*: *zero-bit-def plus-bit-def*)

instance *bit* :: *semiring-parity*
apply *standard*
apply (*auto simp flip*: *Rep-bit-inject simp add*: *modulo-bit-def Abs-bit-inverse Rep-bit-inverse*)
apply (*auto simp add*: *zero-bit-def one-bit-def Abs-bit-inverse Rep-bit-inverse*)
done

lemma *Abs-bit-eq-of-bool* [*code-abbrev*]:
‹*Abs-bit* = *of-bool*›
by (*simp add*: *fun-eq-iff zero-bit-def one-bit-def*)

lemma *Rep-bit-eq-odd*:
‹*Rep-bit* = *odd*›
proof −
have ‹¬ *Rep-bit 0*›
by (*simp only*: *zero-bit-def*) (*subst Abs-bit-inverse*, *auto*)
then show *?thesis*
by (*auto simp flip*: *Rep-bit-inject simp add*: *fun-eq-iff*)
qed

lemma *Rep-bit-iff-odd* [*code-abbrev*]:
‹*Rep-bit b* ⟷ *odd b*›
by (*simp add*: *Rep-bit-eq-odd*)

lemma *Not-Rep-bit-iff-even* [*code-abbrev*]:
‹¬ *Rep-bit b* ⟷ *even b*›

by (*simp add*: *Rep-bit-eq-odd*)

lemma *Not-Not-Rep-bit* [*code-unfold*]:
‹¬ ¬ *Rep-bit b* ⟷ *Rep-bit b*›
by *simp*

code-datatype ‹*0*::*bit*› ‹*1*::*bit*›

lemma *Abs-bit-code* [*code*]:
‹*Abs-bit False* = *0*›
‹*Abs-bit True* = *1*›
by (*simp-all add*: *Abs-bit-eq-of-bool*)

lemma *Rep-bit-code* [*code*]:
‹*Rep-bit 0* ⟷ *False*›
‹*Rep-bit 1* ⟷ *True*›
by (*simp-all add*: *Rep-bit-eq-odd*)

context *zero-neq-one*
begin

abbreviation *of-bit* :: ‹*bit* ⇒ *'a*›
where ‹*of-bit b* ≡ *of-bool* (*odd b*)›

end

context
begin

qualified lemma *bit-eq-iff*:
‹*a* = *b* ⟷ (*even a* ⟷ *even b*)› **for** *a b* :: *bit*
by (*cases a*; *cases b*) *simp-all*

end

lemma *modulo-bit-unfold* [*simp*, *code*]:
‹*a mod b* = *of-bool* (*odd a* ∧ *even b*)› **for** *a b* :: *bit*
by (*simp add*: *modulo-bit-def Abs-bit-eq-of-bool Rep-bit-eq-odd*)

lemma *power-bit-unfold* [*simp*]:
‹*a* ^ *n* = *of-bool* (*odd a* ∨ *n* = *0*)› **for** *a* :: *bit*
by (*cases a*) *simp-all*

instantiation *bit* :: *field*
begin

definition *uminus-bit* :: ‹*bit* ⇒ *bit*›
where [*simp*]: ‹*uminus-bit* = *id*›

definition *inverse-bit* :: ‹*bit* ⇒ *bit*›
where [*simp*]: ‹*inverse-bit* = *id*›

instance
apply *standard*
apply *simp-all*
apply (*simp only*: *Z2.bit-eq-iff even-add even-zero refl*)
done

end

instantiation *bit* :: *semiring-bits*
begin

definition *bit-bit* :: ‹*bit* ⇒ *nat* ⇒ *bool*›
where [*simp*]: ‹*bit-bit b n* ⟷ *odd b* ∧ *n* = *0*›

instance
by *standard*
(*auto intro*: *Abs-bit-induct simp add*: *Abs-bit-eq-of-bool*)

end

instantiation *bit* :: *ring-bit-operations*
begin

context
includes *bit-operations-syntax*
begin

definition *not-bit* :: ‹*bit* ⇒ *bit*›
where [*simp*]: ‹*NOT b* = *of-bool* (*even b*)› **for** *b* :: *bit*

definition *and-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where [*simp*]: ‹*b AND c* = *of-bool* (*odd b* ∧ *odd c*)› **for** *b c* :: *bit*

definition *or-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where [*simp*]: ‹*b OR c* = *of-bool* (*odd b* ∨ *odd c*)› **for** *b c* :: *bit*

definition *xor-bit* :: ‹*bit* ⇒ *bit* ⇒ *bit*›
where [*simp*]: ‹*b XOR c* = *of-bool* (*odd b* ≠ *odd c*)› **for** *b c* :: *bit*

definition *mask-bit* :: ‹*nat* ⇒ *bit*›
where [*simp*]: ‹*mask n* = (*of-bool* (*n* > *0*) :: *bit*)›

definition *set-bit-bit* :: ‹*nat* ⇒ *bit* ⇒ *bit*›
where [*simp*]: ‹*set-bit n b* = *of-bool* (*n* = *0* ∨ *odd b*)› **for** *b* :: *bit*

definition *unset-bit-bit* :: ‹*nat* ⇒ *bit* ⇒ *bit*›

where [*simp*]: ‹*unset-bit n b = of-bool (n > 0 ∧ odd b)*› **for** *b* :: *bit*

definition *flip-bit-bit* :: ‹*nat ⇒ bit ⇒ bit*›
where [*simp*]: ‹*flip-bit n b = of-bool ((n = 0) ≠ odd b)*› **for** *b* :: *bit*

definition *push-bit-bit* :: ‹*nat ⇒ bit ⇒ bit*›
where [*simp*]: ‹*push-bit n b = of-bool (odd b ∧ n = 0)*› **for** *b* :: *bit*

definition *drop-bit-bit* :: ‹*nat ⇒ bit ⇒ bit*›
where [*simp*]: ‹*drop-bit n b = of-bool (odd b ∧ n = 0)*› **for** *b* :: *bit*

definition *take-bit-bit* :: ‹*nat ⇒ bit ⇒ bit*›
where [*simp*]: ‹*take-bit n b = of-bool (odd b ∧ n > 0)*› **for** *b* :: *bit*

end

instance
by *standard auto*

end

lemma *add-bit-eq-xor* [*simp*, *code*]:
‹(+) = (*Bit-Operations.xor* :: *bit ⇒* -)›
by (*auto simp add*: *fun-eq-iff*)

lemma *mult-bit-eq-and* [*simp*, *code*]:
‹(∗) = (*Bit-Operations.and* :: *bit ⇒* -)›
by (*simp add*: *fun-eq-iff*)

lemma *bit-numeral-even* [*simp*]:
‹*numeral* (*Num.Bit0 n*) = (*0* :: *bit*)›
by (*simp only*: *Z2.bit-eq-iff even-numeral*) *simp*

lemma *bit-numeral-odd* [*simp*]:
‹*numeral* (*Num.Bit1 n*) = (*1* :: *bit*)›
by (*simp only*: *Z2.bit-eq-iff odd-numeral*) *simp*

end

110 Pointwise order on product types

theory *Product-Order*
imports *Product-Plus*
begin

110.1 Pointwise ordering

instantiation *prod* :: (*ord*, *ord*) *ord*

begin

definition
$x \leq y \longleftrightarrow fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$(x::'a \times 'b) < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$

instance ..

end

lemma *fst-mono*: $x \leq y \Longrightarrow fst\ x \leq fst\ y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *snd-mono*: $x \leq y \Longrightarrow snd\ x \leq snd\ y$
unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-mono*: $x \leq x' \Longrightarrow y \leq y' \Longrightarrow (x,\ y) \leq (x',\ y')$
unfolding *less-eq-prod-def* **by** *simp*

lemma *Pair-le* [*simp*]: $(a,\ b) \leq (c,\ d) \longleftrightarrow a \leq c \wedge b \leq d$
unfolding *less-eq-prod-def* **by** *simp*

lemma *atLeastAtMost-prod-eq*: $\{a..b\} = \{fst\ a..fst\ b\} \times \{snd\ a..snd\ b\}$
by (*auto simp*: *less-eq-prod-def*)

instance *prod* :: (*preorder*, *preorder*) *preorder*
proof
fix $x\ y\ z :: 'a \times 'b$
show $x < y \longleftrightarrow x \leq y \wedge \neg\ y \leq x$
by (*rule less-prod-def*)
show $x \leq x$
unfolding *less-eq-prod-def*
by *fast*
assume $x \leq y$ **and** $y \leq z$ **thus** $x \leq z$
unfolding *less-eq-prod-def*
by (*fast elim*: *order-trans*)
qed

instance *prod* :: (*order*, *order*) *order*
by *standard auto*

110.2 Binary infimum and supremum

instantiation *prod* :: (*inf*, *inf*) *inf*
begin

definition *inf x y* = (*inf* (*fst x*) (*fst y*), *inf* (*snd x*) (*snd y*))

lemma *inf-Pair-Pair* [*simp*]: *inf* (*a*, *b*) (*c*, *d*) = (*inf a c*, *inf b d*)
unfolding *inf-prod-def* **by** *simp*

lemma *fst-inf* [*simp*]: *fst* (*inf x y*) = *inf* (*fst x*) (*fst y*)
unfolding *inf-prod-def* **by** *simp*

lemma *snd-inf* [*simp*]: *snd* (*inf x y*) = *inf* (*snd x*) (*snd y*)
unfolding *inf-prod-def* **by** *simp*

instance ..

end

instance *prod* :: (*semilattice-inf*, *semilattice-inf*) *semilattice-inf*
by *standard auto*

instantiation *prod* :: (*sup*, *sup*) *sup*
begin

definition
sup x y = (*sup* (*fst x*) (*fst y*), *sup* (*snd x*) (*snd y*))

lemma *sup-Pair-Pair* [*simp*]: *sup* (*a*, *b*) (*c*, *d*) = (*sup a c*, *sup b d*)
unfolding *sup-prod-def* **by** *simp*

lemma *fst-sup* [*simp*]: *fst* (*sup x y*) = *sup* (*fst x*) (*fst y*)
unfolding *sup-prod-def* **by** *simp*

lemma *snd-sup* [*simp*]: *snd* (*sup x y*) = *sup* (*snd x*) (*snd y*)
unfolding *sup-prod-def* **by** *simp*

instance ..

end

instance *prod* :: (*semilattice-sup*, *semilattice-sup*) *semilattice-sup*
by *standard auto*

instance *prod* :: (*lattice*, *lattice*) *lattice* ..

instance *prod* :: (*distrib-lattice*, *distrib-lattice*) *distrib-lattice*
by *standard* (*auto simp add*: *sup-inf-distrib1*)

110.3 Top and bottom elements

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ..

end

lemma *fst-top* [*simp*]: *fst top* = *top*
unfolding *top-prod-def* **by** *simp*

lemma *snd-top* [*simp*]: *snd top* = *top*
unfolding *top-prod-def* **by** *simp*

lemma *Pair-top-top*: (*top*, *top*) = *top*
unfolding *top-prod-def* **by** *simp*

instance *prod* :: (*order-top*, *order-top*) *order-top*
by *standard* (*auto simp add*: *top-prod-def*)

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ..

end

lemma *fst-bot* [*simp*]: *fst bot* = *bot*
unfolding *bot-prod-def* **by** *simp*

lemma *snd-bot* [*simp*]: *snd bot* = *bot*
unfolding *bot-prod-def* **by** *simp*

lemma *Pair-bot-bot*: (*bot*, *bot*) = *bot*
unfolding *bot-prod-def* **by** *simp*

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
by *standard* (*auto simp add*: *bot-prod-def*)

instance *prod* :: (*bounded-lattice*, *bounded-lattice*) *bounded-lattice* **..**

instance *prod* :: (*boolean-algebra*, *boolean-algebra*) *boolean-algebra*
by *standard* (*auto simp add*: *prod-eqI diff-eq*)

110.4 Complete lattice operations

instantiation *prod* :: (*Inf*, *Inf*) *Inf*
begin

definition *Inf A* = (*INF x∈A. fst x*, *INF x∈A. snd x*)

instance ..

end

instantiation *prod* :: (*Sup*, *Sup*) *Sup*
begin

definition *Sup A* = (*SUP x∈A. fst x*, *SUP x∈A. snd x*)

instance ..

end

instance *prod* :: (*conditionally-complete-lattice*, *conditionally-complete-lattice*)
conditionally-complete-lattice
by *standard* (*force simp*: *less-eq-prod-def Inf-prod-def Sup-prod-def bdd-below-def bdd-above-def*
intro!: *cInf-lower cSup-upper cInf-greatest cSup-least*)+

instance *prod* :: (*complete-lattice*, *complete-lattice*) *complete-lattice*
by *standard* (*simp-all add*: *less-eq-prod-def Inf-prod-def Sup-prod-def*
INF-lower SUP-upper le-INF-iff SUP-le-iff bot-prod-def top-prod-def)

lemma *fst-Inf*: *fst* (*Inf A*) = (*INF x∈A. fst x*)
by (*simp add*: *Inf-prod-def*)

lemma *fst-INF*: *fst* (*INF x∈A. f x*) = (*INF x∈A. fst* (*f x*))
by (*simp add*: *fst-Inf image-image*)

lemma *fst-Sup*: *fst* (*Sup A*) = (*SUP x∈A. fst x*)
by (*simp add*: *Sup-prod-def*)

lemma *fst-SUP*: *fst* (*SUP x∈A. f x*) = (*SUP x∈A. fst* (*f x*))
by (*simp add*: *fst-Sup image-image*)

lemma *snd-Inf*: *snd* (*Inf A*) = (*INF x∈A. snd x*)
by (*simp add*: *Inf-prod-def*)

lemma *snd-INF*: *snd* (*INF x∈A. f x*) = (*INF x∈A. snd* (*f x*))
by (*simp add*: *snd-Inf image-image*)

lemma *snd-Sup*: *snd* (*Sup A*) = (*SUP x∈A. snd x*)
by (*simp add*: *Sup-prod-def*)

lemma *snd-SUP*: *snd* (*SUP* $x{\in}A.\ f\ x$) = (*SUP* $x{\in}A.$ *snd* ($f\ x$))
by (*simp add*: *snd-Sup image-image*)

lemma *INF-Pair*: (*INF* $x{\in}A.$ ($f\ x$, $g\ x$)) = (*INF* $x{\in}A.\ f\ x$, *INF* $x{\in}A.\ g\ x$)
by (*simp add*: *Inf-prod-def image-image*)

lemma *SUP-Pair*: (*SUP* $x{\in}A.$ ($f\ x$, $g\ x$)) = (*SUP* $x{\in}A.\ f\ x$, *SUP* $x{\in}A.\ g\ x$)
by (*simp add*: *Sup-prod-def image-image*)

Alternative formulations for set infima and suprema over the product of two complete lattices:

lemma *INF-prod-alt-def*:
Inf (f ‘ A) = (*Inf* ((*fst* ∘ f) ‘ A), *Inf* ((*snd* ∘ f) ‘ A))
by (*simp add*: *Inf-prod-def image-image*)

lemma *SUP-prod-alt-def*:
Sup (f ‘ A) = (*Sup* ((*fst* ∘ f) ‘ A), *Sup*((*snd* ∘ f) ‘ A))
by (*simp add*: *Sup-prod-def image-image*)

110.5 Complete distributive lattices

instance *prod* :: (*complete-distrib-lattice*, *complete-distrib-lattice*) *complete-distrib-lattice*

proof
fix A::($'a{\times}'b$) *set set*
show *Inf* (*Sup* ‘ A) ≤ *Sup* (*Inf* ‘ {f ‘ A |f. ∀ $Y{\in}A.\ f\ Y \in Y$})
by (*simp add*: *Inf-prod-def Sup-prod-def INF-SUP-set image-image*)
qed

110.6 Bekic's Theorem

Simultaneous fixed points over pairs can be written in terms of separate fixed points. Transliterated from HOLCF.Fix by Peter Gammie

lemma *lfp-prod*:
fixes F :: $'a$::*complete-lattice* × $'b$::*complete-lattice* ⇒ $'a$ × $'b$
assumes *mono F*
shows *lfp* F = (*lfp* (λx. *fst* (F (x, *lfp* (λy. *snd* (F (x, y)))))),
(*lfp* (λy. *snd* (F (*lfp* (λx. *fst* (F (x, *lfp* (λy. *snd* (F (x, y)))))), y)))))
(**is** *lfp* F = (*?x*, *?y*))
proof(*rule lfp-eqI*[*OF assms*])
have *1*: *fst* (F (*?x*, *?y*)) = *?x*
by (*rule trans* [*symmetric*, *OF lfp-unfold*])
(*blast intro*!: *monoI monoD*[*OF assms*(*1*)] *fst-mono snd-mono Pair-mono lfp-mono*)+
have *2*: *snd* (F (*?x*, *?y*)) = *?y*
by (*rule trans* [*symmetric*, *OF lfp-unfold*])
(*blast intro*!: *monoI monoD*[*OF assms*(*1*)] *fst-mono snd-mono Pair-mono lfp-mono*)+

from *1 2* **show** *F* (*?x*, *?y*) = (*?x*, *?y*) **by** (*simp add*: *prod-eq-iff*)
next
fix *z* **assume** *F-z*: *F z* = *z*
obtain *x y* **where** *z*: *z* = (*x*, *y*) **by** (*rule prod.exhaust*)
from *F-z z* **have** *F-x*: *fst* (*F* (*x*, *y*)) = *x* **by** *simp*
from *F-z z* **have** *F-y*: *snd* (*F* (*x*, *y*)) = *y* **by** *simp*
let *?y1* = *lfp* (λ*y*. *snd* (*F* (*x*, *y*)))
have *?y1* ≤ *y* **by** (*rule lfp-lowerbound*, *simp add*: *F-y*)
hence *fst* (*F* (*x*, *?y1*)) ≤ *fst* (*F* (*x*, *y*))
by (*simp add*: *assms fst-mono monoD*)
hence *fst* (*F* (*x*, *?y1*)) ≤ *x* **using** *F-x* **by** *simp*
hence *1*: *?x* ≤ *x* **by** (*simp add*: *lfp-lowerbound*)
hence *snd* (*F* (*?x*, *y*)) ≤ *snd* (*F* (*x*, *y*))
by (*simp add*: *assms snd-mono monoD*)
hence *snd* (*F* (*?x*, *y*)) ≤ *y* **using** *F-y* **by** *simp*
hence *2*: *?y* ≤ *y* **by** (*simp add*: *lfp-lowerbound*)
show (*?x*, *?y*) ≤ *z* **using** *z 1 2* **by** *simp*
qed

lemma *gfp-prod*:
fixes *F* :: *'a*::*complete-lattice* × *'b*::*complete-lattice* ⇒ *'a* × *'b*
assumes *mono F*
shows *gfp F* = (*gfp* (λ*x*. *fst* (*F* (*x*, *gfp* (λ*y*. *snd* (*F* (*x*, *y*)))))),
(*gfp* (λ*y*. *snd* (*F* (*gfp* (λ*x*. *fst* (*F* (*x*, *gfp* (λ*y*. *snd* (*F* (*x*, *y*)))))), *y*)))))
(**is** *gfp F* = (*?x*, *?y*))
proof(*rule gfp-eqI*[*OF assms*])
have *1*: *fst* (*F* (*?x*, *?y*)) = *?x*
by (*rule trans* [*symmetric*, *OF gfp-unfold*])
(*blast intro*!: *monoI monoD*[*OF assms*(*1*)] *fst-mono snd-mono Pair-mono gfp-mono*)+
have *2*: *snd* (*F* (*?x*, *?y*)) = *?y*
by (*rule trans* [*symmetric*, *OF gfp-unfold*])
(*blast intro*!: *monoI monoD*[*OF assms*(*1*)] *fst-mono snd-mono Pair-mono gfp-mono*)+
from *1 2* **show** *F* (*?x*, *?y*) = (*?x*, *?y*) **by** (*simp add*: *prod-eq-iff*)
next
fix *z* **assume** *F-z*: *F z* = *z*
obtain *x y* **where** *z*: *z* = (*x*, *y*) **by** (*rule prod.exhaust*)
from *F-z z* **have** *F-x*: *fst* (*F* (*x*, *y*)) = *x* **by** *simp*
from *F-z z* **have** *F-y*: *snd* (*F* (*x*, *y*)) = *y* **by** *simp*
let *?y1* = *gfp* (λ*y*. *snd* (*F* (*x*, *y*)))
have *y* ≤ *?y1* **by** (*rule gfp-upperbound*, *simp add*: *F-y*)
hence *fst* (*F* (*x*, *y*)) ≤ *fst* (*F* (*x*, *?y1*))
by (*simp add*: *assms fst-mono monoD*)
hence *x* ≤ *fst* (*F* (*x*, *?y1*)) **using** *F-x* **by** *simp*
hence *1*: *x* ≤ *?x* **by** (*simp add*: *gfp-upperbound*)
hence *snd* (*F* (*x*, *y*)) ≤ *snd* (*F* (*?x*, *y*))
by (*simp add*: *assms snd-mono monoD*)
hence *y* ≤ *snd* (*F* (*?x*, *y*)) **using** *F-y* **by** *simp*

hence *2*: $y \leq ?y$ **by** (*simp add*: *gfp-upperbound*)
show $z \leq (?x, ?y)$ **using** *z 1 2* **by** *simp*
qed

end

111 Finite Lattices

theory *Finite-Lattice*
imports *Product-Order*
begin

111.1 Finite Complete Lattices

A non-empty finite lattice is a complete lattice. Since types are never empty in Isabelle/HOL, a type of classes *finite* and *lattice* should also have class *complete-lattice*. A type class is defined that extends classes *finite* and *lattice* with the operators *bot*, *top*, *Inf*, and *Sup*, along with assumptions that define these operators in terms of the ones of classes *finite* and *lattice*. The resulting class is a subclass of *complete-lattice*.

class *finite-lattice-complete* = *finite* + *lattice* + *bot* + *top* + *Inf* + *Sup* +
assumes *bot-def*: *bot* = *Inf-fin UNIV*
assumes *top-def*: *top* = *Sup-fin UNIV*
assumes *Inf-def*: *Inf A* = *Finite-Set.fold inf top A*
assumes *Sup-def*: *Sup A* = *Finite-Set.fold sup bot A*

The definitional assumptions on the operators *bot* and *top* of class *finite-lattice-complete* ensure that they yield bottom and top.

lemma *finite-lattice-complete-bot-least*: (*bot*::'*a*::*finite-lattice-complete*) $\leq x$
by (*auto simp*: *bot-def intro*: *Inf-fin.coboundedI*)

instance *finite-lattice-complete* $\subseteq$ *order-bot*
by *standard* (*auto simp*: *finite-lattice-complete-bot-least*)

lemma *finite-lattice-complete-top-greatest*: (*top*::'*a*::*finite-lattice-complete*) $\geq x$
by (*auto simp*: *top-def Sup-fin.coboundedI*)

instance *finite-lattice-complete* $\subseteq$ *order-top*
by *standard* (*auto simp*: *finite-lattice-complete-top-greatest*)

instance *finite-lattice-complete* $\subseteq$ *bounded-lattice* **..**

The definitional assumptions on the operators *Inf* and *Sup* of class *finite-lattice-complete* ensure that they yield infimum and supremum.

lemma *finite-lattice-complete-Inf-empty*: *Inf* {} = (*top* :: '*a*::*finite-lattice-complete*)
by (*simp add*: *Inf-def*)

lemma *finite-lattice-complete-Sup-empty*: *Sup* {} = (*bot* :: 'a::*finite-lattice-complete*)
by (*simp add*: *Sup-def*)

lemma *finite-lattice-complete-Inf-insert*:
fixes *A* :: 'a::*finite-lattice-complete set*
shows *Inf* (*insert x A*) = *inf x* (*Inf A*)
proof −
interpret *comp-fun-idem inf* :: 'a ⇒ -
by (*fact comp-fun-idem-inf*)
show *?thesis* **by** (*simp add*: *Inf-def*)
qed

lemma *finite-lattice-complete-Sup-insert*:
fixes *A* :: 'a::*finite-lattice-complete set*
shows *Sup* (*insert x A*) = *sup x* (*Sup A*)
proof −
interpret *comp-fun-idem sup* :: 'a ⇒ -
by (*fact comp-fun-idem-sup*)
show *?thesis* **by** (*simp add*: *Sup-def*)
qed

lemma *finite-lattice-complete-Inf-lower*:
(*x*::'a::*finite-lattice-complete*) ∈ *A* ⟹ *Inf A* ≤ *x*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Inf-insert intro*: *le-infI2*)

lemma *finite-lattice-complete-Inf-greatest*:
∀ *x*::'a::*finite-lattice-complete* ∈ *A*. *z* ≤ *x* ⟹ *z* ≤ *Inf A*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Inf-empty finite-lattice-complete-Inf-insert*)

lemma *finite-lattice-complete-Sup-upper*:
(*x*::'a::*finite-lattice-complete*) ∈ *A* ⟹ *Sup A* ≥ *x*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Sup-insert intro*: *le-supI2*)

lemma *finite-lattice-complete-Sup-least*:
∀ *x*::'a::*finite-lattice-complete* ∈ *A*. *z* ≥ *x* ⟹ *z* ≥ *Sup A*
using *finite* [*of A*]
by (*induct A*) (*auto simp add*: *finite-lattice-complete-Sup-empty finite-lattice-complete-Sup-insert*)

instance *finite-lattice-complete* ⊆ *complete-lattice*
proof
qed (*auto simp*:
finite-lattice-complete-Inf-lower
finite-lattice-complete-Inf-greatest
finite-lattice-complete-Sup-upper
finite-lattice-complete-Sup-least
finite-lattice-complete-Inf-empty

finite-lattice-complete-Sup-empty)

The product of two finite lattices is already a finite lattice.

lemma *finite-bot-prod*:
(*bot* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*)) =
Inf-fin UNIV
by (*metis Inf-fin.coboundedI UNIV-I bot.extremum-uniqueI finite-UNIV*)

lemma *finite-top-prod*:
(*top* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*)) =
Sup-fin UNIV
by (*metis Sup-fin.coboundedI UNIV-I top.extremum-uniqueI finite-UNIV*)

lemma *finite-Inf-prod*:
Inf(*A* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-prod*:
Sup (*A* :: (*'a*::*finite-lattice-complete* × *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
by (*metis Sup-fold-sup finite*)

instance *prod* :: (*finite-lattice-complete*, *finite-lattice-complete*) *finite-lattice-complete*
by *standard* (*auto simp*: *finite-bot-prod finite-top-prod finite-Inf-prod finite-Sup-prod*)

Functions with a finite domain and with a finite lattice as codomain already form a finite lattice.

lemma *finite-bot-fun*: (*bot* :: (*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*)) = *Inf-fin UNIV*
by (*metis Inf-UNIV Inf-fin-Inf empty-not-UNIV finite*)

lemma *finite-top-fun*: (*top* :: (*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*)) = *Sup-fin UNIV*
by (*metis Sup-UNIV Sup-fin-Sup empty-not-UNIV finite*)

lemma *finite-Inf-fun*:
Inf (*A*::(*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold inf top A
by (*metis Inf-fold-inf finite*)

lemma *finite-Sup-fun*:
Sup (*A*::(*'a*::*finite* ⇒ *'b*::*finite-lattice-complete*) *set*) =
Finite-Set.fold sup bot A
by (*metis Sup-fold-sup finite*)

instance *fun* :: (*finite*, *finite-lattice-complete*) *finite-lattice-complete*
by *standard* (*auto simp*: *finite-bot-fun finite-top-fun finite-Inf-fun finite-Sup-fun*)

111.2 Finite Distributive Lattices

A finite distributive lattice is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

class *finite-distrib-lattice-complete* =
distrib-lattice + *finite-lattice-complete*

lemma *finite-distrib-lattice-complete-sup-Inf*:
sup (*x*::'*a*::*finite-distrib-lattice-complete*) (*Inf A*) = (*INF y*∈*A*. *sup x y*)
using *finite*
by (*induct A rule*: *finite-induct*) (*simp-all add*: *sup-inf-distrib1*)

lemma *finite-distrib-lattice-complete-inf-Sup*:
inf (*x*::'*a*::*finite-distrib-lattice-complete*) (*Sup A*) = (*SUP y*∈*A*. *inf x y*)
using *finite* [*of A*] **by** *induct* (*simp-all add*: *inf-sup-distrib1*)

context *finite-distrib-lattice-complete*
begin
subclass *finite-distrib-lattice*
proof −
show *class.finite-distrib-lattice Inf Sup inf* (≤) (<) *sup bot top*
proof
show *bot* = *Inf UNIV*
unfolding *bot-def top-def Inf-def*
using *Inf-fin.eq-fold Inf-fin.insert inf.absorb2* **by** *force*
next
show *top* = *Sup UNIV*
unfolding *bot-def top-def Sup-def*
using *Sup-fin.eq-fold Sup-fin.insert* **by** *force*
next
show *Inf* {} = *Sup UNIV*
unfolding *Inf-def Sup-def bot-def top-def*
using *Sup-fin.eq-fold Sup-fin.insert* **by** *force*
next
show *Sup* {} = *Inf UNIV*
unfolding *Inf-def Sup-def bot-def top-def*
using *Inf-fin.eq-fold Inf-fin.insert inf.absorb2* **by** *force*
next
interpret *comp-fun-idem-inf*: *comp-fun-idem inf*
by (*fact comp-fun-idem-inf*)
show *Inf* (*insert a A*) = *inf a* (*Inf A*) **for** *a A*
using *comp-fun-idem-inf.fold-insert-idem Inf-def finite* **by** *simp*
next
interpret *comp-fun-idem-sup*: *comp-fun-idem sup*
by (*fact comp-fun-idem-sup*)
show *Sup* (*insert a A*) = *sup a* (*Sup A*) **for** *a A*
using *comp-fun-idem-sup.fold-insert-idem Sup-def finite* **by** *simp*
qed
qed

end

instance *finite-distrib-lattice-complete* ⊆ *complete-distrib-lattice* **..**

The product of two finite distributive lattices is already a finite distributive lattice.

instance *prod* ::
(*finite-distrib-lattice-complete*, *finite-distrib-lattice-complete*)
finite-distrib-lattice-complete
..

Functions with a finite domain and with a finite distributive lattice as codomain already form a finite distributive lattice.

instance *fun* ::
(*finite*, *finite-distrib-lattice-complete*) *finite-distrib-lattice-complete*
..

111.3 Linear Orders

A linear order is a distributive lattice. A type class is defined that extends class *linorder* with the operators *inf* and *sup*, along with assumptions that define these operators in terms of the ones of class *linorder*. The resulting class is a subclass of *distrib-lattice*.

class *linorder-lattice* = *linorder* + *inf* + *sup* +
assumes *inf-def*: *inf x y* = (*if x* ≤ *y then x else y*)
assumes *sup-def*: *sup x y* = (*if x* ≥ *y then x else y*)

The definitional assumptions on the operators *inf* and *sup* of class *linorder-lattice* ensure that they yield infimum and supremum and that they distribute over each other.

lemma *linorder-lattice-inf-le1*: *inf* (*x*::'*a*::*linorder-lattice*) *y* ≤ *x*
unfolding *inf-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-inf-le2*: *inf* (*x*::'*a*::*linorder-lattice*) *y* ≤ *y*
unfolding *inf-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-inf-greatest*:
(*x*::'*a*::*linorder-lattice*) ≤ *y* ⟹ *x* ≤ *z* ⟹ *x* ≤ *inf y z*
unfolding *inf-def* **by** (*metis* (*full-types*))

lemma *linorder-lattice-sup-ge1*: *sup* (*x*::'*a*::*linorder-lattice*) *y* ≥ *x*
unfolding *sup-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-sup-ge2*: *sup* (*x*::'*a*::*linorder-lattice*) *y* ≥ *y*
unfolding *sup-def* **by** (*metis* (*full-types*) *linorder-linear*)

lemma *linorder-lattice-sup-least*:
(*x*::'*a*::*linorder-lattice*) ≥ *y* ⟹ *x* ≥ *z* ⟹ *x* ≥ *sup y z*

by (*auto simp*: *sup-def*)

lemma *linorder-lattice-sup-inf-distrib1*:
sup (*x*::'*a*::*linorder-lattice*) (*inf y z*) = *inf* (*sup x y*) (*sup x z*)
by (*auto simp*: *inf-def sup-def*)

instance *linorder-lattice* ⊆ *distrib-lattice*
proof
qed (*auto simp*:
linorder-lattice-inf-le1
linorder-lattice-inf-le2
linorder-lattice-inf-greatest
linorder-lattice-sup-ge1
linorder-lattice-sup-ge2
linorder-lattice-sup-least
linorder-lattice-sup-inf-distrib1)

111.4 Finite Linear Orders

A (non-empty) finite linear order is a complete linear order.

class *finite-linorder-complete* = *linorder-lattice* + *finite-lattice-complete*

instance *finite-linorder-complete* ⊆ *complete-linorder* **..**

A (non-empty) finite linear order is a complete lattice whose *inf* and *sup* operators distribute over *Sup* and *Inf*.

instance *finite-linorder-complete* ⊆ *finite-distrib-lattice-complete* **..**

end

112 Lexicographic order on lists

theory *List-Lexorder*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: *xs* < *ys* ⟷ (*xs*, *ys*) ∈ *lexord* {(*u*, *v*). *u* < *v*}

definition
list-le-def: (*xs* :: - *list*) ≤ *ys* ⟷ *xs* < *ys* ∨ *xs* = *ys*

instance ..

end

instance *list* :: (*order*) *order*
proof
let *?r* = {(*u*, *v*::'*a*). *u* < *v*}
have *tr*: *trans ?r*
using *trans-def* **by** *fastforce*
have §: *False*
if (*xs*,*ys*) ∈ *lexord ?r* (*ys*,*xs*) ∈ *lexord ?r* **for** *xs ys* :: '*a list*
proof −
have (*xs*,*xs*) ∈ *lexord ?r*
using *that transD* [*OF lexord-transI* [*OF tr*]] **by** *blast*
then show *False*
by (*meson case-prodD lexord-irreflexive less-irrefl mem-Collect-eq*)
qed
show *xs* ≤ *xs* **for** *xs* :: '*a list* **by** (*simp add*: *list-le-def*)
show *xs* ≤ *zs* **if** *xs* ≤ *ys* **and** *ys* ≤ *zs* **for** *xs ys zs* :: '*a list*
using *that transD* [*OF lexord-transI* [*OF tr*]] **by** (*auto simp add*: *list-le-def list-less-def*)
show *xs* = *ys* **if** *xs* ≤ *ys ys* ≤ *xs* **for** *xs ys* :: '*a list*
using § *that list-le-def list-less-def* **by** *blast*
show *xs* < *ys* ⟷ *xs* ≤ *ys* ∧ ¬ *ys* ≤ *xs* **for** *xs ys* :: '*a list*
by (*auto simp add*: *list-less-def list-le-def dest*: §)
qed

instance *list* :: (*linorder*) *linorder*
proof
fix *xs ys* :: '*a list*
have *total* (*lexord* {(*u*, *v*::'*a*). *u* < *v*})
by (*rule total-lexord*) (*auto simp*: *total-on-def*)
then show *xs* ≤ *ys* ∨ *ys* ≤ *xs*
by (*auto simp add*: *total-on-def list-le-def list-less-def*)
qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: '*a list* ⇒ -) = *min*

definition (*sup* :: '*a list* ⇒ -) = *max*

instance
by *standard* (*auto simp add*: *inf-list-def sup-list-def max-min-distrib2*)

end

lemma *not-less-Nil* [*simp*]: ¬ *x* < []
by (*simp add*: *list-less-def*)

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*

by (*simp add*: *list-less-def*)

lemma *Cons-less-Cons* [*simp*]: $a \# x < b \# y \longleftrightarrow a < b \vee a = b \wedge x < y$
by (*simp add*: *list-less-def*)

lemma *le-Nil* [*simp*]: $x \leq [] \longleftrightarrow x = []$
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: $[] \leq x$
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Cons-le-Cons* [*simp*]: $a \# x \leq b \# y \longleftrightarrow a < b \vee a = b \wedge x \leq y$
unfolding *list-le-def* **by** *auto*

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
by *standard* (*simp add*: *bot-list-def*)

end

lemma *less-list-code* [*code*]:
$xs < ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] < (x::'a::\{equal, order\}) \# xs \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs < y \# ys \longleftrightarrow x < y \vee x = y \wedge xs < ys$
by *simp-all*

lemma *less-eq-list-code* [*code*]:
$x \# xs \leq ([]::'a::\{equal, order\}\ list) \longleftrightarrow False$
$[] \leq (xs::'a::\{equal, order\}\ list) \longleftrightarrow True$
$(x::'a::\{equal, order\}) \# xs \leq y \# ys \longleftrightarrow x < y \vee x = y \wedge xs \leq ys$
by *simp-all*

end

113 Lexicographic order on lists

This version prioritises length and can yield wellorderings

theory *List-Lenlexorder*
imports *Main*
begin

instantiation *list* :: (*ord*) *ord*
begin

definition
list-less-def: $xs < ys \longleftrightarrow (xs, ys) \in lenlex\ \{(u, v).\ u < v\}$

definition
list-le-def: $(xs :: - list) \leq ys \longleftrightarrow xs < ys \vee xs = ys$

instance ..

end

instance *list* :: (*order*) *order*
proof
have *tr*: *trans* $\{(u, v::'a).\ u < v\}$
using *trans-def* **by** *fastforce*
have §: *False*
if $(xs,ys) \in lenlex\ \{(u, v).\ u < v\}$ $(ys,xs) \in lenlex\ \{(u, v).\ u < v\}$ **for** *xs ys* :: *'a list*
proof −
have $(xs,xs) \in lenlex\ \{(u, v).\ u < v\}$
using *that transD* [*OF lenlex-transI* [*OF tr*]] **by** *blast*
then show *False*
by (*meson case-prodD lenlex-irreflexive less-irrefl mem-Collect-eq*)
qed
show $xs \leq xs$ **for** *xs* :: *'a list* **by** (*simp add*: *list-le-def*)
show $xs \leq zs$ **if** $xs \leq ys$ **and** $ys \leq zs$ **for** *xs ys zs* :: *'a list*
using *that transD* [*OF lenlex-transI* [*OF tr*]] **by** (*auto simp add*: *list-le-def list-less-def*)
show $xs = ys$ **if** $xs \leq ys$ $ys \leq xs$ **for** *xs ys* :: *'a list*
using § *that list-le-def list-less-def* **by** *blast*
show $xs < ys \longleftrightarrow xs \leq ys \wedge \neg\ ys \leq xs$ **for** *xs ys* :: *'a list*
by (*auto simp add*: *list-less-def list-le-def dest*: §)
qed

instance *list* :: (*linorder*) *linorder*
proof
fix *xs ys* :: *'a list*
have *total* (*lenlex* $\{(u, v::'a).\ u < v\}$)
by (*rule total-lenlex*) (*auto simp*: *total-on-def*)
then show $xs \leq ys \vee ys \leq xs$
by (*auto simp add*: *total-on-def list-le-def list-less-def*)
qed

instance *list* :: (*wellorder*) *wellorder*
proof
fix *P* :: *'a list* ⇒ *bool* **and** *a*
assume $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
then show *P a*
unfolding *list-less-def* **by** (*metis wf-lenlex wf-induct wf-lenlex wf*)

qed

instantiation *list* :: (*linorder*) *distrib-lattice*
begin

definition (*inf* :: *'a list* ⇒ -) = *min*

definition (*sup* :: *'a list* ⇒ -) = *max*

instance
by *standard* (*auto simp add*: *inf-list-def sup-list-def max-min-distrib2*)

end

lemma *not-less-Nil* [*simp*]: ¬ *x* < []
by (*simp add*: *list-less-def*)

lemma *Nil-less-Cons* [*simp*]: [] < *a* # *x*
by (*simp add*: *list-less-def*)

lemma *Cons-less-Cons*: *a* # *x* < *b* # *y* ⟷ *length x* < *length y* ∨ *length x* = *length y* ∧ (*a* < *b* ∨ *a* = *b* ∧ *x* < *y*)
using *lenlex-length*
by (*fastforce simp*: *list-less-def Cons-lenlex-iff*)

lemma *le-Nil* [*simp*]: *x* ≤ [] ⟷ *x* = []
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Nil-le-Cons* [*simp*]: [] ≤ *x*
unfolding *list-le-def* **by** (*cases x*) *auto*

lemma *Cons-le-Cons*: *a* # *x* ≤ *b* # *y* ⟷ *length x* < *length y* ∨ *length x* = *length y* ∧ (*a* < *b* ∨ *a* = *b* ∧ *x* ≤ *y*)
by (*auto simp*: *list-le-def Cons-less-Cons*)

instantiation *list* :: (*order*) *order-bot*
begin

definition *bot* = []

instance
by *standard* (*simp add*: *bot-list-def*)

end

end

114 Prefix order on lists as order class instance

theory *Prefix-Order*
imports *Sublist*
begin

instantiation *list* :: (*type*) *order*
begin

definition $xs \leq ys \equiv$ *prefix xs ys* **for** *xs ys* :: *'a list*
definition $xs < ys \equiv xs \leq ys \wedge \neg (ys \leq xs)$ **for** *xs ys* :: *'a list*

instance
by *standard* (*auto simp*: *less-eq-list-def less-list-def*)

end

lemma *less-list-def'*: $xs < ys \longleftrightarrow$ *strict-prefix xs ys* **for** *xs ys* :: *'a list*
by (*simp add*: *less-eq-list-def order.strict-iff-order prefix-order.less-le*)

lemmas *prefixI* [*intro?*] = *prefixI* [*folded less-eq-list-def*]
lemmas *prefixE* [*elim?*] = *prefixE* [*folded less-eq-list-def*]
lemmas *strict-prefixI'* [*intro?*] = *strict-prefixI'* [*folded less-list-def'*]
lemmas *strict-prefixE'* [*elim?*] = *strict-prefixE'* [*folded less-list-def'*]
lemmas *strict-prefixI* [*intro?*] = *strict-prefixI* [*folded less-list-def'*]
lemmas *strict-prefixE* [*elim?*] = *strict-prefixE* [*folded less-list-def'*]
lemmas *Nil-prefix* [*iff*] = *Nil-prefix* [*folded less-eq-list-def*]
lemmas *prefix-Nil* [*simp*] = *prefix-Nil* [*folded less-eq-list-def*]
lemmas *prefix-snoc* [*simp*] = *prefix-snoc* [*folded less-eq-list-def*]
lemmas *Cons-prefix-Cons* [*simp*] = *Cons-prefix-Cons* [*folded less-eq-list-def*]
lemmas *same-prefix-prefix* [*simp*] = *same-prefix-prefix* [*folded less-eq-list-def*]
lemmas *same-prefix-nil* [*iff*] = *same-prefix-nil* [*folded less-eq-list-def*]
lemmas *prefix-prefix* [*simp*] = *prefix-prefix* [*folded less-eq-list-def*]
lemmas *prefix-Cons* = *prefix-Cons* [*folded less-eq-list-def*]
lemmas *prefix-length-le* = *prefix-length-le* [*folded less-eq-list-def*]
lemmas *strict-prefix-simps* [*simp*, *code*] = *strict-prefix-simps* [*folded less-list-def'*]
lemmas *not-prefix-induct* [*consumes 1*, *case-names Nil Neq Eq*] =
not-prefix-induct [*folded less-eq-list-def*]

end

115 Lexicographic order on product types

theory *Product-Lexorder*
imports *Main*
begin

instantiation *prod* :: (*ord*, *ord*) *ord*
begin

definition
$x \leq y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x \leq snd\ y$

definition
$x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x \leq fst\ y \wedge snd\ x < snd\ y$

instance ..

end

lemma *less-eq-prod-simp* [*simp*, *code*]:
$(x1,\ y1) \leq (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 \leq y2$
by (*simp add*: *less-eq-prod-def*)

lemma *less-prod-simp* [*simp*, *code*]:
$(x1,\ y1) < (x2,\ y2) \longleftrightarrow x1 < x2 \vee x1 \leq x2 \wedge y1 < y2$
by (*simp add*: *less-prod-def*)

A stronger version for partial orders.

lemma *less-prod-def'*:
fixes $x\ y :: 'a{::}order \times 'b{::}ord$
shows $x < y \longleftrightarrow fst\ x < fst\ y \vee fst\ x = fst\ y \wedge snd\ x < snd\ y$
by (*auto simp add*: *less-prod-def le-less*)

instance *prod* :: (*preorder*, *preorder*) *preorder*
by *standard* (*auto simp*: *less-eq-prod-def less-prod-def less-le-not-le intro*: *order-trans*)

instance *prod* :: (*order*, *order*) *order*
by *standard* (*auto simp add*: *less-eq-prod-def*)

instance *prod* :: (*linorder*, *linorder*) *linorder*
by *standard* (*auto simp*: *less-eq-prod-def*)

instantiation *prod* :: (*linorder*, *linorder*) *distrib-lattice*
begin

definition
$(inf :: 'a \times 'b \Rightarrow - \Rightarrow -) = min$

definition
$(sup :: 'a \times 'b \Rightarrow - \Rightarrow -) = max$

instance
by *standard* (*auto simp add*: *inf-prod-def sup-prod-def max-min-distrib2*)

end

instantiation *prod* :: (*bot*, *bot*) *bot*
begin

definition
bot = (*bot*, *bot*)

instance ..

end

instance *prod* :: (*order-bot*, *order-bot*) *order-bot*
by *standard* (*auto simp add*: *bot-prod-def*)

instantiation *prod* :: (*top*, *top*) *top*
begin

definition
top = (*top*, *top*)

instance ..

end

instance *prod* :: (*order-top*, *order-top*) *order-top*
by *standard* (*auto simp add*: *top-prod-def*)

instance *prod* :: (*wellorder*, *wellorder*) *wellorder*
proof
fix $P :: 'a \times 'b \Rightarrow bool$ **and** $z :: 'a \times 'b$
assume P: $\bigwedge x.\ (\bigwedge y.\ y < x \Longrightarrow P\ y) \Longrightarrow P\ x$
show $P\ z$
proof (*induct z*)
case (*Pair a b*)
show $P\ (a,\ b)$
proof (*induct a arbitrary*: *b rule*: *less-induct*)
case (*less* a_1) **note** a_1 = *this*
show $P\ (a_1,\ b)$
proof (*induct b rule*: *less-induct*)
case (*less* b_1) **note** b_1 = *this*
show $P\ (a_1,\ b_1)$
proof (*rule P*)
fix *p* **assume** *p*: $p < (a_1,\ b_1)$
show *P p*
proof (*cases fst p* < a_1)
case *True*
then have *P* (*fst p*, *snd p*) **by** (*rule* a_1)
then show *?thesis* **by** *simp*
next
case *False*

```
        with p have 1: a1 = fst p and 2: snd p < b1
          by (simp-all add: less-prod-def')
        from 2 have P (a1, snd p) by (rule b1)
        with 1 show ?thesis by simp
      qed
    qed
  qed
 qed
qed
```

Legacy lemma bindings

```
lemmas prod-le-def = less-eq-prod-def
lemmas prod-less-def = less-prod-def
lemmas prod-less-eq = less-prod-def'

end
```

116 Subsequence Ordering

```
theory Subseq-Order
imports Sublist
begin
```

This theory defines subsequence ordering on lists. A list *ys* is a subsequence of a list *xs*, iff one obtains *ys* by erasing some elements from *xs*.

116.1 Definitions and basic lemmas

```
instantiation list :: (type) ord
begin

definition less-eq-list
  where ‹xs ≤ ys ⟷ subseq xs ys› for xs ys :: ‹'a list›

definition less-list
  where ‹xs < ys ⟷ xs ≤ ys ∧ ¬ ys ≤ xs› for xs ys :: ‹'a list›

instance ..

end

instance list :: (type) order
proof
  fix xs ys zs :: 'a list
  show xs < ys ⟷ xs ≤ ys ∧ ¬ ys ≤ xs
    unfolding less-list-def ..
  show xs ≤ xs
    by (simp add: less-eq-list-def)
```

show $xs = ys$ **if** $xs \leq ys$ **and** $ys \leq xs$
using *that* **unfolding** *less-eq-list-def*
by (*rule subseq-order.antisym*)
show $xs \leq zs$ **if** $xs \leq ys$ **and** $ys \leq zs$
using *that* **unfolding** *less-eq-list-def*
by (*rule subseq-order.order-trans*)
qed

lemmas *less-eq-list-induct* [*consumes 1*, *case-names empty drop take*] =
list-emb.induct [*of* (=), *folded less-eq-list-def*]

lemma *less-eq-list-empty* [*code*]:
‹[] ≤ *xs* ⟷ *True*›
by (*simp add*: *less-eq-list-def*)

lemma *less-eq-list-below-empty* [*code*]:
‹*x* # *xs* ≤ [] ⟷ *False*›
by (*simp add*: *less-eq-list-def*)

lemma *le-list-Cons2-iff* [*simp*, *code*]:
‹*x* # *xs* ≤ *y* # *ys* ⟷ (*if x* = *y then xs* ≤ *ys else x* # *xs* ≤ *ys*)›
by (*simp add*: *less-eq-list-def*)

lemma *less-list-empty* [*simp*]:
‹[] < *xs* ⟷ *xs* ≠ []›
by (*metis less-eq-list-def list-emb-Nil order-less-le*)

lemma *less-list-empty-Cons* [*code*]:
‹[] < *x* # *xs* ⟷ *True*›
by *simp-all*

lemma *less-list-below-empty* [*simp*, *code*]:
‹*xs* < [] ⟷ *False*›
by (*metis list-emb-Nil less-eq-list-def less-list-def*)

lemma *less-list-Cons2-iff* [*code*]:
‹*x* # *xs* < *y* # *ys* ⟷ (*if x* = *y then xs* < *ys else x* # *xs* ≤ *ys*)›
by (*simp add*: *less-le*)

lemmas *less-eq-list-drop* = *list-emb.list-emb-Cons* [*of* (=), *folded less-eq-list-def*]
lemmas *le-list-map* = *subseq-map* [*folded less-eq-list-def*]
lemmas *le-list-filter* = *subseq-filter* [*folded less-eq-list-def*]
lemmas *le-list-length* = *list-emb-length* [*of* (=), *folded less-eq-list-def*]

lemma *less-list-length*: *xs* < *ys* ⟹ *length xs* < *length ys*
by (*metis list-emb-length subseq-same-length le-neq-implies-less less-list-def less-eq-list-def*)

lemma *less-list-drop*: *xs* < *ys* ⟹ *xs* < *x* # *ys*
by (*unfold less-le less-eq-list-def*) (*auto*)

lemma *less-list-take-iff*: $x \# xs < x \# ys \longleftrightarrow xs < ys$
by (*metis subseq-Cons2-iff less-list-def less-eq-list-def*)

lemma *less-list-drop-many*: $xs < ys \Longrightarrow xs < zs @ ys$
by (*metis subseq-append-le-same-iff subseq-drop-many order-less-le self-append-conv2 less-eq-list-def*)

lemma *less-list-take-many-iff*: $zs @ xs < zs @ ys \longleftrightarrow xs < ys$
by (*metis less-list-def less-eq-list-def subseq-append'*)

lemma *less-list-rev-take*: $xs @ zs < ys @ zs \longleftrightarrow xs < ys$
by (*unfold less-le less-eq-list-def*) *auto*

end

117 Records based on BNF/datatype machinery

theory *Datatype-Records*
imports *Main*
keywords *datatype-record* :: *thy-defn*
begin

This theory provides an alternative, stripped-down implementation of records based on the machinery of the **datatype** package.

It supports:

- similar declaration syntax as records
- record creation and update syntax (using ⦇ ... ⦈ brackets)
- regular datatype features (e.g. dead type variables etc.)
- "after-the-fact" registration of single-free-constructor types as records

Caveats:

- there is no compatibility layer; importing this theory will disrupt existing syntax
- extensible records are not supported

nonterminal
ident **and**
field-type **and**
field-types **and**
field **and**
fields **and**

field-update **and**
field-updates

open-bundle *datatype-record-syntax*
begin

unbundle *no record-syntax*

syntax
-constify :: *id* => *ident* (‹-›)
-constify :: *longid* => *ident* (‹-›)

-datatype-field :: *ident* => *'a* => *field* (‹(‹*indent=2 notation=‹infix field value›*›- =/ -)›)
:: *field* => *fields* (‹-›)
-datatype-fields :: *field* => *fields* => *fields* (‹-,/ -›)
-datatype-record :: *fields* => *'a* (‹(‹*indent=3 notation=‹mixfix datatype record value›*›(‖-‖))›)
-datatype-field-update :: *ident* => *'a* => *field-update* (‹(‹*indent=2 notation=‹infix field update›*›- :=/ -)›)
:: *field-update* => *field-updates* (‹-›)
-datatype-field-updates :: *field-update* => *field-updates* => *field-updates* (‹-,/ -›)
-datatype-record-update :: *'a* => *field-updates* => *'b* (‹(‹*open-block notation=‹mixfix datatype record update›*›-/(*3*(‖-‖))› [*900*, *0*] *900*)

syntax (*ASCII*)
-datatype-record :: *fields* => *'a* (‹(‹*indent=3 notation=‹mixfix datatype record value›*›'(| - |'))›)
-datatype-record-update :: *'a* => *field-updates* => *'b* (‹(‹*open-block notation=‹mixfix datatype record update›*›-/(*3'*(| - |')))› [*900*, *0*] *900*)

end

named-theorems *datatype-record-update*

ML-file ‹*datatype-records.ML*›
setup ‹*Datatype-Records.setup*›

end

118 Implementation of mappings with Association Lists

theory *AList-Mapping*
imports *AList Mapping*
begin

lift-definition *Mapping* :: (*'a* × *'b*) *list* ⇒ (*'a*, *'b*) *mapping* **is** *map-of* **.**

code-datatype *Mapping*

lemma *lookup-Mapping* [*simp*, *code*]: *Mapping.lookup* (*Mapping xs*) = *map-of xs*
by *transfer rule*

lemma *keys-Mapping* [*simp*, *code*]: *Mapping.keys* (*Mapping xs*) = *set* (*map fst xs*)
by *transfer* (*simp add*: *dom-map-of-conv-image-fst*)

lemma *empty-Mapping* [*code*]: *Mapping.empty* = *Mapping* []
by *transfer simp*

lemma *is-empty-Mapping* [*code*]: *Mapping.is-empty* (*Mapping xs*) ⟷ *List.null xs*
by (*cases xs*) (*simp-all add*: *is-empty-def*)

lemma *update-Mapping* [*code*]: *Mapping.update k v* (*Mapping xs*) = *Mapping* (*AList.update k v xs*)
by *transfer* (*simp add*: *update-conv'*)

lemma *delete-Mapping* [*code*]: *Mapping.delete k* (*Mapping xs*) = *Mapping* (*AList.delete k xs*)
by *transfer* (*simp add*: *delete-conv'*)

lemma *ordered-keys-Mapping* [*code*]:
Mapping.ordered-keys (*Mapping xs*) = *sort* (*remdups* (*map fst xs*))
by (*simp only*: *ordered-keys-def keys-Mapping sorted-list-of-set-sort-remdups*) *simp*

lemma *entries-Mapping* [*code*]:
Mapping.entries (*Mapping xs*) = *set* (*AList.clearjunk xs*)
by *transfer* (*fact graph-map-of*)

lemma *ordered-entries-Mapping* [*code*]:
Mapping.ordered-entries (*Mapping xs*) = *sort-key fst* (*AList.clearjunk xs*)
proof −
have *distinct*: *distinct* (*sort-key fst* (*AList.clearjunk xs*))
using *distinct-clearjunk distinct-map distinct-sort* **by** *blast*
note *folding-Map-graph.idem-if-sorted-distinct*[**where** *?m*=*map-of xs*, *OF - sorted-sort-key distinct*]
then show *?thesis*
unfolding *ordered-entries-def*
by (*transfer fixing*: *xs*) (*auto simp*: *graph-map-of*)
qed

lemma *fold-Mapping* [*code*]:
Mapping.fold f (*Mapping xs*) *a* = *List.fold* (*case-prod f*) (*sort-key fst* (*AList.clearjunk xs*)) *a*
by (*simp add*: *Mapping.fold-def ordered-entries-Mapping*)

lemma *size-Mapping* [*code*]: *Mapping.size* (*Mapping xs*) = *length* (*remdups* (*map*

fst xs))
by (*simp add*: *size-def length-remdups-card-conv dom-map-of-conv-image-fst*)

lemma *tabulate-Mapping* [*code*]: *Mapping.tabulate ks f* = *Mapping* (*map* (λ*k*. (*k*, *f k*)) *ks*)
by *transfer* (*simp add*: *map-of-map-restrict*)

lemma *bulkload-Mapping* [*code*]:
Mapping.bulkload vs = *Mapping* (*map* (λ*n*. (*n*, *vs* ! *n*)) [*0*..<*length vs*])
by *transfer* (*simp add*: *map-of-map-restrict fun-eq-iff*)

lemma *equal-Mapping* [*code*]:
HOL.equal (*Mapping xs*) (*Mapping ys*) ⟷
(*let ks* = *map fst xs*; *ls* = *map fst ys*
in (∀ *l*∈*set ls*. *l* ∈ *set ks*) ∧ (∀ *k*∈*set ks*. *k* ∈ *set ls* ∧ *map-of xs k* = *map-of ys k*))
proof −
have ∗: (*a*, *b*) ∈ *set xs* ⟹ *a* ∈ *fst* ‘ *set xs* **for** *a b xs*
by (*auto simp add*: *image-def intro*!: *bexI*)
show *?thesis*
apply *transfer*
apply (*auto intro*!: *map-of-eqI*)
apply (*auto dest*!: *map-of-eq-dom intro*: ∗)
done
qed

lemma *map-values-Mapping* [*code*]:
Mapping.map-values f (*Mapping xs*) = *Mapping* (*map* (λ(*x*,*y*). (*x*, *f x y*)) *xs*)
for *f* :: *'c* ⇒ *'a* ⇒ *'b* **and** *xs* :: (*'c* × *'a*) *list*
apply *transfer*
apply (*rule ext*)
subgoal for *f xs x* **by** (*induct xs*) *auto*
done

lemma *combine-with-key-code* [*code*]:
Mapping.combine-with-key f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options* (*f x*) (*map-of xs x*) (*map-of ys x*)))
apply *transfer*
apply (*rule ext*)
apply (*rule sym*)
subgoal for *f xs ys x*
apply (*cases map-of xs x*; *cases map-of ys x*; *simp*)
apply (*force simp*: *map-of-eq-None-iff combine-options-def option.the-def o-def image-iff*
dest: *map-of-SomeD split*: *option.splits*)+
done
done

lemma *combine-code* [*code*]:
Mapping.combine f (*Mapping xs*) (*Mapping ys*) =
Mapping.tabulate (*remdups* (*map fst xs* @ *map fst ys*))
(λ*x*. *the* (*combine-options f* (*map-of xs x*) (*map-of ys x*)))
apply *transfer*
apply (*rule ext*)
apply (*rule sym*)
subgoal for *f xs ys x*
apply (*cases map-of xs x*; *cases map-of ys x*; *simp*)
apply (*force simp*: *map-of-eq-None-iff combine-options-def option.the-def o-def image-iff dest*: *map-of-SomeD split*: *option.splits*)+
done
done

lemma *map-of-filter-distinct*:
assumes *distinct* (*map fst xs*)
shows *map-of* (*filter P xs*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some y* ⇒ *if P* (*x*,*y*) *then Some y else None*)
using *assms*
by (*auto simp*: *map-of-eq-None-iff filter-map distinct-map-filter dest*: *map-of-SomeD simp del*: *map-of-eq-Some-iff intro*!: *map-of-is-SomeI split*: *option.splits*)

lemma *filter-Mapping* [*code*]:
Mapping.filter P (*Mapping xs*) = *Mapping* (*filter* (λ(*k*,*v*). *P k v*) (*AList.clearjunk xs*))
apply *transfer*
apply (*rule ext*)
apply (*subst map-of-filter-distinct*)
apply (*simp-all add*: *map-of-clearjunk split*: *option.split*)
done

lemma [*code nbe*]: *HOL.equal* (*x* :: ('*a*, '*b*) *mapping*) *x* ⟷ *True*
by (*fact equal-refl*)

end

theory *Code-Abstract-Char*
imports
Main
HOL−Library.Char-ord
begin

definition *Chr* :: ‹*integer* ⇒ *char*›
where [*simp*]: ‹*Chr* = *char-of*›

lemma *char-of-integer-of-char* [*code abstype*]:
‹*Chr* (*integer-of-char c*) = *c*›
by (*simp add*: *integer-of-char-def*)

lemma *char-of-integer-code* [*code*]:
‹*integer-of-char* (*char-of-integer k*) = (*if* 0 ≤ *k* ∧ *k* < 256 *then k else k mod* 256)›
by (*simp add*: *integer-of-char-def char-of-integer-def integer-eq-iff integer-less-eq-iff integer-less-iff*)

lemma *of-char-code* [*code*]:
‹*of-char c* = *of-nat* (*nat-of-integer* (*integer-of-char c*))›
proof −
have ‹*int-of-integer* (*of-char c*) = *of-char c*›
by (*cases c*) *simp*
then show *?thesis*
by (*simp add*: *integer-of-char-def nat-of-integer-def of-nat-of-char*)
qed

definition *byte* :: ‹*bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *bool* ⇒ *integer*›
where [*simp*]: ‹*byte b0 b1 b2 b3 b4 b5 b6 b7* = *horner-sum of-bool* 2 [*b0*, *b1*, *b2*, *b3*, *b4*, *b5*, *b6*, *b7*]›

lemma *byte-code* [*code*]:
‹*byte b0 b1 b2 b3 b4 b5 b6 b7* = (
let
s0 = *if b0 then* 1 *else* 0;
s1 = *if b1 then s0* + 2 *else s0*;
s2 = *if b2 then s1* + 4 *else s1*;
s3 = *if b3 then s2* + 8 *else s2*;
s4 = *if b4 then s3* + 16 *else s3*;
s5 = *if b5 then s4* + 32 *else s4*;
s6 = *if b6 then s5* + 64 *else s5*;
s7 = *if b7 then s6* + 128 *else s6*
in s7)›
by *simp*

lemma *Char-code* [*code*]:
‹*integer-of-char* (*Char b0 b1 b2 b3 b4 b5 b6 b7*) = *byte b0 b1 b2 b3 b4 b5 b6 b7*›
by (*simp add*: *integer-of-char-def*)

lemma *digit-0-code* [*code*]:
‹*digit0 c* ⟷ *bit* (*integer-of-char c*) 0›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-1-code* [*code*]:
‹*digit1 c* ⟷ *bit* (*integer-of-char c*) 1›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-2-code* [*code*]:
‹*digit2 c* ⟷ *bit* (*integer-of-char c*) *2*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-3-code* [*code*]:
‹*digit3 c* ⟷ *bit* (*integer-of-char c*) *3*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-4-code* [*code*]:
‹*digit4 c* ⟷ *bit* (*integer-of-char c*) *4*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-5-code* [*code*]:
‹*digit5 c* ⟷ *bit* (*integer-of-char c*) *5*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-6-code* [*code*]:
‹*digit6 c* ⟷ *bit* (*integer-of-char c*) *6*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *digit-7-code* [*code*]:
‹*digit7 c* ⟷ *bit* (*integer-of-char c*) *7*›
by (*cases c*) (*simp add*: *integer-of-char-def*)

lemma *case-char-code* [*code*]:
‹*case-char f c* = *f* (*digit0 c*) (*digit1 c*) (*digit2 c*) (*digit3 c*) (*digit4 c*) (*digit5 c*) (*digit6 c*) (*digit7 c*)›
by (*fact char.case-eq-if*)

lemma *rec-char-code* [*code*]:
‹*rec-char f c* = *f* (*digit0 c*) (*digit1 c*) (*digit2 c*) (*digit3 c*) (*digit4 c*) (*digit5 c*) (*digit6 c*) (*digit7 c*)›
by (*cases c*) *simp*

lemma *char-of-code* [*code*]:
‹*integer-of-char* (*char-of a*) =
byte (*bit a 0*) (*bit a 1*) (*bit a 2*) (*bit a 3*) (*bit a 4*) (*bit a 5*) (*bit a 6*) (*bit a 7*)›
by (*simp add*: *char-of-def integer-of-char-def*)

lemma *ascii-of-code* [*code*]:
‹*integer-of-char* (*String.ascii-of c*) = (*let k* = *integer-of-char c in if k* < *128 then k else k* − *128*)›
proof (*cases* ‹*of-char c* < (*128* :: *integer*)›)
case *True*
moreover have ‹(*of-nat 0* :: *integer*) ≤ *of-nat* (*of-char c*)›
by *simp*
then have ‹(*0* :: *integer*) ≤ *of-char c*›
by (*simp only*: *of-nat-0 of-nat-of-char*)

ultimately show *?thesis*
by (*simp add*: *Let-def integer-of-char-def take-bit-eq-mod integer-eq-iff integer-less-eq-iff integer-less-iff*)
next
case *False*
then have ‹(*128* :: *integer*) ≤ *of-char c*›
by *simp*
moreover have ‹*of-nat* (*of-char c*) < (*of-nat 256* :: *integer*)›
by (*simp only*: *of-nat-less-iff*) *simp*
then have ‹*of-char c* < (*256* :: *integer*)›
by (*simp add*: *of-nat-of-char*)
moreover define *k* :: *integer* **where** ‹*k* = *of-char c* − *128*›
then have ‹*of-char c* = *k* + *128*›
by *simp*
ultimately show *?thesis*
by (*simp add*: *Let-def integer-of-char-def take-bit-eq-mod integer-eq-iff integer-less-eq-iff integer-less-iff*)
qed

lemma *equal-char-code* [*code*]:
‹*HOL.equal c d* ⟷ *integer-of-char c* = *integer-of-char d*›
by (*simp add*: *integer-of-char-def equal*)

lemma *less-eq-char-code* [*code*]:
‹*c* ≤ *d* ⟷ *integer-of-char c* ≤ *integer-of-char d*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?P* ⟷ *of-nat* (*of-char c*) ≤ (*of-nat* (*of-char d*) :: *integer*)›
by (*simp add*: *less-eq-char-def*)
also have ‹... ⟷ *?Q*›
by (*simp add*: *of-nat-of-char integer-of-char-def*)
finally show *?thesis* **.**
qed

lemma *less-char-code* [*code*]:
‹*c* < *d* ⟷ *integer-of-char c* < *integer-of-char d*› (**is** ‹*?P* ⟷ *?Q*›)
proof −
have ‹*?P* ⟷ *of-nat* (*of-char c*) < (*of-nat* (*of-char d*) :: *integer*)›
by (*simp add*: *less-char-def*)
also have ‹... ⟷ *?Q*›
by (*simp add*: *of-nat-of-char integer-of-char-def*)
finally show *?thesis* **.**
qed

lemma *absdef-simps*:
‹*horner-sum of-bool 2* [] = (*0* :: *integer*)›
‹*horner-sum of-bool 2* (*False* # *bs*) = (*0* :: *integer*) ⟷ *horner-sum of-bool 2 bs* = (*0* :: *integer*)›
‹*horner-sum of-bool 2* (*True* # *bs*) = (*1* :: *integer*) ⟷ *horner-sum of-bool 2 bs* = (*0* :: *integer*)›

‹*horner-sum of-bool 2 (False # bs) = (numeral (Num.Bit0 n) :: integer) ⟷ horner-sum of-bool 2 bs = (numeral n :: integer)*›
‹*horner-sum of-bool 2 (True # bs) = (numeral (Num.Bit1 n) :: integer) ⟷ horner-sum of-bool 2 bs = (numeral n :: integer)*›
by *auto* (*auto simp only*: *numeral-Bit0* [*of n*] *numeral-Bit1* [*of n*] *mult-2* [*symmetric*] *add.commute* [*of - 1*] *add.left-cancel mult-cancel-left*)

local-setup ‹
let
val simps = @{*thms absdef-simps integer-of-char-def of-char-Char numeral-One*}
fun prove-eqn lthy n lhs def-eqn =
let
val eqn = (*HOLogic.mk-Trueprop o HOLogic.mk-eq*)
(***term*** ‹*integer-of-char*› $ *lhs*, *HOLogic.mk-number* ***typ*** ‹*integer*› *n*)
in
Goal.prove-future lthy [] [] *eqn* (*fn* {*context* = *ctxt*, ...} =>
unfold-tac ctxt (*def-eqn* :: *simps*))
end
fun define n =
let
val s = *Char-* ^ *String-Syntax.hex n*;
val b = *Binding.name s*;
val b-def = *Thm.def-binding b*;
val b-code = *Binding.name* (*s* ^ *-code*);
in
Local-Theory.define ((*b*, *Mixfix.NoSyn*),
((*Binding.empty*, []), *HOLogic.mk-char n*))
#-> (*fn* (*lhs*, (-, *raw-def-eqn*)) =>
Local-Theory.note ((*b-def*, @{*attributes* [*code-abbrev*]}), [*HOLogic.mk-obj-eq raw-def-eqn*])
#-> (*fn* (-, [*def-eqn*]) => '(*fn lthy* => *prove-eqn lthy n lhs def-eqn*))
#-> (*fn raw-code-eqn* => *Local-Theory.note* ((*b-code*, []), [*raw-code-eqn*]))
#-> (*fn* (-, [*code-eqn*]) => *Code.declare-abstract-eqn code-eqn*))
end
in
fold define (*0 upto 255*)
end
›

code-identifier
code-module *Code-Abstract-Char* ⇀
(*SML*) *Str* **and** (*OCaml*) *Str* **and** (*Haskell*) *Str* **and** (*Scala*) *Str*

end

119 Avoidance of pattern matching on natural numbers

theory *Code-Abstract-Nat*
imports *Main*
begin

When natural numbers are implemented in another than the conventional inductive *0*/*Suc* representation, it is necessary to avoid all pattern matching on natural numbers altogether. This is accomplished by this theory (up to a certain extent).

119.1 Case analysis

Case analysis on natural numbers is rephrased using a conditional expression:

lemma [*code*, *code-unfold*]:
  case-nat = (λ*f g n*. *if n* = *0 then f else g* (*n* − *1*))
  by (*auto simp add*: *fun-eq-iff dest*!: *gr0-implies-Suc*)

119.2 Preprocessors

The term *Suc n* is no longer a valid pattern. Therefore, all occurrences of this term in a position where a pattern is expected (i.e. on the left-hand side of a code equation) must be eliminated. This can be accomplished – as far as possible – by applying the following transformation rule:

lemma *Suc-if-eq*:
  assumes ⋀*n*. *f* (*Suc n*) ≡ *h n*
  assumes *f 0* ≡ *g*
  shows *f n* ≡ *if n* = *0 then g else h* (*n* − *1*)
  by (*rule eq-reflection*) (*cases n*, *insert assms*, *simp-all*)

The rule above is built into a preprocessor that is plugged into the code generator.

setup ‹
let

val Suc-if-eq = *Thm.incr-indexes 1* @{*thm Suc-if-eq*};

fun remove-suc ctxt thms =
  let
    val vname = *singleton* (*Name.variant-list* (*map fst*
      (*fold* (*Term.add-var-names o Thm.full-prop-of*) *thms* [])) *n*;
    val cv = *Thm.cterm-of ctxt* (*Var* ((*vname*, *0*), *HOLogic.natT*));
    val lhs-of = *Thm.dest-arg1 o Thm.cprop-of*;
    val rhs-of = *Thm.dest-arg o Thm.cprop-of*;
    fun find-vars ct = (*case Thm.term-of ct of*

```
      (Const (const-name ‹Suc›, -) $ Var -) => [(cv, snd (Thm.dest-comb ct))]
    | - $ - =>
      let val (ct1, ct2) = Thm.dest-comb ct
      in
        map (apfst (fn ct => Thm.apply ct ct2)) (find-vars ct1) @
        map (apfst (Thm.apply ct1)) (find-vars ct2)
      end
    | - => []);
  val eqs = maps
    (fn thm => map (pair thm) (find-vars (lhs-of thm))) thms;
  fun mk-thms (thm, (ct, cv')) =
    let
      val thm' =
        Thm.implies-elim
        (Conv.fconv-rule (Thm.beta-conversion true)
          (Thm.instantiate'
            [SOME (Thm.ctyp-of-cterm ct)] [SOME (Thm.lambda cv ct),
              SOME (Thm.lambda cv' (rhs-of thm)), NONE, SOME cv']
            Suc-if-eq)) (Thm.forall-intr cv' thm)
    in
      case map-filter (fn thm'' =>
          SOME (thm'', singleton
            (Variable.trade (K (fn [thm'''] => [thm''' RS thm'])))
              (Variable.declare-thm thm'' ctxt)) thm'')
        handle THM - => NONE) thms of
          [] => NONE
        | thmps =>
            let val (thms1, thms2) = split-list thmps
            in SOME (subtract Thm.eq-thm (thm :: thms1) thms @ thms2) end
    end
in get-first mk-thms eqs end;

fun eqn-suc-base-preproc ctxt thms =
  let
    val dest = fst o Logic.dest-equals o Thm.prop-of;
    val contains-suc = exists-Const (fn (c, -) => c = const-name ‹Suc›);
  in
    if forall (can dest) thms andalso exists (contains-suc o dest) thms
    then thms |> perhaps-loop (remove-suc ctxt) |> (Option.map o map) Drule.zero-var-indexes
      else NONE
  end;

val eqn-suc-preproc = Code-Preproc.simple-functrans eqn-suc-base-preproc;

in

  Code-Preproc.add-functrans (eqn-Suc, eqn-suc-preproc)

end
```

›

119.3 Candidates which need special treatment

lemma *drop-bit-int-code* [*code*]:
‹*drop-bit n k = k div 2 ^ n*› **for** *k :: int*
by (*fact drop-bit-eq-div*)

lemma *take-bit-num-code* [*code*]:
‹*take-bit-num n Num.One =*
(case n of 0 ⇒ None | Suc n ⇒ Some Num.One)›
‹*take-bit-num n (Num.Bit0 m) =*
(case n of 0 ⇒ None | Suc n ⇒ (case take-bit-num n m of None ⇒ None | Some q ⇒ Some (Num.Bit0 q)))›
‹*take-bit-num n (Num.Bit1 m) =*
(case n of 0 ⇒ None | Suc n ⇒ Some (case take-bit-num n m of None ⇒ Num.One | Some q ⇒ Num.Bit1 q))›
by (*cases n*; *simp*)+

end

120 Implementation of natural numbers as binary numerals

theory *Code-Binary-Nat*
imports *Code-Abstract-Nat*
begin

When generating code for functions on natural numbers, the canonical representation using *0* and *Suc* is unsuitable for computations involving large numbers. This theory refines the representation of natural numbers for code generation to use binary numerals, which do not grow linear in size but logarithmic.

120.1 Representation

code-datatype *0*::*nat nat-of-num*

lemma [*code*]:
num-of-nat 0 = Num.One
num-of-nat (nat-of-num k) = k
by (*simp-all add*: *nat-of-num-inverse*)

lemma [*code*]:
(*1*::*nat*) = *Numeral1*
by *simp*

lemma [*code-abbrev*]: *Numeral1 = (1*::*nat*)

by *simp*

lemma [*code*]:
Suc n = n + 1
by *simp*

120.2 Basic arithmetic

context
begin

lemma *plus-nat-code* [*code*]:
0 + n = (n::nat)
m + 0 = (m::nat)
nat-of-num k + nat-of-num l = nat-of-num (k + l)
by (*simp-all add*: *nat-of-num-numeral*)

Bounded subtraction needs some auxiliary

qualified definition *dup* :: *nat ⇒ nat* **where**
dup n = n + n

lemma *dup-code* [*code*]:
dup 0 = 0
dup (nat-of-num k) = nat-of-num (Num.Bit0 k)
by (*simp-all add*: *dup-def numeral-Bit0*)

qualified definition *sub* :: *num ⇒ num ⇒ nat option* **where**
sub k l = (if k ≥ l then Some (numeral k − numeral l) else None)

lemma *sub-code* [*code*]:
sub Num.One Num.One = Some 0
sub (Num.Bit0 m) Num.One = Some (nat-of-num (Num.BitM m))
sub (Num.Bit1 m) Num.One = Some (nat-of-num (Num.Bit0 m))
sub Num.One (Num.Bit0 n) = None
sub Num.One (Num.Bit1 n) = None
sub (Num.Bit0 m) (Num.Bit0 n) = map-option dup (sub m n)
sub (Num.Bit1 m) (Num.Bit1 n) = map-option dup (sub m n)
sub (Num.Bit1 m) (Num.Bit0 n) = map-option (λq. dup q + 1) (sub m n)
sub (Num.Bit0 m) (Num.Bit1 n) = (case sub m n of None ⇒ None
| Some q ⇒ if q = 0 then None else Some (dup q − 1))
by (*auto simp*: *nat-of-num-numeral Num.dbl-def Num.dbl-inc-def Num.dbl-dec-def*
Let-def le-imp-diff-is-add BitM-plus-one sub-def dup-def
sub-non-positive nat-add-distrib sub-non-negative)

lemma *minus-nat-code* [*code*]:
0 − n = (0::nat)
m − 0 = (m::nat)
nat-of-num k − nat-of-num l = (case sub k l of None ⇒ 0 | Some j ⇒ j)
by (*simp-all add*: *nat-of-num-numeral sub-non-positive sub-def*)

lemma *times-nat-code* [*code*]:
*0 * n = (0::nat)*
*m * 0 = (0::nat)*
*nat-of-num k * nat-of-num l = nat-of-num (k * l)*
by (*simp-all add*: *nat-of-num-numeral*)

lemma *equal-nat-code* [*code*]:
HOL.equal 0 (0::nat) ⟷ True
HOL.equal 0 (nat-of-num l) ⟷ False
HOL.equal (nat-of-num k) 0 ⟷ False
HOL.equal (nat-of-num k) (nat-of-num l) ⟷ HOL.equal k l
by (*simp-all add*: *nat-of-num-numeral equal*)

lemma *equal-nat-refl* [*code nbe*]:
HOL.equal (n::nat) n ⟷ True
by (*rule equal-refl*)

lemma *less-eq-nat-code* [*code*]:
0 ≤ (n::nat) ⟷ True
nat-of-num k ≤ 0 ⟷ False
nat-of-num k ≤ nat-of-num l ⟷ k ≤ l
by (*simp-all add*: *nat-of-num-numeral*)

lemma *less-nat-code* [*code*]:
(m::nat) < 0 ⟷ False
0 < nat-of-num l ⟷ True
nat-of-num k < nat-of-num l ⟷ k < l
by (*simp-all add*: *nat-of-num-numeral*)

lemma *divmod-nat-code* [*code*]:
Euclidean-Rings.divmod-nat 0 n = (0, 0)
Euclidean-Rings.divmod-nat m 0 = (0, m)
Euclidean-Rings.divmod-nat (nat-of-num k) (nat-of-num l) = divmod k l
by (*simp-all add*: *Euclidean-Rings.divmod-nat-def nat-of-num-numeral*)

end

120.3 Conversions

lemma *of-nat-code* [*code*]:
of-nat 0 = 0
of-nat (nat-of-num k) = numeral k
by (*simp-all add*: *nat-of-num-numeral*)

code-identifier
code-module *Code-Binary-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

121 Code generation of prolog programs

theory *Code-Prolog*
imports *Main*
keywords *values-prolog* :: *diag*
begin

ML-file ‹~~/*src*/*HOL*/*Tools*/*Predicate-Compile*/*code-prolog.ML*›

122 Setup for Numerals

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*numeral*›]›

setup ‹*Predicate-Compile-Data.keep-functions* [***const-name*** ‹*numeral*›]›

end

123 Implementation of integer numbers by target-language integers

theory *Code-Target-Int*
imports *Main*
begin

code-datatype *int-of-integer*

context
includes *integer.lifting*
begin

lemma [*code*]:
integer-of-int (*int-of-integer k*) = *k*
by *transfer rule*

lemma [*code*]:
Int.Pos = *int-of-integer* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code*]:
Int.Neg = *int-of-integer* ∘ *uminus* ∘ *integer-of-num*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code-abbrev*]:
int-of-integer (*numeral k*) = *Int.Pos k*
by *transfer simp*

lemma [*code-abbrev*]:
int-of-integer (− *numeral k*) = *Int.Neg k*
by *transfer simp*

context
begin

qualified definition *positive* :: *num* ⇒ *int*
where [*simp*]: *positive* = *numeral*

qualified definition *negative* :: *num* ⇒ *int*
where [*simp*]: *negative* = *uminus* ∘ *numeral*

lemma [*code-computation-unfold*]:
numeral = *positive*
Int.Pos = *positive*
Int.Neg = *negative*
by (*simp-all add*: *fun-eq-iff*)

end

lemma [*code*, *symmetric*, *code-post*]:
0 = *int-of-integer 0*
by *transfer simp*

lemma [*code*, *symmetric*, *code-post*]:
1 = *int-of-integer 1*
by *transfer simp*

lemma [*code-post*]:
int-of-integer (− *1*) = − *1*
by *simp*

lemma [*code*]:
k + *l* = *int-of-integer* (*of-int k* + *of-int l*)
by *transfer simp*

lemma [*code*]:
− *k* = *int-of-integer* (− *of-int k*)
by *transfer simp*

lemma [*code*]:
k − *l* = *int-of-integer* (*of-int k* − *of-int l*)
by *transfer simp*

lemma [*code*]:
Int.dup k = *int-of-integer* (*Code-Numeral.dup* (*of-int k*))
by *transfer simp*

declare [[*code drop*: *Int.sub*]]

lemma [*code*]:
k ∗ *l* = *int-of-integer* (*of-int k* ∗ *of-int l*)
by *simp*

lemma [*code*]:
k div l = *int-of-integer* (*of-int k div of-int l*)
by *simp*

lemma [*code*]:
k mod l = *int-of-integer* (*of-int k mod of-int l*)
by *simp*

lemma [*code*]:
divmod m n = *map-prod int-of-integer int-of-integer* (*divmod m n*)
unfolding *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*
by *transfer simp*

lemma [*code*]:
HOL.equal k l = *HOL.equal* (*of-int k* :: *integer*) (*of-int l*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
k ≤ *l* ⟷ (*of-int k* :: *integer*) ≤ *of-int l*
by *transfer rule*

lemma [*code*]:
k < *l* ⟷ (*of-int k* :: *integer*) < *of-int l*
by *transfer rule*

lemma *gcd-int-of-integer* [*code*]:
gcd (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*gcd x y*)
by *transfer rule*

lemma *lcm-int-of-integer* [*code*]:
lcm (*int-of-integer x*) (*int-of-integer y*) = *int-of-integer* (*lcm x y*)
by *transfer rule*

end

lemma (**in** *ring-1*) *of-int-code-if*:
of-int k = (*if k* = *0 then 0*
else if k < *0 then* − *of-int* (− *k*)
else let
l = *2* ∗ *of-int* (*k div 2*);
j = *k mod 2*
in if j = *0 then l else l* + *1*)

proof −
from *div-mult-mod-eq* **have** ∗: *of-int k* = *of-int* (*k div 2* ∗ *2* + *k mod 2*) **by** *simp*
show *?thesis*
by (*simp add*: *Let-def of-int-add* [*symmetric*]) (*simp add*: ∗ *mult.commute*)
qed

declare *of-int-code-if* [*code*]

lemma [*code*]:
nat = *nat-of-integer* ∘ *of-int*
including *integer.lifting* **by** *transfer* (*simp add*: *fun-eq-iff*)

definition *char-of-int* :: *int* ⇒ *char*
where [*code-abbrev*]: *char-of-int* = *char-of*

definition *int-of-char* :: *char* ⇒ *int*
where [*code-abbrev*]: *int-of-char* = *of-char*

lemma [*code*]:
char-of-int = *char-of-integer* ∘ *integer-of-int*
including *integer.lifting* **unfolding** *char-of-integer-def char-of-int-def*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code*]:
int-of-char = *int-of-integer* ∘ *integer-of-char*
including *integer.lifting* **unfolding** *integer-of-char-def int-of-char-def*
by *transfer* (*simp add*: *fun-eq-iff*)

context
includes *integer.lifting* **and** *bit-operations-syntax*
begin

lemma [*code*]:
‹*bit* (*int-of-integer k*) *n* ⟷ *bit k n*›
by *transfer rule*

lemma [*code*]:
‹*NOT* (*int-of-integer k*) = *int-of-integer* (*NOT k*)›
by *transfer rule*

lemma [*code*]:
‹*int-of-integer k AND int-of-integer l* = *int-of-integer* (*k AND l*)›
by *transfer rule*

lemma [*code*]:
‹*int-of-integer k OR int-of-integer l* = *int-of-integer* (*k OR l*)›
by *transfer rule*

lemma [*code*]:

‹*int-of-integer k XOR int-of-integer l = int-of-integer (k XOR l)*›
by *transfer rule*

lemma [*code*]:
‹*push-bit n (int-of-integer k) = int-of-integer (push-bit n k)*›
by *transfer rule*

lemma [*code*]:
‹*drop-bit n (int-of-integer k) = int-of-integer (drop-bit n k)*›
by *transfer rule*

lemma [*code*]:
‹*take-bit n (int-of-integer k) = int-of-integer (take-bit n k)*›
by *transfer rule*

lemma [*code*]:
‹*mask n = int-of-integer (mask n)*›
by *transfer rule*

lemma [*code*]:
‹*set-bit n (int-of-integer k) = int-of-integer (set-bit n k)*›
by *transfer rule*

lemma [*code*]:
‹*unset-bit n (int-of-integer k) = int-of-integer (unset-bit n k)*›
by *transfer rule*

lemma [*code*]:
‹*flip-bit n (int-of-integer k) = int-of-integer (flip-bit n k)*›
by *transfer rule*

end

code-identifier
code-module *Code-Target-Int* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

theory *Code-Real-Approx-By-Float*
imports *Complex-Main Code-Target-Int*
begin

WARNING! This theory implements mathematical reals by machine reals (floats). This is inconsistent. See the proof of False at the end of the theory, where an equality on mathematical reals is (incorrectly) disproved by mapping it to machine reals.

The **value** command cannot display real results yet.

The only legitimate use of this theory is as a tool for code generation purposes.

context
begin

qualified definition *real-of-integer* :: *integer* ⇒ *real*
  where [*code-abbrev*]: *real-of-integer* = *of-int* ∘ *int-of-integer*

end

code-datatype *Code-Real-Approx-By-Float.real-of-integer* ‹(/) :: *real* ⇒ *real* ⇒ *real*›

lemma [*code-unfold del*]: *numeral k* ≡ *real-of-rat* (*numeral k*)
  by *simp*

lemma [*code-unfold del*]: − *numeral k* ≡ *real-of-rat* (− *numeral k*)
  by *simp*

context
begin

qualified definition *real-of-int* :: ‹*int* ⇒ *real*›
  where [*code-abbrev*]: ‹*real-of-int* = *of-int*›

lemma [*code*]: *real-of-int* = *Code-Real-Approx-By-Float.real-of-integer* ∘ *integer-of-int*
  by (*simp add*: *fun-eq-iff Code-Real-Approx-By-Float.real-of-integer-def real-of-int-def*)

qualified definition *exp-real* :: ‹*real* ⇒ *real*›
  where [*code-abbrev*, *code drop*]: ‹*exp-real* = *exp*›

qualified definition *sin-real* :: ‹*real* ⇒ *real*›
  where [*code-abbrev*, *code drop*]: ‹*sin-real* = *sin*›

qualified definition *cos-real* :: ‹*real* ⇒ *real*›
  where [*code-abbrev*, *code drop*]: ‹*cos-real* = *cos*›

qualified definition *tan-real* :: ‹*real* ⇒ *real*›
  where [*code-abbrev*, *code drop*]: ‹*tan-real* = *tan*›

end

lemma [*code*]: ‹*r* − *s* = *r* + (− *s*)› **for** *r s* :: *real*
  by (*fact diff-conv-add-uminus*)

lemma [*code*]: ‹*inverse r* = *1* / *r*› **for** *r* :: *real*
  by (*fact inverse-eq-divide*)

lemma [*code*]: ‹*Ratreal r* = (*let* (*p*, *q*) = *quotient-of r in real-of-int p* / *real-of-int*

```
q)›
  by (cases r) (simp add: quotient-of-Fract of-rat-rat)

declare [[code drop:
  ‹HOL.equal :: real ⇒ real ⇒ bool›
  ‹(≤) :: real ⇒ real ⇒ bool›
  ‹(<) :: real ⇒ real ⇒ bool›
  ‹(+) :: real ⇒ real ⇒ real›
  ‹uminus :: real ⇒ real›
  ‹(*) :: real ⇒ real ⇒ real›
  sqrt
  ‹ln :: real ⇒ real›
  pi
  arcsin
  arccos
  arctan]]

code-reserved (SML) Real

code-printing
  type-constructor real ⇀
    (SML) real
    and (OCaml) float
    and (Haskell) Prelude.Double
| constant 0 :: real ⇀
    (SML) 0.0
    and (OCaml) 0.0
    and (Haskell) 0.0
| constant 1 :: real ⇀
    (SML) 1.0
    and (OCaml) 1.0
    and (Haskell) 1.0
| constant HOL.equal :: real ⇒ real ⇒ bool ⇀
    (SML) Real.== ((-), (-))
    and (OCaml) Pervasives.(=)
    and (Haskell) infix 4 ==
| class-instance real :: HOL.equal => (Haskell) -
| constant (≤) :: real ⇒ real ⇒ bool ⇀
    (SML) Real.<= ((-), (-))
    and (OCaml) Pervasives.(<=)
    and (Haskell) infix 4 <=
| constant (<) :: real ⇒ real ⇒ bool ⇀
    (SML) Real.< ((-), (-))
    and (OCaml) Pervasives.(<)
    and (Haskell) infix 4 <
| constant (+) :: real ⇒ real ⇒ real ⇀
    (SML) Real.+ ((-), (-))
    and (OCaml) Pervasives.( +. )
    and (Haskell) infixl 6 +
```

| **constant** (∗) :: *real* ⇒ *real* ⇒ *real* ⇀
  (*SML*) *Real.∗* ((-), (-))
  and (*Haskell*) **infixl** 7 ∗
| **constant** *uminus* :: *real* ⇒ *real* ⇀
  (*SML*) *Real.~*
  and (*OCaml*) *Pervasives.*(~−.)
  and (*Haskell*) *negate*
| **constant** (−) :: *real* ⇒ *real* ⇒ *real* ⇀
  (*SML*) *Real.−* ((-), (-))
  and (*OCaml*) *Pervasives.*(−.)
  and (*Haskell*) **infixl** 6 −
| **constant** (/) :: *real* ⇒ *real* ⇒ *real* ⇀
  (*SML*) *Real.'/* ((-), (-))
  and (*OCaml*) *Pervasives.*('/.)
  and (*Haskell*) **infixl** 7 /
| **constant** *sqrt* :: *real* ⇒ *real* ⇀
  (*SML*) *Math.sqrt*
  and (*OCaml*) *Pervasives.sqrt*
  and (*Haskell*) *Prelude.sqrt*
| **constant** *Code-Real-Approx-By-Float.exp-real* ⇀
  (*SML*) *Math.exp*
  and (*OCaml*) *Pervasives.exp*
  and (*Haskell*) *Prelude.exp*
| **constant** *ln* ⇀
  (*SML*) *Math.ln*
  and (*OCaml*) *Pervasives.ln*
  and (*Haskell*) *Prelude.log*
| **constant** *Code-Real-Approx-By-Float.sin-real* ⇀
  (*SML*) *Math.sin*
  and (*OCaml*) *Pervasives.sin*
  and (*Haskell*) *Prelude.sin*
| **constant** *Code-Real-Approx-By-Float.cos-real* ⇀
  (*SML*) *Math.cos*
  and (*OCaml*) *Pervasives.cos*
  and (*Haskell*) *Prelude.cos*
| **constant** *Code-Real-Approx-By-Float.tan-real* ⇀
  (*SML*) *Math.tan*
  and (*OCaml*) *Pervasives.tan*
  and (*Haskell*) *Prelude.tan*
| **constant** *pi* ⇀
  (*SML*) *Math.pi*

  and (*Haskell*) *Prelude.pi*
| **constant** *arcsin* ⇀
  (*SML*) *Math.asin*
  and (*OCaml*) *Pervasives.asin*
  and (*Haskell*) *Prelude.asin*
| **constant** *arccos* ⇀
  (*SML*) *Math.scos*

and (*OCaml*) *Pervasives.acos*
and (*Haskell*) *Prelude.acos*
| **constant** *arctan* ⇀
(*SML*) *Math.atan*
and (*OCaml*) *Pervasives.atan*
and (*Haskell*) *Prelude.atan*
| **constant** *Code-Real-Approx-By-Float.real-of-integer* ⇀
(*SML*) *Real.fromInt*
and (*OCaml*) *Pervasives.float/* (*Big'-int.to'-int* (-))
and (*Haskell*) *Prelude.fromIntegral* (-)

notepad
begin
have *cos* (*pi/2*) = *0* **by** (*rule cos-pi-half*)
moreover have *cos* (*pi/2*) ≠ *0* **by** *eval*
ultimately have *False* **by** *blast*
end

end

124 Implementation of natural numbers by target-language integers

theory *Code-Target-Nat*
imports *Code-Abstract-Nat*
begin

124.1 Implementation for *nat*

context
includes *integer.lifting*
begin

lift-definition *Nat* :: *integer* ⇒ *nat*
is *nat*
.

lemma [*code-post*]:
Nat 0 = *0*
Nat 1 = *1*
Nat (*numeral k*) = *numeral k*
by (*transfer*, *simp*)+

lemma [*code-abbrev*]:
integer-of-nat = *of-nat*
by *transfer rule*

lemma [*code-unfold*]:

Int.nat (*int-of-integer k*) = *nat-of-integer k*
by *transfer rule*

lemma [*code abstype*]:
Code-Target-Nat.Nat (*integer-of-nat n*) = *n*
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (*nat-of-integer k*) = *max 0 k*
by *transfer auto*

lemma [*code-abbrev*]:
nat-of-integer (*numeral k*) = *nat-of-num k*
by *transfer* (*simp add*: *nat-of-num-numeral*)

context
begin

qualified definition *natural* :: *num* ⇒ *nat*
where [*simp*]: *natural* = *nat-of-num*

lemma [*code-computation-unfold*]:
numeral = *natural*
nat-of-num = *natural*
by (*simp-all add*: *nat-of-num-numeral*)

end

lemma [*code abstract*]:
integer-of-nat (*nat-of-num n*) = *integer-of-num n*
by (*simp add*: *nat-of-num-numeral integer-of-nat-numeral*)

lemma [*code abstract*]:
integer-of-nat 0 = *0*
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat 1 = *1*
by *transfer simp*

lemma [*code*]:
Suc n = *n* + *1*
by *simp*

lemma [*code abstract*]:
integer-of-nat (*m* + *n*) = *of-nat m* + *of-nat n*
by *transfer simp*

lemma [*code abstract*]:

integer-of-nat (*m* − *n*) = *max* 0 (*of-nat m* − *of-nat n*)
by *transfer simp*

lemma [*code abstract*]:
integer-of-nat (*m* ∗ *n*) = *of-nat m* ∗ *of-nat n*
by *transfer* (*simp add*: *of-nat-mult*)

lemma [*code abstract*]:
integer-of-nat (*m div n*) = *of-nat m div of-nat n*
by *transfer* (*simp add*: *zdiv-int*)

lemma [*code abstract*]:
integer-of-nat (*m mod n*) = *of-nat m mod of-nat n*
by *transfer* (*simp add*: *zmod-int*)

context
includes *integer.lifting*
begin

lemma *divmod-nat-code* [*code*]:
Euclidean-Rings.divmod-nat m n = (
let k = *integer-of-nat m*; *l* = *integer-of-nat n*
in map-prod nat-of-integer nat-of-integer
(*if k* = *0 then* (*0*, *0*)
else if l = *0 then* (*0*, *k*) *else*
Code-Numeral.divmod-abs k l))
by (*simp add*: *prod-eq-iff Let-def Euclidean-Rings.divmod-nat-def*; *transfer*)
(*simp add*: *nat-div-distrib nat-mod-distrib*)

end

lemma [*code*]:
divmod m n = *map-prod nat-of-integer nat-of-integer* (*divmod m n*)
by (*simp only*: *prod-eq-iff divmod-def map-prod-def case-prod-beta fst-conv snd-conv*; *transfer*)
(*simp-all only*: *nat-div-distrib nat-mod-distrib*
zero-le-numeral nat-numeral)

lemma [*code*]:
HOL.equal m n = *HOL.equal* (*of-nat m* :: *integer*) (*of-nat n*)
by *transfer* (*simp add*: *equal*)

lemma [*code*]:
m ≤ *n* ⟷ (*of-nat m* :: *integer*) ≤ *of-nat n*
by *simp*

lemma [*code*]:
m < *n* ⟷ (*of-nat m* :: *integer*) < *of-nat n*
by *simp*

lemma *num-of-nat-code* [*code*]:
num-of-nat = *num-of-integer* ∘ *of-nat*
by *transfer* (*simp add*: *fun-eq-iff*)

end

lemma (**in** *semiring-1*) *of-nat-code-if*:
of-nat n = (*if n* = *0 then 0*
else let
(*m*, *q*) = *Euclidean-Rings.divmod-nat n 2*;
m' = *2* ∗ *of-nat m*
in if q = *0 then m' else m'* + *1*)
by (*cases n*)
(*simp-all add*: *Let-def Euclidean-Rings.divmod-nat-def ac-simps*
flip: *of-nat-numeral of-nat-mult minus-mod-eq-mult-div*)

declare *of-nat-code-if* [*code*]

definition *int-of-nat* :: *nat* ⇒ *int* **where**
[*code-abbrev*]: *int-of-nat* = *of-nat*

lemma [*code*]:
int-of-nat n = *int-of-integer* (*of-nat n*)
by (*simp add*: *int-of-nat-def*)

lemma [*code abstract*]:
integer-of-nat (*nat k*) = *max 0* (*integer-of-int k*)
including *integer.lifting* **by** *transfer auto*

definition *char-of-nat* :: *nat* ⇒ *char*
where [*code-abbrev*]: *char-of-nat* = *char-of*

definition *nat-of-char* :: *char* ⇒ *nat*
where [*code-abbrev*]: *nat-of-char* = *of-char*

lemma [*code*]:
char-of-nat = *char-of-integer* ∘ *integer-of-nat*
including *integer.lifting* **unfolding** *char-of-integer-def char-of-nat-def*
by *transfer* (*simp add*: *fun-eq-iff*)

lemma [*code abstract*]:
integer-of-nat (*nat-of-char c*) = *integer-of-char c*
by (*cases c*) (*simp add*: *nat-of-char-def integer-of-char-def integer-of-nat-eq-of-nat*)

lemma *term-of-nat-code* [*code*]:
— Use *nat-of-integer* in term reconstruction instead of *Code-Target-Nat.Nat* such that reconstructed terms can be fed back to the code generator
term-of-class.term-of n =

```
  Code-Evaluation.App
   (Code-Evaluation.Const (STR ''Code-Numeral.nat-of-integer'')
     (typerep.Typerep (STR ''fun'')
       [typerep.Typerep (STR ''Code-Numeral.integer'') [],
        typerep.Typerep (STR ''Nat.nat'') []]))
   (term-of-class.term-of (integer-of-nat n))
 by (simp add: term-of-anything)

lemma nat-of-integer-code-post [code-post]:
  nat-of-integer 0 = 0
  nat-of-integer 1 = 1
  nat-of-integer (numeral k) = numeral k
  including integer.lifting by (transfer, simp)+

context
  includes integer.lifting and bit-operations-syntax
begin

lemma [code]:
  ‹bit m n ⟷ bit (integer-of-nat m) n›
  by transfer (simp add: bit-simps)

lemma [code]:
  ‹integer-of-nat (m AND n) = integer-of-nat m AND integer-of-nat n›
  by transfer (simp add: of-nat-and-eq)

lemma [code]:
  ‹integer-of-nat (m OR n) = integer-of-nat m OR integer-of-nat n›
  by transfer (simp add: of-nat-or-eq)

lemma [code]:
  ‹integer-of-nat (m XOR n) = integer-of-nat m XOR integer-of-nat n›
  by transfer (simp add: of-nat-xor-eq)

lemma [code]:
  ‹integer-of-nat (push-bit n m) = push-bit n (integer-of-nat m)›
  by transfer (simp add: of-nat-push-bit)

lemma [code]:
  ‹integer-of-nat (drop-bit n m) = drop-bit n (integer-of-nat m)›
  by transfer (simp add: of-nat-drop-bit)

lemma [code]:
  ‹integer-of-nat (take-bit n m) = take-bit n (integer-of-nat m)›
  by transfer (simp add: of-nat-take-bit)

lemma [code]:
  ‹integer-of-nat (mask n) = mask n›
  by transfer (simp add: of-nat-mask-eq)
```

lemma [*code*]:
‹*integer-of-nat* (*set-bit n m*) = *set-bit n* (*integer-of-nat m*)›
by *transfer* (*simp add*: *of-nat-set-bit-eq*)

lemma [*code*]:
‹*integer-of-nat* (*unset-bit n m*) = *unset-bit n* (*integer-of-nat m*)›
by *transfer* (*simp add*: *of-nat-unset-bit-eq*)

lemma [*code*]:
‹*integer-of-nat* (*flip-bit n m*) = *flip-bit n* (*integer-of-nat m*)›
by *transfer* (*simp add*: *of-nat-flip-bit-eq*)

end

code-identifier
code-module *Code-Target-Nat* ⇀
(*SML*) *Arith* **and** (*OCaml*) *Arith* **and** (*Haskell*) *Arith*

end

125 Implementation of natural and integer numbers by target-language integers

theory *Code-Target-Numeral*
imports *Code-Target-Nat Code-Target-Int*
begin

end

126 Preprocessor setup for floats implemented by target language numerals

theory *Code-Target-Numeral-Float*
imports *Float Code-Target-Numeral*
begin

lemma *numeral-float-computation-unfold* [*code-computation-unfold*]:
‹*numeral k* = *Float* (*int-of-integer* (*Code-Numeral.positive k*)) *0*›
‹− *numeral k* = *Float* (*int-of-integer* (*Code-Numeral.negative k*)) *0*›
by (*simp-all add*: *Float.compute-float-numeral Float.compute-float-neg-numeral*)

end

theory *Complex-Order*
imports *Complex-Main*

begin

instantiation *complex* :: *order* **begin**

definition ‹$x < y \longleftrightarrow Re\ x < Re\ y \wedge Im\ x = Im\ y$›
definition ‹$x \leq y \longleftrightarrow Re\ x \leq Re\ y \wedge Im\ x = Im\ y$›

instance
apply *standard*
by (*auto simp*: *less-complex-def less-eq-complex-def complex-eq-iff*)
end

lemma *nonnegative-complex-is-real*: ‹$(x::complex) \geq 0 \Longrightarrow x \in \mathbb{R}$›
by (*simp add*: *complex-is-Real-iff less-eq-complex-def*)

lemma *complex-is-real-iff-compare0*: ‹$(x::complex) \in \mathbb{R} \longleftrightarrow x \leq 0 \vee x \geq 0$›
using *complex-is-Real-iff less-eq-complex-def* **by** *auto*

instance *complex* :: *ordered-comm-ring*
apply *standard*
by (*auto simp*: *less-complex-def less-eq-complex-def complex-eq-iff mult-left-mono mult-right-mono*)

instance *complex* :: *ordered-real-vector*
apply *standard*
by (*auto simp*: *less-complex-def less-eq-complex-def mult-left-mono mult-right-mono*)

instance *complex* :: *ordered-cancel-comm-semiring*
by *standard*

end

127 Abstract type of association lists with unique keys

theory *DAList*
imports *AList*
begin

This was based on some existing fragments in the AFP-Collection framework.

127.1 Preliminaries

lemma *distinct-map-fst-filter*:
distinct (*map fst xs*) $\Longrightarrow$ *distinct* (*map fst* (*List.filter P xs*))
by (*induct xs*) *auto*

127.2 Type ('*key*, '*value*) *alist*

typedef ('*key*, '*value*) *alist* = {*xs* :: ('*key* × '*value*) *list*. (*distinct* ∘ *map fst*) *xs*}
  morphisms *impl-of Alist*
proof
  show [] ∈ {*xs*. (*distinct* ∘ *map fst*) *xs*}
    by *simp*
qed

setup-lifting *type-definition-alist*

lemma *alist-ext*: *impl-of xs* = *impl-of ys* ⟹ *xs* = *ys*
  by (*simp add*: *impl-of-inject*)

lemma *alist-eq-iff*: *xs* = *ys* ⟷ *impl-of xs* = *impl-of ys*
  by (*simp add*: *impl-of-inject*)

lemma *impl-of-distinct* [*simp*, *intro*]: *distinct* (*map fst* (*impl-of xs*))
  using *impl-of*[*of xs*] **by** *simp*

lemma *impl-of-Alist*:
  ‹*impl-of* (*Alist xs*) = *xs*› **if** ‹*distinct* (*map fst xs*)›
  using *Alist-inverse* [*of xs*] *that* **by** *simp*

lemma *Alist-impl-of* [*code abstype*]: *Alist* (*impl-of xs*) = *xs*
  by (*rule impl-of-inverse*)

127.3 Primitive operations

lift-definition *lookup* :: ('*key*, '*value*) *alist* ⇒ '*key* ⇒ '*value option* **is** *map-of* .

lift-definition *empty* :: ('*key*, '*value*) *alist* **is** []
  by *simp*

lift-definition *update* :: '*key* ⇒ '*value* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
  is *AList.update*
  by (*simp add*: *distinct-update*)

lift-definition *delete* :: '*key* ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
  is *AList.delete*
  by (*simp add*: *distinct-delete*)

lift-definition *map-entry* ::
    '*key* ⇒ ('*value* ⇒ '*value*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*
  is *AList.map-entry*
  by (*simp add*: *distinct-map-entry*)

lift-definition *filter* :: ('*key* × '*value* ⇒ *bool*) ⇒ ('*key*, '*value*) *alist* ⇒ ('*key*, '*value*) *alist*

is *List.filter*
by (*simp add*: *distinct-map-fst-filter*)

lift-definition *map-default* ::
'key ⇒ *'value* ⇒ (*'value* ⇒ *'value*) ⇒ (*'key*, *'value*) *alist* ⇒ (*'key*, *'value*) *alist*
is *AList.map-default*
by (*simp add*: *distinct-map-default*)

127.4 Abstract operation properties

lemma *lookup-empty* [*simp*]: *lookup empty k* = *None*
by (*simp add*: *empty-def lookup-def Alist-inverse*)

lemma *lookup-update*:
lookup (*update k1 v xs*) *k2* = (*if k1* = *k2 then Some v else lookup xs k2*)
by(*transfer*)(*simp add*: *update-conv'*)

lemma *lookup-update-eq* [*simp*]:
k1 = *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *Some v*
by(*simp add*: *lookup-update*)

lemma *lookup-update-neq* [*simp*]:
k1 ≠ *k2* ⟹ *lookup* (*update k1 v xs*) *k2* = *lookup xs k2*
by(*simp add*: *lookup-update*)

lemma *update-update-eq* [*simp*]:
k1 = *k2* ⟹ *update k2 v2* (*update k1 v1 xs*) = *update k2 v2 xs*
by(*transfer*)(*simp add*: *update-conv'*)

lemma *lookup-delete* [*simp*]: *lookup* (*delete k al*) = (*lookup al*)(*k* := *None*)
by (*simp add*: *lookup-def delete-def Alist-inverse distinct-delete delete-conv'*)

127.5 Further operations

127.5.1 Equality

instantiation *alist* :: (*equal*, *equal*) *equal*
begin

definition *HOL.equal* (*xs* :: (*'a*, *'b*) *alist*) *ys* == *impl-of xs* = *impl-of ys*

instance
by *standard* (*simp add*: *equal-alist-def impl-of-inject*)

end

127.5.2 Size

instantiation *alist* :: (*type*, *type*) *size*
begin

definition *size* (*al* :: ('*a*, '*b*) *alist*) = *length* (*impl-of al*)

instance ..

end

127.6 Quickcheck generators

context
includes *state-combinator-syntax* **and** *term-syntax*
begin

definition
valterm-empty :: ('*key* :: *typerep*, '*value* :: *typerep*) *alist* × (*unit* ⇒ *Code-Evaluation.term*)
where *valterm-empty* = *Code-Evaluation.valtermify empty*

definition
valterm-update :: '*key* :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
'*value* :: *typerep* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
('*key*, '*value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) ⇒
('*key*, '*value*) *alist* × (*unit* ⇒ *Code-Evaluation.term*) **where**
[*code-unfold*]: *valterm-update k v a* = *Code-Evaluation.valtermify update* {·} *k* {·} *v* {·}*a*

fun *random-aux-alist*
where
random-aux-alist i j =
(*if i* = *0 then Pair valterm-empty*
else Quickcheck-Random.collapse
(*Random.select-weight*
[(*i*, *Quickcheck-Random.random j* ∘→ (λ*k*. *Quickcheck-Random.random j* ∘→
(λ*v*. *random-aux-alist* (*i* − *1*) *j* ∘→ (λ*a*. *Pair* (*valterm-update k v a*))))),
(*1*, *Pair valterm-empty*)]))

end

instantiation *alist* :: (*random*, *random*) *random*
begin

definition *random-alist*
where
random-alist i = *random-aux-alist i i*

instance ..

end

instantiation *alist* :: (*exhaustive*, *exhaustive*) *exhaustive*
begin

fun *exhaustive-alist* ::
(('*a*, '*b*) *alist* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where
exhaustive-alist f i =
(*if i* = *0 then None*
else
case f empty of
Some ts ⇒ *Some ts*
| *None* ⇒
exhaustive-alist
(λ*a*. *Quickcheck-Exhaustive.exhaustive*
(λ*k*. *Quickcheck-Exhaustive.exhaustive* (λ*v*. *f* (*update k v a*)) (*i* − *1*))
(*i* − *1*))
(*i* − *1*))

instance ..

end

instantiation *alist* :: (*full-exhaustive*, *full-exhaustive*) *full-exhaustive*
begin

fun *full-exhaustive-alist* ::
(('*a*, '*b*) *alist* × (*unit* ⇒ *term*) ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒
(*bool* × *term list*) *option*
where
full-exhaustive-alist f i =
(*if i* = *0 then None*
else
case f valterm-empty of
Some ts ⇒ *Some ts*
| *None* ⇒
full-exhaustive-alist
(λ*a*.
Quickcheck-Exhaustive.full-exhaustive
(λ*k*. *Quickcheck-Exhaustive.full-exhaustive* (λ*v*. *f* (*valterm-update k v a*)) (*i* − *1*))
(*i* − *1*))
(*i* − *1*))

instance ..

end

128 alist is a BNF

lift-bnf (*dead* 'k, *set*: 'v) *alist* [*wits*: [] :: ('k × 'v) *list*] **for** *map*: *map rel*: *rel*
by *auto*

hide-const *valterm-empty valterm-update random-aux-alist*

hide-fact (**open**) *lookup-def empty-def update-def delete-def map-entry-def filter-def map-default-def*
hide-const (**open**) *impl-of lookup empty update delete map-entry filter map-default map set rel*

end

129 Multisets partially implemented by association lists

theory *DAList-Multiset*
imports *Multiset DAList*
begin

Raw operations on lists

definition *join-raw* ::
('key ⇒ 'val × 'val ⇒ 'val) ⇒
('key × 'val) *list* ⇒ ('key × 'val) *list* ⇒ ('key × 'val) *list*
where *join-raw f xs ys* = *foldr* (λ(*k*, *v*). *map-default k v* (λv'. *f k* (v', *v*))) *ys xs*

lemma *join-raw-Nil* [*simp*]: *join-raw f xs* [] = *xs*
by (*simp add*: *join-raw-def*)

lemma *join-raw-Cons* [*simp*]:
join-raw f xs ((*k*, *v*) # *ys*) = *map-default k v* (λv'. *f k* (v', *v*)) (*join-raw f xs ys*)
by (*simp add*: *join-raw-def*)

lemma *map-of-join-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*join-raw f xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *map-of ys x*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*f x* (*v*, *v'*))))
using *assms*
apply (*induct ys*)
apply (*auto simp add*: *map-of-map-default split*: *option.split*)
apply (*metis map-of-eq-None-iff option.simps*(*2*) *weak-map-of-SomeI*)
apply (*metis Some-eq-map-of-iff map-of-eq-None-iff option.simps*(*2*))
done

lemma *distinct-join-raw*:

assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*join-raw f xs ys*))
using *assms*
proof (*induct ys*)
case *Nil*
then show *?case* **by** *simp*
next
case (*Cons y ys*)
then show *?case* **by** (*cases y*) (*simp add*: *distinct-map-default*)
qed

definition *subtract-entries-raw xs ys* = *foldr* (λ(*k*, *v*). *AList.map-entry k* (λ*v'*. *v'* − *v*)) *ys xs*

lemma *map-of-subtract-entries-raw*:
assumes *distinct* (*map fst ys*)
shows *map-of* (*subtract-entries-raw xs ys*) *x* =
(*case map-of xs x of*
None ⇒ *None*
| *Some v* ⇒ (*case map-of ys x of None* ⇒ *Some v* | *Some v'* ⇒ *Some* (*v* − *v'*)))
using *assms*
unfolding *subtract-entries-raw-def*
apply (*induct ys*)
apply *auto*
apply (*simp split*: *option.split*)
apply (*simp add*: *map-of-map-entry*)
apply (*auto split*: *option.split*)
apply (*metis map-of-eq-None-iff option.simps*(*3*) *option.simps*(*4*))
apply (*metis map-of-eq-None-iff option.simps*(*4*) *option.simps*(*5*))
done

lemma *distinct-subtract-entries-raw*:
assumes *distinct* (*map fst xs*)
shows *distinct* (*map fst* (*subtract-entries-raw xs ys*))
using *assms*
unfolding *subtract-entries-raw-def*
by (*induct ys*) (*auto simp add*: *distinct-map-entry*)

Operations on alists with distinct keys

lift-definition *join* :: ('*a* ⇒ '*b* × '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *join-raw*
by (*simp add*: *distinct-join-raw*)

lift-definition *subtract-entries* :: ('*a*, ('*b* :: *minus*)) *alist* ⇒ ('*a*, '*b*) *alist* ⇒ ('*a*, '*b*) *alist*
is *subtract-entries-raw*
by (*simp add*: *distinct-subtract-entries-raw*)

Implementing multisets by means of association lists

definition *count-of* :: ('*a* × *nat*) *list* ⇒ '*a* ⇒ *nat*
where *count-of xs x* = (*case map-of xs x of None* ⇒ *0* | *Some n* ⇒ *n*)

lemma *count-of-multiset*: *finite* {*x*. *0* < *count-of xs x*}
proof −
let *?A* = {*x*::'*a*. *0* < (*case map-of xs x of None* ⇒ *0*::*nat* | *Some n* ⇒ *n*)}
have *?A* ⊆ *dom* (*map-of xs*)
proof
fix *x*
assume *x* ∈ *?A*
then have *0* < (*case map-of xs x of None* ⇒ *0*::*nat* | *Some n* ⇒ *n*)
by *simp*
then have *map-of xs x* ≠ *None*
by (*cases map-of xs x*) *auto*
then show *x* ∈ *dom* (*map-of xs*)
by *auto*
qed
with *finite-dom-map-of* [*of xs*] **have** *finite ?A*
by (*auto intro*: *finite-subset*)
then show *?thesis*
by (*simp add*: *count-of-def fun-eq-iff*)
qed

lemma *count-simps* [*simp*]:
count-of [] = (λ-. *0*)
count-of ((*x*, *n*) # *xs*) = (λ*y*. *if x* = *y then n else count-of xs y*)
by (*simp-all add*: *count-of-def fun-eq-iff*)

lemma *count-of-empty*: *x* ∉ *fst* ' *set xs* ⟹ *count-of xs x* = *0*
by (*induct xs*) (*simp-all add*: *count-of-def*)

lemma *count-of-filter*: *count-of* (*List.filter* (*P* ∘ *fst*) *xs*) *x* = (*if P x then count-of xs x else 0*)
by (*induct xs*) *auto*

lemma *count-of-map-default* [*simp*]:
count-of (*map-default x b* (λ*x*. *x* + *b*) *xs*) *y* =
(*if x* = *y then count-of xs x* + *b else count-of xs y*)
unfolding *count-of-def* **by** (*simp add*: *map-of-map-default split*: *option.split*)

lemma *count-of-join-raw*:
distinct (*map fst ys*) ⟹
count-of xs x + *count-of ys x* = *count-of* (*join-raw* (λ*x* (*x*, *y*). *x* + *y*) *xs ys*) *x*
unfolding *count-of-def* **by** (*simp add*: *map-of-join-raw split*: *option.split*)

lemma *count-of-subtract-entries-raw*:
distinct (*map fst ys*) ⟹
count-of xs x − *count-of ys x* = *count-of* (*subtract-entries-raw xs ys*) *x*
unfolding *count-of-def* **by** (*simp add*: *map-of-subtract-entries-raw split*: *op-*

tion.split)

Code equations for multiset operations

definition *Bag* :: (*'a*, *nat*) *alist* ⇒ *'a multiset*
where *Bag xs* = *Abs-multiset* (*count-of* (*DAList.impl-of xs*))

code-datatype *Bag*

lemma *count-Bag* [*simp*, *code*]: *count* (*Bag xs*) = *count-of* (*DAList.impl-of xs*)
by (*simp add*: *Bag-def count-of-multiset*)

lemma *Bag-eq*:
‹*Bag ms* = (∑ (*a*, *n*)←*alist.impl-of ms*. *replicate-mset n a*)›
for *ms* :: ‹(*'a*, *nat*) *alist*›
proof −
have ∗: ‹*count-of xs a* = *count* (∑ (*a*, *n*)←*xs*. *replicate-mset n a*) *a*›
if ‹*distinct* (*map fst xs*)›
for *xs* **and** *a* :: *'a*
using *that* **proof** (*induction xs*)
case *Nil*
then show *?case*
by *simp*
next
case (*Cons xn xs*)
then show *?case* **by** (*cases xn*)
(*auto simp add*: *count-eq-zero-iff if-split-mem2 image-iff*)
qed
show *?thesis*
by (*rule multiset-eqI*) (*simp add*: ∗)
qed

lemma *Mempty-Bag* [*code*]: {#} = *Bag* (*DAList.empty*)
by (*simp add*: *multiset-eq-iff alist.Alist-inverse DAList.empty-def*)

lift-definition *is-empty-Bag-impl* :: (*'a*, *nat*) *alist* ⇒ *bool* **is**
λxs. *list-all* (*λx*. *snd x* = *0*) *xs* **.**

lemma *is-empty-Bag* [*code*]: *Multiset.is-empty* (*Bag xs*) ⟷ *is-empty-Bag-impl xs*
proof −
have *Multiset.is-empty* (*Bag xs*) ⟷ (∀ *x*. *count* (*Bag xs*) *x* = *0*)
unfolding *Multiset.is-empty-def multiset-eq-iff* **by** *simp*
also have ... ⟷ (∀ *x*∈*fst* ‘ *set* (*alist.impl-of xs*). *count* (*Bag xs*) *x* = *0*)
proof (*intro iffI allI ballI*)
fix *x* **assume** *A*: ∀ *x*∈*fst* ‘ *set* (*alist.impl-of xs*). *count* (*Bag xs*) *x* = *0*
thus *count* (*Bag xs*) *x* = *0*
proof (*cases x* ∈ *fst* ‘ *set* (*alist.impl-of xs*))
case *False*
thus *?thesis* **by** (*force simp*: *count-of-def split*: *option.splits*)
qed (*insert A*, *auto*)

qed *simp-all*
also have ... $\longleftrightarrow$ *list-all* (λx. *snd x = 0*) (*alist.impl-of xs*)
by (*auto simp*: *count-of-def list-all-def*)
finally show *?thesis* **by** (*simp add*: *is-empty-Bag-impl.rep-eq*)
qed

lemma *union-Bag* [*code*]: *Bag xs + Bag ys = Bag* (*join* (λx (*n1*, *n2*). *n1 + n2*) *xs ys*)
by (*rule multiset-eqI*)
(*simp add*: *count-of-join-raw alist.Alist-inverse distinct-join-raw join-def*)

lemma *add-mset-Bag* [*code*]: *add-mset x* (*Bag xs*) =
Bag (*join* (λx (*n1*, *n2*). *n1 + n2*) (*DAList.update x 1 DAList.empty*) *xs*)
unfolding *add-single-right-eq-add-mset*[*symmetric*, *of x Bag xs*] *union-Bag*[*symmetric*]
by (*simp add*: *multiset-eq-iff update.rep-eq empty.rep-eq*)

lemma *minus-Bag* [*code*]: *Bag xs − Bag ys = Bag* (*subtract-entries xs ys*)
by (*rule multiset-eqI*)
(*simp add*: *count-of-subtract-entries-raw alist.Alist-inverse*
distinct-subtract-entries-raw subtract-entries-def)

lemma *filter-Bag* [*code*]: *filter-mset P* (*Bag xs*) = *Bag* (*DAList.filter* (*P* $\circ$ *fst*) *xs*)
by (*rule multiset-eqI*) (*simp add*: *count-of-filter DAList.filter.rep-eq*)

lemma *mset-eq* [*code*]: *HOL.equal* (*m1*::'*a*::*equal multiset*) *m2* $\longleftrightarrow$ *m1* $\subseteq\#$ *m2* $\wedge$ *m2* $\subseteq\#$ *m1*
by (*metis equal-multiset-def subset-mset.order-eq-iff*)

By default the code for $<$ is $(xs < ys) = (xs \leq ys \wedge xs \neq ys)$. With equality implemented by $\leq$, this leads to three calls of $\leq$. Here is a more efficient version:

lemma *mset-less*[*code*]: *xs* $\subset\#$ (*ys* :: '*a multiset*) $\longleftrightarrow$ *xs* $\subseteq\#$ *ys* $\wedge$ $\neg$ *ys* $\subseteq\#$ *xs*
by (*rule subset-mset.less-le-not-le*)

lemma *mset-less-eq-Bag0*:
Bag xs $\subseteq\#$ *A* $\longleftrightarrow$ ($\forall$ (*x*, *n*) $\in$ *set* (*DAList.impl-of xs*). *count-of* (*DAList.impl-of xs*) *x* $\leq$ *count A x*)
(**is** *?lhs* $\longleftrightarrow$ *?rhs*)
proof
assume *?lhs*
then show *?rhs* **by** (*auto simp add*: *subseteq-mset-def*)
next
assume *?rhs*
show *?lhs*
proof (*rule mset-subset-eqI*)
fix *x*
from ‹*?rhs*› **have** *count-of* (*DAList.impl-of xs*) *x* $\leq$ *count A x*
by (*cases x* $\in$ *fst* ‘ *set* (*DAList.impl-of xs*)) (*auto simp add*: *count-of-empty*)

```
    then show count (Bag xs) x ≤ count A x by (simp add: subset-mset-def)
  qed
qed

lemma mset-less-eq-Bag [code]:
  Bag xs ⊆# (A :: 'a multiset) ⟷ (∀(x, n) ∈ set (DAList.impl-of xs). n ≤ count
A x)
proof −
  {
    fix x n
    assume (x,n) ∈ set (DAList.impl-of xs)
    then have count-of (DAList.impl-of xs) x = n
    proof transfer
      fix x n
      fix xs :: ('a × nat) list
      show (distinct ∘ map fst) xs ⟹ (x, n) ∈ set xs ⟹ count-of xs x = n
      proof (induct xs)
        case Nil
        then show ?case by simp
      next
        case (Cons ym ys)
        obtain y m where ym: ym = (y,m) by force
        note Cons = Cons[unfolded ym]
        show ?case
        proof (cases x = y)
          case False
          with Cons show ?thesis
            unfolding ym by auto
        next
          case True
          with Cons(2−3) have m = n by force
          with True show ?thesis
            unfolding ym by auto
        qed
      qed
    qed
  }
  then show ?thesis
    unfolding mset-less-eq-Bag0 by auto
qed

declare inter-mset-def [code]
declare union-mset-def [code]
declare mset.simps [code]


fun fold-impl :: ('a ⇒ nat ⇒ 'b ⇒ 'b) ⇒ 'b ⇒ ('a × nat) list ⇒ 'b
where
  fold-impl fn e ((a,n) # ms) = (fold-impl fn ((fn a n) e) ms)
```

| *fold-impl fn e* [] = *e*

context
begin

qualified definition *fold* :: ('*a* ⇒ *nat* ⇒ '*b* ⇒ '*b*) ⇒ '*b* ⇒ ('*a*, *nat*) *alist* ⇒ '*b*
  where *fold f e al* = *fold-impl f e* (*DAList.impl-of al*)

end

context *comp-fun-commute*
begin

lemma *DAList-Multiset-fold*:
  assumes *fn*: ⋀*a n x*. *fn a n x* = (*f a* ^^ *n*) *x*
  shows *fold-mset f e* (*Bag al*) = *DAList-Multiset.fold fn e al*
  unfolding *DAList-Multiset.fold-def*
proof (*induct al*)
  fix *ys*
  let *?inv* = {*xs* :: ('*a* × *nat*) *list*. (*distinct* ∘ *map fst*) *xs*}
  note *cs*[*simp del*] = *count-simps*
  have *count*[*simp*]: ⋀*x*. *count* (*Abs-multiset* (*count-of x*)) = *count-of x*
    by (*rule Abs-multiset-inverse*) (*simp add*: *count-of-multiset*)
  assume *ys*: *ys* ∈ *?inv*
  then show *fold-mset f e* (*Bag* (*Alist ys*)) = *fold-impl fn e* (*DAList.impl-of* (*Alist ys*))
    unfolding *Bag-def* **unfolding** *Alist-inverse*[*OF ys*]
  proof (*induct ys arbitrary*: *e rule*: *list.induct*)
    case *Nil*
    show *?case*
      by (*rule trans*[*OF arg-cong*[*of* - {#} *fold-mset f e*, *OF multiset-eqI*]])
        (*auto*, *simp add*: *cs*)
  next
    case (*Cons pair ys e*)
    obtain *a n* **where** *pair*: *pair* = (*a*,*n*)
      by *force*
    from *fn*[*of a n*] **have** [*simp*]: *fn a n* = (*f a* ^^ *n*)
      by *auto*
    have *inv*: *ys* ∈ *?inv*
      using *Cons*(*2*) **by** *auto*
    note *IH* = *Cons*(*1*)[*OF inv*]
    define *Ys* **where** *Ys* = *Abs-multiset* (*count-of ys*)
    have *id*: *Abs-multiset* (*count-of* ((*a*, *n*) # *ys*)) = (((+) {# *a* #}) ^^ *n*) *Ys*
      unfolding *Ys-def*
    proof (*rule multiset-eqI*, *unfold count*)
      fix *c*
      show *count-of* ((*a*, *n*) # *ys*) *c* =
        count (((+) {#*a*#}) ^^ *n*) (*Abs-multiset* (*count-of ys*))) *c* (**is** *?l* = *?r*)
      proof (*cases c* = *a*)

```
      case False
      then show ?thesis
        unfolding cs by (induct n) auto
    next
      case True
      then have ?l = n by (simp add: cs)
      also have n = ?r unfolding True
      proof (induct n)
        case 0
        from Cons(2)[unfolded pair] have a ∉ fst ` set ys by auto
        then show ?case by (induct ys) (simp, auto simp: cs)
      next
        case Suc
        then show ?case by simp
      qed
      finally show ?thesis .
    qed
  qed
  show ?case
    unfolding pair
    apply (simp add: IH[symmetric])
    unfolding id Ys-def[symmetric]
    apply (induct n)
    apply (auto simp: fold-mset-fun-left-comm[symmetric])
    done
 qed
qed

end

context
begin

private lift-definition single-alist-entry :: 'a ⇒ 'b ⇒ ('a, 'b) alist is λa b. [(a,
b)]
 by auto

lemma image-mset-Bag [code]:
 image-mset f (Bag ms) =
   DAList-Multiset.fold (λa n m. Bag (single-alist-entry (f a) n) + m) {#} ms
 unfolding image-mset-def
proof (rule comp-fun-commute.DAList-Multiset-fold, unfold-locales, (auto simp:
ac-simps)[1])
 fix a n m
 show Bag (single-alist-entry (f a) n) + m = ((add-mset ∘ f) a ^^ n) m (is ?l
= ?r)
 proof (rule multiset-eqI)
   fix x
   have count ?r x = (if x = f a then n + count m x else count m x)
```

```
    by (induct n) auto
  also have ... = count ?l x
    by (simp add: single-alist-entry.rep-eq)
  finally show count ?l x = count ?r x ..
 qed
qed

end

— we cannot use λa n. (+) (a * n) for folding, since (*) is not defined in comm-monoid-add
lemma sum-mset-Bag[code]: sum-mset (Bag ms) = DAList-Multiset.fold (λa n.
(((+) a) ^^ n)) 0 ms
 unfolding sum-mset.eq-fold
 apply (rule comp-fun-commute.DAList-Multiset-fold)
 apply unfold-locales
 apply (auto simp: ac-simps)
 done

— we cannot use λa n. (*) (a ^ n) for folding, since (^) is not defined in comm-monoid-mult
lemma prod-mset-Bag[code]: prod-mset (Bag ms) = DAList-Multiset.fold (λa n.
(((*) a) ^^ n)) 1 ms
 unfolding prod-mset.eq-fold
 apply (rule comp-fun-commute.DAList-Multiset-fold)
 apply unfold-locales
 apply (auto simp: ac-simps)
 done

lemma size-fold: size A = fold-mset (λ-. Suc) 0 A (is - = fold-mset ?f - -)
proof −
 interpret comp-fun-commute ?f by standard auto
 show ?thesis by (induct A) auto
qed

lemma size-Bag[code]: size (Bag ms) = DAList-Multiset.fold (λa n. (+) n) 0 ms
 unfolding size-fold
proof (rule comp-fun-commute.DAList-Multiset-fold, unfold-locales, simp)
 fix a n x
 show n + x = (Suc ^^ n) x
   by (induct n) auto
qed


lemma set-mset-fold: set-mset A = fold-mset insert {} A (is - = fold-mset ?f - -)
proof −
 interpret comp-fun-commute ?f by standard auto
 show ?thesis by (induct A) auto
qed

lemma set-mset-Bag[code]:
```

set-mset (*Bag ms*) = *DAList-Multiset.fold* (λ*a n*. (*if n* = *0 then* (λ*m*. *m*) *else insert a*)) {} *ms*
unfolding *set-mset-fold*
proof (*rule comp-fun-commute.DAList-Multiset-fold*, *unfold-locales*, (*auto simp*: *ac-simps*)[*1*])
fix *a n x*
show (*if n* = *0 then* λ*m*. *m else insert a*) *x* = (*insert a* ^^ *n*) *x* (**is** *?l n* = *?r n*)
proof (*cases n*)
case *0*
then show *?thesis* **by** *simp*
next
case (*Suc m*)
then have *?l n* = *insert a x* **by** *simp*
moreover have *?r n* = *insert a x* **unfolding** *Suc* **by** (*induct m*) *auto*
ultimately show *?thesis* **by** *auto*
qed
qed

lemma *sorted-list-of-multiset-Bag* [*code*]:
‹*sorted-list-of-multiset* (*Bag ms*) = *concat* (*map* (λ(*a*, *n*). *replicate n a*)
(*sort-key fst* (*DAList.impl-of ms*)))› (**is** ‹*?lhs* = *?rhs*›)
proof −
have ∗: ‹*sorted* (*concat* (*map* (λ(*a*, *n*). *replicate n a*) *ans*))›
if ‹*sorted* (*map fst ans*)›
for *ans* :: ‹('*a* × *nat*) *list*›
using *that* **by** (*induction ans*) (*auto simp add*: *sorted-append*)
have ‹*mset ?rhs* = *mset ?lhs*›
by (*simp add*: *Bag-eq mset-concat comp-def split-def flip*: *sum-mset-sum-list*)
moreover have ‹*sorted ?rhs*›
by (*rule* ∗) *simp*
ultimately have ‹*sort ?lhs* = *?rhs*›
by (*rule properties-for-sort*)
then show *?thesis*
by *simp*
qed

instantiation *multiset* :: (*exhaustive*) *exhaustive*
begin

definition *exhaustive-multiset* ::
('*a multiset* ⇒ (*bool* × *term list*) *option*) ⇒ *natural* ⇒ (*bool* × *term list*) *option*
where *exhaustive-multiset f i* = *Quickcheck-Exhaustive.exhaustive* (λ*xs*. *f* (*Bag xs*)) *i*

instance ..

end

end

130 Implementation of Red-Black Trees

theory *RBT-Impl*
imports *Main*
begin

For applications, you should use theory *RBT* which defines an abstract type of red-black tree obeying the invariant.

130.1 Datatype of RB trees

datatype *color* = *R* | *B*
datatype ('*a*, '*b*) *rbt* = *Empty* | *Branch color* ('*a*, '*b*) *rbt* '*a* '*b* ('*a*, '*b*) *rbt*

lemma *rbt-cases*:
  obtains (*Empty*) *t* = *Empty*
  | (*Red*) *l k v r* **where** *t* = *Branch R l k v r*
  | (*Black*) *l k v r* **where** *t* = *Branch B l k v r*
proof (*cases t*)
  case *Empty* **with** *that* **show** *thesis* **by** *blast*
next
  case (*Branch c*) **with** *that* **show** *thesis* **by** (*cases c*) *blast*+
qed

130.2 Tree properties

130.2.1 Content of a tree

primrec *entries* :: ('*a*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list*
where
  entries Empty = []
| *entries* (*Branch - l k v r*) = *entries l* @ (*k*,*v*) # *entries r*

abbreviation (*input*) *entry-in-tree* :: '*a* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ *bool*
where
  entry-in-tree k v t ≡ (*k*, *v*) ∈ *set* (*entries t*)

definition *keys* :: ('*a*, '*b*) *rbt* ⇒ '*a list* **where**
  keys t = *map fst* (*entries t*)

lemma *keys-simps* [*simp*]:
  keys Empty = []
  keys (*Branch c l k v r*) = *keys l* @ *k* # *keys r*
  by (*simp-all add*: *keys-def*)

lemma *entry-in-tree-keys*:
  assumes (*k*, *v*) ∈ *set* (*entries t*)
  shows *k* ∈ *set* (*keys t*)
proof −
  from *assms* **have** *fst* (*k*, *v*) ∈ *fst* ' *set* (*entries t*) **by** (*rule imageI*)

then show *?thesis* **by** (*simp add*: *keys-def*)
qed

lemma *keys-entries*:
$k \in set\ (keys\ t) \longleftrightarrow (\exists\, v.\ (k, v) \in set\ (entries\ t))$
by (*auto intro*: *entry-in-tree-keys*) (*auto simp add*: *keys-def*)

lemma *non-empty-rbt-keys*:
$t \neq rbt.Empty \Longrightarrow keys\ t \neq []$
by (*cases t*) *simp-all*

130.2.2 Search tree properties

context *ord* **begin**

definition *rbt-less* :: $'a \Rightarrow ('a, 'b)\ rbt \Rightarrow bool$
where
rbt-less-prop: $rbt\text{-}less\ k\ t \longleftrightarrow (\forall\, x{\in}set\ (keys\ t).\ x < k)$

abbreviation *rbt-less-symbol* (**infix** $\langle|\ll\rangle$ *50*)
where $t\ |\ll\ x \equiv rbt\text{-}less\ x\ t$

definition *rbt-greater* :: $'a \Rightarrow ('a, 'b)\ rbt \Rightarrow bool$ (**infix** $\langle\ll|\rangle$ *50*)
where
rbt-greater-prop: $rbt\text{-}greater\ k\ t = (\forall\, x{\in}set\ (keys\ t).\ k < x)$

lemma *rbt-less-simps* [*simp*]:
$Empty\ |\ll\ k = True$
$Branch\ c\ lt\ kt\ v\ rt\ |\ll\ k \longleftrightarrow kt < k \wedge lt\ |\ll\ k \wedge rt\ |\ll\ k$
by (*auto simp add*: *rbt-less-prop*)

lemma *rbt-greater-simps* [*simp*]:
$k\ \ll|\ Empty = True$
$k\ \ll|\ (Branch\ c\ lt\ kt\ v\ rt) \longleftrightarrow k < kt \wedge k\ \ll|\ lt \wedge k\ \ll|\ rt$
by (*auto simp add*: *rbt-greater-prop*)

lemmas *rbt-ord-props* = *rbt-less-prop rbt-greater-prop*

lemmas *rbt-greater-nit* = *rbt-greater-prop entry-in-tree-keys*
lemmas *rbt-less-nit* = *rbt-less-prop entry-in-tree-keys*

lemma (**in** *order*)
shows *rbt-less-eq-trans*: $l\ |\ll\ u \Longrightarrow u \leq v \Longrightarrow l\ |\ll\ v$
and *rbt-less-trans*: $t\ |\ll\ x \Longrightarrow x < y \Longrightarrow t\ |\ll\ y$
and *rbt-greater-eq-trans*: $u \leq v \Longrightarrow v\ \ll|\ r \Longrightarrow u\ \ll|\ r$
and *rbt-greater-trans*: $x < y \Longrightarrow y\ \ll|\ t \Longrightarrow x\ \ll|\ t$
by (*auto simp*: *rbt-ord-props*)

primrec *rbt-sorted* :: $('a, 'b)\ rbt \Rightarrow bool$

where
rbt-sorted Empty = True
| *rbt-sorted (Branch c l k v r) = (l |« k ∧ k «| r ∧ rbt-sorted l ∧ rbt-sorted r)*

end

context *linorder* **begin**

lemma *rbt-sorted-entries*:
rbt-sorted t ⟹ List.sorted (map fst (entries t))
by (*induct t*) (*force simp*: *sorted-append rbt-ord-props dest*!: *entry-in-tree-keys*)+

lemma *distinct-entries*:
rbt-sorted t ⟹ distinct (map fst (entries t))
by (*induct t*) (*force simp*: *sorted-append rbt-ord-props dest*!: *entry-in-tree-keys*)+

lemma *distinct-keys*:
rbt-sorted t ⟹ distinct (keys t)
by (*simp add*: *distinct-entries keys-def*)

130.2.3 Tree lookup

primrec (**in** *ord*) *rbt-lookup* :: (*'a*, *'b*) *rbt* ⇒ *'a* ⇀ *'b*
where
rbt-lookup Empty k = None
| *rbt-lookup (Branch - l x y r) k =*
(if k < x then rbt-lookup l k else if x < k then rbt-lookup r k else Some y)

lemma *rbt-lookup-keys*: *rbt-sorted t ⟹ dom (rbt-lookup t) = set (keys t)*
by (*induct t*) (*auto simp*: *dom-def rbt-greater-prop rbt-less-prop*)

lemma *dom-rbt-lookup-Branch*:
rbt-sorted (Branch c t1 k v t2) ⟹
dom (rbt-lookup (Branch c t1 k v t2))
= Set.insert k (dom (rbt-lookup t1) ∪ dom (rbt-lookup t2))
proof −
assume *rbt-sorted (Branch c t1 k v t2)*
then show *?thesis* **by** (*simp add*: *rbt-lookup-keys*)
qed

lemma *finite-dom-rbt-lookup* [*simp*, *intro*!]: *finite (dom (rbt-lookup t))*
proof (*induct t*)
case *Empty* **then show** *?case* **by** *simp*
next
case (*Branch color t1 a b t2*)
let *?A = Set.insert a (dom (rbt-lookup t1) ∪ dom (rbt-lookup t2))*
have *dom (rbt-lookup (Branch color t1 a b t2)) ⊆ ?A* **by** (*auto split*: *if-split-asm*)
moreover from *Branch* **have** *finite (insert a (dom (rbt-lookup t1) ∪ dom (rbt-lookup t2)))* **by** *simp*

ultimately show *?case* **by** (*rule finite-subset*)
qed

end

context *ord* **begin**

lemma *rbt-lookup-rbt-less*[*simp*]: *t* |« *k* ⟹ *rbt-lookup t k* = *None*
by (*induct t*) *auto*

lemma *rbt-lookup-rbt-greater*[*simp*]: *k* «| *t* ⟹ *rbt-lookup t k* = *None*
by (*induct t*) *auto*

lemma *rbt-lookup-Empty*: *rbt-lookup Empty* = *Map.empty*
by (*rule ext*) *simp*

end

context *linorder* **begin**

lemma *map-of-entries*:
rbt-sorted t ⟹ *map-of* (*entries t*) = *rbt-lookup t*
proof (*induct t*)
case *Empty* **thus** *?case* **by** (*simp add*: *rbt-lookup-Empty*)
next
case (*Branch c t1 k v t2*)
have *rbt-lookup* (*Branch c t1 k v t2*) = *rbt-lookup t2* ++ [*k*↦*v*] ++ *rbt-lookup t1*
proof (*rule ext*)
fix *x*
from *Branch* **have** *RBT-SORTED*: *rbt-sorted* (*Branch c t1 k v t2*) **by** *simp*
let *?thesis* = *rbt-lookup* (*Branch c t1 k v t2*) *x* = (*rbt-lookup t2* ++ [*k* ↦ *v*] ++ *rbt-lookup t1*) *x*

have *DOM-T1*: !!*k'*. *k'*∈*dom* (*rbt-lookup t1*) ⟹ *k*>*k'*
proof −
fix *k'*
from *RBT-SORTED* **have** *t1* |« *k* **by** *simp*
with *rbt-less-prop* **have** ∀ *k'*∈*set* (*keys t1*). *k*>*k'* **by** *auto*
moreover assume *k'*∈*dom* (*rbt-lookup t1*)
ultimately show *k*>*k'* **using** *rbt-lookup-keys RBT-SORTED* **by** *auto*
qed

have *DOM-T2*: !!*k'*. *k'*∈*dom* (*rbt-lookup t2*) ⟹ *k*<*k'*
proof −
fix *k'*
from *RBT-SORTED* **have** *k* «| *t2* **by** *simp*
with *rbt-greater-prop* **have** ∀ *k'*∈*set* (*keys t2*). *k*<*k'* **by** *auto*
moreover assume *k'*∈*dom* (*rbt-lookup t2*)

ultimately show $k<k'$ **using** *rbt-lookup-keys RBT-SORTED* **by** *auto*
qed

{
assume *C*: $x<k$
hence *rbt-lookup* (*Branch c t1 k v t2*) x = *rbt-lookup t1 x* **by** *simp*
moreover from *C* **have** $x \notin dom\ [k \mapsto v]$ **by** *simp*
moreover have $x \notin$ *dom* (*rbt-lookup t2*)
proof
assume $x \in$ *dom* (*rbt-lookup t2*)
with *DOM-T2* **have** $k<x$ **by** *blast*
with *C* **show** *False* **by** *simp*
qed
ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} moreover {
assume [*simp*]: $x=k$
hence *rbt-lookup* (*Branch c t1 k v t2*) $x = [k \mapsto v]\ x$ **by** *simp*
moreover have $x \notin$ *dom* (*rbt-lookup t1*)
proof
assume $x \in$ *dom* (*rbt-lookup t1*)
with *DOM-T1* **have** $k>x$ **by** *blast*
thus *False* **by** *simp*
qed
ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} moreover {
assume *C*: $x>k$
hence *rbt-lookup* (*Branch c t1 k v t2*) x = *rbt-lookup t2 x* **by** (*simp add*: *less-not-sym*[*of k x*])
moreover from *C* **have** $x \notin dom\ [k \mapsto v]$ **by** *simp*
moreover have $x \notin$ *dom* (*rbt-lookup t1*) **proof**
assume $x \in$ *dom* (*rbt-lookup t1*)
with *DOM-T1* **have** $k>x$ **by** *simp*
with *C* **show** *False* **by** *simp*
qed
ultimately have *?thesis* **by** (*simp add*: *map-add-upd-left map-add-dom-app-simps*)
} ultimately show *?thesis* **using** *less-linear* **by** *blast*
qed
also from *Branch*
have *rbt-lookup t2* ++ $[k \mapsto v]$ ++ *rbt-lookup t1* = *map-of* (*entries* (*Branch c t1 k v t2*)) **by** *simp*
finally show *?case* **by** *simp*
qed

lemma *rbt-lookup-in-tree*: *rbt-sorted t* $\Longrightarrow$ *rbt-lookup t k* = *Some v* $\longleftrightarrow$ $(k, v) \in$ *set* (*entries t*)
by (*simp add*: *map-of-entries* [*symmetric*] *distinct-entries*)

lemma *set-entries-inject*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*

shows *set* (*entries t1*) = *set* (*entries t2*) ⟷ *entries t1* = *entries t2*
proof −
from *rbt-sorted* **have** *distinct* (*map fst* (*entries t1*))
distinct (*map fst* (*entries t2*))
by (*auto intro*: *distinct-entries*)
with *rbt-sorted* **show** *?thesis*
by (*auto intro*: *map-sorted-distinct-set-unique rbt-sorted-entries simp add*: *distinct-map*)
qed

lemma *entries-eqI*:
assumes *rbt-sorted*: *rbt-sorted t1 rbt-sorted t2*
assumes *rbt-lookup*: *rbt-lookup t1* = *rbt-lookup t2*
shows *entries t1* = *entries t2*
proof −
from *rbt-sorted rbt-lookup* **have** *map-of* (*entries t1*) = *map-of* (*entries t2*)
by (*simp add*: *map-of-entries*)
with *rbt-sorted* **have** *set* (*entries t1*) = *set* (*entries t2*)
by (*simp add*: *map-of-inject-set distinct-entries*)
with *rbt-sorted* **show** *?thesis* **by** (*simp add*: *set-entries-inject*)
qed

lemma *entries-rbt-lookup*:
assumes *rbt-sorted t1 rbt-sorted t2*
shows *entries t1* = *entries t2* ⟷ *rbt-lookup t1* = *rbt-lookup t2*
using *assms* **by** (*auto intro*: *entries-eqI simp add*: *map-of-entries* [*symmetric*])

lemma *rbt-lookup-from-in-tree*:
assumes *rbt-sorted t1 rbt-sorted t2*
and ⋀*v*. (*k*, *v*) ∈ *set* (*entries t1*) ⟷ (*k*, *v*) ∈ *set* (*entries t2*)
shows *rbt-lookup t1 k* = *rbt-lookup t2 k*
proof −
from *assms* **have** *k* ∈ *dom* (*rbt-lookup t1*) ⟷ *k* ∈ *dom* (*rbt-lookup t2*)
by (*simp add*: *keys-entries rbt-lookup-keys*)
with *assms* **show** *?thesis* **by** (*auto simp add*: *rbt-lookup-in-tree* [*symmetric*])
qed

end

130.2.4 Red-black properties

primrec *color-of* :: ('*a*, '*b*) *rbt* ⇒ *color*
where
color-of Empty = *B*
| *color-of* (*Branch c* - - - -) = *c*

primrec *bheight* :: ('*a*,'*b*) *rbt* ⇒ *nat*
where
bheight Empty = *0*

| *bheight* (*Branch c lt k v rt*) = (*if c = B then Suc* (*bheight lt*) *else bheight lt*)

primrec *inv1* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv1 Empty = *True*
| *inv1* (*Branch c lt k v rt*) ⟷ *inv1 lt* ∧ *inv1 rt* ∧ (*c* = *B* ∨ *color-of lt* = *B* ∧ *color-of rt* = *B*)

primrec *inv1l* :: ('*a*, '*b*) *rbt* ⇒ *bool* — Weaker version
where
inv1l Empty = *True*
| *inv1l* (*Branch c l k v r*) = (*inv1 l* ∧ *inv1 r*)
lemma [*simp*]: *inv1 t* ⟹ *inv1l t* **by** (*cases t*) *simp+*

primrec *inv2* :: ('*a*, '*b*) *rbt* ⇒ *bool*
where
inv2 Empty = *True*
| *inv2* (*Branch c lt k v rt*) = (*inv2 lt* ∧ *inv2 rt* ∧ *bheight lt* = *bheight rt*)

context *ord* **begin**

definition *is-rbt* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**
is-rbt t ⟷ *inv1 t* ∧ *inv2 t* ∧ *color-of t* = *B* ∧ *rbt-sorted t*

lemma *is-rbt-rbt-sorted* [*simp*]:
is-rbt t ⟹ *rbt-sorted t* **by** (*simp add*: *is-rbt-def*)

theorem *Empty-is-rbt* [*simp*]:
is-rbt Empty **by** (*simp add*: *is-rbt-def*)

end

130.3 Insertion

The function definitions are based on the book by Okasaki.

fun
balance :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
balance (*Branch R a w x b*) *s t* (*Branch R c y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R* (*Branch R a w x b*) *s t c*) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance (*Branch R a w x* (*Branch R b s t c*)) *y z d* = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R b s t* (*Branch R c y z d*)) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a w x (*Branch R* (*Branch R b s t c*) *y z d*) = *Branch R* (*Branch B a w x b*) *s t* (*Branch B c y z d*) |
balance a s t b = *Branch B a s t b*

lemma *balance-inv1*: $[\![inv1l\ l;\ inv1l\ r]\!] \Longrightarrow inv1\ (balance\ l\ k\ v\ r)$
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-bheight*: $bheight\ l = bheight\ r \Longrightarrow bheight\ (balance\ l\ k\ v\ r) = Suc\ (bheight\ l)$
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *balance-inv2*:
assumes *inv2 l inv2 r bheight l* = *bheight r*
shows *inv2* (*balance l k v r*)
using *assms*
by (*induct l k v r rule*: *balance.induct*) *auto*

context *ord* **begin**

lemma *balance-rbt-greater*[*simp*]: $(v\ «|\ balance\ a\ k\ x\ b) = (v\ «|\ a \wedge v\ «|\ b \wedge v < k)$
by (*induct a k x b rule*: *balance.induct*) *auto*

lemma *balance-rbt-less*[*simp*]: $(balance\ a\ k\ x\ b\ |«\ v) = (a\ |«\ v \wedge b\ |«\ v \wedge k < v)$
by (*induct a k x b rule*: *balance.induct*) *auto*

end

lemma (**in** *linorder*) *balance-rbt-sorted*:
fixes $k :: 'a$
assumes *rbt-sorted l rbt-sorted r l* $|«$ *k k* $«|$ *r*
shows *rbt-sorted* (*balance l k v r*)
using *assms* **proof** (*induct l k v r rule*: *balance.induct*)
case (*2-2 a x w b y t c z s va vb vd vc*)
hence $y < z \wedge z\ «|$ *Branch B va vb vd vc*
by (*auto simp add*: *rbt-ord-props*)
hence *y* $«|$ (*Branch B va vb vd vc*) **by** (*blast dest*: *rbt-greater-trans*)
with *2-2* **show** *?case* **by** *simp*
next
case (*3-2 va vb vd vc x w b y s c z*)
from *3-2* **have** $x < y \wedge$ *Branch B va vb vd vc* $|«$ *x*
by *simp*
hence *Branch B va vb vd vc* $|«$ *y* **by** (*blast dest*: *rbt-less-trans*)
with *3-2* **show** *?case* **by** *simp*
next
case (*3-3 x w b y s c z t va vb vd vc*)
from *3-3* **have** $y < z \wedge z\ «|$ *Branch B va vb vd vc* **by** *simp*
hence *y* $«|$ *Branch B va vb vd vc* **by** (*blast dest*: *rbt-greater-trans*)
with *3-3* **show** *?case* **by** *simp*
next
case (*3-4 vd ve vg vf x w b y s c z t va vb vii vc*)
hence $x < y \wedge$ *Branch B vd ve vg vf* $|«$ *x* **by** *simp*

hence *1*: *Branch B vd ve vg vf* |« *y* **by** (*blast dest*: *rbt-less-trans*)
from *3-4* **have** *y* < *z* ∧ *z* «| *Branch B va vb vii vc* **by** *simp*
hence *y* «| *Branch B va vb vii vc* **by** (*blast dest*: *rbt-greater-trans*)
with *1 3-4* **show** *?case* **by** *simp*
next
case (*4-2 va vb vd vc x w b y s c z t dd*)
hence *x* < *y* ∧ *Branch B va vb vd vc* |« *x* **by** *simp*
hence *Branch B va vb vd vc* |« *y* **by** (*blast dest*: *rbt-less-trans*)
with *4-2* **show** *?case* **by** *simp*
next
case (*5-2 x w b y s c z t va vb vd vc*)
hence *y* < *z* ∧ *z* «| *Branch B va vb vd vc* **by** *simp*
hence *y* «| *Branch B va vb vd vc* **by** (*blast dest*: *rbt-greater-trans*)
with *5-2* **show** *?case* **by** *simp*
next
case (*5-3 va vb vd vc x w b y s c z t*)
hence *x* < *y* ∧ *Branch B va vb vd vc* |« *x* **by** *simp*
hence *Branch B va vb vd vc* |« *y* **by** (*blast dest*: *rbt-less-trans*)
with *5-3* **show** *?case* **by** *simp*
next
case (*5-4 va vb vg vc x w b y s c z t vd ve vii vf*)
hence *x* < *y* ∧ *Branch B va vb vg vc* |« *x* **by** *simp*
hence *1*: *Branch B va vb vg vc* |« *y* **by** (*blast dest*: *rbt-less-trans*)
from *5-4* **have** *y* < *z* ∧ *z* «| *Branch B vd ve vii vf* **by** *simp*
hence *y* «| *Branch B vd ve vii vf* **by** (*blast dest*: *rbt-greater-trans*)
with *1 5-4* **show** *?case* **by** *simp*
qed *simp+*

lemma *entries-balance* [*simp*]:
entries (*balance l k v r*) = *entries l* @ (*k*, *v*) # *entries r*
by (*induct l k v r rule*: *balance.induct*) *auto*

lemma *keys-balance* [*simp*]:
keys (*balance l k v r*) = *keys l* @ *k* # *keys r*
by (*simp add*: *keys-def*)

lemma *balance-in-tree*:
entry-in-tree k x (*balance l v y r*) ⟷ *entry-in-tree k x l* ∨ *k* = *v* ∧ *x* = *y* ∨ *entry-in-tree k x r*
by (*auto simp add*: *keys-def*)

lemma (**in** *linorder*) *rbt-lookup-balance*[*simp*]:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l* |« *k k* «| *r*
shows *rbt-lookup* (*balance l k v r*) *x* = *rbt-lookup* (*Branch B l k v r*) *x*
by (*rule rbt-lookup-from-in-tree*) (*auto simp*:*assms balance-in-tree balance-rbt-sorted*)

primrec *paint* :: *color* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where

paint c Empty = Empty
| *paint c (Branch - l k v r) = Branch c l k v r*

lemma *paint-inv1l*[*simp*]: *inv1l t ⟹ inv1l (paint c t)* **by** (*cases t*) *auto*
lemma *paint-inv1*[*simp*]: *inv1l t ⟹ inv1 (paint B t)* **by** (*cases t*) *auto*
lemma *paint-inv2*[*simp*]: *inv2 t ⟹ inv2 (paint c t)* **by** (*cases t*) *auto*
lemma *paint-color-of*[*simp*]: *color-of (paint B t) = B* **by** (*cases t*) *auto*
lemma *paint-in-tree*[*simp*]: *entry-in-tree k x (paint c t) = entry-in-tree k x t* **by** (*cases t*) *auto*

context *ord* **begin**

lemma *paint-rbt-sorted*[*simp*]: *rbt-sorted t ⟹ rbt-sorted (paint c t)* **by** (*cases t*) *auto*
lemma *paint-rbt-lookup*[*simp*]: *rbt-lookup (paint c t) = rbt-lookup t* **by** (*rule ext*) (*cases t*, *auto*)
lemma *paint-rbt-greater*[*simp*]: *(v «| paint c t) = (v «| t)* **by** (*cases t*) *auto*
lemma *paint-rbt-less*[*simp*]: *(paint c t |« v) = (t |« v)* **by** (*cases t*) *auto*

fun
rbt-ins :: *('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt*
where
rbt-ins f k v Empty = Branch R Empty k v Empty |
rbt-ins f k v (Branch B l x y r) = (if k < x then balance (rbt-ins f k v l) x y r
else if k > x then balance l x y (rbt-ins f k v r)
else Branch B l x (f k y v) r) |
rbt-ins f k v (Branch R l x y r) = (if k < x then Branch R (rbt-ins f k v l) x y r
else if k > x then Branch R l x y (rbt-ins f k v r)
else Branch R l x (f k y v) r)

lemma *ins-inv1-inv2*:
assumes *inv1 t inv2 t*
shows *inv2 (rbt-ins f k x t) bheight (rbt-ins f k x t) = bheight t*
color-of t = B ⟹ inv1 (rbt-ins f k x t) inv1l (rbt-ins f k x t)
using *assms*
by (*induct f k x t rule*: *rbt-ins.induct*) (*auto simp*: *balance-inv1 balance-inv2 balance-bheight*)

end

context *linorder* **begin**

lemma *ins-rbt-greater*[*simp*]: *(v «| rbt-ins f (k :: 'a) x t) = (v «| t ∧ k > v)*
by (*induct f k x t rule*: *rbt-ins.induct*) *auto*
lemma *ins-rbt-less*[*simp*]: *(rbt-ins f k x t |« v) = (t |« v ∧ k < v)*
by (*induct f k x t rule*: *rbt-ins.induct*) *auto*
lemma *ins-rbt-sorted*[*simp*]: *rbt-sorted t ⟹ rbt-sorted (rbt-ins f k x t)*
by (*induct f k x t rule*: *rbt-ins.induct*) (*auto simp*: *balance-rbt-sorted*)

lemma *keys-ins*: *set* (*keys* (*rbt-ins f k v t*)) = { *k* } ∪ *set* (*keys t*)
by (*induct f k v t rule*: *rbt-ins.induct*) *auto*

lemma *rbt-lookup-ins*:
fixes *k* :: *'a*
assumes *rbt-sorted t*
shows *rbt-lookup* (*rbt-ins f k v t*) *x* = ((*rbt-lookup t*)(*k* |−> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
using *assms* **by** (*induct f k v t rule*: *rbt-ins.induct*) *auto*

end

context *ord* **begin**

definition *rbt-insert-with-key* :: (*'a* ⇒ *'b* ⇒ *'b* ⇒ *'b*) ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where *rbt-insert-with-key f k v t* = *paint B* (*rbt-ins f k v t*)

definition *rbt-insertw-def*: *rbt-insert-with f* = *rbt-insert-with-key* (λ-. *f*)

definition *rbt-insert* :: *'a* ⇒ *'b* ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
rbt-insert = *rbt-insert-with-key* (λ- - *nv*. *nv*)

end

context *linorder* **begin**

lemma *rbt-insertwk-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with-key f* (*k* :: *'a*) *x t*)
by (*auto simp*: *rbt-insert-with-key-def*)

theorem *rbt-insertwk-is-rbt*:
assumes *inv*: *is-rbt t*
shows *is-rbt* (*rbt-insert-with-key f k x t*)
using *assms*
unfolding *rbt-insert-with-key-def is-rbt-def*
by (*auto simp*: *ins-inv1-inv2*)

lemma *rbt-lookup-rbt-insertwk*:
assumes *rbt-sorted t*
shows *rbt-lookup* (*rbt-insert-with-key f k v t*) *x* = ((*rbt-lookup t*)(*k* |−> *case rbt-lookup t k of None* ⇒ *v*
| *Some w* ⇒ *f k w v*)) *x*
unfolding *rbt-insert-with-key-def* **using** *assms*
by (*simp add*:*rbt-lookup-ins*)

lemma *rbt-insertw-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert-with f k v t*)
by (*simp add*: *rbt-insertwk-rbt-sorted rbt-insertw-def*)

theorem *rbt-insertw-is-rbt*: *is-rbt t* ⟹ *is-rbt* (*rbt-insert-with f k v t*)
by (*simp add*: *rbt-insertwk-is-rbt rbt-insertw-def*)

lemma *rbt-lookup-rbt-insertw*:
is-rbt t ⟹
rbt-lookup (*rbt-insert-with f k v t*) =
(*rbt-lookup t*)(*k* ↦ (*if k* ∈ *dom* (*rbt-lookup t*) *then f* (*the* (*rbt-lookup t k*)) *v else v*))
by (*rule ext*, *cases rbt-lookup t k*) (*auto simp*: *rbt-lookup-rbt-insertwk dom-def rbt-insertw-def*)

lemma *rbt-insert-rbt-sorted*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-insert k v t*)
by (*simp add*: *rbt-insertwk-rbt-sorted rbt-insert-def*)
theorem *rbt-insert-is-rbt* [*simp*]: *is-rbt t* ⟹ *is-rbt* (*rbt-insert k v t*)
by (*simp add*: *rbt-insertwk-is-rbt rbt-insert-def*)

lemma *rbt-lookup-rbt-insert*: *is-rbt t* ⟹ *rbt-lookup* (*rbt-insert k v t*) = (*rbt-lookup t*)(*k*↦*v*)
by (*rule ext*) (*simp add*: *rbt-insert-def rbt-lookup-rbt-insertwk split*: *option.split*)

end

130.4 Deletion

lemma *bheight-paintR'*[*simp*]: *color-of t* = *B* ⟹ *bheight* (*paint R t*) = *bheight t* − *1*
by (*cases t rule*: *rbt-cases*) *auto*

The function definitions are based on the Haskell code by Stefan Kahrs at http://www.cs.ukc.ac.uk/people/staff/smk/redblack/rb.html.

fun
balance-left :: ('*a*,'*b*) *rbt* ⇒ '*a* ⇒ '*b* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
balance-left (*Branch R a k x b*) *s y c* = *Branch R* (*Branch B a k x b*) *s y c* |
balance-left bl k x (*Branch B a s y b*) = *balance bl k x* (*Branch R a s y b*) |
balance-left bl k x (*Branch R* (*Branch B a s y b*) *t z c*) = *Branch R* (*Branch B bl k x a*) *s y* (*balance b t z* (*paint R c*)) |
balance-left t k x s = *Empty*

lemma *balance-left-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt inv1 rt*
shows *bheight* (*balance-left lt k v rt*) = *bheight lt* + *1*
and *inv2* (*balance-left lt k v rt*)
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-left-inv2-app*:
assumes *inv2 lt inv2 rt bheight lt* + *1* = *bheight rt color-of rt* = *B*
shows *inv2* (*balance-left lt k v rt*)

bheight (*balance-left lt k v rt*) = *bheight rt*
using *assms*
by (*induct lt k v rt rule*: *balance-left.induct*) (*auto simp add*: *balance-inv2 balance-bheight*)+

lemma *balance-left-inv1*: ⟦*inv1l a*; *inv1 b*; *color-of b* = *B*⟧ ⟹ *inv1* (*balance-left a k x b*)
by (*induct a k x b rule*: *balance-left.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-left-inv1l*: ⟦ *inv1l lt*; *inv1 rt* ⟧ ⟹ *inv1l* (*balance-left lt k x rt*)
by (*induct lt k x rt rule*: *balance-left.induct*) (*auto simp*: *balance-inv1*)

lemma (**in** *linorder*) *balance-left-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* «| *r* ⟧ ⟹ *rbt-sorted* (*balance-left l k v r*)
apply (*induct l k v r rule*: *balance-left.induct*)
apply (*auto simp*: *balance-rbt-sorted*)
apply (*unfold rbt-greater-prop rbt-less-prop*)
by *force*+

context *order* **begin**

lemma *balance-left-rbt-greater*:
fixes *k* :: *'a*
assumes *k* «| *a k* «| *b k* < *x*
shows *k* «| *balance-left a x t b*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

lemma *balance-left-rbt-less*:
fixes *k* :: *'a*
assumes *a* |« *k b* |« *k x* < *k*
shows *balance-left a x t b* |« *k*
using *assms*
by (*induct a x t b rule*: *balance-left.induct*) *auto*

end

lemma *balance-left-in-tree*:
assumes *inv1l l inv1 r bheight l* + *1* = *bheight r*
shows *entry-in-tree k v* (*balance-left l a b r*) = (*entry-in-tree k v l* ∨ *k* = *a* ∧ *v* = *b* ∨ *entry-in-tree k v r*)
using *assms*
by (*induct l k v r rule*: *balance-left.induct*) (*auto simp*: *balance-in-tree*)

fun
balance-right :: (*'a*,*'b*) *rbt* ⇒ *'a* ⇒ *'b* ⇒ (*'a*,*'b*) *rbt* ⇒ (*'a*,*'b*) *rbt*
where
balance-right a k x (*Branch R b s y c*) = *Branch R a k x* (*Branch B b s y c*) |

balance-right (*Branch B a k x b*) *s y bl* = *balance* (*Branch R a k x b*) *s y bl* |
balance-right (*Branch R a k x* (*Branch B b s y c*)) *t z bl* = *Branch R* (*balance* (*paint R a*) *k x b*) *s y* (*Branch B c t z bl*) |
balance-right t k x s = *Empty*

lemma *balance-right-inv2-with-inv1*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt* + *1 inv1 lt*
shows *inv2* (*balance-right lt k v rt*) ∧ *bheight* (*balance-right lt k v rt*) = *bheight lt*
using *assms*
by (*induct lt k v rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv2 balance-bheight*)

lemma *balance-right-inv1*: ⟦*inv1 a*; *inv1l b*; *color-of a* = *B*⟧ ⟹ *inv1* (*balance-right a k x b*)
by (*induct a k x b rule*: *balance-right.induct*) (*simp add*: *balance-inv1*)+

lemma *balance-right-inv1l*: ⟦ *inv1 lt*; *inv1l rt* ⟧ ⟹*inv1l* (*balance-right lt k x rt*)
by (*induct lt k x rt rule*: *balance-right.induct*) (*auto simp*: *balance-inv1*)

lemma (**in** *linorder*) *balance-right-rbt-sorted*:
⟦ *rbt-sorted l*; *rbt-sorted r*; *rbt-less k l*; *k* «| *r* ⟧ ⟹ *rbt-sorted* (*balance-right l k v r*)
apply (*induct l k v r rule*: *balance-right.induct*)
apply (*auto simp*:*balance-rbt-sorted*)
apply (*unfold rbt-less-prop rbt-greater-prop*)
by *force*+

context *order* **begin**

lemma *balance-right-rbt-greater*:
fixes *k* :: *'a*
assumes *k* «| *a k* «| *b k* < *x*
shows *k* «| *balance-right a x t b*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

lemma *balance-right-rbt-less*:
fixes *k* :: *'a*
assumes *a* |« *k b* |« *k x* < *k*
shows *balance-right a x t b* |« *k*
using *assms* **by** (*induct a x t b rule*: *balance-right.induct*) *auto*

end

lemma *balance-right-in-tree*:
assumes *inv1 l inv1l r bheight l* = *bheight r* + *1 inv2 l inv2 r*
shows *entry-in-tree x y* (*balance-right l k v r*) = (*entry-in-tree x y l* ∨ *x* = *k* ∧ *y* = *v* ∨ *entry-in-tree x y r*)
using *assms* **by** (*induct l k v r rule*: *balance-right.induct*) (*auto simp*: *balance-in-tree*)

fun
combine :: ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt* ⇒ ('*a*,'*b*) *rbt*
where
combine Empty x = *x*
| *combine x Empty* = *x*
| *combine* (*Branch R a k x b*) (*Branch R c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ (*Branch R* (*Branch R a k x b2*) *t z* (*Branch R c2 s y d*)) |
bc ⇒ *Branch R a k x* (*Branch R bc s y d*))
| *combine* (*Branch B a k x b*) (*Branch B c s y d*) = (*case* (*combine b c*) *of*
Branch R b2 t z c2 ⇒ *Branch R* (*Branch B a k x b2*) *t z* (*Branch B c2 s y d*) |
bc ⇒ *balance-left a k x* (*Branch B bc s y d*))
| *combine a* (*Branch R b k x c*) = *Branch R* (*combine a b*) *k x c*
| *combine* (*Branch R a k x b*) *c* = *Branch R a k x* (*combine b c*)

lemma *combine-inv2*:
assumes *inv2 lt inv2 rt bheight lt* = *bheight rt*
shows *bheight* (*combine lt rt*) = *bheight lt inv2* (*combine lt rt*)
using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv2-app split*: *rbt.splits color.splits*)

lemma *combine-inv1*:
assumes *inv1 lt inv1 rt*
shows *color-of lt* = *B* ⟹ *color-of rt* = *B* ⟹ *inv1* (*combine lt rt*)
inv1l (*combine lt rt*)
using *assms*
by (*induct lt rt rule*: *combine.induct*)
(*auto simp*: *balance-left-inv1 split*: *rbt.splits color.splits*)

context *linorder* **begin**

lemma *combine-rbt-greater*[*simp*]:
fixes *k* :: '*a*
assumes *k* «| *l k* «| *r*
shows *k* «| *combine l r*
using *assms*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *balance-left-rbt-greater split*:*rbt.splits color.splits*)

lemma *combine-rbt-less*[*simp*]:
fixes *k* :: '*a*
assumes *l* |« *k r* |« *k*
shows *combine l r* |« *k*
using *assms*
by (*induct l r rule*: *combine.induct*)
(*auto simp*: *balance-left-rbt-less split*:*rbt.splits color.splits*)

lemma *combine-rbt-sorted*:
fixes *k* :: *'a*
assumes *rbt-sorted l rbt-sorted r l* |« *k k* «| *r*
shows *rbt-sorted* (*combine l r*)
using *assms* **proof** (*induct l r rule*: *combine.induct*)
case (*3 a x v b c y w d*)
hence *ineqs*: *a* |« *x x* «| *b b* |« *k k* «| *c c* |« *y y* «| *d*
by *auto*
with *3*
show *?case*
by (*cases combine b c rule*: *rbt-cases*)
(*auto*, (*metis combine-rbt-greater combine-rbt-less ineqs ineqs rbt-less-simps*(*2*) *rbt-greater-simps*(*2*) *rbt-greater-trans rbt-less-trans*)+)
next
case (*4 a x v b c y w d*)
hence *x* < *k* ∧ *rbt-greater k c* **by** *simp*
hence *rbt-greater x c* **by** (*blast dest*: *rbt-greater-trans*)
with *4* **have** *2*: *rbt-greater x* (*combine b c*) **by** (*simp add*: *combine-rbt-greater*)
from *4* **have** *k* < *y* ∧ *rbt-less k b* **by** *simp*
hence *rbt-less y b* **by** (*blast dest*: *rbt-less-trans*)
with *4* **have** *3*: *rbt-less y* (*combine b c*) **by** (*simp add*: *combine-rbt-less*)
show *?case*
proof (*cases combine b c rule*: *rbt-cases*)
case *Empty*
from *4* **have** *x* < *y* ∧ *rbt-greater y d* **by** *auto*
hence *rbt-greater x d* **by** (*blast dest*: *rbt-greater-trans*)
with *4 Empty* **have** *rbt-sorted a* **and** *rbt-sorted* (*Branch B Empty y w d*)
and *rbt-less x a* **and** *rbt-greater x* (*Branch B Empty y w d*) **by** *auto*
with *Empty* **show** *?thesis* **by** (*simp add*: *balance-left-rbt-sorted*)
next
case (*Red lta va ka rta*)
with *2 4* **have** *x* < *va* ∧ *rbt-less x a* **by** *simp*
hence *5*: *rbt-less va a* **by** (*blast dest*: *rbt-less-trans*)
from *Red 3 4* **have** *va* < *y* ∧ *rbt-greater y d* **by** *simp*
hence *rbt-greater va d* **by** (*blast dest*: *rbt-greater-trans*)
with *Red 2 3 4 5* **show** *?thesis* **by** *simp*
next
case (*Black lta va ka rta*)
from *4* **have** *x* < *y* ∧ *rbt-greater y d* **by** *auto*
hence *rbt-greater x d* **by** (*blast dest*: *rbt-greater-trans*)
with *Black 2 3 4* **have** *rbt-sorted a* **and** *rbt-sorted* (*Branch B* (*combine b c*) *y w d*)
and *rbt-less x a* **and** *rbt-greater x* (*Branch B* (*combine b c*) *y w d*) **by** *auto*
with *Black* **show** *?thesis* **by** (*simp add*: *balance-left-rbt-sorted*)
qed
next
case (*5 va vb vd vc b x w c*)
hence *k* < *x* ∧ *rbt-less k* (*Branch B va vb vd vc*) **by** *simp*
hence *rbt-less x* (*Branch B va vb vd vc*) **by** (*blast dest*: *rbt-less-trans*)

```
  with 5 show ?case by (simp add: combine-rbt-less)
next
  case (6 a x v b va vb vd vc)
  hence x < k ∧ rbt-greater k (Branch B va vb vd vc) by simp
  hence rbt-greater x (Branch B va vb vd vc) by (blast dest: rbt-greater-trans)
  with 6 show ?case by (simp add: combine-rbt-greater)
qed simp+

end

lemma combine-in-tree:
  assumes inv2 l inv2 r bheight l = bheight r inv1 l inv1 r
  shows entry-in-tree k v (combine l r) = (entry-in-tree k v l ∨ entry-in-tree k v r)
using assms
proof (induct l r rule: combine.induct)
  case (4 - - - b c)
  hence a: bheight (combine b c) = bheight b by (simp add: combine-inv2)
  from 4 have b: inv1l (combine b c) by (simp add: combine-inv1)

  show ?case
  proof (cases combine b c rule: rbt-cases)
    case Empty
    with 4 a show ?thesis by (auto simp: balance-left-in-tree)
  next
    case (Red lta ka va rta)
    with 4 show ?thesis by auto
  next
    case (Black lta ka va rta)
    with a b 4  show ?thesis by (auto simp: balance-left-in-tree)
  qed
qed (auto split: rbt.splits color.splits)

context ord begin

fun
  rbt-del-from-left :: 'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt and
  rbt-del-from-right :: 'a ⇒ ('a,'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt and
  rbt-del :: 'a⇒ ('a,'b) rbt ⇒ ('a,'b) rbt
where
  rbt-del x Empty = Empty |
  rbt-del x (Branch c a y s b) =
   (if x < y then rbt-del-from-left x a y s b
    else (if x > y then rbt-del-from-right x a y s b else combine a b)) |
  rbt-del-from-left x (Branch B lt z v rt) y s b = balance-left (rbt-del x (Branch B
lt z v rt)) y s b |
  rbt-del-from-left x a y s b = Branch R (rbt-del x a) y s b |
  rbt-del-from-right x a y s (Branch B lt z v rt) = balance-right a y s (rbt-del x
(Branch B lt z v rt)) |
  rbt-del-from-right x a y s b = Branch R a y s (rbt-del x b)
```

end

context *linorder* **begin**

lemma
assumes *inv2 lt inv1 lt*
shows
⟦*inv2 rt*; *bheight lt* = *bheight rt*; *inv1 rt*⟧ ⟹
inv2 (*rbt-del-from-left x lt k v rt*) ∧
bheight (*rbt-del-from-left x lt k v rt*) = *bheight lt* ∧
(*color-of lt* = *B* ∧ *color-of rt* = *B* ∧ *inv1* (*rbt-del-from-left x lt k v rt*) ∨
(*color-of lt* ≠ *B* ∨ *color-of rt* ≠ *B*) ∧ *inv1l* (*rbt-del-from-left x lt k v rt*))
and ⟦*inv2 rt*; *bheight lt* = *bheight rt*; *inv1 rt*⟧ ⟹
inv2 (*rbt-del-from-right x lt k v rt*) ∧
bheight (*rbt-del-from-right x lt k v rt*) = *bheight lt* ∧
(*color-of lt* = *B* ∧ *color-of rt* = *B* ∧ *inv1* (*rbt-del-from-right x lt k v rt*) ∨
(*color-of lt* ≠ *B* ∨ *color-of rt* ≠ *B*) ∧ *inv1l* (*rbt-del-from-right x lt k v rt*))
and *rbt-del-inv1-inv2*: *inv2* (*rbt-del x lt*) ∧ (*color-of lt* = *R* ∧ *bheight* (*rbt-del x lt*) = *bheight lt* ∧ *inv1* (*rbt-del x lt*)
∨ *color-of lt* = *B* ∧ *bheight* (*rbt-del x lt*) = *bheight lt* − *1* ∧ *inv1l* (*rbt-del x lt*))
using *assms*
proof (*induct x lt k v rt* **and** *x lt k v rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
case (*2 y c - y'*)
have *y* = *y'* ∨ *y* < *y'* ∨ *y* > *y'* **by** *auto*
thus *?case* **proof** (*elim disjE*)
assume *y* = *y'*
with *2* **show** *?thesis* **by** (*cases c*) (*simp add*: *combine-inv2 combine-inv1*)+
next
assume *y* < *y'*
with *2* **show** *?thesis* **by** (*cases c*) *auto*
next
assume *y'* < *y*
with *2* **show** *?thesis* **by** (*cases c*) *auto*
qed
next
case (*3 y lt z v rta y' ss bb*)
thus *?case* **by** (*cases color-of* (*Branch B lt z v rta*) = *B* ∧ *color-of bb* = *B*) (*simp add*: *balance-left-inv2-with-inv1 balance-left-inv1 balance-left-inv1l*)+
next
case (*5 y a y' ss lt z v rta*)
thus *?case* **by** (*cases color-of a* = *B* ∧ *color-of* (*Branch B lt z v rta*) = *B*) (*simp add*: *balance-right-inv2-with-inv1 balance-right-inv1 balance-right-inv1l*)+
next
case (*6-1 y a y' ss*) **thus** *?case* **by** (*cases color-of a* = *B* ∧ *color-of Empty* = *B*) *simp*+
qed *auto*

lemma

rbt-del-from-left-rbt-less: $[\![lt \mid\!\ll v; rt \mid\!\ll v; k < v]\!] \Longrightarrow$ *rbt-del-from-left x lt k y rt* $\mid\!\ll v$
and *rbt-del-from-right-rbt-less*: $[\![lt \mid\!\ll v; rt \mid\!\ll v; k < v]\!] \Longrightarrow$ *rbt-del-from-right x lt k y rt* $\mid\!\ll v$
and *rbt-del-rbt-less*: $lt \mid\!\ll v \Longrightarrow$ *rbt-del x lt* $\mid\!\ll v$
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)

(*auto simp*: *balance-left-rbt-less balance-right-rbt-less*)

lemma *rbt-del-from-left-rbt-greater*: $[\![v \ll\!| lt; v \ll\!| rt; k > v]\!] \Longrightarrow v \ll\!|$ *rbt-del-from-left x lt k y rt*
and *rbt-del-from-right-rbt-greater*: $[\![v \ll\!| lt; v \ll\!| rt; k > v]\!] \Longrightarrow v \ll\!|$ *rbt-del-from-right x lt k y rt*
and *rbt-del-rbt-greater*: $v \ll\!| lt \Longrightarrow v \ll\!|$ *rbt-del x lt*
by (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
(*auto simp*: *balance-left-rbt-greater balance-right-rbt-greater*)

lemma $[\![$*rbt-sorted lt*; *rbt-sorted rt*; $lt \mid\!\ll k; k \ll\!| rt]\!] \Longrightarrow$ *rbt-sorted* (*rbt-del-from-left x lt k y rt*)
and $[\![$*rbt-sorted lt*; *rbt-sorted rt*; $lt \mid\!\ll k; k \ll\!| rt]\!] \Longrightarrow$ *rbt-sorted* (*rbt-del-from-right x lt k y rt*)
and *rbt-del-rbt-sorted*: *rbt-sorted lt* $\Longrightarrow$ *rbt-sorted* (*rbt-del x lt*)
proof (*induct x lt k y rt* **and** *x lt k y rt* **and** *x lt rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
case (*3 x lta zz v rta yy ss bb*)
from *3* **have** *Branch B lta zz v rta* $\mid\!\ll$ *yy* **by** *simp*
hence *rbt-del x* (*Branch B lta zz v rta*) $\mid\!\ll$ *yy* **by** (*rule rbt-del-rbt-less*)
with *3* **show** *?case* **by** (*simp add*: *balance-left-rbt-sorted*)
next
case (*4-2 x vaa vbb vdd vc yy ss bb*)
hence *Branch R vaa vbb vdd vc* $\mid\!\ll$ *yy* **by** *simp*
hence *rbt-del x* (*Branch R vaa vbb vdd vc*) $\mid\!\ll$ *yy* **by** (*rule rbt-del-rbt-less*)
with *4-2* **show** *?case* **by** *simp*
next
case (*5 x aa yy ss lta zz v rta*)
hence *yy* $\ll\!|$ *Branch B lta zz v rta* **by** *simp*
hence *yy* $\ll\!|$ *rbt-del x* (*Branch B lta zz v rta*) **by** (*rule rbt-del-rbt-greater*)
with *5* **show** *?case* **by** (*simp add*: *balance-right-rbt-sorted*)
next
case (*6-2 x aa yy ss vaa vbb vdd vc*)
hence *yy* $\ll\!|$ *Branch R vaa vbb vdd vc* **by** *simp*
hence *yy* $\ll\!|$ *rbt-del x* (*Branch R vaa vbb vdd vc*) **by** (*rule rbt-del-rbt-greater*)
with *6-2* **show** *?case* **by** *simp*
qed (*auto simp*: *combine-rbt-sorted*)

lemma $[\![$*rbt-sorted lt*; *rbt-sorted rt*; $lt \mid\!\ll kt; kt \ll\!| rt$; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* $=$ *bheight rt*; $x < kt]\!] \Longrightarrow$ *entry-in-tree k v* (*rbt-del-from-left x lt kt y rt*) $=$ (*False* $\vee$ ($x \neq k \wedge$ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and $[\![$*rbt-sorted lt*; *rbt-sorted rt*; $lt \mid\!\ll kt; kt \ll\!| rt$; *inv1 lt*; *inv1 rt*; *inv2 lt*; *inv2 rt*; *bheight lt* $=$ *bheight rt*; $x > kt]\!] \Longrightarrow$ *entry-in-tree k v* (*rbt-del-from-right x lt kt*

y rt) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v* (*Branch c lt kt y rt*)))
and *rbt-del-in-tree*: ⟦*rbt-sorted t*; *inv1 t*; *inv2 t*⟧ ⟹ *entry-in-tree k v* (*rbt-del x t*) = (*False* ∨ (*x* ≠ *k* ∧ *entry-in-tree k v t*))
proof (*induct x lt kt y rt* **and** *x lt kt y rt* **and** *x t rule*: *rbt-del-from-left-rbt-del-from-right-rbt-del.induct*)
case (*2 xx c aa yy ss bb*)
have *xx* = *yy* ∨ *xx* < *yy* ∨ *xx* > *yy* **by** *auto*
from *this 2* **show** *?case* **proof** (*elim disjE*)
assume *xx* = *yy*
with *2* **show** *?thesis* **proof** (*cases xx* = *k*)
case *True*
from *2* ‹*xx* = *yy*› ‹*xx* = *k*› **have** *rbt-sorted* (*Branch c aa yy ss bb*) ∧ *k* = *yy* **by** *simp*
hence ¬ *entry-in-tree k v aa* ¬ *entry-in-tree k v bb* **by** (*auto simp*: *rbt-less-nit rbt-greater-prop*)
with ‹*xx* = *yy*› *2* ‹*xx* = *k*› **show** *?thesis* **by** (*simp add*: *combine-in-tree*)
qed (*simp add*: *combine-in-tree*)
qed *simp+*
next
case (*3 xx lta zz vv rta yy ss bb*)
define *mt* **where** [*simp*]: *mt* = *Branch B lta zz vv rta*
from *3* **have** *inv2 mt* ∧ *inv1 mt* **by** *simp*
hence *inv2* (*rbt-del xx mt*) ∧ (*color-of mt* = *R* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* ∧ *inv1* (*rbt-del xx mt*) ∨ *color-of mt* = *B* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* − *1* ∧ *inv1l* (*rbt-del xx mt*)) **by** (*blast dest*: *rbt-del-inv1-inv2*)
with *3* **have** *4*: *entry-in-tree k v* (*rbt-del-from-left xx mt yy ss bb*) = (*False* ∨ *xx* ≠ *k* ∧ *entry-in-tree k v mt* ∨ (*k* = *yy* ∧ *v* = *ss*) ∨ *entry-in-tree k v bb*) **by** (*simp add*: *balance-left-in-tree*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
from *3 True* **have** *yy* «| *bb* ∧ *yy* > *k* **by** *simp*
hence *k* «| *bb* **by** (*blast dest*: *rbt-greater-trans*)
with *3 4 True* **show** *?thesis* **by** (*auto simp*: *rbt-greater-nit*)
qed *auto*
next
case (*4-1 xx yy ss bb*)
show *?case* **proof** (*cases xx* = *k*)
case *True*
with *4-1* **have** *yy* «| *bb* ∧ *k* < *yy* **by** *simp*
hence *k* «| *bb* **by** (*blast dest*: *rbt-greater-trans*)
with *4-1* ‹*xx* = *k*›
have *entry-in-tree k v* (*Branch R Empty yy ss bb*) = *entry-in-tree k v Empty* **by** (*auto simp*: *rbt-greater-nit*)
thus *?thesis* **by** *auto*
qed *simp+*
next
case (*4-2 xx vaa vbb vdd vc yy ss bb*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
with *4-2* **have** *k* < *yy* ∧ *yy* «| *bb* **by** *simp*

hence *k «| bb* **by** (*blast dest*: *rbt-greater-trans*)
with *True 4-2* **show** *?thesis* **by** (*auto simp*: *rbt-greater-nit*)
qed *auto*
next
case (*5 xx aa yy ss lta zz vv rta*)
define *mt* **where** [*simp*]: *mt* = *Branch B lta zz vv rta*
from *5* **have** *inv2 mt* ∧ *inv1 mt* **by** *simp*
hence *inv2* (*rbt-del xx mt*) ∧ (*color-of mt* = *R* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* ∧ *inv1* (*rbt-del xx mt*) ∨ *color-of mt* = *B* ∧ *bheight* (*rbt-del xx mt*) = *bheight mt* − *1* ∧ *inv1l* (*rbt-del xx mt*)) **by** (*blast dest*: *rbt-del-inv1-inv2*)
with *5* **have** *3*: *entry-in-tree k v* (*rbt-del-from-right xx aa yy ss mt*) = (*entry-in-tree k v aa* ∨ (*k* = *yy* ∧ *v* = *ss*) ∨ *False* ∨ *xx* ≠ *k* ∧ *entry-in-tree k v mt*) **by** (*simp add*: *balance-right-in-tree*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
from *5 True* **have** *aa |« yy* ∧ *yy* < *k* **by** *simp*
hence *aa |« k* **by** (*blast dest*: *rbt-less-trans*)
with *3 5 True* **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *auto*
next
case (*6-1 xx aa yy ss*)
show *?case* **proof** (*cases xx* = *k*)
case *True*
with *6-1* **have** *aa |« yy* ∧ *k* > *yy* **by** *simp*
hence *aa |« k* **by** (*blast dest*: *rbt-less-trans*)
with *6-1* ‹*xx* = *k*› **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *simp*
next
case (*6-2 xx aa yy ss vaa vbb vdd vc*)
thus *?case* **proof** (*cases xx* = *k*)
case *True*
with *6-2* **have** *k* > *yy* ∧ *aa |« yy* **by** *simp*
hence *aa |« k* **by** (*blast dest*: *rbt-less-trans*)
with *True 6-2* **show** *?thesis* **by** (*auto simp*: *rbt-less-nit*)
qed *auto*
qed *simp*

definition (**in** *ord*) *rbt-delete* **where**
rbt-delete k t = *paint B* (*rbt-del k t*)

theorem *rbt-delete-is-rbt* [*simp*]: **assumes** *is-rbt t* **shows** *is-rbt* (*rbt-delete k t*)
proof −
from *assms* **have** *inv2 t* **and** *inv1 t* **unfolding** *is-rbt-def* **by** *auto*
hence *inv2* (*rbt-del k t*) ∧ (*color-of t* = *R* ∧ *bheight* (*rbt-del k t*) = *bheight t* ∧ *inv1* (*rbt-del k t*) ∨ *color-of t* = *B* ∧ *bheight* (*rbt-del k t*) = *bheight t* − *1* ∧ *inv1l* (*rbt-del k t*)) **by** (*rule rbt-del-inv1-inv2*)
hence *inv2* (*rbt-del k t*) ∧ *inv1l* (*rbt-del k t*) **by** (*cases color-of t*) *auto*
with *assms* **show** *?thesis*
unfolding *is-rbt-def rbt-delete-def*

by (*auto intro*: *paint-rbt-sorted rbt-del-rbt-sorted*)
qed

lemma *rbt-delete-in-tree*:
assumes *is-rbt t*
shows *entry-in-tree k v* (*rbt-delete x t*) = (*x* ≠ *k* ∧ *entry-in-tree k v t*)
using *assms* **unfolding** *is-rbt-def rbt-delete-def*
by (*auto simp*: *rbt-del-in-tree*)

lemma *rbt-lookup-rbt-delete*:
assumes *is-rbt*: *is-rbt t*
shows *rbt-lookup* (*rbt-delete k t*) = (*rbt-lookup t*)|‘(−{*k*})
proof
fix *x*
show *rbt-lookup* (*rbt-delete k t*) *x* = (*rbt-lookup t* |‘ (−{*k*})) *x*
proof (*cases x* = *k*)
assume *x* = *k*
with *is-rbt* **show** *?thesis*
by (*cases rbt-lookup* (*rbt-delete k t*) *k*) (*auto simp*: *rbt-lookup-in-tree rbt-delete-in-tree*)
next
assume *x* ≠ *k*
thus *?thesis*
by *auto* (*metis is-rbt rbt-delete-is-rbt rbt-delete-in-tree is-rbt-rbt-sorted rbt-lookup-from-in-tree*)
qed
qed

end

130.5 Modifying existing entries

context *ord* **begin**

primrec
rbt-map-entry :: *'a* ⇒ (*'b* ⇒ *'b*) ⇒ (*'a*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt*
where
rbt-map-entry k f Empty = *Empty*
| *rbt-map-entry k f* (*Branch c lt x v rt*) =
(*if k* < *x then Branch c* (*rbt-map-entry k f lt*) *x v rt*
else if k > *x then* (*Branch c lt x v* (*rbt-map-entry k f rt*))
else Branch c lt x (*f v*) *rt*)

lemma *rbt-map-entry-color-of*: *color-of* (*rbt-map-entry k f t*) = *color-of t* **by** (*induct t*) *simp*+
lemma *rbt-map-entry-inv1*: *inv1* (*rbt-map-entry k f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *rbt-map-entry-color-of*)+
lemma *rbt-map-entry-inv2*: *inv2* (*rbt-map-entry k f t*) = *inv2 t bheight* (*rbt-map-entry k f t*) = *bheight t* **by** (*induct t*) *simp*+
lemma *rbt-map-entry-rbt-greater*: *rbt-greater a* (*rbt-map-entry k f t*) = *rbt-greater*

a t **by** (*induct t*) *simp+*
lemma *rbt-map-entry-rbt-less*: *rbt-less a* (*rbt-map-entry k f t*) = *rbt-less a t* **by** (*induct t*) *simp+*
lemma *rbt-map-entry-rbt-sorted*: *rbt-sorted* (*rbt-map-entry k f t*) = *rbt-sorted t*
by (*induct t*) (*simp-all add*: *rbt-map-entry-rbt-less rbt-map-entry-rbt-greater*)

theorem *rbt-map-entry-is-rbt* [*simp*]: *is-rbt* (*rbt-map-entry k f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *rbt-map-entry-inv2 rbt-map-entry-color-of rbt-map-entry-rbt-sorted rbt-map-entry-inv1*)

end

theorem (**in** *linorder*) *rbt-lookup-rbt-map-entry*:
rbt-lookup (*rbt-map-entry k f t*) = (*rbt-lookup t*)(*k* := *map-option f* (*rbt-lookup t k*))
by (*induct t*) (*auto split*: *option.splits simp add*: *fun-eq-iff*)

130.6 Mapping all entries

primrec
map :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*c*) *rbt*
where
map f Empty = *Empty*
| *map f* (*Branch c lt k v rt*) = *Branch c* (*map f lt*) *k* (*f k v*) (*map f rt*)

lemma *map-entries* [*simp*]: *entries* (*map f t*) = *List.map* (λ(*k*, *v*). (*k*, *f k v*)) (*entries t*)
by (*induct t*) *auto*
lemma *map-keys* [*simp*]: *keys* (*map f t*) = *keys t* **by** (*simp add*: *keys-def split-def*)
lemma *map-color-of*: *color-of* (*map f t*) = *color-of t* **by** (*induct t*) *simp+*
lemma *map-inv1*: *inv1* (*map f t*) = *inv1 t* **by** (*induct t*) (*simp add*: *map-color-of*)+
lemma *map-inv2*: *inv2* (*map f t*) = *inv2 t bheight* (*map f t*) = *bheight t* **by** (*induct t*) *simp+*

context *ord* **begin**

lemma *map-rbt-greater*: *rbt-greater k* (*map f t*) = *rbt-greater k t* **by** (*induct t*) *simp+*
lemma *map-rbt-less*: *rbt-less k* (*map f t*) = *rbt-less k t* **by** (*induct t*) *simp+*
lemma *map-rbt-sorted*: *rbt-sorted* (*map f t*) = *rbt-sorted t* **by** (*induct t*) (*simp add*: *map-rbt-less map-rbt-greater*)+
theorem *map-is-rbt* [*simp*]: *is-rbt* (*map f t*) = *is-rbt t*
unfolding *is-rbt-def* **by** (*simp add*: *map-inv1 map-inv2 map-rbt-sorted map-color-of*)

end

theorem (**in** *linorder*) *rbt-lookup-map*: *rbt-lookup* (*map f t*) *x* = *map-option* (*f x*) (*rbt-lookup t x*)
by (*induct t*) (*auto simp*: *antisym-conv3*)

hide-const (**open**) *map*

130.7 Folding over entries

definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **where**
fold f t = *List.fold* (*case-prod f*) (*entries t*)

lemma *fold-simps* [*simp*]:
fold f Empty = *id*
fold f (*Branch c lt k v rt*) = *fold f rt* ∘ *f k v* ∘ *fold f lt*
by (*simp-all add*: *fold-def fun-eq-iff*)

lemma *fold-code* [*code*]:
fold f Empty x = *x*
fold f (*Branch c lt k v rt*) *x* = *fold f rt* (*f k v* (*fold f lt x*))
by(*simp-all*)

— fold with continuation predicate
fun *foldi* :: ('*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a* :: *linorder*, '*b*) *rbt* ⇒ '*c* ⇒ '*c*
where
foldi c f Empty s = *s* |
foldi c f (*Branch col l k v r*) *s* = (
if (*c s*) *then*
let s' = *foldi c f l s in*
if (*c s'*) *then*
foldi c f r (*f k v s'*)
else s'
else
s
)

130.8 Bulkloading a tree

definition (**in** *ord*) *rbt-bulkload* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-bulkload xs = *foldr* (λ(*k*, *v*). *rbt-insert k v*) *xs Empty*

context *linorder* **begin**

lemma *rbt-bulkload-is-rbt* [*simp*, *intro*]:
is-rbt (*rbt-bulkload xs*)
unfolding *rbt-bulkload-def* **by** (*induct xs*) *auto*

lemma *rbt-lookup-rbt-bulkload*:
rbt-lookup (*rbt-bulkload xs*) = *map-of xs*
proof −
obtain *ys* **where** *ys* = *rev xs* **by** *simp*
have ⋀*t*. *is-rbt t* ⟹

rbt-lookup (*List.fold* (*case-prod rbt-insert*) *ys t*) = *rbt-lookup t* ++ *map-of* (*rev ys*)
by (*induct ys*) (*simp-all add*: *rbt-bulkload-def rbt-lookup-rbt-insert case-prod-beta*)
from *this Empty-is-rbt* **have**
rbt-lookup (*List.fold* (*case-prod rbt-insert*) (*rev xs*) *Empty*) = *rbt-lookup Empty* ++ *map-of xs*
by (*simp add*: ‹*ys* = *rev xs*›)
then show *?thesis* **by** (*simp add*: *rbt-bulkload-def rbt-lookup-Empty foldr-conv-fold*)
qed

end

130.9 Building a RBT from a sorted list

These functions have been adapted from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *rbtreeify-f* :: *nat* ⇒ ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* × ('*a* × '*b*) *list*
and *rbtreeify-g* :: *nat* ⇒ ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* × ('*a* × '*b*) *list*
where
rbtreeify-f n kvs =
(*if n* = *0 then* (*Empty*, *kvs*)
else if n = *1 then*
case kvs of (*k*, *v*) # *kvs'* ⇒ (*Branch R Empty k v Empty*, *kvs'*)
else if (*n mod 2* = *0*) *then*
case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*)
else case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-f* (*n div 2*) *kvs'*))

| *rbtreeify-g n kvs* =
(*if n* = *0* ∨ *n* = *1 then* (*Empty*, *kvs*)
else if n mod 2 = *0 then*
case rbtreeify-g (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*)
else case rbtreeify-f (*n div 2*) *kvs of* (*t1*, (*k*, *v*) # *kvs'*) ⇒
apfst (*Branch B t1 k v*) (*rbtreeify-g* (*n div 2*) *kvs'*))

definition *rbtreeify* :: ('*a* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt*
where *rbtreeify kvs* = *fst* (*rbtreeify-g* (*Suc* (*length kvs*)) *kvs*)

declare *rbtreeify-f.simps* [*simp del*] *rbtreeify-g.simps* [*simp del*]

lemma *rbtreeify-f-code* [*code*]:
rbtreeify-f n kvs =
(*if n* = *0 then* (*Empty*, *kvs*)
else if n = *1 then*
case kvs of (*k*, *v*) # *kvs'* ⇒
(*Branch R Empty k v Empty*, *kvs'*)
else let (*n'*, *r*) = *Euclidean-Rings.divmod-nat n 2 in*

```
    if r = 0 then
      case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-g n' kvs')
    else case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
        apfst (Branch B t1 k v) (rbtreeify-f n' kvs'))
by (subst rbtreeify-f.simps) (simp only: Let-def Euclidean-Rings.divmod-nat-def
prod.case)

lemma rbtreeify-g-code [code]:
  rbtreeify-g n kvs =
   (if n = 0 ∨ n = 1 then (Empty, kvs)
    else let (n', r) = Euclidean-Rings.divmod-nat n 2 in
      if r = 0 then
        case rbtreeify-g n' kvs of (t1, (k, v) # kvs') ⇒
          apfst (Branch B t1 k v) (rbtreeify-g n' kvs')
      else case rbtreeify-f n' kvs of (t1, (k, v) # kvs') ⇒
          apfst (Branch B t1 k v) (rbtreeify-g n' kvs'))
by(subst rbtreeify-g.simps)(simp only: Let-def Euclidean-Rings.divmod-nat-def prod.case)

lemma Suc-double-half: Suc (2 * n) div 2 = n
by simp

lemma div2-plus-div2: n div 2 + n div 2 = (n :: nat) - n mod 2
by arith

lemma rbtreeify-f-rec-aux-lemma:
  ⟦k - n div 2 = Suc k'; n ≤ k; n mod 2 = Suc 0⟧
  ⟹ k' - n div 2 = k - n
apply(rule add-right-imp-eq[where a = n - n div 2])
apply(subst add-diff-assoc2, arith)
apply(simp add: div2-plus-div2)
done

lemma rbtreeify-f-simps:
  rbtreeify-f 0 kvs = (Empty, kvs)
  rbtreeify-f (Suc 0) ((k, v) # kvs) =
  (Branch R Empty k v Empty, kvs)
  0 < n ⟹ rbtreeify-f (2 * n) kvs =
   (case rbtreeify-f n kvs of (t1, (k, v) # kvs') ⇒
     apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
  0 < n ⟹ rbtreeify-f (Suc (2 * n)) kvs =
   (case rbtreeify-f n kvs of (t1, (k, v) # kvs') ⇒
     apfst (Branch B t1 k v) (rbtreeify-f n kvs'))
by(subst (1) rbtreeify-f.simps, simp add: Suc-double-half)+

lemma rbtreeify-g-simps:
  rbtreeify-g 0 kvs = (Empty, kvs)
  rbtreeify-g (Suc 0) kvs = (Empty, kvs)
  0 < n ⟹ rbtreeify-g (2 * n) kvs =
```

```
   (case rbtreeify-g n kvs of (t1, (k, v) # kvs') ⇒
     apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
  0 < n ⟹ rbtreeify-g (Suc (2 * n)) kvs =
   (case rbtreeify-f n kvs of (t1, (k, v) # kvs') ⇒
     apfst (Branch B t1 k v) (rbtreeify-g n kvs'))
by(subst (1) rbtreeify-g.simps, simp add: Suc-double-half)+

declare rbtreeify-f-simps[simp] rbtreeify-g-simps[simp]

lemma length-rbtreeify-f: n ≤ length kvs
  ⟹ length (snd (rbtreeify-f n kvs)) = length kvs − n
  and length-rbtreeify-g:⟦ 0 < n; n ≤ Suc (length kvs) ⟧
  ⟹ length (snd (rbtreeify-g n kvs)) = Suc (length kvs) − n
proof(induction n kvs and n kvs rule: rbtreeify-f-rbtreeify-g.induct)
  case (1 n kvs)
  show ?case
  proof(cases n ≤ 1)
    case True thus ?thesis using 1.prems
      by(cases n kvs rule: nat.exhaust[case-product list.exhaust]) auto
  next
    case False
    hence n ≠ 0 n ≠ 1 by simp-all
    note IH = 1.IH[OF this]
    show ?thesis
    proof(cases n mod 2 = 0)
      case True
      hence length (snd (rbtreeify-f n kvs)) =
        length (snd (rbtreeify-f (2 * (n div 2)) kvs))
        by(metis minus-nat.diff-0 minus-mod-eq-mult-div [symmetric])
      also from 1.prems False obtain k v kvs'
        where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
      also have 0 < n div 2 using False by(simp)
      note rbtreeify-f-simps(3)[OF this]
      also note kvs[symmetric]
      also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
      from 1.prems have n div 2 ≤ length kvs by simp
      with True have len: length ?rest1 = length kvs − n div 2 by(rule IH)
      with 1.prems False obtain t1 k' v' kvs''
        where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
         by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
      note this also note prod.case also note list.simps(5)
      also note prod.case also note snd-apfst
      also have 0 < n div 2 n div 2 ≤ Suc (length kvs'')
        using len 1.prems False unfolding kvs'' by simp-all
      with True kvs''[symmetric] refl refl
      have length (snd (rbtreeify-g (n div 2) kvs'')) =
        Suc (length kvs'') − n div 2 by(rule IH)
      finally show ?thesis using len[unfolded kvs''] 1.prems True
      by(simp add: Suc-diff-le[symmetric] mult-2[symmetric] minus-mod-eq-mult-div
```

```
[symmetric])
    next
      case False
      hence length (snd (rbtreeify-f n kvs)) =
        length (snd (rbtreeify-f (Suc (2 * (n div 2))) kvs))
        by (simp add: mod-eq-0-iff-dvd)
      also from 1.prems ‹¬ n ≤ 1› obtain k v kvs'
        where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
      also have 0 < n div 2 using ‹¬ n ≤ 1› by(simp)
      note rbtreeify-f-simps(4)[OF this]
      also note kvs[symmetric]
      also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
      from 1.prems have n div 2 ≤ length kvs by simp
      with False have len: length ?rest1 = length kvs − n div 2 by(rule IH)
      with 1.prems ‹¬ n ≤ 1› obtain t1 k' v' kvs''
        where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
        by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
      note this also note prod.case also note list.simps(5)
      also note prod.case also note snd-apfst
      also have n div 2 ≤ length kvs''
        using len 1.prems False unfolding kvs'' by simp arith
      with False kvs''[symmetric] refl refl
      have length (snd (rbtreeify-f (n div 2) kvs'')) = length kvs'' − n div 2
        by(rule IH)
      finally show ?thesis using len[unfolded kvs''] 1.prems False
        by simp(rule rbtreeify-f-rec-aux-lemma[OF sym])
    qed
  qed
next
  case (2 n kvs)
  show ?case
  proof(cases n > 1)
    case False with ‹0 < n› show ?thesis
      by(cases n kvs rule: nat.exhaust[case-product list.exhaust]) simp-all
  next
    case True
    hence ¬ (n = 0 ∨ n = 1) by simp
    note IH = 2.IH[OF this]
    show ?thesis
    proof(cases n mod 2 = 0)
      case True
      hence length (snd (rbtreeify-g n kvs)) =
        length (snd (rbtreeify-g (2 * (n div 2)) kvs))
        by(metis minus-nat.diff-0 minus-mod-eq-mult-div [symmetric])
      also from 2.prems True obtain k v kvs'
        where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
      also have 0 < n div 2 using ‹1 < n› by(simp)
      note rbtreeify-g-simps(3)[OF this]
      also note kvs[symmetric]
```

```
    also let ?rest1 = snd (rbtreeify-g (n div 2) kvs)
    from 2.prems ‹1 < n›
    have 0 < n div 2 n div 2 ≤ Suc (length kvs) by simp-all
    with True have len: length ?rest1 = Suc (length kvs) − n div 2 by(rule IH)
    with 2.prems obtain t1 k' v' kvs''
      where kvs'': rbtreeify-g (n div 2) kvs = (t1, (k', v') # kvs'')
      by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm)
    note this also note prod.case also note list.simps(5)
    also note prod.case also note snd-apfst
    also have n div 2 ≤ Suc (length kvs'')
      using len 2.prems unfolding kvs'' by simp
    with True kvs''[symmetric] refl refl ‹0 < n div 2›
    have length (snd (rbtreeify-g (n div 2) kvs'')) = Suc (length kvs'') − n div 2
      by(rule IH)
    finally show ?thesis using len[unfolded kvs''] 2.prems True
    by(simp add: Suc-diff-le[symmetric] mult-2[symmetric] minus-mod-eq-mult-div
[symmetric])
  next
    case False
    hence length (snd (rbtreeify-g n kvs)) =
      length (snd (rbtreeify-g (Suc (2 * (n div 2))) kvs))
      by (simp add: mod-eq-0-iff-dvd)
    also from 2.prems ‹1 < n› obtain k v kvs'
      where kvs: kvs = (k, v) # kvs' by(cases kvs) auto
    also have 0 < n div 2 using ‹1 < n› by(simp)
    note rbtreeify-g-simps(4)[OF this]
    also note kvs[symmetric]
    also let ?rest1 = snd (rbtreeify-f (n div 2) kvs)
    from 2.prems have n div 2 ≤ length kvs by simp
    with False have len: length ?rest1 = length kvs − n div 2 by(rule IH)
    with 2.prems ‹1 < n› False obtain t1 k' v' kvs''
      where kvs'': rbtreeify-f (n div 2) kvs = (t1, (k', v') # kvs'')
      by(cases ?rest1)(auto simp add: snd-def split: prod.split-asm, arith)
    note this also note prod.case also note list.simps(5)
    also note prod.case also note snd-apfst
    also have n div 2 ≤ Suc (length kvs'')
      using len 2.prems False unfolding kvs'' by simp arith
    with False kvs''[symmetric] refl refl ‹0 < n div 2›
    have length (snd (rbtreeify-g (n div 2) kvs'')) = Suc (length kvs'') − n div 2
      by(rule IH)
    finally show ?thesis using len[unfolded kvs''] 2.prems False
      by(simp add: div2-plus-div2)
  qed
 qed
qed

lemma rbtreeify-induct [consumes 1, case-names f-0 f-1 f-even f-odd g-0 g-1 g-even
g-odd]:
 fixes P Q
```

```
  defines f0 == (⋀kvs. P 0 kvs)
  and f1 == (⋀k v kvs. P (Suc 0) ((k, v) # kvs))
  and feven ==
    (⋀n kvs t k v kvs'. ⟦ n > 0; n ≤ length kvs; P n kvs;
       rbtreeify-f n kvs = (t, (k, v) # kvs'); n ≤ Suc (length kvs'); Q n kvs' ⟧
     ⟹ P (2 * n) kvs)
  and fodd ==
    (⋀n kvs t k v kvs'. ⟦ n > 0; n ≤ length kvs; P n kvs;
       rbtreeify-f n kvs = (t, (k, v) # kvs'); n ≤ length kvs'; P n kvs' ⟧
    ⟹ P (Suc (2 * n)) kvs)
  and g0 == (⋀kvs. Q 0 kvs)
  and g1 == (⋀kvs. Q (Suc 0) kvs)
  and geven ==
    (⋀n kvs t k v kvs'. ⟦ n > 0; n ≤ Suc (length kvs); Q n kvs;
       rbtreeify-g n kvs = (t, (k, v) # kvs'); n ≤ Suc (length kvs'); Q n kvs' ⟧
    ⟹ Q (2 * n) kvs)
  and godd ==
    (⋀n kvs t k v kvs'. ⟦ n > 0; n ≤ length kvs; P n kvs;
       rbtreeify-f n kvs = (t, (k, v) # kvs'); n ≤ Suc (length kvs'); Q n kvs' ⟧
    ⟹ Q (Suc (2 * n)) kvs)
  shows ⟦ n ≤ length kvs;
          PROP f0; PROP f1; PROP feven; PROP fodd;
          PROP g0; PROP g1; PROP geven; PROP godd ⟧
        ⟹ P n kvs
  and ⟦ n ≤ Suc (length kvs);
          PROP f0; PROP f1; PROP feven; PROP fodd;
          PROP g0; PROP g1; PROP geven; PROP godd ⟧
      ⟹ Q n kvs
proof −
  assume f0: PROP f0 and f1: PROP f1 and feven: PROP feven and fodd:
PROP fodd
    and g0: PROP g0 and g1: PROP g1 and geven: PROP geven and godd:
PROP godd
  show n ≤ length kvs ⟹ P n kvs and n ≤ Suc (length kvs) ⟹ Q n kvs
  proof(induction rule: rbtreeify-f-rbtreeify-g.induct)
    case (1 n kvs)
    show ?case
    proof(cases n ≤ 1)
      case True thus ?thesis using 1.prems
        by(cases n kvs rule: nat.exhaust[case-product list.exhaust])
          (auto simp add: f0[unfolded f0-def] f1[unfolded f1-def])
    next
      case False
      hence ns: n ≠ 0 n ≠ 1 by simp-all
      hence ge0: n div 2 > 0 by simp
      note IH = 1.IH[OF ns]
      show ?thesis
      proof(cases n mod 2 = 0)
        case True note ge0
```

```
      moreover from 1.prems have n2: n div 2 ≤ length kvs by simp
      moreover from True n2 have P (n div 2) kvs by(rule IH)
      moreover from length-rbtreeify-f[OF n2] ge0 1.prems obtain t k v kvs'
        where kvs': rbtreeify-f (n div 2) kvs = (t, (k, v) # kvs')
        by(cases snd (rbtreeify-f (n div 2) kvs))
          (auto simp add: snd-def split: prod.split-asm)
      moreover from 1.prems length-rbtreeify-f[OF n2] ge0
      have n2': n div 2 ≤ Suc (length kvs') by(simp add: kvs')
      moreover from True kvs'[symmetric] refl refl n2'
      have Q (n div 2) kvs' by(rule IH)
      moreover note feven[unfolded feven-def]

      ultimately have P (2 * (n div 2)) kvs by −
        thus ?thesis using True by (metis minus-mod-eq-div-mult [symmetric]
minus-nat.diff-0 mult.commute)
    next
      case False note ge0
      moreover from 1.prems have n2: n div 2 ≤ length kvs by simp
      moreover from False n2 have P (n div 2) kvs by(rule IH)
      moreover from length-rbtreeify-f[OF n2] ge0 1.prems obtain t k v kvs'
        where kvs': rbtreeify-f (n div 2) kvs = (t, (k, v) # kvs')
        by(cases snd (rbtreeify-f (n div 2) kvs))
          (auto simp add: snd-def split: prod.split-asm)
      moreover from 1.prems length-rbtreeify-f[OF n2] ge0 False
      have n2': n div 2 ≤ length kvs' by(simp add: kvs') arith
       moreover from False kvs'[symmetric] refl refl n2' have P (n div 2) kvs'
by(rule IH)
      moreover note fodd[unfolded fodd-def]
      ultimately have P (Suc (2 * (n div 2))) kvs by −
      thus ?thesis using False
     by simp (metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend
minus-mod-eq-mult-div [symmetric])
    qed
  qed
 next
  case (2 n kvs)
  show ?case
  proof(cases n ≤ 1)
    case True thus ?thesis using 2.prems
      by(cases n kvs rule: nat.exhaust[case-product list.exhaust])
        (auto simp add: g0[unfolded g0-def] g1[unfolded g1-def])
  next
    case False
    hence ns: ¬ (n = 0 ∨ n = 1) by simp
    hence ge0: n div 2 > 0 by simp
    note IH = 2.IH[OF ns]
    show ?thesis
    proof(cases n mod 2 = 0)
      case True note ge0
```

```
      moreover from 2.prems have n2: n div 2 ≤ Suc (length kvs) by simp
      moreover from True n2 have Q (n div 2) kvs by(rule IH)
     moreover from length-rbtreeify-g[OF ge0 n2] ge0 2.prems obtain t k v kvs'

       where kvs': rbtreeify-g (n div 2) kvs = (t, (k, v) # kvs')
       by(cases snd (rbtreeify-g (n div 2) kvs))
        (auto simp add: snd-def split: prod.split-asm)
      moreover from 2.prems length-rbtreeify-g[OF ge0 n2] ge0
      have n2': n div 2 ≤ Suc (length kvs') by(simp add: kvs')
      moreover from True kvs'[symmetric] refl refl n2'
      have Q (n div 2) kvs' by(rule IH)
      moreover note geven[unfolded geven-def]
      ultimately have Q (2 * (n div 2)) kvs by −
      thus ?thesis using True
     by(metis minus-mod-eq-div-mult [symmetric] minus-nat.diff-0 mult.commute)
    next
      case False note ge0
      moreover from 2.prems have n2: n div 2 ≤ length kvs by simp
      moreover from False n2 have P (n div 2) kvs by(rule IH)
       moreover from length-rbtreeify-f[OF n2] ge0 2.prems False obtain t k v
kvs'
       where kvs': rbtreeify-f (n div 2) kvs = (t, (k, v) # kvs')
       by(cases snd (rbtreeify-f (n div 2) kvs))
        (auto simp add: snd-def split: prod.split-asm, arith)
      moreover from 2.prems length-rbtreeify-f[OF n2] ge0 False
      have n2': n div 2 ≤ Suc (length kvs') by(simp add: kvs') arith
      moreover from False kvs'[symmetric] refl refl n2'
      have Q (n div 2) kvs' by(rule IH)
      moreover note godd[unfolded godd-def]
      ultimately have Q (Suc (2 * (n div 2))) kvs by −
      thus ?thesis using False
     by simp (metis One-nat-def Suc-eq-plus1-left le-add-diff-inverse mod-less-eq-dividend
minus-mod-eq-mult-div [symmetric])
   qed
  qed
 qed
qed

lemma inv1-rbtreeify-f: n ≤ length kvs
 ⟹ inv1 (fst (rbtreeify-f n kvs))
 and inv1-rbtreeify-g: n ≤ Suc (length kvs)
 ⟹ inv1 (fst (rbtreeify-g n kvs))
by(induct n kvs and n kvs rule: rbtreeify-induct) simp-all

fun plog2 :: nat ⇒ nat
where plog2 n = (if n ≤ 1 then 0 else plog2 (n div 2) + 1)

declare plog2.simps [simp del]
```

lemma *plog2-simps* [*simp*]:
plog2 0 = 0 plog2 (Suc 0) = 0
*0 < n ⟹ plog2 (2 * n) = 1 + plog2 n*
*0 < n ⟹ plog2 (Suc (2 * n)) = 1 + plog2 n*
by(*subst plog2.simps*, *simp add*: *Suc-double-half*)+

lemma *bheight-rbtreeify-f*: *n ≤ length kvs*
⟹ bheight (fst (rbtreeify-f n kvs)) = plog2 n
and *bheight-rbtreeify-g*: *n ≤ Suc (length kvs)*
⟹ bheight (fst (rbtreeify-g n kvs)) = plog2 n
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*) *simp-all*

lemma *bheight-rbtreeify-f-eq-plog2I*:
⟦ *rbtreeify-f n kvs = (t, kvs')*; *n ≤ length kvs* ⟧
⟹ bheight t = plog2 n
using *bheight-rbtreeify-f*[*of n kvs*] **by** *simp*

lemma *bheight-rbtreeify-g-eq-plog2I*:
⟦ *rbtreeify-g n kvs = (t, kvs')*; *n ≤ Suc (length kvs)* ⟧
⟹ bheight t = plog2 n
using *bheight-rbtreeify-g*[*of n kvs*] **by** *simp*

hide-const (**open**) *plog2*

lemma *inv2-rbtreeify-f*: *n ≤ length kvs*
⟹ inv2 (fst (rbtreeify-f n kvs))
and *inv2-rbtreeify-g*: *n ≤ Suc (length kvs)*
⟹ inv2 (fst (rbtreeify-g n kvs))
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*)
(*auto simp add*: *bheight-rbtreeify-f bheight-rbtreeify-g*
intro: *bheight-rbtreeify-f-eq-plog2I bheight-rbtreeify-g-eq-plog2I*)

lemma *n ≤ length kvs ⟹ True*
and *color-of-rbtreeify-g*:
⟦ *n ≤ Suc (length kvs)*; *0 < n* ⟧
⟹ color-of (fst (rbtreeify-g n kvs)) = B
by(*induct n kvs* **and** *n kvs rule*: *rbtreeify-induct*) *simp-all*

lemma *entries-rbtreeify-f-append*:
n ≤ length kvs
⟹ entries (fst (rbtreeify-f n kvs)) @ snd (rbtreeify-f n kvs) = kvs
and *entries-rbtreeify-g-append*:
n ≤ Suc (length kvs)
⟹ entries (fst (rbtreeify-g n kvs)) @ snd (rbtreeify-g n kvs) = kvs
by(*induction rule*: *rbtreeify-induct*) *simp-all*

lemma *length-entries-rbtreeify-f*:
n ≤ length kvs ⟹ length (entries (fst (rbtreeify-f n kvs))) = n
and *length-entries-rbtreeify-g*:

$n \leq Suc\ (length\ kvs) \Longrightarrow length\ (entries\ (fst\ (rbtreeify\text{-}g\ n\ kvs))) = n - 1$
by(*induct rule*: *rbtreeify-induct*) *simp-all*

lemma *rbtreeify-f-conv-drop*:
$n \leq length\ kvs \Longrightarrow snd\ (rbtreeify\text{-}f\ n\ kvs) = drop\ n\ kvs$
using *entries-rbtreeify-f-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *rbtreeify-g-conv-drop*:
$n \leq Suc\ (length\ kvs) \Longrightarrow snd\ (rbtreeify\text{-}g\ n\ kvs) = drop\ (n - 1)\ kvs$
using *entries-rbtreeify-g-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *entries-rbtreeify-f* [*simp*]:
$n \leq length\ kvs \Longrightarrow entries\ (fst\ (rbtreeify\text{-}f\ n\ kvs)) = take\ n\ kvs$
using *entries-rbtreeify-f-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-f*)

lemma *entries-rbtreeify-g* [*simp*]:
$n \leq Suc\ (length\ kvs) \Longrightarrow$
$entries\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = take\ (n - 1)\ kvs$
using *entries-rbtreeify-g-append*[*of n kvs*]
by(*simp add*: *append-eq-conv-conj length-entries-rbtreeify-g*)

lemma *keys-rbtreeify-f* [*simp*]: $n \leq length\ kvs$
$\Longrightarrow keys\ (fst\ (rbtreeify\text{-}f\ n\ kvs)) = take\ n\ (map\ fst\ kvs)$
by(*simp add*: *keys-def take-map*)

lemma *keys-rbtreeify-g* [*simp*]: $n \leq Suc\ (length\ kvs)$
$\Longrightarrow keys\ (fst\ (rbtreeify\text{-}g\ n\ kvs)) = take\ (n - 1)\ (map\ fst\ kvs)$
by(*simp add*: *keys-def take-map*)

lemma *rbtreeify-fD*:
$[\![\ rbtreeify\text{-}f\ n\ kvs = (t,\ kvs');\ n \leq length\ kvs\ ]\!]$
$\Longrightarrow entries\ t = take\ n\ kvs \wedge kvs' = drop\ n\ kvs$
using *rbtreeify-f-conv-drop*[*of n kvs*] *entries-rbtreeify-f*[*of n kvs*] **by** *simp*

lemma *rbtreeify-gD*:
$[\![\ rbtreeify\text{-}g\ n\ kvs = (t,\ kvs');\ n \leq Suc\ (length\ kvs)\ ]\!]$
$\Longrightarrow entries\ t = take\ (n - 1)\ kvs \wedge kvs' = drop\ (n - 1)\ kvs$
using *rbtreeify-g-conv-drop*[*of n kvs*] *entries-rbtreeify-g*[*of n kvs*] **by** *simp*

lemma *entries-rbtreeify* [*simp*]: $entries\ (rbtreeify\ kvs) = kvs$
by(*simp add*: *rbtreeify-def entries-rbtreeify-g*)

context *linorder* **begin**

lemma *rbt-sorted-rbtreeify-f*:
$[\![\ n \leq length\ kvs;\ sorted\ (map\ fst\ kvs);\ distinct\ (map\ fst\ kvs)\ ]\!]$

$\Longrightarrow$ *rbt-sorted* (*fst* (*rbtreeify-f n kvs*))
and *rbt-sorted-rbtreeify-g*:
⟦ $n \leq$ *Suc* (*length kvs*); *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
$\Longrightarrow$ *rbt-sorted* (*fst* (*rbtreeify-g n kvs*))
proof(*induction n kvs* **and** *n kvs rule*: *rbtreeify-induct*)
case (*f-even n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹$n \leq$ *length kvs*›]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* $\leq$ *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *k* $\leq$ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* $\neq$ *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *x* $\neq$ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* < *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *k* < *x*)
by *fastforce*
hence *fst* (*rbtreeify-f n kvs*) |« *k k* «| *fst* (*rbtreeify-g n kvs'*)
using ‹$n \leq$ *Suc* (*length kvs'*)› ‹$n \leq$ *length kvs*› *set-take-subset*[*of n* − *1 kvs'*]
by(*auto simp add*: *ord.rbt-greater-prop ord.rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule f-even.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule f-even.IH*)
ultimately show *?case*
using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
next
case (*f-odd n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹$n \leq$ *length kvs*›]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* $\leq$ *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *k* $\leq$ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* $\neq$ *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *x* $\neq$ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have ($\forall$(*x*, *y*) $\in$ *set* (*take n kvs*). *x* < *k*) $\wedge$ ($\forall$(*x*, *y*) $\in$ *set kvs'*. *k* < *x*)
by *fastforce*
hence *fst* (*rbtreeify-f n kvs*) |« *k k* «| *fst* (*rbtreeify-f n kvs'*)
using ‹$n \leq$ *length kvs'*› ‹$n \leq$ *length kvs*› *set-take-subset*[*of n kvs'*]
by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›

have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule f-odd.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append*)+
hence *rbt-sorted* (*fst* (*rbtreeify-f n kvs'*)) **by**(*rule f-odd.IH*)
ultimately show *?case*
using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
next
case (*g-even n kvs t k v kvs'*)
from *rbtreeify-gD*[*OF* ‹*rbtreeify-g n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *Suc* (*length kvs*)›]
have *t*: *entries t* = *take* (*n* − *1*) *kvs*
and *kvs'*: *drop* (*n* − *1*) *kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take* (*n* − *1*) *kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* ≤ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* ≠ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have (∀(*x*, *y*) ∈ *set* (*take* (*n* − *1*) *kvs*). *x* < *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* < *x*)
by *fastforce*
hence *fst* (*rbtreeify-g n kvs*) |« *k* *k* «| *fst* (*rbtreeify-g n kvs'*)
using ‹*n* ≤ *Suc* (*length kvs'*)› ‹*n* ≤ *Suc* (*length kvs*)› *set-take-subset*[*of n* − *1 kvs'*]
by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
have *rbt-sorted* (*fst* (*rbtreeify-g n kvs*)) **by**(*rule g-even.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule g-even.IH*)
ultimately show *?case* **using** ‹*0* < *n*› ‹*rbtreeify-g n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
next
case (*g-odd n kvs t k v kvs'*)
from *rbtreeify-fD*[*OF* ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› ‹*n* ≤ *length kvs*›]
have *entries t* = *take n kvs*
and *kvs'*: *drop n kvs* = (*k*, *v*) # *kvs'* **by** *simp-all*
hence *unfold*: *kvs* = *take n kvs* @ (*k*, *v*) # *kvs'* **by**(*metis append-take-drop-id*)
from ‹*sorted* (*map fst kvs*)› *kvs'*
have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* ≤ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* ≤ *x*)
by(*subst* (*asm*) *unfold*)(*auto simp add*: *sorted-append*)
moreover from ‹*distinct* (*map fst kvs*)› *kvs'*
have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* ≠ *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *x* ≠ *k*)
by(*subst* (*asm*) *unfold*)(*auto intro*: *rev-image-eqI*)
ultimately have (∀(*x*, *y*) ∈ *set* (*take n kvs*). *x* < *k*) ∧ (∀(*x*, *y*) ∈ *set kvs'*. *k* < *x*)

by *fastforce*
hence *fst* (*rbtreeify-f n kvs*) |« *k k* «| *fst* (*rbtreeify-g n kvs'*)
using ‹*n* ≤ *Suc* (*length kvs'*)› ‹*n* ≤ *length kvs*› *set-take-subset*[*of n − 1 kvs'*]
by(*auto simp add*: *rbt-greater-prop rbt-less-prop take-map split-def*)
moreover from ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
have *rbt-sorted* (*fst* (*rbtreeify-f n kvs*)) **by**(*rule g-odd.IH*)
moreover have *sorted* (*map fst kvs'*) *distinct* (*map fst kvs'*)
using ‹*sorted* (*map fst kvs*)› ‹*distinct* (*map fst kvs*)›
by(*subst* (*asm*) (*1 2*) *unfold*, *simp add*: *sorted-append*)+
hence *rbt-sorted* (*fst* (*rbtreeify-g n kvs'*)) **by**(*rule g-odd.IH*)
ultimately show *?case*
using ‹*0* < *n*› ‹*rbtreeify-f n kvs* = (*t*, (*k*, *v*) # *kvs'*)› **by** *simp*
qed *simp-all*

lemma *rbt-sorted-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹ *rbt-sorted* (*rbtreeify kvs*)
by(*simp add*: *rbtreeify-def rbt-sorted-rbtreeify-g*)

lemma *is-rbt-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧
⟹ *is-rbt* (*rbtreeify kvs*)
by(*simp add*: *is-rbt-def rbtreeify-def inv1-rbtreeify-g inv2-rbtreeify-g rbt-sorted-rbtreeify-g color-of-rbtreeify-g*)

lemma *rbt-lookup-rbtreeify*:
⟦ *sorted* (*map fst kvs*); *distinct* (*map fst kvs*) ⟧ ⟹
rbt-lookup (*rbtreeify kvs*) = *map-of kvs*
by(*simp add*: *map-of-entries*[*symmetric*] *rbt-sorted-rbtreeify*)

end

Functions to compare the height of two rbt trees, taken from Andrew W. Appel, Efficient Verified Red-Black Trees (September 2011)

fun *skip-red* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-red (*Branch color.R l k v r*) = *l*
| *skip-red t* = *t*

definition *skip-black* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
where
skip-black t = (*let t'* = *skip-red t in case t' of Branch color.B l k v r* ⇒ *l* | - ⇒ *t'*)

datatype *compare* = *LT* | *GT* | *EQ*

partial-function (*tailrec*) *compare-height* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ *compare*
where
compare-height sx s t tx =

```
(case (skip-red sx, skip-red s, skip-red t, skip-red tx) of
   (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
     compare-height (skip-black sx') s' t' (skip-black tx')
 | (-, rbt.Empty, -, Branch - - - - -) ⇒ LT
 | (Branch - - - - -, -, rbt.Empty, -) ⇒ GT
 | (Branch - sx' - - -, Branch - s' - - -, Branch - t' - - -, rbt.Empty) ⇒
     compare-height (skip-black sx') s' t' rbt.Empty
 | (rbt.Empty, Branch - s' - - -, Branch - t' - - -, Branch - tx' - - -) ⇒
     compare-height rbt.Empty s' t' (skip-black tx')
 | - ⇒ EQ)
```

declare *compare-height.simps* [*code*]

hide-type (**open**) *compare*
hide-const (**open**)
compare-height skip-black skip-red LT GT EQ case-compare rec-compare
Abs-compare Rep-compare
hide-fact (**open**)
Abs-compare-cases Abs-compare-induct Abs-compare-inject Abs-compare-inverse
Rep-compare Rep-compare-cases Rep-compare-induct Rep-compare-inject Rep-compare-inverse
compare.simps compare.exhaust compare.induct compare.rec compare.simps
compare.size compare.case-cong compare.case-cong-weak compare.case
compare.nchotomy compare.split compare.split-asm compare.eq.refl compare.eq.simps
equal-compare-def
skip-red.simps skip-red.cases skip-red.induct
skip-black-def
compare-height.simps

130.10 union and intersection of sorted associative lists

context *ord* **begin**

function *sunion-with* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* × '*b*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*a* × '*b*) *list*
where
sunion-with f ((k, v) # as) ((k', v') # bs) =
(if k > k' then (k', v') # sunion-with f ((k, v) # as) bs
else if k < k' then (k, v) # sunion-with f as ((k', v') # bs)
else (k, f k v v') # sunion-with f as bs)
| *sunion-with f [] bs = bs*
| *sunion-with f as [] = as*
by *pat-completeness auto*
termination by *lexicographic-order*

function *sinter-with* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a* × '*b*) *list* ⇒ ('*a* × '*b*) *list* ⇒ ('*a* × '*b*) *list*
where
sinter-with f ((k, v) # as) ((k', v') # bs) =
(if k > k' then sinter-with f ((k, v) # as) bs

else if $k < k'$ *then sinter-with f as* $((k', v') \# bs)$
else $(k, f\ k\ v\ v') \#$ *sinter-with f as bs*)
| *sinter-with f* [] - = []
| *sinter-with f* - [] = []
by *pat-completeness auto*
termination by *lexicographic-order*

end

declare *ord.sunion-with.simps* [*code*] *ord.sinter-with.simps*[*code*]

context *linorder* **begin**

lemma *set-fst-sunion-with*:
set (*map fst* (*sunion-with f xs ys*)) = *set* (*map fst xs*) ∪ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sunion-with.induct*) *auto*

lemma *sorted-sunion-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sunion-with f xs ys*))
by(*induct f xs ys rule*: *sunion-with.induct*)
(*auto simp add*: *set-fst-sunion-with simp del*: *set-map*)

lemma *distinct-sunion-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*); *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*sunion-with f xs ys*))
proof(*induct f xs ys rule*: *sunion-with.induct*)
case (*1 f k v xs k' v' ys*)
have ⟦ ¬ $k < k'$; ¬ $k' < k$ ⟧ ⟹ $k = k'$ **by** *simp*
thus *?case* **using** *1*
by(*auto simp add*: *set-fst-sunion-with simp del*: *set-map*)
qed *simp-all*

lemma *map-of-sunion-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *map-of* (*sunion-with f xs ys*) *k* =
(*case map-of xs k of None* ⇒ *map-of ys k*
| *Some v* ⇒ *case map-of ys k of None* ⇒ *Some v*
| *Some w* ⇒ *Some* (*f k v w*))
by(*induct f xs ys rule*: *sunion-with.induct*)(*auto split*: *option.split dest*: *map-of-SomeD bspec*)

lemma *set-fst-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *set* (*map fst* (*sinter-with f xs ys*)) = *set* (*map fst xs*) ∩ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp del*: *set-map*)

lemma *set-fst-sinter-with-subset1*:

set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst xs*)
by(*induct f xs ys rule*: *sinter-with.induct*) *auto*

lemma *set-fst-sinter-with-subset2*:
set (*map fst* (*sinter-with f xs ys*)) ⊆ *set* (*map fst ys*)
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp del*: *set-map*)

lemma *sorted-sinter-with* [*simp*]:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *sorted* (*map fst* (*sinter-with f xs ys*))
by(*induct f xs ys rule*: *sinter-with.induct*)(*auto simp del*: *set-map*)

lemma *distinct-sinter-with* [*simp*]:
⟦ *distinct* (*map fst xs*); *distinct* (*map fst ys*) ⟧
⟹ *distinct* (*map fst* (*sinter-with f xs ys*))
proof(*induct f xs ys rule*: *sinter-with.induct*)
case (*1 f k v as k' v' bs*)
have ⟦ ¬ *k* < *k'*; ¬ *k'* < *k* ⟧ ⟹ *k* = *k'* **by** *simp*
thus *?case* **using** *1 set-fst-sinter-with-subset1*[*of f as bs*]
set-fst-sinter-with-subset2[*of f as bs*]
by(*auto simp del*: *set-map*)
qed *simp-all*

lemma *map-of-sinter-with*:
⟦ *sorted* (*map fst xs*); *sorted* (*map fst ys*) ⟧
⟹ *map-of* (*sinter-with f xs ys*) *k* =
(*case map-of xs k of None* ⇒ *None* | *Some v* ⇒ *map-option* (*f k v*) (*map-of ys k*))
apply(*induct f xs ys rule*: *sinter-with.induct*)
apply(*auto simp add*: *map-option-case split*: *option.splits dest*: *map-of-SomeD bspec*)
done

end

lemma *distinct-map-of-rev*: *distinct* (*map fst xs*) ⟹ *map-of* (*rev xs*) = *map-of xs*
by(*induct xs*)(*auto 4 3 simp add*: *map-add-def intro*!: *ext split*: *option.split intro*: *rev-image-eqI*)

lemma *map-filter-map-option-const*:
List.map-filter (λ*x*. *map-option* (λ*y*. *f x*) (*g* (*f x*))) *xs* = *filter* (λ*x*. *g x* ≠ *None*) (*map f xs*)
by(*auto simp add*: *map-filter-def filter-map o-def*)

lemma *set-map-filter*: *set* (*List.map-filter P xs*) = *the* ‘ (*P* ‘ *set xs* − {*None*})
by(*auto simp add*: *List.map-filter-def intro*: *rev-image-eqI*)

definition *is-rbt-empty* :: ('*a*, '*b*) *rbt* ⇒ *bool* **where**

is-rbt-empty t $\longleftrightarrow$ (*case t of RBT-Impl.Empty* $\Rightarrow$ *True* | - $\Rightarrow$ *False*)

lemma *is-rbt-empty-prop*[*simp*]: *is-rbt-empty t* $\longleftrightarrow$ *t = RBT-Impl.Empty*
by (*auto simp*: *is-rbt-empty-def split*: *RBT-Impl.rbt.splits*)

definition *small-rbt* :: ('*a*, '*b*) *rbt* $\Rightarrow$ *bool* **where**
small-rbt t $\longleftrightarrow$ *bheight t* < *4*

definition *flip-rbt* :: ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ *bool* **where**
flip-rbt t1 t2 $\longleftrightarrow$ *bheight t2* < *bheight t1*

abbreviation (*input*) *MR* **where** *MR l a b r* $\equiv$ *Branch RBT-Impl.R l a b r*
abbreviation (*input*) *MB* **where** *MB l a b r* $\equiv$ *Branch RBT-Impl.B l a b r*

fun *rbt-baliL* :: ('*a*, '*b*) *rbt* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
rbt-baliL (*MR* (*MR t1 a b t2*) *a' b' t3*) *a'' b'' t4* = *MR* (*MB t1 a b t2*) *a' b'* (*MB t3 a'' b'' t4*)
| *rbt-baliL* (*MR t1 a b* (*MR t2 a' b' t3*)) *a'' b'' t4* = *MR* (*MB t1 a b t2*) *a' b'* (*MB t3 a'' b'' t4*)
| *rbt-baliL t1 a b t2* = *MB t1 a b t2*

fun *rbt-baliR* :: ('*a*, '*b*) *rbt* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
rbt-baliR t1 a b (*MR t2 a' b'* (*MR t3 a'' b'' t4*)) = *MR* (*MB t1 a b t2*) *a' b'* (*MB t3 a'' b'' t4*)
| *rbt-baliR t1 a b* (*MR* (*MR t2 a' b' t3*) *a'' b'' t4*) = *MR* (*MB t1 a b t2*) *a' b'* (*MB t3 a'' b'' t4*)
| *rbt-baliR t1 a b t2* = *MB t1 a b t2*

fun *rbt-baldL* :: ('*a*, '*b*) *rbt* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
rbt-baldL (*MR t1 a b t2*) *a' b' t3* = *MR* (*MB t1 a b t2*) *a' b' t3*
| *rbt-baldL t1 a b* (*MB t2 a' b' t3*) = *rbt-baliR t1 a b* (*MR t2 a' b' t3*)
| *rbt-baldL t1 a b* (*MR* (*MB t2 a' b' t3*) *a'' b'' t4*) =
MR (*MB t1 a b t2*) *a' b'* (*rbt-baliR t3 a'' b''* (*paint RBT-Impl.R t4*))
| *rbt-baldL t1 a b t2* = *MR t1 a b t2*

fun *rbt-baldR* :: ('*a*, '*b*) *rbt* $\Rightarrow$ '*a* $\Rightarrow$ '*b* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
rbt-baldR t1 a b (*MR t2 a' b' t3*) = *MR t1 a b* (*MB t2 a' b' t3*)
| *rbt-baldR* (*MB t1 a b t2*) *a' b' t3* = *rbt-baliL* (*MR t1 a b t2*) *a' b' t3*
| *rbt-baldR* (*MR t1 a b* (*MB t2 a' b' t3*)) *a'' b'' t4* =
MR (*rbt-baliL* (*paint RBT-Impl.R t1*) *a b t2*) *a' b'* (*MB t3 a'' b'' t4*)
| *rbt-baldR t1 a b t2* = *MR t1 a b t2*

fun *rbt-app* :: ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* $\Rightarrow$ ('*a*, '*b*) *rbt* **where**
rbt-app RBT-Impl.Empty t = *t*
| *rbt-app t RBT-Impl.Empty* = *t*
| *rbt-app* (*MR t1 a b t2*) (*MR t3 a'' b'' t4*) = (*case rbt-app t2 t3 of*
MR u2 a' b' u3 $\Rightarrow$ (*MR* (*MR t1 a b u2*) *a' b'* (*MR u3 a'' b'' t4*))
| *t23* $\Rightarrow$ *MR t1 a b* (*MR t23 a'' b'' t4*))
| *rbt-app* (*MB t1 a b t2*) (*MB t3 a'' b'' t4*) = (*case rbt-app t2 t3 of*

```
    MR u2 a' b' u3 ⇒ MR (MB t1 a b u2) a' b' (MB u3 a'' b'' t4)
  | t23 ⇒ rbt-baldL t1 a b (MB t23 a'' b'' t4))
| rbt-app t1 (MR t2 a b t3) = MR (rbt-app t1 t2) a b t3
| rbt-app (MR t1 a b t2) t3 = MR t1 a b (rbt-app t2 t3)

fun rbt-joinL :: ('a, 'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt where
  rbt-joinL l a b r = (if bheight l ≥ bheight r then MR l a b r
    else case r of MB l' a' b' r' ⇒ rbt-baliL (rbt-joinL l a b l') a' b' r'
    | MR l' a' b' r' ⇒ MR (rbt-joinL l a b l') a' b' r')

declare rbt-joinL.simps[simp del]

fun rbt-joinR :: ('a, 'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt where
  rbt-joinR l a b r = (if bheight l ≤ bheight r then MR l a b r
    else case l of MB l' a' b' r' ⇒ rbt-baliR l' a' b' (rbt-joinR r' a b r)
    | MR l' a' b' r' ⇒ MR l' a' b' (rbt-joinR r' a b r))

declare rbt-joinR.simps[simp del]

definition rbt-join :: ('a, 'b) rbt ⇒ 'a ⇒ 'b ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt where
  rbt-join l a b r =
    (let bhl = bheight l; bhr = bheight r
    in if bhl > bhr
    then paint RBT-Impl.B (rbt-joinR l a b r)
    else if bhl < bhr
    then paint RBT-Impl.B (rbt-joinL l a b r)
    else MB l a b r)

lemma size-paint[simp]: size (paint c t) = size t
  by (cases t) auto

lemma size-baliL[simp]: size (rbt-baliL t1 a b t2) = Suc (size t1 + size t2)
  by (induction t1 a b t2 rule: rbt-baliL.induct) auto

lemma size-baliR[simp]: size (rbt-baliR t1 a b t2) = Suc (size t1 + size t2)
  by (induction t1 a b t2 rule: rbt-baliR.induct) auto

lemma size-baldL[simp]: size (rbt-baldL t1 a b t2) = Suc (size t1 + size t2)
  by (induction t1 a b t2 rule: rbt-baldL.induct) auto

lemma size-baldR[simp]: size (rbt-baldR t1 a b t2) = Suc (size t1 + size t2)
  by (induction t1 a b t2 rule: rbt-baldR.induct) auto

lemma size-rbt-app[simp]: size (rbt-app t1 t2) = size t1 + size t2
  by (induction t1 t2 rule: rbt-app.induct)
    (auto split: RBT-Impl.rbt.splits RBT-Impl.color.splits)

lemma size-rbt-joinL[simp]: size (rbt-joinL t1 a b t2) = Suc (size t1 + size t2)
  by (induction t1 a b t2 rule: rbt-joinL.induct)
```

(*auto simp*: *rbt-joinL.simps split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *size-rbt-joinR*[*simp*]: *size* (*rbt-joinR t1 a b t2*) = *Suc* (*size t1* + *size t2*)
by (*induction t1 a b t2 rule*: *rbt-joinR.induct*)
(*auto simp*: *rbt-joinR.simps split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *size-rbt-join*[*simp*]: *size* (*rbt-join t1 a b t2*) = *Suc* (*size t1* + *size t2*)
by (*auto simp*: *rbt-join-def Let-def*)

definition *inv-12 t* ⟷ *inv1 t* ∧ *inv2 t*

lemma *rbt-Node*: *inv-12* (*RBT-Impl.Branch c l a b r*) ⟹ *inv-12 l* ∧ *inv-12 r*
by (*auto simp*: *inv-12-def*)

lemma *paint2*: *paint c2* (*paint c1 t*) = *paint c2 t*
by (*cases t*) *auto*

lemma *inv1-rbt-baliL*: *inv1l l* ⟹ *inv1 r* ⟹ *inv1* (*rbt-baliL l a b r*)
by (*induct l a b r rule*: *rbt-baliL.induct*) *auto*

lemma *inv1-rbt-baliR*: *inv1 l* ⟹ *inv1l r* ⟹ *inv1* (*rbt-baliR l a b r*)
by (*induct l a b r rule*: *rbt-baliR.induct*) *auto*

lemma *rbt-bheight-rbt-baliL*: *bheight l* = *bheight r* ⟹ *bheight* (*rbt-baliL l a b r*) = *Suc* (*bheight l*)
by (*induct l a b r rule*: *rbt-baliL.induct*) *auto*

lemma *rbt-bheight-rbt-baliR*: *bheight l* = *bheight r* ⟹ *bheight* (*rbt-baliR l a b r*) = *Suc* (*bheight l*)
by (*induct l a b r rule*: *rbt-baliR.induct*) *auto*

lemma *inv2-rbt-baliL*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* = *bheight r* ⟹ *inv2* (*rbt-baliL l a b r*)
by (*induct l a b r rule*: *rbt-baliL.induct*) *auto*

lemma *inv2-rbt-baliR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* = *bheight r* ⟹ *inv2* (*rbt-baliR l a b r*)
by (*induct l a b r rule*: *rbt-baliR.induct*) *auto*

lemma *inv-rbt-baliR*: *inv2 l* ⟹ *inv2 r* ⟹ *inv1 l* ⟹ *inv1l r* ⟹ *bheight l* = *bheight r* ⟹
inv1 (*rbt-baliR l a b r*) ∧ *inv2* (*rbt-baliR l a b r*) ∧ *bheight* (*rbt-baliR l a b r*) = *Suc* (*bheight l*)
by (*induct l a b r rule*: *rbt-baliR.induct*) *auto*

lemma *inv-rbt-baliL*: *inv2 l* ⟹ *inv2 r* ⟹ *inv1l l* ⟹ *inv1 r* ⟹ *bheight l* = *bheight r* ⟹
inv1 (*rbt-baliL l a b r*) ∧ *inv2* (*rbt-baliL l a b r*) ∧ *bheight* (*rbt-baliL l a b r*) = *Suc* (*bheight l*)

by (*induct l a b r rule*: *rbt-baliL.induct*) *auto*

lemma *inv2-rbt-baldL-inv1*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l* + *1* = *bheight r* $\Longrightarrow$ *inv1 r* $\Longrightarrow$
inv2 (*rbt-baldL l a b r*) $\wedge$ *bheight* (*rbt-baldL l a b r*) = *bheight r*
by (*induct l a b r rule*: *rbt-baldL.induct*) (*auto simp*: *inv2-rbt-baliR rbt-bheight-rbt-baliR*)

lemma *inv2-rbt-baldL-B*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l* + *1* = *bheight r* $\Longrightarrow$ *color-of r* = *RBT-Impl.B* $\Longrightarrow$
inv2 (*rbt-baldL l a b r*) $\wedge$ *bheight* (*rbt-baldL l a b r*) = *bheight r*
by (*induct l a b r rule*: *rbt-baldL.induct*) (*auto simp add*: *inv2-rbt-baliR rbt-bheight-rbt-baliR*)

lemma *inv1-rbt-baldL*: *inv1l l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$ *color-of r* = *RBT-Impl.B* $\Longrightarrow$ *inv1* (*rbt-baldL l a b r*)
by (*induct l a b r rule*: *rbt-baldL.induct*) (*simp-all add*: *inv1-rbt-baliR*)

lemma *inv1lI*: *inv1 t* $\Longrightarrow$ *inv1l t*
by (*cases t*) *auto*

lemma *neq-Black*[*simp*]: (*c* $\neq$ *RBT-Impl.B*) = (*c* = *RBT-Impl.R*)
by (*cases c*) *auto*

lemma *inv1l-rbt-baldL*: *inv1l l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$ *inv1l* (*rbt-baldL l a b r*)
by (*induct l a b r rule*: *rbt-baldL.induct*) (*auto simp*: *inv1-rbt-baliR paint2*)

lemma *inv2-rbt-baldR-inv1*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l* = *bheight r* + *1* $\Longrightarrow$ *inv1 l* $\Longrightarrow$
inv2 (*rbt-baldR l a b r*) $\wedge$ *bheight* (*rbt-baldR l a b r*) = *bheight l*
by (*induct l a b r rule*: *rbt-baldR.induct*) (*auto simp*: *inv2-rbt-baliL rbt-bheight-rbt-baliL*)

lemma *inv1-rbt-baldR*: *inv1 l* $\Longrightarrow$ *inv1l r* $\Longrightarrow$ *color-of l* = *RBT-Impl.B* $\Longrightarrow$ *inv1* (*rbt-baldR l a b r*)
by (*induct l a b r rule*: *rbt-baldR.induct*) (*simp-all add*: *inv1-rbt-baliL*)

lemma *inv1l-rbt-baldR*: *inv1 l* $\Longrightarrow$ *inv1l r* $\Longrightarrow$*inv1l* (*rbt-baldR l a b r*)
by (*induct l a b r rule*: *rbt-baldR.induct*) (*auto simp*: *inv1-rbt-baliL paint2*)

lemma *inv2-rbt-app*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l* = *bheight r* $\Longrightarrow$
inv2 (*rbt-app l r*) $\wedge$ *bheight* (*rbt-app l r*) = *bheight l*
by (*induct l r rule*: *rbt-app.induct*)
(*auto simp*: *inv2-rbt-baldL-B split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv1-rbt-app*: *inv1 l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$ (*color-of l* = *RBT-Impl.B* $\wedge$
color-of r = *RBT-Impl.B* $\longrightarrow$ *inv1* (*rbt-app l r*)) $\wedge$ *inv1l* (*rbt-app l r*)
by (*induct l r rule*: *rbt-app.induct*)
(*auto simp*: *inv1-rbt-baldL split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv-rbt-baldL*: *inv2 l* $\Longrightarrow$ *inv2 r* $\Longrightarrow$ *bheight l* + *1* = *bheight r* $\Longrightarrow$ *inv1l l* $\Longrightarrow$ *inv1 r* $\Longrightarrow$

inv2 (rbt-baldL l a b r) ∧ bheight (rbt-baldL l a b r) = bheight r ∧
inv1l (rbt-baldL l a b r) ∧ (color-of r = RBT-Impl.B ⟶ inv1 (rbt-baldL l a b r))
by (*induct l a b r rule*: *rbt-baldL.induct*) (*auto simp*: *inv-rbt-baliR rbt-bheight-rbt-baliR paint2*)

lemma *inv-rbt-baldR*: *inv2 l ⟹ inv2 r ⟹ bheight l = bheight r + 1 ⟹ inv1 l ⟹ inv1l r ⟹*
inv2 (rbt-baldR l a b r) ∧ bheight (rbt-baldR l a b r) = bheight l ∧
inv1l (rbt-baldR l a b r) ∧ (color-of l = RBT-Impl.B ⟶ inv1 (rbt-baldR l a b r))
by (*induct l a b r rule*: *rbt-baldR.induct*) (*auto simp*: *inv-rbt-baliL rbt-bheight-rbt-baliL paint2*)

lemma *inv-rbt-app*: *inv2 l ⟹ inv2 r ⟹ bheight l = bheight r ⟹ inv1 l ⟹ inv1 r ⟹*
inv2 (rbt-app l r) ∧ bheight (rbt-app l r) = bheight l ∧
inv1l (rbt-app l r) ∧ (color-of l = RBT-Impl.B ∧ color-of r = RBT-Impl.B ⟶ inv1 (rbt-app l r))
by (*induct l r rule*: *rbt-app.induct*)
(*auto simp*: *inv2-rbt-baldL-B inv-rbt-baldL split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *inv1l-rbt-joinL*: *inv1 l ⟹ inv1 r ⟹ bheight l ≤ bheight r ⟹*
inv1l (rbt-joinL l a b r) ∧
(bheight l ≠ bheight r ∧ color-of r = RBT-Impl.B ⟶ inv1 (rbt-joinL l a b r))
proof (*induct l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
then show *?case*
by (*auto simp*: *inv1-rbt-baliL rbt-joinL.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *inv1l-rbt-joinR*: *inv1 l ⟹ inv2 l ⟹ inv1 r ⟹ inv2 r ⟹ bheight l ≥ bheight r ⟹*
inv1l (rbt-joinR l a b r) ∧
(bheight l ≠ bheight r ∧ color-of l = RBT-Impl.B ⟶ inv1 (rbt-joinR l a b r))
proof (*induct l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
then show *?case*
by (*fastforce simp*: *inv1-rbt-baliR rbt-joinR.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *bheight-rbt-joinL*: *inv2 l ⟹ inv2 r ⟹ bheight l ≤ bheight r ⟹*
bheight (rbt-joinL l a b r) = bheight r
proof (*induct l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
then show *?case*
by (*auto simp*: *rbt-bheight-rbt-baliL rbt-joinL.simps*[*of l a b r*]

split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *inv2-rbt-joinL*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≤ *bheight r* ⟹ *inv2* (*rbt-joinL l a b r*)
proof (*induct l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
then show *?case*
by (*auto simp*: *inv2-rbt-baliL bheight-rbt-joinL rbt-joinL.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *bheight-rbt-joinR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≥ *bheight r* ⟹
bheight (*rbt-joinR l a b r*) = *bheight l*
proof (*induct l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
then show *?case*
by (*fastforce simp*: *rbt-bheight-rbt-baliR rbt-joinR.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *inv2-rbt-joinR*: *inv2 l* ⟹ *inv2 r* ⟹ *bheight l* ≥ *bheight r* ⟹ *inv2* (*rbt-joinR l a b r*)
proof (*induct l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
then show *?case*
by (*fastforce simp*: *inv2-rbt-baliR bheight-rbt-joinR rbt-joinR.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *keys-paint*[*simp*]: *RBT-Impl.keys* (*paint c t*) = *RBT-Impl.keys t*
by (*cases t*) *auto*

lemma *keys-rbt-baliL*: *RBT-Impl.keys* (*rbt-baliL l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliL.cases*) *auto*

lemma *keys-rbt-baliR*: *RBT-Impl.keys* (*rbt-baliR l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliR.cases*) *auto*

lemma *keys-rbt-baldL*: *RBT-Impl.keys* (*rbt-baldL l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baldL.cases*) (*auto simp*: *keys-rbt-baliL keys-rbt-baliR*)

lemma *keys-rbt-baldR*: *RBT-Impl.keys* (*rbt-baldR l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baldR.cases*) (*auto simp*: *keys-rbt-baliL keys-rbt-baliR*)

lemma *keys-rbt-app*: *RBT-Impl.keys* (*rbt-app l r*) = *RBT-Impl.keys l* @ *RBT-Impl.keys r*
by (*induction l r rule*: *rbt-app.induct*)
(*auto simp*: *keys-rbt-baldL keys-rbt-baldR split*: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)

lemma *keys-rbt-joinL*: *bheight l* ≤ *bheight r* ⟹
RBT-Impl.keys (*rbt-joinL l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
proof (*induction l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
thus *?case*
by (*auto simp*: *keys-rbt-baliL rbt-joinL.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *keys-rbt-joinR*: *RBT-Impl.keys* (*rbt-joinR l a b r*) = *RBT-Impl.keys l* @ *a* # *RBT-Impl.keys r*
proof (*induction l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
thus *?case*
by (*force simp*: *keys-rbt-baliR rbt-joinR.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *rbt-set-rbt-baliL*: *set* (*RBT-Impl.keys* (*rbt-baliL l a b r*)) =
set (*RBT-Impl.keys l*) ∪ {*a*} ∪ *set* (*RBT-Impl.keys r*)
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliL.cases*) *auto*

lemma *set-rbt-joinL*: *set* (*RBT-Impl.keys* (*rbt-joinL l a b r*)) =
set (*RBT-Impl.keys l*) ∪ {*a*} ∪ *set* (*RBT-Impl.keys r*)
proof (*induction l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
thus *?case*
by (*auto simp*: *rbt-set-rbt-baliL rbt-joinL.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *rbt-set-rbt-baliR*: *set* (*RBT-Impl.keys* (*rbt-baliR l a b r*)) =
set (*RBT-Impl.keys l*) ∪ {*a*} ∪ *set* (*RBT-Impl.keys r*)
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliR.cases*) *auto*

lemma *set-rbt-joinR*: *set* (*RBT-Impl.keys* (*rbt-joinR l a b r*)) =
set (*RBT-Impl.keys l*) ∪ {*a*} ∪ *set* (*RBT-Impl.keys r*)
proof (*induction l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
thus *?case*
by (*force simp*: *rbt-set-rbt-baliR rbt-joinR.simps*[*of l a b r*]
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *set-keys-paint*: *set* (*RBT-Impl.keys* (*paint c t*)) = *set* (*RBT-Impl.keys t*)
by (*cases t*) *auto*

lemma *set-rbt-join*: *set* (*RBT-Impl.keys* (*rbt-join l a b r*)) =
set (*RBT-Impl.keys l*) ∪ {*a*} ∪ *set* (*RBT-Impl.keys r*)
by (*simp add*: *set-rbt-joinL set-rbt-joinR set-keys-paint rbt-join-def Let-def*)

lemma *inv-rbt-join*: *inv-12 l* ⟹ *inv-12 r* ⟹ *inv-12* (*rbt-join l a b r*)
by (*auto simp*: *rbt-join-def Let-def inv1l-rbt-joinL inv1l-rbt-joinR*
inv2-rbt-joinL inv2-rbt-joinR inv-12-def)

fun *rbt-recolor* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-recolor (*Branch RBT-Impl.R t1 k v t2*) =
(*if color-of t1* = *RBT-Impl.B* ∧ *color-of t2* = *RBT-Impl.B then Branch RBT-Impl.B t1 k v t2*
else Branch RBT-Impl.R t1 k v t2)
| *rbt-recolor t* = *t*

lemma *rbt-recolor*: *inv-12 t* ⟹ *inv-12* (*rbt-recolor t*)
by (*induction t rule*: *rbt-recolor.induct*) (*auto simp*: *inv-12-def*)

fun *rbt-split-min* :: ('*a*, '*b*) *rbt* ⇒ '*a* × '*b* × ('*a*, '*b*) *rbt* **where**
rbt-split-min RBT-Impl.Empty = *undefined*
| *rbt-split-min* (*RBT-Impl.Branch - l a b r*) =
(*if is-rbt-empty l then* (*a*,*b*,*r*) *else let* (*a*′,*b*′,*l*′) = *rbt-split-min l in* (*a*′,*b*′,*rbt-join l*′ *a b r*))

lemma *rbt-split-min-set*:
rbt-split-min t = (*a*,*b*,*t*′) ⟹ *t* ≠ *RBT-Impl.Empty* ⟹
a ∈ *set* (*RBT-Impl.keys t*) ∧ *set* (*RBT-Impl.keys t*) = {*a*} ∪ *set* (*RBT-Impl.keys t*′)
by (*induction t arbitrary*: *t*′) (*auto simp*: *set-rbt-join split*: *prod.splits if-splits*)

lemma *rbt-split-min-inv*: *rbt-split-min t* = (*a*,*b*,*t*′) ⟹ *inv-12 t* ⟹ *t* ≠ *RBT-Impl.Empty* ⟹ *inv-12 t*′
by (*induction t arbitrary*: *t*′)
(*auto simp*: *inv-rbt-join split*: *if-splits prod.splits dest*: *rbt-Node*)

definition *rbt-join2* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-join2 l r = (*if is-rbt-empty r then l else let* (*a*,*b*,*r*′) = *rbt-split-min r in rbt-join l a b r*′)

lemma *set-rbt-join2*[*simp*]: *set* (*RBT-Impl.keys* (*rbt-join2 l r*)) =
set (*RBT-Impl.keys l*) ∪ *set* (*RBT-Impl.keys r*)
by (*simp add*: *rbt-join2-def rbt-split-min-set set-rbt-join split*: *prod.split*)

lemma *inv-rbt-join2*: *inv-12 l* ⟹ *inv-12 r* ⟹ *inv-12* (*rbt-join2 l r*)
by (*simp add*: *rbt-join2-def inv-rbt-join rbt-split-min-set rbt-split-min-inv split*: *prod.split*)

context *ord* **begin**

fun *rbt-split* :: ('*a*, '*b*) *rbt* ⇒ '*a* ⇒ ('*a*, '*b*) *rbt* × '*b option* × ('*a*, '*b*) *rbt* **where**
rbt-split RBT-Impl.Empty k = (*RBT-Impl.Empty*, *None*, *RBT-Impl.Empty*)
| *rbt-split* (*RBT-Impl.Branch - l a b r*) *x* =
(*if x* < *a then* (*case rbt-split l x of* (*l1*, β, *l2*) ⇒ (*l1*, β, *rbt-join l2 a b r*))
else if a < *x then* (*case rbt-split r x of* (*r1*, β, *r2*) ⇒ (*rbt-join l a b r1*, β, *r2*))
else (*l*, *Some b*, *r*))

lemma *rbt-split*: *rbt-split t x* = (*l*,β,*r*) ⟹ *inv-12 t* ⟹ *inv-12 l* ∧ *inv-12 r*
by (*induction t arbitrary*: *l r*)
(*auto simp*: *set-rbt-join inv-rbt-join rbt-greater-prop rbt-less-prop*
split: *if-splits prod.splits dest*!: *rbt-Node*)

lemma *rbt-split-size*: (*l2*,β,*r2*) = *rbt-split t2 a* ⟹ *size l2* + *size r2* ≤ *size t2*
by (*induction t2 a arbitrary*: *l2 r2 rule*: *rbt-split.induct*) (*auto split*: *if-splits prod.splits*)

function *rbt-union-rec* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-union-rec f t1 t2 = (*let* (*f*, *t2*, *t1*) =
if flip-rbt t2 t1 then (λ*k v v*'. *f k v*' *v*, *t1*, *t2*) *else* (*f*, *t2*, *t1*) *in*
if small-rbt t2 then RBT-Impl.fold (*rbt-insert-with-key f*) *t2 t1*
else (*case t1 of RBT-Impl.Empty* ⇒ *t2*
| *RBT-Impl.Branch - l1 a b r1* ⇒
case rbt-split t2 a of (*l2*, β, *r2*) ⇒
rbt-join (*rbt-union-rec f l1 l2*) *a* (*case* β *of None* ⇒ *b* | *Some b*' ⇒ *f a b b*') (*rbt-union-rec f r1 r2*)))
by *pat-completeness auto*
termination
using *rbt-split-size*
by (*relation measure* (λ(*f*,*t1*,*t2*). *size t1* + *size t2*)) (*fastforce split*: *if-splits*)+

declare *rbt-union-rec.simps*[*simp del*]

function *rbt-union-swap-rec* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ *bool* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-union-swap-rec f γ *t1 t2* = (*let* (γ, *t2*, *t1*) =
if flip-rbt t2 t1 then (¬γ, *t1*, *t2*) *else* (γ, *t2*, *t1*);
f' = (*if* γ *then* (λ*k v v*'. *f k v*' *v*) *else f*) *in*
if small-rbt t2 then RBT-Impl.fold (*rbt-insert-with-key f*') *t2 t1*
else (*case t1 of RBT-Impl.Empty* ⇒ *t2*
| *RBT-Impl.Branch - l1 a b r1* ⇒
case rbt-split t2 a of (*l2*, β, *r2*) ⇒
rbt-join (*rbt-union-swap-rec f* γ *l1 l2*) *a* (*case* β *of None* ⇒ *b* | *Some b*' ⇒ *f*' *a b b*') (*rbt-union-swap-rec f* γ *r1 r2*)))
by *pat-completeness auto*
termination

using *rbt-split-size*
by (*relation measure* (λ(*f*,γ,*t1*,*t2*). *size t1* + *size t2*)) (*fastforce split*: *if-splits*)+

declare *rbt-union-swap-rec.simps*[*simp del*]

lemma *rbt-union-swap-rec*: *rbt-union-swap-rec f* γ *t1 t2* =
rbt-union-rec (*if* γ *then* (λ*k v v'*. *f k v' v*) *else f*) *t1 t2*
proof (*induction f* γ *t1 t2 rule*: *rbt-union-swap-rec.induct*)
case (*1 f* γ *t1 t2*)
show *?case*
using *1*[*OF refl - refl refl - refl - refl*]
unfolding *rbt-union-swap-rec.simps*[*of - - t1*] *rbt-union-rec.simps*[*of - t1*]
by (*auto simp*: *Let-def split*: *rbt.splits prod.splits option.splits*)
qed

lemma *rbt-fold-rbt-insert*:
assumes *inv-12 t2*
shows *inv-12* (*RBT-Impl.fold* (*rbt-insert-with-key f*) *t1 t2*)
proof −
define *xs* **where** *xs* = *RBT-Impl.entries t1*
from *assms* **show** *?thesis*
unfolding *RBT-Impl.fold-def xs-def*[*symmetric*]
by (*induct xs rule*: *rev-induct*)
(*auto simp*: *inv-12-def rbt-insert-with-key-def ins-inv1-inv2*)
qed

lemma *rbt-union-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-union-rec f t1 t2*)
proof (*induction f t1 t2 rule*: *rbt-union-rec.induct*)
case (*1 t1 t2*)
thus *?case*
by (*auto simp*: *rbt-union-rec.simps*[*of t1 t2*] *inv-rbt-join rbt-split rbt-fold-rbt-insert*
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split if-splits dest*: *rbt-Node*)
qed

definition *map-filter-inter f t1 t2* = *List.map-filter* (λ(*k*, *v*).
case rbt-lookup t1 k of None ⇒ *None*
| *Some v'* ⇒ *Some* (*k*, *f k v' v*)) (*RBT-Impl.entries t2*)

function *rbt-inter-rec* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-inter-rec f t1 t2 = (*let* (*f*, *t2*, *t1*) =
if flip-rbt t2 t1 then (λ*k v v'*. *f k v' v*, *t1*, *t2*) *else* (*f*, *t2*, *t1*) *in*
if small-rbt t2 then rbtreeify (*map-filter-inter f t1 t2*)
else case t1 of RBT-Impl.Empty ⇒ *RBT-Impl.Empty*
| *RBT-Impl.Branch - l1 a b r1* ⇒
case rbt-split t2 a of (*l2*, β, *r2*) ⇒ *let l'* = *rbt-inter-rec f l1 l2*; *r'* = *rbt-inter-rec f r1 r2 in*
(*case* β *of None* ⇒ *rbt-join2 l' r'* | *Some b'* ⇒ *rbt-join l' a* (*f a b b'*) *r'*))

by *pat-completeness auto*
termination
using *rbt-split-size*
by (*relation measure* (λ(*f*,*t1*,*t2*). *size t1* + *size t2*)) (*fastforce split*: *if-splits*)+

declare *rbt-inter-rec.simps*[*simp del*]

function *rbt-inter-swap-rec* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ *bool* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-inter-swap-rec f γ *t1 t2* = (*let* (γ, *t2*, *t1*) =
if flip-rbt t2 t1 then (¬γ, *t1*, *t2*) *else* (γ, *t2*, *t1*);
f' = (*if* γ *then* (λ*k v v'*. *f k v' v*) *else f*) *in*
if small-rbt t2 then rbtreeify (*map-filter-inter f' t1 t2*)
else case t1 of RBT-Impl.Empty ⇒ *RBT-Impl.Empty*
| *RBT-Impl.Branch - l1 a b r1* ⇒
case rbt-split t2 a of (*l2*, β, *r2*) ⇒ *let l'* = *rbt-inter-swap-rec f* γ *l1 l2*; *r'* = *rbt-inter-swap-rec f* γ *r1 r2 in*
(*case* β *of None* ⇒ *rbt-join2 l' r'* | *Some b'* ⇒ *rbt-join l' a* (*f' a b b'*) *r'*))
by *pat-completeness auto*
termination
using *rbt-split-size*
by (*relation measure* (λ(*f*,γ,*t1*,*t2*). *size t1* + *size t2*)) (*fastforce split*: *if-splits*)+

declare *rbt-inter-swap-rec.simps*[*simp del*]

lemma *rbt-inter-swap-rec*: *rbt-inter-swap-rec f* γ *t1 t2* =
rbt-inter-rec (*if* γ *then* (λ*k v v'*. *f k v' v*) *else f*) *t1 t2*
proof (*induction f* γ *t1 t2 rule*: *rbt-inter-swap-rec.induct*)
case (*1 f* γ *t1 t2*)
show *?case*
using *1*[*OF refl - refl refl - refl - refl*]
unfolding *rbt-inter-swap-rec.simps*[*of - - t1*] *rbt-inter-rec.simps*[*of - t1*]
by (*auto simp add*: *Let-def split*: *rbt.splits prod.splits option.splits*)
qed

lemma *rbt-rbtreeify*[*simp*]: *inv-12* (*rbtreeify kvs*)
by (*simp add*: *inv-12-def rbtreeify-def inv1-rbtreeify-g inv2-rbtreeify-g*)

lemma *rbt-inter-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-inter-rec f t1 t2*)
proof(*induction f t1 t2 rule*: *rbt-inter-rec.induct*)
case (*1 t1 t2*)
thus *?case*
by (*auto simp*: *rbt-inter-rec.simps*[*of t1 t2*] *inv-rbt-join inv-rbt-join2 rbt-split Let-def*
split!: *RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split if-splits*
option.splits dest!: *rbt-Node*)
qed

definition *filter-minus t1 t2* = *filter* (λ(*k*, -). *rbt-lookup t2 k* = *None*) (*RBT-Impl.entries*

t1)

fun *rbt-minus-rec* :: ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **where**
rbt-minus-rec t1 t2 = (*if small-rbt t2 then RBT-Impl.fold* (λ*k* - *t*. *rbt-delete k t*) *t2 t1*
else if small-rbt t1 then rbtreeify (*filter-minus t1 t2*)
else case t2 of RBT-Impl.Empty ⇒ *t1*
| *RBT-Impl.Branch* - *l2 a b r2* ⇒
case rbt-split t1 a of (*l1*, -, *r1*) ⇒ *rbt-join2* (*rbt-minus-rec l1 l2*) (*rbt-minus-rec r1 r2*))

declare *rbt-minus-rec.simps*[*simp del*]

end

context *linorder* **begin**

lemma *rbt-sorted-entries-right-unique*:
⟦ (*k*, *v*) ∈ *set* (*entries t*); (*k*, *v*′) ∈ *set* (*entries t*);
rbt-sorted t ⟧ ⟹ *v* = *v*′
by(*auto dest*!: *distinct-entries inj-onD*[**where** *x*=(*k*, *v*) **and** *y*=(*k*, *v*′)] *simp add*: *distinct-map*)

lemma *rbt-sorted-fold-rbt-insertwk*:
rbt-sorted t ⟹ *rbt-sorted* (*List.fold* (λ(*k*, *v*). *rbt-insert-with-key f k v*) *xs t*)
by(*induct xs rule*: *rev-induct*)(*auto simp add*: *rbt-insertwk-rbt-sorted*)

lemma *is-rbt-fold-rbt-insertwk*:
assumes *is-rbt t1*
shows *is-rbt* (*fold* (*rbt-insert-with-key f*) *t2 t1*)
proof −
define *xs* **where** *xs* = *entries t2*
from *assms* **show** *?thesis* **unfolding** *fold-def xs-def*[*symmetric*]
by(*induct xs rule*: *rev-induct*)(*auto simp add*: *rbt-insertwk-is-rbt*)
qed

lemma *rbt-delete*: *inv-12 t* ⟹ *inv-12* (*rbt-delete x t*)
using *rbt-del-inv1-inv2*[*of t x*]
by (*auto simp*: *inv-12-def rbt-delete-def rbt-del-inv1-inv2*)

lemma *rbt-sorted-delete*: *rbt-sorted t* ⟹ *rbt-sorted* (*rbt-delete x t*)
by (*auto simp*: *rbt-delete-def rbt-del-rbt-sorted*)

lemma *rbt-fold-rbt-delete*:
assumes *inv-12 t2*
shows *inv-12* (*RBT-Impl.fold* (λ*k* - *t*. *rbt-delete k t*) *t1 t2*)
proof −
define *xs* **where** *xs* = *RBT-Impl.entries t1*
from *assms* **show** *?thesis*

unfolding *RBT-Impl.fold-def xs-def*[*symmetric*]
by (*induct xs rule*: *rev-induct*) (*auto simp*: *rbt-delete*)
qed

lemma *rbt-minus-rec*: *inv-12 t1* ⟹ *inv-12 t2* ⟹ *inv-12* (*rbt-minus-rec t1 t2*)
proof(*induction t1 t2 rule*: *rbt-minus-rec.induct*)
case (*1 t1 t2*)
thus *?case*
by (*auto simp*: *rbt-minus-rec.simps*[*of t1 t2*] *inv-rbt-join inv-rbt-join2 rbt-split rbt-fold-rbt-delete split*!: *RBT-Impl.rbt.splits RBT-Impl.color.splits prod.split if-splits dest*: *rbt-Node*)
qed

end

context *linorder* **begin**

lemma *rbt-sorted-rbt-baliL*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹ *rbt-sorted* (*rbt-baliL l a b r*)
using *rbt-greater-trans rbt-less-trans*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliL.cases*) *fastforce*+

lemma *rbt-lookup-rbt-baliL*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹ *rbt-lookup* (*rbt-baliL l a b r*) *k* = (*if k* < *a then rbt-lookup l k else if k* = *a then Some b else rbt-lookup r k*)
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliL.cases*) (*auto split*!: *if-splits*)

lemma *rbt-sorted-rbt-baliR*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹ *rbt-sorted* (*rbt-baliR l a b r*)
using *rbt-greater-trans rbt-less-trans*
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliR.cases*) *fastforce*+

lemma *rbt-lookup-rbt-baliR*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹ *rbt-lookup* (*rbt-baliR l a b r*) *k* = (*if k* < *a then rbt-lookup l k else if k* = *a then Some b else rbt-lookup r k*)
by (*cases* (*l*,*a*,*b*,*r*) *rule*: *rbt-baliR.cases*) (*auto split*!: *if-splits*)

lemma *rbt-sorted-rbt-joinL*: *rbt-sorted* (*RBT-Impl.Branch c l a b r*) ⟹ *bheight l* ≤ *bheight r* ⟹ *rbt-sorted* (*rbt-joinL l a b r*)
proof (*induction l a b r arbitrary*: *c rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
thus *?case*
by (*auto simp*: *rbt-set-rbt-baliL rbt-joinL.simps*[*of l a b r*] *set-rbt-joinL rbt-less-prop intro*!: *rbt-sorted-rbt-baliL split*!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *rbt-lookup-rbt-joinL*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹

rbt-lookup (*rbt-joinL l a b r*) *k* =
(*if k* < *a then rbt-lookup l k else if k* = *a then Some b else rbt-lookup r k*)
proof (*induction l a b r rule*: *rbt-joinL.induct*)
case (*1 l a b r*)
have *less-rbt-joinL*:
rbt-sorted r1 ⟹ *r1* |« *x* ⟹ *a* «| *r1* ⟹ *a* < *x* ⟹ *rbt-joinL l a b r1* |« *x* **for** *x r1*
using *1*(*5*)
by (*auto simp*: *rbt-less-prop rbt-greater-prop set-rbt-joinL*)
show *?case*
using *1 less-rbt-joinL rbt-lookup-rbt-baliL*[*OF rbt-sorted-rbt-joinL*[*of - l a b*], **where** *?k*=*k*]
by (*auto simp*: *rbt-joinL.simps*[*of l a b r*] *split*!: *if-splits rbt.splits color.splits*)
qed

lemma *rbt-sorted-rbt-joinR*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹
rbt-sorted (*rbt-joinR l a b r*)
proof (*induction l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
thus *?case*
by (*auto simp*: *rbt-set-rbt-baliR rbt-joinR.simps*[*of l a b r*] *set-rbt-joinR rbt-greater-prop intro*!: *rbt-sorted-rbt-baliR split*!: *RBT-Impl.rbt.splits RBT-Impl.color.splits*)
qed

lemma *rbt-lookup-rbt-joinR*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹
rbt-lookup (*rbt-joinR l a b r*) *k* =
(*if k* < *a then rbt-lookup l k else if k* = *a then Some b else rbt-lookup r k*)
proof (*induction l a b r rule*: *rbt-joinR.induct*)
case (*1 l a b r*)
have *less-rbt-joinR*:
rbt-sorted l1 ⟹ *x* «| *l1* ⟹ *l1* |« *a* ⟹ *x* < *a* ⟹ *x* «| *rbt-joinR l1 a b r* **for** *x l1*
using *1*(*6*)
by (*auto simp*: *rbt-less-prop rbt-greater-prop set-rbt-joinR*)
show *?case*
using *1 less-rbt-joinR rbt-lookup-rbt-baliR*[*OF - rbt-sorted-rbt-joinR*[*of - r a b*], **where** *?k*=*k*]
by (*auto simp*: *rbt-joinR.simps*[*of l a b r*] *split*!: *if-splits rbt.splits color.splits*)
qed

lemma *rbt-sorted-paint*: *rbt-sorted* (*paint c t*) = *rbt-sorted t*
by (*cases t*) *auto*

lemma *rbt-sorted-rbt-join*: *rbt-sorted* (*RBT-Impl.Branch c l a b r*) ⟹
rbt-sorted (*rbt-join l a b r*)
by (*auto simp*: *rbt-sorted-paint rbt-sorted-rbt-joinL rbt-sorted-rbt-joinR rbt-join-def Let-def*)

lemma *rbt-lookup-rbt-join*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹ *l* |« *a* ⟹ *a* «| *r* ⟹
rbt-lookup (*rbt-join l a b r*) *k* =
(*if k* < *a then rbt-lookup l k else if k* = *a then Some b else rbt-lookup r k*)
by (*auto simp*: *rbt-join-def Let-def rbt-lookup-rbt-joinL rbt-lookup-rbt-joinR*)

lemma *rbt-split-min-rbt-sorted*: *rbt-split-min t* = (*a*,*b*,*t'*) ⟹ *rbt-sorted t* ⟹ *t* ≠ *RBT-Impl.Empty* ⟹
rbt-sorted t' ∧ (∀ *x* ∈ *set* (*RBT-Impl.keys t'*). *a* < *x*)
by (*induction t arbitrary*: *t'*)
(*fastforce simp*: *rbt-split-min-set rbt-sorted-rbt-join set-rbt-join rbt-less-prop rbt-greater-prop*
split: *if-splits prod.splits*)+

lemma *rbt-split-min-rbt-lookup*: *rbt-split-min t* = (*a*,*b*,*t'*) ⟹ *rbt-sorted t* ⟹ *t* ≠ *RBT-Impl.Empty* ⟹
rbt-lookup t k = (*if k* < *a then None else if k* = *a then Some b else rbt-lookup t' k*)
apply (*induction t arbitrary*: *a b t'*)
apply(*simp-all split*: *if-splits prod.splits*)
apply(*auto simp*: *rbt-less-prop rbt-split-min-set rbt-lookup-rbt-join rbt-split-min-rbt-sorted*)
done

lemma *rbt-sorted-rbt-join2*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹
∀ *x* ∈ *set* (*RBT-Impl.keys l*). ∀ *y* ∈ *set* (*RBT-Impl.keys r*). *x* < *y* ⟹ *rbt-sorted* (*rbt-join2 l r*)
by (*simp add*: *rbt-join2-def rbt-sorted-rbt-join rbt-split-min-set rbt-split-min-rbt-sorted set-rbt-join*
rbt-greater-prop rbt-less-prop split: *prod.split*)

lemma *rbt-lookup-rbt-join2*: *rbt-sorted l* ⟹ *rbt-sorted r* ⟹
∀ *x* ∈ *set* (*RBT-Impl.keys l*). ∀ *y* ∈ *set* (*RBT-Impl.keys r*). *x* < *y* ⟹
rbt-lookup (*rbt-join2 l r*) *k* = (*case rbt-lookup l k of None* ⇒ *rbt-lookup r k* | *Some v* ⇒ *Some v*)
using *rbt-lookup-keys*
by (*fastforce simp*: *rbt-join2-def rbt-greater-prop rbt-less-prop rbt-lookup-rbt-join*
rbt-split-min-rbt-lookup rbt-split-min-rbt-sorted rbt-split-min-set split: *option.splits prod.splits*)

lemma *rbt-split-props*: *rbt-split t x* = (*l*,*β*,*r*) ⟹ *rbt-sorted t* ⟹
set (*RBT-Impl.keys l*) = {*a* ∈ *set* (*RBT-Impl.keys t*). *a* < *x*} ∧
set (*RBT-Impl.keys r*) = {*a* ∈ *set* (*RBT-Impl.keys t*). *x* < *a*} ∧
rbt-sorted l ∧ *rbt-sorted r*
apply (*induction t arbitrary*: *l r*)
apply(*simp-all split*!: *prod.splits if-splits*)
apply(*force simp*: *set-rbt-join rbt-greater-prop rbt-less-prop*
intro: *rbt-sorted-rbt-join*)+
done

lemma *rbt-split-lookup*: *rbt-split t x* = (*l*,*β*,*r*) ⟹ *rbt-sorted t* ⟹

rbt-lookup t k = (if k < x then rbt-lookup l k else if k = x then β else rbt-lookup r k)
proof (*induction t arbitrary*: *x l β r*)
case (*Branch c t1 a b t2*)
have *rbt-sorted r1 r1* |« *a* **if** *rbt-split t1 x = (l, β, r1)* **for** *r1*
using *rbt-split-props Branch*(*4*) *that*
by (*fastforce simp*: *rbt-less-prop*)+
moreover have *rbt-sorted l1 a* «| *l1* **if** *rbt-split t2 x = (l1, β, r)* **for** *l1*
using *rbt-split-props Branch*(*4*) *that*
by (*fastforce simp*: *rbt-greater-prop*)+
ultimately show *?case*
using *Branch rbt-lookup-rbt-join*[*of t1 - a b k*] *rbt-lookup-rbt-join*[*of - t2 a b k*]
by (*auto split*!: *if-splits prod.splits*)
qed *simp*

lemma *rbt-sorted-fold-insertwk*: *rbt-sorted t* ⟹
rbt-sorted (*RBT-Impl.fold* (*rbt-insert-with-key f*) *t' t*)
by (*induct t' arbitrary*: *t*)
(*simp-all add*: *rbt-insertwk-rbt-sorted*)

lemma *rbt-lookup-iff-keys*:
rbt-sorted t ⟹ *set* (*RBT-Impl.keys t*) = {*k*. ∃ *v*. *rbt-lookup t k = Some v*}
rbt-sorted t ⟹ *rbt-lookup t k = None* ⟷ *k* ∉ *set* (*RBT-Impl.keys t*)
rbt-sorted t ⟹ (∃ *v*. *rbt-lookup t k = Some v*) ⟷ *k* ∈ *set* (*RBT-Impl.keys t*)
using *entry-in-tree-keys rbt-lookup-keys*[*of t*]
by *force*+

lemma *rbt-lookup-fold-rbt-insertwk*:
assumes *t1*: *rbt-sorted t1* **and** *t2*: *rbt-sorted t2*
shows *rbt-lookup* (*fold* (*rbt-insert-with-key f*) *t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *rbt-lookup t2 k*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *Some v*
| *Some w* ⇒ *Some* (*f k w v*))
proof −
define *xs* **where** *xs = entries t1*
hence *dt1*: *distinct* (*map fst xs*) **using** *t1* **by**(*simp add*: *distinct-entries*)
with *t2* **show** *?thesis*
unfolding *fold-def map-of-entries*[*OF t1*, *symmetric*]
xs-def[*symmetric*] *distinct-map-of-rev*[*OF dt1*, *symmetric*]
apply(*induct xs rule*: *rev-induct*)
apply(*auto simp add*: *rbt-lookup-rbt-insertwk rbt-sorted-fold-rbt-insertwk split*: *option.splits*)
apply(*auto simp add*: *distinct-map-of-rev intro*: *rev-image-eqI*)
done
qed

lemma *rbt-lookup-union-rec*: *rbt-sorted t1* ⟹ *rbt-sorted t2* ⟹
rbt-sorted (*rbt-union-rec f t1 t2*) ∧ *rbt-lookup* (*rbt-union-rec f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *rbt-lookup t2 k*

```
  | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v
              | Some w ⇒ Some (f k v w)))
proof(induction f t1 t2 arbitrary: k rule: rbt-union-rec.induct)
  case (1 f t1 t2)
  obtain f' t1' t2' where flip: (f', t2', t1') =
    (if flip-rbt t2 t1 then (λk v v'. f k v' v, t1, t2) else (f, t2, t1))
    by fastforce
  have rbt-sorted': rbt-sorted t1' rbt-sorted t2'
    using 1(3,4) flip
    by (auto split: if-splits)
  show ?case
  proof (cases t1')
    case Empty
    show ?thesis
      unfolding rbt-union-rec.simps[of - t1] flip[symmetric]
      using flip rbt-sorted' rbt-split-props[of t2]
      by (auto simp: Empty rbt-lookup-fold-rbt-insertwk
          intro!: rbt-sorted-fold-insertwk split: if-splits option.splits)
  next
    case (Branch c l1 a b r1)
    {
      assume not-small: ¬small-rbt t2'
      obtain l2 β r2 where rbt-split-t2': rbt-split t2' a = (l2, β, r2)
        by (cases rbt-split t2' a) auto
      have rbt-sort: rbt-sorted l1 rbt-sorted r1
        using 1(3,4) flip
        by (auto simp: Branch split: if-splits)
      note rbt-split-t2'-props = rbt-split-props[OF rbt-split-t2' rbt-sorted'(2)]
      have union-l1-l2: rbt-sorted (rbt-union-rec f' l1 l2) rbt-lookup (rbt-union-rec
f' l1 l2) k =
        (case rbt-lookup l1 k of None ⇒ rbt-lookup l2 k
        | Some v ⇒ (case rbt-lookup l2 k of None ⇒ Some v | Some w ⇒ Some (f'
k v w))) for k
          using 1(1)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
        by (auto simp: not-small)
      have union-r1-r2: rbt-sorted (rbt-union-rec f' r1 r2) rbt-lookup (rbt-union-rec
f' r1 r2) k =
        (case rbt-lookup r1 k of None ⇒ rbt-lookup r2 k
        | Some v ⇒ (case rbt-lookup r2 k of None ⇒ Some v | Some w ⇒ Some (f'
k v w))) for k
          using 1(2)[OF flip refl refl - Branch rbt-split-t2'[symmetric]] rbt-sort
rbt-split-t2'-props
        by (auto simp: not-small)
      have union-l1-l2-keys: set (RBT-Impl.keys (rbt-union-rec f' l1 l2)) =
       set (RBT-Impl.keys l1) ∪ set (RBT-Impl.keys l2)
        using rbt-sorted'(1) rbt-split-t2'-props
        by (auto simp: Branch rbt-lookup-iff-keys(1) union-l1-l2 split: option.splits)
      have union-r1-r2-keys: set (RBT-Impl.keys (rbt-union-rec f' r1 r2)) =
```

```
      set (RBT-Impl.keys r1) ∪ set (RBT-Impl.keys r2)
       using rbt-sorted'(1) rbt-split-t2'-props
      by (auto simp: Branch rbt-lookup-iff-keys(1) union-r1-r2 split: option.splits)
     have union-l1-l2-less: rbt-union-rec f' l1 l2 |« a
       using rbt-sorted'(1) rbt-split-t2'-props
       by (auto simp: Branch rbt-less-prop union-l1-l2-keys)
     have union-r1-r2-greater: a «| rbt-union-rec f' r1 r2
       using rbt-sorted'(1) rbt-split-t2'-props
       by (auto simp: Branch rbt-greater-prop union-r1-r2-keys)
     have rbt-lookup (rbt-union-rec f t1 t2) k =
      (case rbt-lookup t1' k of None ⇒ rbt-lookup t2' k
      | Some v ⇒ (case rbt-lookup t2' k of None ⇒ Some v | Some w ⇒ Some (f'
k v w)))
       using rbt-sorted' union-l1-l2 union-r1-r2 rbt-split-t2'-props
         union-l1-l2-less union-r1-r2-greater not-small
       by (auto simp: rbt-union-rec.simps[of - t1] flip[symmetric] Branch
            rbt-split-t2' rbt-lookup-rbt-join rbt-split-lookup[OF rbt-split-t2'] split:
option.splits)
     moreover have rbt-sorted (rbt-union-rec f t1 t2)
       using rbt-sorted' rbt-split-t2'-props not-small
     by (auto simp: rbt-union-rec.simps[of - t1] flip[symmetric] Branch rbt-split-t2'
          union-l1-l2 union-r1-r2 union-l1-l2-keys union-r1-r2-keys rbt-less-prop
          rbt-greater-prop intro!: rbt-sorted-rbt-join)
     ultimately have ?thesis
       using flip
       by (auto split: if-splits option.splits)
   }
   then show ?thesis
     unfolding rbt-union-rec.simps[of - t1] flip[symmetric]
     using rbt-sorted' flip
      by (auto simp: rbt-sorted-fold-insertwk rbt-lookup-fold-rbt-insertwk split: op-
tion.splits)
  qed
qed

lemma rbtreeify-map-filter-inter:
  fixes f :: 'a ⇒ 'b ⇒ 'b ⇒ 'b
  assumes rbt-sorted t2
  shows rbt-sorted (rbtreeify (map-filter-inter f t1 t2))
    rbt-lookup (rbtreeify (map-filter-inter f t1 t2)) k =
    (case rbt-lookup t1 k of None ⇒ None
    | Some v ⇒ (case rbt-lookup t2 k of None ⇒ None | Some w ⇒ Some (f k v
w)))
proof −
  have map-of-map-filter: map-of (List.map-filter (λ(k, v).
    case rbt-lookup t1 k of None ⇒ None | Some v' ⇒ Some (k, f k v' v)) xs) k =
    (case rbt-lookup t1 k of None ⇒ None
    | Some v ⇒ (case map-of xs k of None ⇒ None | Some w ⇒ Some (f k v w)))
for xs k
```

```
    by (induction xs) (auto simp: List.map-filter-def split: option.splits)
  have map-fst-map-filter: map fst (List.map-filter (λ(k, v).
    case rbt-lookup t1 k of None ⇒ None | Some v' ⇒ Some (k, f k v' v)) xs) =
    filter (λk. rbt-lookup t1 k ≠ None) (map fst xs) for xs
    by (induction xs) (auto simp: List.map-filter-def split: option.splits)
  have sorted (map fst (map-filter-inter f t1 t2))
    using sorted-filter[of id] rbt-sorted-entries[OF assms]
    by (auto simp: map-filter-inter-def map-fst-map-filter)
  moreover have distinct (map fst (map-filter-inter f t1 t2))
    using distinct-filter distinct-entries[OF assms]
    by (auto simp: map-filter-inter-def map-fst-map-filter)
  ultimately show
    rbt-sorted (rbtreeify (map-filter-inter f t1 t2))
    rbt-lookup (rbtreeify (map-filter-inter f t1 t2)) k =
      (case rbt-lookup t1 k of None ⇒ None
      | Some v ⇒ (case rbt-lookup t2 k of None ⇒ None | Some w ⇒ Some (f k v
w)))
    using rbt-sorted-rbtreeify
    by (auto simp: rbt-lookup-rbtreeify map-filter-inter-def map-of-map-filter
        map-of-entries[OF assms] split: option.splits)
qed

lemma rbt-lookup-inter-rec: rbt-sorted t1 ⟹ rbt-sorted t2 ⟹
  rbt-sorted (rbt-inter-rec f t1 t2) ∧ rbt-lookup (rbt-inter-rec f t1 t2) k =
  (case rbt-lookup t1 k of None ⇒ None
  | Some v ⇒ (case rbt-lookup t2 k of None ⇒ None | Some w ⇒ Some (f k v w)))
proof(induction f t1 t2 arbitrary: k rule: rbt-inter-rec.induct)
  case (1 f t1 t2)
  obtain f' t1' t2' where flip: (f', t2', t1') =
    (if flip-rbt t2 t1 then (λk v v'. f k v' v, t1, t2) else (f, t2, t1))
    by fastforce
  have rbt-sorted': rbt-sorted t1' rbt-sorted t2'
    using 1(3,4) flip
    by (auto split: if-splits)
  show ?case
  proof (cases t1')
    case Empty
    show ?thesis
      unfolding rbt-inter-rec.simps[of - t1] flip[symmetric]
     using flip rbt-sorted' rbt-split-props[of t2] rbtreeify-map-filter-inter[OF rbt-sorted'(2)]
      by (auto simp: Empty split: option.splits)
  next
    case (Branch c l1 a b r1)
    {
      assume not-small: ¬small-rbt t2'
      obtain l2 β r2 where rbt-split-t2': rbt-split t2' a = (l2, β, r2)
        by (cases rbt-split t2' a) auto
      note rbt-split-t2'-props = rbt-split-props[OF rbt-split-t2' rbt-sorted'(2)]
      have rbt-sort: rbt-sorted l1 rbt-sorted r1 rbt-sorted l2 rbt-sorted r2
```

using *1*(*3*,*4*) *flip*
by (*auto simp*: *Branch rbt-split-t2'-props split*: *if-splits*)
have *inter-l1-l2*: *rbt-sorted* (*rbt-inter-rec f' l1 l2*) *rbt-lookup* (*rbt-inter-rec f' l1 l2*) *k* =
(*case rbt-lookup l1 k of None* ⇒ *None*
| *Some v* ⇒ (*case rbt-lookup l2 k of None* ⇒ *None* | *Some w* ⇒ *Some* (*f' k v w*))) **for** *k*
using *1*(*1*)[*OF flip refl refl - Branch rbt-split-t2'*[*symmetric*]] *rbt-sort rbt-split-t2'-props*
by (*auto simp*: *not-small*)
have *inter-r1-r2*: *rbt-sorted* (*rbt-inter-rec f' r1 r2*) *rbt-lookup* (*rbt-inter-rec f' r1 r2*) *k* =
(*case rbt-lookup r1 k of None* ⇒ *None*
| *Some v* ⇒ (*case rbt-lookup r2 k of None* ⇒ *None* | *Some w* ⇒ *Some* (*f' k v w*))) **for** *k*
using *1*(*2*)[*OF flip refl refl - Branch rbt-split-t2'*[*symmetric*]] *rbt-sort rbt-split-t2'-props*
by (*auto simp*: *not-small*)
have *inter-l1-l2-keys*: *set* (*RBT-Impl.keys* (*rbt-inter-rec f' l1 l2*)) =
set (*RBT-Impl.keys l1*) ∩ *set* (*RBT-Impl.keys l2*)
using *inter-l1-l2*(*1*)
by (*auto simp*: *rbt-lookup-iff-keys*(*1*) *inter-l1-l2*(*2*) *rbt-sort split*: *option.splits*)
have *inter-r1-r2-keys*: *set* (*RBT-Impl.keys* (*rbt-inter-rec f' r1 r2*)) =
set (*RBT-Impl.keys r1*) ∩ *set* (*RBT-Impl.keys r2*)
using *inter-r1-r2*(*1*)
by (*auto simp*: *rbt-lookup-iff-keys*(*1*) *inter-r1-r2*(*2*) *rbt-sort split*: *option.splits*)
have *inter-l1-l2-less*: *rbt-inter-rec f' l1 l2* |« *a*
using *rbt-sorted'*(*1*) *rbt-split-t2'-props*
by (*auto simp*: *Branch rbt-less-prop inter-l1-l2-keys*)
have *inter-r1-r2-greater*: *a* «| *rbt-inter-rec f' r1 r2*
using *rbt-sorted'*(*1*) *rbt-split-t2'-props*
by (*auto simp*: *Branch rbt-greater-prop inter-r1-r2-keys*)
have *rbt-lookup-join2*: *rbt-lookup* (*rbt-join2* (*rbt-inter-rec f' l1 l2*) (*rbt-inter-rec f' r1 r2*)) *k* =
(*case rbt-lookup* (*rbt-inter-rec f' l1 l2*) *k of None* ⇒ *rbt-lookup* (*rbt-inter-rec f' r1 r2*) *k*
| *Some v* ⇒ *Some v*) **for** *k*
using *rbt-lookup-rbt-join2*[*OF inter-l1-l2*(*1*) *inter-r1-r2*(*1*)] *rbt-sorted'*(*1*)
by (*fastforce simp*: *Branch inter-l1-l2-keys inter-r1-r2-keys rbt-less-prop rbt-greater-prop*)
have *rbt-lookup-l1-k*: *rbt-lookup l1 k* = *Some v* ⟹ *k* < *a* **for** *k v*
using *rbt-sorted'*(*1*) *rbt-lookup-iff-keys*(*3*)
by (*auto simp*: *Branch rbt-less-prop*)
have *rbt-lookup-r1-k*: *rbt-lookup r1 k* = *Some v* ⟹ *a* < *k* **for** *k v*
using *rbt-sorted'*(*1*) *rbt-lookup-iff-keys*(*3*)
by (*auto simp*: *Branch rbt-greater-prop*)
have *rbt-lookup* (*rbt-inter-rec f t1 t2*) *k* =
(*case rbt-lookup t1' k of None* ⇒ *None*

```
      | Some v ⇒ (case rbt-lookup t2' k of None ⇒ None | Some w ⇒ Some (f' k
v w)))
      by (auto simp: Let-def rbt-inter-rec.simps[of - t1] flip[symmetric] not-small
Branch
            rbt-split-t2' rbt-lookup-join2 rbt-lookup-rbt-join inter-l1-l2-less in-
ter-r1-r2-greater
          rbt-split-lookup[OF rbt-split-t2' rbt-sorted'(2)] inter-l1-l2 inter-r1-r2
          split!: if-splits option.splits dest: rbt-lookup-l1-k rbt-lookup-r1-k)
    moreover have rbt-sorted (rbt-inter-rec f t1 t2)
      using rbt-sorted' inter-l1-l2 inter-r1-r2 rbt-split-t2'-props not-small
       by (auto simp: Let-def rbt-inter-rec.simps[of - t1] flip[symmetric] Branch
rbt-split-t2'
          rbt-less-prop rbt-greater-prop inter-l1-l2-less inter-r1-r2-greater
        inter-l1-l2-keys inter-r1-r2-keys intro!: rbt-sorted-rbt-join rbt-sorted-rbt-join2
          split: if-splits option.splits dest!: bspec)
    ultimately have ?thesis
      using flip
      by (auto split: if-splits split: option.splits)
  }
  then show ?thesis
    unfolding rbt-inter-rec.simps[of - t1] flip[symmetric]
    using rbt-sorted' flip rbtreeify-map-filter-inter[OF rbt-sorted'(2)]
    by (auto split: option.splits)
 qed
qed

lemma rbt-lookup-delete:
 assumes inv-12 t rbt-sorted t
 shows rbt-lookup (rbt-delete x t) k = (if x = k then None else rbt-lookup t k)
proof −
 note rbt-sorted-del = rbt-del-rbt-sorted[OF assms(2), of x]
 show ?thesis
   using assms rbt-sorted-del rbt-del-in-tree rbt-lookup-from-in-tree[OF assms(2)
rbt-sorted-del]
   by (fastforce simp: inv-12-def rbt-delete-def rbt-lookup-iff-keys(2) keys-entries)
qed

lemma fold-rbt-delete:
 assumes inv-12 t1 rbt-sorted t1 rbt-sorted t2
 shows inv-12 (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) ∧
   rbt-sorted (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) ∧
   rbt-lookup (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) k =
   (case rbt-lookup t1 k of None ⇒ None
   | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v | - ⇒ None))
proof −
 define xs where xs = RBT-Impl.entries t2
 show inv-12 (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) ∧
   rbt-sorted (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) ∧
   rbt-lookup (RBT-Impl.fold (λk - t. rbt-delete k t) t2 t1) k =
```

```
    (case rbt-lookup t1 k of None ⇒ None
    | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v | - ⇒ None))
    using assms(1,2)
   unfolding map-of-entries[OF assms(3), symmetric] RBT-Impl.fold-def xs-def[symmetric]
    by (induction xs arbitrary: t1 rule: rev-induct)
       (auto simp: rbt-delete rbt-sorted-delete rbt-lookup-delete split!: option.splits)
qed

lemma rbtreeify-filter-minus:
  assumes rbt-sorted t1
  shows rbt-sorted (rbtreeify (filter-minus t1 t2)) ∧
    rbt-lookup (rbtreeify (filter-minus t1 t2)) k =
    (case rbt-lookup t1 k of None ⇒ None
    | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v | - ⇒ None))
proof −
  have map-of-filter: map-of (filter (λ(k, -). rbt-lookup t2 k = None) xs) k =
    (case map-of xs k of None ⇒ None
    | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v | Some x ⇒ Map.empty
x))
      for xs :: ('a × 'b) list
    by (induction xs) (auto split: option.splits)
  have map-fst-filter-minus: map fst (filter-minus t1 t2) =
    filter (λk. rbt-lookup t2 k = None) (map fst (RBT-Impl.entries t1))
    by (auto simp: filter-minus-def filter-map comp-def case-prod-unfold)
  have sorted (map fst (filter-minus t1 t2)) distinct (map fst (filter-minus t1 t2))
    using distinct-filter distinct-entries[OF assms]
      sorted-filter[of id] rbt-sorted-entries[OF assms]
    by (auto simp: map-fst-filter-minus intro!: rbt-sorted-rbtreeify)
  then show ?thesis
   by (auto simp: rbt-lookup-rbtreeify filter-minus-def map-of-filter map-of-entries[OF
assms]
       intro!: rbt-sorted-rbtreeify)
qed

lemma rbt-lookup-minus-rec: inv-12 t1 ⟹ rbt-sorted t1 ⟹ rbt-sorted t2 ⟹
  rbt-sorted (rbt-minus-rec t1 t2) ∧ rbt-lookup (rbt-minus-rec t1 t2) k =
  (case rbt-lookup t1 k of None ⇒ None
  | Some v ⇒ (case rbt-lookup t2 k of None ⇒ Some v | - ⇒ None))
proof(induction t1 t2 arbitrary: k rule: rbt-minus-rec.induct)
  case (1 t1 t2)
  show ?case
  proof (cases t2)
    case Empty
    show ?thesis
      using rbtreeify-filter-minus[OF 1(4)] 1(4)
      by (auto simp: rbt-minus-rec.simps[of t1] Empty split: option.splits)
  next
    case (Branch c l2 a b r2)
    {
```

assume *not-small*: ¬*small-rbt t2* ¬*small-rbt t1*
obtain *l1 β r1* **where** *rbt-split-t1*: *rbt-split t1 a* = (*l1*, *β*, *r1*)
by (*cases rbt-split t1 a*) *auto*
note *rbt-split-t1-props* = *rbt-split-props*[*OF rbt-split-t1 1*(*4*)]
have *minus-l1-l2*: *rbt-sorted* (*rbt-minus-rec l1 l2*)
rbt-lookup (*rbt-minus-rec l1 l2*) *k* =
(*case rbt-lookup l1 k of None* ⇒ *None*
| *Some v* ⇒ (*case rbt-lookup l2 k of None* ⇒ *Some v* | *Some x* ⇒ *None*))
for *k*
using *1*(*1*)[*OF not-small Branch rbt-split-t1*[*symmetric*] *refl*] *1*(*5*) *rbt-split-t1-props*
rbt-split[*OF rbt-split-t1 1*(*3*)]
by (*auto simp*: *Branch*)
have *minus-r1-r2*: *rbt-sorted* (*rbt-minus-rec r1 r2*)
rbt-lookup (*rbt-minus-rec r1 r2*) *k* =
(*case rbt-lookup r1 k of None* ⇒ *None*
| *Some v* ⇒ (*case rbt-lookup r2 k of None* ⇒ *Some v* | *Some x* ⇒ *None*))
for *k*
using *1*(*2*)[*OF not-small Branch rbt-split-t1*[*symmetric*] *refl*] *1*(*5*) *rbt-split-t1-props*
rbt-split[*OF rbt-split-t1 1*(*3*)]
by (*auto simp*: *Branch*)
have *minus-l1-l2-keys*: *set* (*RBT-Impl.keys* (*rbt-minus-rec l1 l2*)) =
set (*RBT-Impl.keys l1*) − *set* (*RBT-Impl.keys l2*)
using *minus-l1-l2*(*1*) *1*(*5*) *rbt-lookup-iff-keys*(*3*) *rbt-split-t1-props*
by (*auto simp*: *Branch rbt-lookup-iff-keys*(*1*) *minus-l1-l2*(*2*) *split*: *option.splits*)
have *minus-r1-r2-keys*: *set* (*RBT-Impl.keys* (*rbt-minus-rec r1 r2*)) =
set (*RBT-Impl.keys r1*) − *set* (*RBT-Impl.keys r2*)
using *minus-r1-r2*(*1*) *1*(*5*) *rbt-lookup-iff-keys*(*3*) *rbt-split-t1-props*
by (*auto simp*: *Branch rbt-lookup-iff-keys*(*1*) *minus-r1-r2*(*2*) *split*: *option.splits*)
have *rbt-lookup-join2*: *rbt-lookup* (*rbt-join2* (*rbt-minus-rec l1 l2*) (*rbt-minus-rec r1 r2*)) *k* =
(*case rbt-lookup* (*rbt-minus-rec l1 l2*) *k of None* ⇒ *rbt-lookup* (*rbt-minus-rec r1 r2*) *k*
| *Some v* ⇒ *Some v*) **for** *k*
using *rbt-lookup-rbt-join2*[*OF minus-l1-l2*(*1*) *minus-r1-r2*(*1*)] *rbt-split-t1-props*
by (*fastforce simp*: *minus-l1-l2-keys minus-r1-r2-keys*)
have *lookup-l1-r1-a*: *rbt-lookup l1 a* = *None rbt-lookup r1 a* = *None*
using *rbt-split-t1-props*
by (*auto simp*: *rbt-lookup-iff-keys*(*2*))
have *rbt-lookup* (*rbt-minus-rec t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *None*
| *Some v* ⇒ (*case rbt-lookup t2 k of None* ⇒ *Some v* | - ⇒ *None*))
using *not-small rbt-lookup-iff-keys*(*2*)[*of l1*] *rbt-lookup-iff-keys*(*3*)[*of l1*]
rbt-lookup-iff-keys(*3*)[*of r1*] *rbt-split-t1-props*
using [[*simp-depth-limit* = *2*]]
by (*auto simp*: *rbt-minus-rec.simps*[*of t1*] *Branch rbt-split-t1 rbt-lookup-join2*
minus-l1-l2(*2*) *minus-r1-r2*(*2*) *rbt-split-lookup*[*OF rbt-split-t1 1*(*4*)]
lookup-l1-r1-a

```
        split: option.splits)
    moreover have rbt-sorted (rbt-minus-rec t1 t2)
    using not-small minus-l1-l2(1) minus-r1-r2(1) rbt-split-t1-props rbt-sorted-rbt-join2
    by (fastforce simp: rbt-minus-rec.simps[of t1] Branch rbt-split-t1 minus-l1-l2-keys
minus-r1-r2-keys)
    ultimately have ?thesis
      by (auto split: if-splits split: option.splits)
  }
  then show ?thesis
    using fold-rbt-delete[OF 1(3,4,5)] rbtreeify-filter-minus[OF 1(4)]
    by (auto simp: rbt-minus-rec.simps[of t1])
 qed
qed

end

context ord begin

definition rbt-union-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
⇒ ('a, 'b) rbt
where
  rbt-union-with-key f t1 t2 = paint B (rbt-union-swap-rec f False t1 t2)

definition rbt-union-with where
  rbt-union-with f = rbt-union-with-key (λ-. f)

definition rbt-union where
  rbt-union = rbt-union-with-key (%- - rv. rv)

definition rbt-inter-with-key :: ('a ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a, 'b) rbt ⇒ ('a, 'b) rbt
⇒ ('a, 'b) rbt
where
  rbt-inter-with-key f t1 t2 = paint B (rbt-inter-swap-rec f False t1 t2)

definition rbt-inter-with where
  rbt-inter-with f = rbt-inter-with-key (λ-. f)

definition rbt-inter where
  rbt-inter = rbt-inter-with-key (λ- - rv. rv)

definition rbt-minus where
  rbt-minus t1 t2 = paint B (rbt-minus-rec t1 t2)

end

context linorder begin

lemma is-rbt-rbt-unionwk [simp]:
  ⟦ is-rbt t1; is-rbt t2 ⟧ ⟹ is-rbt (rbt-union-with-key f t1 t2)
```

using *rbt-union-rec rbt-lookup-union-rec*
by (*fastforce simp*: *rbt-union-with-key-def rbt-union-swap-rec is-rbt-def inv-12-def*)

lemma *rbt-lookup-rbt-unionwk*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-union-with-key f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *rbt-lookup t2 k*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *Some v*
| *Some w* ⇒ *Some* (*f k v w*))
using *rbt-lookup-union-rec*
by (*auto simp*: *rbt-union-with-key-def rbt-union-swap-rec*)

lemma *rbt-unionw-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union-with f lt rt*)
by(*simp add*: *rbt-union-with-def*)

lemma *rbt-union-is-rbt*: ⟦ *is-rbt lt*; *is-rbt rt* ⟧ ⟹ *is-rbt* (*rbt-union lt rt*)
by(*simp add*: *rbt-union-def*)

lemma *rbt-lookup-rbt-union*:
⟦ *rbt-sorted s*; *rbt-sorted t* ⟧ ⟹
rbt-lookup (*rbt-union s t*) = *rbt-lookup s* ++ *rbt-lookup t*
by(*rule ext*)(*simp add*: *rbt-lookup-rbt-unionwk rbt-union-def map-add-def split*: *option.split*)

lemma *rbt-interwk-is-rbt* [*simp*]:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with-key f t1 t2*)
using *rbt-inter-rec rbt-lookup-inter-rec*
by (*fastforce simp*: *rbt-inter-with-key-def rbt-inter-swap-rec is-rbt-def inv-12-def rbt-sorted-paint*)

lemma *rbt-interw-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter-with f t1 t2*)
by(*simp add*: *rbt-inter-with-def*)

lemma *rbt-inter-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ ⟹ *is-rbt* (*rbt-inter t1 t2*)
by(*simp add*: *rbt-inter-def*)

lemma *rbt-lookup-rbt-interwk*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧
⟹ *rbt-lookup* (*rbt-inter-with-key f t1 t2*) *k* =
(*case rbt-lookup t1 k of None* ⇒ *None*
| *Some v* ⇒ *case rbt-lookup t2 k of None* ⇒ *None*
| *Some w* ⇒ *Some* (*f k v w*))
using *rbt-lookup-inter-rec*
by (*auto simp*: *rbt-inter-with-key-def rbt-inter-swap-rec*)

lemma *rbt-lookup-rbt-inter*:
⟦ *rbt-sorted t1*; *rbt-sorted t2* ⟧

$\Longrightarrow$ *rbt-lookup* (*rbt-inter t1 t2*) = *rbt-lookup t2* |‘ *dom* (*rbt-lookup t1*)
by(*auto simp add*: *rbt-inter-def rbt-lookup-rbt-interwk restrict-map-def split*: *option.split*)

lemma *rbt-minus-is-rbt*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧ $\Longrightarrow$ *is-rbt* (*rbt-minus t1 t2*)
using *rbt-minus-rec*[*of t1 t2*] *rbt-lookup-minus-rec*[*of t1 t2*]
by (*auto simp*: *rbt-minus-def is-rbt-def inv-12-def*)

lemma *rbt-lookup-rbt-minus*:
⟦ *is-rbt t1*; *is-rbt t2* ⟧
$\Longrightarrow$ *rbt-lookup* (*rbt-minus t1 t2*) = *rbt-lookup t1* |‘ (− *dom* (*rbt-lookup t2*))
by (*rule ext*)
(*auto simp*: *rbt-minus-def is-rbt-def inv-12-def restrict-map-def rbt-lookup-minus-rec split*: *option.splits*)

end

130.11 Code generator setup

lemmas [*code*] =
ord.rbt-less-prop
ord.rbt-greater-prop
ord.rbt-sorted.simps
ord.rbt-lookup.simps
ord.is-rbt-def
ord.rbt-ins.simps
ord.rbt-insert-with-key-def
ord.rbt-insertw-def
ord.rbt-insert-def
ord.rbt-del-from-left.simps
ord.rbt-del-from-right.simps
ord.rbt-del.simps
ord.rbt-delete-def
ord.rbt-split.simps
ord.rbt-union-swap-rec.simps
ord.map-filter-inter-def
ord.rbt-inter-swap-rec.simps
ord.filter-minus-def
ord.rbt-minus-rec.simps
ord.rbt-union-with-key-def
ord.rbt-union-with-def
ord.rbt-union-def
ord.rbt-inter-with-key-def
ord.rbt-inter-with-def
ord.rbt-inter-def
ord.rbt-minus-def
ord.rbt-map-entry.simps
ord.rbt-bulkload-def

More efficient implementations for *entries* and *keys*

definition *gen-entries* ::
(('a × 'b) × ('a, 'b) rbt) list ⇒ ('a, 'b) rbt ⇒ ('a × 'b) list
where
gen-entries kvts t = *entries t* @ *concat* (*map* (λ(*kv*, *t*). *kv* # *entries t*) *kvts*)

lemma *gen-entries-simps* [*simp*, *code*]:
gen-entries [] *Empty* = []
gen-entries ((*kv*, *t*) # *kvts*) *Empty* = *kv* # *gen-entries kvts t*
gen-entries kvts (*Branch c l k v r*) = *gen-entries* (((*k*, *v*), *r*) # *kvts*) *l*
by(*simp-all add*: *gen-entries-def*)

lemma *entries-code* [*code*]:
entries = *gen-entries* []
by(*simp add*: *gen-entries-def fun-eq-iff*)

definition *gen-keys* :: ('a × ('a, 'b) rbt) list ⇒ ('a, 'b) rbt ⇒ 'a list
where *gen-keys kts t* = *RBT-Impl.keys t* @ *concat* (*List.map* (λ(*k*, *t*). *k* # *keys t*) *kts*)

lemma *gen-keys-simps* [*simp*, *code*]:
gen-keys [] *Empty* = []
gen-keys ((*k*, *t*) # *kts*) *Empty* = *k* # *gen-keys kts t*
gen-keys kts (*Branch c l k v r*) = *gen-keys* ((*k*, *r*) # *kts*) *l*
by(*simp-all add*: *gen-keys-def*)

lemma *keys-code* [*code*]:
keys = *gen-keys* []
by(*simp add*: *gen-keys-def fun-eq-iff*)

Restore original type constraints for constants

setup ‹
fold Sign.add-const-constraint
[(***const-name*** ‹*rbt-less*›, *SOME* ***typ*** ‹('a :: *order*) ⇒ ('a, 'b) *rbt* ⇒ *bool*›),
(***const-name*** ‹*rbt-greater*›, *SOME* ***typ*** ‹('a :: *order*) ⇒ ('a, 'b) *rbt* ⇒ *bool*›),
(***const-name*** ‹*rbt-sorted*›, *SOME* ***typ*** ‹('a :: *linorder*, 'b) *rbt* ⇒ *bool*›),
(***const-name*** ‹*rbt-lookup*›, *SOME* ***typ*** ‹('a :: *linorder*, 'b) *rbt* ⇒ 'a ⇀ 'b›),
(***const-name*** ‹*is-rbt*›, *SOME* ***typ*** ‹('a :: *linorder*, 'b) *rbt* ⇒ *bool*›),
(***const-name*** ‹*rbt-ins*›, *SOME* ***typ*** ‹('a::*linorder* ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-insert-with-key*›, *SOME* ***typ*** ‹('a::*linorder* ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ 'a ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-insert-with*›, *SOME* ***typ*** ‹('b ⇒ 'b ⇒ 'b) ⇒ ('a :: *linorder*) ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-insert*›, *SOME* ***typ*** ‹('a :: *linorder*) ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-del-from-left*›, *SOME* ***typ*** ‹('a::*linorder*) ⇒ ('a,'b) *rbt* ⇒ 'a ⇒ 'b ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-del-from-right*›, *SOME* ***typ*** ‹('a::*linorder*) ⇒ ('a,'b) *rbt* ⇒

'a ⇒ 'b ⇒ ('a,'b) rbt ⇒ ('a,'b) rbt›),
(***const-name*** ‹*rbt-del*›, *SOME* ***typ*** ‹('a::*linorder*) ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-delete*›, *SOME* ***typ*** ‹('a::*linorder*) ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-union-with-key*›, *SOME* ***typ*** ‹('a::*linorder* ⇒ 'b ⇒ 'b ⇒ 'b) ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-union-with*›, *SOME* ***typ*** ‹('b ⇒ 'b ⇒ 'b) ⇒ ('a::*linorder*,'b) *rbt* ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-union*›, *SOME* ***typ*** ‹('a::*linorder*,'b) *rbt* ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-map-entry*›, *SOME* ***typ*** ‹'a::*linorder* ⇒ ('b ⇒ 'b) ⇒ ('a,'b) *rbt* ⇒ ('a,'b) *rbt*›),
(***const-name*** ‹*rbt-bulkload*›, *SOME* ***typ*** ‹('a × 'b) *list* ⇒ ('a::*linorder*,'b) *rbt*›)]
›

hide-const (**open**) *MR MB R B Empty entries keys fold gen-keys gen-entries*

end

131 Abstract type of RBT trees

theory *RBT*
imports *Main RBT-Impl*
begin

131.1 Type definition

typedef (**overloaded**) ('a, 'b) *rbt* = {*t* :: ('a::*linorder*, 'b) *RBT-Impl.rbt*. *is-rbt t*}
morphisms *impl-of RBT*
proof −
have *RBT-Impl.Empty* ∈ *?rbt* **by** *simp*
then show *?thesis* **..**
qed

lemma *rbt-eq-iff*:
t1 = *t2* ⟷ *impl-of t1* = *impl-of t2*
by (*simp add*: *impl-of-inject*)

lemma *rbt-eqI*:
impl-of t1 = *impl-of t2* ⟹ *t1* = *t2*
by (*simp add*: *rbt-eq-iff*)

lemma *is-rbt-impl-of* [*simp*, *intro*]:
is-rbt (*impl-of t*)
using *impl-of* [*of t*] **by** *simp*

lemma *RBT-impl-of* [*simp*, *code abstype*]:
RBT (*impl-of t*) = *t*

by (*simp add*: *impl-of-inverse*)

131.2 Primitive operations

setup-lifting *type-definition-rbt*

lift-definition *lookup* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a* ⇀ '*b* **is** *rbt-lookup* **.**

lift-definition *empty* :: ('*a*::*linorder*, '*b*) *rbt* **is** *RBT-Impl.Empty*
by (*simp add*: *empty-def*)

lift-definition *insert* :: '*a*::*linorder* ⇒ '*b* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-insert*

by *simp*

lift-definition *delete* :: '*a*::*linorder* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-delete*
by *simp*

lift-definition *entries* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a* × '*b*) *list* **is** *RBT-Impl.entries*
.

lift-definition *keys* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a list* **is** *RBT-Impl.keys* **.**

lift-definition *bulkload* :: ('*a*::*linorder* × '*b*) *list* ⇒ ('*a*, '*b*) *rbt* **is** *rbt-bulkload* **..**

lift-definition *map-entry* :: '*a* ⇒ ('*b* ⇒ '*b*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
is *rbt-map-entry*
by *simp*

lift-definition *map* :: ('*a* ⇒ '*b* ⇒ '*c*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*c*) *rbt* **is**
RBT-Impl.map
by *simp*

lift-definition *fold* :: ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ '*c* ⇒ '*c* **is**
RBT-Impl.fold **.**

lift-definition *union* :: ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* **is**
rbt-union
by (*simp add*: *rbt-union-is-rbt*)

lift-definition *foldi* :: ('*c* ⇒ *bool*) ⇒ ('*a* ⇒ '*b* ⇒ '*c* ⇒ '*c*) ⇒ ('*a* :: *linorder*, '*b*)
rbt ⇒ '*c* ⇒ '*c*
is *RBT-Impl.foldi* **.**

lift-definition *combine-with-key* :: ('*a* ⇒ '*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒
('*a*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt*
is *RBT-Impl.rbt-union-with-key* **by** (*rule is-rbt-rbt-unionwk*)

lift-definition *combine* :: ('*b* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ ('*a*, '*b*) *rbt* ⇒

(*'a*, *'b*) *rbt*
is *RBT-Impl.rbt-union-with* **by** (*rule rbt-unionw-is-rbt*)

131.3 Derived operations

definition *is-empty* :: (*'a*::*linorder*, *'b*) *rbt* ⇒ *bool* **where**
[*code*]: *is-empty t* = (*case impl-of t of RBT-Impl.Empty* ⇒ *True* | - ⇒ *False*)

definition *filter* :: (*'a* ⇒ *'b* ⇒ *bool*) ⇒ (*'a*::*linorder*, *'b*) *rbt* ⇒ (*'a*, *'b*) *rbt* **where**
[*code*]: *filter P t* = *fold* (λ*k v t*. *if P k v then insert k v t else t*) *t empty*

131.4 Abstract lookup properties

lemma *lookup-RBT*:
is-rbt t ⟹ *lookup* (*RBT t*) = *rbt-lookup t*
by (*simp add*: *lookup-def RBT-inverse*)

lemma *lookup-impl-of*:
rbt-lookup (*impl-of t*) = *lookup t*
by *transfer* (*rule refl*)

lemma *entries-impl-of*:
RBT-Impl.entries (*impl-of t*) = *entries t*
by *transfer* (*rule refl*)

lemma *keys-impl-of*:
RBT-Impl.keys (*impl-of t*) = *keys t*
by *transfer* (*rule refl*)

lemma *lookup-keys*:
dom (*lookup t*) = *set* (*keys t*)
by *transfer* (*simp add*: *rbt-lookup-keys*)

lemma *lookup-empty* [*simp*]:
lookup empty = *Map.empty*
by (*simp add*: *empty-def lookup-RBT fun-eq-iff*)

lemma *lookup-insert* [*simp*]:
lookup (*insert k v t*) = (*lookup t*)(*k* ↦ *v*)
by *transfer* (*rule rbt-lookup-rbt-insert*)

lemma *lookup-delete* [*simp*]:
lookup (*delete k t*) = (*lookup t*)(*k* := *None*)
by *transfer* (*simp add*: *rbt-lookup-rbt-delete restrict-complement-singleton-eq*)

lemma *map-of-entries* [*simp*]:
map-of (*entries t*) = *lookup t*
by *transfer* (*simp add*: *map-of-entries*)

lemma *entries-lookup*:
entries t1 = entries t2 ⟷ lookup t1 = lookup t2
by *transfer* (*simp add*: *entries-rbt-lookup*)

lemma *lookup-bulkload* [*simp*]:
lookup (*bulkload xs*) = *map-of xs*
by *transfer* (*rule rbt-lookup-rbt-bulkload*)

lemma *lookup-map-entry* [*simp*]:
lookup (*map-entry k f t*) = (*lookup t*)(*k* := *map-option f* (*lookup t k*))
by *transfer* (*rule rbt-lookup-rbt-map-entry*)

lemma *lookup-map* [*simp*]:
lookup (*map f t*) *k* = *map-option* (*f k*) (*lookup t k*)
by *transfer* (*rule rbt-lookup-map*)

lemma *lookup-combine-with-key* [*simp*]:
lookup (*combine-with-key f t1 t2*) *k* = *combine-options* (*f k*) (*lookup t1 k*) (*lookup t2 k*)
by *transfer* (*simp-all add*: *combine-options-def rbt-lookup-rbt-unionwk*)

lemma *combine-altdef*: *combine f t1 t2* = *combine-with-key* (λ-. *f*) *t1 t2*
by *transfer* (*simp add*: *rbt-union-with-def*)

lemma *lookup-combine* [*simp*]:
lookup (*combine f t1 t2*) *k* = *combine-options f* (*lookup t1 k*) (*lookup t2 k*)
by (*simp add*: *combine-altdef*)

lemma *fold-fold*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
by *transfer* (*rule RBT-Impl.fold-def*)

lemma *impl-of-empty*:
impl-of empty = *RBT-Impl.Empty*
by *transfer* (*rule refl*)

lemma *is-empty-empty* [*simp*]:
is-empty t ⟷ t = *empty*
unfolding *is-empty-def* **by** *transfer* (*simp split*: *rbt.split*)

lemma *RBT-lookup-empty* [*simp*]:
rbt-lookup t = *Map.empty ⟷ t* = *RBT-Impl.Empty*
by (*cases t*) (*auto simp add*: *fun-eq-iff*)

lemma *lookup-empty-empty* [*simp*]:
lookup t = *Map.empty ⟷ t* = *empty*
by *transfer* (*rule RBT-lookup-empty*)

lemma *keys-empty-eq* [*simp*]:

‹*keys empty* = []›
by *transfer simp*

lemma *sorted-keys* [*iff*]:
sorted (*keys t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def rbt-sorted-entries*)

lemma *distinct-keys* [*iff*]:
distinct (*keys t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def distinct-entries*)

lemma *finite-dom-lookup* [*simp*, *intro*!]: *finite* (*dom* (*lookup t*))
by *transfer simp*

lemma *lookup-union*: *lookup* (*union s t*) = *lookup s* ++ *lookup t*
by *transfer* (*simp add*: *rbt-lookup-rbt-union*)

lemma *lookup-in-tree*: (*lookup t k* = *Some v*) = ((*k*, *v*) ∈ *set* (*entries t*))
by *transfer* (*simp add*: *rbt-lookup-in-tree*)

lemma *keys-entries*: (*k* ∈ *set* (*keys t*)) = (∃ *v*. (*k*, *v*) ∈ *set* (*entries t*))
by *transfer* (*simp add*: *keys-entries*)

lemma *fold-def-alt*:
fold f t = *List.fold* (*case-prod f*) (*entries t*)
by *transfer* (*auto simp*: *RBT-Impl.fold-def*)

lemma *distinct-entries*: *distinct* (*List.map fst* (*entries t*))
by *transfer* (*simp add*: *distinct-entries*)

lemma *sorted-entries*: *sorted* (*List.map fst* (*entries t*))
by (*transfer*) (*simp add*: *rbt-sorted-entries*)

lemma *non-empty-keys*: *t* ≠ *empty* ⟹ *keys t* ≠ []
by *transfer* (*simp add*: *non-empty-rbt-keys*)

lemma *keys-def-alt*:
keys t = *List.map fst* (*entries t*)
by *transfer* (*simp add*: *RBT-Impl.keys-def*)

context
begin

private lemma *lookup-filter-aux*:
assumes *distinct* (*List.map fst xs*)
shows *lookup* (*List.fold* (λ(*k*, *v*) *t*. *if P k v then insert k v t else t*) *xs t*) *k* =
(*case map-of xs k of*
None ⇒ *lookup t k*
| *Some v* ⇒ *if P k v then Some v else lookup t k*)

using *assms* **by** (*induction xs arbitrary*: *t*) (*force split*: *option.splits*)+

lemma *lookup-filter*:
lookup (*filter P t*) *k* =
(*case lookup t k of None ⇒ None | Some v ⇒ if P k v then Some v else None*)
unfolding *filter-def* **using** *lookup-filter-aux*[*of entries t P empty k*]
by (*simp add*: *fold-fold distinct-entries split*: *option.splits*)

end

131.5 Quickcheck generators

quickcheck-generator *rbt predicate*: *is-rbt constructors*: *empty*, *insert*

131.6 Hide implementation details

lifting-update *rbt.lifting*
lifting-forget *rbt.lifting*

hide-const (**open**) *impl-of empty lookup keys entries bulkload delete map fold union insert map-entry foldi*
is-empty filter
hide-fact (**open**) *empty-def lookup-def keys-def entries-def bulkload-def delete-def map-def fold-def*
union-def insert-def map-entry-def foldi-def is-empty-def filter-def

end

132 Implementation of mappings with Red-Black Trees

This theory defines abstract red-black trees as an efficient representation of finite maps, backed by the implementation in *HOL−Library.RBT-Impl*.

132.1 Data type and invariant

The type (′*k*, ′*v*) *RBT-Impl.rbt* denotes red-black trees with keys of type ′*k* and values of type ′*v*. To function properly, the key type musorted belong to the *linorder* class.

A value *t* of this type is a valid red-black tree if it satisfies the invariant *is-rbt t*. The abstract type (′*k*, ′*v*) *RBT.rbt* always obeys this invariant, and for this reason you should only use this in our application. Going back to (′*k*, ′*v*) *RBT-Impl.rbt* may be necessary in proofs if not yet proven properties about the operations must be established.

The interpretation function *RBT.lookup* returns the partial map represented by a red-black tree:

RBT.lookup

This function should be used for reasoning about the semantics of the RBT operations. Furthermore, it implements the lookup functionality for the data structure: It is executable and the lookup is performed in $O(\log n)$.

132.2 Operations

Currently, the following operations are supported:

RBT.empty

Returns the empty tree. $O(1)$

RBT.insert

Updates the map at a given position. $O(\log n)$

RBT.delete

Deletes a map entry at a given position. $O(\log n)$

RBT.entries

Return a corresponding key-value list for a tree.

RBT.bulkload

Builds a tree from a key-value list.

RBT.map-entry

Maps a single entry in a tree.

RBT.map

Maps all values in a tree. $O(n)$

RBT.fold

Folds over all entries in a tree. $O(n)$

132.3 Invariant preservation

is-rbt rbt.Empty (*Empty-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-insert ?k ?v ?t*) (*rbt-insert-is-rbt*)
is-rbt ?t $\Longrightarrow$ *is-rbt* (*rbt-delete ?k ?t*) (*delete-is-rbt*)
is-rbt (*rbt-bulkload ?xs*) (*bulkload-is-rbt*)
is-rbt (*rbt-map-entry ?k ?f ?t*) = *is-rbt ?t* (*map-entry-is-rbt*)
is-rbt (*RBT-Impl.map ?f ?t*) = *is-rbt ?t* (*map-is-rbt*)
⟦*is-rbt ?lt*; *is-rbt ?rt*⟧ $\Longrightarrow$ *is-rbt* (*rbt-union ?lt ?rt*) (*union-is-rbt*)

132.4 Map Semantics

lookup-empty

Mapping.lookup Mapping.empty ?k = None

lookup-insert

RBT.lookup (RBT.insert ?k ?v ?t) = (RBT.lookup ?t)(?k ↦ ?v)

lookup-delete

Mapping.lookup (Mapping.delete ?k ?m) ?k = None

lookup-bulkload

RBT.lookup (RBT.bulkload ?xs) = map-of ?xs

lookup-map

RBT.lookup (RBT.map ?f ?t) ?k = map-option (?f ?k) (RBT.lookup ?t ?k)

end

133 Implementation of sets using RBT trees

theory *RBT-Set*
imports *RBT Product-Lexorder*
begin

134 Definition of code datatype constructors

definition *Set* :: (*'a*::*linorder*, *unit*) *rbt* ⇒ *'a set*
  where *Set t* = {*x* . *RBT.lookup t x* = *Some* ()}

definition *Coset* :: (*'a*::*linorder*, *unit*) *rbt* ⇒ *'a set*
  where [*simp*]: *Coset t* = − *Set t*

135 Lemmas

135.1 Auxiliary lemmas

lemma [*simp*]: *x* ≠ *Some* () ⟷ *x* = *None*
by (*auto simp*: *not-Some-eq*[*THEN iffD1*])

lemma *Set-set-keys*: *Set x* = *dom* (*RBT.lookup x*)
by (*auto simp*: *Set-def*)

lemma *finite-Set* [*simp*, *intro*!]: *finite* (*Set x*)
by (*simp add*: *Set-set-keys*)

lemma *set-keys*: *Set t* = *set*(*RBT.keys t*)
by (*simp add*: *Set-set-keys lookup-keys*)

135.2 fold and filter

lemma *finite-fold-rbt-fold-eq*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*set* (*RBT.entries t*)) = *RBT.fold* (*curry f*) *t A*
proof −
interpret *comp-fun-commute*: *comp-fun-commute f*
by (*fact assms*)
have ∗: *remdups* (*RBT.entries t*) = *RBT.entries t*
using *distinct-entries distinct-map* **by** (*auto intro*: *distinct-remdups-id*)
show *?thesis* **using** *assms* **by** (*auto simp*: *fold-def-alt comp-fun-commute.fold-set-fold-remdups* ∗)
qed

definition *fold-keys* :: ('*a* :: *linorder* ⇒ '*b* ⇒ '*b*) ⇒ ('*a*, -) *rbt* ⇒ '*b* ⇒ '*b*
where [*code-unfold*]:*fold-keys f t A* = *RBT.fold* (λ*k* - *t*. *f k t*) *t A*

lemma *fold-keys-def-alt*:
fold-keys f t s = *List.fold f* (*RBT.keys t*) *s*
by (*auto simp*: *fold-map o-def split-def fold-def-alt keys-def-alt fold-keys-def*)

lemma *finite-fold-fold-keys*:
assumes *comp-fun-commute f*
shows *Finite-Set.fold f A* (*Set t*) = *fold-keys f t A*
using *assms*
proof −
interpret *comp-fun-commute f* **by** *fact*
have *set* (*RBT.keys t*) = *fst* ' (*set* (*RBT.entries t*)) **by** (*auto simp*: *fst-eq-Domain keys-entries*)
moreover have *inj-on fst* (*set* (*RBT.entries t*)) **using** *distinct-entries distinct-map* **by** *auto*
ultimately show *?thesis*
by (*auto simp add*: *set-keys fold-keys-def curry-def fold-image finite-fold-rbt-fold-eq*

comp-comp-fun-commute)
qed

definition *rbt-filter* :: ('*a* :: *linorder* ⇒ *bool*) ⇒ ('*a*, '*b*) *rbt* ⇒ '*a set* **where**
rbt-filter P t = *RBT.fold* (λ*k* - *A*'. *if P k then Set.insert k A*' *else A*') *t* {}

lemma *Set-filter-rbt-filter*:
Set.filter P (*Set t*) = *rbt-filter P t*
by (*subst Set-filter-fold*)

(*simp-all add*: *fold-keys-def rbt-filter-def finite-fold-fold-keys* [*OF comp-fun-commute-filter-fold*])

135.3 foldi and Ball

lemma *Ball-False*: *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t False* = *False*
by (*induction t*) *auto*

lemma *rbt-foldi-fold-conj*:
RBT-Impl.foldi (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t val*
proof (*induction t arbitrary*: *val*)
case (*Branch c t1*) **then show** *?case*
by (*cases RBT-Impl.fold* (λ*k v s*. *s* ∧ *P k*) *t1 True*) (*simp-all add*: *Ball-False*)
qed *simp*

lemma *foldi-fold-conj*: *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t val* = *fold-keys* (λ*k s*. *s* ∧ *P k*) *t val*
unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer* (*rule rbt-foldi-fold-conj*)

135.4 foldi and Bex

lemma *Bex-True*: *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t True* = *True*
by (*induction t*) *auto*

lemma *rbt-foldi-fold-disj*:
RBT-Impl.foldi (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t val* = *RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t val*
proof (*induction t arbitrary*: *val*)
case (*Branch c t1*) **then show** *?case*
by (*cases RBT-Impl.fold* (λ*k v s*. *s* ∨ *P k*) *t1 False*) (*simp-all add*: *Bex-True*)
qed *simp*

lemma *foldi-fold-disj*: *RBT.foldi* (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t val* = *fold-keys* (λ*k s*. *s* ∨ *P k*) *t val*
unfolding *fold-keys-def* **including** *rbt.lifting* **by** *transfer* (*rule rbt-foldi-fold-disj*)

135.5 folding over non empty trees and selecting the minimal and maximal element

135.5.1 concrete

The concrete part is here because it's probably not general enough to be moved to *RBT-Impl*

definition *rbt-fold1-keys* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a*::*linorder*, '*b*) *RBT-Impl.rbt* ⇒ '*a*
where *rbt-fold1-keys f t* = *List.fold f* (*tl*(*RBT-Impl.keys t*)) (*hd*(*RBT-Impl.keys t*))

minimum **definition** *rbt-min* :: ('*a*::*linorder*, *unit*) *RBT-Impl.rbt* ⇒ '*a*

where *rbt-min t = rbt-fold1-keys min t*

lemma *key-le-right*: *rbt-sorted (Branch c lt k v rt) ⟹ (⋀x. x ∈set (RBT-Impl.keys rt) ⟹ k ≤ x)*
by (*auto simp*: *rbt-greater-prop less-imp-le*)

lemma *left-le-key*: *rbt-sorted (Branch c lt k v rt) ⟹ (⋀x. x ∈set (RBT-Impl.keys lt) ⟹ x ≤ k)*
by (*auto simp*: *rbt-less-prop less-imp-le*)

lemma *fold-min-triv*:
fixes *k* :: - :: *linorder*
shows *(∀ x∈set xs. k ≤ x) ⟹ List.fold min xs k = k*
by (*induct xs*) (*auto simp add*: *min-def*)

lemma *rbt-min-simps*:
is-rbt (Branch c RBT-Impl.Empty k v rt) ⟹ rbt-min (Branch c RBT-Impl.Empty k v rt) = k
by (*auto intro*: *fold-min-triv dest*: *key-le-right is-rbt-rbt-sorted simp*: *rbt-fold1-keys-def rbt-min-def*)

fun *rbt-min-opt* **where**
rbt-min-opt (Branch c RBT-Impl.Empty k v rt) = k |
rbt-min-opt (Branch c (Branch lc llc lk lv lrt) k v rt) = rbt-min-opt (Branch lc llc lk lv lrt)

lemma *rbt-min-opt-Branch*:
t1 ≠ rbt.Empty ⟹ rbt-min-opt (Branch c t1 k () t2) = rbt-min-opt t1
by (*cases t1*) *auto*

lemma *rbt-min-opt-induct* [*case-names empty left-empty left-non-empty*]:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes *⋀color t1 a b t2. P t1 ⟹ P t2 ⟹ t1 = rbt.Empty ⟹ P (Branch color t1 a b t2)*
assumes *⋀color t1 a b t2. P t1 ⟹ P t2 ⟹ t1 ≠ rbt.Empty ⟹ P (Branch color t1 a b t2)*
shows *P t*
using *assms*
proof (*induct t*)
case *Empty*
then show *?case* **by** *simp*
next
case (*Branch x1 t1 x3 x4 t2*)
then show *?case* **by** (*cases t1 = rbt.Empty*) *simp-all*
qed

lemma *rbt-min-opt-in-set*:
fixes *t* :: (*'a* :: *linorder*, *unit*) *RBT-Impl.rbt*

assumes $t \neq$ *rbt.Empty*
shows *rbt-min-opt t* $\in$ *set* (*RBT-Impl.keys t*)
using *assms* **by** (*induction t rule*: *rbt-min-opt.induct*) (*auto*)

lemma *rbt-min-opt-is-min*:
fixes t :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes $t \neq$ *rbt.Empty*
shows $\bigwedge y.\ y \in$ *set* (*RBT-Impl.keys t*) $\Longrightarrow y \geq$ *rbt-min-opt t*
using *assms*
proof (*induction t rule*: *rbt-min-opt-induct*)
case *empty*
then show *?case* **by** *simp*
next
case *left-empty*
then show *?case* **by** (*auto intro*: *key-le-right simp del*: *rbt-sorted.simps*)
next
case (*left-non-empty c t1 k v t2 y*)
then consider $y = k \mid y \in$ *set* (*RBT-Impl.keys t1*) $\mid y \in$ *set* (*RBT-Impl.keys t2*)
by *auto*
then show *?case*
proof *cases*
case *1*
with *left-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-min-opt-Branch intro*: *left-le-key rbt-min-opt-in-set*)
next
case *2*
with *left-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-min-opt-Branch*)
next
case *y*: *3*
have *rbt-min-opt t1* $\leq k$
using *left-non-empty* **by** (*simp add*: *left-le-key rbt-min-opt-in-set*)
moreover have $k \leq y$
using *left-non-empty y* **by** (*simp add*: *key-le-right*)
ultimately show *?thesis*
using *left-non-empty y* **by** (*simp add*: *rbt-min-opt-Branch*)
qed
qed

lemma *rbt-min-eq-rbt-min-opt*:
assumes $t \neq$ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-min t* = *rbt-min-opt t*
proof −
from *assms* **have** *hd* (*RBT-Impl.keys t*) # *tl* (*RBT-Impl.keys t*) = *RBT-Impl.keys t* **by** (*cases t*) *simp-all*
with *assms* **show** *?thesis*

by (*simp add*: *rbt-min-def rbt-fold1-keys-def rbt-min-opt-is-min Min.set-eq-fold* [*symmetric*] *Min-eqI rbt-min-opt-in-set*)
qed

maximum **definition** *rbt-max* :: ('*a*::*linorder*, *unit*) *RBT-Impl.rbt* ⇒ '*a*
where *rbt-max t* = *rbt-fold1-keys max t*

lemma *fold-max-triv*:
fixes *k* :: - :: *linorder*
shows (∀ *x*∈*set xs*. *x* ≤ *k*) ⟹ *List.fold max xs k* = *k*
by (*induct xs*) (*auto simp add*: *max-def*)

lemma *fold-max-rev-eq*:
fixes *xs* :: ('*a* :: *linorder*) *list*
assumes *xs* ≠ []
shows *List.fold max* (*tl xs*) (*hd xs*) = *List.fold max* (*tl* (*rev xs*)) (*hd* (*rev xs*))
using *assms* **by** (*simp add*: *Max.set-eq-fold* [*symmetric*])

lemma *rbt-max-simps*:
assumes *is-rbt* (*Branch c lt k v RBT-Impl.Empty*)
shows *rbt-max* (*Branch c lt k v RBT-Impl.Empty*) = *k*
proof −
have *List.fold max* (*tl* (*rev*(*RBT-Impl.keys lt* @ [*k*]))) (*hd* (*rev*(*RBT-Impl.keys lt* @ [*k*]))) = *k*
using *assms* **by** (*auto intro*!: *fold-max-triv dest*!: *left-le-key is-rbt-rbt-sorted*)
then show *?thesis* **by** (*auto simp add*: *rbt-max-def rbt-fold1-keys-def fold-max-rev-eq*)
qed

fun *rbt-max-opt* **where**
rbt-max-opt (*Branch c lt k v RBT-Impl.Empty*) = *k* |
rbt-max-opt (*Branch c lt k v* (*Branch rc rlc rk rv rrt*)) = *rbt-max-opt* (*Branch rc rlc rk rv rrt*)

lemma *rbt-max-opt-Branch*:
t2 ≠ *rbt.Empty* ⟹ *rbt-max-opt* (*Branch c t1 k* () *t2*) = *rbt-max-opt t2*
by (*cases t2*) *auto*

lemma *rbt-max-opt-induct* [*case-names empty right-empty right-non-empty*]:
fixes *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *P rbt.Empty*
assumes ⋀*color t1 a b t2*. *P t1* ⟹ *P t2* ⟹ *t2* = *rbt.Empty* ⟹ *P* (*Branch color t1 a b t2*)
assumes ⋀*color t1 a b t2*. *P t1* ⟹ *P t2* ⟹ *t2* ≠ *rbt.Empty* ⟹ *P* (*Branch color t1 a b t2*)
shows *P t*
using *assms*
proof (*induct t*)
case *Empty*
then show *?case* **by** *simp*

next
case (*Branch x1 t1 x3 x4 t2*)
then show *?case* **by** (*cases t2 = rbt.Empty*) *simp-all*
qed

lemma *rbt-max-opt-in-set*:
fixes *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *t* ≠ *rbt.Empty*
shows *rbt-max-opt t* ∈ *set* (*RBT-Impl.keys t*)
using *assms* **by** (*induction t rule*: *rbt-max-opt.induct*) (*auto*)

lemma *rbt-max-opt-is-max*:
fixes *t* :: ('*a* :: *linorder*, *unit*) *RBT-Impl.rbt*
assumes *rbt-sorted t*
assumes *t* ≠ *rbt.Empty*
shows ⋀*y*. *y* ∈ *set* (*RBT-Impl.keys t*) ⟹ *y* ≤ *rbt-max-opt t*
using *assms*
proof (*induction t rule*: *rbt-max-opt-induct*)
case *empty*
then show *?case* **by** *simp*
next
case *right-empty*
then show *?case* **by** (*auto intro*: *left-le-key simp del*: *rbt-sorted.simps*)
next
case (*right-non-empty c t1 k v t2 y*)
then consider *y* = *k* | *y* ∈ *set* (*RBT-Impl.keys t2*) | *y* ∈ *set* (*RBT-Impl.keys t1*)
by *auto*
then show *?case*
proof *cases*
case *1*
with *right-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-max-opt-Branch intro*: *key-le-right rbt-max-opt-in-set*)
next
case *2*
with *right-non-empty* **show** *?thesis*
by (*auto simp add*: *rbt-max-opt-Branch*)
next
case *y*: *3*
have *rbt-max-opt t2* ≥ *k*
using *right-non-empty* **by** (*simp add*: *key-le-right rbt-max-opt-in-set*)
moreover have *y* ≤ *k*
using *right-non-empty y* **by** (*simp add*: *left-le-key*)
ultimately show *?thesis*
using *right-non-empty* **by** (*simp add*: *rbt-max-opt-Branch*)
qed
qed

lemma *rbt-max-eq-rbt-max-opt*:

assumes $t \neq$ *RBT-Impl.Empty*
assumes *is-rbt t*
shows *rbt-max t* = *rbt-max-opt t*
proof −
from *assms* **have** *hd* (*RBT-Impl.keys t*) # *tl* (*RBT-Impl.keys t*) = *RBT-Impl.keys t* **by** (*cases t*) *simp-all*
with *assms* **show** *?thesis*
by (*simp add*: *rbt-max-def rbt-fold1-keys-def rbt-max-opt-is-max Max.set-eq-fold* [*symmetric*] *Max-eqI rbt-max-opt-in-set*)
qed

135.5.2 abstract

context includes *rbt.lifting* **begin**
lift-definition *fold1-keys* :: ('*a* ⇒ '*a* ⇒ '*a*) ⇒ ('*a*::*linorder*, '*b*) *rbt* ⇒ '*a*
is *rbt-fold1-keys* **.**

lemma *fold1-keys-def-alt*:
fold1-keys f t = *List.fold f* (*tl* (*RBT.keys t*)) (*hd* (*RBT.keys t*))
by *transfer* (*simp add*: *rbt-fold1-keys-def*)

lemma *finite-fold1-fold1-keys*:
assumes *semilattice f*
assumes ¬ *RBT.is-empty t*
shows *semilattice-set.F f* (*Set t*) = *fold1-keys f t*
proof −
from ‹*semilattice f*› **interpret** *semilattice-set f* **by** (*rule semilattice-set.intro*)
show *?thesis* **using** *assms*
by (*auto simp*: *fold1-keys-def-alt set-keys fold-def-alt non-empty-keys set-eq-fold* [*symmetric*])
qed

minimum **lift-definition** *r-min* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min* **.**

lift-definition *r-min-opt* :: ('*a* :: *linorder*, *unit*) *rbt* ⇒ '*a* **is** *rbt-min-opt* **.**

lemma *r-min-alt-def*: *r-min t* = *fold1-keys min t*
by *transfer* (*simp add*: *rbt-min-def*)

lemma *r-min-eq-r-min-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-min t* = *r-min-opt t*
using *assms* **unfolding** *is-empty-empty* **by** *transfer* (*auto intro*: *rbt-min-eq-rbt-min-opt*)

lemma *fold-keys-min-top-eq*:
fixes *t* :: ('*a*::{*linorder*,*bounded-lattice-top*}, *unit*) *rbt*
assumes ¬ (*RBT.is-empty t*)
shows *fold-keys min t top* = *fold1-keys min t*
proof −

have $*$: $\bigwedge t$. *RBT-Impl.keys* $t \neq [] \Longrightarrow$ *List.fold min* (*RBT-Impl.keys* t) *top* =
List.fold min (*hd* (*RBT-Impl.keys* t) # *tl* (*RBT-Impl.keys* t)) *top*
by (*simp add*: *hd-Cons-tl*[*symmetric*])
have $**$: *List.fold min* (x # *xs*) *top* = *List.fold min xs x* **for** x :: $'a$ **and** *xs*
by (*simp add*: *inf-min*[*symmetric*])
show *?thesis*
using *assms*
unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
apply *transfer*
apply (*case-tac t*)
apply *simp*
apply (*subst* $*$)
apply *simp*
apply (*subst* $**$)
apply *simp*
done
qed

maximum **lift-definition** *r-max* :: ($'a$ :: *linorder*, *unit*) *rbt* ⇒ $'a$ **is** *rbt-max* **.**

lift-definition *r-max-opt* :: ($'a$ :: *linorder*, *unit*) *rbt* ⇒ $'a$ **is** *rbt-max-opt* **.**

lemma *r-max-alt-def*: *r-max t* = *fold1-keys max t*
by *transfer* (*simp add*: *rbt-max-def*)

lemma *r-max-eq-r-max-opt*:
assumes ¬ (*RBT.is-empty t*)
shows *r-max t* = *r-max-opt t*
using *assms* **unfolding** *is-empty-empty* **by** *transfer* (*auto intro*: *rbt-max-eq-rbt-max-opt*)

lemma *fold-keys-max-bot-eq*:
fixes *t* :: ($'a$::{*linorder*,*bounded-lattice-bot*}, *unit*) *rbt*
assumes ¬ (*RBT.is-empty t*)
shows *fold-keys max t bot* = *fold1-keys max t*
proof −
have $*$: $\bigwedge t$. *RBT-Impl.keys* $t \neq [] \Longrightarrow$ *List.fold max* (*RBT-Impl.keys* t) *bot* =
List.fold max (*hd*(*RBT-Impl.keys* t) # *tl*(*RBT-Impl.keys* t)) *bot*
by (*simp add*: *hd-Cons-tl*[*symmetric*])
have $**$: *List.fold max* (x # *xs*) *bot* = *List.fold max xs x* **for** x :: $'a$ **and** *xs*
by (*simp add*: *sup-max*[*symmetric*])
show *?thesis*
using *assms*
unfolding *fold-keys-def-alt fold1-keys-def-alt is-empty-empty*
apply *transfer*
apply (*case-tac t*)
apply *simp*
apply (*subst* $*$)
apply *simp*
apply (*subst* $**$)

apply *simp*
done
qed

end

136 Code equations

code-datatype *Set Coset*

declare *list.set*[*code*]

lemma *empty-Set* [*code*]:
Set.empty = Set RBT.empty
by (*auto simp*: *Set-def*)

lemma *UNIV-Coset* [*code*]:
UNIV = Coset RBT.empty
by (*auto simp*: *Set-def*)

lemma *is-empty-Set* [*code*]:
Set.is-empty (*Set t*) = *RBT.is-empty t*
using *non-empty-keys* [*of t*] **by** (*auto simp add*: *set-keys*)

lemma *compl-code* [*code*]:
– *Set xs = Coset xs*
– *Coset xs = Set xs*
by (*simp-all add*: *Set-def*)

lemma *member-code* [*code*]:
x ∈ (*Set t*) = (*RBT.lookup t x = Some* ())
x ∈ (*Coset t*) = (*RBT.lookup t x = None*)
by (*simp-all add*: *Set-def*)

lemma *insert-code* [*code*]:
Set.insert x (*Set t*) = *Set* (*RBT.insert x* () *t*)
Set.insert x (*Coset t*) = *Coset* (*RBT.delete x t*)
by (*auto simp*: *Set-def*)

lemma *remove-code* [*code*]:
Set.remove x (*Set t*) = *Set* (*RBT.delete x t*)
Set.remove x (*Coset t*) = *Coset* (*RBT.insert x* () *t*)
by (*auto simp*: *Set-def*)

lemma *inter-Set* [*code*]:
A ∩ *Set t = rbt-filter* (λ*k*. *k* ∈ *A*) *t*
by (*simp flip*: *Set-filter-rbt-filter add*: *inter-Set-filter*)

lemma *union-Set-Set* [*code*]:

Set t1 ∪ Set t2 = Set (RBT.union t1 t2)
by (*auto simp add*: *lookup-union map-add-Some-iff Set-def*)

lemma *union-Set* [*code*]:
Set t ∪ A = fold-keys Set.insert t A
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
from *finite-fold-fold-keys*[*OF comp-fun-commute-axioms*]
show *?thesis* **by** (*auto simp add*: *union-fold-insert*)
qed

lemma *minus-Set* [*code*]:
A − Set t = fold-keys Set.remove t A
proof −
interpret *comp-fun-idem Set.remove*
by (*fact comp-fun-idem-remove*)
from *finite-fold-fold-keys*[*OF comp-fun-commute-axioms*]
show *?thesis* **by** (*auto simp add*: *minus-fold-remove*)
qed

lemma *inter-Coset-Coset* [*code*]:
Coset t1 ∩ Coset t2 = Coset (RBT.union t1 t2)
by (*auto simp add*: *lookup-union map-add-Some-iff Set-def*)

lemma *inter-Coset* [*code*]:
A ∩ Coset t = fold-keys Set.remove t A
by (*simp add*: *Diff-eq* [*symmetric*] *minus-Set*)

lemma *union-Coset* [*code*]:
Coset t ∪ A = − rbt-filter (*λk. k ∉ A*) *t*
proof −
have ∗: *⋀A B.* (*−A ∪ B*) = *−*(*−B ∩ A*) **by** *blast*
show *?thesis* **by** (*simp del*: *boolean-algebra-class.compl-inf add*: ∗ *inter-Set*)
qed

lemma *minus-Coset* [*code*]:
A − Coset t = rbt-filter (*λk. k ∈ A*) *t*
by (*simp add*: *inter-Set*[*simplified Int-commute*])

lemma *filter-Set* [*code*]:
Set.filter P (*Set t*) = *rbt-filter P t*
by (*fact Set-filter-rbt-filter*)

lemma *image-Set* [*code*]:
image f (*Set t*) = *fold-keys* (*λk A. Set.insert* (*f k*) *A*) *t* {}
proof −
have *comp-fun-commute* (*λk. Set.insert* (*f k*))
by *standard auto*

then show *?thesis*
by (*auto simp add*: *image-fold-insert intro*!: *finite-fold-fold-keys*)
qed

lemma *Ball-Set* [*code*]:
Ball (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *P k*) *t True*
proof −
have *comp-fun-commute* (λ*k s*. *s* ∧ *P k*)
by *standard auto*
then show *?thesis*
by (*simp add*: *foldi-fold-conj*[*symmetric*] *Ball-fold finite-fold-fold-keys*)
qed

lemma *Bex-Set* [*code*]:
Bex (*Set t*) *P* ⟷ *RBT.foldi* (λ*s*. *s* = *False*) (λ*k v s*. *s* ∨ *P k*) *t False*
proof −
have *comp-fun-commute* (λ*k s*. *s* ∨ *P k*)
by *standard auto*
then show *?thesis*
by (*simp add*: *foldi-fold-disj*[*symmetric*] *Bex-fold finite-fold-fold-keys*)
qed

lemma *subset-code* [*code*]:
Set t ≤ *B* ⟷ (∀ *x*∈*Set t*. *x* ∈ *B*)
A ≤ *Coset t* ⟷ (∀ *y*∈*Set t*. *y* ∉ *A*)
by *auto*

lemma *subset-Coset-empty-Set-empty* [*code*]:
Coset t1 ≤ *Set t2* ⟷ (*case* (*RBT.impl-of t1*, *RBT.impl-of t2*) *of*
(*rbt.Empty*, *rbt.Empty*) ⇒ *False* |
(-, -) ⇒ *Code.abort* (*STR* ''*non-empty-trees*'') (λ-. *Coset t1* ≤ *Set t2*))
proof −
have ∗: ⋀*t*. *RBT.impl-of t* = *rbt.Empty* ⟹ *t* = *RBT rbt.Empty*
by (*subst*(*asm*) *RBT-inverse*[*symmetric*]) (*auto simp*: *impl-of-inject*)
have ∗∗: *eq-onp is-rbt rbt.Empty rbt.Empty* **unfolding** *eq-onp-def* **by** *simp*
show *?thesis*
by (*auto simp*: *Set-def lookup.abs-eq*[*OF* ∗∗] *dest*!: ∗ *split*: *rbt.split*)
qed

A frequent case – avoid intermediate sets

lemma [*code-unfold*]:
Set t1 ⊆ *Set t2* ⟷ *RBT.foldi* (λ*s*. *s* = *True*) (λ*k v s*. *s* ∧ *k* ∈ *Set t2*) *t1 True*
by (*simp add*: *subset-code Ball-Set*)

lemma *card-Set* [*code*]:
card (*Set t*) = *fold-keys* (λ- *n*. *n* + *1*) *t 0*
by (*auto simp add*: *card.eq-fold intro*: *finite-fold-fold-keys comp-fun-commute-const*)

lemma *sum-Set* [*code*]:

sum f (Set xs) = fold-keys (plus ∘ f) xs 0
proof −
have *comp-fun-commute* (λ*x*. (+) (*f x*))
by *standard* (*auto simp*: *ac-simps*)
then show *?thesis*
by (*auto simp add*: *sum.eq-fold finite-fold-fold-keys o-def*)
qed

lemma *prod-Set* [*code*]:
prod f (Set xs) = fold-keys (times ∘ f) xs 1
proof −
have *comp-fun-commute* (λ*x*. (∗) (*f x*))
by *standard* (*auto simp*: *ac-simps*)
then show *?thesis*
by (*auto simp add*: *prod.eq-fold finite-fold-fold-keys o-def*)
qed

lemma *the-elem-set* [*code*]:
fixes *t* :: ('*a* :: *linorder*, *unit*) *rbt*
shows *the-elem* (*Set t*) = (*case RBT.impl-of t of*
(*Branch RBT-Impl.B RBT-Impl.Empty x* () *RBT-Impl.Empty*) ⇒ *x*
| - ⇒ *Code.abort* (*STR ''not-a-singleton-tree''*) (λ-. *the-elem* (*Set t*)))
proof −
{
fix *x* :: '*a* :: *linorder*
let *?t* = *Branch RBT-Impl.B RBT-Impl.Empty x* () *RBT-Impl.Empty*
have ∗:*?t* ∈ {*t*. *is-rbt t*} **unfolding** *is-rbt-def* **by** *auto*
then have ∗∗:*eq-onp is-rbt ?t ?t* **unfolding** *eq-onp-def* **by** *auto*

have *RBT.impl-of t* = *?t* ⟹ *the-elem* (*Set t*) = *x*
by (*subst*(*asm*) *RBT-inverse*[*symmetric*, *OF* ∗])
(*auto simp*: *Set-def the-elem-def lookup.abs-eq*[*OF* ∗∗] *impl-of-inject*)
}
then show *?thesis*
by(*auto split*: *rbt.split unit.split color.split*)
qed

lemma *Pow-Set* [*code*]: *Pow* (*Set t*) = *fold-keys* (λ*x A*. *A* ∪ *Set.insert x* ' *A*) *t* {{}}
by (*simp add*: *Pow-fold finite-fold-fold-keys*[*OF comp-fun-commute-Pow-fold*])

lemma *product-Set* [*code*]:
Product-Type.product (*Set t1*) (*Set t2*) =
fold-keys (λ*x A*. *fold-keys* (λ*y*. *Set.insert* (*x*, *y*)) *t2 A*) *t1* {}
proof −
have ∗: *comp-fun-commute* (λ*y*. *Set.insert* (*x*, *y*)) **for** *x*
by *standard auto*
show *?thesis* **using** *finite-fold-fold-keys*[*OF comp-fun-commute-product-fold*, *of Set t2* {} *t1*]

by (*simp add*: *product-fold Product-Type.product-def finite-fold-fold-keys*[*OF* ∗])
qed

lemma *Id-on-Set* [*code*]: *Id-on* (*Set t*) = *fold-keys* (λ*x*. *Set.insert* (*x*, *x*)) *t* {}
proof −
have *comp-fun-commute* (λ*x*. *Set.insert* (*x*, *x*))
by *standard auto*
then show *?thesis*
by (*auto simp add*: *Id-on-fold intro*!: *finite-fold-fold-keys*)
qed

lemma *Image-Set* [*code*]:
(*Set t*) `` *S* = *fold-keys* (λ(*x*,*y*) *A*. *if x* ∈ *S then Set.insert y A else A*) *t* {}
by (*auto simp add*: *Image-fold finite-fold-fold-keys*[*OF comp-fun-commute-Image-fold*])

lemma *trancl-set-ntrancl* [*code*]:
trancl (*Set t*) = *ntrancl* (*card* (*Set t*) − *1*) (*Set t*)
by (*simp add*: *finite-trancl-ntranl*)

lemma *relcomp-Set*[*code*]:
(*Set t1*) *O* (*Set t2*) = *fold-keys*
(λ(*x*,*y*) *A*. *fold-keys* (λ(*w*,*z*) *A*′. *if y* = *w then Set.insert* (*x*,*z*) *A*′ *else A*′) *t2 A*)
t1 {}
proof −
interpret *comp-fun-idem Set.insert*
by (*fact comp-fun-idem-insert*)
have ∗: ⋀*x y*. *comp-fun-commute* (λ(*w*, *z*) *A*′. *if y* = *w then Set.insert* (*x*, *z*) *A*′
else A′)
by *standard* (*auto simp add*: *fun-eq-iff*)
show *?thesis*
using *finite-fold-fold-keys*[*OF comp-fun-commute-relcomp-fold*, *of Set t2* {} *t1*]
by (*simp add*: *relcomp-fold finite-fold-fold-keys*[*OF* ∗])
qed

lemma *wf-set*: *wf* (*Set t*) = *acyclic* (*Set t*)
by (*simp add*: *wf-iff-acyclic-if-finite*)

lemma *wf-code-set*[*code*]: *wf-code* (*Set t*) = *acyclic* (*Set t*)
unfolding *wf-code-def* **using** *wf-set* **.**

lemma *Min-fin-set-fold* [*code*]:
Min (*Set t*) =
(*if RBT.is-empty t*
then Code.abort (*STR* ′′*not-non-empty-tree*′′) (λ-. *Min* (*Set t*))
else r-min-opt t)
proof −
have ∗: *semilattice* (*min* :: ′*a* ⇒ ′*a* ⇒ ′*a*) **..**
with *finite-fold1-fold1-keys* [*OF* ∗, *folded Min-def*]
show *?thesis*

by (*simp add*: *r-min-alt-def r-min-eq-r-min-opt* [*symmetric*])
qed

lemma *Inf-fin-set-fold* [*code*]:
Inf-fin (*Set t*) = *Min* (*Set t*)
by (*simp add*: *inf-min Inf-fin-def Min-def*)

lemma *Inf-Set-fold* [*code*]:
Inf (*Set t*) = (*if RBT.is-empty t then top else r-min-opt t*)
for *t* :: ('*a* :: {*linorder*, *complete-lattice*}, *unit*) *rbt*
proof −
have *comp-fun-commute* (*min* :: '*a* ⇒ '*a* ⇒ '*a*)
by *standard* (*simp add*: *fun-eq-iff ac-simps*)
then have *t* ≠ *RBT.empty* ⟹ *Finite-Set.fold min top* (*Set t*) = *fold1-keys min t*
by (*simp add*: *finite-fold-fold-keys fold-keys-min-top-eq*)
then show *?thesis*
by (*auto simp add*: *Inf-fold-inf inf-min empty-Set*[*symmetric*]
r-min-eq-r-min-opt[*symmetric*] *r-min-alt-def*)
qed

lemma *Max-fin-set-fold* [*code*]:
Max (*Set t*) =
(*if RBT.is-empty t*
then Code.abort (*STR* ''*not-non-empty-tree*'') (λ-. *Max* (*Set t*))
else r-max-opt t)
proof −
have ∗: *semilattice* (*max* :: '*a* ⇒ '*a* ⇒ '*a*) **..**
with *finite-fold1-fold1-keys* [*OF* ∗, *folded Max-def*]
show *?thesis*
by (*simp add*: *r-max-alt-def r-max-eq-r-max-opt* [*symmetric*])
qed

lemma *Sup-fin-set-fold* [*code*]:
Sup-fin (*Set t*) = *Max* (*Set t*)
by (*simp add*: *sup-max Sup-fin-def Max-def*)

lemma *Sup-Set-fold* [*code*]:
Sup (*Set t*) = (*if RBT.is-empty t then bot else r-max-opt t*)
for *t* :: ('*a* :: {*linorder*, *complete-lattice*}, *unit*) *rbt*
proof −
have *comp-fun-commute* (*max* :: '*a* ⇒ '*a* ⇒ '*a*)
by *standard* (*simp add*: *fun-eq-iff ac-simps*)
then have *t* ≠ *RBT.empty* ⟹ *Finite-Set.fold max bot* (*Set t*) = *fold1-keys max t*
by (*simp add*: *finite-fold-fold-keys fold-keys-max-bot-eq*)
then show *?thesis*
by (*auto simp add*: *Sup-fold-sup sup-max empty-Set*[*symmetric*]
r-max-eq-r-max-opt[*symmetric*] *r-max-alt-def*)

qed

lemma [*code*]:
Gcd$_{fin}$ (*Set t*) = *fold-keys gcd t* (*0*::*'a*::{*semiring-gcd*, *linorder*})
proof −
have *comp-fun-commute* (*gcd* :: *'a* ⇒ -)
by *standard* (*simp add*: *fun-eq-iff ac-simps*)
with *finite-fold-fold-keys* [*of* - *0 t*]
have *Finite-Set.fold gcd 0* (*Set t*) = *fold-keys gcd t 0*
by *blast*
then show *?thesis*
by (*simp add*: *Gcd-fin.eq-fold*)
qed

lemma [*code*]:
Gcd (*Set t*) = *Gcd*$_{fin}$ (*Set t*) **for** *t* :: ‹(*'a*::{*semiring-Gcd*, *linorder*}, *unit*) *RBT.rbt*›
by *simp*

lemma [*code*]:
Gcd (*Set t*) = (*Gcd*$_{fin}$ (*Set t*) :: *nat*)
by *simp*

lemma [*code*]:
Gcd (*Set t*) = (*Gcd*$_{fin}$ (*Set t*) :: *int*)
by *simp*

lemma [*code*]:
Lcm$_{fin}$ (*Set t*) = *fold-keys lcm t* (*1*::*'a*::{*semiring-gcd*, *linorder*})
proof −
have *comp-fun-commute* (*lcm* :: *'a* ⇒ -)
by *standard* (*simp add*: *fun-eq-iff ac-simps*)
with *finite-fold-fold-keys* [*of* - *1 t*]
have *Finite-Set.fold lcm 1* (*Set t*) = *fold-keys lcm t 1*
by *blast*
then show *?thesis*
by (*simp add*: *Lcm-fin.eq-fold*)
qed

lemma [*code*]:
Lcm (*Set t*) = *Lcm*$_{fin}$ (*Set t*) **for** *t* :: ‹(*'a*::{*semiring-Gcd*, *linorder*}, *unit*) *RBT.rbt*›
by *simp*

lemma [*code*]:
Lcm (*Set t*) = (*Lcm*$_{fin}$ (*Set t*) :: *nat*)
by *simp*

lemma [*code*]:

Lcm (Set t) = (Lcm$_{fin}$ (Set t) :: int)
by *simp*

lemma *Least-code* [*code*]:
‹*Lattices-Big.Least (Set t) = (if RBT.is-empty t then Lattices-Big.Least-abort {} else Min (Set t))*›
apply (*auto simp add*: *Lattices-Big.Least-abort-def simp flip*: *empty-Set*)
apply (*subst Least-Min*)
using *is-empty-Set*
apply *auto*
done

lemma *Greatest-code* [*code*]:
‹*Lattices-Big.Greatest (Set t) = (if RBT.is-empty t then Lattices-Big.Greatest-abort {} else Max (Set t))*›
apply (*auto simp add*: *Lattices-Big.Greatest-abort-def simp flip*: *empty-Set*)
apply (*subst Greatest-Max*)
using *is-empty-Set*
apply *auto*
done

lemma *sorted-list-set* [*code*]: *sorted-list-of-set (Set t) = RBT.keys t*
by (*auto simp add*: *set-keys intro*: *sorted-distinct-set-unique*)

lemma [*code*]:
‹*Option.these A = the ' Set.filter (Not ∘ Option.is-none) A*›
by (*fact Option.these-eq*)

lemma [*code*]:
‹*Option.image-filter f A = Option.these (image f A)*›
by (*fact Option.image-filter-eq*)

lemma [*code*]:
‹*Set.can-select P A = is-singleton (Set.filter P A)*›
by (*fact Set.can-select-iff-is-singleton*)

declare [[*code drop*:
‹*Inf* :: - ⇒ *'a set*›
‹*Sup* :: - ⇒ *'a set*›
‹*Inf* :: - ⇒ *'a Predicate.pred*›
‹*Sup* :: - ⇒ *'a Predicate.pred*›
pred-of-set
Wellfounded.acc
]]

hide-const (**open**) *RBT-Set.Set RBT-Set.Coset*

end

theory *Time-Manual*
imports *HOL−Library.Time-Commands*
begin

137 Introduction

This manual describes the framework for the automatic definition of step-counting 'running-time' functions from HOL functions. The principles of the translation are described in Section 1.5, Running Time, of the ACM book Functional Data Structures and Algorithms. A Proof Assistant Approach. https://fdsa-book.net To load the framework import *HOL−Library.Time-Commands* The framework was implemented by Jonas Stahl.

As a first simple example consider *len*, which we define here returning an *int* (to distinguish it from the time functions returning *nat*):

fun *len* :: *'a list* ⇒ *int* **where**
len [] = *0* |
len (*x*#*xs*) = *1* + *len xs*

time-fun *len*

Command *time-fun* defines a new function *T-len* of type *'a list* ⇒ *nat*, the time function for *len* that counts the number of computation steps. The definition is printed by *time-fun*: *fun T-len* :: *'a list* ⇒ *nat where T-len* [] = *1* | *T-len* (*x* # *xs*) = *T-len xs* + *1* The details of this translation are described in the book referenced above. This manual is about the use of the time framework.

Command *time-fun f* retrieves the definition of *f* and defines a corresponding step-counting running-time function *T-f*. For all auxiliary functions used by *f* (excluding constructors and predefined functions (see below)), running time functions must already have been defined. Example:

fun *aux* :: *'a* ⇒ *'a* **where**
aux x = *x*

time-fun *aux*

fun *main* :: *bool* ⇒ *bool* **where**
main x = *aux x*

time-fun *main*

For functions defined by *definition*, there is a corresponding *time-definition* command. Example:

definition *gdef* :: *'a* ⇒ *'a* **where** *gdef x* = *x*

time-definition *gdef*

thm *T-gdef.simps*

Note that *T-gdef* is defined via *fun*, which means that the defining equation is not named *T-gdef-def* but *T-gdef.simps* and is a simp-rule.

The time functions for many standard functions (in particular on lists) are already defined in theory *HOL−Library.Time-Functions* and basic upper bounds are proved.

138 Termination

If the definition of a recursive function requires a manual termination proof, and you have performed that proof,

function *sum-to* :: *int* ⇒ *int* ⇒ *int* **where**
sum-to i j = (*if j* ≤ *i then 0 else i* + *sum-to* (*i*+*1*) *j*)
by *pat-completeness auto*
termination
by (*relation measure* (λ(*i*,*j*). *nat*(*j* − *i*))) *auto*

then you can use *time-fun*:

time-fun *sum-to*

In case *time-fun* takes forever or fails (this can happen with large definitions) simply use *time-function* instead, followed by an explicit termination proof:

function *sum-to2* :: *int* ⇒ *int* ⇒ *int* **where**
sum-to2 i j = (*if j* ≤ *i then 0 else i* + *sum-to2* (*i*+*1*) *j*)
by *pat-completeness auto*
termination
by (*relation measure* (λ(*i*,*j*). *nat*(*j* − *i*))) *auto*

time-function *sum-to2*
termination
by (*relation measure* (λ(*i*,*j*). *nat*(*j* − *i*))) *auto*

The termination proof should be the same as for the original function.

139 Partial Functions

Partial functions can also be 'timed'.

partial-function (*tailrec*) *positive* :: *int* ⇒ *bool* **where**
positive i = (*if i* = *1 then True else positive* (*i*−*1*))

time-partial-function *positive*

The difference is that *T-positive* has return type *nat option* because *positive* may not terminate.

Timing a function defined with *partial-function* (*option*) is trickier and we do not go into it here.

140 Higher-Order Functions

A large subclass of higher-order functions are supported, covering *map*, *filter* and other standard functions. For example,

time-fun *map*

defines a time function *T-map* :: $('a \Rightarrow nat) \Rightarrow 'a\ list \Rightarrow nat$. The first argument (called *T-f* below) is the time function for the first argument *f* of *map*. We ignore the definition of *T-map* because the output of *time-fun map* suggests that you should add these lemmas

lemma *T-map-simps* [*simp*,*code*]:
T-map T-f [] = *1*
T-map T-f (*x* # *xs*) = *T-f x* + *T-map T-f xs* + *1*
by (*simp-all add*: *T-map-def*)

which are what you would expect as defining equations. You can click on the suggestion to have it copied into your theory. Afterwards, you can work with *T-map* as if it were defined via those equations.

In general, things are a bit more complicated, which is why *T-map* is defined the way it is. Consider

fun *foldl* :: $('b \Rightarrow 'a \Rightarrow 'b) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow 'b$ **where**
foldl f a [] = *a* |
foldl f a (*x* # *xs*) = *foldl f* (*f a x*) *xs*

time-fun *foldl*

This definition is generated:

fun T-foldl :: $('b \Rightarrow 'a \Rightarrow 'b) \times ('b \Rightarrow 'a \Rightarrow nat) \Rightarrow 'b \Rightarrow 'a\ list \Rightarrow nat$ *where T-foldl* (*f*,*T-f*) *a* [] = *1* | *T-foldl* (*f*,*T-f*) *a* (*x* # *xs*) = *T-f a x* + *T-foldl f* (*f a x*) *xs* + *1*

The meaning of the pair (*f*, *T-f*) is obvious. The difference to *T-map* is that *T-foldl* needs not just *T-f* (like *T-map*) but also *f*. Function *T-map* does not need *f*: in the recursion equation *map f* (*x* # *xs*) = *f x* # *map f xs* the result of subterm *f x* is irrelevant for the computation of *T-map* because the running time of (#) is constant. This is in contrast to *foldl*, whose running time may depend on its second argument.

All higher-order functions are translated like *foldl*, but if the first element in (*f*,*T-f*) is unused, a simplified definition is derived. This is the case for *T-map*.

In case you wonder how it is ensured that *T-foldl* is always passed a corresponding pair of a function and its timing function: this is the responsibility of the time framework when translating functions that use *foldl*. Example:

definition *len2* :: *'a list* ⇒ *int* **where** *len2 xs* = *foldl* (λ*i x*. *i*+*1*) *0 xs*
time-definition *len2*

In the defining equation *T-len2 xs* = *T-foldl* (λ*i x*. *i* + *1*, λ*i x*. *0*) *0 xs* we find that *T-foldl* is passed a pair (*T-f*, *f*).

140.1 Limitations

Partial application is not supported. If possible, supply additional arguments. For example,

definition *fHO* :: *int list* ⇒ *int list* **where** ‹*fHO* = *map* (λ*i*. *2*∗*i*)›

is not acceptable (i.e. *time-definition fHO* fails), but can be replaced with

definition *fHO'* :: *int list* ⇒ *int list* **where** ‹*fHO' xs* = *map* (λ*i*. *2*∗*i*) *xs*›

time-definition *fHO'*

141 Predefined Functions

The time framework requires executable functions. However, many basic types and functions are not defined via *datatype* and *fun* but in an abstract mathematical fashion and are not executable, i.e. the time framework does not apply (or gives the 'wrong' result).

In order to model actual hardware that executes these predefined functions in constant time, there is a command for axiomatically declaring that some function takes 0 time. (This is how we model constant time, to simplify the resulting time expressions. This does not change the asymptotic running time of user-defined functions using the predefined functions because 1 is added for every user-defined function call.) Theory *HOL−Library.Time-Commands* declares a number of predefined functions as 0-time functions. This includes (+), −, (∗), /, *div*, *min*, *max*, <, ≤, ¬, ∧, ∨ and = and can be extended with the command *time-fun-0*. This feature has to be used with care:

- Many of these functions are polymorphic and reside in type classes. The constant-time assumption is justified only for those types where the hardware offers suitable support, e.g. numeric types. The argument size is implicitly bounded, too.
- The constant-time assumption for (=) is justified for recursive data types such as lists and trees as long as the comparison is of the form $t = c$ where c is a constant term, for example *xs* = [].

  Users of the time framework need to ensure that 0-time functions are used only within these bounds.

142 Locales

If we want to apply the time framework to a function g defined within a locale, we need to add additional locale parameters $T\text{-}f :: \tau \Rightarrow nat$ for every locale parameter $f :: \tau \Rightarrow \tau'$ used in the definition of g.

In the following example we do not only parameterize the locale with $T\text{-}f$ but also assume a property of $T\text{-}f$. As a result we can prove a property of $T\text{-}g$ inside the locale:

lemma *T-map-sum*: *T-map T-f xs* = *sum-list* (*map T-f xs*) + *length xs* + *1*
by(*induction xs*) (*auto*)

locale *LT* =
fixes *f* :: *'a* ⇒ *'a*
and *T-f* :: *'a* ⇒ *nat*
assumes *T-f*: *T-f x* ≤ *1*
begin

definition *g* **where** *g xs* = *map f xs*

time-definition *g*

lemma *sum-list-map-T-f-ub*: *sum-list* (*map T-f xs*) ≤ *length xs*
proof(*induction xs*)
  case (*Cons a xs*) **thus** *?case* **using** *T-f*[*of a*] **by** *auto*
qed *simp*

lemma *T-g-ub*: *T-g xs* ≤ *2* ∗ *length xs* + *1*
unfolding *T-g.simps T-map-sum* **using** *sum-list-map-T-f-ub*[*of xs*]
by *linarith*

end

Of course now you need to prove $T\text{-}f\ x \leq 1$ for every interpretation of the locale. A more flexible approach is not to constrain $T\text{-}f$ inside the locale. It may then be difficult to derive a generic time bound for $T\text{-}g$ inside the locale (in the above example it would not be difficult). If that is the case, one may also derive a bound for $T\text{-}g$ conditional on some specific bound for $T\text{-}f$. Or one can derive the bound for $T\text{-}g$ after a specific interpretation with a specific $T\text{-}f$. For a larger realistic example of the latter approach see theory *HOL−Data-Structures.Time-Locale-Example*.

143 Fine Points

Time functions for mutually recursive functions f, g, ...: *time-fun f g* ....

If you want to generate time functions not from the defining equations of a function but from lemmas proved as equations, you can provide those

lemmas explicitly. Example:

fun *f0* :: *nat* ⇒ *nat* **where**
f0 0 = *0* |
f0 (*Suc n*) = *f0 n*

lemma *f0-eq*: *f0 n* = *0*
by (*induction n*) *auto*

time-fun *f0* **equations** *f0-eq*

This may be necessary when using functions defined in locales. Example:

locale *L* =
fixes *p* :: *nat*
begin
definition *f* :: *nat* **where** *f* = *p*
end

time-definition *L.f* **equations** *L.f-def*

The *T-* prefix can be changed by modifying the *time-prefix* attribute. Example:

declare [[*time-prefix* = *t-*]]

The time framework is not verified (which is why the framework always prints out what it defines). There is no underlying formal model. This remains future work.

end

theory *Suc-Notation*
imports *Main*
begin

Nested *Suc* terms of depth $2 \leq n \leq 9$ are abbreviated with new notations Suc^n:

abbreviation (*input*) *Suc2* **where** *Suc2 n* ≡ *Suc* (*Suc n*)
abbreviation (*input*) *Suc3* **where** *Suc3 n* ≡ *Suc* (*Suc2 n*)
abbreviation (*input*) *Suc4* **where** *Suc4 n* ≡ *Suc* (*Suc3 n*)
abbreviation (*input*) *Suc5* **where** *Suc5 n* ≡ *Suc* (*Suc4 n*)
abbreviation (*input*) *Suc6* **where** *Suc6 n* ≡ *Suc* (*Suc5 n*)
abbreviation (*input*) *Suc7* **where** *Suc7 n* ≡ *Suc* (*Suc6 n*)
abbreviation (*input*) *Suc8* **where** *Suc8 n* ≡ *Suc* (*Suc7 n*)
abbreviation (*input*) *Suc9* **where** *Suc9 n* ≡ *Suc* (*Suc8 n*)

notation *Suc2* (Suc^2)
notation *Suc3* (Suc^3)

notation *Suc4* (Suc^4)
notation *Suc5* (Suc^5)
notation *Suc6* (Suc^6)
notation *Suc7* (Suc^7)
notation *Suc8* (Suc^8)
notation *Suc9* (Suc^9)

Beyond 9, the syntax Suc^n kicks in:

```
syntax
  -Suc-tower :: num-token ⇒ nat ⇒ nat  (Suc^-)

parse-translation ‹
  let
    fun mk-sucs-aux 0 t = t
      | mk-sucs-aux n t = mk-sucs-aux (n − 1) (const‹Suc› $ t)
    fun mk-sucs n = Abs(n, typ‹nat›, mk-sucs-aux n (Bound 0))

    fun Suc-tr [Free (str, -)] = mk-sucs (the (Int.fromString str))

  in [(syntax-const‹-Suc-tower›, K Suc-tr)] end
›

print-translation ‹
  let
    val digit-consts =
      [const-syntax‹Suc2›, const-syntax‹Suc3›, const-syntax‹Suc4›, const-syn-
tax‹Suc5›,
       const-syntax‹Suc6›, const-syntax‹Suc7›, const-syntax‹Suc8›, const-syn-
tax‹Suc9›]
    val num-token-T = Simple-Syntax.read-typ num-token
    val T = num-token-T --> HOLogic.natT --> HOLogic.natT
    fun mk-num-token n = Free (Int.toString n, num-token-T)
    fun dest-Suc-tower (Const (const-syntax‹Suc›, -) $ t) acc =
          dest-Suc-tower t (acc + 1)
      | dest-Suc-tower t acc = (t, acc)

    fun Suc-tr' [t] =
      let
        val (t', n) = dest-Suc-tower t 1
      in
        if n > 9 then
          Const (syntax-const‹-Suc-tower›, T) $ mk-num-token n $ t'
        else if n > 1 then
          Const (List.nth (digit-consts, n − 2), T) $ t'
        else
          raise Match
      end

  in [(const-syntax‹Suc›, K Suc-tr')]
```

end
›

end

theory *Predicate-Compile-Alternative-Defs*
imports *Main*
begin

144 Common constants

declare *HOL.if-bool-eq-disj*[*code-pred-inline*]

declare *bool-diff-def*[*code-pred-inline*]
declare *inf-bool-def*[*abs-def*, *code-pred-inline*]
declare *less-bool-def*[*abs-def*, *code-pred-inline*]
declare *le-bool-def*[*abs-def*, *code-pred-inline*]

lemma *min-bool-eq* [*code-pred-inline*]: (*min* :: *bool* => *bool* => *bool*) == (∧)
by (*rule eq-reflection*) (*auto simp add*: *fun-eq-iff min-def*)

lemma [*code-pred-inline*]:
((*A*::*bool*) ≠ (*B*::*bool*)) = ((*A* ∧ ¬ *B*) ∨ (*B* ∧ ¬ *A*))
by *fast*

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*Let*›]›

145 Pairs

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*fst*›, ***const-name*** ‹*snd*›, ***const-name*** ‹*case-prod*›]›

146 Filters

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*Abs-filter*›, ***const-name*** ‹*Rep-filter*›]›

147 Bounded quantifiers

declare *Ball-def*[*code-pred-inline*]
declare *Bex-def*[*code-pred-inline*]

148 Operations on Predicates

lemma *Diff*[*code-pred-inline*]:

$(A - B) = (\%x.\ A\ x \wedge \neg\ B\ x)$
by (*simp add*: *fun-eq-iff*)

lemma *subset-eq*[*code-pred-inline*]:
$(P :: 'a \Rightarrow bool) < (Q :: 'a \Rightarrow bool) \equiv ((\exists x.\ Q\ x \wedge (\neg\ P\ x)) \wedge (\forall x.\ P\ x \longrightarrow Q\ x))$
by (*rule eq-reflection*) (*auto simp add*: *less-fun-def le-fun-def*)

lemma *set-equality*[*code-pred-inline*]:
$A = B \longleftrightarrow (\forall x.\ A\ x \longrightarrow B\ x) \wedge (\forall x.\ B\ x \longrightarrow A\ x)$
by (*auto simp add*: *fun-eq-iff*)

149 Setup for Numerals

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*numeral*›]›
setup ‹*Predicate-Compile-Data.keep-functions* [***const-name*** ‹*numeral*›]›
setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*Char*›]›
setup ‹*Predicate-Compile-Data.keep-functions* [***const-name*** ‹*Char*›]›

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*divide*›, ***const-name*** ‹*modulo*›, ***const-name*** ‹*times*›]›

150 Arithmetic operations

150.1 Arithmetic on naturals and integers

definition *plus-eq-nat* :: *nat => nat => nat => bool*
where
plus-eq-nat x y z = $(x + y = z)$

definition *minus-eq-nat* :: *nat => nat => nat => bool*
where
minus-eq-nat x y z = $(x - y = z)$

definition *plus-eq-int* :: *int => int => int => bool*
where
plus-eq-int x y z = $(x + y = z)$

definition *minus-eq-int* :: *int => int => int => bool*
where
minus-eq-int x y z = $(x - y = z)$

definition *subtract*
where
[*code-unfold*]: *subtract x y* = $y - x$

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 iio = Fun (Input, Fun (Input, Fun (Output, Bool)))
  val ioi = Fun (Input, Fun (Output, Fun (Input, Bool)))
  val oii = Fun (Output, Fun (Input, Fun (Input, Bool)))
  val ooi = Fun (Output, Fun (Output, Fun (Input, Bool)))
  val plus-nat = Core-Data.functional-compilation const-name‹plus› iio
  val minus-nat = Core-Data.functional-compilation const-name‹minus› iio
  fun subtract-nat compfuns (- : typ) =
    let
      val T = Predicate-Compile-Aux.mk-monadT compfuns typ‹nat›
    in
      absdummy typ‹nat› (absdummy typ‹nat›
        (Const (const-name‹If›, typ‹bool› --> T --> T --> T) $
          (term‹(>) :: nat => nat => bool› $ Bound 1 $ Bound 0) $
          Predicate-Compile-Aux.mk-empty compfuns typ‹nat› $
          Predicate-Compile-Aux.mk-single compfuns
          (term‹(-) :: nat => nat => nat› $ Bound 0 $ Bound 1)))
    end
  fun enumerate-addups-nat compfuns (- : typ) =
    absdummy typ‹nat› (Predicate-Compile-Aux.mk-iterate-upto compfuns typ‹nat
* nat›
    (absdummy typ‹natural› (term‹Pair :: nat => nat => nat * nat› $
      (term‹nat-of-natural› $ Bound 0) $
       (term‹(-) :: nat => nat => nat› $ Bound 1 $ (term‹nat-of-natural› $
Bound 0))),
      term‹0 :: natural›, term‹natural-of-nat› $ Bound 0))
  fun enumerate-nats compfuns  (- : typ) =
    let
      val (single-const, -) = strip-comb (Predicate-Compile-Aux.mk-single compfuns
term‹0 :: nat›)
      val T = Predicate-Compile-Aux.mk-monadT compfuns typ‹nat›
    in
      absdummy typ‹nat› (absdummy typ‹nat›
        (Const (const-name‹If›, typ‹bool› --> T --> T --> T) $
          (term‹(=) :: nat => nat => bool› $ Bound 0 $ term‹0::nat›) $
       (Predicate-Compile-Aux.mk-iterate-upto compfuns typ‹nat› (term‹nat-of-natural›,
            term‹0::natural›, term‹natural-of-nat› $ Bound 1)) $
             (single-const $ (term‹(+) :: nat => nat => nat› $ Bound 1 $ Bound
0))))
    end
in
  Core-Data.force-modes-and-compilations const-name‹plus-eq-nat›
    [(iio, (plus-nat, false)), (oii, (subtract-nat, false)), (ioi, (subtract-nat, false)),
     (ooi, (enumerate-addups-nat, false))]
  #> Predicate-Compile-Fun.add-function-predicate-translation
       (term‹plus :: nat => nat => nat›, term‹plus-eq-nat›)
```

```
  #> Core-Data.force-modes-and-compilations const-name‹minus-eq-nat›
       [(iio, (minus-nat, false)), (oii, (enumerate-nats, false))]
  #> Predicate-Compile-Fun.add-function-predicate-translation
      (term‹minus :: nat => nat => nat›, term‹minus-eq-nat›)
  #> Core-Data.force-modes-and-functions const-name‹plus-eq-int›
    [(iio, (const-name‹plus›, false)), (ioi, (const-name‹subtract›, false)),
     (oii, (const-name‹subtract›, false))]
  #> Predicate-Compile-Fun.add-function-predicate-translation
       (term‹plus :: int => int => int›, term‹plus-eq-int›)
  #> Core-Data.force-modes-and-functions const-name‹minus-eq-int›
    [(iio, (const-name‹minus›, false)), (oii, (const-name‹plus›, false)),
     (ioi, (const-name‹minus›, false))]
  #> Predicate-Compile-Fun.add-function-predicate-translation
      (term‹minus :: int => int => int›, term‹minus-eq-int›)
end
›
```

150.2 Inductive definitions for ordering on naturals

```
inductive less-nat
where
  less-nat 0 (Suc y)
| less-nat x y ==> less-nat (Suc x) (Suc y)

lemma less-nat[code-pred-inline]:
  x < y = less-nat x y
apply (rule iffI)
apply (induct x arbitrary: y)
apply (case-tac y) apply (auto intro: less-nat.intros)
apply (case-tac y)
apply (auto intro: less-nat.intros)
apply (induct rule: less-nat.induct)
apply auto
done

inductive less-eq-nat
where
  less-eq-nat 0 y
| less-eq-nat x y ==> less-eq-nat (Suc x) (Suc y)

lemma [code-pred-inline]:
x <= y = less-eq-nat x y
apply (rule iffI)
apply (induct x arbitrary: y)
apply (auto intro: less-eq-nat.intros)
apply (case-tac y) apply (auto intro: less-eq-nat.intros)
apply (induct rule: less-eq-nat.induct)
apply auto done
```

151 Alternative list definitions

151.1 Alternative rules for *length*

definition *size-list'* :: *'a list => nat*
where *size-list' = size*

lemma *size-list'-simps*:
size-list' [] = 0
size-list' (x # xs) = Suc (size-list' xs)
by (*auto simp add*: *size-list'-def*)

declare *size-list'-simps*[*code-pred-def*]
declare *size-list'-def*[*symmetric*, *code-pred-inline*]

151.2 Alternative rules for *list-all2*

lemma *list-all2-NilI* [*code-pred-intro*]: *list-all2 P [] []*
by *auto*

lemma *list-all2-ConsI* [*code-pred-intro*]: *list-all2 P xs ys ==> P x y ==> list-all2 P (x#xs) (y#ys)*
by *auto*

code-pred [*skip-proof*] *list-all2*
proof −
case *list-all2*
from *this* **show** *thesis*
apply −
apply (*case-tac xb*)
apply (*case-tac xc*)
apply *auto*
apply (*case-tac xc*)
apply *auto*
done
qed

151.3 Alternative rules for membership in lists

lemma *in-set-member* [*code-pred-inline*]:
x ∈ set xs ⟷ List.member xs x
by *simp*

lemma *member-intros* [*code-pred-intro*]:
List.member (x#xs) x
List.member xs x ⟹ List.member (y#xs) x
by *simp-all*

code-pred *List.member*
by(*auto simp add*: *elim*: *list.set-cases*)

code-identifier constant *member-i-i*
  ⇀ (*SML*) *List.member-i-i*
 and (*OCaml*) *List.member-i-i*
 and (*Haskell*) *List.member-i-i*
 and (*Scala*) *List.member-i-i*

code-identifier constant *member-i-o*
  ⇀ (*SML*) *List.member-i-o*
 and (*OCaml*) *List.member-i-o*
 and (*Haskell*) *List.member-i-o*
 and (*Scala*) *List.member-i-o*

152 Setup for String.literal

setup ‹*Predicate-Compile-Data.ignore-consts* [***const-name*** ‹*String.Literal*›]›

153 Simplification rules for optimisation

lemma [*code-pred-simp*]: ¬ *False* == *True*
by *auto*

lemma [*code-pred-simp*]: ¬ *True* == *False*
by *auto*

lemma *less-nat-k-0* [*code-pred-simp*]: *less-nat k 0* == *False*
unfolding *less-nat*[*symmetric*] **by** *auto*

end

154 A Prototype of Quickcheck based on the Predicate Compiler

theory *Predicate-Compile-Quickcheck*
 imports *Predicate-Compile-Alternative-Defs*
begin

ML-file ‹../*Tools*/*Predicate-Compile*/*predicate-compile-quickcheck.ML*›

end

155 TFL: recursive function definitions

theory *Old-Recdef*
imports *Main*
keywords
 recdef :: *thy-defn* **and**

permissive congs hints
begin

155.1 Lemmas for TFL

lemma *tfl-wf-induct*: $\forall R.\ wf\ R \longrightarrow$
$(\forall P.\ (\forall x.\ (\forall y.\ (y,x)\in R \longrightarrow P\ y) \longrightarrow P\ x) \longrightarrow (\forall x.\ P\ x))$
apply *clarify*
apply (*rule-tac* $r = R$ **and** $P = P$ **and** $a = x$ **in** *wf-induct*, *assumption*, *blast*)
done

lemma *tfl-cut-def*: $Wfrec.cut\ f\ r\ x \equiv (\lambda y.\ if\ (y,x) \in r\ then\ f\ y\ else\ undefined)$
unfolding *cut-def* **.**

lemma *tfl-cut-apply*: $\forall f\ R.\ (x,a)\in R \longrightarrow (Wfrec.cut\ f\ R\ a)(x) = f(x)$
apply *clarify*
apply (*rule cut-apply*, *assumption*)
done

lemma *tfl-wfrec*:
$\forall M\ R\ f.\ (f=wfrec\ R\ M) \longrightarrow wf\ R \longrightarrow (\forall x.\ f\ x = M\ (Wfrec.cut\ f\ R\ x)\ x)$
apply *clarify*
apply (*erule wfrec*)
done

lemma *tfl-eq-True*: $(x = True) \longrightarrow x$
by *blast*

lemma *tfl-rev-eq-mp*: $(x = y) \longrightarrow y \longrightarrow x$
by *blast*

lemma *tfl-simp-thm*: $(x \longrightarrow y) \longrightarrow (x = x') \longrightarrow (x' \longrightarrow y)$
by *blast*

lemma *tfl-P-imp-P-iff-True*: $P \Longrightarrow P = True$
by *blast*

lemma *tfl-imp-trans*: $(A \longrightarrow B) \Longrightarrow (B \longrightarrow C) \Longrightarrow (A \longrightarrow C)$
by *blast*

lemma *tfl-disj-assoc*: $(a \vee b) \vee c \equiv a \vee (b \vee c)$
by *simp*

lemma *tfl-disjE*: $P \vee Q \Longrightarrow P \longrightarrow R \Longrightarrow Q \longrightarrow R \Longrightarrow R$
by *blast*

lemma *tfl-exE*: $\exists x.\ P\ x \Longrightarrow \forall x.\ P\ x \longrightarrow Q \Longrightarrow Q$
by *blast*

ML-file ‹*old-recdef.ML*›

155.2 Rule setup

lemmas [*recdef-simp*] =
inv-image-def
measure-def
lex-prod-def
same-fst-def
less-Suc-eq [*THEN iffD2*]

lemmas [*recdef-cong*] =
if-cong let-cong image-cong INF-cong SUP-cong bex-cong ball-cong imp-cong
map-cong filter-cong takeWhile-cong dropWhile-cong foldl-cong foldr-cong

lemmas [*recdef-wf*] =
wf-trancl
wf-less-than
wf-lex-prod
wf-inv-image
wf-measure
wf-measures
wf-pred-nat
wf-same-fst
wf-on-bot

end

156 Program extraction from proofs involving datatypes and inductive predicates

theory *Realizers*
imports *Main*
begin

ML-file ‹~~/*src/HOL/Tools/datatype-realizer.ML*›
ML-file ‹~~/*src/HOL/Tools/inductive-realizer.ML*›

end

157 Refute

theory *Refute*
imports *Main*
keywords
refute :: *diag* **and**
refute-params :: *thy-decl*
begin

ML-file ‹*refute.ML*›

refute-params
[*itself* = *1*,
minsize = *1*,
maxsize = *8*,
maxvars = *10000*,
maxtime = *60*,
satsolver = *auto*,
no-assms = *false*]

```
(* ------------------------------------------------------------------------- *)
(* REFUTE                                                                    *)
(*                                                                           *)
(* We use a SAT solver to search for a (finite) model that refutes a given   *)
(* HOL formula.                                                              *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* NOTE                                                                      *)
(*                                                                           *)
(* I strongly recommend that you install a stand-alone SAT solver if you     *)
(* want to use 'refute'.  For details see 'HOL/Tools/sat_solver.ML'.  If you *)
(* have installed (a supported version of) zChaff, simply set 'ZCHAFF_HOME'  *)
(* in 'etc/settings'.                                                        *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* USAGE                                                                     *)
(*                                                                           *)
(* See the file 'HOL/ex/Refute_Examples.thy' for examples.  The supported    *)
(* parameters are explained below.                                           *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* CURRENT LIMITATIONS                                                       *)
(*                                                                           *)
(* 'refute' currently accepts formulas of higher-order predicate logic (with *)
(* equality), including free/bound/schematic variables, lambda abstractions, *)
(* sets and set membership, "arbitrary", "The", "Eps", records and           *)
(* inductively defined sets.  Constants are unfolded automatically, and sort *)
(* axioms are added as well.  Other, user-asserted axioms however are        *)
(* ignored.  Inductive datatypes and recursive functions are supported, but  *)
(* may lead to spurious countermodels.                                       *)
(*                                                                           *)
(* The (space) complexity of the algorithm is non-elementary.                *)
(*                                                                           *)
(* Schematic type variables are not supported.                               *)
```

```
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* PARAMETERS                                                                *)
(*                                                                           *)
(* The following global parameters are currently supported (and required,    *)
(* except for "expect"):                                                     *)
(*                                                                           *)
(* Name          Type    Description                                         *)
(*                                                                           *)
(* "minsize"     int     Only search for models with size at least           *)
(*                       'minsize'.                                          *)
(* "maxsize"     int     If >0, only search for models with size at most     *)
(*                       'maxsize'.                                          *)
(* "maxvars"     int     If >0, use at most 'maxvars' boolean variables      *)
(*                       when transforming the term into a propositional     *)
(*                       formula.                                            *)
(* "maxtime"     int     If >0, terminate after at most 'maxtime' seconds.   *)
(*                       This value is ignored under some ML compilers.      *)
(* "satsolver"   string  Name of the SAT solver to be used.                  *)
(* "no_assms"    bool    If "true", assumptions in structured proofs are     *)
(*                       not considered.                                     *)
(* "expect"      string  Expected result ("genuine", "potential", "none", or *)
(*                       "unknown").                                         *)
(*                                                                           *)
(* The size of particular types can be specified in the form type=size       *)
(* (where 'type' is a string, and 'size' is an int).  Examples:              *)
(* "'a"=1                                                                    *)
(* "List.list"=2                                                             *)
(* ------------------------------------------------------------------------- *)

(* ------------------------------------------------------------------------- *)
(* FILES                                                                     *)
(*                                                                           *)
(* HOL/Tools/prop_logic.ML     Propositional logic                           *)
(* HOL/Tools/sat_solver.ML     SAT solvers                                   *)
(* HOL/Tools/refute.ML         Translation HOL -> propositional logic and    *)
(*                             Boolean assignment -> HOL model               *)
(* HOL/Refute.thy              This file: loads the ML files, basic setup,   *)
(*                             documentation                                 *)
(* HOL/SAT.thy                 Sets default parameters                       *)
(* HOL/ex/Refute_Examples.thy  Examples                                      *)
(* ------------------------------------------------------------------------- *)
```

end

References

[1] F. Haftmann and T. Nipkow. Code generation via higher-order rewrite systems. In M. Blume, N. Kobayashi, and G. Vidal, editors, *Functional and Logic Programming: 10th International Symposium: FLOPS 2010*, volume 6009, 2010.

[2] D. Leijen. Division and modulus for computer scientists. 2001.

[3] A. Lochbihler and P. Stoop. Lazy algebraic types in Isabelle/HOL. In *Isabelle Workshop 2018*, 2018.

[4] A. Podelski and A. Rybalchenko. Transition invariants. In *19th Annual IEEE Symposium on Logic in Computer Science (LICS'04)*, pages 32–41, 2004.